Pontifícia Universidade Católica
do Rio de Janeiro

Hugo Musso Gualandi

The Pallene Programming Language

Tese de Doutorado

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências – Informática.

Advisor: Prof. Roberto Ierusalimschy

Rio de Janeiro
May 2020

Pontifícia Universidade Católica
do Rio de Janeiro

Hugo Musso Gualandi

The Pallene Programming Language

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências – Informática. Approved by the Examination Committee.

Prof. Roberto Ierusalimschy
Advisor
Departamento de Informática – PUC-Rio

Prof. Waldemar Celes Filho
Pontifícia Universidade Católica do Rio de Janeiro – PUC-Rio

Prof. Francisco Figueiredo Goytacaz Sant'Anna
Universidade Estadual do Rio de Janeiro – UERJ

Prof. Philip Wadler
University of Edinburgh

Renato Fontoura de Gusmão Cerqueira
IBM Research Brazil

Rio de Janeiro, May 8th, 2020

Hugo Musso Gualandi

Graduated in 2011 with a Bachelor's Degree in Molecular Sciences from the University of São Paulo (USP). In 2015, received a Master's Degree in Informatics from the Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio).

Bibliographic data

Musso Gualandi, Hugo

The Pallene Programming Language / Hugo Musso Gualandi; advisor: Roberto Ierusalimschy. – Rio de Janeiro: PUC-Rio, Departamento de Informática, 2020.

v., 95 f: il. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Informática – Teses. 2. Linguagens de Scripting. 3. Tipagem Gradual. 4. Compiladores. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

This thesis is dedicated to my family, my friends, and my professors.
This would not have been possible without you.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

It was also financed in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico – Brasil (CNPQ) – Finance Codes 153918/2015-2 and 305001/2017-5.

Abstract

Musso Gualandi, Hugo; Ierusalimschy, Roberto (Advisor). **The Pallene Programming Language**. Rio de Janeiro, 2020. 95p. Tese de doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The simplicity and flexibility of dynamic languages makes them popular for many applications but also makes them hard to optimize. Programmers and language designers have approached this problem in many ways, including scripting, just-in-time compilers, and optional types, each one with different tradeoffs.

In a scripting architecture, a dynamic language glues together components written in a faster system language. Each language is used for the tasks for which it is well-suited. However, the cost of passing data from one language to the other can be hard to predict. Just-in-time compilers translate dynamic languages to fast machine code, based on type information collected at runtime. However, these compilers are complex and some constructions cannot be optimized as well as others. Optional type systems can retrofit a static type system into an existing dynamic language but it is not easy to design a type system that is simultaneously simple, expressive, and fast.

In this thesis we describe a new approach to the problem of the performance of dynamic languages. Combining ideas from scripting, just-in-time compilers, and optional types, we propose using the framework of optional types to create an efficient subset of an existing dynamic language. This typed subset is designed to allow ahead-of-time compilation to efficient machine code and to minimize the overhead of function calls between typed and untyped code. The subset can be seen as a companion language for the dynamic language, conceived to be used in conjunction with it and playing the role of a system language in the traditional scripting paradigm. Our example is Pallene, a typed language designed to be scripted from Lua.

Pallene's performance is comparable to that of a just-in-time compiler for Lua but its ahead-of-time compiler is simpler and more portable; this is particularly relevant in the context of Lua, which is often used as an embedded scripting language. The overhead of calling Pallene from Lua is small. We show that in programs that mix Lua and Pallene modules, rewriting a Lua module in Pallene usually improves the program's performance, and never worsens it. This is not always the case in other languages, where the cost of crossing a language boundary can be high.

Finally, we also present the basis for a formalization of Pallene's semantics. That work suggests that in addition to controlling when type errors may

occur it is also important to control when values are stored in a tagged (boxed) or in an untagged (unboxed) data representation.

Keywords

Scripting Languages; Gradual Typing; Compilers;

Resumo

Musso Gualandi, Hugo; Ierusalimschy, Roberto. **A Linguagem de Programação Pallene**. Rio de Janeiro, 2020. 95p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A simplicidade e flexibilidade das linguagens dinâmicas as torna populares em várias aplicações, porém também as torna difíceis de serem otimizadas. Programadores e criadores de linguagens de programação têm abordado este problema de várias maneiras, incluindo a arquitetura de scripting, compiladores just-in-time e sistemas de tipos opcionais. Cada uma destas abordagens traz consigo diversos prós e contras.

Na arquitetura de scripting, a linguagem dinâmica agrega componentes escritos em uma linguagem de sistema mais rápida. Cada linguagem é usada para as tarefas para as quais ela é a mais adequada. No entanto, pode ser difícil prever o custo da conversão dos dados quando estes são passados de uma linguagem para a outra. Compiladores just-in-time traduzem linguagens dinâmicas para código de máquina eficiente, baseando-se em informações de tipo coletadas durante a execução do programa. Contudo, estes compiladores são complexos e algumas construções linguísticas não podem ser otimizadas tão bem quanto outras. Sistemas de tipos opcionais permitem adicionar um sistema de tipos estático a uma linguagem dinâmica existente, porém não é fácil criar um sistema de tipos que seja simultaneamente simples, expressivo e rápido.

Nesta tese, descrevemos uma nova abordagem para o problema do desempenho das linguagens dinâmicas. Combinando ideias de scripting, compiladores just-in-time e sistemas de tipos opcionais, nós propomos usar conceitos de tipos opcionais para criar um subconjunto tipado de uma linguagem dinâmica existente. Este subconjunto tipado é projetado para ser compilável para código de máquina eficiente e para minimizar o custo das chamadas de função entre código tipado e não tipado. O subconjunto pode ser visto como uma linguagem companheira concebida para ser usada junto com a linguagem dinâmica, cumprindo o papel da linguagem de sistema na tradicional arquitetura de scripting. Nosso exemplo é Pallene, uma linguagem tipada criada para ser usada junto com Lua.

O desempenho de Pallene é comparável ao de um compilador just-in-time para Lua mas o compilador ahead-of-time de Pallene é mais simples e portátil; isto é especialmente relevante no contexto de Lua, que é comumente usada como uma linguagem de programação embutida. O custo de chamar Pallene a partir de Lua é baixo. Nós mostramos que em programas que combinam

módulos escritos em Lua com módulos escritos em Pallene, reescrever um módulo Lua em Pallene normalmente melhora o desempenho do programa, e nunca o piora. Isto nem sempre ocorre em outras linguagens, onde o custo de atravessar a fronteira entre linguagens pode ser bem alto.

Finalmente, nós também apresentamos a base para uma formalização da semântica de Pallene. Este trabalho sugere que, além de levar em conta quando os erros de tipo podem ocorrer, também é importante levar em conta quando os valores são armazenados em uma representação de dados etiquetada ou em uma não etiquetada.

Palavras-chave

Linguagens de Scripting; Tipagem Gradual; Compiladores;

Table of contents

1
Introduction

Dynamically typed languages are popular for a wide variety of applications, in part due to their ease-of-use and flexibility. However, performance is usually not their strongest point. Over the years, programmers and language designers have approached this problem from multiple directions, among them scripting, just-in-time compilers, and static types. In this thesis we seek to combine some of the positive aspects of these approaches, while avoiding their shortcomings. We propose that, starting from an existing dynamic language, it should be possible to create a typed companion language suitable for the development of performant extension modules. We exemplify this approach with Pallene, a typed companion language for Lua.

One of the classic applications of dynamic languages is in a scripting architecture. In this setting, the dynamic language glues together components written in a faster system language (Ousterhout 1998). "Rewrite it in C" is an old piece of advice that many scripting language programmers may have heard before. However, rewriting a piece of code in a completely different language is easier said than done. Furthermore, the overhead of transferring data across a language boundary can negate some or all of the speed gained from the typed language, making it harder to predict the final impact of the rewrite in the performance. In this thesis (in Chapter 3) we will show some examples where rewriting part of a Lua program in C can make it slower.

Just-in-time (JIT) compilers represent the state of the art in optimization for dynamic languages. They compile the program to efficient machine code at run time, optimizing it based on information collected while the program is running (Deutsch & Schiffman 1984). However, JIT compilers are complex to implement and maintain. Additionally, the optimization they deliver is not uniform because there often is a subset of the language that benefits more from the compilation, which programmers are implicitly encouraged to target. Programmers may need to rewrite their programs in order to achieve the best performance results (Giraldez 2017, Antonov et al. 2007). Portability is also an issue: JIT compilation may not be an option in certain mobile or embedded platforms and may also not be an option in situations where the code must reside in ROM (either due to limited RAM or by platform mandate).

As the name suggests, optional type systems allow programmers to partially add types to programs. These systems combine the static typing and dynamic typing disciplines in a single language. From static types they seek better compile-time error checking, machine-checked lightweight documentation, and run-time performance. From dynamic typing they seek flexibility and ease of use. One of the selling points of optional type systems is that they promise a smooth transition from small dynamically typed scripts to larger statically typed applications (Tobin-Hochstadt & Felleisen, Siek & Taha 2006, Bracha 2004). For this to be true these type systems attempt to accommodate common idioms of the dynamic language, which introduces additional complexities and implementation challenges. It is hard to design a type system that is at the same time simple, correct, and fast.

Both the scripting approach and optional types assume that programmers need to restrict the dynamism of their code when they seek better performance. Although in theory JIT compilers do not require this, in practice programmers also need to restrict themselves to achieve maximum performance. Realizing how these self-imposed restrictions result in the creation of vaguely defined language subsets, and how restricting dynamism seems unavoidable, we asked the question: what if we accepted the restrictions and defined a new programming language based on them? By focusing on this "well behaved" subset and making it explicit, instead of trying to optimize or type a dynamic language in its full generality, we would be able to drastically simplify the type system and the compiler.

Inspired by scripting, just-in-time compilation, and optional type systems, we have designed Pallene, a low-level counterpart to Lua. Pallene is a typed subset of Lua, built on top of the Lua runtime. It is designed to be fast, interoperable with Lua, and easy for Lua programmers to learn.

The key inspiration from JIT compilers is that they suggest that most dynamic languages contain a subset which is efficient when it is compiled to machine code. JIT compilers typically operate by collecting run-time type information during an initial stage of execution, and then using that type information to speculatively optimize the program, generating a more efficient and type-specialized version of it. Our observation is that type annotations make it possible to avoid some of the most complex aspects of JIT compilers. One of these aspects is that JIT compilers need a way to undo their optimizations when the optimized code encounters values of a type that was not predicted by the profiling phase. With type annotations this is not necessary because the type information is not speculative.

Optional type systems, in particular gradual typing, provided a theoret-

ical framework for how to add types to Lua in order to obtain Pallene. We followed the guiding principle of the gradual guarantee (Siek et al. 2015). It states that a program in the typed language (Pallene) should produce the same result as the equivalent untyped (Lua) program obtained by removing all the type annotations. The exception being that the typed program may raise run-time type errors the untyped one would not. This means that if you convert a Lua program to Pallene for better performance, the result will not silently change.

Structure of this thesis

In the first chapters of this thesis we discuss the problem of optimizing dynamic languages and we introduce Pallene, a language that we created to demonstrate the idea of typed companion languages. This introduction chapter and chapters 2 and 3, borrow heavily from one of our papers, *Pallene: A companion language for Lua* (Gualandi & Ierusalimschy 2020), which is an extended version of a previous paper with a similar title (Gualandi & Ierusalimschy 2018).

In Chapter 2 we review existing approaches used to improve the performance of dynamic languages, namely scripting, just-in-time compilers, ahead-of-time compilers, and optional type systems. We also present the Lua-AOT, an ahead-of-time compiler for Lua.

In Chapter 3 we introduce Pallene, a companion language for Lua intended for writing fast extension modules. Pallene is a typed subset of Lua, which uses run-time tag checking to enforce type safety. It shares the same runtime and garbage collector with Lua and it is compiled with an ahead-of-time compiler. We also describe the syntax and type system of Pallene. We briefly discuss its semantics, which is presented in more detail in Chapter 5. We also discuss the implementation of Pallene and how our goals for Pallene affected the implementation of its compiler. We evaluate the performance of this implementation on a set of micro benchmarks, comparing it against pure Lua (using either the reference interpreter or a just-in-time compiler), against C extension modules which use the Lua–C API (Ierusalimschy et al. 2011), and against pure C code. We also measure the overhead of run-time tag checking in Pallene.

In Chapter 4 we turn our attention to the problem of performance for gradually typed languages. We discuss how Pallene avoids the performance slowdowns that are often present when other gradually typed languages mix typed and untyped code. We show that converting parts of a Lua program into

Pallene often improves the performance and never worsens it. This part of the thesis is based on the manuscript for another of our papers, *A gradually typed subset of a scripting language can be simple and efficient*. This paper has been submitted for publication at the Journal of Functional Programming and was still undergoing peer review as of May/2020.

In Chapter 5 we present the semantics of Pallene in more detail. We formalize a subset of Pallene and a corresponding subset of Lua, proving that they are equivalent, with the exception of run-time type checking. We also show how the guiding principle of the gradual guarantee allows for flexibility in the choice of semantics for Pallene, by describing two different semantics for Pallene that both obey the gradual guarantee.

Finally, in Chapter 6 we finalize our thoughts and summarize our contributions.

2
Optimizing Scripting Languages

In this chapter we discuss four approaches that have been used to attack the problem of the performance of dynamic languages. Namely, the scripting architecture, just-in-time compilers, optional type systems, and ahead-of-time compilers. In the section about ahead-of-time compilers we also present Lua-AOT, an ahead-of-time compiler for Lua that we will use in some of our experiments.

2.1
The Scripting Architecture

The archetypal application for a scripting language is a multi-language system where the high-level scripting language is used in combination with a low-level statically-typed *system language* (Ousterhout 1998). In this architecture, the system language tends to be used for performance sensitive tasks, while the more flexible scripting language is more suited for configuration, orchestration, and situations where ease-of-development is at a premium.

Lua has been designed from the start with scripting in mind and many applications that use Lua follow this approach (Ierusalimschy et al. 2007). For instance, a computer game like Grim Fandango has a basic engine, written in C++, that performs physical simulations, graphics rendering, and other machine intensive tasks. The game designers, who were not professional programmers, wrote all the game logic in Lua (Mogilefsky 1999).

From a performance point of view, the scripting architecture is pragmatic and predictable. Each language can be used where it is more adequate and the software architect can be relatively confident that the parts written in the system language will have good performance. Moreover, scripting languages are often implemented by small interpreters, which facilitates maintainability and portability. Lua's reference interpreter, for instance, has successfully been ported to a wide variety of operating systems and hardware architectures, from large web-servers to micro-controllers.

The fundamental downside of a multi-language architecture is the conceptual mismatch between the languages. Rewriting a module to use a different language is difficult. A common piece of advice when a Lua programmer seeks

better performance is to "rewrite it in C", but this is easier said than done. In practice, programmers only follow this advice when the code is mainly about low-level operations that are easy to express in C, such as doing arithmetic and calling external libraries. Another obstacle to this suggestion is that it is hard to estimate in advance both the costs of rewriting the code and the performance benefits to be achieved by the change. Often, the gain in performance is not what one would expect. As we will show in Section 3.5, the overhead of the language interface can sometimes cancel out the inherent performance advantage of the system language.

2.2 Just-in-time Compilers

Just-in-time (JIT) compilers are the state of the art in dynamic language optimization. A JIT compiler initially executes the program without any optimization, observes its behavior, and then, based on this, generates highly specialized and optimized executable code. For example, if it observes that some code is always operating on values of type *double*, the compiler will optimistically produce a version of this code that is specialized for that type. It will also insert tests (called guards) that revert back to the generic version of the code in case one of the values is not of type *double* as expected.

JIT compilers are broadly classified as either method-based or trace-based (Gal et al. 2006), according to their main unit of compilation. In method-based JITs, the unit of compilation is the function or subroutine. In trace-based JITs, the unit of compilation is a linear trace of the program execution, which may cross over function boundaries. Trace compilation allows for a more embeddable implementation and is better at compiling across abstraction boundaries. However, it has trouble optimizing programs which contain unpredictable branch statements. For this reason, most JIT compilers now tend to use the method-based approach, with the notable exceptions of LuaJIT (Pall 2005) and RPython-based JITs (Bolz et al. 2009, PyPy 2016).

Implementing a JIT compiler is a significant undertaking. Firstly, the system must combine both an interpreter and a compiler. The interpreter, in addition to being able to run programs, must also be able to profile their execution. Furthermore, the implementation must be able to switch between generic and optimized versions of the code, while the code itself is running. This on-stack replacement can be particularly challenging to implement. Furthermore, the overall performance is heavily influenced by compilation and profiling overheads in the warm-up phase, meaning that there is a powerful incentive to specialize the implementation to the target platform and to the

language being compiled, at the cost of portability and reusability.

There is ongoing research in the area of JIT development frameworks to simplify JIT development—such as the *metatracing* of the previously mentioned RPython framework and the partial evaluation strategy of Truffle (Würthinger et al. 2013)—but unfortunately these tools currently do not support the wide variety of environments where Lua runs. For example, neither RPython nor Truffle can run on the Android or iOS operating systems. The large size of their generated binaries also precludes their use in low-memory environments. For reference, Lua can be used in devices with as low as 300KB of RAM.

From the point of view of the software developer, the most attractive aspect of JIT compilers is that they promise increased performance without needing to modify the original dynamically typed program. However, these gains are not always easy to achieve, because the effectiveness of JIT compiler optimizations can be inconsistent. Certain code patterns, known as *optimization killers*, may cause the whole section around them to be de-optimized, resulting in a dramatic performance impact. Therefore, programmers must accept that good performance depends on adapting the code to avoid these optimization killers, by following advice from the official documentation or from folk knowledge (Pall 2014, Antonov et al. 2007).

Since there may be an order of magnitude difference in performance between JIT optimized and unoptimized code, there is a powerful incentive to write programs in a style that is more amenable to optimization. This often encourages unintuitive programming idioms. For example, LuaJIT's documentation recommends caching Lua functions from other modules in a local variable (Pall 2012), as is shown in Figure 2.1. However, for C functions accessed via the foreign function interface the rule is the other way around.

Another example from LuaJIT is the function in Figure 2.2, which runs into several LuaJIT optimization killers (which the LuaJIT documentation calls "Not Yet Implemented" features). As of LuaJIT 2.1, traces that call string pattern-matching methods such as `gsub` are not compiled into machine code by the JIT. That is also the case for traces that define anonymous functions, even if the anonymous function does not access any variables of the enclosing scope.

These patterns are not unique to LuaJIT, as every JIT has its own set of quirks. For example, until 2016 the V8 JavaScript implementation could not optimize functions containing a try-catch statement (Antonov et al. 2007). Encouraging different coding styles is not the only way that JIT behavior affects the software development process, either. Programmers resort to specialized

```
-- Slower
local function mytan(x)
  return math.sin(x) / math.cos(x)
end

-- Faster
local sin, cos = math.sin, math.cos
local function mytan(x)
  return sin(x) / cos(x)
end
```

```
-- Faster (!)
local function hello()
    C.printf("Hello, world!")
end

-- Slower (!)
local printf = C.printf
local function hello()
    printf("Hello, world!")
end
```

Figure 2.1: LuaJIT encourages programmers to cache imported Lua functions in local variables. However, the recommendation for C functions called through the foreign function interface is surprisingly the other way around.

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

Figure 2.2: This function cannot be optimized by LuaJIT because it calls the `gsub` method and because it uses an anonymous callback function. These optimization killers negatively affect the performance not only of the `increment_numbers` function but also of any trace that calls it.

debugging tools to discover which optimization killer is causing the performance problems (Giraldez 2016). This may require reasoning at a low level of abstraction, involving the intermediate representation of the JIT compiler or its generated machine code.

Another aspect of JIT compilers is that before they can start optimizing, they must run the program for many iterations in interpreter mode, collecting run-time information. During this initial warm-up period the JIT will run only as fast or even slower than a non-JIT implementation. Sometimes the warm-up time can even be erratic, without a well-defined warmup phase (Barret et al. 2017).

2.3 Optional Type Systems

Static types can serve several purposes. They are useful for error detection, can act as a form of lightweight documentation, and help the compiler generate efficient code. As a result, there are many projects aiming to bring these benefits to dynamic languages, using optional type systems to combine the benefits of static and dynamic typing (Gualandi 2015).

A recurring idea to help the compiler produce more efficient code is to allow the programmer to add type annotations to the program. Compared with a more traditional scripting approach, optional typing promises a single language instead of two different ones, making it easier for the static and dynamic parts of the program to interact with each other. The pros and cons of these optional type system approaches vary from case to case, since each type system is designed for a different purpose. For example, the optional type annotations of Common LISP allow the compiler to generate extremely efficient code, but without any safeguards (Graham 1995). Meanwhile, in Typed Lua the type annotations are only used for compile-time error detection, with no effect on run-time performance.

A research area deserving special attention is Gradual Typing, which aims to provide a solid theoretical framework for designing type systems that integrate static and dynamic typing in a single language (Siek et al. 2015). However, gradual type systems still face difficulties when it comes to run-time performance. On the one hand, systems that fully check types as they cross the boundary between the static and dynamic parts of the code are often plagued with a high verification overhead cost (Takikawa et al. 2016). On the other hand, type systems that perform type erasure at run-time usually give up on the opportunity to optimize the static parts of the program.

To facilitate a smooth transition from untyped programs to typed ones,

optional type systems are typically designed to accommodate common idioms from the dynamically typed language. However, this additional flexibility may lead to a more complex type system, which is more difficult to use and, more importantly to us, to optimize. In Lua, for example, out of bound accesses result in `nil` and removing an element from a list is done by assigning `nil` to its position. Both cases require Typed Lua's type system to accept `nil` as a valid element of all Lua lists (Maidl et al. 2015). Array elements in Typed Lua are always nullable.

2.4 Ahead-of-time Compilers

In the previous chapter we discussed how just-in-time compilers have been successfully used to improve the performance of dynamically typed languages, although at a high cost in terms of implementation complexity. In this context, it is natural to ask: what about ahead-of-time compilers? They are the standard way to optimize typed languages, so why not apply the same technique to dynamic languages as well? This is something that has been tried many times. One example is Biggar's phc compiler for PHP (Biggar 2010). Biggar reported speedups between 1.5× and 2.5× compared to the reference PHP interpreter. Meanwhile, the authors of HHVM, a JIT compiler for PHP, reported larger speedups between 1.5× and 10× compared to the reference interpreter (Adams et al. 2014). While the two benchmark suites are not the same, it is worth noting that there is a large difference in the maximum speedup that was reported. This difference may be related to the inherent difficulty of optimizing a dynamic language using an ahead-of-time compiler. Biggar has said the following on this topic:

> Conventional wisdom states that a compiled program should run at least an order-of-magnitude faster than an interpreted program. In our experience, however, dynamic scripting languages do not follow this rule of thumb. Instead, a program written in a scripting language spends most of its run-time handling dynamic features, such as dynamic types and zvals. This limits the potential improvement of simply removing the interpreter loop. This is particularly important for a compiler like phc which reuses the PHP system, as many of the code paths executed will be the same, whether the program is interpreted or compiled. — (Biggar 2010)

A similar pattern can also be found in other languages and compilers. Since our research with Pallene is tied to the Lua language, we bring attention

to the ahead-of-time compilers that have been written for Lua. We know of two such compilers: lua2c (Manura 2008) and Lua Low Level (Ligneul 2016).

Lua2c converts Lua source code into an equivalent program written in C. It does not perform any kind of type inference so the compiled program also uses Lua objects and data strutures, which are manipulated from C using the Lua-C API. The performance results are not great. Due to the overhead of the Lua-C API, the compiled programs usually run slower than Lua, from 25% to 75% of the speed of the reference Lua interpreter (Manura 2008).

Lua Low Level (LLL) is a compiler that uses the LLVM framework to translate Lua bytecode into executable machine code. Frequently called functions are identified at run time and compiled into machine code by converting each bytecode instruction into a block of LLVM IR in single-static-assignment form (Ligneul 2016, Lattner 2002). Ligneul reported speedups between 1.6× and 3× compared with the reference Lua interpreter. However, this is still less than what can be achieved by a full JIT compiler. In the same set of benchmarks, Ligneul reported that LuaJIT was between 6× and 15× faster than the reference Lua interpreter.

While LLL is technically a just-in-time compiler, it behaves more like an ahead-of-time compiler because its code generation does not depend on any run-time information. However, the JIT-like behavior of compiling things at run time makes it harder to study the performance of the resulting code, since the measurements of running time will include both the code execution and the compilation. To avoid this problem, we replicated the approach of LLL using a more traditional ahead-of-time compiler, which we have named Lua-AOT.

Before we explain how Lua-AOT works in more detail, we must first step back and talk about how the Lua interpreter operates. The reference Lua interpreter is a bytecode interpreter for a register-based virtual machine, written in C, as is illustrated by the simplified interpreter in Figure 2.3 (Ierusalimschy et al. 2005). Its core is a loop that fetches the next instruction and then dispatches to the appropriate instruction handler, using a large switch-case statement.

In a register-based interpreter, values are stored in an array of virtual registers. These LuaValues are dynamically typed. They carry a type tag plus the actual value, which may be an integer, a string, a table, or one of the other possible Lua types.

The most common kind of instructions are basic data manipulation operations such as arithmetic operations, table reads and writes, function calls, etc. In a register-based interpreter the arguments for these operations are virtual registers. For example, the instruction ADD(0,1,2) adds the Lua

```
void interpret(uint32_t bytecode[], LuaValue regs[])
{
    int pc = 0;
    while(1) {
        uint32_t instr = bytecode[pc++];
        switch(opType(instr)) {
            case ADD:
                int i = opArg1(instr);
                int j = opArg2(instr);
                int k = opArg3(instr);
                regs[i] = add(regs[j], regs[k]);
                break;

            case MUL:
                int i = opArg1(instr);
                int j = opArg2(instr);
                int k = opArg3(instr);
                regs[i] = mul(regs[j], regs[k]);
                break;

            case JMP:
                pc += opArg1(instr);
                break;

            /*...*/
        }
    }
}
```

Figure 2.3: A simple bytecode interpreter for a register-based virtual machine.

values found in registers 1 and 2 and stores the result in register 0.

Another important kind of instructions are the control-flow instructions, such as conditional and unconditional jumps. Inside the interpreter these operations work by modifying the value of the virtual program counter. In the example shown in Figure 2.3 this would be the `pc` variable.

Figures 2.4 and 2.5 exemplify the compilation of a Lua program into bytecode. The Lua function in Figure 2.4 is compiled into the sequence of three bytecode instructions shown in Figure 2.5. Each instruction is represented as a 32-bit integer: the first seven bits encode the operation type and the remaining bits encode the operation arguments. The bytecode instruction set is at a lower level of abstraction than the original Lua code. Local variables are identified by virtual registers, with variables `a`, `b`, and `c` being mapped to virtual registers 0, 1, and 2, respectively. The loop is performed by the `JMP -3`, which tells the interpreter to jump back by three instructions.

```
function f(a, b, c)
    while true do
        a = b + c
        b = c * a
    end
end
```

Figure 2.4: A small Lua program.

```
0x02010020 // ADD 0 1 2
0x000200a2 // MUL 1 2 0
0x7ffffd36 // JMP -3
```

Figure 2.5: The simple Lua program, compiled to bytecode.

```
int main()
{
    LuaValue regs[256];
  L0:
    regs[0] = add(regs[1], regs[2]); // ADD 0 1 2
    regs[1] = mul(regs[2], regs[0]); // MUL 1 2 0
    goto L0;                         // JMP -3
}
```

Figure 2.6: The simple Lua program, compiled to C using the same technique as Lua-AOT.

Now we can return to Lua-AOT. The basic idea behind LLL and Lua-AOT is to compile each bytecode instruction into a block of code, as illustrated in Figure 2.6. The main difference between them is that Lua-AOT compiles everything ahead of time. The other main difference is that LLL uses LLVM as a backend while Lua-AOT generates C source code. This use of C helps remove the LLVM factor from the equation: the code generated by Lua-AOT is more similar to the C code from the original Lua interpreter. The generated C code is based on the instruction handler for the instruction, except that the arguments are specified as compile-time constants. Control flow instructions are converted into labels and gotos. As Futamura observed, this kind of compilation can be seen as partial evaluation of the bytecode interpreter combined with an unrolling of the main interpreter loop (Futamura 1999).

The motivation behind compiling programs like this is to get rid of the code for dispatching bytecode instructions. Furthermore, it also helps the optimizer by presenting more opportunities for constant folding and by

exposing the control flow graph of the program. The main downsides are that it is harder to load code at run time and that the resulting executable is larger, as each 32-bit bytecode instruction may expand into several lines of C code. In our example each instruction only expands to a single line of code but in a real interpreter the instructions can be more complicated than that. Note that this compilation strategy does not perform any type-guided optimization. The resulting programs are still dynamically-typed, using the same `add` and `mul` operations as before. Just as in the original interpreter, all values are tagged objects of type LuaValue.

2.5 Performance Evaluation of Lua-AOT

To evaluate the performance of Lua-AOT we measured the execution time of the compiled programs on a set of bencharks and compared the results with the reference Lua interpreter and the LuaJIT just-in-time compiler.

The Fannkuch, Fasta, Mandelbrot, Nbody, and Spectral Norm problems come from the popular Computer Language Benchmarks Game (Guoy 2013). However we reimplemented these benchmarks using idiomatic Lua because the code from the CLBG website uses many unnatural tricks, including eval-based metaprogramming. The Queens benchmark comes from an existing Lua benchmark suite and solves the famous N-queens puzzle. The Stream Sieve benchmark computes prime numbers using lazy streams, and is adapted from the Typed Racket benchmark suite (Takikawa et al. 2016).

We measured the running time of these benchmarks on a desktop computer with a 3.10 GHz Intel Core i5-4440 processor and 8 GB of RAM, running Fedora Linux. The interpreters and compilers 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. The results are summarized in Figures 2.7 and 2.8. Figure 2.7 lists the average time in seconds for each benchmark and implementation. Figure 2.8 further summarizes the results by presenting the average time for each benchmark divided by the average time for the reference Lua interpreter. In both tables, lower numbers are faster.

In this set of benchmarks, the Lua programs compiled with Lua-AOT were faster than the regular Lua interpreter by a factor of 1.15× to 2.5×, which is close to the speedups of up to 3× encountered by Lua Low Level. In a head-to-head comparison, the speedup reported by LLL was slightly higher. However, this comparison is obfuscated by the fact that the baseline measurement is not the same. LLL uses Lua 5.3 while Lua-AOT uses Lua 5.4.

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

Figure 2.7: A comparison of the performance of Pallene with ahead-of-time and just-in-time compilers for Lua. The notation $N \pm n$ represents an interval, where N is the average time in seconds and n is the difference between the average time and the maximum or minimum time.

Benchmark	Lua-AOT	LuaJIT
Fannkuch	0.48	0.17
Fasta	0.73	0.19
Mandelbrot	0.40	0.11
Nbody	0.63	0.06
Spectral Norm	0.51	0.08
Queens	0.59	0.10
Stream Sieve	0.86	0.18

Figure 2.8: The average time for each benchmark, divided by the average time for the reference Lua implementation. Lower numbers are faster.

In our benchmark suite, we observed that Lua 5.4 was about 20% faster than Lua 5.3 and this different baseline could reflect in a smaller speedup for the compiler targeting Lua 5.4.

We can also compare Lua-AOT's performance with that of LuaJIT, which achieved speedups of up to 15×. This suggests that the performance gain from replacing interpretation with compilation is not as impactful as the further performance gain from replacing dynamically-typed code by type-specialized code.

In addition to execution time we also measured the size of the resulting executables and of the Lua bytecode, which are listed in Figure 2.9. The AOT-compiled machine code was substantially larger than the equivalent Lua bytecode, by a factor of 25× on average (using the geometric mean). This larger size is partly due to constant overheads as is 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 produced programs that were about twice as fast

Benchmark	Lua	Lua-AOT
Empty	0.1	3.7
Fannkuch	0.9	29.5
Fasta	1.8	43.4
Mandelbrot	0.9	24.5
Nbody	1.3	37.0
Spectral Norm	1.4	26.8
Queens	1.0	34.0
Stream Sieve	1.5	30.1

Figure 2.9: Size of the benchmark programs, in kilobytes. The Lua column refers to the size of the bytecode, and the Lua-AOT column refers to the size of the executable. The "Empty" row shows the resulting sizes when compiling an empty file.

as Lua, albeit at the cost of larger code size. However, the overall speedup was still far from the speedups achievable by a JIT compiler, which can be more than 10×. One hypothesis to explain this is that Lua-AOT does not perform type-driven optimizations. The code is dynamically typed and values are represented in a tagged and boxed representation. Could we close the gap if we gave the ahead-of-time compiler more type information to work with? Would it be possible to achieve JIT-like performance without committing to the complexity of a just-in-time compiler? In the following chapters we will answer yes to both of these questions. We will introduce Pallene, a typed dialect of Lua that we designed to give more opportunities for type-driven optimization, and we will show that its performance is comparable to that of a JIT compiler.

3
The Pallene Language

In this chapter, we discuss what led us to create the Pallene programming language and how we designed it to achieve its goals of performance and interoperability with Lua. We also describe the syntax and type system of the language, as well as its semantics.

Our inspiration for Pallene came from the problem of high-performance Lua programs. Over the years we have seen an increase in the number of applications that use Lua in performance-sensitive code paths, often in combination with LuaJIT, an alternative just-in-time compiler for Lua (Pall 2005). For example, the OpenResty framework embeds LuaJIT inside the nginx web server, and enables the development of high-performance web applications written mostly in Lua (Zhang 2011). However, problems related to the high complexity and the difficulty of maintaining a just-in-time compiler such as LuaJIT have emerged. LuaJIT has diverged from the reference PUC-Lua implementation, as certain language features introduced in PUC-Lua would not get added to LuaJIT. Additionally, we observed that the complex performance landscape of the JIT compiler led programmers to adopt unusual programming idioms that eschewed language features deemed "too slow".

With the insight that programmers are willing to restrict their use of dynamic language features when performance matters, we have decided to explore a return to the traditional scripting architecture through Pallene, a system programming language that we have designed specifically to complement Lua. Since Pallene has static types, it can obtain good performance with an ahead-of-time compiler, avoiding the complexities of just-in-time compilation; and since it is designed specifically to be used with Lua, we hope Pallene will be attractive in situations where using C would be too cumbersome.

Overall, our goals for Pallene are that it should be safe, predictably efficient, seamlessly interoperable with Lua, and easy for Lua programmers to learn. Furthermore, it should have a simple and maintainable implementation that is as portable as Lua itself. Lua can run on a wide variety of architectures from mainframes to embedded systems (Ierusalimschy et al. 2018), and we would Pallene to do the same. To achieve these goals, we designed Pallene as an ahead-of-time compiled, typed subset of Lua, which can directly manipulate

```
function sum(xs: {float}): float
    local s: float = 0.0
    for i = 1, #xs do
        s = s + xs[i]
    end
    return s
end
```

Figure 3.1: A Pallene function for adding up the numbers in a Lua array. It is also a valid Lua program, except for the type annotations, which start with a colon.

Lua data structures, which shares Lua's runtime and garbage collector, and which uses run-time tag checks to enforce type safety. In this section, we describe how these design choices accomplish our goals, and how all of them play a part in enabling good performance.

Pallene is a subset of Lua Inspired by optional and gradual typing, Pallene is very close to a typed subset of Lua. For example, the Pallene program for computing the sum of an array of floating-point numbers that is shown in Figure 3.1 is also a valid Lua program, except for the type annotations. Furthermore, the behavior of Pallene programs is the same as that of Lua, except that Pallene may raise a run-time type error if it receives a value from Lua that does not have the expected type. This syntactical and semantical similarity enables the seamless interoperability between Lua and Pallene, and also makes Pallene easier for Lua programmers to learn.

Incidentally, this similarity also means that when it is desired to speed up a Lua module it should be easier to rewrite it in Pallene than it would be to rewrite it in a wholly different system language like C. That said, this transition might not be a simple matter of inserting type annotations, since Pallene's type system is designed for performance first and is not flexible enough for many common Lua idioms. This sets Pallene apart from gradual type systems such as Typed Lua (Maidl et al. 2015). Typed Lua's type system is designed to accommodate a wide variety of Lua programs but this flexibility also means that it is not able to offer better performance than plain Lua.

Pallene is typed, and uses tag checking to enforce type safety One of the major sources of performance for Pallene when compared to Lua is that Pallene programs are annotated with types and that the compiler uses this type information to generate efficient, type-specialized code. However, there are many situations where Pallene will manipulate values that originate from

Lua, in which case it must introduce a run-time type check to ensure that the value has the declared type. These run-time type checks inspect the type tags that are already present in dynamically-typed Lua values. As we will show in Section 3.5, the overhead of this tag checking is modest and is more than compensated by the substantial performance gains from type specialization.

Pallene shares the runtime and garbage collector with Lua Pallene offers first-class support for manipulating Lua objects and data structures that adhere to its type system. Not only can Pallene programs use objects such as Lua arrays, but Pallene modules also share the same runtime and garbage collector as Lua. This is in contrast to other system languages, which are only able to indirectly manipulate Lua values through the Lua–C API (Ierusalimschy et al. 2011). As we will show in Section 3.5, bypassing the traditional Lua–C API in this manner can significantly improve performance.

Pallene is compiled ahead-of-time The presence of static type information in Pallene programs allows it to be implemented with a straightforward ahead-of-time compiler, which is simpler than a JIT[1].

Guided by the type annotations, Pallene's compiler can produce efficient and type-specialized code that is much faster than the Lua equivalent. This type specialization is similar to the one that JIT compilers perform with their run-time guards, but simpler to implement. Since a failed tag check in Pallene is a run-time type error, Pallene does not need to worry about deoptimizing speculatively-optimized code. There is also no need for a separate interpreter or a warm-up profiling phase.

The Pallene compiler generates C source code and uses a conventional C compiler as a backend. This is simple to implement and is also useful for portability. The generated C code for a Pallene module is standard C code that uses the same headers and system calls that the reference Lua interpreter does. This means that Pallene code should be portable to any platforms where Lua itself can run. Ensuring this portability would have been more challenging if we had to implement our own compiler backend or if we had pursued a JIT-compilation strategy.

[1]Using lines of code as a proxy for code complexity, Pallene's compiler contains only 8 000 lines of Lua and C code while LuaJIT contains over 135 000 lines of C and assembly language.

3.1 Syntax

As Pallene is intentionally not attempting to innovate through novel type system or language semantics, the point we want to emphasize is that the main difference between Pallene and Lua—and what sets them apart as system and scripting languages, respectively—is the presence of a type system which restricts what language features and idioms can be used.

In the interest of brevity, we limit this description of Pallene to the parts that concern primitive operations, arrays, and records, which are sufficient to demonstrate the behavior of Pallene's run-time type checks, to highlight the cooperation with the Lua runtime system, and to run initial performance experiments. To simplify the presentation, we assume that variable declarations always carry a type annotation. The full version of Pallene uses a form of bidirectional type inference to infer the types of most local variables.

A Pallene module consists of a sequence of function definitions, which are to be exported to Lua. The language syntax is summarized in Figure 3.2. It is essentially the same as Lua's syntax except for the requirement that variables and functions must be typed, and that language features incompatible with the type system cannot be expressed. We will list these features when describing the type system. For brevity, we omit some primitive operators (e.g. bitwise operators), as well as syntactical niceties like optional semicolons. A complete manual for Pallene including those features can be found online on `https://github.com/pallene-lang/pallene`.

The syntax should be familiar to those who have previously programmed in a statically typed imperative language and should be immediately recognizable by Lua programmers. A prefixed '`#`' is the length operator and an infixed '`..`' is string concatenation. For clarity, we represent the empty statement as `nop`.

Curly braces are used both for arrays and records, following the notation used by Lua's table constructors. A `{}` expression means an empty array. To avoid a syntactical ambiguity between empty arrays and empty records, empty record types are not allowed.

3.2 Type System

Pallene's type system is a conventional imperative-language type system, extended with a dynamic type called `any`. Figure 3.3 describes the typing rules for expressions and Figure 3.4 does the same for statements. Figure 3.5 lists the types for primitive operators.

τ	:=		TYPES
		`nil` | `boolean` | `integer` |	*primitive types*
		`float` | `string`	
	|	`{`τ`}`	*array type*
	|	`{` $\overline{l : \tau}$ `}`	*record type*
	|	$\overline{\tau} \to \tau$	*function type*
	|	`any`	*dynamic type*
e	:=		EXPRESSIONS
	|	`nil`	*nil literal*
	|	`true` | `false`	*boolean literals*
	|	***int***	*integer literals*
	|	***flt***	*floating-point literals*
	|	***str***	*string literals*
	|	`{` $\overline{e_i}$ `}`	*array constructor*
	|	`{` $\overline{l = e}$ `}`	*record constructor*
	|	x	*variable*
	|	op_1 e	*unary operations*
	|	e op_2 e	*binary operations*
	|	e`[`e`]`	*array-read*
	|	e`.`l	*record-read*
	|	e`(`$\overline{e}$`)`	*function call*
	|	e `as` τ	*type cast*
op_1	:=	`-` | `not` | `#`	UNARY OPERATORS
op_2	:=	`+` | `-` | `*` | `/` | `..` | `and` | `or` | `==` | `<` | `>`	BINARY OPERATORS
stmt	:=		STATEMENTS
		`nop`	*empty statement*
	|	*stmt* `;` *stmt*	*sequence*
	|	`while` e `do` *stmt* `end`	*while loop*
	|	`for` x `=` e`,` e `do` *stmt* `end`	*for loop*
	|	`if` e `then` *stmt* `else` *stmt* `end`	*conditional*
	|	`local` x`:`τ `=` e`;` *stmt*	*variable declaration*
	|	x `=` e	*assignment*
	|	e`[`e`]` `=` e	*array-write*
	|	e`.`l `=` e	*record-write*
	|	e`(`$\overline{e}$`)`	*function call*
	|	`return` e	*function return*
fun	:=	`function` f`(`$\overline{x:\tau}$`)`$:\tau$ *stmt* `end`	FUNCTION DEFN.
prog	:=	$\overline{fun}$	PALLENE MODULE

Figure 3.2: Abstract syntax for Pallene. It is a subset of Lua, but with added type annotations, and the restriction that the toplevel must consist of a sequence of function definitions.

$$\Gamma \vdash \texttt{nil} : \texttt{nil} \qquad \Gamma \vdash \texttt{true} : \texttt{boolean} \qquad \Gamma \vdash \texttt{false} : \texttt{boolean}$$

$$\Gamma \vdash \mathit{int} : \texttt{integer} \qquad \Gamma \vdash \mathit{flt} : \texttt{float} \qquad \Gamma \vdash \mathit{str} : \texttt{string}$$

$$\frac{\Gamma \vdash \overline{e : \tau}}{\Gamma \vdash \texttt{\{}\ \overline{e}\ \texttt{\}} : \texttt{\{}\tau\texttt{\}}} \qquad \frac{\Gamma \vdash \overline{e : \tau}}{\Gamma \vdash \texttt{\{}\ \overline{l = e}\ \texttt{\}} : \texttt{\{}\ \overline{l : \tau}\ \texttt{\}}} \qquad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash (e\ \texttt{as}\ \tau_2) : \tau_2}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e : \tau_1 \qquad \text{optype1}(op_1, \tau_1) = \tau}{\Gamma \vdash (op_1\ e) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \text{optype2}(op_1, \tau_1, \tau_2) = \tau}{\Gamma \vdash (e_1\ op_2\ e) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{\{}\tau\texttt{\}} \qquad \Gamma \vdash e_2 : \texttt{integer}}{\Gamma \vdash e_1\texttt{[}e_2\texttt{]} : \tau} \qquad \frac{\Gamma \vdash e : \texttt{\{}\ \overline{l : \tau}\ \texttt{\}}}{\Gamma \vdash e.l_i : \tau_i}$$

$$\frac{\Gamma \vdash e : \overline{\tau} \to \tau' \qquad \Gamma \vdash \overline{e : \tau}}{\Gamma \vdash e(\overline{e}) : \tau'}$$

Figure 3.3: Pallene typing rules for expressions. $\Gamma \vdash e : \tau$ means that expression e has type τ under the environment Γ.

Variables of type `any` can hold any Lua value. Under the hood they are just tagged Lua values, which is the internal representation of values in the Lua runtime. Upcasting to `any` is always allowed, as is downcasting from `any` to another type. Downcasts are checked at run time (and we explain how in Chapter 5). This kind of variable must be present in Pallene to implement values that come from Lua, before they are tag checked. Exposing this feature to the programmer is a simple way to improve interoperation with Lua. It also adds flexibility to the type system, since this dynamic typing can be used to emulate unions and parametric polymorphism. This is similar to the role that the Object type played in Java before version 5, which introduced generics (Bracha et al. 1998).

Restrictions of Pallene's type system

The Pallene type system is not particularly innovative. The interesting discussion concerns which Lua features and idioms the type system can and cannot express, as this is what fundamentally positions Pallene as a system language instead of as a scripting one. In this section we list some examples of these more dynamic features or idioms that are present in Lua but not in

$$\Gamma \vdash \texttt{nop} \qquad \frac{\Gamma \vdash stmt_1 \qquad \Gamma \vdash stmt_2}{\Gamma \vdash stmt_1 \texttt{ ; } stmt_2} \qquad \frac{\Gamma \vdash e : \texttt{boolean} \qquad \Gamma \vdash stmt}{\Gamma \vdash \texttt{while } e \texttt{ do } stmt \texttt{ end}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{integer} \qquad \Gamma \vdash e_2 : \texttt{integer} \qquad \Gamma, x : \texttt{integer} \vdash stmt}{\Gamma \vdash \texttt{for } x = e_1, e_2 \texttt{ do } stmt \texttt{ end}}$$

$$\frac{\Gamma \vdash e : \texttt{boolean} \qquad \Gamma \vdash stmt_1 \qquad \Gamma \vdash stmt_2}{\Gamma \vdash \texttt{if } e \texttt{ then } stmt_1 \texttt{ else } stmt_2 \texttt{ end}}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \tau \vdash stmt}{\Gamma \vdash \texttt{local } x : \tau \texttt{ = } e \texttt{ ; } stmt} \qquad \frac{(x : \tau) \in \Gamma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x \texttt{ = } e}$$

$$\frac{\Gamma \vdash e_1[e_2] : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1[e_2] \texttt{ = } e_3} \qquad \frac{\Gamma \vdash e_1.l : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1.l = e_2} \qquad \frac{\Gamma \vdash f(\overline{e}) : \texttt{nil}}{\Gamma \vdash f(\overline{e})}$$

$$\frac{\Gamma \text{ , } (f : \overline{\tau} \to \tau') \text{ , } (\overline{x : \tau}) \text{ , } (\$ret : \tau') \vdash stmt}{\Gamma \quad \vdash \quad \begin{array}{l}\texttt{function } f(\overline{x:\tau}) : \tau' \\ \qquad stmt \\ \texttt{end}\end{array}} \qquad \frac{\Gamma \vdash e : \tau \qquad (\$ret : \tau) \in \Gamma}{\Gamma \vdash \texttt{return } e}$$

Figure 3.4: Pallene typing rules for statements and functions. $\Gamma \vdash stmt$ means that the statement *stmt* is well-typed under the environment Γ. The special entry $\$ret$ in the environment corresponds to the return type for the surrounding function. $\$ret$ is not a valid Pallene identifier, and only appears in these typing rules.

Pallene.

Polymorphic functions Pallene does not include type system features such as subtype polymorphism or parametric polymorphism. If this kind of flexibility is needed, the `any` type is the escape hatch that is available.

Optional parameters If a Lua function is called with less arguments than it expects, the missing arguments are set to `nil`. The conditional operator `or` can be used to assign a default value, as illustrated in Figure 3.6. In Pallene this idiom is not supported because functions have a fixed arity and the `or` operator only accepts boolean operands.

Multiple return values Lua functions may return multiple values and in some cases such as the `unpack` function the number of returned values may

optype1

op_1	operand	result
`-`	`integer`	`integer`
`-`	`float`	`float`
`not`	`boolean`	`boolean`
`#`	`{`τ`}`	`integer`
`#`	`string`	`integer`

optype2

op_2	first operand	second operand	result
`+ - * /`	`integer`	`integer`	`integer`
`+ - * /`	`integer`	`float`	`float`
`+ - * /`	`float`	`integer`	`float`
`+ - * /`	`float`	`float`	`float`
`..`	`string`	`string`	`string`
`and or`	`boolean`	`boolean`	`boolean`
`< == >`	`boolean`	`boolean`	`boolean`
`< == >`	`integer`	`integer`	`boolean`
`< == >`	`integer`	`float`	`boolean`
`< == >`	`float`	`integer`	`boolean`
`< == >`	`float`	`float`	`boolean`
`< == >`	`string`	`string`	`boolean`

Figure 3.5: Typing relations for primitive operators. Arithmetic operators work on either integers or floating point numbers. The logic operators like `and`, `or` and `not` operate on booleans. Comparison operators work on non-nil primitive types.

even vary depending on what arguments the function received. In Pallene, the number of return values has to be fixed at compile time. In the version of the language we describe here all functions must return a single value. In the development branch of the language we are currently adding support for multiple return values but still restricted to a fixed number of return values.

Dynamic arguments to primitive operators Primitive operators such as `+` or `-` may receive arguments of any type in Lua. For example, string arguments are coerced to numbers. These operators can also be overloaded using Lua's metamethod feature (Ierusalimschy et al. 2020), in which case the primitive operator will call the appropriate metamethod function. In Pallene, this kind of dynamism is not allowed and the arguments to operators such as `+` or `-` must be numbers. This is because if an operator can potentially call a Lua function, then this possibility may impair many compiler optimizations, even if the function never ends up being called.

Coroutines Lua has a powerful coroutine library which can be used for asynchronous programming or cooperative parallelism (Moura & Ierusalimschy, 2009). Adding support for coroutines would involve adding rules to the type system for yielding and awaiting from coroutines. However, the biggest challenge is an implementation one. Pallene is com-

```
function greet(name)
    name = name or "world"
    return "Hello " .. name
end

print(greet("friend"))
print(greet())
```

Figure 3.6: This Lua program cannot be directly converted to Pallene by just adding type annotations. Pallene functions must have fixed arity, and the typed version of the `or` operator would reject the string operands.

piled to C and, just like C extension modules, it is not possible to yield across a Pallene stack frame. We think that there are two possible ways to work around this. The first would be to use a technique similar to the CoCo library (Pall 2004). The other possibility would be to compile Pallene functions using continuation-passing style, which is the more portable way to allow yielding across C stack frames in Lua. Splitting functions into a set of mutually-recursive callbacks would have a performance cost, therefore this would probably have to be an optional feature that is not active for every function.

Tables One of the more substantial restriction imposed by Pallene concerns tables. The Lua table is a versatile datatype that maps arbitrary keys to arbitrary values, and is the building block for most data structures in Lua. Two particular use-cases are arrays and records. Arrays in Lua are just tables with integer keys, and records are tables with string keys that are known at compile time. Pallene supports these use cases with its array and record types. However, Pallene is not able to describe other uses of tables, such as associative arrays.

3.3 Semantics

To be compatible with Lua and to be familiar to Lua programmers, we have kept Pallene's semantics as close as possible to a strict subset of Lua. As a guiding principle, we try to follow the Gradual Guarantee of Siek et al. (Siek et al. 2015), which dictates that removing the type annotations from a Pallene program and running it as a Lua module should produce the same result, except that Pallene may raise run-time type errors that Lua would not.

When Pallene manipulates values that come from Lua, it must use run-time type checks to ensure that the types are the ones it expects. For example,

```
-- Pallene Code:
function add(x: float, y:float): float
    return x + y
end

-- Lua Code:
local big = 1 << 54
print(add(big, 1) == big)
```

Figure 3.7: An example illustrating why Pallene raises an error instead of automatically coercing integer numbers to float. If Pallene silently converted integer function parameters to float, this function would produce a different result than the equivalent Lua code that does not have any type annotations.

let's consider again the list-summing function from Figure 3.1. When Lua calls this function, it may pass it an ill-typed value that is not an array, or an array with the wrong type of element. To protect itself against this possibility, Pallene checks if `xs` is actually an array before reading from it, and checks if `xs[i]` is a floating-point number before the floating-point addition in `s = s + xs[i]`. That said, we do not specify exactly when these run-time checks should occur. The main purpose of Pallene's type system is performance and this flexibility gives the compiler the freedom to move these type checks around the code and lift them out of loops. For instance, in the previously-mentioned example the type of `xs` would be checked before the loop, and the type of `xs[i]` would be checked as needed inside the loop. This sort of delayed or "superficial" checking is the norm for non-primitive types in Pallene. Pallene is type safe in the same way that a dynamically typed language is type safe: untrapped type errors and segfaults never happen, but trapped type errors are commonplace.

So far, keeping Pallene's semantics similar to a subset of Lua's semantics is mostly a matter of not allowing the type annotations to affect behavior (other than run-time type checking, of course). For instance, Pallene does not perform automatic coercions between numeric types when assigning values or calling functions, unlike most statically typed languages. To illustrate, consider the example in Figure 3.7. In Lua, as in many dynamic languages, the addition of two integers produces an integer while the addition of two floating-point numbers produces another floating point number. If we remove the type annotations from the Pallene function `add`, and treat it as Lua code, Lua will perform an integer addition and the program will print `false`. On the other hand, were Pallene to automatically coerce the integer arguments to match the floating-point type annotations, the program would have printed `true`. Pallene would have performed a floating-point addition and, since floating-

point numbers cannot accurately represent $2^{54}+1$, it would be rounded down to 2^{54}. To avoid this inconsistency, Pallene does not perform automatic coercions between `float` and `integer`, when assigning a value. It instead complains that an integer was used where a floating-point value was expected.

Although Pallene's type annotations never introduce coercions between integers and floating point numbers, it is still possible to perform arithmetic operations on integer and floating point numbers—the result is a floating point number, just like it would be in Lua. For example, `1 + 2.0` evaluates to `3.0`. This is because the arithmetic operators are overloaded to accept any combination of integer and floating point parameters, as is indicated by the tables in Figure 3.5. This subtle distinction is specially relevant for the comparison operators. When comparing an integer and a floating point number, Lua and Pallene use accurate algorithms that always return the correct result. They do not silently convert the integer to a floating point number, because that is a lossy conversion when large integers are involved.

3.4 Implementation

Our implementation of the Pallene compiler and runtime was driven by our goals of efficiency, seamless interoperation with Lua, simplicity, and portability. The compiler itself is quite conventional. After a standard parsing step, it converts the program to a high-level intermediate form and from that it emits C code, which is then fed into a C compiler such as GCC. The final executable complies with Lua's Application Binary Interface (ABI) and can be imported by Lua programs as a standard extension module.

The choice of C as a backend allowed us to simplify various aspects of the Pallene compiler. Firstly, we could rely on a mature compiler for machine code generation, register allocation, and several optimizations. Secondly, we could take advantage of the fact that the reference Lua interpreter is also written in C. Pallene shares many of its data structures and semantics with Lua, and by also using C we could include in our generated code many C snippets and macros lifted directly from the Lua interpreter source code.

The C backend is also useful for portability. Any platform that can run the Lua interpreter should also be able to run Pallene modules, since the generated C code for a Pallene module is standard C code that is similar to the code that is present in the reference Lua interpreter.

From the point of view of performance, the most important characteristic of Pallene is that it is typed. For instance, when we write `xs[i]` in Pallene (as in our running example from Figure 3.1), the compiler knows that `xs` is an

array and `i` is an integer, and generates code accordingly. The equivalent array access in Lua (or in C code using the Lua–C API) would need to account for the possibility that `xs` or `i` could have different types or even that `xs` might actually be an array-like object with an `__index` metamethod overloading the `[]` operator.

Also important for performance is how Pallene directly manipulates private fields of Lua data structures and of the Lua runtime. Regular C extension modules for Lua must interact with Lua data structures through the Lua–C API (Ierusalimschy et al. 2011). The high-level stack-based design of this API is backwards compatible, and allows the Lua garbage collector to accurately keep track of live objects.[2] However, it necessarily introduces some overhead to the interaction between C and Lua. By bypassing the usual Lua–C API, Pallene can achieve better performance, as we show in Section 3.5. It is important to remember that Pallene can do this without sacrificing memory safety or garbage collection. The compiler is careful to generate code that respects the various invariants of the Lua runtime. For example, Pallene ensures that whenever the garbage collector is invoked, all pointers to garbage-collectable Lua objects are rooted in the Lua stack, where the garbage collector can see them. Between runs of the garbage collector, some pointers might only be stored in local C variables. These safety measures can be enforced by Pallene's compiler, but enforcing them in hand-written C code would be a nightmare.

3.5 Performance Validation of Pallene

In this section, we perform experiments to evaluate the following assumptions and hypothesis about Pallene:

1. Pallene's performance is comparable with that of a good JIT compiler, despite its much simpler implementation.

2. Rewriting parts of a Lua program in C does not always improve performance.

3. The cost of run-time tag checking in Pallene is small and is more than compensated by gains due to type specialization.

We selected six micro benchmarks for this evaluation. All are array-focused benchmarks that have been commonly used by the Lua community.

[2] Lua's stack-based API contrasts with the lower-level C APIs of languages such as Python and Ruby, which offer fewer guarantees to the programmer. In Python the programmer is tasked with keeping track of object reference counts for memory management, and in Ruby the garbage collector is conservative regarding pointers held by the C stack.

They are based on known algorithms, and implemented in a straightforward manner, without any performance tuning for a particular implementation.

Matmul: multiplies two matrices represented as arrays of arrays.

Binsearch: performs a binary search over an array of integers.

Sieve: produces a list of primes using the sieve of Eratosthenes.

Queens: solves the classic eight-queens puzzle.

Conway: a cellular automaton for Conway's Game of Life.

Centroid: computes the centroid (average) of a list of points, which are represented arrays of length 2.

For the main experiment, we ran each benchmark in five ways: Lua, LuaJIT, Lua–C API, Pallene and C. Lua means the standard Lua interpreter, version 5.4-work2. For LuaJIT we ran the same source code under the 2.1.0-beta3 version of the 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 Pallene programs that are similar to the Lua ones, except for the type annotations. To provide a familiar baseline, we also implemented the benchmarks in C, using C data structures.

We executed our experiments in a workstation with a 3.10 GHz Intel Core i5-4440 CPU, with 8 GB of RAM. The operating system was Fedora Linux, and our C compiler was GCC 8.2. For each benchmark implementation we performed forty measurements of the total wall-clock running time. To maximize the percentage of time spent running the benchmarks (as opposed to initializing them), we calibrated our benchmarks so each took at least one second to run. The Matmul, Queens, and Conway benchmarks feature algorithms with super-linear time complexity, so we simply increased the input size until it was large enough. For the Binsearch, Sieve, and Centroid benchmarks, we repeated the benchmark inside a loop to achieve the same effect. All the measurements we list are for total running time, including the ones for LuaJIT. In this set of benchmarks we observed that the LuaJIT warm-up time was negligible so we did not calculate separate warm-up and peak performance times, as is customarily done when evaluating just-in-time compilers.

Figure 3.8 shows the elapsed time for each benchmark. The bars represent the median running time for each benchmark, normalized by the Lua running time. The vertical lines represent a 90% confidence interval: the range of normalized running times encountered, excluding the 5% slowest and fastest

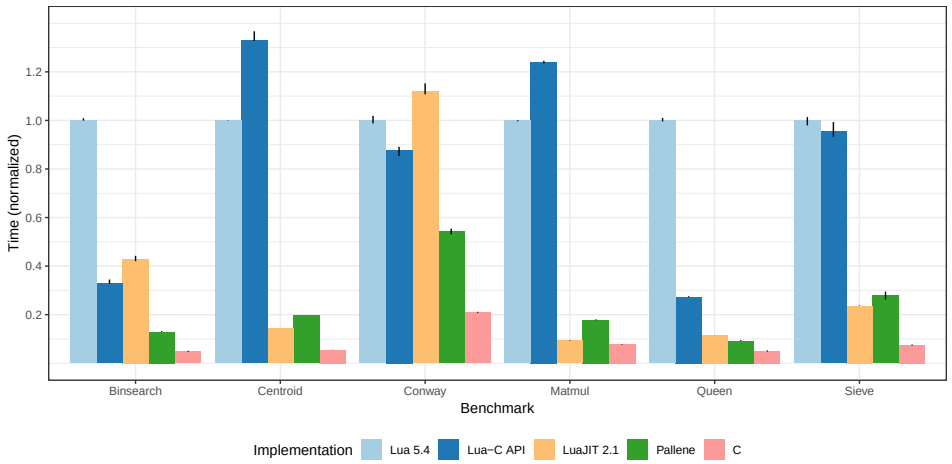


Figure 3.8: Comparison of performance between, Lua, the Lua–C API, LuaJIT, Pallene, and idiomatic C. Times are normalized by the Lua result. Lower is better. Lines represent 90% confidence intervals.

Benchmark	Lua	Lua–C API	LuaJIT	Pallene	Palene No Check	C
Binsearch	11.05	3.63	4.74	1.43	1.40	0.54
Centroid	18.62	24.73	2.64	3.67	3.44	1.03
Conway	4.44	3.90	4.97	2.41	2.37	0.93
Matmul	20.15	25.00	1.88	3.61	3.64	1.58
Queen	14.30	3.90	1.65	1.31	1.30	0.72
Sieve	11.22	10.72	2.65	3.14	3.02	0.84

Figure 3.9: Median running times for our benchmarks, in seconds. Pallene No Check is a memory-unsafe configuration of Pallene that skips run-time tag checks, which we created for this experiment.

results. Most results cluster near the median, with the exception of a small number of slower outliers. The exact times in seconds are summarized in Figure 3.9.

In these tests, Pallene running times are comparable with LuaJIT, at around half the speed of idiomatic C. The fact that it can achieve results comparable with a mature optimizing compiler suggests that Pallene may be suitable as system language for writing lower-level modules for Lua, at least in terms of performance.

The Lua–C running times were all over the place. For Matmul and Centroid, benchmarks with more operations on Lua arrays, the Lua–C API overhead outweighs the gains from using C instead of Lua. This is because the Lua–C API operations for manipulating a Lua array have to push and pop values from the Lua stack and also check if all the type tags are correct.

Let us now analyze the Pallene versus LuaJIT situation in more detail.

N	M	Time ratio	Pallene time	LuaJIT time	Pallene LLC miss	LuaJIT LLC miss
100	1024	1.02	1.46	1.44	0.25%	0.26%
200	128	1.15	1.46	1.28	15.83%	2.26%
400	16	1.86	2.74	1.48	49.59%	37.34%
800	2	1.90	2.86	1.51	48.83%	38.81%

Figure 3.10: Median running time and cache misses for the Matmul benchmark on different input sizes. N is the size of the input matrices. M is how many times we repeated the multiplication. Time ratio is Pallene time divided by LuaJIT time. Times are in seconds. Last Level Cache (LLC) load misses are a percentage of all LL-cache hits.

The only benchmark where LuaJIT is substantially faster than Pallene is Matmul. We have found that this difference is due to memory access. LuaJIT uses the *NaN-boxing* technique to pack arbitrary Lua values and their type tags inside a single IEEE-754 floating-point number (Pall 2009). In particular, this means that in LuaJIT an array of floating-point numbers consumes only 8 bytes per number, against the 16 bytes used by Lua and Pallene. This results in a higher cache miss rate and worse performance for Pallene. To confirm that this is the case, we also ran this benchmark under smaller values of N, measuring running times and cache-miss rates using the Linux `perf` tool. The results are summarized in Figure 3.10. For smaller values of N, the matrices always fit inside the machine cache and Pallene and LuaJIT are just as fast. For larger N, LuaJIT's more compact data representation leads to less cache misses and better performance. This NaN-boxing effect also explains the difference between 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. For example, NaN tagging is incompatible with unboxed 64-bit integers, which is one of reasons why this technique is not used in the standard Lua interpreter 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, forking the hot path. This is not an issue for Pallene and other ahead-of-time compilers.

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

The Conway benchmark results are surprising because LuaJIT performed worse than standard Lua. This unusual result is due to the new generational garbage collector introduced in Lua 5.4. It turns out that the bulk of the time

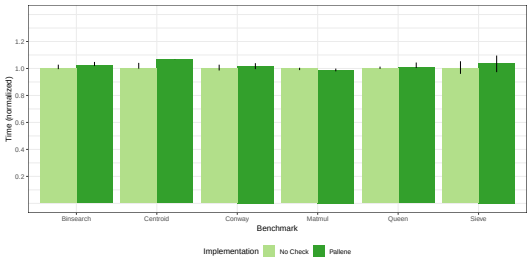


Figure 3.11: Pallene run-time tag checking overhead. Times are normalized by the No Check result. Lower is better. Lines represent 90% confidence intervals.

in this benchmark is spent doing string manipulation and garbage collection and LuaJIT still relies on the incremental garbage collector that was used until version 5.3. In a separate experiment, we disabled the generational mode of the 5.4 garbage collector and its performance slowed down to become comparable to LuaJIT's. This difference highlights that JIT compilation is not a panacea and that sometimes other aspects can dominate the execution time.

The cost of run-time tag checks

We also investigated the effect of run-time tag checking in our implementation of Pallene arrays. Since Pallene arrays can be directly modified by untyped Lua code, Pallene must perform a run-time tag check whenever it reads from an array slot that might have been written by untyped code. In this experiment we ran our benchmark suite under both the default Pallene compiler and No Check, an unsafe version of Pallene that skips all the run-time tag checks. Figure 3.11 shows the results of this experiment, normalized against the No Check implementation. The exact times are also summarized in Figure 3.9.

In all our benchmarks, the tag checks next to the array reads introduce only little overhead, ranging from 0% to 10%, which we consider surprisingly low. Many of our benchmarks perform run-time checks inside their innermost loops, and this kind of run-time tag checking often introduces greater overheads in other languages. For example, Reticulated Python also checks types when accessing lists and object fields, but in their case the average slowdown was by a factor of 2.5x (Vitousek et al. 2017).

To understand why our tag checks were cheaper than expected, we again resorted to Linux's `perf` tool. The extreme case of the Matmul benchmark is particularly illuminating. It had a 0% tag-checking overhead despite the presence of multiple run-time tag checks for the array reads in its innermost loop. In Figure 3.12 we show the Instructions per Cycle statistics for the

Implementation	Time	Cycles	Instructions	Intructions per Cycle
Pallene	2.86	9.36×10^9	16.68×10^9	1.78
Pallene No Check	2.86	9.34×10^9	7.46×10^9	0.80

Figure 3.12: Run-time tag checks correspond to more than half of the x86-64 instructions executed in the Matmul benchmark. However, they do not impact the total running time, due to the Intel Core i5's instruction-level parallelism. Times in seconds, $N = 800$, $M = 2$.

Matmul that we collected with `perf`. Although the tag checks accounted for more than half of the machine-level instructions that were executed, the final running time is still the same with or without them. This happens because the Matmul benchmark is memory bound, as indicated by the low (≤ 1.0) Instructions per Cycle statistic for Pallene No Check. This gives plenty of room for the pipelining in the Intel Core CPU to execute the extra tag-checking instructions in parallel, effectively for free.

In our other benchmarks we observed a similar pipelining benefit, albeit less pronounced than in Matmul. Naturally, the run-time overhead of tag checking may be higher for CPU-bound benchmarks, or if the benchmark is run on other kinds of CPUs. Nevertheless, these results suggest that run-time tag checking is not incompatible with efficient typed code, specially if the tag checks are ultimately compiled down to machine-level jumps (as opposed to source-level conditional statements, or bytecode-level jumps).

3.6 Related Work

In this section we review how Pallene compares to related work in type systems and optimization for dynamic languages.

Typed Lua (Maidl et al. 2015) is a gradual type system for Lua. Its types can be used for documentation and compile-time error detection. Typed Lua aims to be flexible enough to type a wide variety of Lua programs and has a rich set of table types to model the many different uses of Lua tables. Similarly to TypeScript (Bierman et al. 2014), Typed Lua is intentionally unsound, meaning its types cannot be used for program optimization. They are erased before the program runs, and have no effect at run time.

Common Lisp is another language that has used optional type annotations to provide better performance. As said by Paul Graham in his ANSI Common Lisp Book (Graham 1995), "Lisp is really two languages: a language for writing fast programs and a language for writing programs fast". Pallene

and Common Lisp differ in how their sub-languages are connected. In Common Lisp, they live together under the Lisp umbrella, while in Lua and Pallene they are segregated, under the assumption that modules can be written in different languages. Common Lisp also has an option to unsafely disable run-time tag checks.

RPython (Ancona 2007, Bolz et al. 2009, PyPy 2016) is a restricted subset of Python which can be compiled to efficient machine code, either through ahead-of-time compilation (to C source code) or just-in-time compilation (through the PyJitPl JIT). Unlike Pallene, RPython does not have explicit type annotations, and is not defined by a syntax-directed type system. The types are inferred by a whole-program flow-based analysis that uses a graph of live Python objects as starting point. Finally, one very important difference is that Pallene modules can be independently-compiled, and called from Lua code running in an unmodified version of the reference Lua interpreter. RPython-based executables, on the other hand, must be built as a single compilation unit. Compiled RPython modules are also incompatible with CPython, the reference Python interpreter.

Cython (Behnel et al. 2010) is an extension of Python with C datatypes. It is well suited for interfacing with C libraries and for numerical computation, but its type system cannot describe Python types. Cython is unable to provide large speedups for programs that spend most of their time operating on Python data structures.

Terra (DeVito 2014) is a low-level system language that is embedded in and meta-programmed by Lua. Similarly to Pallene, Terra is also focused on performance and has a syntax that is very similar to Lua, to make the combination of the two languages more pleasant to use. However, while Pallene uses Lua as a scripting language, Terra uses it as stage-programming tool. The Terra system uses Lua to generate Terra programs aimed at high-performance numerical computation and, once produced, these programs run independently of Lua. Terra uses manual memory management and features low-level C-like datatypes. There are no language features to aid in interacting with a scripting language at run time.

4
Pallene and the Performance of Gradually Typed Languages

Our initial performance results with Pallene were encouraging. In the comparison with LuaJIT, Pallene's performance was in the same ballpark as the mature just-in-time compiler, despite Pallene having a vastly simpler implementation. This was possible due to Pallene's choice of restricting the language to a typed subset of Lua. In the comparison with C, pure C code was the fastest, but C code that made heavy use of Lua data structures was slower than Pallene and sometimes even slower than Lua. This suggested that the effort to minimize the overhead of the interface between Lua and Pallene was worthwhile.

After the encouraging initial results we turned our attention to studying what aspects of Pallene's design contributed to its good performance, and whether those lessons could be applied to other languages as well. Of particular interest to us was the problem of performance for gradually typed languages. In the context of gradual typing, pathological performance when mixing the typed and untyped code is the norm. In some gradually typed languages, programs that combine typed modules with untyped modules can be much slower than entirely untyped programs (Takikawa et al. 2016, Vitousek et al. 2014).

The conventional wisdom states that this bad performance is due to the overhead of run-time type checking, caused by the run-time checks the typed language uses to safely interoperate with the untyped language. However, just-in-time compilers also make extensive use of run-time type checks, without running into those performance problems. In fact, type guards are a fundamental part of their optimization strategy.

In most gradually typed languages, the compiler uses the untyped language as a compilation target, unlike Pallene's compiler, which ultimately compiles down to machine code. We hypothesized that this might be the reason why run-time type checks are costly in many gradually typed languages. Reusing the existing implementation for the untyped language is a natural direction to take when implementing a gradually typed language based on an existing untyped language. However, a type check that takes the form of an if-statement or function-call in the untyped language might be slower than a type check that is compiled down to a single CPU branch instruction.

In this chapter we promote Pallene as an example of a performant gradually typed language, which avoids some performance pitfalls encountered by other gradually typed languages. We show that adding types (converting Lua to Pallene) often improves the overall performance, and never harms it. We also discuss how aspects of Pallene's design may be applied to other languages. This material is based on a paper authored with Roberto Ierusalimschy which has been submitted to the Journal of Functional Programming, and as of April/2020 is still undergoing peer review.

4.1 Performance Challenges for Gradually Typed Languages

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., 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). Richards 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 is that if the gradually typed language is designed from scratch, instead of being based on an existing dynamic language, then it might be possible to obtain better performance. One such example would be the Nom language from Muehlboeck and Tate (Muehlboeck & Tate 2017), which uses

nominal typing pervasively to reduce the overhead of run-time type checking.

In this thesis and in previous papers 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 be dynamically-loaded by Lua, in a similar manner to C extension modules. In this chapter 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 chapter is organized as follows: Section 4.2 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 4.3 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 4.4 contains experiments evaluating the overhead of migrating programs from Lua to Pallene. These experiments indicate that migrating from Lua to Pallene usually leads to better performance, for all possible combinations of Lua and Pallene modules. Furthermore, Section 4.5 discusses how Pallene is intended to always be used side-by-side with Lua, and how this allowed us to simplify its implementation.

4.2 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 the 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.

Just-in-time compilation is capable of large speedups. This can be seen in the experiments we made with LuaJIT in previous chapters. In Section 2.5, LuaJIT achieved a speedup between 5× and 20× compared to Lua and in Section 3.5 we observed speedups of up to 10×.

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 are implemented in a similar manner to JIT deoptimization checks. We have confirmed this hypothesis with Pallene, which we will talk more about in Section 4.3. We believe that this may also be applicable to other gradually typed languages 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 do not 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. One common approach for 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 do not 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. The

PyPy implementation for Python was only made compatible with Python 3 after many years and a significant effort (PyPy 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.

4.3 Adding Types

In Chapter 3 we introduced Pallene, a typed variant of the Lua programming language designed with performance in mind. 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 language 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.

To recall what a Pallene program looks like, consider the example program in Figure 4.1. It 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 stressed that in Pallene, the absence of a type annotation does not mean that the variable has the dynamic type `any`.

When it comes to semantics, Pallene programs behave the same as Lua, except that they may raise run-time type errors that Lua would have not. In general terms, its type-checking strategy 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). We provide more details of this in Chapter 5. The most important aspect is that type checks in Pallene always take the form of type-tag checks, which run in constant time. 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 should not. Pallene checks the type of values that come from Lua arrays or Lua

```
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
```

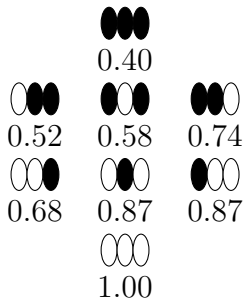
Figure 4.1: An example Pallene program, implementing the inner loop of the Nbody benchmark. The syntax is the same as the equivalent Lua program, but with added type annotations.

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

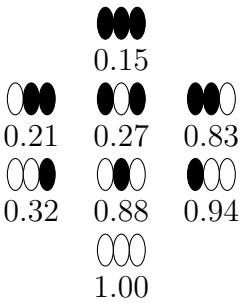
Pallene was designed for performance, and one fundamental part of that is that its compiler generates efficient machine code. To simplify the implementation, and for portability, Pallene generates C source code instead of directly generating assembly language. This is similar to the approach that we used for Lua-AOT in Section 2.4, 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 CPU 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.

4.4 Program Migration Performance Experiments for Pallene

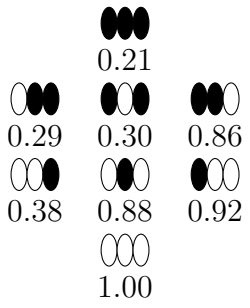
We evaluated the performance of combining Lua and Pallene using the standard performance lattice analysis for gradual typing systems (Takikawa et al. 2016). For this experiment we adapted four benchmarks from the set of benchmarks we used in Section 2.5: Spectral Norm, Nbody, Queens, and Stream Sieve. These were the benchmarks from that set which contained several functions and which therefore could be split into more than one module. 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 4.2 and Figure 4.3. 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 number shown under each configuration is the running time, normalized by the time of the pure Lua configuration. Lower numbers are faster.



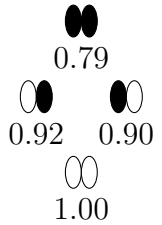
4.2(a): Spectral Norm



4.2(b): Queens



4.2(c): Nbody

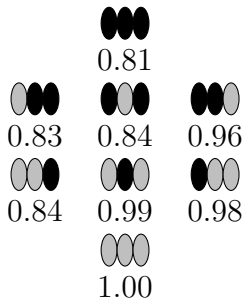


4.2(d): Stream Sieve

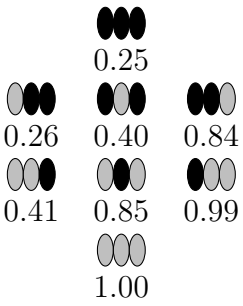
Figure 4.2: Performance lattice for combinations of Pallene and interpreted Lua modules. Dark ovals represent (typed) Pallene modules and white ovals represent (untyped) Lua modules, running on the Lua 5.4 interpreter. The running times for each benchmark are normalized by the running time of the pure 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.

A concrete example might help explain the lattices. Consider the case of the Nbody benchmark. In preparation for this experiment, we refactored the code from Figure 4.1 so that each of the three functions was put in a separate module. In the performance lattice shown in Figure 4.2(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 the `update_position` and `update_speeds` functions are implemented in Lua.

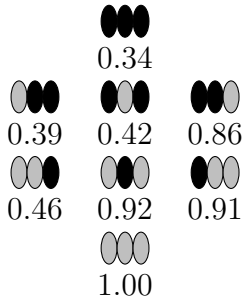
In the lattice of configurations we can move from one configuration to another by changing one module at a time from Lua to Pallene or vice-versa. For example, we can go from ○○○ to ●○○ by changing the implementation of the first module from Lua to Pallene. In this set of benchmarks, for every possible 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 with interpreted Lua,



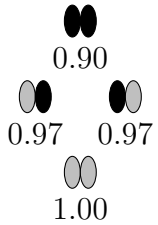
4.3(a): Spectral Norm



4.3(b): Queens



4.3(c): Nbody



4.3(d): Stream Sieve

Figure 4.3: Performance lattice for combinations of Pallene and Lua, this time with the Lua modules being compiled by an ahead-of-time compiler (Lua-AOT). Black ovals are Pallene, and gray ovals are Lua-AOT. As in Figure 4.2, 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.

the more typed configurations were always faster.

To get a better idea of how much of the performance gain from Pallene is due to typing or due to ahead-of-time compilation we repeated the experiments but using the Lua-AOT compiler from Section 2.4 as the baseline. This time, 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.

Astute readers might notice that the speedups for the full-Pallene configurations in the lattices are not as large as the ones we saw in Section 3.5. This can be seen more clearly in Figures 4.4 and 4.5, which show Pallene's performance on the single-module versions of these benchmarks. The normalized running time for the full-Pallene configuration in the multi-module benchmarks (0.40, 0.15, 0.21, and 0.79) are all slower than the corresponding running times

Benchmark	Lua	Lua-AOT	LuaJIT	Pallene
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

Figure 4.4: Running times for the single-module versions of the benchmarks used for the lattice experiments. The notation $N \pm n$ represents an interval, where N is the average time in seconds and n is the difference between the average time and the maximum or minimum time.

Benchmark	Lua-AOT	LuaJIT	Pallene
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

Figure 4.5: Normalized times for the single-module versions of the benchmarks. Each number is the average running time for that benchmark divided by the running time for the reference Lua interpreter. Lower numbers are faster.

in the single-module versions of the benchmarks (0.08, 0.08, 0.14, and 0.65). We believe that the main explanation for this performance difference is due to function calling conventions. Pallene uses a more efficient calling convention when directly calling other Pallene functions within the same module and a slower calling convention when calling functions from a different module. Cross-module function calls use the same calling convention used when Pallene calls a Lua function, which involves passing the arguments via the Lua stack. Since the benchmarks in the lattice experiment feature many cross-module function calls, precisely to test the communication and type checking overhead between Lua and Pallene, this difference in calling convention can be significant. We are currently investigating how to use the more efficient Pallene calling convention for cross-module function calls as well.

4.5 Focusing the Design of Pallene

Our main goal for Pallene was to design a language with good performance, aimed at the Lua ecosystem. Additionally, 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

```
-- 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
```

Figure 4.6: 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.

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. However, the only primitive operation that can be used on them is downcasting to a different type. For instance, they must be converted to a numeric type (integer or float) before being used in arithmetic, as exemplified in Figure 4.6. Similarly, they must be cast 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 whether a language is gradually typed: 1) it should encompass both fully-typed programs and fully-untyped programs; 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.

Pallene fits the second and third criteria, as we discussed 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 union of Lua plus Pallene is trivially a superset of Lua.

4.6 Summary

In this chapter, we showed 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×. 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 of 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.

5
A Pair of Semantics for Pallene

In the previous chapters we mentioned that Pallene uses run-time tag checks to verify the types of values originating from Lua and that the behavior of a Pallene program should be similar to the behavior of the Lua program that is obtained by removing all the type annotations. However, we did not formally specify what exactly this meant.

In this chapter we will address these questions by formalizing a restricted subset of Pallene. We present λ-Dyn, λ-Pallene, and Pallene IR, three small lambda-calculi that model the very simplified subsets of Lua and Pallene that we study in this chapter.

To simplify the proofs and to focus on the matter of run-time tag checking and data representation, we restrict our formalization to a purely-functional subset of Pallene, in the same style as other lightweight calculi such as Featherweight Java (Igarashi 2001). Our simplified version of Pallene only has a handful of types: functions, integers, arrays, and nil. Functions are fundamental to any lambda-calculus. The integer and nil types demonstrate how Pallene deals with base types. The array type allows us to discuss how Pallene implements arrays and records using Lua tables.

Language	Typing	Evaluation
λ-Dyn		$\longrightarrow_{dyn}$
λ-Pallene	$\Gamma \vdash_{pln} e : \tau$	
Palllene IR (PIR)	$\Gamma \vdash_{pir} e : \tau$	$\longrightarrow_{pir}$

Figure 5.1: The languages described in this chapter

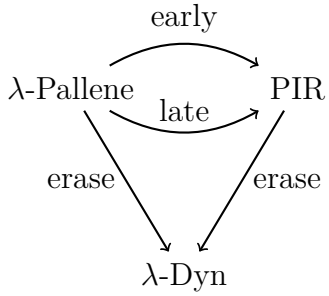


Figure 5.2: Transformations between the languages

Figure 5.1 summarizes the three languages that we discuss in this chapter. λ-Dyn is an untyped calculus with a reduction relation $\longrightarrow_{dyn}$. λ-Pallene is a typed language with a typing judgement $\vdash_{pln}$ but without a reduction relation. Following previous work on gradually typed languages, we specify the semantics of λ-Pallene by first converting it to a lower-level intermediate representation with explicit run-time type checks, called Pallene Intermediate Representation (PIR). It features a type system, with typing judgment $\vdash_{pir}$, as well as a reduction relation $\longrightarrow_{pir}$. To reduce visual noise, we may omit the subscripts when the language we are referring to is unambiguous.

Figure 5.2 shows the ways to convert between these three languages. The early-checking and late-checking transformations lead to two possible semantics for λ-Pallene. The late-checking translation, discussed in Section 5.4, produces a PIR program where all values are stored in a tagged and boxed representation. These tags are checked at the last possible moment before the value is used, similarly to a dynamically-typed language. Conversely, in Section 5.5 we will show the early-checking translation, which checks the type tags sooner, allowing more values to be represented in an unboxed and untagged manner.

The translation to Pallene IR serves two main purposes. The first is to model exactly when type checking happens at run time, by making the run-time type-checking explicit. The second important purpose is to model how values are represented at run time. Pallene is designed to interoperate with Lua and the design of PIR reflects this. For example, Pallene arrays are implemented as Lua tables and Pallene functions conform to the Lua function calling convention. We will discuss this in further detail in Section 5.3.

5.1 λ-Dyn

The first part of our formalization of Pallene is λ-Dyn, a dynamically typed language representing the very simplified subset of Lua that corresponds to the subset of Pallene that we are formalizing. Figure 5.3 shows the syntax of this language. It is a normal lambda calculus extended with integers, arrays, and a unit type (nil). In the various diagrams, integer literals such as 0 and 1 are represented by n. The notation for arrays and function calls are borrowed from Lua: array constructors use curly braces, array indexing uses square brackets, and function calls are written with the arguments in parentheses. Overlines represent repetition; the array constructor expression may contain zero or more sub-expressions.

e	:=		EXPRESSIONS
	|	`nil`	*nil literal*
	|	n	*integer literal* (0, 1, 2, ...)
	|	e_1 `+` e_2	*integer arithmetic*
	|	`{` $\overline{e}$ `}`	*array constructor*
	|	e_1`[`e_2`]`	*array read*
	|	x	*variable*
	|	e_1`(`e_2`)`	*function call*
	|	$\lambda x.\ e$	*untyped lambda*

Figure 5.3: λ-Dyn syntax

λ-Dyn evaluation

v	:=		VALUES
	|	`nil`	*nil value*
	|	n	*integer value*
	|	`{` $\overline{v}$ `}`	*array value*
	|	$\lambda x.\ e$	*lambda value*
C	:=		EVAL. CONTEXTS
	|	$\square$	*hole*
	|	C `+` e | v `+` C	*arithmetic*
	|	`{` $\overline{v}$ C $\overline{e}$ `}`	*array constructor*
	|	C`[`e`]` | v`[`C`]`	*array indexing*
	|	C`(`e`)` | v`(`C`)`	*function call*

Reduction relation $\boxed{e \longrightarrow_{dyn} e'}$

$$\text{(R-ADD)}\quad C\Big[n_a \texttt{ + } n_b\Big] \longrightarrow C\Big[n_c\Big], \qquad n_c = n_a + n_b$$

$$\text{(R-INDEX)}\quad C\Big[\texttt{\{ } \overline{v} \texttt{ \}[}n\texttt{]}\Big] \longrightarrow C\Big[v_n\Big], \qquad n \in \text{keys}(\{\overline{v}\})$$

$$\text{(R-MISSINGKEY)}\quad C\Big[\texttt{\{ } \overline{v} \texttt{ \}[}n\texttt{]}\Big] \longrightarrow C\Big[\texttt{nil}\Big], \qquad n \notin \text{keys}(\{\overline{v}\})$$

$$\text{(R-APP)}\quad C\Big[\texttt{(}\lambda x.e\texttt{)(}v\texttt{)}\Big] \longrightarrow C\Big[e[x \leftarrow v]\Big]$$

Figure 5.4: A small-step semantics for λ-Dyn

In Figure 5.4 we provide a small-step semantics for λ-Dyn. Values in λ-Dyn are integers, arrays of values, lambdas, and the special `nil` value. Evaluation happens one step at a time, as determined by the $\longrightarrow_{dyn}$ relation. As usual, the evaluation contexts C are expressions with a "hole" dictating where reduction may occur. The evaluation proceeds in a call-by-value manner, from left to right. In the definition of evaluation, beware that the monospaced brackets in $C\texttt{[}e\texttt{]}$ and $v\texttt{[}C\texttt{]}$ are for array indexing, not for filling in the hole in the context.

The $\longrightarrow_{dyn}$ relation has four cases, corresponding to arithmetic, array indexing and function calls. As in Lua, an out-of-bounds array read returns the special value `nil`. There are no rules that generate a run-time error. The evaluation gets stuck in erroneous programs such as attempting to call a table as if it were a function. One of the reasons that we chose to leave this behavior unspecified is that in Lua these operations are not always an error. For example, if a Lua table implements the function call metamethod then Lua code may call that table as if it were a function. This flexible and extensible behavior is a hallmark of many dynamic programming languages.

Another choice that we made is that λ-Dyn is purely-functional, without operations to mutate arrays or local variables. This is solely to simplify the semantics and make the proofs easier to understand. In Section 5.7 we sketch a way to extend our results to also work in the presence of mutable state.

5.2 λ-Pallene

The typed counterpart of λ-Dyn is λ-Pallene, whose syntax we show in Figure 5.5. It is a typed language representing a subset of Pallene. Syntactically, the main difference compared to λ-Dyn is that lambdas are typed with mandatory type annotations and that there is a type cast operator, e `as` τ.

Figure 5.6 summarizes the type system of λ-Pallene. It is a simply-typed lambda calculus extended with a dynamic type (`any`) and a type cast operator inspired by gradual typing (Siek & Taha 2006). All Pallene values can be converted to a value of type `any` using "`as any`".

Type casts are restricted by a symmetric type-consistency relation $\sim$ that specifies which casts "make sense". Two types are consistent if the only

e	$:=$		EXPRESSIONS
	$\mid$	`nil`	*nil literal*
	$\mid$	n	*integer literal* (0, 1, 2, ...)
	$\mid$	e_1 `+` e_2	*integer arithmetic*
	$\mid$	`{` $\overline{e}$ `}`	*array constructor*
	$\mid$	e_1`[`e_2`]`	*array read*
	$\mid$	x	*variable*
	$\mid$	e_1`(`e_2`)`	*function call*
	$\mid$	$\lambda x{:}\tau.\ e$	*typed lambda*
	$\mid$	e `as` τ	*type cast*
τ	$:=$		TYPES
	$\mid$	`any`	*dynamic type*
	$\mid$	`nil`	*integer type*
	$\mid$	`int`	*integer type*
	$\mid$	`{`τ`}`	*array type*
	$\mid$	$\tau \to \tau$	*function type*

Figure 5.5: λ-Pallene syntax

λ-Pallene types $\boxed{\Gamma \vdash_{pln} e : \tau}$

$$\text{(T-INT)}\ \frac{}{\Gamma \vdash n : \texttt{int}} \qquad \text{(T-ADD)}\ \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}$$

$$\text{(T-CONS)}\ \frac{\Gamma \vdash \overline{e} : \tau}{\Gamma \vdash \texttt{\{ } \overline{e} \texttt{ \}} : \texttt{\{}\tau\texttt{\}}} \qquad \text{(T-INDEX)}\ \frac{\Gamma \vdash e_1 : \texttt{\{}\tau\texttt{\}} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1\texttt{[}e_2\texttt{]} : \tau}$$

$$\text{(T-VAR)}\ \frac{x{:}\tau \in \Gamma}{\Gamma \vdash \texttt{x} : \tau} \qquad \text{(T-LAMBDA)}\ \frac{\Gamma, x{:}\tau \vdash e : \tau'}{\Gamma \vdash (\lambda x{:}\tau.e) : \tau \to \tau'}$$

$$\text{(T-CAST)}\ \frac{\Gamma \vdash e : \tau \qquad \tau \sim \tau'}{\Gamma \vdash e \texttt{ as } \tau' : \tau'} \qquad \text{(T-APP)}\ \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\texttt{(}e_2\texttt{)} : \tau'}$$

$$\text{(T-NIL)}\ \frac{}{\Gamma \vdash \texttt{nil} : \texttt{nil}}$$

Type consistency $\boxed{\tau_1 \sim \tau_2}$

$$\text{(C-REFL)}\ \frac{}{\tau \sim \tau} \qquad \text{(C-ANY-1)}\ \frac{}{\tau \sim \texttt{any}} \qquad \text{(C-ANY-2)}\ \frac{}{\texttt{any} \sim \tau}$$

$$\text{(C-ARRAY)}\ \frac{\tau \sim \tau'}{\texttt{\{}\tau\texttt{\}} \sim \texttt{\{}\tau'\texttt{\}}} \qquad \text{(C-FUN)}\ \frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}$$

Figure 5.6: λ-Pallene type system

differences between them are parts that are `any` in one of the sides. For example:

$$\texttt{int} \sim \texttt{int}$$
$$\texttt{\{any\}} \sim \texttt{\{int\}}$$
$$\texttt{any} \sim \texttt{\{int\}}$$
$$\texttt{int}\to\texttt{any} \sim \texttt{any}\to\texttt{int}$$

$$\texttt{int} \nsim \texttt{nil}$$
$$\texttt{int}\to\texttt{int} \nsim \texttt{\{any\}}$$

Dynamically typed values of type `any` can be passed and returned from functions and can be stored in arrays of type `{any}`. However, they cannot be directly used in primitive operations such as arithmetic. It is necessary to explicitly downcast to the appropriate type first, using the `as` operator, as in the following example:

$$(\lambda x\texttt{:any. 17 + (x as integer))(10 as any)}$$

The integer `10` can be converted to the dynamic type with `as any` but then it must be converted back to integer with `as integer` before it can be used by the addition operator.

λ-Pallene is a subset of the full Pallene language described in Chapter 3, except for the addition of lambda functions. Nested function expressions are not currently implemented by the Pallene compiler but are a feature that we would like to support in the future.

5.3 Pallene Intermediate Representation (PIR)

To describe the semantics of λ-Pallene we will use Pallene IR (PIR), a lower level calculus that has explicit run-time tag checks. We will first describe this low level calculus and then we will describe how to translate λ-Pallene to it.

Two of the main objectives of PIR are to model how values are represented and to describe how types are checked at run time. To achieve this, PIR introduces an explicit distinction between boxed and unboxed values, as well as `box` and `unbox` operations to convert between them. The complete syntax is shown in Figure 5.7.

PIR's type system models the way that objects are represented in the Lua runtime, particularly the way that boxed values carry a type tag. It also

e	:=		EXPRESSIONS
	|	`nil`	*nil literal*
	|	n	*integer literal* (0, 1, 2, ...)
	|	e_1 `+` e_2	*integer arithmetic*
	|	`{` $\overline{e}$ `}`	*array constructor*
	|	e_1`[`e_2`]`	*array read*
	|	x	*variable*
	|	e_1`(`e_2`)`	*function call*
	|	$\lambda x{:}\tau.\ e$	*typed lambda*
	|	$\texttt{box}_G\ e$	*upcast*
	|	$\texttt{unbox}_G\ e$	*downcast*
τ	:=		TYPES
	|	`any`	*dynamic type*
	|	`nil`	*nil type*
	|	`int`	*integer type*
	|	`{any}`	*array of any*
	|	$\tau \to \tau$	*function type*
G	:=		GROUND TYPES
	|	`nil`	*nil type*
	|	`int`	*integer type*
	|	`{any}`	*array of any*
	|	`any` $\to$ `any`	*function of any*

Figure 5.7: Pallene IR Syntax

reflects how arrays and functions are implemented as Lua arrays and functions, which is a design choice we made to improve the interoperability with Lua.

In PIR, the type tags correspond to the subset of types identified as *ground types*. The T-BOX rule in the type system mandates that only values with a ground type may be stored inside a box. Values of other types can be used by PIR but may not be directly exposed to Lua. In the case of functions, this means that only functions of type `any`→`any` can be put inside a box. These are the functions that receive and return boxed values, as the default Lua calling convention does. Nevertheless, although there is special treatment for the `any`→`any` type, regular function types of the form $\tau \to \tau'$ are still allowed. This lets Pallene functions call other Pallene functions using a more efficient calling convention, without the need to convert all the arguments to `any`. Furthermore, as in other lambda calculi, the lambdas are also used to represent local variables. Here, the types model how local variables are stored at run time, which can matter for performance. For example, local variables with unboxed types may potentially be stored in machine registers.

The PIR type system also treats arrays differently than λ-Pallene does.

PIR types $\boxed{\Gamma \vdash_{pir} e : \tau}$

$$\text{(T-INT)}\ \frac{}{\Gamma \vdash n : \texttt{int}} \qquad \text{(T-ADD)}\ \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}$$

$$\text{(T-CONS)}\ \frac{\Gamma \vdash \overline{e : \texttt{any}}}{\Gamma \vdash \texttt{\{ } \overline{e} \texttt{ \}} : \texttt{\{any\}}} \qquad \text{(T-INDEX)}\ \frac{\Gamma \vdash e_1 : \texttt{\{any\}} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{[} e_2 \texttt{]} : \texttt{any}}$$

$$\text{(T-BOX)}\ \frac{\Gamma \vdash e : G}{\Gamma \vdash \texttt{box}_G\ e : \texttt{any}} \qquad \text{(T-UNBOX)}\ \frac{\Gamma \vdash e : \texttt{any}}{\Gamma \vdash \texttt{unbox}_G\ e : G}$$

$$\text{(T-VAR)}\ \frac{x{:}\tau \in \Gamma}{\Gamma \vdash \texttt{x} : \tau} \qquad \text{(T-LAMBDA)}\ \frac{\Gamma, x{:}\tau \vdash e : \tau'}{\Gamma \vdash (\lambda x{:}\tau.e) : \tau \to \tau'}$$

$$\text{(T-NIL)}\ \frac{}{\Gamma \vdash \texttt{nil} : \texttt{nil}} \qquad \text{(T-APP)}\ \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \texttt{(} e_2 \texttt{)} : \tau'}$$

Figure 5.8: Pallene IR type system

The only kind of array that the Lua runtime and garbage collector knows how to work with are arrays of boxed objects, which are represented in PIR by the type `{any}`. Other array types in λ-Pallene are all converted to arrays of `any` in PIR, as we will show in more detail when we discuss the early-checking and the late-checking translations from λ-Pallene to PIR.

We now turn our attention to the evaluation of PIR programs. In Figure 5.9 we provide a small-step semantics for PIR, which is similar to the small-step semantics of λ-Dyn. The differences are that each evaluation step may now produce a run-time error, written `panic`; that the `nil` in the R-MISSINGKEY case is now boxed; and that there are additional reduction rules to describe the semantics of the `box` and `unbox` operations. The `unbox` operation returns the value that is inside a box, after checking if its type tag is the expected one. Note that the right hand side of the R-ERROR rule does not have an evaluation context C. If a tag-check error happens inside a sub-expression, then the evaluation for the entire expression is interrupted, yielding `panic`.

Soundness for PIR

In the context of PIR, soundness informally means that well-typed PIR programs don't get stuck. Their evaluation may "go wrong" if a run-time tag check fails but it does not get stuck—getting stuck would indicate a memory safety bug in λ-Pallene. More formally, we can specify this using the usual

PIR evaluation

v	$:=$		VALUES
	$\mid$	`nil`	*nil value*
	$\mid$	n	*integer value*
	$\mid$	`{` $\overline{v}$ `}`	*array value*
	$\mid$	$\lambda x{:}\tau.\ e$	*lambda value*
	$\mid$	$\texttt{box}_G\ v$	*boxed value*
C	$:=$		EVAL. CONTEXTS
	$\mid$	$\square$	*hole*
	$\mid$	C `+` $e \mid v$ `+` C	*arithmetic*
	$\mid$	`{` $\overline{v}$ C $\overline{e}$ `}`	*array constructor*
	$\mid$	C`[`e`]` $\mid v$`[`C`]`	*array indexing*
	$\mid$	C`(`e`)` $\mid v$`(`C`)`	*function call*
	$\mid$	$\texttt{box}_G\ C$	*upcast*
	$\mid$	$\texttt{unbox}_G\ C$	*downcast*
r	$:=$		RESULTS
	$\mid$	e	*successful step*
	$\mid$	`panic`	*run-time error*

Reduction relation $\boxed{e \longrightarrow_{pir} r}$

$$\text{(R-ADD)}\quad C\Big[n_a \texttt{ + } n_b\Big] \longrightarrow C\Big[n_c\Big], \qquad n_c = n_a + n_b$$

$$\text{(R-INDEX)}\quad C\Big[\texttt{\{ } \overline{v} \texttt{ \}[}n\texttt{]}\Big] \longrightarrow C\Big[v_n\Big], \qquad n \in \text{keys}(\{\overline{v}\})$$

$$\text{(R-MISSINGKEY)}\quad C\Big[\texttt{\{ } \overline{v} \texttt{ \}[}n\texttt{]}\Big] \longrightarrow C\Big[\texttt{box}_{\texttt{nil}}\ \texttt{nil}\Big], \qquad n \notin \text{keys}(\{\overline{v}\})$$

$$\text{(R-APP)}\quad C\Big[\texttt{(}\lambda x{:}\tau.e\texttt{)(}v\texttt{)}\Big] \longrightarrow C\Big[e[x \leftarrow v]\Big]$$

$$\text{(R-UNBOX)}\quad C\Big[\texttt{unbox}_G\ \texttt{(box}_G\ v\texttt{)}\Big] \longrightarrow C\Big[v\Big]$$

$$\text{(R-ERROR)}\quad C\Big[\texttt{unbox}_{G1}\ \texttt{(box}_{G2}\ v\texttt{)}\Big] \longrightarrow \texttt{panic}, \qquad G1 \neq G2$$

Figure 5.9: A small-step semantics for Pallene IR

technique of progress and preservation lemmas. Our proofs are based on the proofs from Pierce's Types and Programming Languages book, particularly the proofs of progress and preservation for the simply typed lambda calculus (Pierce 2002).

Lemma 5.1 (Canonical forms lemma for PIR)**.**

- *If v is a value of type* `nil` *then it is an integer literal n.*
- *If v is a value of type* `int` *then it is an integer literal n.*
- *If v is a value of type* `{any}` *then it is an array literal* `{` $\overline{v}$ `}`
- *If v is a value of type $\tau \to \tau'$ then it is has the form $\lambda x{:}\tau.e$*
- *If v is a value of type* `any` *then it has the form* $\texttt{box}_G\ v'$

Proof. By a straightforward induction on the definition of value. The novel case is the one for the `any` type. The only typing rules that can produce such type are rules T-BOX, T-INDEX, and T-APP. However, only in the T-BOX case the expression can be a value. Therefore, if a value has type `any` then it must be a `box` expression. □

Lemma 5.2 (Progress for PIR)**.** *If $\Gamma \vdash_{pir} e : \tau$ then either e is a value or it can take a reduction step ($e \longrightarrow_{pir} e'$ or $e \longrightarrow_{pir}$* `panic`*).*

Proof. By induction on the typing derivations. The novel cases are the ones for T-INDEX, T-BOX and T-UNBOX.

- T-INDEX: If either e_1 or e_2 is not a value, then the induction hypothesis says that $e_1\texttt{[}e_2\texttt{]}$ can take a step. If both are values, the canonical forms lemma states that they must be an array value and an integer value, respectively. Therefore, reduction can take place with either the R-INDEX rule or the R-MISSINGKEY rule.
- T-BOX: If e is not a value, then the induction hypothesis says that $\texttt{box}_G\ e$ can take a reduction step. Alternatively, if e is a value then $\texttt{box}_G\ e$ is also a value.
- T-UNBOX: If e is not a value, again the induction hypothesis says that it is possible to make an evaluation step. Otherwise, if e is a value then by the canonical forms lemma it must be a boxed value. The complete expression will have the form $\texttt{unbox}_{G1}\texttt{(box}_{G2}\ v\texttt{)}$. It can take a reduction step either by rule R-UNBOX or by rule R-ERROR, depending on whether the type tags match or not.

□

Lemma 5.3 (Preservation for PIR)**.** *If* $\Gamma \vdash_{pir} e : \tau$ *and* $e \longrightarrow_{pir} e'$ *then* $\Gamma \vdash_{pir} e' : \tau$

Proof. Again, the proof is by induction on the typing derivation and the interesting cases are the ones for T-INDEX, T-BOX and T-UNBOX.

- T-INDEX: If $e_1\,[e_2]$ is well-typed then it must have type `any`, e_1 must have type `{any}`, and e_2 must have type `integer`. If e_1 or e_2 are not values then the reduction step happens inside them and preservation follows from the induction hypothesis. Otherwise, the reduction is either a R-INDEX or R-MISSINGKEY step, depending on whether the integer index is in-bounds or out of bounds. In both cases the result has type `any`, as required for preservation.
- T-BOX: If $\texttt{box}_G\, e$ is well-typed we can say that it has type `any` and that e has type G. Furthermore, the only possible reduction that can happen is $\texttt{box}_G\, e \longrightarrow_{pir} \texttt{box}_G\, e'$. By the induction hypothesis we can deduce that e' also has type G and therefore that $\texttt{box}_G\, e'$ also has type `any`.
- T-UNBOX: As before, if $\texttt{unbox}_G\, e \longrightarrow_{pir} \texttt{unbox}_G\, e'$ then we apply the induction hypothesis. The other possibilities are the R-ERROR and R-UNBOX rules. The R-ERROR case can be ruled out by the $e \longrightarrow_{pir} e'$ hypothesis because `panic` is not an expression. In the R-UNBOX case we have $\texttt{unbox}_G\texttt{(box}_G\, v\texttt{)} \longrightarrow_{pir} v$. The hypothesis that the left side is well-typed tells us that both $\texttt{unbox}_G\texttt{(box}_G\, v\texttt{)}$ and v have type G and therefore the type of the expression is preserved by the reduction step.

□

5.4 The Late-checking Translation

One way to convert λ-Pallene to PIR is to use a tagged and boxed representation for all values and only check the tags when the value is about to be used. This late-checking translation is shown in Figure 5.10. All type annotations are ignored and the variables are converted to type `any`. The run-time types are checked at the last possible moment: the PIR program will check if the arguments of an arithmetic operation are really numbers, if the function being called is really a function, and so on. This is similar to λ-Dyn, except for one crucial difference. While in a dynamic language such as λ-Dyn the type-checking happens "inside" primitive operations such as the addition operator, in PIR this type checking happens in the `unbox` operation.

$$
\begin{array}{lcl}
\lfloor\ \tau\ \rfloor & = & \texttt{any} \\
 & & \\
\lfloor\ \texttt{nil}\ \rfloor & = & \texttt{box}_{\texttt{nil}}\ \texttt{nil} \\
\lfloor\ n\ \rfloor & = & \texttt{box}_{\texttt{int}}\ n \\
\lfloor\ e_1\ \texttt{+}\ e_2\ \rfloor & = & \texttt{box}_{\texttt{int}}\ \texttt{((unbox}_{\texttt{int}}\ \lfloor e_1 \rfloor\texttt{) + (unbox}_{\texttt{int}}\ \lfloor e_2 \rfloor\texttt{))} \\
\lfloor\ \texttt{\{}\ \overline{e}\ \texttt{\}}\ \rfloor & = & \texttt{box}_{\texttt{\{any\}}}\ \texttt{\{}\ \overline{\lfloor e \rfloor}\ \texttt{\}} \\
\lfloor\ e_1\texttt{[}e_2\texttt{]}\ \rfloor & = & \texttt{(unbox}_{\texttt{\{any\}}}\ \lfloor e_1 \rfloor\texttt{) [unbox}_{\texttt{int}}\ \lfloor e_2 \rfloor\ \texttt{]} \\
\lfloor\ x\ \rfloor & = & x \\
\lfloor\ \lambda x : \tau . e\ \rfloor & = & \texttt{box}_{\texttt{any}\to\texttt{any}}\ \lambda x\texttt{:any.}\lfloor e \rfloor \\
\lfloor\ e_1\texttt{(}e_2\texttt{)}\ \rfloor & = & \texttt{(unbox}_{\texttt{any}\to\texttt{any}}\ \lfloor e_1 \rfloor\texttt{) (}\ \lfloor e_2 \rfloor\ \texttt{)} \\
\lfloor\ e\ \texttt{as}\ \tau\ \rfloor & = & \lfloor e \rfloor
\end{array}
$$

Figure 5.10: The late-checking conversion from λ-Pallene to PIR

We can show that under this late-checking translation, well-typed Pallene programs are converted into well-typed PIR programs. Together with the previous theorem showing that PIR is sound, this implies that Pallene programs compiled under the late-checking translation don't get stuck at run time.

Theorem 5.4 (The late-checking translation produces well-typed programs)**.** *If* $\Gamma \vdash_{pln} e : \tau$ *then* $\lfloor\Gamma\rfloor_{late} \vdash_{pir} \lfloor e \rfloor_{late} : \texttt{any}$

Proof. By induction on the structure of the $\vdash_{pln}$ typing judgment. We will briefly present the reasoning behind each case.

- Integer literals and nils are immediately put into a box, which has type `any`. Arithmetic operations unbox their arguments (with a tag check) and then imediately put the resulting number back in a box.
- The arguments to the array literals are translated into expressions of type `any`, which is precisely what the PIR array expects.
- The array read operation checks if the array is actually an array and if the index is actually an integer. The result has type `any` because the PIR array always has type `{any}`.
- Variable names are preserved but all the types are changed to `any`.
- All the lambdas are changed to have type `any→any`. This reflects how in the late-checking translation all variables and expressions are mapped to something of type `any`. The lambda itself is immediately boxed with $\texttt{box}_{\texttt{any}\to\texttt{any}}$, meaning that the overall result of translating a λ-Pallene lambda is a PIR expression of type `any`.
- A function call checks whether the function is actually a function. Since all functions are translated as a function of type `any→any` the result of the function call will have type `any`. There are no type checks involving

the arguments or return values—those type checks will only happen when those values are used in a primitive operation.

– Type casts in λ-Pallene are completely erased. The result is the translation of the inner expression, which will have type `any`.

□

Corollary 5.5 (Soundness for late-checking λ-Pallene)**.** *If* $\Gamma \vdash_{pln} e : \tau$ *then the PIR term* $\lfloor e \rfloor_{late}$ *does not get stuck.*

Proof. Follows from the theorem the soundness for PIR (Lemmas 5.2 and 5.3) and Theorem 5.4, which says that the late-checking translation produces well-typed PIR terms. □

5.5 The Early-checking Translation

One characteristic of the late-checking translation is that all the values are represented in a tagged and boxed manner, which is bad for performance. The early-checking translation goes in the opposite direction: it prefers to store values in an unboxed manner when possible, anticipating the run-time tag checks if necessary. The translation has three parts, which are shown in Figure 5.11.

The first part of the translation maps λ-Pallene types to PIR types. Note that all λ-Pallene types map to either a ground type or to `any`.

– $\lfloor \texttt{any} \rfloor_{early}$ is `any`

– $\lfloor \texttt{int} \rfloor_{early}$ and $\lfloor \texttt{nil} \rfloor_{early}$ are ground types.

– All array types are mapped to `{any}`, the only array type in PIR.

– All function types become `any` → `any`, the only boxable function type.

The second part of the translation is the ⇐ helper function for converting betweeen two PIR types. It inserts a `box` or `unbox` where appropriate. The two types must be either `any` or a ground type.

– A cast from `any` to `any` is compiled to a no-op.

– An upcast from a ground type to `any` becomes a `box` operation.

– A downcast from `any` to a ground type becomes an `unbox` operation.

– A cast from a ground type to itself is also compiled to a no-op.

– The conversion from a ground type to a different ground type is undefined. However, the $\tau_1 \sim \tau_2$ restriction ensures that this never happens.

$$
\begin{array}{lcl}
\lfloor\ \texttt{any}\ \rfloor & = & \texttt{any} \\
\lfloor\ \texttt{nil}\ \rfloor & = & \texttt{nil} \\
\lfloor\ \texttt{int}\ \rfloor & = & \texttt{int} \\
\lfloor\ \{\tau\}\ \rfloor & = & \{\texttt{any}\} \\
\lfloor\ \tau_1 \to \tau_2\ \rfloor & = & \texttt{any} \to \texttt{any} \\
\\
(\texttt{any} \Leftarrow \texttt{any})\ e & = & e \\
(\texttt{any} \Leftarrow G)\ e & = & \texttt{box}_G\ e \\
(G \Leftarrow \texttt{any})\ e & = & \texttt{unbox}_G\ e \\
(G \Leftarrow G)\ e & = & e \\
\\
\lfloor\ \texttt{nil}\ \rfloor & = & \texttt{nil} \\
\lfloor\ n\ \rfloor & = & n \\
\lfloor\ e_1\ \texttt{+}\ e_2\ \rfloor & = & \underline{\lfloor e_1 \rfloor\ \texttt{+}\ \lfloor e_2 \rfloor} \\
\lfloor\ \{\ \overline{e}\ \}\ :\ \{\tau\}\ \rfloor & = & \{\ \overline{(\texttt{any} \Leftarrow \lfloor\tau\rfloor)\lfloor e \rfloor}\ \} \\
\lfloor\ e_1[e_2]\ :\tau\ \rfloor & = & (\lfloor\tau\rfloor \Leftarrow \texttt{any})\ (\lfloor e_1 \rfloor[\lfloor e_2 \rfloor]) \\
\lfloor\ x\ \rfloor & = & x \\
\lfloor\ \lambda x : \tau . e\ \rfloor & = & \lambda x' \texttt{:any}.\ (\texttt{any} \Leftarrow \lfloor\tau'\rfloor)\ ((\lambda x : \lfloor\tau\rfloor . \lfloor e \rfloor)\ ((\lfloor\tau\rfloor \Leftarrow \texttt{any}) x')) \\
\lfloor\ (e_1 : \tau \to \tau')(e_2)\ \rfloor & = & (\lfloor\tau'\rfloor \Leftarrow \texttt{any})\ (\lfloor e_1 \rfloor((\texttt{any} \Leftarrow \lfloor\tau\rfloor)\lfloor e_2 \rfloor)) \\
\lfloor\ (e : \tau)\ \texttt{as}\ \tau'\ \rfloor & = & (\lfloor\tau'\rfloor \Leftarrow \lfloor\tau\rfloor)\ \lfloor e \rfloor
\end{array}
$$

Figure 5.11: The early-checking conversion from λ-Pallene to PIR. We use the notation $e : \tau$ to annotate the types of certain expressions. These type annotations are not part of the syntax of λ-Pallene.

The third and final part of the translation are the translation rules for expressions. Similarly to the late-checking translation, all arrays are implemented as `{any}` because we want to implement Pallene arrays as Lua tables and because we want to allow Pallene to use arrays created by Lua. However, this time the array values are unboxed as soon as they are read from the array instead of only when they are about to be used. Lambdas are converted to typed lambdas, wrapped inside a lambda of type `any` $\to$ `any`. The inner typed lambda allows the local variables to be represented unboxed while the outer lambda allows the function itself to be boxed (and potentially exposed to Lua). Finally, using a similar logic as before we can prove that the early-checking translation converts well-typed λ-Pallene programs to well-typed PIR programs.

Theorem 5.6 (The early-checking translation produces well-typed programs)**.** *If* $\Gamma \vdash_{pln} e : \tau$ *then* $\lfloor\Gamma\rfloor_{early} \vdash_{pir} \lfloor e \rfloor_{early} : \lfloor\tau\rfloor_{early}$

Proof. By induction on the structure of the $\vdash_{pln}$ typing judgement.

- Integer literals are converted into unboxed integers.
- The arguments to the integer arithmetic operation are unboxed integers and it returns an unboxed integer.

- The array constructor casts the values to `any` before they are stored in the array. Values that are not already boxed will be boxed by the $(\texttt{any} \Leftarrow \lfloor\tau\rfloor)$ cast. Remember that in PIR the only array type is `{any}`.
- Conversely, values that are read from an array are immediately cast from `any` to the appropriate type. If the λ-Pallene type of the array is `{any}` then this will be an $(\texttt{any} \Leftarrow \texttt{any})$ cast which is a no-op. Otherwise, this cast will unbox the value immediately after it is read from the array.
- λ-Pallene variables of type τ are compiled into variables of type $\lfloor\tau\rfloor$. If the λ-Pallene type of the variable is `any` then it will be represented in a boxed form, otherwise it will be represented in an unboxed form.
- A λ-Pallene lambda of type $\tau \to \tau'$ is converted into a PIR lambda of type $\lfloor\tau\rfloor \to \lfloor\tau'\rfloor$, wrapped inside another lambda of type $\texttt{any} \to \texttt{any}$. The wrapper lambda casts the input argument from `any` to $\lfloor\tau\rfloor$, passes it to the inner lambda and then converts the result from $\lfloor\tau'\rfloor$ to `any`.
- Function calls box the input argument and unbox the return value. This is similar to how we box a value when storing it into an array and unbox when reading from it.
- Type casts in λ-Pallene are converted to either a `box`, an `unbox`, or a no-op, depending on the types. The $\tau \sim \tau'$ requirement in λ-Pallene's T-CAST rule ensures that $(\lfloor\tau'\rfloor \Leftarrow \lfloor\tau\rfloor)$ is well-defined.

□

Corollary 5.7 (Soundness for early-checking λ-Pallene)**.** *If* $\Gamma \vdash_{pln} e : \tau$ *then the PIR term* $\lfloor e \rfloor_{early}$ *does not get stuck.*

Proof. Follows from the theorem the soundness for PIR (Lemmas 5.2 and 5.3) and Theorem 5.6, which says that the early-checking translation produces well-typed PIR terms. □

One of the most interesting features of the early-checking translation is that each lambda is translated to typed lambda wrapped inside an "untyped" lambda of type `any`→`any`. The primary reason for this is to allow interoperability with Lua. Only functions of type `any`→`any` may be exposed to Lua. Recall that only functions with this type can be boxed and passed as a parameter to a Lua function or stored inside a Lua table.

However, this design choice of always adding an `any`→`any` wrapper was also made with performance in mind. One problem that can happen in gradually-typed languages is that type casts between function types can introduce layers of wrappers which are a big source of performance overhead,

$$\begin{aligned}\lfloor (e : \tau_1 \to \tau_1')\ \texttt{as}\ \tau_2 \to \tau_2' \rfloor_{early} &= (\lfloor \tau_2 \to \tau_2' \rfloor \Leftarrow \lfloor \tau_1 \to \tau_1' \rfloor)\, \lfloor e \rfloor \\ &= (\texttt{any}{\to}\texttt{any} \Leftarrow \texttt{any}{\to}\texttt{any})\, \lfloor e \rfloor \\ &= \lfloor e \rfloor\end{aligned}$$

Figure 5.12: In the early-checking translation, casts between function types become a no-op. They are not a source of performance overhead.

$$\begin{aligned}\lfloor (\lambda x{:}\tau.e)(v) \rfloor_{early} &= (\lfloor \tau' \rfloor \Leftarrow \texttt{any})(\lfloor \lambda x{:}\tau.e \rfloor((\texttt{any} \Leftarrow \lfloor \tau \rfloor)\lfloor v \rfloor)) \\ &= (\lfloor \tau' \rfloor \Leftarrow \texttt{any})(\lambda x'{:}\texttt{any}. \\ &\qquad (\texttt{any} \Leftarrow \lfloor \tau' \rfloor)((\lambda x{:}\lfloor \tau \rfloor.\lfloor e \rfloor)((\lfloor \tau \rfloor \Leftarrow \texttt{any})x')) \\ &\qquad)((\texttt{any} \Leftarrow \lfloor \tau \rfloor)\lfloor v \rfloor) \qquad (1) \\ &= (\lfloor \tau' \rfloor \Leftarrow \texttt{any})(\texttt{any} \Leftarrow \lfloor \tau' \rfloor) \\ &\qquad ((\lambda x{:}\lfloor \tau \rfloor.\lfloor e \rfloor)((\lfloor \tau \rfloor \Leftarrow \texttt{any})(\texttt{any} \Leftarrow \lfloor \tau \rfloor)\lfloor v \rfloor)) \qquad (2) \\ &= (\lambda x{:}\lfloor \tau \rfloor.\lfloor e \rfloor)(\lfloor v \rfloor) \qquad (3)\end{aligned}$$

Figure 5.13: The wrapper lambdas in the early-checking translation can be optimized away if the callee is known at compile time.

as we discussed in Section 4.1. The early-checking translation for Pallene avoids this problem because casts between function types are effectively a no-op, as illustrated in Figure 5.12. For example, if a λ-Pallene function is successively cast using the `as` operator from `int`→`int` to `any`→`any` then further cast to another function type, no matter how many layers of type casting there are, the end result is always the same: a single typed lambda wrapped inside a lambda of type `any`→`any`.

Nevertheless, sometimes we can avoid the wrappers altogether. One very important special case is when the called function is known at compile time. As we show in Figure 5.13, if we are calling a known function then we can optimize away the `any`→`any` wrapper, as well as the need to box and unbox the function arguments and return values. The steps involved in the optimization are: (1) Convert the λ-Pallene term to PIR, using the early-checking translation. (2) Apply a β-reduction transformation to get rid of the x' lambda. (3) Optimize away the two pairs of casts of the form $(\lfloor \tau \rfloor \Leftarrow \texttt{any})(\texttt{any} \Leftarrow \lfloor \tau \rfloor)$. If τ is `any` then both casts expand to a no-op. Otherwise, they expand to $\texttt{unbox}_{\lfloor \tau \rfloor}(\texttt{box}_{\lfloor \tau \rfloor}\ e)$, in which case we can use reduction rule R-UNBOX to cancel out the box and the unbox.

$$
\begin{array}{lcl}
\lfloor n \rfloor_{\text{erase}} & = & n \\
\lfloor e_1 \texttt{ + } e_2 \rfloor_{\text{erase}} & = & \lfloor e_1 \rfloor \texttt{ + } \lfloor e_2 \rfloor \\
\lfloor \texttt{\{ } \overline{e} \texttt{ \}} \rfloor_{\text{erase}} & = & \texttt{\{ } \overline{\lfloor e \rfloor} \texttt{ \}} \\
\lfloor e_1 \texttt{[} e_2 \texttt{]} \rfloor_{\text{erase}} & = & \lfloor e_1 \rfloor \texttt{[} \lfloor e_2 \rfloor \texttt{]} \\
\lfloor x \rfloor_{\text{erase}} & = & x \\
\lfloor e_1 \texttt{(} e_2 \texttt{)} \rfloor_{\text{erase}} & = & \lfloor e_1 \rfloor \texttt{(} \lfloor e_2 \rfloor \texttt{)} \\
\lfloor \lambda x{:}\tau.e \rfloor_{\text{erase}} & = & \lambda x. \lfloor e \rfloor \\
\\
\lfloor e \texttt{ as } \tau \rfloor_{\text{erase}} & = & \lfloor e \rfloor \\
\\
\lfloor \texttt{box}_G \ e \rfloor_{\text{erase}} & = & \lfloor e \rfloor \\
\lfloor \texttt{unbox}_G \ e \rfloor_{\text{erase}} & = & \lfloor e \rfloor
\end{array}
$$

Figure 5.14: The type-erasure transformation removes all the type annotations and type casts from a λ-Pallene or PIR term, producing a λ-Dyn term.

The Pallene compiler implements this optimization. When a Pallene function calls another Pallene function which is known at compile time, we use a more efficient Pallene calling convention, which in PIR is represented by the inner typed lambda. Higher order functions and cross-module function calls still use the slower calling convention of passing the arguments as boxed objects via the Lua stack, which in PIR is represented by the `any`→`any` wrapper. However, this is not that bad because higher order functions are not as common in Lua and Pallene as they would be in a functional language. Furthermore, as we showed in Section 4.4, even the slower calling convention is already faster than Lua.

5.6 Gradual Guarantee

Back in Chapter 3 we said that one of the guiding principles of Pallene's design was the Gradual Guarantee of Siek et al. (Siek et al. 2015). It states that Pallene programs should evaluate to the same result as an equivalent Lua program obtained by erasing all the type annotations, except in cases where Pallene raises a run-time type error and Lua does not. In this section we formally enunciate and prove this in the context of λ-Pallene.

The first step is to clarify what we mean by erasing types. In Figure 5.14 we introduce a type-erasing function called $\lfloor\ \rfloor_{\text{erase}}$, which converts λ-Pallene terms to λ-Dyn by removing all the type annotations and type casts. We also define how to erase types from PIR terms, which will be useful because evaluation happens at the PIR level. Either way, erasing the PIR terms produced by the late-checking and early-checking translations has basically the same result as erasing the original λ-Pallene term:

- If we erase the types after applying the late-checking transformation then the result is exactly the same as if we erased the types from the original λ-Pallene program.
- If we erase the types after applying the early-checking transformation then the only difference is that it might introduce some wrapper lambdas of the form $(\lambda x'.(\lambda x.e)x')$. It is safe to optimize these to just $(\lambda x.e)$ and if we do so then the final result is identical to what we get by erasing the types of the original λ-Pallene term.

We also define type-erasure for PIR evaluation contexts. The notation $\lfloor C \rfloor$ denotes the λ-Dyn evaluation context that is obtained by removing all the box and unbox operations from the PIR evaluation context C. One useful lemma is the following: erasing the result of filling in the hole of a PIR context C is equivalent to filling in the hole of the erased context $\lfloor C \rfloor$. Another lemma is how type erasure interacts with variable substitution.

Lemma 5.8.
For all PIR evaluation contexts C and terms e, $\lfloor C\left[e\right] \rfloor_{erase} = \lfloor C \rfloor \left[\lfloor e \rfloor_{erase}\right]$

Proof. Follows from a simple induction on the structure of the PIR evaluation context. □

Lemma 5.9.
For all PIR terms e and values v, $\lfloor e[x \leftarrow v] \rfloor_{erase} = \lfloor e \rfloor [x \leftarrow \lfloor v \rfloor]$

Proof. Follows from a straightforward induction on structure of the term e. □

Now lets return to the matter of the gradual guarantee, starting with a concrete example. We will compare the result of evaluating the following λ-Pallene program either by converting it to PIR or by converting it to λ-Dyn via type erasure.

$$((\lambda x{:}\texttt{int}.\, x)\ \texttt{as any}{\to}\texttt{any})(\texttt{nil as any})$$

This program takes a typed identity function, casts it to type `any` $\to$ `any` and then calls it passing the wrong type of argument. The example uses `nil` but any non-int ground type would also suffice. The first case we analyze is type erasure:

$$\begin{aligned}
&\lfloor\ ((\lambda x{:}\texttt{int}.\, x)\ \texttt{as any}{\to}\texttt{any})(\texttt{nil as any})\ \rfloor_{\text{erase}} \\
&= (\lambda x.\, x)(\texttt{nil}) \\
&\longrightarrow_{dyn} \texttt{nil}
\end{aligned}$$

Next, we try converting the program to PIR using the late-checking conversion. It works similarly to the λ-Dyn version, except for the explicit boxing and unboxing.

$$
\begin{aligned}
&\lfloor\ ((\lambda x{:}\texttt{int}.\ x)\ \texttt{as any}{\to}\texttt{any})(\texttt{nil as any})\ \rfloor_{late} \\
&= (\texttt{unbox}_{\texttt{any}\to\texttt{any}}\ (\texttt{box}_{\texttt{any}\to\texttt{any}}(\lambda x{:}\texttt{any}.\ x)))(\texttt{box}_{\texttt{nil}}\ \texttt{nil}) \\
&\longrightarrow_{pir} (\lambda x{:}\texttt{any}.\ x)(\texttt{box}_{\texttt{nil}}\ \texttt{nil}) \\
&\longrightarrow_{pir} \texttt{box}_{\texttt{nil}}\ \texttt{nil}
\end{aligned}
$$

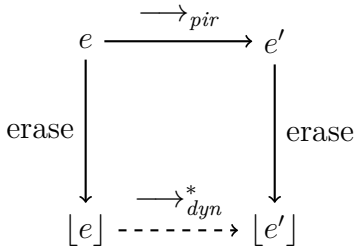
Finally, we look at what happens when using the early-checking conversion. This time the evaluation ends with a run-time error because of the $\texttt{unbox}_{\texttt{int}}\ (\texttt{box}_{\texttt{nil}}\ \texttt{nil})$.

$$
\begin{aligned}
&\lfloor\ ((\lambda x{:}\texttt{int}.\ x)\ \texttt{as any}{\to}\texttt{any})(\texttt{nil as any})\ \rfloor_{early} \\
&= (\lambda x'{:}\texttt{any}.\ \texttt{box}_{\texttt{int}}\ ((\lambda x{:}\texttt{int}.x)(\texttt{unbox}_{\texttt{int}}\ x')))(\texttt{box}_{\texttt{nil}}\ \texttt{nil}) \\
&\longrightarrow_{pir} \texttt{box}_{\texttt{int}}\ ((\lambda x{:}\texttt{int}.x)(\texttt{unbox}_{\texttt{int}}\ (\texttt{box}_{\texttt{nil}}\ \texttt{nil}))) \\
&\longrightarrow_{pir} \texttt{panic}
\end{aligned}
$$

The first thing that we can see is that PIR may encounter some run-time errors that are not encountered if we erase the types. The early-checking version raised an exception (`panic`) while the type-erasure version did not. This is because the early-checking semantics checks the type of the function argument. The late-checking semantics and the type-erasure only check the types of values when they are used; they do not check the types in the identity function because it only passes the value around.

The second thing this example shows us is that the compilation via the late-checking semantics and via type-erasure produced the same final value, except for the additional boxing. This is not a coincidence. We will prove that if compilation to PIR evaluates without errors to PIR term, then compilation to λ-Dyn will also produce the same result, up to type-erasure.

The core of the proof involves showing that one reduction step in PIR can be simulated by zero or more reduction steps in λ-Dyn, as illustrated by the following diagram:



It is important to note that the starting point is PIR, because that is where evaluation happens. An astute reader might wonder why not use λ-Pallene, since the informal statement of the gradual guarantee talked about erasing λ-Pallene terms, not PIR terms. The answer is that, as we discussed before, erasing the types of a λ-Pallene program has basically the same effect as erasing the types after conversion to PIR. Therefore, using PIR as the starting point works just as well. Moving on, we return to the proof:

Lemma 5.10. $(\Gamma \vdash_{pir} e : \tau) \wedge (e \longrightarrow_{pir} e')$ *implies* $\lfloor e \rfloor_{erase} \longrightarrow^*_{dyn} \lfloor e' \rfloor_{erase}$.

Proof. By case analysis on the hypothesis $(e \longrightarrow_{pir} e')$.

– In the R-ADD case our hypothesis is $C\Big[n_a \texttt{ + } n_b\Big] \longrightarrow_{pir} C\Big[n_c\Big]$ and we want to show that $\lfloor C\Big[n_a \texttt{ + } n_b\Big] \rfloor_{\text{erase}} \longrightarrow^*_{dyn} \lfloor C\Big[n_c\Big] \rfloor_{\text{erase}}$.

$$
\begin{aligned}
&\lfloor C\Big[n_a \texttt{ + } n_b\Big] \rfloor_{\text{erase}} && \\
&= \lfloor C \rfloor \Big[\lfloor n_a \texttt{ + } n_b \rfloor \Big] && \text{by Lemma 5.8} \\
&= \lfloor C \rfloor \Big[n_a \texttt{ + } n_b \Big] && \text{by definition of } \lfloor\ \rfloor_{\text{erase}} \\
&\longrightarrow_{dyn} \lfloor C \rfloor \Big[n_c \Big] && \text{using R-ADD} \\
&= \lfloor C \rfloor \Big[\lfloor n_c \rfloor \Big] && \text{by definition of } \lfloor\ \rfloor_{\text{erase}} \\
&= \lfloor C\Big[n_c\Big] \rfloor_{\text{erase}} && \text{by Lemma 5.8}
\end{aligned}
$$

– The R-INDEX case is similar to the one for addition. The third step, where we apply R-INDEX on the λ-Dyn side, uses the fact that the index n is inside the bounds of the array. We know this because of the hypothesis that the PIR term took a step using the R-INDEX rule.

$$
\begin{aligned}
&\lfloor C\Big[\{\overline{v}\}\texttt{[}n\texttt{]}\Big] \rfloor_{\text{erase}} && \\
&= \lfloor C \rfloor \Big[\lfloor \{\overline{v}\}\texttt{[}n\texttt{]} \rfloor \Big] && \text{by Lemma 5.8} \\
&= \lfloor C \rfloor \Big[\{\overline{\lfloor v \rfloor}\}\texttt{[}n\texttt{]} \Big] && \text{by definition of } \lfloor\ \rfloor_{\text{erase}} \\
&\longrightarrow_{dyn} \lfloor C \rfloor \Big[\lfloor v_n \rfloor \Big] && \text{using R-INDEX} \\
&= \lfloor C\Big[v_n\Big] \rfloor_{\text{erase}} && \text{by Lemma 5.8}
\end{aligned}
$$

– The R-MISSINGKEY case uses the same logic as the one for R-INDEX, but this time the hypothesis tells us that the index n is out of bounds.

$$
\begin{aligned}
&\lfloor C\Big[\{\overline{v}\}[n]\Big]\rfloor_{\text{erase}} \\
&= \lfloor C\rfloor\Big[\lfloor\{\overline{v}\}[n]\rfloor\Big] && \text{by Lemma 5.8} \\
&= \lfloor C\rfloor\Big[\{\overline{\lfloor v\rfloor}\}[n]\Big] && \text{by definition of } \lfloor\ \rfloor_{\text{erase}} \\
&\longrightarrow_{dyn} \lfloor C\rfloor\Big[\texttt{nil}\Big] && \text{using R-MISSINGKEY} \\
&= \lfloor C\rfloor\Big[\lfloor\texttt{box}_{\texttt{nil}}\,\texttt{nil}\rfloor\Big] && \text{by definition of } \lfloor\ \rfloor_{\text{erase}} \\
&= \lfloor C\Big[\texttt{box}_{\texttt{nil}}\,\texttt{nil}\Big]\rfloor_{\text{erase}} && \text{by Lemma 5.8}
\end{aligned}
$$

– In the R-APP case the key step uses Lemma 5.9 about substitution.

$$
\begin{aligned}
&\lfloor C\Big[(\lambda x{:}\tau.e)(v)\Big]\rfloor_{\text{erase}} \\
&= \lfloor C\rfloor\Big[\lfloor(\lambda x{:}\tau.e)(v)\rfloor\Big] && \text{by Lemma 5.8} \\
&= \lfloor C\rfloor\Big[(\lambda x.\lfloor e\rfloor)(\lfloor v\rfloor)\Big] && \text{by definition of } \lfloor\ \rfloor_{\text{erase}} \\
&\longrightarrow_{dyn} \lfloor C\rfloor\Big[\lfloor e\rfloor[x \leftarrow \lfloor v\rfloor]\Big] && \text{using R-APP} \\
&= \lfloor C\rfloor\Big[\lfloor e[x \leftarrow v]\rfloor\Big] && \text{by Lemma 5.9} \\
&= \lfloor C\Big[e[x \leftarrow v]\Big]\rfloor_{\text{erase}} && \text{by Lemma 5.8}
\end{aligned}
$$

– In the R-UNBOX case the λ-Dyn part does not need any reduction steps.

$$
\begin{aligned}
&\lfloor C\Big[\texttt{unbox}_G\,(\texttt{box}_G\,v)\Big]\rfloor_{\text{erase}} \\
&= \lfloor C\rfloor\Big[\lfloor\texttt{unbox}_G\,(\texttt{box}_G\,v)\rfloor\Big] && \text{by Lemma 5.8} \\
&= \lfloor C\rfloor\Big[\lfloor v\rfloor\Big] && \text{by definition of } \lfloor\ \rfloor_{\text{erase}} \\
&= \lfloor C\Big[v\Big]\rfloor_{\text{erase}} && \text{by Lemma 5.8}
\end{aligned}
$$

– The last case is R-ERROR, which is a case that can never happen. The hypothesis ($e \longrightarrow_{pir} e'$) states that the result is a PIR expression but `panic` is not a PIR expression.

□

Finally, we complete the proof of the gradual guarantee theorem by extending it to any number of PIR evaluation steps.

Theorem 5.11 (Gradual guarantee for PIR)**.**
$(\Gamma \vdash_{pir} e : \tau) \wedge (e \longrightarrow^*_{pir} e')$ *implies* $\lfloor e\rfloor_{erase} \longrightarrow^*_{dyn} \lfloor e'\rfloor_{erase}$.

Proof. By induction on the number of steps of $\longrightarrow^*_{pir}$. The base case is trivial: if the PIR evaluation took zero steps then e is equal to e' and therefore $\lfloor e \rfloor$ evaluates to $\lfloor e' \rfloor$ in zero steps. The inductive case follows from Lemma 5.10 and the transitivity of $\longrightarrow^*_{dyn}$. □

One interesting corollary of Theorem 5.11 is that if a λ-Dyn term gets stuck during the evaluation then any corresponding λ-Pallene term will panic at run time (assuming it is well-typed in the first place). From the contrapositive of the gradual guarantee theorem, if the erased term does not evaluate to a value then the PIR term also does not evaluate to a value. As the soundness theorem from PIR says that well-typed PIR terms can't get stuck, we can conclude that the only possibility is that the PIR term evaluates to `panic`.

5.7 Mutable PIR

In the interest of keeping the presentation and the proofs clear, we have so far formalized a purely-functional fragment of Pallene. This begs the question: have we simplified too much? Do these results still apply if we add more language features to the model? In this section, we address some of these issues by sketching a way to add Lua-like mutable tables to Pallene IR.

One of the defining features of Lua is its versatile table datatype. As we mentioned before, Lua tables are associative arrays that map keys to values. Any Lua value can be used as a key or as a value. Many Lua data structures are implemented using tables; for example, arrays are just tables that happen to have sequential integer keys and records are just tables with string keys. In this section we will show how PIR would look like with more general tables instead of just arrays.

In addition to allowing other kinds of table keys, we will make the tables mutable, because Lua is an imperative language[1]. This is something that needs to be done carefully because it can cause problems if it is not done correctly. In particular, there must be an answer to the problem of dynamically typed code mutating typed objects in a way that disrespects the types.

One way that some gradually typed languages attack this problem is using "guarded objects". The idea is that when a typed object is passed to untyped code, the runtime wraps it inside a proxy object which checks the types of all the writes that happen through the proxy. One example of this approach is the Chaperone and Impersonator system of Racket (Strickland et al. 2012).

[1]In this section we will focus on tables and ignore the matter of mutable variables. That said, the latter could be modeled using a standard assignment-conversion step (Kranz 1986).

A downside of the guarded object approach is that it can have a high cost both in terms of time and space performance (Takikawa et al. 2016). Furthermore, the wrappers do not interact well with object identity, which is something that is important in many languages, including Lua. In order to preserve object identity in the presence of wrappers the interpreter and the runtime must be modified to bypass the wrappers in all the operations that care about object identity.

In the case of Pallene, the performance problems of proxy objects, together with the desire to not modify the core Lua runtime, encouraged us to look for a different approach. The one we ended up using is similar to the Transient Checking strategy of Reticulated Python (Vitousek et al. 2014) or the "Execution Strategy" of GradualTalk (Allende et al. 2013). The basic idea is that whenever Pallene reads from a Lua table, it must check if the type of the value is the one it expects, before using the value. This places the burden of run-time type checking on the typed Pallene code, as opposed to the guarded object strategy, where the highest cost is paid when dynamic code attempts to access typed objects. However, as we showed in Section 3.5, if the compiler is designed to take advantage of the type system then these run-time type checks can be quite cheap and more than compensated by the speed boost from the types.

Let's now discuss how exactly we can add support for Lua-like tables to PIR. We will present an extension of PIR called Mutable PIR and discuss how the previous proofs may be adapted to it.

Syntax The main syntactic differences between purely-functional PIR and mutable PIR are shown in Figure 5.15. Instead of arrays, we now have tables. They are also constructed using curly braces but the table constructor only allows the construction of empty tables. To insert more elements or to mutate existing elements, there is now an assignment operation. The expression $v_1\texttt{[}v_2\texttt{]} = v_3$ assigns v_3 to the key v_2 of table v_1.

We also introduced expressions for table addresses, written α. They represent the location of the table object in the Lua heap. Expressions that have a table type evaluate to one of these addresses. This is a classic technique for modeling mutable references in a lambda calculus, which will be clearer when we talk about the reduction rules for Mutable PIR. Nevertheless, it is worth noting that these table addresses are never directly created by the programmer. They only appear during the evaluation of the Mutable PIR programs.

e	:=		EXPRESSIONS
	|	`{ }`	*empty table constructor*
	|	e_1`[`e_2`]`	*table read*
	|	e_1`[`e_2`] = `e_3	*table write*
	|	α	*table address**
	|	...	*(rest is the same as PIR)*
τ	:=		TYPES
	|	`any`	*dynamic type*
	|	`int`	*integer type*
	|	`nil`	*nil type*
	|	`string`	*string type*
	|	`table`	*table type*
	|	$\tau \to \tau$	*function type*

Figure 5.15: New syntax in Mutable PIR. The table addresses are used during the evaluation of Mutable PIR terms but never directly created by the translation from λ-Pallene.

$$\text{(T-CONS)}\ \frac{}{\Gamma \vdash \texttt{\{\}} : \texttt{table}} \qquad \text{(T-INDEX)}\ \frac{\Gamma \vdash e_1 : \texttt{table} \qquad \Gamma \vdash e_2 : \texttt{any}}{\Gamma \vdash e_1\texttt{[}e_2\texttt{]} : \texttt{any}}$$

$$\text{(T-ADDR)}\ \frac{}{\Gamma \vdash \alpha : \texttt{table}} \qquad \text{(T-WRITE)}\ \frac{\Gamma \vdash e_1 : \texttt{table} \qquad \Gamma \vdash e_2, e_3 : \texttt{any}}{\Gamma \vdash e_1\texttt{[}e_2\texttt{]=}e_3 : \texttt{nil}}$$

Figure 5.16: New type rules in Mutable PIR

Types To support Lua-like tables we have to make changes to the type system, as indicated by Figure 5.16. Firstly, the array type `{any}` is renamed to `table`. The main difference is that the keys can now be any boxed Lua value. In the rules T-INDEX and T-WRITE, the e_2 term has type `any` instead of `integer`.

Another thing that we can add to λ-Pallene and Mutable PIR to make things more interesting is a string type, which paves the way for adding a record type to λ-Pallene. Recall from Chapter 3 that Pallene arrays and records are both implemented as Lua tables. This is reflected in mutable PIR: both are encoded using type `table` and thus have the same type tag at run time. Therefore, if Lua passes an array to a Pallene function that expects a record, it will initially succeed because the type tag is also `table`. It will only fail when Pallene tries to read one of the table keys and obtains `nil` instead of whatever it was expecting.

Mutable-PIR evaluation

$$
\begin{array}{lcll}
v & := & & \textsc{Values} \\
 & | & n & \textit{integer value} \\
 & | & \lambda x{:}\tau.\ e & \textit{lambda value} \\
 & | & \texttt{box}_G\ v & \textit{boxed value} \\
 & | & \alpha & \textit{table address} \\
 & & & \\
r & := & & \textsc{Results} \\
 & | & e & \textit{successful step} \\
 & | & \texttt{panic} & \textit{run-time error} \\
 & & & \\
\mu & : & \alpha \to (\texttt{any} \to \texttt{any}) & \textsc{Lua Heap}
\end{array}
$$

Reduction relation $\boxed{\mu|e \longrightarrow_{pir} \mu|r}$

$$
\begin{array}{lllll}
\text{(R-CONS)} & \mu \mid C\Big[\texttt{\{\}}\Big] & \longrightarrow & \mu[\alpha \mapsto \emptyset] \mid C\Big[\alpha\Big], & \text{where } \alpha \text{ is fresh} \\
\text{(R-INDEX)} & \mu \mid C\Big[\alpha\texttt{[}v\texttt{]}\Big] & \longrightarrow & \mu \mid C\Big[\mu[\alpha][v]\Big], & v \in \text{keys}(\mu[\alpha]) \\
\text{(R-INDEXNIL)} & \mu \mid C\Big[\alpha\texttt{[}v\texttt{]}\Big] & \longrightarrow & \mu \mid C\Big[\texttt{box}_{\texttt{nil}}\,\texttt{nil}\Big], & v \notin \text{keys}(\mu[\alpha]) \\
\text{(R-WRITE)} & \mu \mid C\Big[\alpha\texttt{[}v\texttt{]=}v'\Big] & \longrightarrow & \mu[\alpha][v \mapsto v'] \mid C\Big[\texttt{nil}\Big] & v \neq \texttt{box}_{\texttt{nil}}\,\texttt{nil} \\
\text{(R-WRITENIL)} & \mu \mid C\Big[\alpha\texttt{[}v\texttt{]=}v'\Big] & \longrightarrow & \mu \mid \texttt{panic} & v = \texttt{box}_{\texttt{nil}}\,\texttt{nil}
\end{array}
$$

Figure 5.17: New reduction rules in Mutable PIR

Reduction Next, we define the reduction rules for Mutable PIR. They are shown in Figure 5.17. The reduction relation takes an additional parameter μ, representing the Lua heap. This heap maps table addresses α to their contents, which themselves are a map from keys to values.

Rule R-CONS shows how new tables are created. When we evaluate a table constructor expression `{}`, the result is a fresh table address α. The notation $\mu[\alpha \mapsto \emptyset]$ means that we extend the Lua heap with a mapping from α to an empty set, representing the newly created empty table.

The R-WRITE operation is the other operation that modifies the state. If the key is not present in the table then it adds a new key-value pair to it. Otherwise, it overwrites an existing key-value pair. Using `nil` as a key is not allowed and attempting to do so will raise a run-time exception, just like in Lua.

In the other direction, we have the rules R-INDEX and R-INDEXNIL, which read from the heap state. If the key v is present in the table contents $\mu[\alpha]$, then evaluation proceeds with rule R-INDEX, producing value $\mu[\alpha][v]$. If the key

does not exist in the table then the evaluation produces a boxed `nil` instead.

Translations Finally, we have to update the early-checking and late-checking translations to account for arrays being replaced by tables. The main difference is that the array keys are now boxed. For the late-checking translation this means that we can now use the key without unboxing it:

$$\lfloor e_1\,\texttt{[}e_2\texttt{]}\rfloor_{late} = (\texttt{unbox}_{\texttt{table}}\,\lfloor e_1\rfloor)\texttt{[}\lfloor e_2\rfloor\texttt{]}$$

For the early-checking translation we have to box the array key:

$$\lfloor e_1\,\texttt{[}e_2\texttt{]} : \tau\rfloor_{early} = (\lfloor\tau\rfloor \Leftarrow \texttt{any})(\lfloor e_1\rfloor\,\texttt{[box}_{\texttt{int}}\lfloor e_2\rfloor\texttt{]})$$

Proofs The proofs of soundness for PIR and for the gradual guarantee can be adapted to Mutable PIR without too much trouble. The crucial detail that makes this possible is that arrays in PIR are fundamentally dynamic and we do not maintain any invariants involving the array contents. The arrays always contain boxed values of type `any`, which can be anything boxable. This matches the memory model of the Lua heap in Mutable PIR, where the contents of the tables are also boxed values of type `any`. The only invariant that we have to maintain is that $\mu[\alpha]$ is always well defined; all addresses α that appear in the program also appear on the Lua heap. We now sketch how each of the proofs in this chapter could be extended to Mutable PIR.

In Lemma 5.1, the Canonical Forms Lemma, we have to replace the case for arrays with a case that says that if a value has type `table` then it takes the form of a table address α.

In Lemma 5.2, the Progress Lemma for PIR, the tricky thing we need to show is that the α that we get when evaluating a table must be part of the heap μ, so that $\mu[\alpha]$ is well defined in the reduction rules. This follows from the fact that table addresses are only created by the R-CONS rule, which also adds them to the heap at the same time. Note that in our model, table addresses are never removed from the heap.

Next, we have Lemma 5.3, the Preservation Lemma for PIR. The proof for the T-INDEX still works with Mutable PIR because the contents of the tables are also boxed values of type `any`. We also need to add a case for the array-write operations but the logic is similar to the array-read case.

The Soundness Theorems for early-checking and late-checking Pallene (Theorems 5.4–5.7) are essentially unchanged. The only important difference is that all the references to `{any}` are replaced by `table`.

Finally, for the proof of the gradual guarantee (Lemma 5.10) we would have to update the definitions of λ-Dyn and $\lfloor \, \rfloor_{\text{erase}}$ to include mutable tables. The important thing is that the Lua heap is essentially the same in Mutable PIR and λ-Dyn. The only difference is that in Mutable PIR the boxing around all the values is explicit while in λ-Dyn it would be implicit. But at the end of the day, the two heaps map the same keys to the same values. This codifies the principle that Pallene shares the same runtime and garbage collector as Lua and that tables can be passed between Pallene and Lua without needing to be converted.

6
Conclusion

Dynamic languages are popular due to their simplicity and flexibility, however that same flexibility also makes it hard to optimize their performance. Programmers and language designers have approached this problem from many directions, including scripting, just-in-time compilers, and optional type systems. All these approaches have shortcomings. In the case of scripting, the overhead of converting data as it crosses the boundary between the languages can offset the advantage of using the system language. In the case of JIT compilers, the complexity of their implementations can pose many challenges: they are hard to implement, maintain, and port to new architectures. The optimization that JIT compilers provide is also not uniform. There is a subset of the language that can be compiled to machine code and which programmers are implicitly encouraged to conform to. Finally, in the case of optional type systems, it is not easy to design a type system that can balance correctness, performance, and simplicity. Performance in particular has been a sticking point.

In this thesis we described another approach to the problem of the performance of dynamic languages. This approach combines positive aspects from scripting, JIT compilation, and optional types, while avoiding some of their shortcomings. Motivated by Lua, we also wanted an approach that could be implemented in a way that was simple, maintainable, and easy to port to different architectures.

Our proposal is to use ideas from gradual typing and just-in-time compilers to produce a typed companion language for an existing dynamic language. The typed language plays the role of a system language in the scripting paradigm, but one designed from the start to be used in conjunction with the dynamic language. Our example of this is Pallene, a typed language designed to be scripted from Lua.

Pallene is a typed subset of Lua which uses run-time tag checks to enforce type safety when interacting with untyped Lua code. This tag-checking strategy is getting good performance because the tag checks are relatively cheap and more than compensated by the speedup from the types. Pallene's runtime is also tightly integrated with Lua's. The garbage collector is shared

with Lua and objects can be exchanged between Lua and Pallene with negligible overhead.

In a set of benchmarks from the Computer Language Benchmarks Game (CLBG), Pallene achieved speedups between 1.5× and 15× when compared to the reference Lua interpreter, with a geometric mean speedup of 7.2× . These results were comparable with the speedups achieved by LuaJIT, a JIT compiler for Lua. In the same benchmarks, LuaJIT achieved speedups between 5× and 20×, with a mean of 8.5×. These good results for Pallene happen despite its relatively simple implementation and the fact that we are still working on adding many compiler optimizations. Pallene's compiler is written in Lua and C, adding up to 8 000 lines of code. This is less than the 28 000 lines of C code in the reference Lua interpreter and is much less than the 135 000 lines of C and assembly language in LuaJIT.

Pallene exhibited good performance even when mixed with Lua. In the subset of CLBG benchmarks that could be separated into multiple modules, we evaluated the performance of combining Lua and Pallene modules in the same program. For each program, we measured the performance of every possible configuration of modules, with each module being implemented in either Lua or Pallene. We observed that replacing Lua with Pallene usually improved the overall performance of the program, and never worsened it. This compares favorably with what has been reported for other gradually typed languages, where the performance of configurations that mix typed and untyped modules often are worse than the performance of fully untyped programs. Pallene also avoided the problem we encountered in other benchmark experiments we performed, where sometimes rewriting a Lua module in C would result in an overall performance that was worse than keeping the code in Lua.

In this thesis we have also provided the basis for a formalization of Pallene's semantics. We introduced an intermediate representation for Pallene, called PIR, which models when Pallene programs perform run-time tag checks and when they convert data between tagged and untagged representations. PIR provides a formal notation for exploring potential semantics for Pallene. These different semantics correspond to different strategies for inserting run-time tag checks when compiling Pallene to PIR. One of the most interesting aspects of PIR is that tag checks also correspond to locations in the program where a tagged value is converted to an untagged one, which has implications for performance. Checking tags sooner may allow values to be stored in a more efficient untagged representation. On the other hand, checking tags later may allow some tag checks to be skipped if they only happen conditionally.

Finally, in this thesis we also present some ideas that are applicable

to the problem of designing efficient gradually-typed languages. The first of these ideas is something fairly obvious. If the typed language is compiled to the dynamic language with added run-time checks, then performance is likely to be as slow as the dynamic language or worse. However, if the compiler takes advantage of the types to generate efficient type-specialized code, then the performance can be much better. We show that when the run-time type checking takes the form of tag checks it can be quite cheap. In Pallene the cost of the run-time tag checks is usually less than 10% and sometimes even zero due to the CPU's instruction pipelining and branch prediction.

The second idea that Pallene highlights is the observation that the optimizations in just-in-time compilers are analogous to the optimizations that an ahead-of-time compiler for a gradually typed language may perform. In both cases the optimizer uses type information to generate efficient machine code. The difference is that JIT compilers obtain that type information by profiling the program while it is executing, while in a gradually typed language the types would be known from the type annotations. Since JIT compilation has been found useful in many dynamic languages, this suggests that many dynamic languages may contain an efficient typed language subset inside them.

The third idea from Pallene that can be applied in the context of gradual typing is that perhaps the typed language should be designed to complement the dynamic language and not to supersede it. The goal of designing a type system for good performance may push the design in a different direction than the goal of supporting a wide variety of idioms from the dynamic language. This is exemplified by the many differences between Pallene and Typed Lua, a gradually typed variant of Lua which was designed with the latter goal in mind. Another reason why we intentionally designed Pallene to be a subset ot Lua, instead of a superset, is that we observed that some parts of Lua do not benefit as much from compilation to machine code. Pallene leaves out many of the more "dynamic" features of Lua which are hard to generate fast code for. We focus the design of Pallene and our implementation effort on the parts of Lua where ahead-of-time compilation is most advantageous.

Bibliography

[Adams et al. 2014] ADAMS, K.; EVANS, J.; MAHER, B.; OTTONI, G.; PAROSKI, A.; SIMMERS, B.; SMITH, E. ; YAMAUCHI, O.. **The Hiphop virtual machine**. In: PROCEEDINGS OF THE 2014 ACM INTERNATIONAL CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES & APPLICATIONS, OOPSLA '14, p. 777–790, 2014.

[Allende et al. 2013] ALLENDE, E.; FABRY, J. ; TANTER, E.. **Cast insertion strategies for gradually-typed objects**. In: PROCEEDINGS OF THE 9TH SYMPOSIUM ON DYNAMIC LANGUAGES, DLS '13, p. 27–36, 2013.

[Ancona 2007] ANCONA, D.; ANCONA, M.; CUNI, A. ; MATSAKIS, N. D.. **RPython: A step towards reconciling dynamically and statically typed OO languages**. In: PROCEEDINGS OF THE 2007 SYMPOSIUM ON DYNAMIC LANGUAGES, DLS, 2007.

[Antonov et al. 2007] ANTONOV, P.; OTHERS. **V8 optimization killers**, 2013. Retrieved in 2017-01-08. `https://github.com/petkaantonov/bluebird/wiki/Optimization-killers`.

[Barret et al. 2017] BARRETT, E.; BOLZ-TEREICK, C. F.; KILLICK, R.; MOUNT, S. ; TRATT, L.. **Virtual machine warmup blows hot and cold**. In: PROCEEDINGS OF THE 32ND ANNUAL CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, OOPSLA, 2017.

[Bauman et al. 2015] BAUMAN, S.; BOLZ, C. F.; HIRSCHFELD, R.; KIRILICHEV, V.; PAPE, T.; SIEK, J. G. ; TOBIN-HOCHSTADT, S.. **Pycket: A tracing JIT for a functional language**. In: PROCEEDINGS OF THE 20TH ACM SIGPLAN INTERNATIONAL CONFERENCE ON FUNCTIONAL PROGRAMMING, ICFP, 2015.

[Behnel et al. 2010] BEHNEL, S.; BRADSHAW, R.; CITRO, C.; DALCIN, L.; SELJEBOTN, D. ; SMITH, K.. **Cython: The best of both worlds**. Computing in Science Engineering, 2011.

[Bierman et al. 2014] BIERMAN, G.; ABADI, M. ; TORGERSEN, M.. **Understanding TypeScript**. In: 28TH EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, ECOOP, 2014.

[Biggar 2010] BIGGAR, P.. **Design and implementation of an ahead-of-time compiler for PHP**. PhD thesis, Trinity College Dublin, 2010. `https://paulbiggar.com/research/#phd-dissertation`.

[Bolz et al. 2009] BOLZ, C. F.; CUNI, A.; FIJALKOWSKI, M. ; 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, ICOOOLPS, 2009.

[Bracha 2004] BRACHA, G.. **Pluggable type systems**. OOPSLA Workshop on Revival of Dynamic Languages, 2004. `http://bracha.org/pluggableTypesPosition.pdf`.

[Bracha et al. 1998] BRACHA, G.; ODERSKY, M.; STOUTAMIRE, D. ; WADLER, P.. **Making the future safe for the past: Adding genericity to the java programming language**. In: PROCEEDINGS OF THE 13TH ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, OOPSLA '98, p. 183–200, 1998.

[Campora et al.] CAMPORA, J. P.; CHEN, S. ; WALKINGSHAW, E.. **Casts and costs: Harmonizing safety and performance in gradual typing**. Proceedings of the ACM on Programming Languages, 2018.

[DeVito 2014] DEVITO, Z.. **Terra: Simplifying High-Performance Programming Using Multi-Stage Programming**. PhD thesis, Stanford University, 2014.

[Deutsch & Schiffman 1984] DEUTSCH, L. P.; SCHIFFMAN, A. M.. **Efficient implementation of the Smalltalk-80 system**. In: PROCEEDINGS OF THE 11TH ACM SIGACT-SIGPLAN SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, POPL, 1984.

[Futamura 1999] FUTAMURA, Y.. **Partial evaluation of computation process, revisited**. Higher Order Symbolic Computation, 1999.

[Gal et al. 2006] GAL, A.; PROBST, C. W. ; FRANZ, M.. **HotpathVM: An effective JIT compiler for resource-constrained devices**. In: PRO-

CEEDINGS OF THE 2ND INTERNATIONAL CONFERENCE ON VIRTUAL EXECUTION ENVIRONMENTS, VEE, 2006.

[Giraldez 2016] GUERRA GIRALDEZ, J.. **LOOM - a LuaJIT performance visualizer**, 2016. `https://github.com/cloudflare/loom`.

[Giraldez 2017] GUERRA GIRALDEZ, J.. **LuaJIT hacking: Getting next() out of the NYI list**. CloudFare Blog, 2017. `https://blog.cloudflare.com/luajit-hacking-getting-next-out-of-the-nyi-list/`.

[Graham 1995] GRAHAM, P.. **ANSI Common LISP**. Apt, Alan R., 1996.

[Greenman & Felleisen, 2018] GREENMAN, B.; FELLEISEN, M.. **A spectrum of type soundness and performance**. Proc. ACM Program. Lang., 2018.

[Gualandi & Ierusalimschy 2018] GUALANDI, H. M.; IERUSALIMSCHY, R.. **Pallene: a statically typed companion language for lua**. In: PROCEEDINGS OF THE XXII BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES, SBLP, 2018.

[Gualandi & Ierusalimschy 2020] GUALANDI, H.; IERUSALIMSCHY, R.. **Pallene: A companion language for lua**. Science of Computer Programming, 2020.

[Gualandi 2015] GUALANDI, H. M.. **Typing dynamic languages – a review**. Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), 2015.

[Guoy 2013] GOUY, I.. **The computer language benchmarks game**, 2013. `https://benchmarksgame-team.pages.debian.net/benchmarksgame`.

[Ierusalimschy et al. 2005] IERUSALIMSCHY, R.; DE FIGUEIREDO, L. H. ; CELES, W.. **The implementation of Lua 5.0**. Journal Universal Computer Science, 2005.

[Ierusalimschy et al. 2007] IERUSALIMSCHY, R.; DE FIGUEIREDO, L. H. ; CELES, W.. **The evolution of Lua**. In: PROCEEDINGS OF THE THIRD ACM SIGPLAN CONFERENCE ON HISTORY OF PROGRAMMING LANGUAGES, HOPL, 2007.

[Ierusalimschy et al. 2011] IERUSALIMSCHY, R.; DE FIGUEIREDO, L. H. ; CELES, W.. **Passing a language through the eye of a needle**. Communications of the ACM, 2011.

[Ierusalimschy et al. 2018] IERUSALIMSCHY, R.; DE FIGUEIREDO, L. H. ; CELES, W.. **A look at the design of Lua**. Communications of the ACM, 61(11):114–123, Oct. 2018.

[Ierusalimschy et al. 2020] IERUSALIMSCHY, R.; FIGUEIREDO, L. H. D. ; CELES, W.. **Lua 5.4 reference manual**. http://www.lua.org/manual/5.4/, 2020.

[Igarashi 2001] IGARASHI, A.; PIERCE, B. C. ; WADLER, P.. **Featherweight java: A minimal core calculus for java and gj**. ACM Transactions on Programming Languages and Systems, 23(3):396–450, May 2001.

[Kranz 1986] KRANZ, D.; ADAMS, N.; KELSEY, R.; REES, J.; HUDAK, P. ; PHILBIN, J.. **Orbit: An optimizing compiler for scheme**. In: PROCEEDINGS OF THE 1986 SIGPLAN SYMPOSIUM ON COMPILER CONSTRUCTION, SIGPLAN '86, p. 219–233. Association for Computing Machinery, 1986.

[Kuhlenschmidt et al. 2019] KUHLENSCHMIDT, A.; ALMAHALLAWI, D. ; SIEK, J. G.. **Toward efficient gradual typing for structural types via coercions**. In: PROCEEDINGS OF THE 40TH ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, PLDI, 2019.

[Lattner 2002] LATTNER, C.. **Llvm: An infrastructure for multi-stage optimization**. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. See https://www.llvm.org/pubs/.

[Ligneul 2016] LIGNEUL, G.. **Um compilador de lua vm para llvm**. Technical report, Pontificia Universidade Católica do Rio de Janeiro, 2016.

[Maidl et al. 2015] MAIDL, A. M.; MASCARENHAS, F. ; IERUSALIMSCHY, R.. **A formalization of Typed Lua**. In: PROCEEDINGS OF THE 11TH SYMPOSIUM ON DYNAMIC LANGUAGES, DLS, 2015.

[Manura 2008] MANURA, D.. **The lua2c compiler**. Source code repository for the lua2c compiler., 2008. `https://github.com/davidm/lua2c/`.

[Mogilefsky 1999] MOGILEFSKY, B.. **Lua in Grim Fandango**. Grim Fandango Network, May 1999. `https://www.grimfandango.net/features/articles/lua-in-grim-fandango`.

[Moura & Ierusalimschy, 2009] DE MOURA, A. L.; IERUSALIMSCHY, R.. **Revisiting coroutines**. ACM Transactions on Programming Languages and Systems, 31(2):6:1–6:31, Feb. 2009.

[Muehlboeck & Tate 2017] MUEHLBOECK, F.; TATE, R.. **Sound gradual typing is nominally alive and well**. Proceedings of the ACM on Programming Languages, 2017.

[Ousterhout 1998] OUSTERHOUT, J. K.. **Scripting: Higher-level programming for the 21st century**. Computer, 1998.

[Pall 2004] PALL, M.. **Coco — true C coroutines for Lua**, 2004. `https://coco.luajit.org`.

[Pall 2005] PALL, M.. **LuaJIT, a just-in-time compiler for Lua**, 2005. `http://luajit.org/luajit.html`.

[Pall 2009] PALL, M.. **LuaJIT 2.0 intellectual property disclosure and research opportunities**, 2009. `http://lua-users.org/lists/lua-l/2009-11/msg00089.html`.

[Pall 2012] PALL, M.. **LuaJIT performance tips**, 2012. `http://wiki.luajit.org/Numerical-Computing-Performance-Guide`.

[Pall 2014] PALL, M.; OTHERS. **Not Yet Implemented operations in LuaJIT**, 2014. `http://wiki.luajit.org/NYI`.

[Pierce 2002] PIERCE, B. C.. **Types and Programming Languages**. MIT Press, 2002.

[PyPy 2011] THE PYPY PROJECT. **Call for donations - pypy to support python3!** `https://pypy.org/py3donate.html`, 2011.

[PyPy 2016] THE PYPY PROJECT. **RPython official documentation**, 2016. `https://rpython.readthedocs.io/`.

[Richards et al. 2017] RICHARDS, G.; ARTECA, E. ; TURCOTTE, A.. **The VM already knew that: Leveraging compile-time knowledge to optimize gradual typing**. Proceedings of the ACM on Programming Languages, 2017.

[Siek & Taha 2006] SIEK, J. G.; TAHA, W.. **Gradual typing for functional languages**. Scheme and Functional Programming Workshop, 2006.

[Siek et al. 2015] SIEK, J. G.; VITOUSEK, M. M.; CIMINI, M. ; BOYLAND, J. T.. **Refined criteria for gradual typing**. In: 1ST SUMMIT ON ADVANCES IN PROGRAMMING LANGUAGES, SNAPL, 2015.

[Southern & Renau 2016] SOUTHERN, G.; RENAU, J.. **Overhead of deoptimization checks in the V8 javascript engine**. In: 2016 INTERNATIONAL SYMPOSIUM ON WORKLOAD CHARACTERIZATION, IISWC, 2016.

[Strickland et al. 2012] STRICKLAND, T. S.; TOBIN-HOCHSTADT, S.; FINDLER, R. B. ; FLATT, M.. **Chaperones and impersonators: Run-time support for reasonable interposition**. In: PROCEEDINGS OF THE ACM INTERNATIONAL CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS, OOPSLA '12, p. 943–962. ACM, 2012.

[Takikawa et al. 2016] TAKIKAWA, A.; FELTEY, D.; GREENMAN, B.; NEW, M. S.; VITEK, J. ; FELLEISEN, M.. **Is sound gradual typing dead?** In: PROCEEDINGS OF THE 43RD SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, POPL, 2016.

[Tobin-Hochstadt & Felleisen] TOBIN-HOCHSTADT, S.; FELLEISEN, M.. **Interlanguage migration: From scripts to programs**. In: 21ST SYMPOSIUM ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, OOPSLA, 2006.

[Vitousek et al. 2014] VITOUSEK, M. M.; KENT, A. M.; SIEK, J. G. ; BAKER, J.. **Design and evaluation of gradual typing for Python**. In: PROCEEDINGS OF THE 10TH ACM SYMPOSIUM ON DYNAMIC LANGUAGES, DLS, 2014.

[Vitousek et al. 2017] VITOUSEK, M. M.; SWORDS, C. ; SIEK, J. G.. **Big types in little runtime: Open-world soundness and collaborative blame for gradual type systems**. In: PROCEEDINGS OF THE 44TH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, POPL, 2017.

[Vitousek et al. 2019] VITOUSEK, M. M.; SIEK, J. G. ; CHAUDHURI, A.. **Optimizing and evaluating transient gradual typing**. In: PROCEEDINGS OF THE 15TH ACM SIGPLAN INTERNATIONAL SYMPOSIUM ON DYNAMIC LANGUAGES, DLS, 2019.

[Würthinger et al. 2013] WÜRTHINGER, T.; WIMMER, C.; WÖSS, A.; STADLER, L.; DUBOSCQ, G.; HUMER, C.; RICHARDS, G.; SIMON, D. ; 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, Onward, 2013.

[Zhang 2011] ZHANG, Y.. **Openresty, scalable web platform by extending NGINX with Lua**, 2011. `https://openresty.org/en/`.