

MSc Computer Science
Final Project

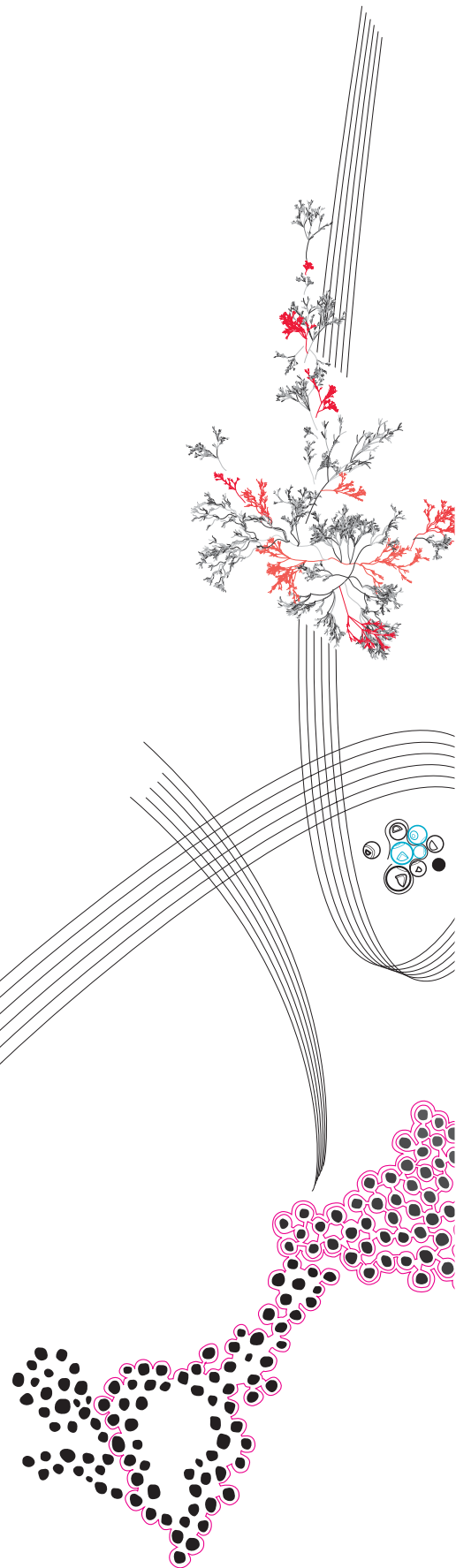
Towards gradual typing: a
type-shifting Python language
implementation

Manzi Aimé Ntagengerwa

Supervisor: Peter Lammich

August, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente



Contents

1	Introduction	2
1	Motivation	2
2	Research questions	3
3	Related Work	4
3.1	Codon	4
3.2	Mypy	5
3.3	Reticulated Python	5
3.4	TypeScript	5
3.5	Cinder and Static Python	5
2	Background	7
1	Program execution	7
1.1	Interpretation	7
1.2	Compilation	8
1.3	Abstraction Reduction	8
1.4	Just-in-time compilation	8
2	Common Compiler Pipeline	9
2.1	Lexical Analysis (Lexing)	9
2.2	Syntax Analysis (Parsing)	10
2.3	Semantic Analysis	10
2.4	Intermediate Code Generation	10
2.5	Optimization	11
2.6	Code Generation	11
2.7	Assembly and Linking	11
3	Static Single Assignment Form	11
3.1	Compiler optimizations	13
4	The LLVM toolchain	13
4.1	LLVM Core	13
4.2	Clang	14
5	The Python Programming Language	14
5.1	The Selection of Python	15
5.2	CPython	16
6	Type systems	16
6.1	Dynamic typing	17
6.2	Static Typing	17
6.3	Gradual Typing	17

3	Design	18
1	The Type System	18
2	A Specializing Data and Execution Model	19
2.1	The Binary Representation of Type Instances	19
2.2	Type Guards	19
2.3	Type Inference	20
3	Namespaces and Scoping	21
3.1	Name Binding and Resolution	21
4	Type Shifting	22
5	Limitations	24
4	Implementation	25
1	The Runtime	25
1.1	The Internal Execution Model	25
1.2	Type Guards	26
1.3	Type Coercion	27
1.4	The Semantics of Specialized Types	27
2	Memory Management	27
2.1	Scopes and Memory	28
2.2	Local Values	28
3	An Example	28
4	Type Shifting	30
4.1	Type Inference	31
4.2	Type guards	32
4.3	An Example	32
5	Evaluation	35
1	Methodology	35
1.1	Benchmarks	35
1.2	Environment	36
1.3	Build configurations	36
2	Measurements	37
3	Benchmarks	39
3.1	Fibonacci	39
3.2	Fibonacci Recursive	41
3.3	NBODY	42
3.4	PI	46
3.5	Sieve	47
3.6	Quicksort	50
3.7	NQueens	51
4	Discussion	52
4.1	Tython	53
4.2	CPython	54
4.3	Codon	54
6	Concluding remarks	56
1	What static type information can be inferred from a Python program without type annotations?	56
2	How can we propagate static type information to reduce dynamic type checks?	57

3	How can we specialize runtime values to circumvent the object execution model?	57
4	What is the performance impact of the reduction of dynamic type checks and the specialization of runtime values?	58
5	Future work	58
	Annex	64
	A Benchmark profiler measurements	65

Abstract

Programming languages are fundamental to the use of a machine, and there are continuously new languages being devised and published to tailor to domain-specific needs, or to improve the general interaction between a programmer and a machine. Existing languages are also continuously updated and improved.

Higher-level programming languages are more expressive in use. This enables programmers to develop programs efficiently and comfortably. However, all high-level languages must be made concrete before or at runtime, and there is a lot of machinery involved with this task.

One of the common interests in the implementation of programming languages is performance optimization. This thesis designs and implements an alternative implementation of the Python programming language, by applying ahead-of-time compilation rather than interpretation, enabling a new class of performance optimizations. Additionally, we will shift Python's dynamic type system towards a more static one, reducing runtime type checking overhead. The design proposes an execution model that allows for this type shifting. The implementation of this design, Tython, leverages the LLVM compiler infrastructure to obtain general optimizations, resulting in the emergent behaviour of type shifting. The performance of the executables generated by the compiler is measured under a standard set of benchmarks.

Keywords: python, compiler, LLVM, type system, computer, science

Acknowledgements

Firstly, I would like to express my most sincere gratitude to my supervisor, dr. Peter Lammich, for his generous availability, pragmatic feedback and inspiring optimism.

I would also like to extend my sincere thanks to the other members of the graduation committee, dr.ir. Vadim Zaytsev and prof.dr.ir. Ana-Lucia Varbanescu, for reading an early draft of this thesis and providing invaluable feedback.

Lastly, I am indebted to my family and friends without whom this thesis would not have been possible. My parents and my brother have been endlessly patient and supportive through the highs and lows of this project. And my heartfelt appreciation goes out to those of you enduring my unhinged monologues at the coffee machine. In particular, I must acknowledge the unyielding support of Melina who has seen the best and the worst, and to whom I lack the words to express my gratitude.

Thank you all for making this possible.

Chapter 1

Introduction

The goal of this thesis is to propose and measure the performance impact of an alternative implementation of the Python programming language. The reference Python implementation, CPython, is an interpreter implemented in the C programming language. We propose, prototype and evaluate an ahead-of-time Python implementation. The proposed compiler specializes some expressions over scalar values. We apply a novel approach to shifting away from the runtime overhead incurred by Python's dynamic type system where static types can be inferred from source code. Lastly, we run a benchmark suite to evaluate the performance impact of this specialization and type shifting.

We introduce the concepts required to read this work as self-contained in [chapter 2](#). In [chapter 3](#) we will show how we can design a flexible type system which can benefit from general compiler optimizations to achieve type-shifting. In [chapter 4](#) we introduce Tython, an implementation of this design. Tython makes use of the powerful general optimizations in the LLVM compiler toolchain to implement type shifting. These optimizations are not designed to have anything to do with type systems, but in [Chapter 1 Section 4](#), we show how they are composed to achieve this emergent behaviour.

In [chapter 5](#) we measure the performance of the language implementation and evaluate the impact of the compiler design (specifically its specialization execution model and type-shifting). In addition to measuring the design implementation, we will also measure the performance of CPython and Codon, a static ahead-of-time compiler.

In [chapter 6](#) we answer the research questions and make some concluding remarks. Lastly, we list some suggestions for future work.

1 Motivation

The motivation for carrying out this project lies in the fact that the Python programming language is currently very popular [42], but its reference implementation CPython is notoriously slow compared to other high-level programming languages [21]. There have been many efforts to create a faster implementation of Python [2], including projects by large corporations such as Google [49] and Facebook [16]. Recently, Codon was introduced as an extensible high-performance Python ahead-of-time compiler implementation with static typing [37].

CPython 3.11 implements many performance optimizations, and is on average 25% faster than version 3.10 [10]. This includes the Specializing Adaptive Interpreter proposed in PEP 659 [38]. This is instrumentation which can replace bytecode instructions by faster, more specialized instructions when a code segment is sufficiently "hot".

The popularity of Python and the current work in optimizing CPython or creating

alternative Python implementations focusing on performance make up the motivation for this project. We aim to contribute to these recent efforts with, among other things, a novel approach to dynamic type check reduction.

2 Research questions

Python is a dynamically typed language. This means that runtime type checks are required for most expressions, introducing a continual overhead throughout the lifetime of a program.

We are particularly interested in the design of the Python reference implementation, CPython [5]. This is the language’s most popular implementation. The Python language specification does allow for type annotations, allowing users to *express* gradual or static typing, but these annotations are completely ignored by the CPython implementation. The language specification mentions that other tools could use those annotations to perform type checking on Python programs [44], and instances of stand-alone type checkers [33] and static analysis tools [15] are available. However, no such tooling is provided by the reference implementation. The Python documentation explicitly states that the specification is aimed towards static *analysis* and bespoke *runtime* type checking, listing the use of type hints for performance optimizations as a non-goal [8]. Notably, the documentation reaffirms Python’s strong commitment to dynamic typing:

“Python will remain a dynamically typed language, and there is no desire to ever make type hints mandatory, even by convention.”

In this project, we explore a novel approach to propagating implicit source-level type information for performance optimizations without foregoing this Pythonic dedication to dynamically typed source code. The propagation of available type information is applied to reduce type checks at runtime. We structure the implementation’s internal data model to enable using a selection of general program optimizations for this purpose. The inference of types and smaller number of runtime type checks is characteristic for statically typed languages. Our approach of inference and type check reduction at compile time *shifts* the dynamic type system of Python towards a more static one.

CPython is implemented as an interpreter. It can run in two configurations: as a Read-Evaluate-Print-Loop (REPL) and as an interpreter of source code files. In both cases, the source code is first compiled into a proprietary bytecode format, which is then run in an evaluation loop. This approach allows for certain efficiencies; bytecode is much easier to parse than source code (its format is much terser), and control flow graph (CFG) optimizations can be applied at the bytecode level. However, the work of translating source code into bytecode introduces overhead at the start of a program, and the runtime handling of opcodes introduces continual overhead [50].

CPython represents all data as objects, and it comprises a large object execution model to support this. The interpreter is implemented in C, and name resolution, type checking, and function application all involve many levels of pointer indirection. There is runtime performance to be gained from reducing this pointer indirection. Furthermore, the representation of all values as objects creates some overhead. It could be beneficial for performance to reduce this overhead by specializing operations on objects of scalar types.

The goal of this work is obtaining emergent type-shifting behaviour through the application of general program optimizations. This involves static type inference and type propagation. Furthermore, we apply specialization of operations on scalar values. We then wish to quantify the performance impact of this type shifting and specialization.

This leads us to the following research questions:

1. What static type information can be inferred from a Python program without type annotations?
2. How can we propagate static type information to reduce dynamic type checks?
3. How can we specialize runtime values to circumvent the object execution model?
4. What is the performance impact of the reduction of dynamic type checks and the specialization of runtime values?

3 Related Work

In this section we'll take a look at some other work that has been done in this area before. This helps us understand what's already known and what questions are still unanswered. By briefly summarizing previous studies and their findings, we can see where our research fits in and why it's important. This section also helps us decide on the best methods to use for our study based on what has worked well or hasn't been done in the past.

3.1 Codon

Codon is a static ahead-of-time Python compiler, which creates binary executables from Python source code [37]. It focuses on enabling Domain Specific Languages (DSLs) in Pythonic code. The implementation leverages LLVM as a backend, giving it access to many of LLVM's out-of-the-box general optimizations [31]. Codon also implements the proprietary Codon Intermediate Representation (CIR), which lies at a higher level of abstraction than LLVM IR. The compiler allows for domain-specific optimizations on the CIR format before lowering it to LLVM IR. The result of a CIR optimization pass is iteratively fed back into the pipeline at the type checking stage, a maneuver Codon calls *bidirectional* compilation. After all CIR-level optimizations are applied, the code is lowered to LLVM IR. Lastly, LLVM's general optimizations are applied and the compiler artifacts are emitted.

Codon applies a novel approach to bidirectional static type inference to resolve unannotated variable declarations. It is called LTS-DI, and based on Hindley-Milner (HM)-style type inference. LTS-DI handles common Python constructs by means of monomorphization (similar to C++ template instantiation [46]), localization (treating each function as an isolated type-checking unit, for which Python variable scope semantics must be restricted), and delayed instantiation. If this static inference leads to conflicts, compilation fails. This leads to many valid Python programs being rejected by Codon. Codon *must* statically map all Python types to equivalent¹ LLVM types.

The Codon compiler is similar to our proposed implementation in terms of being a static ahead-of-time compiler that leverages LLVM. However, Codon does not track any type information at runtime, making dynamic typing impossible [37]. It is also hard to tweak and evaluate the individual performance impact of its constituent parts. This makes Codon unsuitable for answering the research sub-questions.

¹This equivalence is not strictly valid according to the Python language specification [7]. For instance, Python's arbitrary-sized scalar literals are converted to 64-bit floating-point or integer primitives [37].

3.2 Mypy

Mypy is a static type checker for annotated Python programs. It allows for gradual typing, "combining the expressive power and convenience of Python with a powerful type system and compile-time type checking" [33]. Mypy is effective in reducing type-related errors in Python programs [20]. It evaluates the standard Python type annotations introduced in PEP-484 [45], but plays no role at runtime. It therefore has no effect on the runtime performance of Python programs.

3.3 Reticulated Python

Reticulated Python [47] brings gradual typing to Python. Users can annotate Python source code using Python's own syntactic type annotations, for which Reticulated Python will perform and generate the appropriate static and runtime type checks. The compiler provides a special dynamic `Any` type to indicate dynamic values. Reticulated Python accepts pure, unadorned Python code by implicitly attributing the `Any` type to non-annotated values. It rejects programs with statically detectable errors and generates runtime type checks and casts on the boundary between annotated and non-annotated code. Reticulated Python is implemented as a source-to-source translator, and the runtime type checks it generates are often followed by internal type checks in the CPython runtime (even when Reticulated's static type system has proven they are unnecessary). This means that programs generated through Reticulated Python are never faster than CPython, and show significant slowdowns of up to 10x [47] compared to CPython. Performance optimization is not a goal of Reticulated Python.

3.4 TypeScript

TypeScript is an extension of the JavaScript programming language. JavaScript is a dynamically typed language. TypeScript adds static typing to JavaScript. Interestingly, it is not a sound type system, but it can have practical use [1].

TypeScript accepts pure, unadorned JavaScript programs. In that extreme case, a program's correct typing cannot be guaranteed, but it makes the TypeScript compiler very welcoming to JavaScript developers. In this scenario, all type checks will be performed dynamically. As the programmer adds more type annotations to the program, the static type checker will be able to guarantee the correct typing of portions of the program. This is a good example of applying gradual typing.

However, since the TypeScript compiler is "simply" a transpiler to JavaScript, the JavaScript runtime engine will still perform dynamic type-checking on the entire program. This means that even though we can get a good deal of type safety from TypeScript, in the end we are doing double work and obtain no performance benefits.

There are adaptations of TypeScript which achieve soundness, such as Safe TypeScript [36] and TS* [41], which introduce runtime type checks. The cost of this soundness is the introduction of runtime performance overhead for dynamically typed code.

3.5 Cinder and Static Python

Cinder is an open-source Python 3.10 implementation developed and maintained by Meta. [16] It is oriented on high-performance and is used in production environments, such as the main Instagram webservers where the project originated. It is a fork of CPython, but offers some interesting optimizations selected to speed up the bottlenecks identified by profiling Instagram's production webservers. It introduces "immortal instances", which

allows for certain object instances to be excluded from garbage collection (GC). This is a rather significant optimization of about 5% in the Instagram use case [16, 22], because a single server instance will spawn many worker processes which are dependent on data in the parent process. Updating the reference count of an object of the parent process from a child process forces the OS to copy the entire memory page. The optimization for Instagram does introduce a performance penalty on on straight-line code. It introduces some overhead in the form of a branching statement guarding *all* GC operations on objects, to determine whether they take part in garbage collection or not.

Chapter 2

Background

In this section we will introduce and describe the background knowledge required to understand the literary environment and academic contribution of this project. It aims to be self-contained, such that, though provided, no references need to be followed to have a working understanding of the relevant topics in this work. A general understanding of core computer science concepts and basic mathematical notation is assumed.

1 Program execution

Regardless of the language of implementation, programs are written to be executed. This means taking some data as input, and transforming it into some output. To achieve this, the operations and abstractions of the programs must themselves be transformed into machine instructions. This transformation can be done in two different ways, distinguished by when this program transformation is applied. There is the method of interpretation, where the result of the transformation is immediately executed on the host machine. There is also compilation, where the result of the program transformation is stored for later execution on the host machine (or another machine of the same platform and architecture). The latter method also allows for cross-compilation, where a program is transformed to machine language of a target platform different from the host machine.

Some language implementations mix the principles of interpretation and compilation, such as the compilation of source code into an intermediate program representation for later interpretation or further compilation. This again boils down to the composition of the two fundamental strategies of program execution.

1.1 Interpretation

An interpreter for a programming language works by directly executing the instructions in the source code line-by-line, without translating the entire program into machine code upfront. The interpreter reads a statement from the source code, analyzes its meaning, and performs the specified operations on the fly. This involves parsing the code to understand its structure, performing semantic analysis to ensure the operations are valid, and then executing the operations using a combination of built-in functions and dynamic code evaluation. Interpreters typically handle tasks such as variable binding, control flow, and function calls at runtime, allowing for interactive execution and debugging. This approach contrasts with compilers, which translate the entire program into machine code before execution, offering faster execution speed but lacking the immediacy and flexibility of an interpreter.

Definition 1 *An interpreter I for language L , written in language L_o , is a program which implements a partial function: $I_L^{L_o} : (Prog^L \times D) \rightarrow D$ such that $I_L^{L_o}(Prog^L, Input) = Prog^L(Input)$ where D is the set of possible input data, $Input \in D$, and $Prog^L$ is the set of programs which can be expressed in language L [24].*

It should be noted that in Definition 1 the input data of the two programs is assumed to be the same. If this is not the case, that data must also be transformed [24] through some morphism $\phi : D \times D_o$ such that I comprises ϕ and its codomain is D_o . We have $I_L^{L_o} : (Prog^L \times D) \rightarrow D_o$ and $I_L^{L_o}(Prog^L, Input) = \phi(Prog^L(Input))$.

REPL

A Read-Evaluate-Print Loop (REPL) is an interactive mode for an interpreter which reads (a snippet of source code), evaluates (the snippet), and prints (the result if the snippet is an expression). It allows users to type in and immediately run program constructs, such as statements and expressions, without writing them to a file. This is a quick way of running code snippets in an interactive environment.

1.2 Compilation

A compiler is similar in function to an interpreter. The main difference is the time of program transformation. Where an interpreter transforms and executes a statement at runtime, a compiler does all the transformation work before a statement is ever executed.

When talking about compilers specifically, it is useful to make a distinction between the two time-separated processes of program transformation (compile-time) and program execution (runtime). In fact, we can model our computing hardware as an interpreter of machine code (it transforms instructions in memory to electrical impulses) and find that a compiler is only concerned with program transformation and is not involved in program execution. Do note that program transformation may require partial program execution, such as the evaluation of `constexpr` in a language like C++ [4].

1.3 Abstraction Reduction

We distinguish between interpretation and compilation by their time of application. We may also consider differences in the outputs of multi-stage program transformation. We look at instances where a transformation does bring the abstraction of the source program down towards machine language, but to some intermediate level of abstraction between source code and machine code. We observe that the transformation $\phi : A_i \times M, \phi = \phi' \circ \phi''$ is a composition of $\phi' : A_i \times A_j$ and $\phi'' : A_j \times M$. Implementations of this intermediate representational approach may perform optimizations and transformations on the intermediate level of abstraction A_j through closures $\mu \in M$ where $M : A_j^2$, and choose to either continue the descent to machine language directly, or store the intermediate representation in memory or on disk for later evaluation. Note that ϕ' typically performs compilation; ϕ'' may also perform compilation, but it could be an interpreter instead.

1.4 Just-in-time compilation

Just-in-time (JIT) compilation is a hybrid approach to program execution [19]. Unlike Ahead-of-Time (AoT) compilers, which translate the entire source code into machine code before execution, JIT compilers translate code at runtime, converting it into machine code on-the-fly as needed. This process involves initially compiling or interpreting suboptimal

code and identifying frequently executed parts, known as "hot spots." When a hot spot is detected, the JIT compiler compiles this portion of the code into optimized or specialized machine code, which is then executed directly by the CPU. This on-the-fly compilation allows the JIT compiler to apply aggressive optimizations based on the actual runtime behavior of the program, leading to significant performance improvements.

A key advantage of JIT compilation is its ability to adapt to the program's execution environment. By compiling code at runtime, the JIT compiler can make optimization decisions based on the current state of the system, including available hardware resources and runtime conditions. This dynamic adaptability often results in more efficient execution compared to static compilation, which relies on compile-time assumptions. Additionally, JIT compilation can leverage profile-guided optimizations, where runtime profiling information is used to inform and refine optimization strategies, further improving performance.

The warm-up phase is a critical aspect of JIT compilation, where the runtime system initially executes the program to gather profiling information and identify hot spots. During this phase, the program may run slower than an AoT-compiled counterpart. However, as the runtime system collects data on frequently executed code paths, it starts compiling these hot spots into optimized machine code. This transition from interpretation to compilation typically leads to a performance boost, as the program gradually shifts from interpreted execution to running predominantly optimized native code. The warm-up phase is essential because it allows the runtime system to make informed decisions about which parts of the code to optimize, balancing the overhead of compilation with the benefits of executing optimized machine code.

2 Common Compiler Pipeline

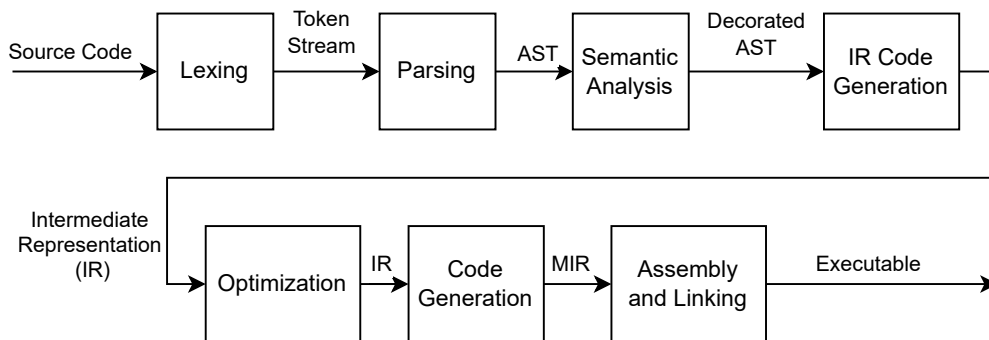


FIGURE 2.1: A common compilation pipeline.

A compiler plays a crucial role in the software development process by translating programs written in high-level programming languages into machine code that a computer's processor can execute. [Figure 2.1](#) shows a high-level overview of a common compiler pipeline [19].

2.1 Lexical Analysis (Lexing)

The compiler pipeline begins with lexical analysis, often referred to as lexing. The primary objective of this phase is to convert the raw source code into a sequence of tokens, which are the basic units of meaning in the programming language. The lexer, or lexical analyzer, reads the input source code character by character, grouping characters into tokens

based on predefined patterns. These tokens represent syntactic elements such as keywords, identifiers, literals, and operators.

For instance, the input `int x = 42;` would be tokenized into a sequence like `[INT_KEYWORD, IDENTIFIER(x), ASSIGNMENT_OP, INT_LITERAL(42), SEMICOLON]`. Each token is typically associated with a type and, in some cases, a value. The output of the lexical analysis phase is a stream of tokens that serves as the input for the next stage, syntax analysis.

2.2 Syntax Analysis (Parsing)

Following lexical analysis, the compiler moves on to syntax analysis, or parsing. The parser's goal is to construct a syntactic structure from the token stream produced by the lexer. Using a context-free grammar, the parser analyzes the sequence of tokens and builds a parse tree, which represents the syntactic structure of the source code according to the grammar rules.

Often, the parse tree is transformed into an abstract syntax tree (AST). The AST is a simplified, hierarchical representation that abstracts away some of the syntactic details present in the parse tree, focusing instead on the essential structural elements. For example, the input `int x = 42;` might yield an AST representing an assignment statement with nodes for the declaration, the variable `x`, and the literal `42`.

The parser ensures that the source code conforms to the syntactic rules of the language, and its output—a parse tree or AST—provides a structured foundation for further analysis and transformation.

2.3 Semantic Analysis

Semantic analysis follows parsing and serves to ensure that the program's constructs are semantically valid according to the language rules. This phase involves checking for type errors, resolving scope and symbol references, and verifying other semantic rules specific to the language. The semantic analyzer traverses the AST, ensuring that the types of expressions are compatible, variables are declared before use, and functions are called with the correct arguments.

For example, in the statement `int x = 42;`, the semantic analysis phase would confirm that the variable `x` is correctly declared as an integer and that the assignment is type-compatible. The semantic analyzer often annotates the AST with additional information such as types and symbol table references, enriching the tree with semantic context that aids in subsequent stages.

2.4 Intermediate Code Generation

Once semantic analysis is complete, the compiler proceeds to intermediate code generation. The goal of this phase is to translate the AST into an intermediate representation (IR). The IR is a lower-level abstraction of the source code, designed to be both machine-independent and easy to analyze and transform. Common forms of IR include three-address code and static single assignment (SSA) form.

For example, the assignment `int x = 42;` might be translated into an IR instruction like `%x = 42`. This intermediate code serves as the basis for optimization and is intended to facilitate transformations that enhance the performance and efficiency of the final executable.

2.5 Optimization

Optimization is a critical phase in the compiler pipeline where the intermediate code is improved for performance and efficiency. The optimizer performs various transformations on the IR to eliminate redundancies, improve execution speed, and reduce resource consumption. Common optimization techniques include constant folding, dead code elimination, loop unrolling, and function inlining.

During constant folding, for example, expressions with known constant values are evaluated at compile time rather than runtime. Dead code elimination removes code segments that do not affect the program's outcome, thereby streamlining the code. These optimizations ensure that the final machine code runs more efficiently on the target hardware.

2.6 Code Generation

Following optimization, the compiler enters the code generation phase, where the optimized IR is translated into machine code. This phase involves mapping high-level operations to low-level instructions specific to the target CPU architecture, allocating registers, and selecting appropriate machine instructions.

For instance, an IR instruction like `x = 42` might be translated into a machine-specific instruction such as `MOV R1, #42`. The code generator aims to produce efficient and correct machine code that closely matches the capabilities and constraints of the target hardware.

2.7 Assembly and Linking

The final stages of the compiler pipeline are assembly and linking. The assembler converts the generated machine code into object code, which is a binary representation of the machine instructions. The linker then combines multiple object files, resolves external references, and incorporates library code to produce a single executable binary. Note that when linking against dynamic libraries, those libraries are not included in the executable and form part of the executable's runtime.

During linking, the linker ensures that all function calls and variable references are correctly resolved, combining various code modules into a cohesive executable. The output of this phase is a standalone executable file that can be run on the target machine.

3 Static Single Assignment Form

Static Single Assignment (SSA) form is a property of intermediate representations (IR) in compilers. It plays a crucial role in optimizing code. The SSA form simplifies many compiler optimizations by ensuring that each variable is assigned a value exactly once, and that every variable is defined before it is used. This representation helps improve the efficiency and accuracy of program analysis and transformation.

SSA form was introduced by researchers in the late 1980s as a response to the increasing complexity of program optimization in (then) modern compilers. Certain compiler optimizations, such as constant propagation, dead code elimination, and common subexpression elimination, were challenging due to the frequent reassignments of values to the same variables in intermediate representations (IRs). The concept of Single Static Assignment evolved as a way to streamline these optimizations, making them both more efficient and easier to implement.

In SSA form, each variable is assigned exactly once, and each variable is defined at one unique point. When a variable would otherwise be assigned multiple times, SSA introduces

new "versions" of the variable for each assignment. For example, instead of assigning a value to a previously defined variable x , the compiler generates new variables x_1, x_2, \dots, x_n , for each subsequent assignment.

A key concept of SSA form is the ϕ -function (phi-function), which is placed at join nodes of a control flow graph (CFG). These are points in the code where multiple control flow paths converge. The ϕ -function merges different versions of a variable coming from different paths in the CFG. For example, Figure 2.2 shows an if-else construct in a CFG where the ϕ -function is used to select the correct variable version depending on the path the program takes at runtime.

Def-use chains are also simplified by SSA form. A def-use chain relates the definition of a variable with the set of its uses. In SSA form, this information is combined efficiently because the uses of a variable are generally close to their definition; they are replaced by a new version upon assignment or at CFG joins. Use-def chains relate the use of a variable to their definition, and are fairly trivial to compute in SSA form. The use-def chain is implicitly encoded and maintained by the edges of an SSA graph – this is how the SSA graph is constructed.

Converting a program into SSA form involves two main steps: the renaming of variables and the insertion of ϕ -functions. During renaming, each assignment to a variable creates a new version of that variable. This ensures that each variable is assigned exactly once. Inserting ϕ -functions involves identifying points in the CFG where variables from different paths merge and introducing ϕ -functions at the start of these joins to select the correct version.

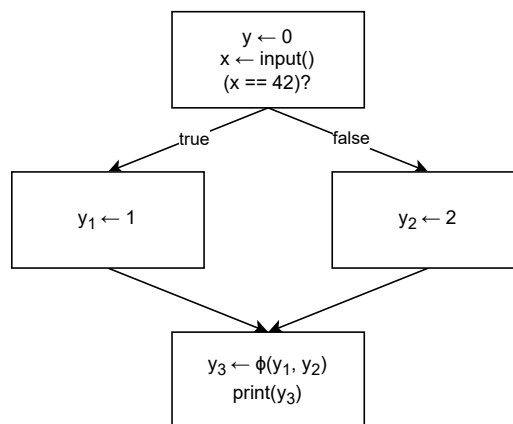


FIGURE 2.2: The phi-function inserted at a CFG junction, adapted from [34].

To generate executable code, SSA form must usually be converted back to a non-SSA form at some stage. This means eliminating ϕ -functions (replacing them with some runtime machinery) and merging the different variable versions. The conversion must ensure that the resulting code remains semantically equivalent to the SSA form. This process is called destruction.

The downside of SSA form is that it introduces a additional complexity in handling variables that range across different scopes, as is the case with loops and other nested constructs. Techniques like minimal SSA form and pruned SSA form have been developed to address these challenges [34].

3.1 Compiler optimizations

SSA significantly simplifies many compiler optimization techniques.

Constant Propagation SSA form significantly enhances the process of constant propagation. This compiler optimization technique substitutes the values of known constants in expressions. In SSA form, each variable is assigned exactly once, and the use-def chain relates every use of a variable to a single unique definition. This greatly simplifies the tracking of constant values throughout the program.

Dead Code Elimination SSA form makes it easier to identify and eliminate variables that are never used: since each variable is assigned exactly once, the corresponding assignment can be safely removed if that variable has no uses.

Register Allocation SSA simplifies the task of register allocation by reducing the complexity of variable lifetimes. In Static Single Assignment (SSA) form, each variable is assigned exactly once, which makes the flow of values through variables more explicit and simplifies data flow analysis. This clear assignment is beneficial for register allocation in compilers. To allocate registers efficiently, an interference graph is constructed where each node represents a variable, and an edge between two nodes indicates that the corresponding variables are "live" simultaneously and therefore cannot share the same register.

Register allocation is then treated as a graph coloring problem, where each "color" represents a register. Under SSA, the strictness property of a procedure is equivalent to the dominance property. A dominance analysis (resulting in a dominance tree) on a program in SSA form, a liveness analysis, and an interference graph (which is an intersection in live ranges), is a chordal graph [34]. This is important to the problem of register allocation, since chordal graphs have linear-time solutions for graph coloring. In general graphs, this problem is NP-complete.

4 The LLVM toolchain

LLVM started as a research compiler toolset at the University of Illinois Urbana-Champaign, for experimentation with static and dynamic compilation techniques for any programming language [32]. Since then, it has grown into an umbrella project which comprises a number of different subprojects, successful in both commercial and open-source production systems as well as academic research. LLVM's most notable projects are briefly introduced below.

4.1 LLVM Core

LLVM Core [30] revolves around the LLVM Intermediate Representation (IR) format. It comprises the development APIs for bespoke IR generation. The project includes IR-level code optimizers which work independently from source languages and target platforms.

Figure 2.3 shows a high-level overview of the LLVM Core compilation pipeline. The pipeline translates source code of an arbitrary programming language into machine instructions of an arbitrary platform. The pipeline is split up into two discrete stages. There is the frontend, which translates concrete source code into the general intermediate representation (IR) format. There are optimizers and code analysis tools which can transform this IR. Lastly, this IR is translated into Machine IR (MIR) by a target-specific backend.

This frontend/backend split means that developing a frontend for a concrete language makes it immediately available to all existing platforms. Similarly, developing a backend for a concrete target makes all existing frontends immediately available to that specific platform.

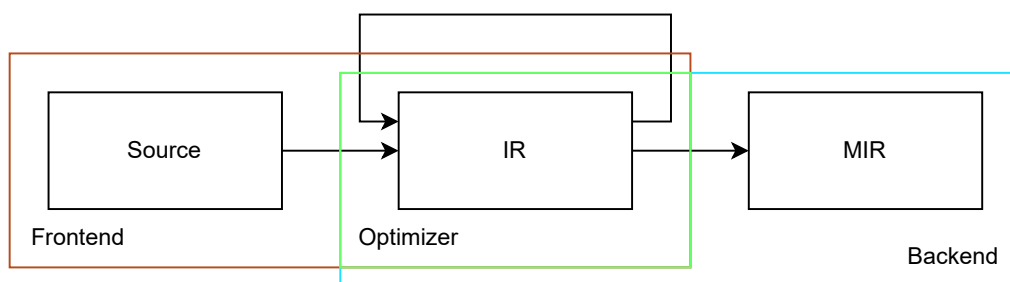


FIGURE 2.3: A high-level overview of the LLVM Core compilation pipeline

Another strength of this approach is that IR-level optimizations and analysis can be applied independently of source language or target platform. This makes general optimizations such as constant propagation, dead code elimination, and loop unrolling [31] available to all source languages and target platforms¹.

There is no common structure imposed on the implementations of LLVM frontends or backends. As long as they respectively produce and consume valid LLVM IR, they can be used in the LLVM pipeline and benefit from the source language and platform agnostic optimizations at the IR level. However, LLVM Core does expose an actively maintained C++ API for implementing both frontends and backends.

Frontends will essentially need to perform all steps of traditional compilation (see [Chapter 2 Section 1.2](#)), with the key difference being that instead of emitting machine code directly, they emit LLVM IR.

4.2 Clang

Clang [3] is an LLVM frontend implementing C-like languages such as C, C++, Objective-C, and OpenCL. It aims to be fast and have a low memory profile. It features several clients, such as static analysis and refactoring, and provides expressive diagnostics. It is one of the most popular production-ready C compilers available today [17].

5 The Python Programming Language

Python is a general-purpose programming language, originally designed and developed at the Centrum Wiskunde & Informatica in the late 1980's [43] by Guido van Rossum. There have been three major releases of Python, the most recent version being Python 3.12.

Python is one of the most popular general-purpose programming languages in the world today [42]. The use cases of the language span from writing short-lived scripts (similar to how one might use BASH scripts), to large-scale production applications [16, 18]. It has a notable role in the development and implementation of machine learning and AI systems. The design philosophy of Python focuses on the readability of source code.

The language is a multi-paradigm language, supporting imperative and functional programming styles. It can natively conveniently express object-oriented programming. It is dynamically typed. It features *duck typing*; "if it quacks like a duck and wags like a duck, it's probably a duck". Academically, this practise is called structural typing (see [Chapter](#)

¹This does not mean that a general optimization is always effective on all source languages. For instance, a programming language with no loop semantics may not benefit from loop unrolling.

2 Section 6). Together with the dynamic typing of variables, this creates Python’s late binding access operators (including function name resolution).

5.1 The Selection of Python

The selection of Python for this project was made based on a few simple inclusion criteria. The research goal is set to measure the performance impact of type-shifting a dynamic language using LLVM’s general optimizations. We choose an existing language to experiment on, to avoid contriving a language specification which is tailored for this purpose. The language must be dynamically typed. In an attempt to achieve maximum impact, the language should also be general-purpose and widely used in practise. Lastly, there should be an authoritative open-source reference implementation, so that that can be used as the foundation to experiment on, or so that a bespoke minimal implementation could be based on this reference. Keeping the foundation for the experimental setup close to the reference implementation creates a claim to comparability.

Python is the one language which fits all of these criteria the closest. It is dynamically typed and general-purpose. According to the Tiobe Index, is currently the most popular programming language in the world [42]. The Python organization also maintains a reference implementation named CPython [6], which is developed in tandem with the language specification. It is by far the most widely used Python implementation [44].











Jun 2024	Jun 2023	Change	Programming Language	Ratings	Change
1	1		 Python	15.39%	+2.93%
2	3	▲	 C++	10.03%	-1.33%
3	2	▼	 C	9.23%	-3.14%
4	4		 Java	8.40%	-2.88%
5	5		 C#	6.65%	-0.06%
6	7	▲	 JavaScript	3.32%	+0.51%
7	14	▲▲	 Go	1.93%	+0.93%
8	9	▲	 SQL	1.75%	+0.28%
9	6	▼	 Visual Basic	1.66%	-1.67%
10	15	▲	 Fortran	1.53%	+0.53%

FIGURE 2.4: The 10 most popular programming languages today (www.tiobe.com) [42].

Figure 2.4 shows the most popular languages in use today. Python is the only² dynamically typed language in the top-5. In the top-10 we also find the dynamically typed and immensely popular JavaScript, though its ubiquity is still mostly limited to the domain of webtechnologies, where it has a de-facto monopoly. This arguably makes it less general-purpose than Python. JavaScript also lacks a reference implementation.

²C# does feature the `dynamic` keyword, allowing for the type of individual variables to be determined at runtime. This is effectively a syntactical alternative to the verbosity of hiding from the language’s default static type system through object type casting.

5.2 CPython

CPython [6] is the Python organization's reference implementation. It can be considered both a compiler and an interpreter, since it can compile Python source code into a proprietary bytecode format Ahead-of-Time, which is then interpreted at runtime.

Python's data model considers all data to be of the object type. Even constants and literals are subclasses of this top-level type. Classes, functions, methods, numbers, strings, and types – all inherit from object. CPython is implemented in C. The C language does not support polymorphism, instead providing some basic data types from which a (library) programmer can create abstractions. The way the CPython programmers overcome this is by taking care that all Python objects start with identical memory layouts. The first n bytes describe the same fields for all objects. These fields are the object's unique identifier, a reference counter, and a reference to its type. Such a type is an object by itself, whose type field refers to the "type" type. Type objects provide further information about the properties and capabilities of the objects that reference them.

Since types are implemented as objects in CPython, they contain a pointer to a type. For types, the reference type is a singleton type instance called "type", which is an object by itself. This singleton instance is special, in that its type pointer points to itself (that is, the type is its own definition).

An object's type can never change in Python. This is according to the language specification. A consequence of this is that CPython can simply allocate enough memory to hold any instance of the type (which is lower-bound by `sizeof(T)`) and initialize it, which is also a discrete step in the Python language design. The system keeps track of a pointer to that memory location. The interpreter can at any point in runtime rediscover the size of an object by checking its type. It can cast the pointer to a pointer to an instance of the actual object type, thus giving the implementation access to all extended fields.

The CPython Type System

CPython implements Python's dynamic "duck" typesystem. Even though the language specification does allow for type hints in its syntax, CPython does not process any type annotations. CPython implements a type system comprising the runtime evaluation of which operations are legal and implemented on a given object type, what type conversions are allowed, and the concepts of (runtime) class inheritance. It is late-binding, resolving names to variables, functions and class methods at runtime. It implements hash-tables and caching to improve the performance of these dynamic resolutions.

6 Type systems

Type systems allow a machine (an interpreter or a compiler) to determine the properties of variables and values in a program. Mitchell [26] defines types as collections, and values as members of those collections.

The properties of a value fundamentally determine under what operations they have well-defined semantics. For instance, an addition operation over a floating point value and a string value may not have well-defined semantics in the language. A type checker may therefore choose to reject such operations. The property of a system rejecting operations with undefined semantics on values of a given type is known as type safety.

We distinguish two classes of type systems, distinct in the time at which they perform type checking. Note that this distinction is similar to that between compilation and interpretation.

6.1 Dynamic typing

Dynamically typed languages perform type safety checks at runtime. Implementations keep track of a *value*'s type tag. In this way, assigning a value to a named variable can dynamically change the type associated with that variable at runtime.

Dynamic typing decouples the concept of a named variable (a labeled reference, changeable — *variable*) from the concept of types and values. Language implementations may or may not perform any static type checking on dynamic languages, but must in any case evaluate the type associated with a variable at runtime.

6.2 Static Typing

Statically typed languages associate a type with a variable. The type of a variable is determined at its declaration, and only values of that type can be assigned to that variable. This means that, without running the program, we can check whether operations (such as assignment) on a *variable*, and operations on any value associated with that variable conform to its type.

This does not mean that the evaluation of types in these languages are restricted only to compile-time. In particular, object-oriented languages that support downcasting, such as Java, may require runtime type checking to determine if a downcast is valid.

Statically typed languages often allow for generalization over types (generic typing), providing similar flexibility to dynamically typed languages in terms of value assignment. This, again, is particularly prevalent in object-oriented languages such as Java and C#, where concepts such as inheritance, downcasting, and type erasure play an important role.

6.3 Gradual Typing

In gradual type systems, programmers can express aspects of both static typing and dynamic typing in the same codebase. The type checker is then tasked with finding and rejecting incompatibilities between the known parts of a type [39]. This approach involves type consistency in the static semantics of the gradual system, and runtime type casts in the dynamic semantics. In this hybrid approach, the programmer can leverage the flexibility of dynamic typing where desired, while maintaining the safety (and possibly the performance benefits) of static typing.

Gradual type systems are often introduced to incrementally change an existing dynamically typed code base to a statically typed one. Such gradual type systems accept the entire existing dynamic code base, statically checking more and more annotated regions of code as they are added by the project maintainers. The recent popularity of TypeScript is an example of this [48].

Chapter 3

Design

In this chapter we introduce the idea of performance optimization through type-dependent operator specialization and a reduction of runtime type checking through a novel approach to source language-agnostic type inference through the common constant propagation and branch elimination compiler optimizations.

The compiler design is based on a few simple mantras:

1. A user should be able to write Pythonic code.
2. Semantics are second to performance.
3. The reference implementation is always right (even when it is not).

1 The Type System

The distinguishing feature of a dynamic language is runtime type checking. To achieve this, we need to have type information available at runtime.

In this type system, a $Variable : Name \times Value$ is an ordered pair relating a name to a value. Names are unique labels corresponding to symbols in source code. Values are also ordered pairs, defined as $Value : T \times U$. The set $T := Prim \cup \{Object\}$ is a finite enumerated set, and is the union of the set of primitive types $Prim := \{Integer, Floating-Point\}$, and a singleton set containing the built-in object types;

$Object := \{IntegerObject, FloatObject, StringObject, ListObject, TupleObject, DictionaryObject, etc.\}$. U is the universal set of bit-strings, representing contiguous (virtual) computer memory. The type of a variable can be evaluated with the function $type : Variable \mapsto T$.

Types have a set of values they can define, which is determined by the relation $I := U \times T$, such that cIt iff $c \in U$ is an instance of $t \in T$. Inversely, cIt iff $t \in T$ can define $c \in U$. Assuming the bitwidth l of instances of all elements in T is consistent, we can restrict U to length l , such that $U_l := \{c \in U. |c| = l\}$ ¹, $U_l \subset U$.

The unparameterized use of bit-strings makes membership of I trivial for any bit-string and type: $\forall c \in U_l, t \in T. cIt$. This also allows for arbitrary type conversions between equal-length byte strings (the bits of a 64-bit float can also define the bits of a 64-bit integer). To avoid semantically meaningless interpretation of bit-strings and implicit conversion, we create an inequivalence between two values of the same content but with different types:

¹Fixing l to 64 conveniently gives us all bit-strings that can fit in a 64-bit CPU register.

Definition 2 *Two values are equivalent iff their types and their content are equivalent:*
 $\forall v, v' \in Value. type(v) = type(v') \wedge content(v) = content(v') \iff v = v'$.

Note that to avoid arbitrary equivalence of two values under a fixed l , it is enough to show that $type(v) \neq type(v')$. The interpretation of a value under a given type is defined as the function $\Upsilon : (T \times T \times U) \mapsto U$. Similarly, we define the restricted function $\Upsilon|_{Value} : (T \times Value) \mapsto U_l$. Υ allows us to define the semantics of (implicit) type conversion.

2 A Specializing Data and Execution Model

We propose a specialization strategy for two data types, the *Integer* and the *Floating-Point*. We define the specialization data structure as an ordered pair, comprising a tag and a content field; $Spec := Tag \times U$. The interpretation of the content field depends on the value of the tag in the tuple. The tag field is an enumerator, defined as being one of $Tag := \{0, 1, 2\}$ where $0 \mapsto Integer$, $1 \mapsto FloatingPoint$, and $2 \mapsto Object$. The tag of a value $v \in Spec$ can be obtained through the 0th projection map $\tau : Spec \mapsto Tag = proj_0$. The content of a value can be obtained through the 1st projection map $\mu : Spec \mapsto U = proj_1$.

2.1 The Binary Representation of Type Instances

The tag field determines the semantics of the interpretation of a value's content field, with a function $\pi : Tag \times Spec \mapsto U$. What the function π allows us to do, is to encode the specialization of low-level instructions that expect a certain bit-layout for their operands. This bit-layout goes beyond the traditional type system of some high-level programming languages, but is necessary for the correct concretization of high-level semantics such as arithmetic operations. A platform's ISA does not maintain or check type information, instead accepting any operands of the right bit-width for any instruction. For example, the machine instruction for the sum of two operands will always succeed, but that result may not be semantically meaningful if one operand is laid out as a floating-point value in mantissa format, and the other as an integer in the two's complement format. The concretization of high-level semantics for such low-level operations is encoded in π .

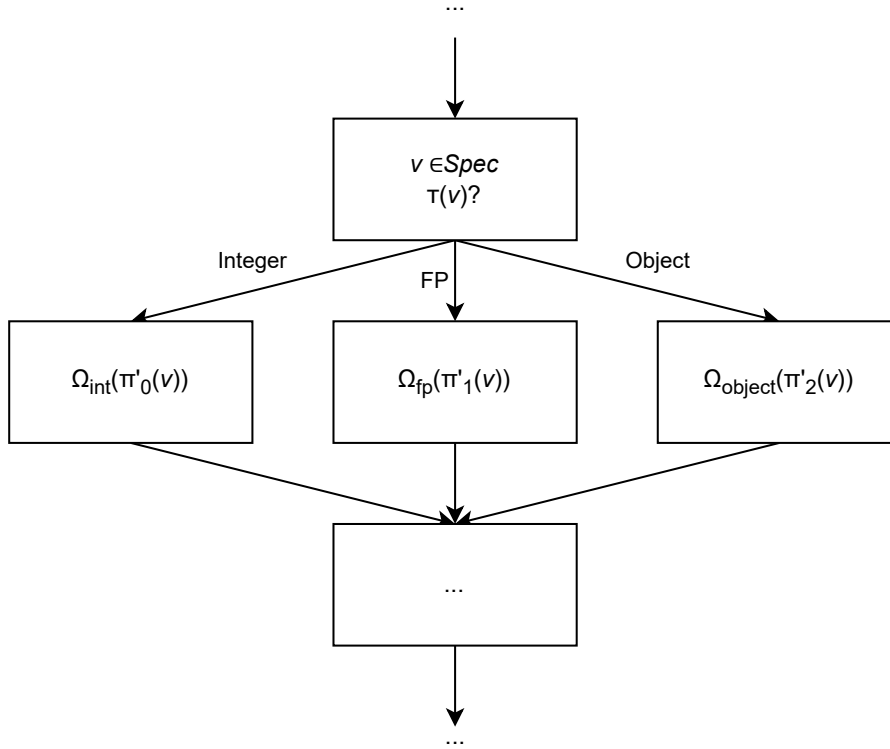
2.2 Type Guards

Type guards ensure that all read operations on a value v use the correct binary interpretation of that value. They are nodes in the control flow graph (CFG) with an outgoing edge for each type $t \in Tag$, branching on $\tau(v)$. [Figure 3.1](#) shows how a specializing use Ω of value v is guarded.

This uses three partial applications of the function π ; one for each of the possible values of the tag $\tau(v)$. The resulting functions $\pi'_i : Spec \mapsto U$ interpret the value and are input to the specialized versions of Ω .

Note that these type guards are evaluated at runtime, preserving the dynamic behaviour of the Python programming language and Tython's first mantra:

“A user should be able to write Pythonic code.”

FIGURE 3.1: The guarded specialization of an operation Ω on value v .

2.3 Type Inference

There are several methods of type inference, using such techniques as principal typing (such as Algorithm W described in the Hindley-Milner type system [25]), and the derivation of coercions as applied by Rastogi et al. [35].

And although Python syntactically supports type annotations, the Python organization considers them very un-Pythonic [8]. In keeping with Tython's mantras, we will not currently consider the use of this syntactical language feature. Without type annotations, we can only attempt to infer expression types from observing the static properties of values in source code. We can then use that static information to infer the type of a variable.

Our type system only handles the three discrete types $\{Integer, FloatingPoint, Object\}$, and there is no polymorphic relationship between these types. We can therefore see the inference of the type of a variable in our context (Definition 3) as similar to Rastogi et al. [35], looking only at the inflows of values into a (type) variable.

Definition 3 $type(\alpha) = t$ iff $\forall w \in W. \tau(w) = t$, where $W \subseteq Spec$ is the set of operands corresponding to all write operations to the variable α .

This definition does preserve the requirement of locality. If all write operations to a variable α are known at compile-time, and the type associated with all those write operations are known at compile-time to be some type t , then we can statically infer that $type(\alpha) = t$.

3 Namespaces and Scoping

Scopes are sets of variables which describe the definitions a variables in a program. We say a variable v is defined in a scope s iff $v \in s$.

We define a transitive, asymmetric relation E , where a sEp for any scopes s and p iff s is nested in p . This gives us a hierarchical structure with which we can define the semantics of variable visibility ([Definition 4](#)).

Definition 4 *A variable v is visible in the scope of s ($visible(v, s)$) iff $v \in s \vee (\exists s'. sEs' \wedge v \in s')$.*

The converse intuition also holds;

Definition 5 *A variable v is not visible in a scope p ($\neg visible(v, p)$) if it is defined in a child scope s ; $v \in s$, sEp .*

Namespaces are the set of variables visible from a given scope; $N : Scope \times Name \times Value$. For a given scope s , a namespace is defined by the parameterized set $N_s := v \in Variable.visible(v, s)$. A namespace can be queried with the function $find_s : (Name \times N_s) \mapsto Value$ to resolve the value related to a name from a given scope s .

The concept of variable shadowing is defined as such: if a name v is defined in more than one scope visible from a given scope s , the *nearest* definition is selected [44]. The nearest definition of a name v from s is found by following the transitivity of the nesting relation E up the hierarchy, and yielding the first scope s' , $s' = s \vee sEs'$ where v is defined. This allows for variable shadowing, as illustrated in [Listing 3.2](#), where the name x is defined both as an argument to the function foo and as a variable in global scope. Name resolution correlates the expression x at line 4 to $x \in s_{foo}$ because that definition is *nearer* than the definition $x \in s_g$.

3.1 Name Binding and Resolution

We propose to bind names statically, a principle known as early binding. This requires the function $find_s$ to be implemented at compile-time. This is in contrast to CPython's late binding strategy. The benefit of early binding is that, when it can be applied, it brings name resolution to compile time. In late binding, name resolution must happen at runtime through a namespace search. In CPython, namespaces are implemented as objects containing a hash table. The table's key is the hash of the name to resolve. In Tython, no such search is necessary for variable names or global functions. Variable names are statically bound in the code generator and subsequently encoded in SSA form, binding them at the IR level. This effectively results in the aliasing of registers for specialization structures which the LLVM backend allocates to registers. For values which are pushed to the stack, the name binds to a pointer to a specialization structure. See [Chapter 3 Section 2](#) for a description of the data model.

Tython informs the user at compile time, before any executable code is emitted, of illegal name bindings. This fail-fast behaviour is safer than the `NameError` CPython throws at runtime. Being a late binding interpreter, CPython is only aware of a name not existing at any place in a program when it has exhaustively searched all context-relevant namespaces at runtime.

CPython implements fast hash tables for name resolution in an effort to minimize the performance impact of these very common operations. However, regardless of how fast

these algorithms perform, they still introduce a performance overhead for every use of a variable or function name.

Tython still relies on late binding for object properties (and collection objects which can be indexed by a hashable key such as a string, even if the string is a literal – more on this in [Chapter 3 Section 5](#)). The effects are similar to those in CPython; name errors are thrown at runtime, and there is a performance overhead for the use of any late binding name. For early binding names, however, Tython provides a safer compile-time error and does not generate the overhead for runtime name resolution.

[Listing 3.1](#) shows an example of a name error. The lexical scope of y is limited to the `if`-block it is defined in. Let the scope of the `if`-block the variable y is defined in be s_{if} , and let the global scope be s_g . Having $y \in s_{if}$ and [Definition 5](#), then $s_{if}Es_g \implies \neg visible(v, s_g)$.

```

1 x = input()
2
3 if x == "42":
4     y = 1
5
6 #CPython executes all code
7 before the name error...
8 print("Hello_world!")
9
10 #... failing only when a name
11 cannot be found in the
12 relevant namespaces
13 print(y)
```

LISTING 3.1: An illegal use of the reference y , which is undefined at global scope.

```

1 x = input()
2
3 if x == "42":
4     x = 24
5
6 print(x) #prints 24
```

LISTING 3.3: Assignment to a visible name is never a redefinition in Python.

Note that, under Python language semantics, line 4 in [Listing 3.3](#) does not define a new name x in the scope s_{if} . This is because, lacking a keyword for the purpose, name definition is implicit in Python through the assignment of a value to an fresh name. Certain other constructs, such as the function name and argument x shown in [Listing 3.2](#) are also defining constructs [44]. There are certain surprising consequences to the binding semantics of Python, such as the `UnboundLocalError` and the existence of the `global` keyword [9].

4 Type Shifting

Type shifting is the emergence of local static typing in a dynamically typed program. This is similar to gradual typing, but requires no type information or let-style binding to be expressed by the programmer. The source code of a Python program can entirely

```

1 x = 42
2
3 def foo(x):
4     print(x)
5
6 foo(24) #prints 24
```

LISTING 3.2: The shadowing of a name in a nested scope.

dynamically typed, and yet we can still locally obtain some of the benefits of static typing. Most notably, we can reduce the number of runtime type checks if the type of a variable is known at compile time.

Early-binding gives the compiler direct access to all write operations W on a variable α . When all types of these write operations can be statically determined to be the same, the type of α is trivial by [Definition 3](#). If the type of α is known at compile time, we can remove the unused branches of the type guards generated for all uses of α .

[Figure 3.2](#) shows the CFG of the application of a specializing operation Ω over a variable α . Early-binding ensures the runtime branching condition is $\tau(v)$ (the type of the value associated with α) instead of $type(\alpha)$. [Figure 3.3](#) shows the result of type inference on that operand. $\tau(v)$ is statically known and can be replaced with the constant enumerator *Integer*. [Figure 3.4](#) shows how comparison of constants as a branching condition allows for trivial edges to be pruned. Because there is branch for each tag in *Tag*, upon resolution of $\tau(v)$ there is always exactly one outgoing branch remaining. Since the CFG nodes for specializations of Ω had been generated for this unique use of α , they have no other incoming edges. In [Figure 3.5](#), the unreachable CFG nodes are deleted and the straight-line nodes are merged.

This turned a dynamically expressed use of a variable α in source code into a static use in the generated code.

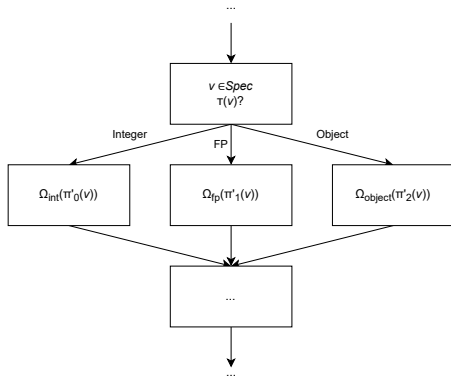


FIGURE 3.2: The CFG for the specializing application of Ω over a value v .

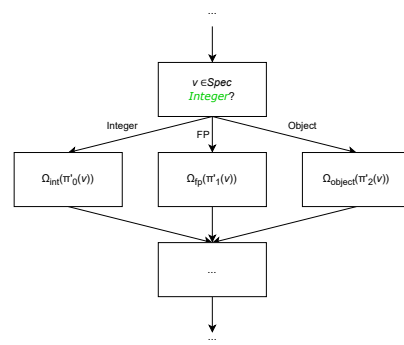


FIGURE 3.3: Step 1: type inference.

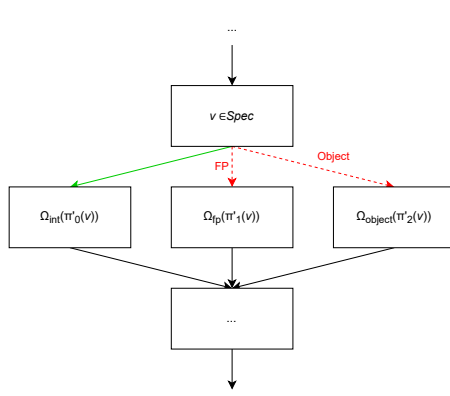


FIGURE 3.4: Step 2: prune trivial conditional branching.

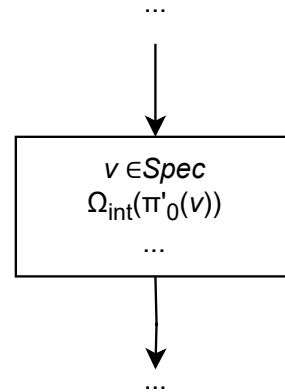


FIGURE 3.5: Step 3: delete unreachable nodes and merge straight-line nodes.

5 Limitations

There are some limitations associated with the design of Tython. Regarding name binding and resolution, there are many language constructs which do not yet benefit from static binding. Notably, the access of built-in object functions (for instance, the method call `list.append(e)`) is late binding. This means that the name of the object function is resolved at runtime through a linear search over an array of key-value pairs maintained by the object's type object. This is less efficient than CPython's name resolution through fast hash tables.

Another limitation of the design is the extensibility of the set *Tag*. As new types are added to the set, an increasing number of CFG nodes is generated. Any implementation that extends *Tag* should take care that the overhead introduced by an increased number of generated CFG nodes for all read uses of a value is not counterproductive towards the goal of performance improvements.

Chapter 4

Implementation

In this chapter, we introduce Tython [27]. Tython is a Python compiler, implementing the design described in [chapter 3](#). It supports a minimum set of key language features, chosen for their relevance to running a selection of benchmarks (see [Chapter 4 Section 1.1](#)). A part of the implementation is inspired by CPython, but distinguishes itself by being an ahead-of-time compiler instead of an interpreter. Fundamentally, it leverages a similar object data model to CPython. However, Tython aggressively applies specialization strategies to integer and floating-point operations, and applies LLVM’s general optimizations to statically reduce the dynamic type checks into specialized straight-line code at compile-time.

The compiler is implemented in C++, using the ANTLR parser-generator for lexing and parsing, and the LLVM Core API for code generation and optimization. It uses Clang as an intelligent linker (with link-time optimizations), and can create executables for most LLVM target platforms¹.

1 The Runtime

In this section we will describe how Tython’s data and execution models are implemented. We describe the internal representations of runtime values, how expressions are evaluated, and how control flow is handled.

Tython’s runtime comprises a dominant internal execution model and a supporting object execution model. We say the internal execution model is dominant because it is in charge of program control flow, and Tython’s performance-oriented design prefers all program data and expressions to be represented and executed in this part of the runtime. Control is only yielded to a routine in the object execution model when necessary. The internal execution model regains control when the object routine exits². Between the execution models, there is bridging machinery with the responsibility of marshalling operands to and from the specialization and object data representations.

1.1 The Internal Execution Model

The internal execution model handles the structure and the control flow of the source program. Tython implements this using the LLVM intermediate representation (IR). During compilation, this is an SSA-form representation of the source program with its own

¹The most efficient target platforms are 64-bit architectures, due to the assumption of 64-bit address width in the specialization data structure.

²There is no concurrency relation between the execution models.

low-level type system. The internal execution model is responsible for (initiating) the instantiation of all source-level data, such as Python literals.

It has a data model which allows for the specialization of uses of values with the scalar types *Prim* defined in [Chapter 4 Section 1](#). For the use all other types, namely the *Object* types, it delegates control to the object execution model.

The Internal Data Model

The data model *Spec* enabling the specialization of scalar types is implemented in Tython as a tagged union. The memory layout of values is shown in [Table 4.1](#).

Field	Type	Size (bytes)	Offset (bytes)
tag	integer	4	0x0
content	union	8	0x4

TABLE 4.1: The data layout of the specialization tagged union.

The data structure has a representation in the runtime library, where it is implemented as a `struct` with the name `specialization_t`. The definition of the C struct in [Listing 4.1](#) serves as a convenient notation to discuss the union type of the content field. Lines 4-6 show the definition of three 8-byte wide types³.

```

1 typedef struct specialization {
2     uint32_t tag;
3     union {
4         long long integer;
5         double floating_point;
6         struct object_t* object;
7     };
8 } specialization_t;

```

LISTING 4.1: The definition of `specialization_t` in the runtime library.

All values in Tython are of this form. Specialization values are contained directly as a field in the data structure, and object values are contained as a pointer to the heap memory location where the object is located.

1.2 Type Guards

To create a type guard, Tython generates runtime code to first extract the tag field of a given value v with the IR instruction `extractvalue %Spec %v, 0`, implementing the function τ . Then a branching instruction on $\tau(v)$ is generated with four targets: a specialized handler block for the *Integer* and *FloatingPoint* types, a dynamic handler for the *Object* type (delegating to the object execution model), and an exception handler for encountering an unexpected tag value $t \notin Tag$. ?? shows a Control Flow Graph (CFG) of the generated branching.

Recall [Figure 3.1](#), where a specializing operation Ω on a value v is guarded. The partial applications of π are implemented using a `bitcast` instruction on $\mu(v)$ for π_1 , and an `inttoptr` instruction on $\mu(v)$ for π_2 . μ , similarly to τ is implemented with the IR instruction `extractvalue %Spec %v, 1`. Given the implementation of the content field as

³Assuming a 64-bit platform on which `sizeof(void*)== 8`.

an `i64` in the data layout of specialization values in LLVM IR, π'_0 is equivalent to μ : there is no bitcast required to interpret an `i64` as an `i64`.

Thanks to LLVM IR's low-level type system, the implementation of concretization ends at the IR level for Tython. Tython applies π to generate IR-typed instructions which the LLVM backend will further concretize for a specified target platform.

1.3 Type Coercion

The Python language specification describes the semantics of type coercion under certain operations [44]. For infix binary operations, it specifies that the type of the left-hand operand (LHS) is leading, and that the right-hand operand (*RHS*) must be coerced to the type of *LHS*.

Infix binary operations in the object execution model are syntactic sugar for the invocation of magic methods. Type coercion is handled in these magic methods at runtime and, having resolved the type of LHS, follows the resolution of the type of RHS. The magic method belongin to the type object of LHS is responsible for the conversion of RHS, or throwing a type error.

Tython implements this with a nesting of type guards. For each specialized handler of a value v , an inline type-guarded conversion of RHS is generated.

1.4 The Semantics of Specialized Types

Both specialized types are a more efficient counterpart to their object implementation in the runtime library's data model. They are implemented as finite-length bit-strings, tailored to a target register size, which allows for the application of single machine instructions as a concretization of most of the source-level infix binary expressions⁴. This single machine instruction is by definition faster than the object implementation; the object implementation results, eventually, in at least one equivalent machine instruction. In practise, the dereferencing of several indirections and type checks on objects is an order of magnitude slower than a single (arithmetic) instruction.

Integer objects in Python should allow for arbitrary length. However, Tython will aggressively specialize on this type if it has the chance, diverging from the language specification. The Python language specification specifies that `floats` are implemented in double precision. Tython's *FloatingPoint* specialization type matches this exactly.

Recall one of Tython's mantras:

"Semantics are second to performance."

2 Memory Management

Tython has some infrastructure for garbage collection of objects through reference counting, but it is currently not fully implemented. This means that most objects allocated at runtime are not cleaned up before the program exists. This leaky behaviour is far from desirable, but a necessary consequence of project time constraints.

All Tython objects live on the heap. This is similar to CPython's implementation of memory allocation. CPython also employs reference counting as a mechanism for deterministic garbage collection. One of the general shortcomings of reference counting is cyclical references. CPython solves this with a cycle breaking algorithm. Tython does not

⁴The most notable exception being the exponentiation operator `**`. Exponentiation does not have a corresponding LLVM IR instruction.

have a mechanism to resolve the issue of cyclical references. This, again, due to project time constraints.

The lacking of features in memory management are justified by the focus of the project on specialization and type shifting, constrained by the project timeline. Heap memory allocation can have a large impact on performance, and so a compiler generating leaky code is not optimal. However, in [chapter 5](#) we are most interested in the relative performance of unoptimized (leaky) executables versus optimized (leaky) executables, to measure the impact of specialization and type shifting on runtime type guards.

2.1 Scopes and Memory

The visibility of variables leads to an interesting interplay between scoping and memory management: a variable v defined in scope s semantically cannot have any uses in any parent p of s . If no uses are possible outside of s , we may yield all memory maintained uniquely for values associated with the variables in s . The memory associated with values which are referenced elsewhere may not be yielded.

In the case of reference counting as employed by Tython, this simply means that the reference counters of all heap objects are decremented by one at the end of their lexical scope. Local objects are never created, but stack allocations of values does occur. However, these never outlive their stackframe (see [Chapter 4 Section 2](#)).

2.2 Local Values

Another feature of Tython is the specialization of certain data types (see [Chapter 4 Section 2](#)). This allows for integer and floating point values to exist on the stack or in registers. Memory management of register values in an ahead-of-time compiler takes the form of register allocation at runtime. This is performed over LLVM IR in SSA form by the LLVM backend. In this setting, the problem of register allocation is akin to the graph coloring problem over the interference graph as described in [Chapter 4 Section 3](#). For stack values, memory allocation is not mirrored by a later deallocation in the same way that heap allocation is. Stack values live in a stack frame. This stack frame is supplied by the operating system at runtime. It is allocated upon entering a function call, and released after the call returns. This means that all "garbage" memory on the stack is reclaimed by the operating system automatically.

Tython's frontend does not allocate values on the stack explicitly. An LLVM backend may choose to lift a register value into stack memory when the register pressure is too high and not all interfering values can fit into the target's finite registers.

3 An Example

We will contrast the object execution model with the specialized execution model through the example of the addition of two objects. In Python, this can be expressed compactly in infix notation: `lhs + rhs`. This single Pythonic expression, however, gets expanded into an algorithm with non-trivial control flow, as shown in [Listing 4.2](#) and with the concrete continuation of the example for integer objects in [Listing 4.3](#).

```
1 1. Load LHS type object
2 2. Check if the type supports number operations
3   2.1. True: Load LHS number operations
4   2.2. False: Throw TypeError
5 3. Check if the number operations support addition
```

```

6   3.1. True: Load the addition function pointer from LHS number
      operations
7   3.2: False: Throw TypeError
8  4. Invoke the binary addition function

```

LISTING 4.2: The algorithm for applying addition in the object execution model.

Listing 4.3 shows the object’s binary addition function. It is responsible for (some redundant) type checks, and the conversion of the RHS operand. Lastly, it sums the values of the LHS and RHS objects and returns the result as a new object of the same type as LHS.

```

1  1. Check LHS type is IntegerObject
2  2. Cast LHS to IntegerObject
3  3. Check RHS type
4    3.1. IntegerObject: Cast RHS to IntegerObject
5    3.2. Object: Resolve RHS type
6      3.2.1. Check RHS type supports number operations
7        3.2.1.1. True: Resolve RHS number operations
8        3.2.1.2. False: Throw TypeError
9      3.2.2. Check RHS type supports IntegerObject conversion
10     3.2.2.1. True: Resolve the conversion function from RHS
              number operations
11     3.2.2.2. False: Throw TypeError
12     3.2.3. Invoke the type’s unary IntegerObject conversion
              function on RHS
13 4. Load the value field of the LHS and RHS IntegerObjects
14 5. Sum the loaded value fields
15 6. Allocate a new IntegerObject instance
16 7. Set the value of the new IntegerObject instance to the
      calculated sum
17 8. Return the new IntegerObject instance

```

LISTING 4.3: The addition of an object *RHS* to an integer object *LHS* in the object execution model.

The simple example of addition illustrates the degree of indirection involved with the object execution model, and the frequent allocation of new objects. Step 3.2.3. in Listing 4.3 potentially hides more conditional object allocation. In contrast, the case of addition in Tython’s specialization execution model (shown in Listing 4.4), is more compact and more local, and does not require a dynamic function dispatch for the operation or object allocation for its result.

```

1  1.  $\tau(LHS)$ 
2    1.1. Integer:  $\tau(RHS)$ 
3      1.1.1. Integer: add i64  $\pi'_0(LHS)$   $\pi'_0(RHS)$ 
4      1.1.2. Floating-Point: Convert RHS to Integer
5        1.1.2.1. %0 = fptosi float  $\pi'_1(RHS)$  to i64
6        1.1.2.2. add i64  $\pi'_0(LHS)$  %0
7      1.1.3. Object: convert object to Integer
8        1.1.3.1. %0 = call object_to_primitive on RHS
9        1.1.3.2. add i64  $\pi'_0(LHS)$   $\pi'_0(\%0)$ 
10     1.2. Floating-Point:  $\tau(RHS)$ 
11       1.2.1. Integer: Convert RHS to Floating-Point
12         1.2.2.1. %0 = sitofp i64  $\pi'_0(RHS)$  to float
13         1.2.2.2. add float  $\pi'_1(LHS)$  %0
14       1.2.2. Floating-Point: add float  $\pi'_1(LHS)$   $\pi'_1(RHS)$ 

```

```

15     1.2.3. Object: convert object to Floating-Point
16         1.2.3.1. %0 = call object_to_primitive on RHS
17         1.2.3.2. add float  $\pi'_1(LHS)$   $\pi'_1(\%0)$ 
18     1.3. Object:  $\tau(RHS)$ 
19         1.3.1. Integer: convert LHS object to Integer
20             1.3.1.1. %0 = call object_to_primitive on LHS
21             1.3.1.2. add i64  $\pi'_0(\%0)$   $\pi'_0(RHS)$ 
22         1.3.2. Floating-Point: convert LHS object to Floating-Point
23             1.3.2.1. %0 = call object_to_primitive on LHS
24             1.3.2.2. add float  $\pi'_1(\%0)$   $\pi'_1(RHS)$ 
25         1.3.3. Object: delegate to object execution model
26             1.3.3.1. See Listing 4.2

```

LISTING 4.4: The addition of scalar types in Tython’s specialization execution model.

Note that in Listing 4.4, the π' functions hide bitcasts. These bitcasts are no-ops inserted only for the benefit of the LLVM IR-level type system and do not result in code being emitted into the executable (and so they have no effect at runtime). Note that the semantics of type coercion in Tython are subtly different from Python. For LHS values of specialized types *Integer* or *FloatingPoint*, the semantics align with Python: the type of LHS is leading. If both LHS and RHS are *Object* types, the semantics also align with Python through delegation to the object execution model. However, if the LHS is an *Object* type, and RHS is a specialized type, Tython is able to avoid redundant object type checks and the allocation of new objects by letting the RHS type be leading in type coercion. If the LHS type would be leading, object type resolution of LHS would have to be applied, and the specialized value RHS would have to be *boxed* into either a newly allocated *IntegerObject* or *FloatObject*.

The control flow of Listing 4.4 is also not trivial. However, the conditional branching instructions on the comparison of a register value⁵ with a constant are much faster than the equivalent type resolution and checking in the object execution model.

In the worst case, LHS and RHS are both unspecialized objects, and the conditional branching results in a delegation of the addition operation to the object execution model. However, in a program where values can be specialized, the relatively small overhead incurred from this conditional branching can be offset by the large performance gain when specialization is applied. See chapter 5 for the performance measurements of Tython’s specialization strategy.

Additionally, this data layout and control structure for type guards allows for optimizations which can, in the best case, remove the type guard and reduce the addition of two values of the type *Integer* to the single instruction `add i64 $\pi'_0(LHS)$ $\pi'_0(RHS)$` \equiv `add i64 $\mu(LHS)$ $\mu(RHS)$` because $\tau(LHS) = \tau(RHS) = 0$. For two values of the type *FloatingPoint*, the best case is the single instruction `add float $\pi'_1(LHS)$ $\pi'_1(RHS)$` . Chapter 4 Section 4 discusses the general case and this concrete this example further.

4 Type Shifting

Tython implements several optimization strategies to improve the performance of Pythonic source code. Chapter 4 Section 2 describes how Tython employs an internal data and ex-

⁵Even if the value is pushed to the stack because of high register pressure, a single load from the stack is faster than multiple loads on the heap (which often results in page faults, as objects can be stored "far" from their associated type object).

ecution model that can implement specialized operations on *Integer* and *FloatingPoint* values using highly efficient low-level instructions, circumventing the slower object execution model. This is implemented as a tagged union, where the tag is any of the enumerator set $Tag = \{Integer, FloatingPoint, Object\}$. The dynamic typing of Python is preserved through the generation of runtime type guards and casts. These type guards introduce a runtime overhead of a conditional branches on all specialized high-level operations. However, the operands of these conditional branches are implemented as a comparison between a register value and a constant, making the overhead small compared to the type checking performed by the object execution model.

As discussed in [Chapter 4 Section 3.1](#), for some symbols Tython can apply static name binding. This allows for direct compile-time access to the definition of a name, without the need for runtime name resolution. CPython uses late binding, requiring a runtime hash table look-up for every use of a name. For those symbols where Tython can bind statically, this leads to earlier (compile-time) error reporting of undefined symbols than CPython, without the runtime performance overhead incurred by late binding.

The LLVM infrastructure provides a scala of program optimizations [31]. These are transformations which operate at the IR level, and are front-end agnostic.

Some transformations stand out for optimizing Tython's type system. This is because of the combination of Tython's specialization data layout and branching type guard structure. We will show how general LLVM optimizations can be applied in type inference, and can then use that inferred type information to reduce the number of type guards in a program.

In this section, we will discuss the effects of type shifting on a Python program compiled with Tython.

4.1 Type Inference

One inefficiency of late binding dynamic languages is that the type of the value associated with any variable must be checked at runtime for each use. For objects, this type checking involves pointer dereferencing and several steps of indirection. All objects live on the heap, which is a "free-for-all" memory region where write access is granted from across compilation units, and effectively from throughout the entire program. This makes it hard, if all information is statically available, or impossible, if not all information is visible to the compiler, to determine the state of an object on the heap at compile-time⁶.

If we are dealing with a local value, we can allocate that value locally on the stack. Their lifetime then becomes restricted by the parent function: stack members do not outlive their stack frame. Similarly, storing a local value in registers has no general meaning outside of a function at the LLVM IR level⁷. These relatively short and locally visible lifetimes allow LLVM to identify all read/write operations on local values. This completeness of visibility of local values is one of the benefits of ahead-of-time compilation over interpretation. Full read/write visibility is fundamental to LLVM's ability to prove that a field of a local value is constant.

There are certain expressions for which type inference is trivial. Literals are constant atomic expressions which do not depend on any other data, and whose value is defined locally. For specialized expressions over values with known types, the type of the expression is also trivial. These literals and specialized expressions form the roots from which type inference can propagate.

⁶The same applies to global values, regardless of their data model. Global values live on the heap, which is why LLVM cannot "see" all their uses.

⁷At the LLVM IR level, we are dealing with SSA names, which are not aliased to a register yet.

The LLVM transformation which takes care of the propagation of static type information is the Scalar Replacement of Aggregates (SROA). SROA breaks up stack allocations of aggregate members into separate allocation instructions. This is a clearer SSA form, which immediately allows LLVM to identify write access to the type field of a local value. If all write access sets the same scalar constant (a value's type is implemented as an integer enumerator), then LLVM removes unnecessary writes to the field and replaces all subsequent reads of the field by the identified constant.

Recall that type guards are implemented as conditional branches on the comparison between the type field of the guardee and a constant type enumerator (one for each element in the set *Tag*). This replacement of type tags by constants transforms the type guard into conditional branches where the condition is a result of the comparison of two constants.

4.2 Type guards

The constant propagation of the type tag of a value makes type guard conditions trivial. LLVM can replace the comparison of two constants by a constant which is the result of the comparison. It can then remove all conditional branching instructions whose conditions are trivially false. Conditional branches where the condition is trivially true are replaced by an unconditional branch. The Sparse Conditional Constant Propagation (SCCP) transformation handles all of this.

Lastly, after the successful removal of conditional branches in a type guard, the control flow graph (CFG) of the program is left untidy. The specialized handlers for all types other than the inferred static type of the guardee have become unreachable nodes. These nodes can safely be removed, as they will never execute. The transformation that handles this is called Simplify the CFG (*simplifycfg*). It also merges CFG nodes which have one parent in the graph, and are reached through an unconditional branch.

4.3 An Example

We apply three transformations:

1. Scalar Replacement of Aggregates (SROA)
2. Sparse Conditional Constant Propagation (SCCP)
3. Simplify the CFG (*simplifycfg*)

[Listing 4.5](#), [Listing 4.6](#), and [Listing 4.7](#) show these optimizations applied to the type guard algorithm showing the concrete example of addition described in [Listing 4.4](#). Note that the unreachable CFG nodes after step 2 (shown in [Figure 3.4](#)) are not represented in the algorithm in [Listing 4.6](#), because these steps are never considered.

This example shows how the general LLVM optimizations can concretely implement the type shifting described in [Chapter 4 Section 4](#).

```

1 1. Integer
2   1.1. Integer: Integer
3     1.1.1. Integer: add i64  $\pi'_0(LHS)$   $\pi'_0(RHS)$ 
4     1.1.2. Floating-Point: Convert RHS to Integer
5       1.1.2.1. %0 = fptosi float  $\pi'_1(RHS)$  to i64
6       1.1.2.2. add i64  $\pi'_0(LHS)$  %0
7     1.1.3. Object: convert object to Integer
8       1.1.3.1. %0 = call object_to_primitive on RHS
9       1.1.3.2. add i64  $\pi'_0(LHS)$   $\pi'_0(%0)$ 
10  1.2. Floating-Point: Integer
11    1.2.1. Integer: Convert RHS to Floating-Point
12      1.2.2.1. %0 = sitofp i64  $\pi'_0(RHS)$  to float
13      1.2.2.2. add float  $\pi'_1(LHS)$  %0
14    1.2.2. Floating-Point: add float  $\pi'_1(LHS)$   $\pi'_1(RHS)$ 
15    1.2.3. Object: convert object to Floating-Point
16      1.2.3.1. %0 = call object_to_primitive on RHS
17      1.2.3.2. add float  $\pi'_1(LHS)$   $\pi'_1(%0)$ 
18  1.3. Object: Integer
19    1.3.1. Integer: convert LHS object to Integer
20      1.3.1.1. %0 = call object_to_primitive on LHS
21      1.3.1.2. add i64  $\pi'_0(%0)$   $\pi'_0(RHS)$ 
22    1.3.2. Floating-Point: convert LHS object to Floating-Point
23      1.3.2.1. %0 = call object_to_primitive on LHS
24      1.3.2.2. add float  $\pi'_1(%0)$   $\pi'_1(RHS)$ 
25    1.3.3. Object: delegate to object execution model
26      1.3.3.1. See Listing 4.2

```

LISTING 4.5: The inference of $\tau(LHS)$ and $\tau(RHS)$.

```

1 1.1. True:
2   1.1.1. True: add i64  $\pi'_0(LHS)$   $\pi'_0(RHS)$ 
3   1.1.2. False: Convert RHS to Integer
4     1.1.2.1. %0 = fptosi float  $\pi'_1(RHS)$  to i64
5     1.1.2.2. add i64  $\pi'_0(LHS)$  %0
6   1.1.3. False: convert object to Integer
7     1.1.3.1. %0 = call object_to_primitive on RHS
8     1.1.3.2. add i64  $\pi'_0(LHS)$   $\pi'_0(\%0)$ 
9 1.2. False:
10  1.2.1. False: Convert RHS to Floating-Point
11    1.2.2.1. %0 = sitofp i64  $\pi'_0(RHS)$  to float
12    1.2.2.2. add float  $\pi'_1(LHS)$  %0
13  1.2.2. False: add float  $\pi'_1(LHS)$   $\pi'_1(RHS)$ 
14  1.2.3. False: convert object to Floating-Point
15    1.2.3.1. %0 = call object_to_primitive on RHS
16    1.2.3.2. add float  $\pi'_1(LHS)$   $\pi'_1(\%0)$ 
17 1.3. False:
18  1.3.1. False: convert LHS object to Integer
19    1.3.1.1. %0 = call object_to_primitive on LHS
20    1.3.1.2. add i64  $\pi'_0(\%0)$   $\pi'_0(RHS)$ 
21  1.3.2. False: convert LHS object to Floating-Point
22    1.3.2.1. %0 = call object_to_primitive on LHS
23    1.3.2.2. add float  $\pi'_1(\%0)$   $\pi'_1(RHS)$ 
24  1.3.3. False: delegate to object execution model
25    1.3.3.1. See Listing 4.2

```

LISTING 4.6: The pruning of CFG edges and the resulting unreachability of nodes.

```

1 add i64  $\pi'_0(LHS)$   $\pi'_0(RHS)$ 

```

LISTING 4.7: The merging of straight-line blocks and the deletion of unreachable blocks.

Chapter 5

Evaluation

In this chapter, we first describe the methodology of measuring the performance of Tython, CPython and Codon on a standard set of benchmarks. In [Chapter 5 Section 2](#) we show an overview of the results of these measurements. In [Chapter 5 Section 3](#) we describe each benchmark in more detail and look at profiler data. Lastly, in section [Chapter 5 Section 4](#) we discuss how the measurement results of Tython, CPython and Codon align with our expectations.

1 Methodology

In this section we evaluate the performance benefits of type shifting and specialization.

We run benchmarks to measure the executables generated by Tython to quantitatively argue that the specialization and type shifting of values in the execution model have a positive performance impact. We also measure the execution of the same source code in CPython.

The programs we run comprise a benchmark suite, selected to put the various features of the language implementation under stress. We run the benchmarks under a changing set of build configurations, with the aim of identifying their relative performance impact. The primary metric of interest is execution time. Each benchmark is run 10 times to obtain the fastest, slowest, mean, and median execution times, as well as their variance.

All benchmarks are compiled against a CPU profiler, allowing for a procedure-level breakdown of their execution. When discussing the number of samples found in a function, we refer to the samples found in that function *and* its delegates. The percentage of samples found is a percentage of the total number of samples taken during the execution of the benchmark. This encapsulation means that the percentages we discuss in running text should not be expected to add up 100%.

1.1 Benchmarks

Because Tython is a minimal language implementation, we choose benchmarks that use only those features that the compiler currently supports. Benchmarks are selected to only make use of built-in Python data types. Since Tython is purpose-built, it does not currently support user-defined classes. Tython implements specializations for scalar data types, which we are interested in measuring. We are also interested in measuring function calls, as Tython applies early binding. Lastly, the list and tuple data types are fundamentally Pythonic, as well as for-loops [\[29\]](#).

Benchmark	Scalar values	Function calls	Container types	For-loops
Fibonacci	x			
Fibonacci (recursive)	x	x		
NBODY	x	x	x	x
NQUEENS	x	x	x	x
PI	x		x	x
QuickSort	x	x	x	x
Sieve	x		x	x

TABLE 5.1: The selection of benchmarks, and their contribution to the measurement goals.

Table 5.1 shows the selected benchmarks, and how each contributes to the measurement goals.

1.2 Environment

All benchmark results were obtained on the same hardware. The operating system is Ubuntu 23.10¹. The CPU is *AMD Ryzen 7 5800U*. The system has two identical RAM modules, installed in two banks: Samsung M471A1G44AB0-CWE 8GiB SODIMM DDR4 at 3200 MHz, for a total of 16GiB of DDR4 RAM.

The kernel parameter "rtprio" is set to "99" for the logged-in user, allowing us to run the benchmarks with real-time priority using `chrt -f 99`. The command `chrt -f 99 perf stat -e user_time,instructions,system_time -x '□' <executable>` uses `perf stat` to perform measurements on the execution times of the benchmarks.

We use the CPU profiler included with the gperftools performance analysis toolset [14], version 2.0. All benchmarks are dynamically linked against the profiler with the Clang argument `-lprofiler`. It samples at a rate of 10 000 samples per second.

We are evaluating Tython v0.1 [28]. The reference implementation, CPython, is version 3.11.6. The Codon implementation is version 0.16.3.

1.3 Build configurations

We run the benchmarks on Tython with four different build configurations. Table 5.3 shows the exact compiler flags used for each build configuration. All configurations are prepended with `tython -m <source_path> -l <runtime_library_path> -o <output_path>`, forming the complete compilation command.

The results can be reproduced by running the script `run.sh`, included in the Tython Git repository [27]. The results listed in this section are obtained by running the script from the `benchmarks/` directory with Tython v0.1 [28]: `./run.sh src 10 --tython --cpython --codon`.

¹The exact `uname -a` output: `Linux IdeaPad-5-Pro-14ACN6 6.5.0-41-generic #41-Ubuntu SMP PRE-EMPT_DYNAMIC Mon May 20 15:55:15 UTC 2024 x86_64 x86_64 x86_64 GNU/Linux`

Configuration1

First, we compile the benchmarks without generating specialized values. This means that all values are objects: $\forall v \in Spec. \tau(v) = Object$. Type guards are generated for all uses of all values, but no type guard reduction is applied.

Configuration2

Next, we run the benchmarks with a similar build configuration, the only difference being that type guards are not generated. All runtime type checks and operations happen through the object execution model. Having all values as objects allows us to deduce the overhead incurred by the additional type checks Tython performs on the use of values. The runtime directly delegates type checks and operations on all values to the object execution model. Out of the four configurations, the build artefacts generated by Configuration 2 are most similar in execution model to CPython.

Configuration3

The next build configuration enables specialized values. Recall that the specialized types are *Integer* and *FloatingPoint*. This optimization particularly affects arithmetic-intensive benchmarks. Other data types, such as container types, are non-specialized and evaluated through the object execution model. Elements contained in a container type can only be of type *Object*.

Configuration4

Lastly, we build the benchmarks with type-shifting optimizations applied. Type shifting is emergent from the application of general program optimizations. These optimizations may affect more aspects of the program than just the propagation of type information and the reduction of type guards.

2 Measurements

In this section we give an overview of the measurement results. [Figure 5.1](#) plots the mean execution times of the performance of the benchmarks under different build configurations. Note that the bars for Fibonacci¹ (recursive implementation) and Fibonacci² (forward implementation) are present, but the execution times of Configuration 3 and Configuration 4 are significantly smaller than those of Configuration 1 and Configuration 2, making the bars appear very small. Configuration 1 and Configuration 2 failed to run the PI benchmark due to high memory consumption. We will discuss this further in [Chapter 5 Section 3](#). [Table 5.2](#) shows the relative speed-up between Configurations 1 and 2, between Configuration 2 and 3, and between Configuration 1 and 4. Recall that the difference between Configurations 1 and 2 is only that Configuration 1 generates type guards whereas Configuration 2 does not. Both only use the object execution model. The difference between Configurations 3 and 4 is only that Configuration 4 applies the general optimizations that enable type shifting. Both make use of the specialization execution model. The differences between Configurations 1 and 4 are that Configuration 1 always generates type guards – even though it always uses the object execution model – and Configuration 4 leverages the specialization execution model and reduces the number of generated type guards.

[Table 5.4](#) through [Table 5.7](#) show the execution time of benchmark executables produced by Tython under the four different build configurations.

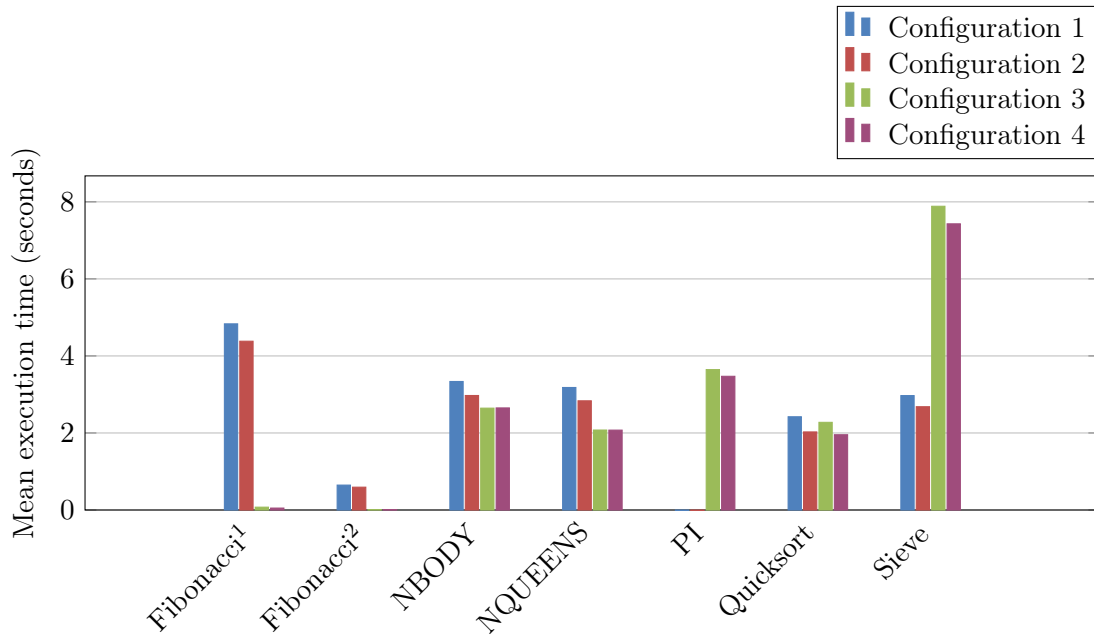


FIGURE 5.1: The mean execution times for each benchmark under the four different build configurations.

	Configuration 1 / Configuration 2	Configuration 3 / Configuration 4	Configuration 1 / Configuration 3
Fibonacci (recursive)	9.3%	31.9%	98.5%
Fibonacci	8.3%	12.5%	98.1%
NBODY	10.9%	-0.2%	20.7%
NQUEENS	10.9%	0.1%	34.7%
PI	N/A	4.8%	N/A
QuickSort	16.3%	14.1%	6.0%
Sieve	9.7%	5.8%	-165.4%

TABLE 5.2: The relative speed-up of mean execution time between build configurations.

Label	Configuration
Configuration1	-d -no-specialize
Configuration2	-d -no-specialize -no-guards
Configuration3	-d
Configuration4	

TABLE 5.3: The benchmark build configurations.

	Fastest Time	Mean Time	Median Time	Max Time	Variance (s ²)
fibonacci (recursive)	4.5059	4.5620	4.5435	4.7805	0.0057
fibonacci (while)	0.5995	0.6709	0.6806	0.7702	0.0030
nbody	3.1348	3.1817	3.1861	3.2249	0.0005
nqueens	2.8806	2.9376	2.9194	3.1624	0.0058
pi	error	error	error	error	error
quicksort	2.1752	2.1993	2.1980	2.2434	0.0004
sieve	2.8970	3.0524	3.0601	3.1669	0.0084

TABLE 5.4: The benchmark results for Configuration1 (times in seconds).

3 Benchmarks

In this section we describe each benchmark in more detail and inspect profiler measurements to decompose the program execution and reason about the performance implications of its components.

3.1 Fibonacci

The Fibonacci sequence is a sequence where each number is the sum of the two numbers preceding it. [Listing 5.1](#) shows a straightforward implementation of summing the first 20 Fibonacci numbers. It works by starting with the sequence 0 and 1, and then iteratively

	Fastest Time	Mean Time	Median Time	Max Time	Variance (s ²)
fibonacci (recursive)	4.4355	4.4709	4.4629	4.5194	0.0007
fibonacci (while)	0.6060	0.6123	0.6125	0.6186	0.0000
nbody	2.9824	3.0279	3.0283	3.0622	0.0008
nqueens	2.8078	2.8464	2.8474	2.8869	0.0005
pi	error	error	error	error	error
quicksort	2.0405	2.0853	2.0890	2.1099	0.0006
sieve	2.6616	2.7709	2.7861	2.8379	0.0030

TABLE 5.5: The benchmark results for Configuration2 (times in seconds).

	Fastest Time	Mean Time	Median Time	Max Time	Variance (s ²)
fibonacci (recursive)	0.0741	0.0750	0.0750	0.0757	0.0000
fibonacci (while)	0.0119	0.0128	0.0128	0.0135	0.0000
nbody	2.5531	2.5927	2.6037	2.6189	0.0006
nqueens	2.0471	2.1044	2.0818	2.1816	0.0027
pi	3.6000	3.7017	3.6761	3.8081	0.0050
quicksort	2.1067	2.1611	2.1360	2.2735	0.0025
sieve	8.0167	8.5803	8.2839	9.3875	0.2698

TABLE 5.6: The benchmark results for Configuration3 (times in seconds).

	Fastest Time	Mean Time	Median Time	Max Time	Variance (s ²)
fibonacci (recursive)	0.0497	0.0515	0.0516	0.0525	0.0000
fibonacci (while)	0.0102	0.0110	0.0108	0.0124	0.0000
nbody	2.6325	2.6992	2.7056	2.7355	0.0010
nqueens	2.0867	2.1084	2.1117	2.1268	0.0002
pi	3.4680	3.5113	3.5014	3.5637	0.0011
quicksort	2.0057	2.0516	2.0705	2.0873	0.0009
sieve	7.5190	7.6125	7.6391	7.7054	0.0039

TABLE 5.7: The benchmark results for Configuration4 (times in seconds).

accumulating the sum of the preceding two fibonacci numbers in a while-loop. To obtain a longer execution time, the whole algorithm is wrapped in an outer loop of 174 000 iterations.

Features Under Stress

This benchmark mostly stresses integer arithmetics. There are no higher-order types (such as lists, range objects, tuples, etc.) involved with the algorithm.

Profiler Data

For build configuraion 1, the profiler shows that arithmetics in the object execution model involve the creation of many integer objects on the heap. Approximately 85% of samples are found in the function `int_create`. Integer objects are created to represent source-level literals, as the result of the comparison between integer objects, and to represent the result of arithmetic operations between integer objects. For build configuration 4, the profiler does not manage to interrupt the CPU sufficiently fast to produce more than one sample in the program entry point. Since all values in this benchmark are of the type *Integer*, and the only operations on those values are boolean comparisons, addition and assignment, Tython manages to execute the entire program without delegation to the object execution model. [Figure A.1](#) shows a graphical representation of the complete profiler output for build configuration 1.

```
1 def bench(n):
2
3     sum = 0
4     a = 0
5     b = 1
6
7     i = 0
8     while i < 174000:
9         i += 1
10
11        a = 0
12        b = 1
13
14        j = 0
15        while j < n:
16            a = b
17            b = a + b
18
19            j += 1
20
21        sum += a
22
23 bench(20)
```

LISTING 5.1: The iterative Fibonacci benchmark.

3.2 Fibonacci Recursive

Listing 5.2 shows an implementation of calculating the sum of the same Fibonacci sequence as described above in Chapter 5 Section 3.1. The difference is that this implementation uses the naive recursive approach. It starts at calculating the 20th as being the sum of the 19th and the 18th number. It then recursively determines the 19th number to be the sum of the 18th and the 17th. Notice how in the second step, the 18th number is already calculated twice. Other recursive algorithms cache previously calculated values, improving performance. This naive approach is computationally more expensive, making for a more interesting benchmark.

Features Under Stress

The heavy recursion in this benchmark explicitly stresses function calls, in addition to stressing integer arithmetics like the forward implementation of the Fibonacci sequence.

Profiler Data

The profiler data for the benchmark executable generated under configuration 1 shows that for this recursive implementation, similarly to the forward implementation, about 85% of samples are found in the function `int_create`. It also shows that the function `fib` recursively calls itself many times, as is expected. The executable generated under build configuration 4 also does not produce interesting profiler output. Tython is able to specialize all values and operations, resulting in no delegation to the object execution model. All samples are found in the `fib` function, calling itself recursively. Figure A.2 shows a graphical representation of the complete profiler output for build configuration 1.

```
1 def fib(n):
2
3     if n == 0:
4         return 0
5
6     else if n == 1:
7         return 1
8     else if n == 2:
9         return 1
10
11     else:
12         return fib(n - 1) + fib(n - 2)
13
14 def bench(n):
15
16     sum = 0
17
18     for i in range(0, 1000, 1):
19         sum += fib(n)
20
21     print(sum)
22
23 bench(20)
```

LISTING 5.2: The naive recursive Fibonacci benchmark.

3.3 NBODY

The NBODY benchmark models the orbits and interaction of the Jovian planets in the solar system. It creates a pairing of all bodies, and iteratively compares all pairs to advance the motion of all bodies by a discrete timestep. An energy scalar is calculated before and after each discrete timestep.

Features under stress

The NBODY benchmark stresses floating-point arithmetics, and list and tuple access. The modelling of the motion of the helical bodies involves the solving of physics formulae involving real numbers. The scalar representation of the "energy" of a body is also a function in the real domain. The position of the N bodies are stored in a nested data structure of lists and tuples. The benchmark also involves many function calls.

Profiler Data

For configuration 1, the profiler shows that floating-point arithmetics make up about 34% of samples. The results of arithmetic operations involves the creation of new floating-point objects with the function `float_create`, which is good for about 27% of samples. List and tuple access by an integer key is nested in for-loops. This creates many integer objects on the heap, resulting in about 52% of samples found in the function `int_create`. List and tuple access amount to about 5% of samples. Configuration 4 can specialize on the integer and floating-point literals found in the source code. The arithmetic operations make up about 12% of samples under this build configuration. However, since container objects are handled exclusively in the object execution model, all specialized values must be boxed

before they can be used as indices or set as an element in a container object. Boxing of both integers and floating-point specialized values accounts for approximately 20% of samples. In total, integer creation is good for 59% of samples, and the creation of floating-point objects for 19%. List and tuple access amount to approximately 26.2% of samples. A graphical representation of the complete profiler output for build configurations 1 and 4 is shown in [Figure A.3](#) and [Figure A.4](#).

```

1  # Adapted from https://github.com/python/pyperformance/blob/main/
    pyperformance/data-files/benchmarks/bm_nbody/run_benchmark.py
2
3  DEFAULT_ITERATIONS = 1000
4  DEFAULT_REFERENCE = "sun"
5  DEFAULT_LOOPS = 100
6
7  PI = 3.14159265358979323
8  SOLAR_MASS = 4.0 * PI * PI
9  DAYS_PER_YEAR = 365.24
10
11 BODIES = {
12     "sun": ([0.0, 0.0, 0.0], [0.0, 0.0, 0.0], SOLAR_MASS),
13
14     "jupiter": ([4.84143144246472090e+00,
15                 -1.16032004402742839e+00,
16                 -1.03622044471123109e-01],
17                [1.66007664274403694e-03 * DAYS_PER_YEAR,
18                 7.69901118419740425e-03 * DAYS_PER_YEAR,
19                 -6.90460016972063023e-05 * DAYS_PER_YEAR],
20                9.54791938424326609e-04 * SOLAR_MASS),
21
22     "saturn": ([8.34336671824457987e+00,
23                4.12479856412430479e+00,
24                -4.03523417114321381e-01],
25               [-2.76742510726862411e-03 * DAYS_PER_YEAR,
26                4.99852801234917238e-03 * DAYS_PER_YEAR,
27                2.30417297573763929e-05 * DAYS_PER_YEAR],
28               2.85885980666130812e-04 * SOLAR_MASS),
29
30     "uranus": ([1.28943695621391310e+01,
31                -1.51111514016986312e+01,
32                -2.23307578892655734e-01],
33               [2.96460137564761618e-03 * DAYS_PER_YEAR,
34                2.37847173959480950e-03 * DAYS_PER_YEAR,
35                -2.96589568540237556e-05 * DAYS_PER_YEAR],
36               4.36624404335156298e-05 * SOLAR_MASS),
37
38     "neptune": ([1.53796971148509165e+01,
39                 -2.59193146099879641e+01,
40                 1.79258772950371181e-01],
41                [2.68067772490389322e-03 * DAYS_PER_YEAR,
42                 1.62824170038242295e-03 * DAYS_PER_YEAR,
43                 -9.51592254519715870e-05 * DAYS_PER_YEAR],
44                5.15138902046611451e-05 * SOLAR_MASS)
45 }
46
47 def combinations(l):
48

```



```
49     result = []
50     q = len(l) - 1
51
52     for x in range(0, q, 1):
53         i = x + 1
54         ls = l[i:]
55
56         for y in ls:
57             result.append((l[x], y))
58
59     return result
60
61 SYSTEM = list(BODIES.values())
62 PAIRS = combinations(SYSTEM)
63
64 def advance(dt, n):
65     for i in range(0, n, 1):
66         for p in PAIRS:
67
68             x1 = p[0][0][0]
69             y1 = p[0][0][1]
70             z1 = p[0][0][2]
71             v1 = p[0][1]
72             m1 = p[0][2]
73
74             x2 = p[1][0][0]
75             y2 = p[1][0][1]
76             z2 = p[1][0][2]
77             v2 = p[1][1]
78             m2 = p[1][2]
79
80             dx = x1 - x2
81             dy = y1 - y2
82             dz = z1 - z2
83             mag = dt * ((dx * dx + dy * dy + dz * dz) ** (-1.5))
84             b1m = m1 * mag
85             b2m = m2 * mag
86             v1[0] -= dx * b2m
87             v1[1] -= dy * b2m
88             v1[2] -= dz * b2m
89             v2[0] += dx * b1m
90             v2[1] += dy * b1m
91             v2[2] += dz * b1m
92
93         for b in SYSTEM:
94
95             r = b[0]
96             vx = b[1][0]
97             vy = b[1][1]
98             vz = b[1][2]
99             m = b[2]
100
101             r[0] += dt * vx
102             r[1] += dt * vy
103             r[2] += dt * vz
104
```

```
105 def report_energy():
106
107     e = 0.0
108
109     for p in PAIRS:
110
111         x1 = p[0][0][0]
112         y1 = p[0][0][1]
113         z1 = p[0][0][2]
114         v1 = p[0][1]
115         m1 = p[0][2]
116
117         x2 = p[1][0][0]
118         y2 = p[1][0][1]
119         z2 = p[1][0][2]
120         v2 = p[1][1]
121         m2 = p[1][2]
122
123         dx = x1 - x2
124         dy = y1 - y2
125         dz = z1 - z2
126         e -= (m1 * m2) / ((dx * dx + dy * dy + dz * dz) ** 0.5)
127
128     for b in SYSTEM:
129
130         r = b[0]
131         vx = b[1][0]
132         vy = b[1][1]
133         vz = b[1][2]
134         m = b[2]
135
136         e += m * (vx * vx + vy * vy + vz * vz) / 2.0
137
138     return e
139
140
141 def offset_momentum(ref):
142
143     px=0.0
144     py=0.0
145     pz=0.0
146
147     for b in SYSTEM:
148
149         r = b[0]
150         vx = b[1][0]
151         vy = b[1][1]
152         vz = b[1][2]
153         m = b[2]
154
155         px -= vx * m
156         py -= vy * m
157         pz -= vz * m
158
159     v = ref[1]
160     m = ref[2]
```

```

161
162     v[0] = px / m
163     v[1] = py / m
164     v[2] = pz / m
165
166
167 def bench_nbody(loops, reference, iterations):
168     offset_momentum(BODIES[reference])
169
170     range_it = range(0, loops, 1)
171
172     for _ in range_it:
173         report_energy()
174         advance(0.01, iterations)
175         report_energy()
176
177 bench_nbody(DEFAULT_LOOPS, DEFAULT_REFERENCE, DEFAULT_ITERATIONS)

```

LISTING 5.3: The NBODY benchmark implementation, adapted from [40].

3.4 PI

The PI benchmark shown in Listing 5.4 approximates the value of π to 800 digits using an iterative algorithm. The benchmark does not finish on the evaluation platform under configurations 1 or 2. The operating system kills the process because of high memory consumption. Configurations 3 and 4 consume less memory and do finish.

Features Under Stress

The PI benchmark uses a lot of integer arithmetics to iteratively calculate the next digits of pi. Found digits are stored in a list object and earlier digits are retrieved from that same list to find and append the next digit, making it heavy on list access as well.

Profiler Data

As mentioned, the benchmark generated under configuration 1 does not finish on the evaluation platform due to high memory consumption. No profiler data is available for the benchmark under this configuration. Configuration 4 does finish. The profiler data shows that list iteration makes up about 64% of samples. Included in this is the boxing of specialized integer values, good for a portion of 54% of total samples. Integer object creation is responsible for 81% of samples. Figure A.5 shows a graphical representation of the complete profiler output for build configuration 4.

```

1     # Adapted from https://crypto.stanford.edu/pbc/notes/pi/code.
2     html
3
4     LIMIT = 2800
5
6     def bench(limit):
7
8         for k in range(0, 100):
9
10            r = []; # len(r) will be limit + 1
11            i = 0

```

```
11         b = 0
12         c = 0
13
14         for _ in range(0, limit):
15             r.append(2000)
16
17         r.append(0)
18
19         result = []
20
21         for k in range(limit, 0, -14):
22
23             d = 0
24             i = k
25
26             while True:
27
28                 d += r[i] * 10000
29                 b = 2 * i - 1
30
31                 r[i] = d % b
32                 d = d / b
33                 i -= 1
34
35                 if (i == 0):
36                     break
37
38                 d = d * i
39
40             result.append(c + d / 10000)
41             c = d % 10000
42
43     bench(LIMIT)
```

LISTING 5.4: The PI benchmark implementation, adapted from [23].

Note: there is an implementation of a PI benchmark which specifically stresses large integers. Tython’s semantics do not allow for more than 4 digits of π before an integer overflow occurs (see [Chapter 5 Section 5](#)).

3.5 Sieve

The Sieve benchmark, shown in [Listing 5.5](#), finds all prime numbers up to $N = 30\,000\,000$. It works by first instantiating a list of booleans of length N , marking all indexes as candidate prime numbers. At the end of the algorithm, a number k is prime iff the element in the list at index k is *True*. The algorithm iterates over the elements of the list until the condition $k^2 \leq N$ evaluates to false. For $N = 30\,000\,000$ this happens at $k = 5\,478$. Lastly, the benchmark iterates over the entire list and counts the number of primes encountered.

Features Under Stress

This benchmark is heavy on list access. The instantiation of the list of candidate primes involves 30 000 001 calls to the list object’s `append` method. The program then iterates over list elements 5 478 times (reading an element value at every iteration) with an inner loop

Function	% of samples
<code>__set__</code>	28%
<code>object_is_truthy</code>	27%
<code>range_iterator_next</code>	16%
<code>list_subscript</code>	9%
<code>box_internal</code>	8%
<code>resolve_builtin_method</code>	8%

TABLE 5.8: A summary of the profiler output for the Sieve benchmark under build configuration 1.

iterating over even more list elements setting list elements to False. Lastly, the benchmark iterates over 29 999 999 list elements, and reads their value.

Profiler Data

The profiler data in [Table 5.8](#) and [Table 5.9](#) quantify the involvement of list operations. See [Figure A.6](#) and [Figure A.7](#) for more detailed graphical profiler output.

Shown in [Table 5.8](#) is a summary of the profiler data for the benchmark under build configuration 1. Roughly 28% of samples are found in the `__set__` function, which deals with setting the value of a list element.

27% of samples are found in the function `object_is_truthy`, which handles the truthiness evaluation of arbitrary objects. In this program, the loop conditions and if-statements are evaluated very often. For the range-based for-loops, the iterator is incremented at every iteration. Hence, 16% of samples are encountered in the function `range_iterator_next`.

The function `list_subscript` handles list access by index and makes up approximately 9% of profiler samples. This function is evoked from the program source code on line 12 and 19, and indirectly as a delegate of `__set__` on line 14. Another 8% of samples is found in the function `box_internal`. This function is responsible for boxing specialization values, making them suitable for the object runtime. `box_internal` is only called from `__set__` in this program. About 8% of samples are found in the function `resolve_builtin_method`, which resolves the list object's `append` method.

For configuration 4, the profiler measures a very different result. [Table 5.9](#) shows that the function `int_create` comprises 45% of samples. The function is responsible for the allocation and instantiation of integer objects. It is called from `__set__` through `box_internal`. The function `int_create` is also iteratively called to transform the boolean literal (implemented as a specialized integer) on line 7 from a register value into an object. The function `object_is_truthy` takes up 31% of samples. The functions `resolve_builtin_method` and `range_iterator_next` are good for about 7% and 6% of samples respectively. Only about 1% of samples is found in the function `list_subscript`.

Reasoning

[Figure 5.1](#) shows that the Sieve benchmark executables under non-specializing build configurations are significantly faster than the specializing builds. The profiler data offers us an understanding of why that is. [Table 5.9](#) shows that the benchmark executable generated under build configuration 4 spends a lot of time on the creation of integer objects.

Function	% of samples
int_create	45%
__set__	40%
box_internal	31%
object_is_truthy	31%
resolve_builtin_method	7%
range_iterator_next	6%
list_subscript	1%

TABLE 5.9: A summary of the profiler output for the Sieve benchmark under build configuration 4.

This is because specialization values must be boxed before they can be used by the object runtime, which handles list operations. Lists are handled through the object execution model because they have no specialized representation. Configuration 1 does not create specialization values; recall from [Chapter 5 Section 1.3](#) that $\forall v \in \text{Value}.\tau(v) = \text{Object}$. This means that boxing is not necessary. In the function `box_internal`, if a value v has type *Object*, the function simply returns the contained object pointer.

```

1  # Adapted from https://www.geeksforgeeks.org/python-program-for-
    sieve-of-eratosthenes/
2
3  def SieveOfEratosthenes(num):
4
5      prime = []
6      for i in range(0, num+1):
7          prime.append(True)
8
9      p = 2
10     while (p * p <= num):
11
12         if (prime[p] == True):
13             for i in range(p * p, num+1, p):
14                 prime[i] = False
15         p += 1
16
17     count = 0
18     for p in range(2, num+1):
19         if prime[p]:
20             count += 1
21
22     print("count:_" + str(count))
23
24 SieveOfEratosthenes(30000000)

```

LISTING 5.5: The sieve benchmark implementation, adapted from [13].

3.6 Quicksort

The benchmark in [Listing 5.6](#) is a recursive implementation of the quicksort algorithm, adapted from [12]. Quicksort is an efficient general sorting algorithm.

Features Under Stress

Quicksort sorts lists, and is therefore continually involved with reading from and writing to the list to be sorted. It also compares many of the list elements. The benchmark is a recursive implementation, and is heavy on function calls.

Profiler Data

The profiler shows that, under configuration 1, the generated executable is heavy on the comparison of integer objects (17%). The addition of integers for the pivot and the indices of the partitions has a significant portion of samples at about 20%. Most of these samples are found in the delegate function `in_create`. The creation of integers for source-level literals, for the results of integer object comparisons, and for the results of integer arithmetics adds up to about 59% of total samples. List access is about 14% of samples. For configuration 4, profiling the generated executable still results in approximately 58% of samples found in the function `int_create`. However, integer values have been specialized and arithmetics operations are not run through the object execution model (so no integer objects are created to store the operation results). List access does now require the boxing of specialized integer values (21%), which also results in the creation of integer objects. Comparison between elements of a list object, good for 20% of samples, goes through the object execution model and still involves the creation of integer objects. A graphical representation of the complete profiler output for build configurations 1 and 4 is shown in [Figure A.8](#) and [Figure A.9](#).

```
1  # Adapted from https://www.geeksforgeeks.org/python-program-for-
   quicksort/
2
3  def partition(array, low, high):
4
5      pivot = array[high]
6
7      i = low - 1
8
9      for j in range(low, high):
10         if array[j] <= pivot:
11
12             i = i + 1
13
14             temp = array[i]
15             array[i] = array[j]
16             array[j] = temp
17
18
19         temp = array[i + 1]
20         array[i + 1] = array[high]
21         array[high] = temp
22
23     return i + 1
24
```

```
25 def quickSort(array, low, high):
26
27     if low < high:
28
29         pi = partition(array, low, high)
30
31         quickSort(array, low, pi - 1)
32         quickSort(array, pi + 1, high)
33
34 def foo():
35
36     data = [1, 7, 4, 1, 10, 9, -2, ...] #... omitted for brevity
37
38     size = len(data)
39
40     quickSort(data, 0, size - 1)
41
42 for _ in range(0, 1000):
43     foo()
```

LISTING 5.6: The quicksort benchmark implementation, adapted from [12].

3.7 NQueens

The NQueens benchmark, shown in Listing 5.7, determines whether N queens can be placed on a chessboard of dimensions $N * N$ such that no queens attack each other.

Features Under Stress

The implementation uses recursion to solve the NQueens problem, involving many function calls. The chessboard is implemented as a list of lists, making the benchmark involve a large degree of list access.

Profiler Data

For configuration 1, the profiler shows a large number of samples (49%) in the `zip` function. This function makes heavy use of subscript access to range objects (30%). List access is good for about 5% of samples. Integer object comparison makes up 11% of samples. In total, integer object creation accounts for 38% of samples. Under configuration 4, the `zip` function makes up about 64% of samples. Range subscript access is also high at 38% of total samples. List access makes up 8% of samples. Integer creation contributes less than under configuration 1, with approximately 21% of total samples. A graphical representation of the complete profiler output for build configurations 1 and 4 is shown in Figure A.10 and Figure A.11.

```
1 # Adapted from https://www.geeksforgeeks.org/python-program-for-n-
   queen-problem-backtracking-3/
2 import util
3
4 N = 18
5
6 def isSafe(board, row, col):
7
8     for i in range(0, col):
```



```
9         if board[row][i] == 1:
10             return False
11
12     for z in zip(range(row, -1, -1), range(col, -1, -1)):
13         if board[z[0]][z[1]] == 1:
14             return False
15
16     for z in zip(range(row, N, 1), range(col, -1, -1)):
17         if board[z[0]][z[1]] == 1:
18             return False
19
20     return True
21
22 def solveNQUtil(board, col):
23     if col >= N:
24         return True
25
26     for i in range(0, N):
27
28         if isSafe(board, i, col):
29             board[i][col] = 1
30
31             if solveNQUtil(board, col + 1) == True:
32                 return True
33
34             board[i][col] = 0
35
36     return False
37
38 def solveNQ():
39
40     board = []
41     for _ in range(0, N):
42         row = []
43         for _ in range(0, N):
44             row.append(0)
45         board.append(row)
46
47     solveNQUtil(board, 0)
48
49 solveNQ()
```

LISTING 5.7: The nqueens benchmark implementation, adapted from [11].

4 Discussion

In this section, we discuss the benchmark measurements of Tython, CPython and Codon. Figure 5.2 shows the mean execution times for each benchmark under the three different compilers.

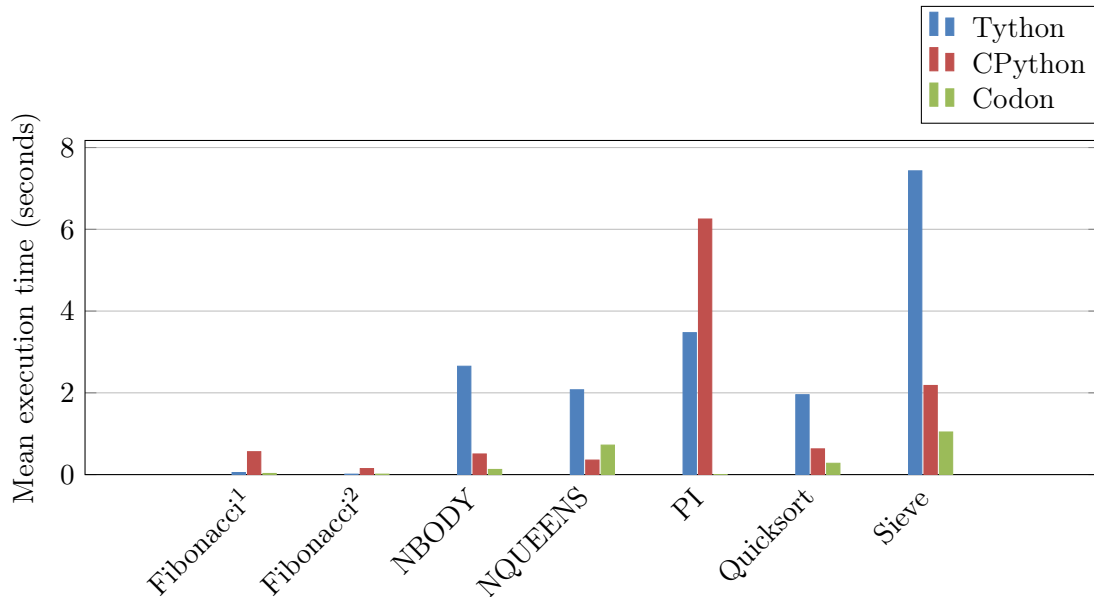


FIGURE 5.2: The mean execution times for each benchmark under the three different compilers.

4.1 Tython

The four build configurations allow for the isolated evaluation of the performance impact to type shifting and specialization. To summarize, build configuration 1 only generates object values. Since $\forall v \in \text{Spec}.\tau(v) = \text{Object}$ no specialization of operations on scalar types takes place at runtime, although the full type guards are generated for all uses of all values. All operations on objects are delegated to the object execution model. Configuration 2 maintains the same invariant, generating only object values. However, it does not generate type guards at all. All operations on values are delegated to the object execution model without an unnecessary first-layer type check. Configuration 3 applies specialization of values where possible. It generates type guards for all uses of all values, and does not apply any type guard reduction. Similarly, Configuration 4 also applies specialization of values where possible, and generates type guards for all uses of all values. However, configuration 4 also applies type guard reduction.

Configuration 1 simulates a worst-case scenario where all type guards result in a delegation to the object execution model for further type checking. Comparing build configurations 1 and 2 shows the potential overhead incurred by type guards, Tython’s first layer of type checking. The measured overhead puts an upper bound on the potential performance increase to be gained from the reduction of type checks. Table 5.2 shows this overhead per benchmark. The mean value is 10.9%.

Comparing configurations 3 and 4 give us an intuition on how successfully Tython can reduce type checks. Since the reduction of dynamic type checks is the result of general optimizations, this comparison can be overly optimistic in describing the performance impact of type check reduction. This comparison therefore puts an upper bound on the efficacy of Tython’s type check reduction. Table 5.2 shows the performance increase per benchmark, yielding a mean value of 9.9%. For the benchmarks NBODY and NQueens, the impact of type shifting through general optimizations is negligible (-0.2% and 0.1% respectively).

The maximum potential performance impact of type guard reduction is at most the

relative comparison of configurations 1 and 2. At the same time, the performance impact of Tython's implementation of type guard reduction through general optimizations is at most the relative comparison between configuration 3 and 4. For the measurements taken in [Chapter 5 Section 2](#), the mean maximum potential performance improvement of type guard reduction is 10.9%. The optimistic estimate of the performance gain from Tython's implementation of type guard reduction through general optimizations reaches a mean value of 9.9%.

Lastly, since neither configuration reduces type checks, comparing build configurations 1 and 3 gives us an isolated measure of the performance impact of Tython's specialization strategy. [Table 5.2](#) gives a mean improvement of 15.4%, but this varies greatly per benchmark. The Fibonacci benchmarks both have a performance increase of over 98%. Tython is able to specialize all values and operations, completely bypassing the object execution model. Most benchmarks have a positive performance impact. However, the Sieve benchmark sees a performance decrease of 165% applying the specialization strategy. The rationale for this is that the specializing values generated under configuration 4 must be boxed before they can be used to index or be put into a list object.

The measurements and profiler data show that the clear bottleneck in performance is the creation of objects. Tython generates leaky executables which negatively affects the measurements of operations in the object execution model. However, since all build configurations we compare suffer from the same mechanism of memory leakage, the findings in this project are relevant regardless.

4.2 CPython

[Table 5.10](#) shows the benchmark results for CPython. Both Fibonacci benchmarks are faster in Tython. The recursive implementation is 91% faster, and the forward implementation is 93% faster. The PI benchmark is also faster in Tython, with a 44% performance gain. All other benchmarks are slower, with NBODY 408% slower, NQueens 404% slower, Quicksort 211% slower, and Sieve 23% slower (in the best case). We attribute the speed-up of Tython over CPython for the Fibonacci benchmark to the fact that Tython employs its specializing execution model and type shifting maximally in both, eliminating all runtime type checks and entirely circumventing the object execution model. Tython also successfully applies specialization and type shifting in the PI benchmark. Furthermore, Tython, being an ahead-of-time compiler, does not have the general overhead of an interpreter [50]. The benchmarks where Tython is slower than CPython are also those where Tython shows smaller or negative performance gains between its own build configurations. The creation of heap objects makes up a significant portion of the execution of those benchmarks. In contrast, CPython has an effective garbage collector, making object creation less of an issue. In addition, CPython employs its own bespoke optimizations to achieve its relatively fast performance.

4.3 Codon

The benchmark results for Codon are shown in [Table 5.11](#). Codon is faster than CPython on all benchmarks except NQueens, where it is 103% slower. It is 95% faster on the recursive Fibonacci benchmark, 93% faster on forward Fibonacci, 76% faster on NBODY, 56% faster on Quicksort, and 52% faster on Sieve.

These performance gains are achieved by the fact that Codon is also an ahead-of-time compiler and does not have an interpretation overhead. It applies program-wide static

	Fastest Time	Mean Time	Median Time	Max Time	Variance (s ²)
fibonacci (recursive)	0.5397	0.5598	0.5429	0.6521	0.0011
fibonacci (while)	0.1427	0.1468	0.1459	0.1545	0.0000
nbody	0.5044	0.5220	0.5219	0.5376	0.0001
nqueens	0.3461	0.3550	0.3497	0.3815	0.0001
pi	5.9917	6.2516	6.2417	6.5880	0.0325
quicksort	0.6080	0.6293	0.6245	0.6842	0.0004
sieve	2.0504	2.1801	2.1195	2.8238	0.0477

TABLE 5.10: The benchmark results for CPython (times in seconds).

type inference and does not maintain any runtime type information.

Note that Codon fails to run the PI benchmark, raising a type error that `float` does not match the expected type `int` on line 32. This is, however, valid Python code.

	Fastest Time	Mean Time	Median Time	Max Time	Variance (s ²)
fibonacci (recursive)	0.0254	0.0274	0.0273	0.0292	0.0000
fibonacci (while)	0.0099	0.0107	0.0109	0.0114	0.0000
nbody	0.1244	0.1257	0.1256	0.1273	0.0000
nqueens	0.6957	0.7213	0.7262	0.7321	0.0001
pi	0.0000	0.0000	0.0000	0.0000	0.0000
quicksort	0.2723	0.2769	0.2770	0.2824	0.0000
sieve	1.0315	1.0420	1.0389	1.0578	0.0001

TABLE 5.11: The benchmark results for Codon (compiled, times in seconds).

Chapter 6

Concluding remarks

In this section, we will answer the research questions laid out in [Chapter 6 Section 2](#). The project comprises an abstract compiler design and a concrete implementation for evaluation. In [chapter 3](#), the design describes an execution model for a specializing Python compiler, with a method for type-shifting Python’s dynamic type system towards a static one at compile time. In [chapter 4](#) we introduce Tython, which implements this design using LLVM Core for code generation, a tagged union for the specialization data structure, a C runtime library for the object execution model and a composition of general LLVM optimizations to achieve emergent type shifting. [chapter 5](#) evaluates Tython, quantifying the performance impact of the compiler design.

This project has put forward the following research questions:

1. What static type information can be inferred from a Python program without type annotations?
2. How can we propagate static type information to reduce dynamic type checks?
3. How can we specialize runtime values to circumvent the object execution model?
4. What is the performance impact of the reduction of dynamic type checks and the specialization of runtime values?

1 What static type information can be inferred from a Python program without type annotations?

The main characteristic of a dynamic language is that types are associated with values, not variables. To obtain static typing, where types *are* associated with variables, we will need to propagate the known type information of values to variables.

[Chapter 6 Section 2.3](#) describes a method for achieving this. [Definition 3](#) says that if all write operations to a variable α are statically known, and all write operands are of the same type t , we can statically determine the type of α to be t .

To achieve this, we must find all write operations to a variable at compile-time. Tython achieves this with early binding ([Chapter 6 Section 3.1](#)). Early binding provides compile-time access to all uses of a variable.

We must then also statically know the types of the operands of these write operations. [Chapter 6 Section 4.1](#) shows that static type information is known for source-level literals. These are atomic expressions, so their type is unambiguous. The implementation

furthermore shows that for some expressions composed of operands with known types, the resulting type of evaluating the expression can also statically be determined.

If for a variable α all write operations are statically known, and all those write operations are literals or expressions whose type can be statically determined we can use [Definition 3](#) to infer the type of a variable α : $\forall w \in W. \tau(w) = t \implies \text{type}(\alpha) = t$.

2 How can we propagate static type information to reduce dynamic type checks?

The first research question shows how static type information can be obtained from Python source code. [Chapter 6 Section 1](#) shows that Tython contains two layers of dynamic type information. The first layer distinguishes the two scalar specialization types *Integer* and *FloatingPoint*, and the general *Object* type. [Chapter 6 Section 2.2](#) describes type guards, which are the dynamic type checks for this first layer. [Chapter 6 Section 1.2](#) shows how type guards are implemented in Tython.

The second layer of dynamic type information is handled by delegating to the object execution model. The design of this domain is entirely dynamically typed, and so no static type information can be propagated into it. We can therefore only reduce dynamic type checks for the first layer.

Tython implements type guards in the internal execution model described in [Chapter 6 Section 1.1](#). The reduction of these type guards is described in [Chapter 6 Section 4](#) and its implementation in Tython is shown in [Chapter 6 Section 4.2](#). Tython achieves type guard reduction through emergent behaviour as a result of the application of three general LLVM optimizations. *Scalar Replacement of Aggregates* (SROA) effectively propagates the static type information associated with variables. If successful, this makes a type guard a conditional branch on the comparison of two constants (the static type of a guarded variable α and the type of a *tag*) for each $\text{tag} \in \text{Tag}$. *Sparse Conditional Constant Propagation* (SCCP) removes these trivial conditional branching instructions, leaving one unconditional branch and two unreachable CFG nodes. Lastly, *Simplify the CFG* (*simplifycfg*) deletes the unreachable CFG nodes and merges the now unconditional branch, where $\text{type}(\alpha) = \text{tag}$, into straight-line code.

The successful application of these three transformations removes a type guard entirely.

3 How can we specialize runtime values to circumvent the object execution model?

[Chapter 6 Section 2](#) describes a data and execution model that can specialize on the scalar types *Integer* and *FloatingPoint*. Tython implements the *Spec* data model as a tagged union ([Chapter 6 Section 1.1](#)). All values in the internal execution model are of the form *Spec*, such that $\forall v \in \text{Spec}. \tau(v) \in \text{Tag}$. This invariant allows us the generation of exhaustive type guards at compile time. Being able to distinguish between values of type *Integer*, *FloatingPoint*, and *Object* enables Tython's first layer of dynamic typing.

The content of a value is obtained by applying the function μ . Tython's tagged union implementation stores the content of all values in a address-wide bitstring. The interpretation of this bitstring depends on the value's tag, and is at compile time encoded in each of the type guard branches by the partial application of the function π . If an operation Ω is a specializing operation, the content of a value of type *Integer* or *FloatingPoint* can

be operated on directly by the appropriate specialization Ω_{int} or Ω_{fp} respectively. This happens without any delegation to the object execution model.

4 What is the performance impact of the reduction of dynamic type checks and the specialization of runtime values?

In [Chapter 6 Section 2](#) we hypothesize that the reduction of runtime type checks and the specialization of values leads to performance improvements. The metric of interest is execution time. [chapter 5](#) evaluates the performance of benchmark executables generated by the Tython implementation under different build configurations. We have designed the 4 different build configurations Tython to give us an isolated insight into the relative performance impact of compiler design choices.

The impact of type shifting is found in the comparison between configurations 1 and 2, and between configurations 3 and 4.

The results in [Chapter 6 Section 2](#) show that the reduction of type guards has a positive potential impact on all benchmarks, with a mean performance gain of 10.9%. The application of general optimizations to achieve type shifting has a positive impact for most benchmarks, with a mean value of 9.9%. This latter value is likely an overestimation, since the application of general optimizations likely has more effects than only reducing type guards.

The impact of specialization is found in the comparison between configurations 1 and 3. The mean performance impact is positive, with a performance gain of 15.4%. However, this impact varies greatly depending on the benchmark, and is even shown to have a large negative impact of -165.4% in one particular benchmark. The greatest performance gain is 98.5%, where Tython is able to entirely circumvent the object execution model.

5 Future work

One of the most obvious missing language features is user-defined classes. The omission of it in this thesis is related to the limited scope of the project. It could warrant a separate study. Tython may be a good platform to start this experimentation from.

Tython still uses late binding (the lookup of symbols at runtime) for object methods. A next version can look into the early binding of methods and object attributes, which has the potential to further improve performance.

Specialization on more types could also result in better performance. In particular the inability to specialize on collection access (such as lists, tuples and dictionaries) creates a significant overhead in Tython. All specialization values are boxed when delegated to the object execution model. Bringing these collection objects into the internal execution model has the potential to drastically reduce this boxing overhead. Furthermore, Python collection types are heterogeneous, making static type inference hard or impossible. A common solution for this problem is the introduction of a dynamic (or `any`) type, as implemented in for instance Reticulated Python [\[47\]](#) and TypeScript [\[1\]](#). This making a heterogeneous collection such as a list of type `List<Any>`.

Currently, the general optimizations that Tython applies to achieve type shifting are local to functions. This means that static type information is never propagated across the function boundary. Future work could investigate the possibility of monomorphization (as

applied by Codon) or the "seeding" of functions with static type information for the LLVM optimization passes to pick up on.

There is also potential in supporting gradual typing through Python type annotations. Similar to the idea of "seeding" functions, these annotations could serve as a way to nudge type shifting in the right direction.

Lastly, future efforts could look into allowing a programmer to choose when to give up Pythonic semantics for performance. Currently, if specialization is enabled, Tython aggressively tries to specialize values where it can. Having user-defined symbols or regions of specialization (or inversely, symbols or regions of safe semantics) could benefit the adoption of Tython.

Bibliography

- [1] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 257–281, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. doi:10.1007/978-3-662-44202-9_11.
- [2] David Boddie. Python Implementations. URL: <https://wiki.python.org/moin/PythonImplementations>.
- [3] Clang. Clang C Language Family Frontend for LLVM. URL: <https://clang.llvm.org/>.
- [4] cppreference.com. constexpr specifier. URL: <https://en.cppreference.com/w/cpp/language/constexpr>.
- [5] Python Software Foundation. Compiler design. URL: <https://github.com/python/cpython/blob/main/InternalDocs/compiler.md>.
- [6] Python Software Foundation. CPython – The Python programming language reference implementation. URL: <https://github.com/python/cpython>.
- [7] Python Software Foundation. Python 3.12.2 documentation – Data model. URL: <https://docs.python.org/3/reference/datamodel.html#objects-values-and-types>.
- [8] Python Software Foundation. The Python Type System. URL: <https://typing.readthedocs.io/en/latest/spec/type-system.html>.
- [9] Python Software Foundation. Why am I getting an UnboundLocalError when the variable has a value? URL: <https://docs.python.org/3/faq/programming.html#why-am-i-getting-an-unboundlocalerror-when-the-variable-has-a-value>.
- [10] Pablo Galindo Salgado. What’s New In Python 3.11, 2022. URL: <https://docs.python.org/3.11/whatsnew/3.11.html#faster-cpython>.
- [11] GeeksforGeeks. Python Program for N Queen Problem. URL: <https://www.geeksforgeeks.org/python-program-for-n-queen-problem-backtracking-3/>.
- [12] GeeksforGeeks. Python Program for QuickSort. URL: <https://www.geeksforgeeks.org/python-program-for-quicksort/>.
- [13] GeeksforGeeks. Python Program for Sieve of Eratosthenes. URL: <https://www.geeksforgeeks.org/python-program-for-sieve-of-eratosthenes/>.
- [14] Sanjay Ghemawat. Gperftools CPU Profiler, 2008. URL: <https://gperftools.github.io/gperftools/cpuprofile.html>.

- [15] Hristina Gulabovska and Zoltán Porkoláb. Survey on Static Analysis Tools of Python Programs. In Z. Budimac and B. Koteska, editors, *SQAMIA 2019: 8th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications*, 2019. URL: <https://api.semanticscholar.org/CorpusID:209149918>.
- [16] Meta Incubator. Cinder, Meta’s internal performance-oriented production version of CPython. URL: <https://github.com/facebookincubator/cinder>.
- [17] JetBrains. C Programming – The State of Developer Ecosystem in 2022 Infographic, 2022. URL: <https://www.jetbrains.com/lp/devecosystem-2022/c/>.
- [18] JetBrains. Python Developers Survey, 2022. URL: <https://lp.jetbrains.com/python-developers-survey-2022/>.
- [19] Linda Torczon Keith Cooper. *Engineering A Compiler*. Morgan-Kaufmann, 2 edition, 2012.
- [20] Faizan Khan, Boqi Chen, Daniel Varro, and Shane McIntosh. An Empirical Study of Type-Related Defects in Python Projects. *IEEE Transactions on Software Engineering*, 48:3145–3158, August 2022. doi:10.1109/TSE.2021.3082068.
- [21] David Lion, Adrian Chiu, Michael Stumm, and Ding Yuan. Investigating Managed Language Runtime Performance: Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be Faster? In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 835–852, Carlsbad, CA, July 2022. USENIX Association. URL: <https://www.usenix.org/conference/atc22/presentation/lion>.
- [22] Kuang-Chen Lu, Ben Greenman, Carl Meyer, Dino Viehland, Aniket Panse, and Shriram Krishnamurthi. Gradual Soundness: Lessons from Static Python. *Art, Science, and Engineering of Programming*, 7, June 2022. doi:10.22152/programming-journal.org/2023/7/2.
- [23] Ben Lynn. Computing Pi in C. URL: <https://crypto.stanford.edu/pbc/notes/pi/code.html>.
- [24] Simone Martini Maurizio Gabbrielli. *Programming Languages: Principles and Paradigms*. Undergraduate Topics in Computer Science. Springer-Verlag London, 1 edition, 2010.
- [25] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, December 1978. doi:10.1016/0022-0000(78)90014-4.
- [26] John C Mitchell. *Foundations for programming languages*, volume 1. MIT press Cambridge, 1996.
- [27] Manzi Aimé Ntagengerwa. Tython – An AoT-compiled Python implementation. URL: <https://github.com/EppeMedia/tython>.
- [28] Manzi Aimé Ntagengerwa. Tython v0.1 source code. doi:10.5281/zenodo.13366853.
- [29] Yun Peng, Yu Zhang, and Mingzhe Hu. An Empirical Study for Common Language Features Used in Python Projects. *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 24–35, 2021. doi:10.1109/SANER50967.2021.00012.

-
- [30] LLVM Project. LLVM 19.0.0 documentation. URL: <https://www.llvm.org/docs/>.
- [31] LLVM Project. LLVM’s Analysis and Transform Passes. URL: <https://llvm.org/docs/Passes.html>.
- [32] LLVM Project. The LLVM Compiler Infrastructure. URL: <https://llvm.org/>.
- [33] The Mypy Project. Mypy – Optional Static Typing for Python, 2014. URL: <https://mypy-lang.org/>.
- [34] Fabrice Rastello and Florent Bouchez Tichadou. *SSA-based Compiler Design*. Springer Cham, 1 edition, 2022. doi:10.1007/978-3-030-80515-9.
- [35] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. *SIGPLAN Not.*, 47(1):481–494, January 2012. doi:10.1145/2103621.2103714.
- [36] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe and efficient gradual typing for TypeScript. *ACM SIGPLAN Notices*, 50:167–180, January 2015. doi:10.1145/2676726.2676971.
- [37] Ariya Shajii, Gabriel Ramirez, Haris Smajlović, Jessica Ray, Bonnie Berger, Saman Amarasinghe, and Ibrahim Numanagić. Codon: A Compiler for High-Performance Pythonic Applications and DSLs. *CC 2023 - Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, pages 191–202, February 2023. doi:10.1145/3578360.3580275.
- [38] Mark Shannon. PEP 659 – Specializing Adaptive Interpreter, 2021. URL: <https://peps.python.org/pep-0659/>.
- [39] Jeremy Siek and Walid Taha. Gradual Typing for Functional Languages. In *Proceedings of the Scheme and Functional Programming Workshop*, January 2006. URL: <https://api.semanticscholar.org/CorpusID:1398902>.
- [40] Victor Stinner. The Python Performance Benchmark Suite, 2017. URL: <https://pyperformance.readthedocs.io/>.
- [41] Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in JavaScript. *ACM SIGPLAN Notices*, 49(1):425–437, January 2014. doi:10.1145/2578855.2535889.
- [42] TIOBE. TIOBE Index, 2024. URL: <https://www.tiobe.com/tiobe-index/>.
- [43] Guido van Rossum. The History of Python: A Brief Timeline of Python, 2009. URL: <https://python-history.blogspot.com/2009/01/brief-timeline-of-python.html>.
- [44] Guido van Rossum and The Python development team. The Python Language Reference Release 3.12.3, 2024. URL: <https://fossies.org/linux/python-docs-pdf-a4/reference.pdf>.
- [45] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. PEP 484 – Type Hints, 2014. URL: <https://peps.python.org/pep-0484/>.

-
- [46] David Vandevorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 1 edition, 2002.
 - [47] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for Python. *ACM SIGPLAN Notices*, 50:45–56, February 2015. doi:10.1145/2661088.2661101.
 - [48] David Watson. JavaScript and TypeScript Trends 2024: Insights From the Developer Ecosystem Survey, February 2024.
 - [49] Collin Winter, Jeffrey Yasskin, and Reid Kleckner. PEP 3146 – Merging Unladen Swallow into CPython, 2010. URL: <https://peps.python.org/pep-3146/>.
 - [50] Qiang Zhang, Lei Xu, Xiangyu Zhang, and Baowen Xu. Quantifying the interpretation overhead of Python. *Science of Computer Programming*, 215:102759, March 2022. doi:10.1016/J.SCIC0.2021.102759.

Annex

Annex A

Benchmark profiler measurements

build/bin/fibonacci_while

Total samples: 192

Focusing on: 192

Dropped nodes with ≤ 0 abs(samples)

Dropped edges with ≤ 0 samples

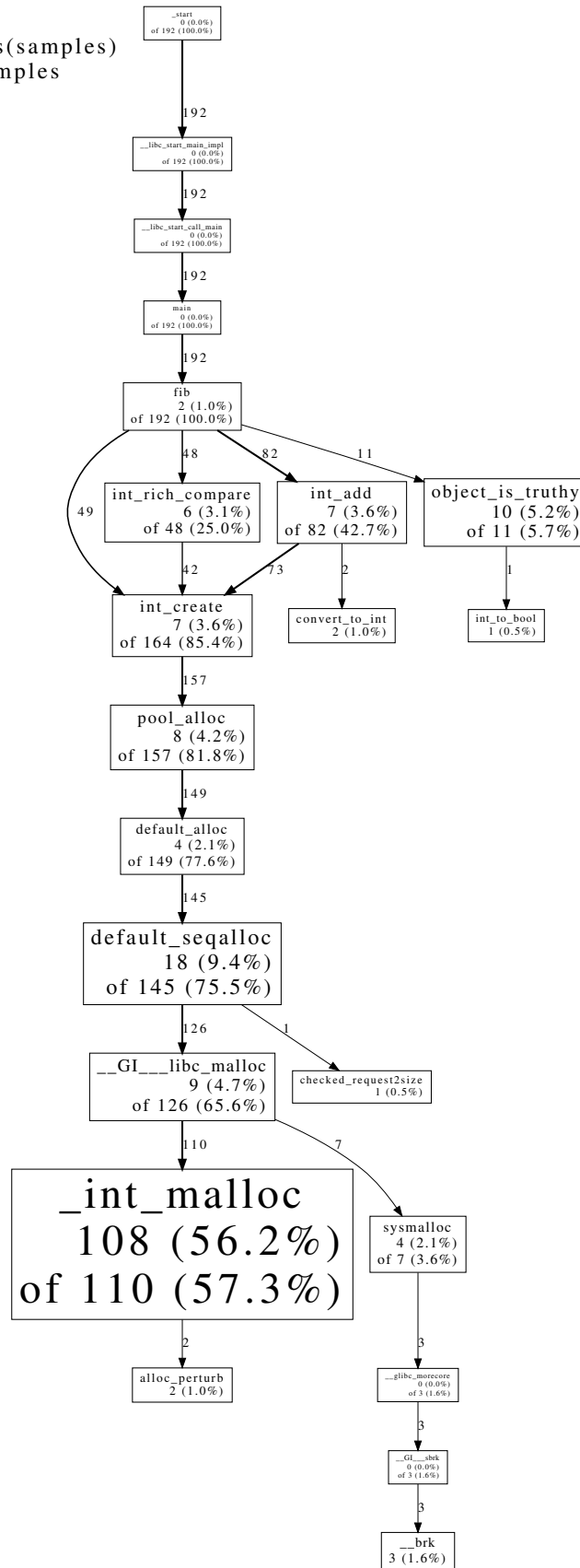


FIGURE A.1: The profiler output of the forward Fibonacci benchmark under configuration 1.

build/bin/fibonacci_rec
 Total samples: 1139
 Focusing on: 1139
 Dropped nodes with ≤ 5 abs(samples)
 Dropped edges with ≤ 1 samples

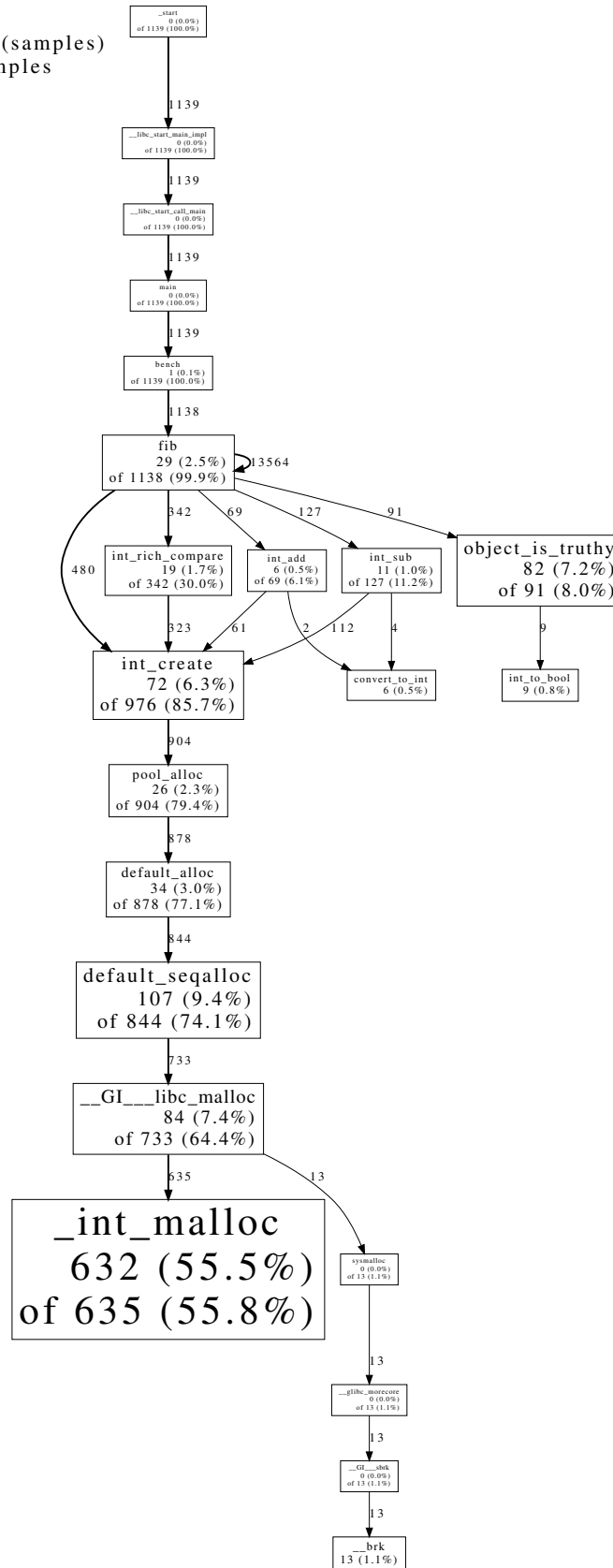


FIGURE A.2: The profiler output of the recursive Fibonacci benchmark under configuration 1.

build/bin/nbody
 Total samples: 815
 Focusing on: 815
 Dropped nodes with ≤ 4 abs(samples)
 Dropped edges with ≤ 0 samples

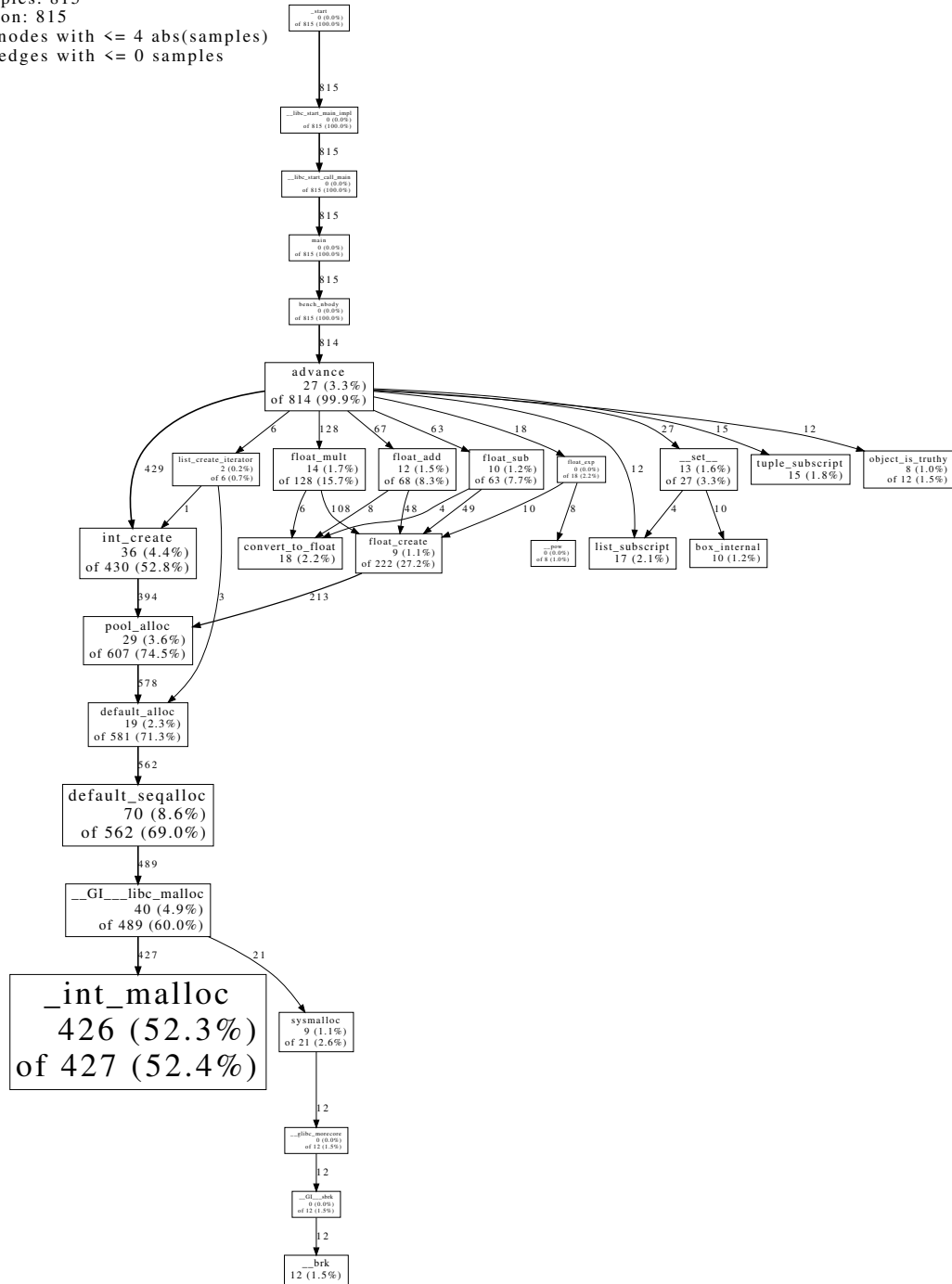


FIGURE A.3: The profiler output of the NBODY benchmark under configuration 1.

build/bin/nbody
 Total samples: 700
 Focusing on: 700
 Dropped nodes with <= 3 abs(samples)
 Dropped edges with <= 0 samples

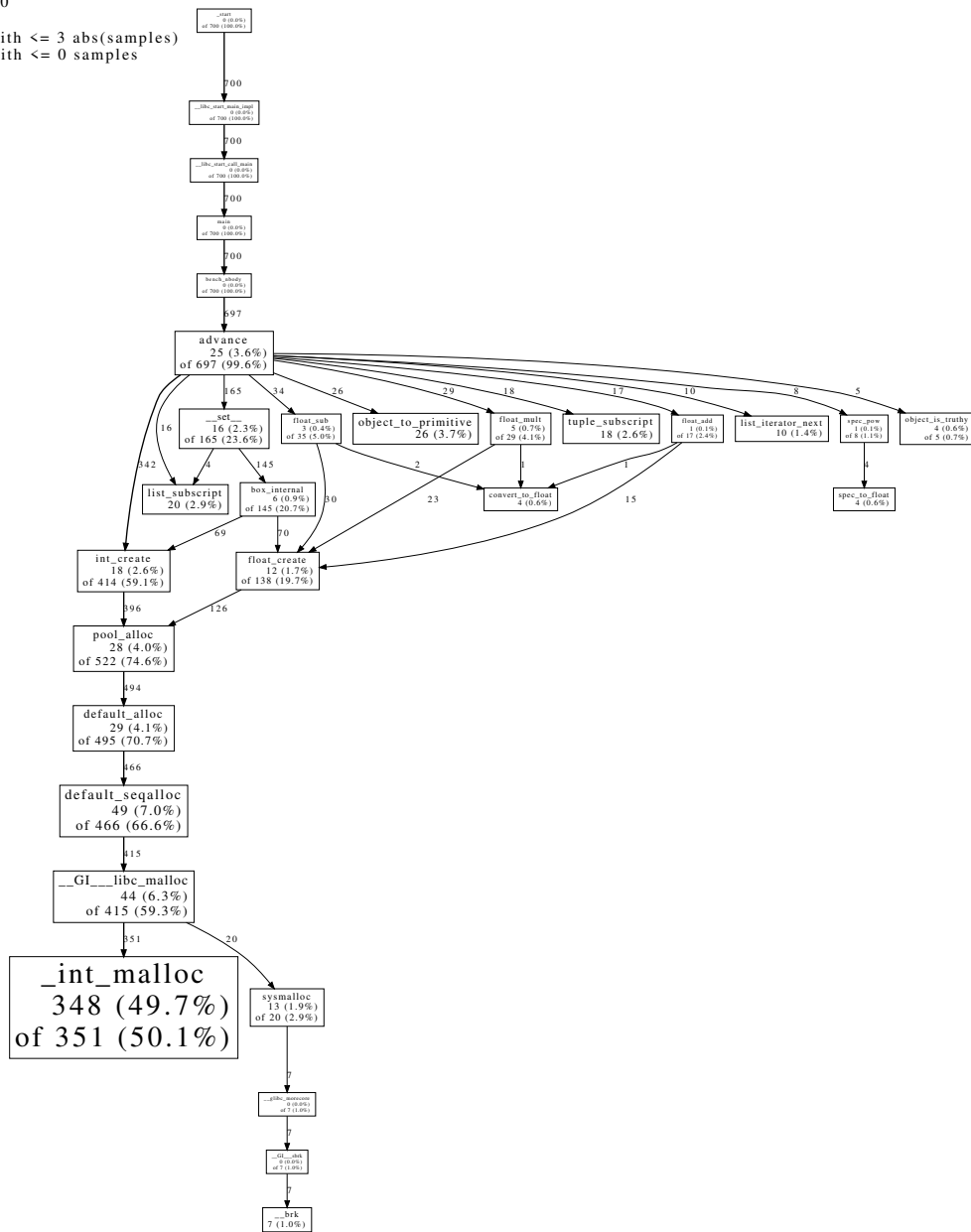


FIGURE A.4: The profiler output of the NBODY benchmark under configuration 4.

build/bin/pi
 Total samples: 952
 Focusing on: 952
 Dropped nodes with ≤ 4 abs(samples)
 Dropped edges with ≤ 0 samples

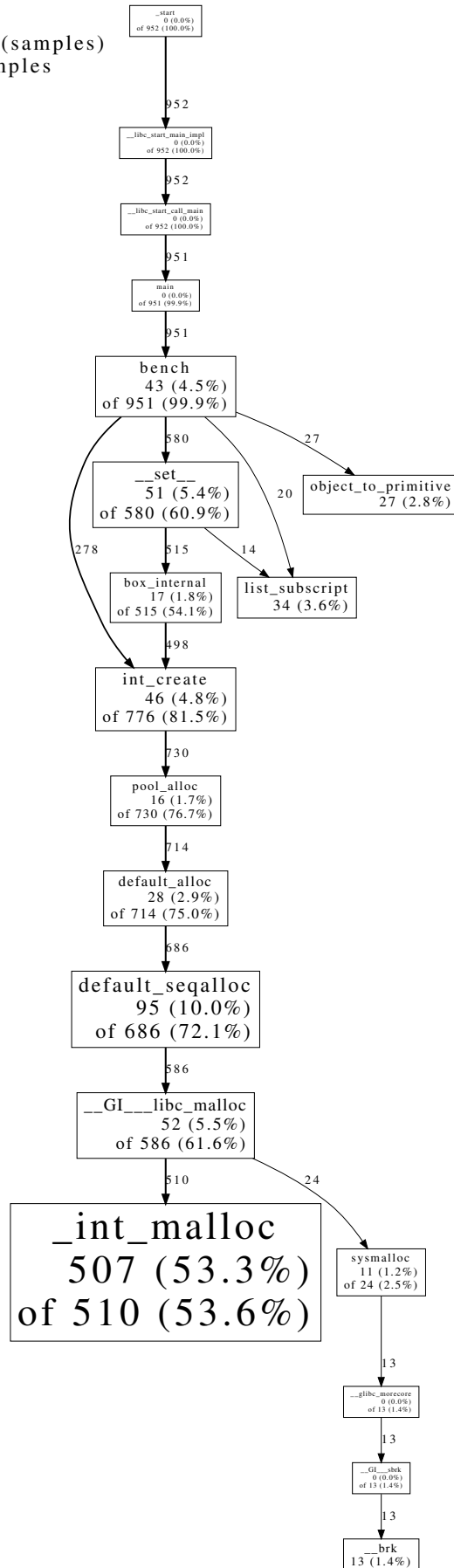


FIGURE A.5: The profiler output of the PI benchmark under configuration 4.

build/bin/sieve
 Total samples: 776
 Focusing on: 776
 Dropped nodes with ≤ 3 abs(samples)
 Dropped edges with ≤ 0 samples

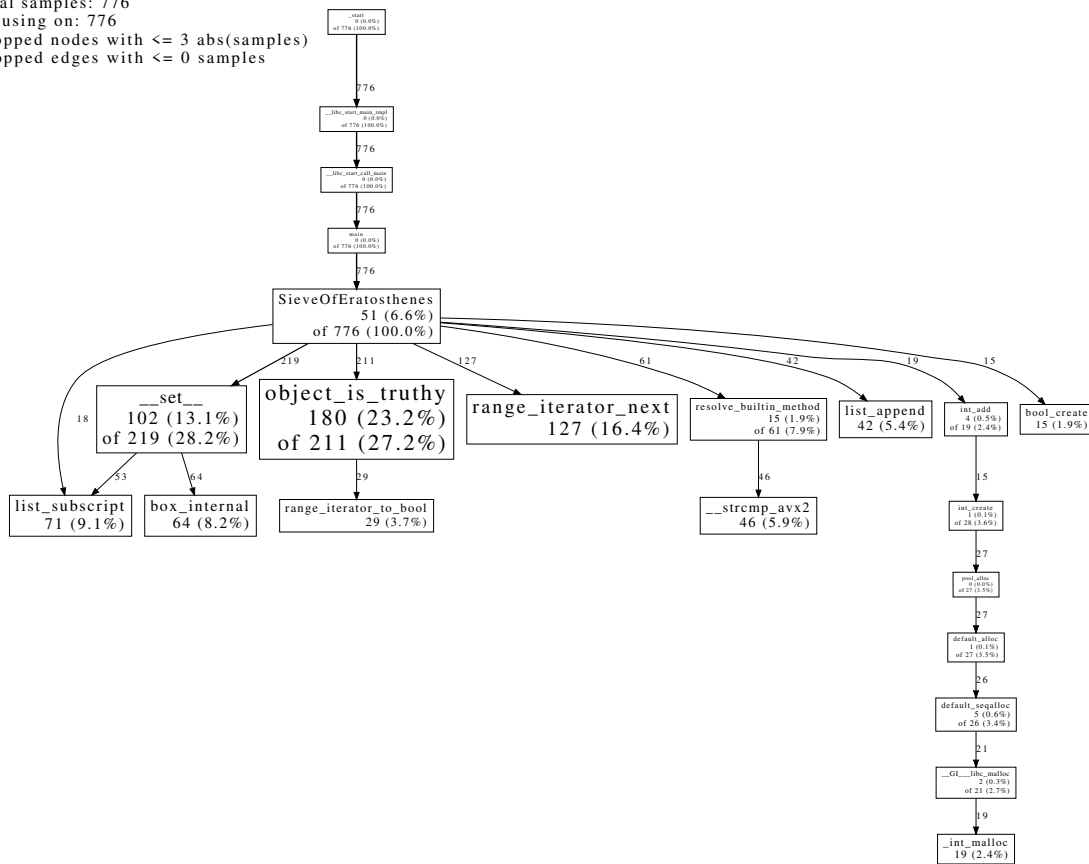


FIGURE A.6: The profiler output of the Sieve benchmark under configuration 1.

build/bin/sieve
 Total samples: 1914
 Focusing on: 1914
 Dropped nodes with <= 9 abs(samples)
 Dropped edges with <= 1 samples

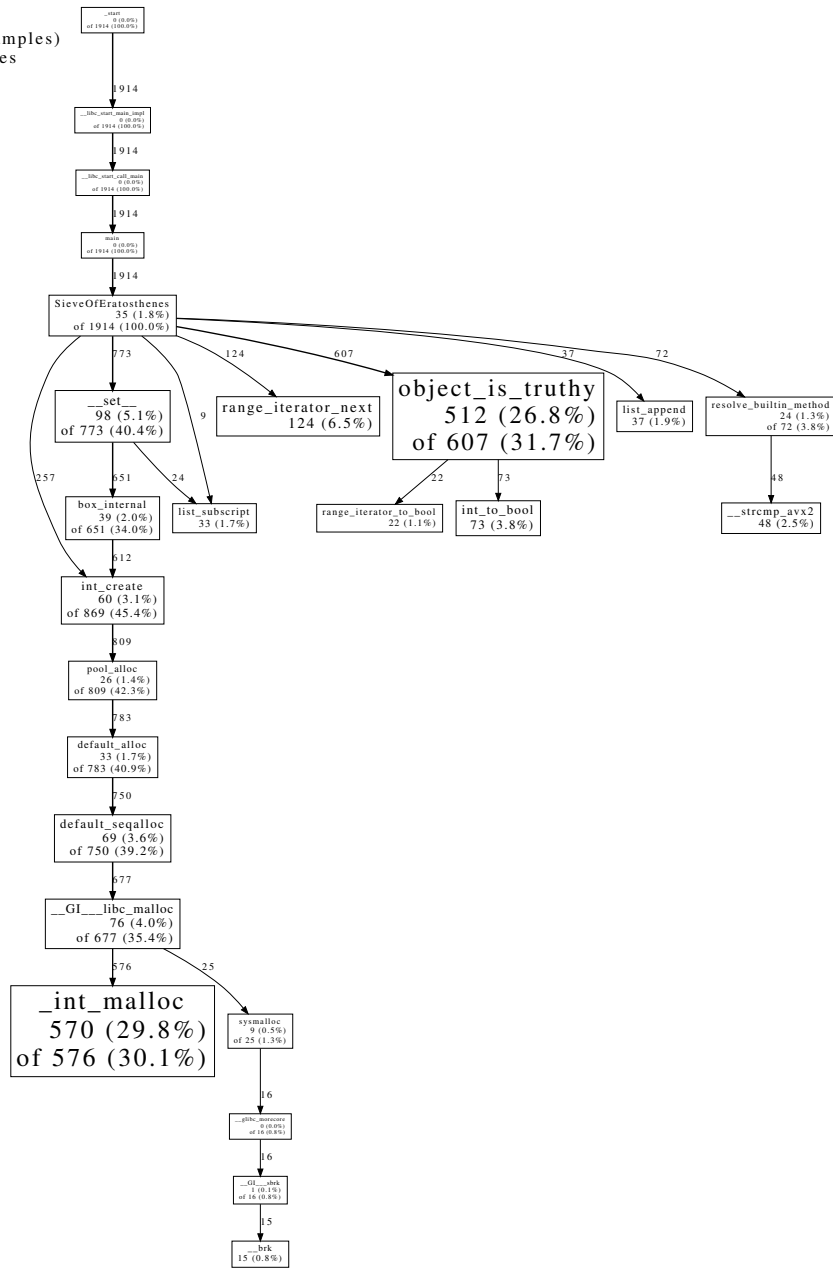


FIGURE A.7: The profiler output of the Sieve benchmark under configuration 4.

build/bin/quicksort
 Total samples: 624
 Focusing on: 624
 Dropped nodes with <= 3 abs(samples)
 Dropped edges with <= 0 samples

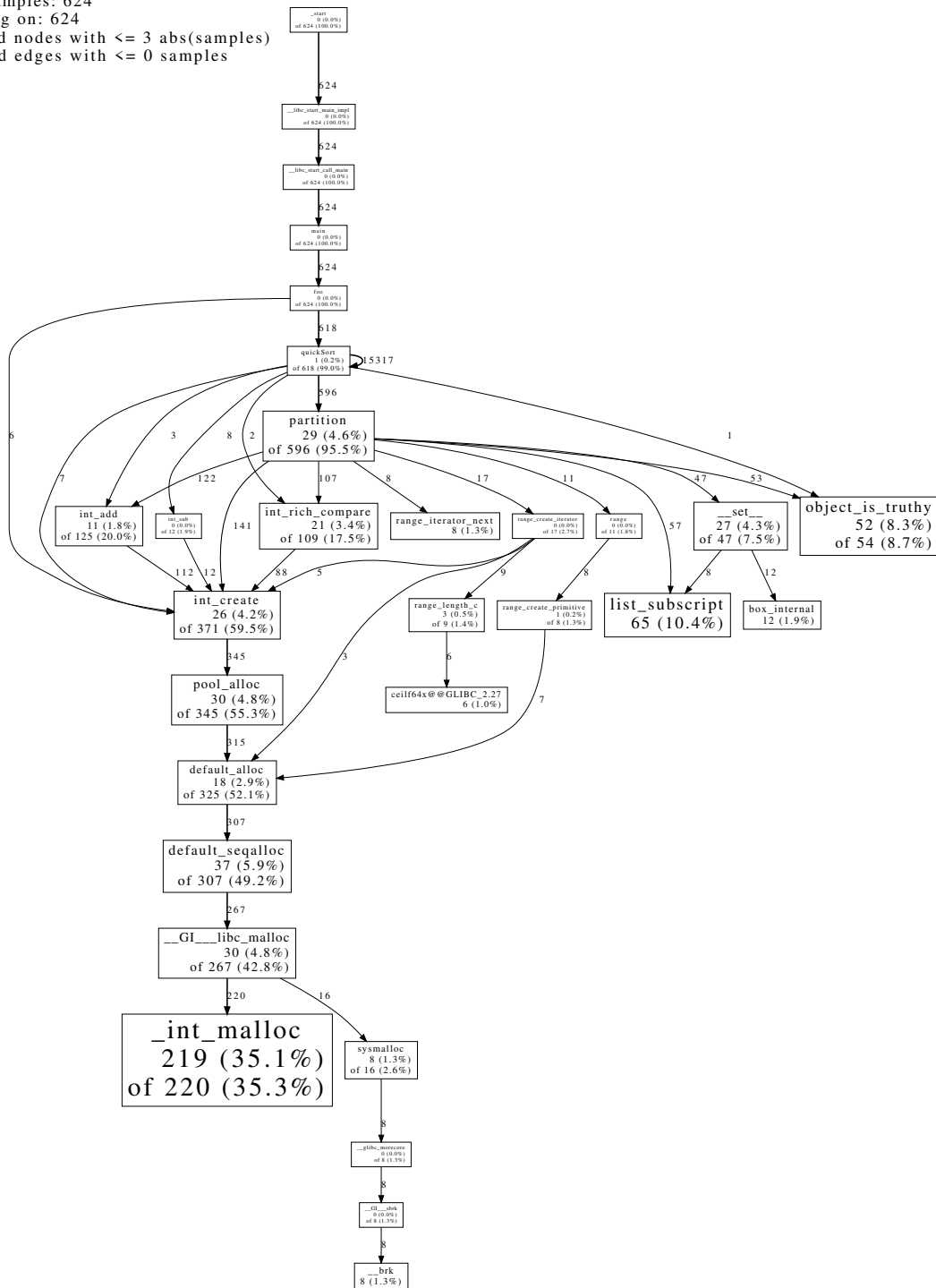


FIGURE A.8: The profiler output of the Quicksort benchmark under configuration 1.

build/bin/quicksort
 Total samples: 550
 Focusing on: 550
 Dropped nodes with ≤ 2 abs(samples)
 Dropped edges with ≤ 0 samples

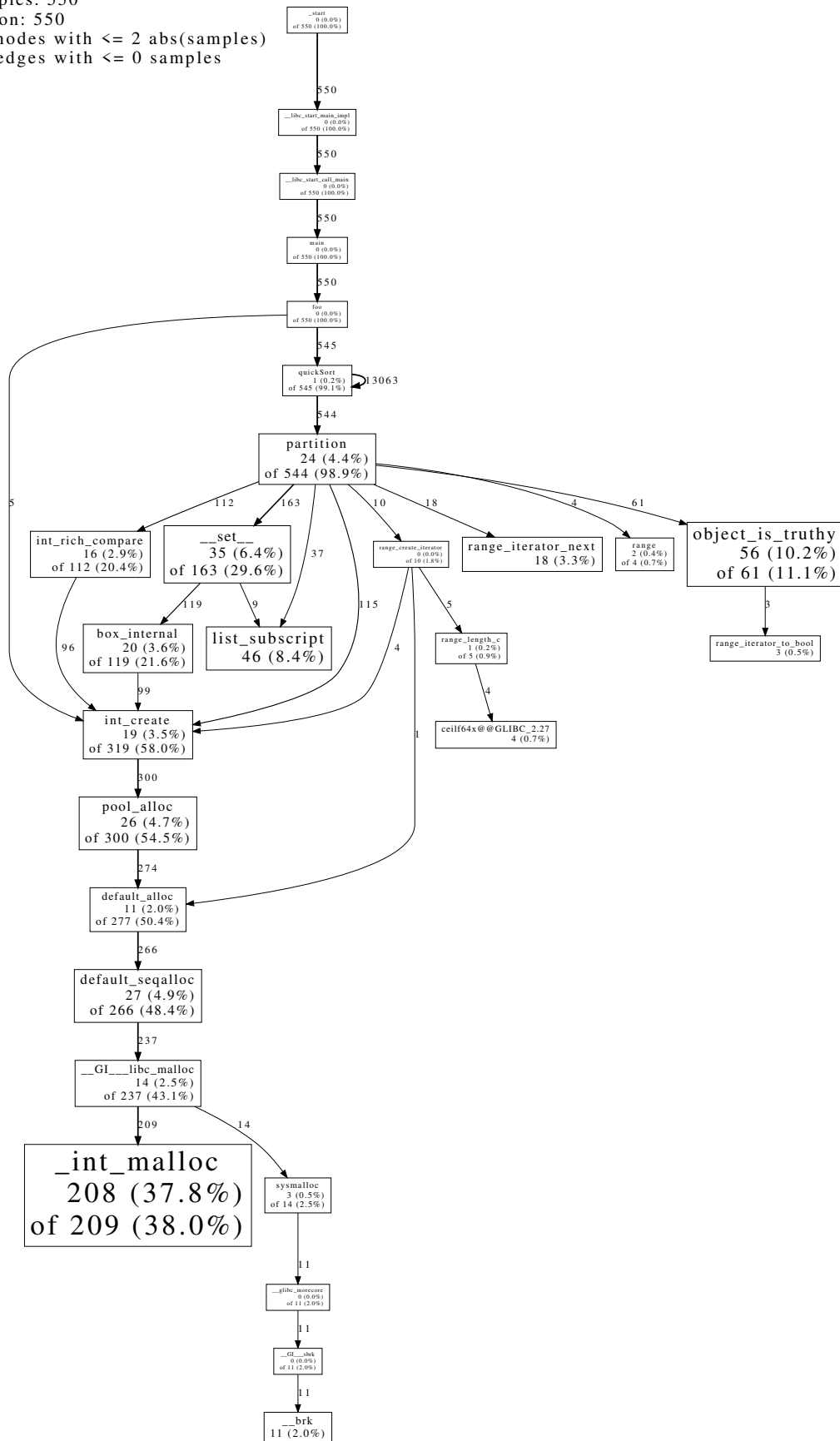


FIGURE A.9: The profiler output of the Quicksort benchmark under configuration 4.

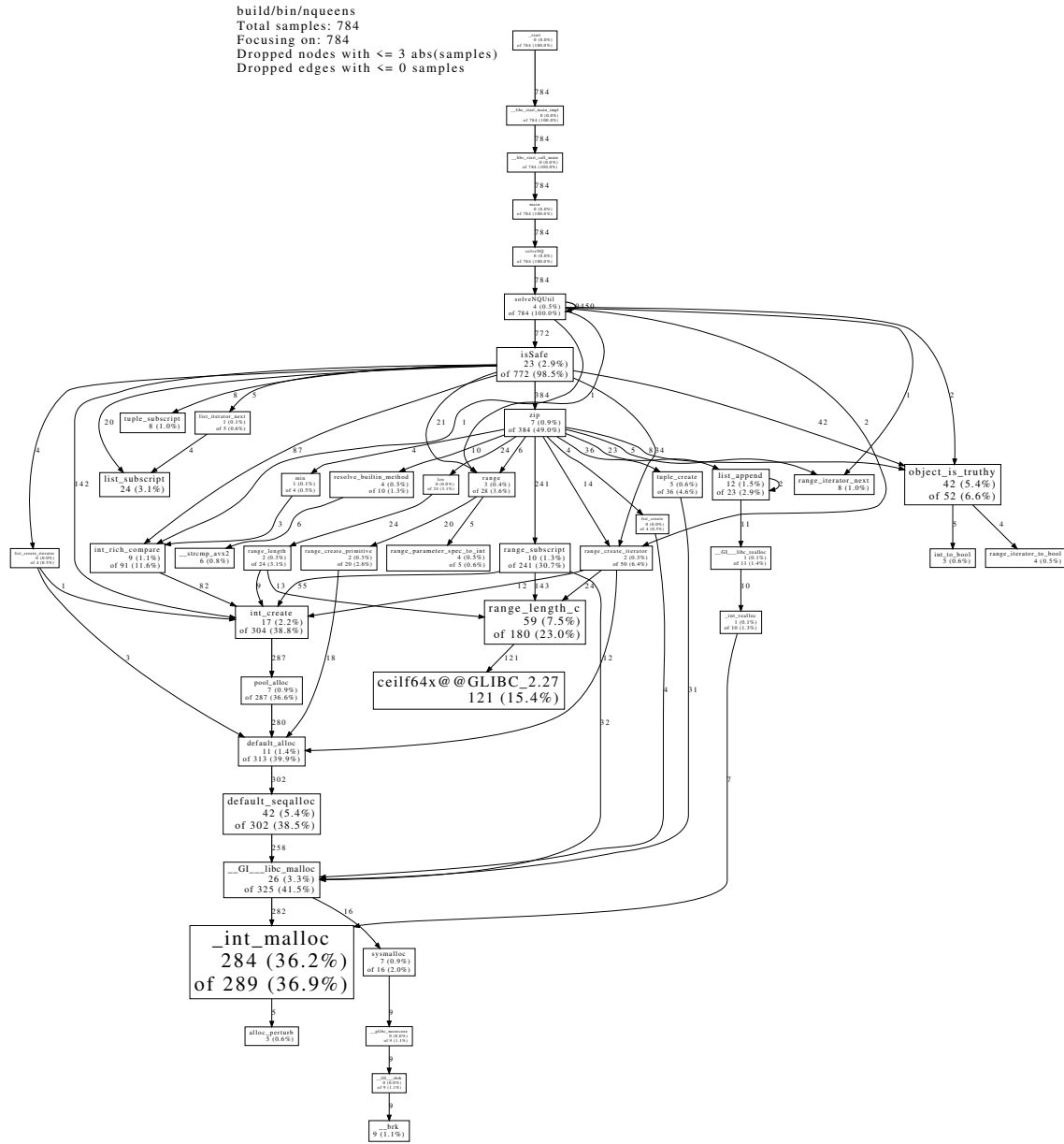


FIGURE A.10: The profiler output of the NQueens benchmark under configuration 1.

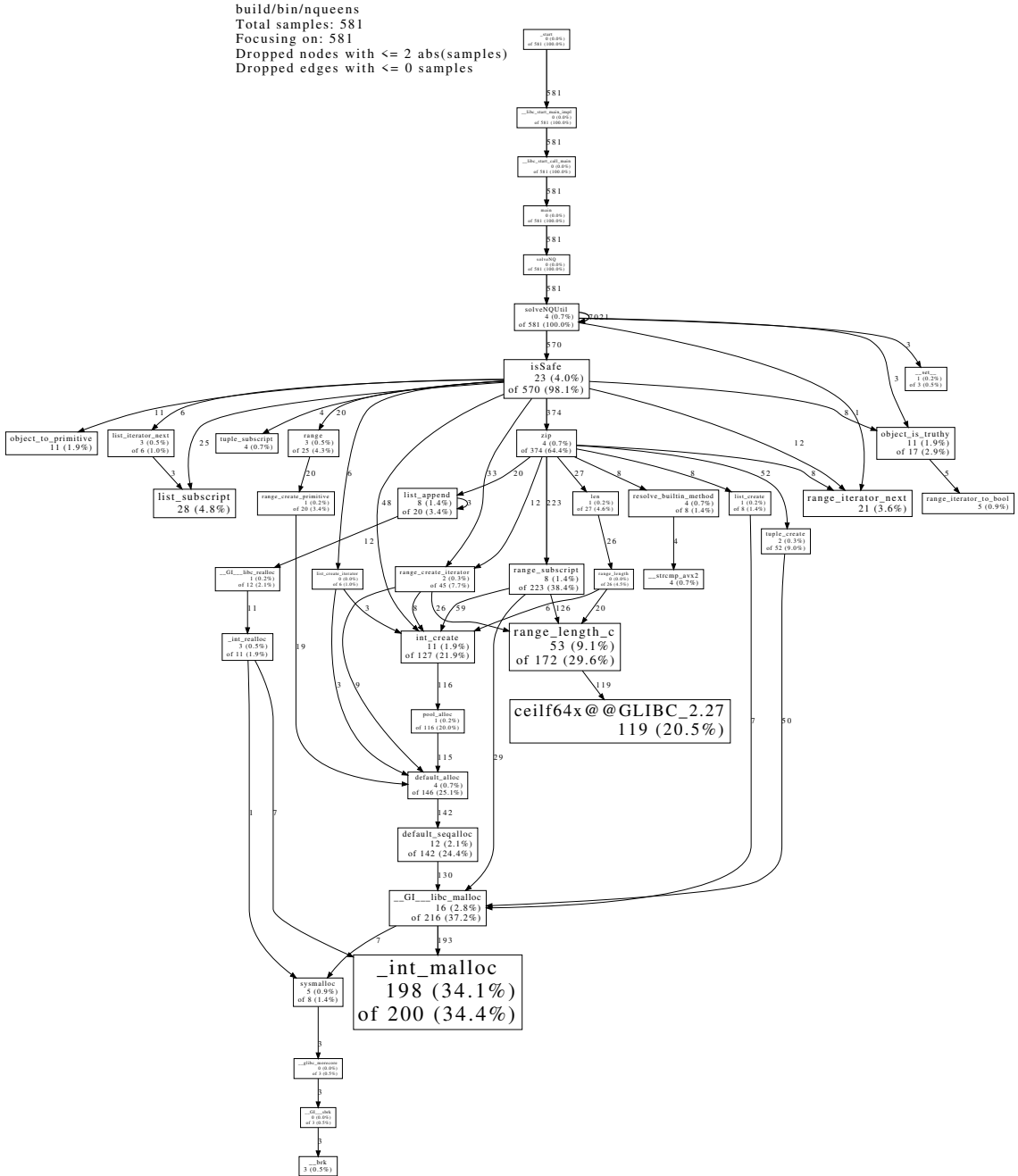


FIGURE A.11: The profiler output of the NQueens benchmark under configuration 4.