BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

IMPROVING THE GRAALPY INTERPRETER

ZLEPŠOVÁNÍ INTERPRETU GRAALPY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR — **ADAM RUDOLF HRBÁČ**
AUTOR PRÁCE

SUPERVISOR — **Ing. DAVID KOZÁK**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

154280

Institut:	Department of Intelligent Systems (DITS)
Student:	**Hrbáč Adam**
Programme:	Information Technology
Title:	**Improving the GraalPy interpreter**
Category:	Compiler Construction
Academic year:	2023/24

Assignment:

1. Familiarize yourself with the GraalVM platform, the Truffle language implementation framework, and specifically GraalPy, a Python interpreter developed using Truffle.
2. Gain an understanding of GraalPy and its compatibility with CPython, which serves as the reference implementation of Python.
3. Propose enhancements to the GraalPy interpreter, with a focus on improving the compatibility and stability of the debugger, as well as advancing asynchronous programming capabilities. Implement the suggested changes accordingly.
4. Evaluate the implemented modifications using a suitable set of benchmarks.
5. Describe and discuss the attained results, along with potential avenues for further improvement.

Literature:

- Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software (Onward! 2013). Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581
- Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. In Proceedings of the 8th symposium on Dynamic languages (DLS '12). Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/2384577.2384587
- Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical partial evaluation for high-performance dynamic language runtimes. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017). Association for Computing Machinery, New York, NY, USA, 662–676. https://doi.org/10.1145/3062341.3062381

Requirements for the semestral defence:
The first two points of the assignment and at least some initial work on the third point.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Kozák David, Ing.**
Consultant:	Vojnar Tomáš, prof. Ing., Ph.D.
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	6.11.2023

Abstract

GraalPy is a Python implementation for the Java Virtual Machine, designed for easy embedding into Java applications. Such embedding is primarily useful for adopting 3rd party Python packages in existing Java codebases. GraalPy also often has greater performance compared to CPython, the canonical implementation of Python. This work implements two major features. First, the async API, one of the last major missing Python features in GraalPy, used primarily for web development, allowing writing concurrent code without parallelism, using so-called colored async, where each context switch point must be explicitly annotated. It is composed of two major parts, a library providing an event loop, asyncio in this work, as well as the syntactic components of Python, providing the way with which to indicate context switches. The second feature is the tracing API, a CPython API for implementing Python debuggers, used by integrated debuggers in IDEs, coverage tools, etc. It works by analyzing the Python bytecode in order to determine whether a new line is being executed, and if so, invokes a registered callback. This callback is also used when returning a value, calling a function and raising an exception, allowing a debugger to set a breakpoint for these events. Both features are part of the GraalPy releases and have had a notable benefit to compatibility with 3rd party packages.

Abstrakt

GraalPy je implementace jazyka Python pro Java Virtual Machine (JVM), určená pro její vkládání do Java aplikací. Do existujícího kódu Javy lze takto přidávat Python balíčky třetích stran. GraalPy také, ve srovnání s CPython jako referenční implementací Pythonu, často dosahuje vyššího výkonu. Tato práce realizuje dvě významné funkce Pythonu v GraalPy. První, kterou je API pro asynchronní programování, jedna z posledních hlavních funkcí jazyka Python co v GraalPy chyběla, a která se využívá hlavně pro programování webových aplikací, kde umožňuje psaní souběžného kódu bez potřeby vláken použitím takzvaného barevného asynchronního programování, kde programátor musí explicitně anotovat možné změny toku programu. Obsahuje dvě části, knihovnu pro smyčky událostí, v případé této práce asyncio, a syntax pro anotaci změn toku. Druhou funkcí v této práci je trasovací API, tedy API CPythonu pro implementaci ladících nástrojů, nástrojů pro pokrytí kódu testy, apod. Funguje pomocí analýzy bajtkódu Pythonu při kterém se vyhodnocuje zda je spouštěn nový řádek, a pokud ano, je pro něj zavoláno nakonfigurované zpětné volání. Pokud běžící kód vrátí hodnotu, zavolá funkci nebo vyhodí výjimku, použije se znovu toto zpětné volání, což umožní ladícímu nástroji nastavit pro takovouto událost body přerušení. Obě tyto funkce jsou součástí vydané verze GraalPy a mají významný přínos pro kompatibilitu s balíčky třetích stran.

Keywords

Python, async, debugger, GraalPy

Klíčová slova

Python, async, debugger, GraalPy

Reference

HRBÁČ, Adam Rudolf. *Improving the GraalPy interpreter.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. David Kozák

Rozšířený abstrakt

GraalPy je implementace jazyka Python určená pro zjednodušení používání balíčků třetích stran Pythonu v aplikacích Javy. Za tímto účelem běží přímo na Java Virtual Machine (`JVM`), narozdíl od referenční implementace jazyka Python, CPython. Tímto se vyhne práci s C API obou interpretrů a uživatel se tak vyhne sjednocování různých modelů objektů obou interpretrů. Dále GrallPy často spouští programy rychleji než CPython díky využití frameworku pro implementaci jazyků Truffle, který využívá překladače Graal, jenž generuje nativní kód z `JVM` bajtkódu a je sám napsaný v jazyce Java.

Práce implementuje dvě významné funkce jazyka Python do GraalPy. První z nich je takzvané asynchronní programování, používané primárně pro web servery. Narozdíl od typických programů se souběžnými toky programu pomocí vláken, je při asynchronním programování potřeba explicitně určit místa, kdy se tok programu může přesunout – tzv barevný async. Tímto se dá snáze předejít souběhům a podobných chybám. Tato funkce je postavená na dvou částečně nezávislých aspektech, a to specifické syntaxi a knihovně poskytující smyčku událostí, v tomto případě `asyncio`.

Syntaxe umožňuje označit místa, kdy se má tok programu vrátit ke smyčce událostí pomocí `await`, a poskytuje alternativy k `for` smyčkám a `with` blokům. Tyto alternativy jsou postaveny na generátorech Pythonu, které byly implementovány v GraalPy před touto prací. Pro podporu syntaktických prvků byl rozšířen bajtkód GraalPy o vhodné operace, a implementován jejich překlad do bajtkódu. Dále byly generátory rozšířeny, aby uměly pracovat i s novými variantami pro asynchronní programování. Poté byly implementovány tzv. asynchronní generátory, které umožňují programátorům jednoduše zapsat iterátory pro asynchronní smyčky.

Knihovna `asyncio` je jedna ze standardních knihoven jazyka Python a je předchůdcem relevantní syntaxe pro asynchronní programování popsané výše. Dříve fungovala pomocí generatórů, ovšem právě s přidáním nové syntaxe se stala ergonomičtější a primárním způsobem asynchronního programování v jazyce Python. Pro tuto knihovnu byly části, které jsou v implementaci CPython napsané v C, přepsány do Javy pro použití GraalPy. Dále byla implementována datová struktura Hash-Array Mapped Trie (HAMT) pro stav "lokální" k toku programu, která využívá techniky copy-on-write, čímž se z kopií datové struktury stává operace s konstantní časovou a paměťovou složitostí.

Druhá funkce je `API` pro trasování programů. Umožňuje implementaci ladících nástrojů, nástrojů na měření pokrytí testů, apod. Narozdíl od `asyncio` se jedná o implementační detail implementace CPython, který je ale důležitý z hlediska kompatibility. Funguje na principu analýzy bajtkódu za běhu a volání callbacku pro každý nový řádek, volání funkce, chybu nebo vrácení hodnoty z funkce. Tato analýza byla adaptována z interpretu CPython. Dále trasovací `API` umožňuje nastavit řádek, který se má vyhodnotit jako další, což ovšem může výrazně narušit vyhodnocování programu, a tedy je potřeba danou funkcionalitu vhodně omezit. Za tímto účelem byl adaptován algoritmus z implementace CPython pro bajtkód GraalPy, který zajišťuje, že tyto změny řádků mohou způsobit jen chyby na úrovni Pythonu, ne na úrovni Javy, v případě CPythonu C.

Obě zmíněné funkce jsou součástí současných produkčních verzí GraalPy, async od verze 23.2.2, a tracing API od verze 24. Díky async implementaci je možné testovat kompatibilitu s balíčky, které závisí na async funkcích Pythonu, což umožňuje identifikovat další chyby v GraalPy. Některé balíčky fungují přímo, například Flask a httpx. Implementace trasování znamená, že GraalPy aplikace je možné ladit pomocí standardních nástrojů pro Python, například `PDB` ze standardní knihovny. Dále je možné měřit pokrytí testů pomocí nástroje `coverage.py`. Obecně práce své cíle splnila a její výsledky jsou aktivně používány v praxi.

Improving the GraalPy interpreter

Declaration

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Davida Kozáka. Další informace mi poskytl GraalPy tým. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Adam Rudolf Hrbáč
May 5, 2024

Acknowledgements

Rád bych zde poděkoval mému vedoucímu ing. Davidu Kozákovi za jeho pomoc a trpělivost s touto prací. Dále bych rád poděkoval týmu GraalPy a obecně Oracle Labs za možnost věnovat se této práci, a Mgr. Danielu Hrbáčovi za korekturu textu.

Contents

List of Figures

Chapter 1

Introduction

GraalPy [27], an implementation of the Python programming language utilising the language implementation framework Truffle [47], is designed for embedding Python into Java applications without having to go through the C APIs of the language runtimes. In order to maximize the usability of GraalPy, it is desirable to support as much of Python as possible. Unlike similar language implementations using Truffle, notably GraalJS [21] and TruffleRuby [22], the most common Python implementation, CPython [29], leaks its internals, both via the well-supported C API and the language itself [40], making it almost useless to implement Python without matching CPython implementation details as well if the Python ecosystem is desirable. Since a big reason for embedding Python into Java applications is facilitating the use of Python packages, the ecosystem is indeed desirable.

There are two major features this work implements in GraalPy, the first of which is async programming, which allows writing concurrent code without parallelism, instead having to explicitly state where context switches may occur, making it easier to avoid data races and race condition. This requires two separate components, the syntax, providing the means by which to indicate where these context switches may happen, as well as async alternatives to the `for` loop and the `with` statement. The second part is the library providing the event loop in which this concurrency occurs. In this work, `asyncio` is ported from CPython, being part of the standard library and the most used such library. Since only fairly small parts of `asyncio` are written via the CPython C API, it was not necessary to reimplement the entire library in Java, only a few key functions. For more details, see Chapter 3.

The second feature is the CPython tracing API, which is primarily used to implement debuggers, and also backs the popular coverage.py [5] library, used to measure test coverage. While GraalPy has its own debugging API provided by Truffle, conventional editors used for Python tend to have their integrated debuggers written with the CPython API in mind, making the API desirable to support, despite being considered a CPython implementation detail. Although it is not reasonably possible to have both APIs behave in identical ways, CPython itself changes the behaviour heavily between minor versions, meaning minor differences are acceptable. For more details, see Chapter 4.

Both features are part of mainline GraalPy releases, `asyncio` since version 23.0.2, and the tracing API since version 24.

Chapter 2

Important Technologies

2.1 Efficient Interpreters

The simplest way to implement an interpreter is by reading instructions and executing the corresponding operations in a loop until the whole program is done executing. This is typical of shell languages, esoteric languages, and most importantly so-called bytecode interpreters, where simple operations are serialized into raw bytes, similar to the language the CPU itself evaluates. For languages which primarily employ procedural programming, it is common to compile their source code into bytecode, and then interpret the bytecode with this interpreter, allowing for greater performance. The typical terminology used for bytecode interpreters includes:

- Bytecode – The program itself, represented as an array or stream of bytes.
- Instruction – A single operation of the bytecode. Composed of an opcode and zero or more arguments.
- Opcode – The kind of instruction. For example, an instruction would be *load the 10th local variable*, while the opcode of this instruction would be just *load local variable*.
- Argument – The 10th in *load the 10th local variable*, used to avoid having to make separate instructions for cases where it is useful to have one instruction per for a large number of usually integers. Some instructions have no arguments, usually the no-op instruction.

A more common way to interpret a programming language involves constructing an abstract syntax tree (AST) using some parser of choice, then giving each node of this tree a method, for example `evaluate`, which calls evaluate on the children of this node, combining the results in a way that yields the result for the given node, for example `left.evaluate() + right.evalute()` for a node representing an addition. This is called *AST walking*, and unlike the previous kind of interpreters, it can support a wider variety of syntaxes and pre-runtime transformation of the source code, e.g. allowing using a symbol before its declaration as long as the declaration is present. It is however usually quite lacking in performance due to the prevalence of dynamic dispatch, though Truffle improves this, as described later.

A conventionally better method, used by both CPython and GraalPy, as well as most similar interpreters, involves compiling the source program into some high-level bytecode

for a bytecode interpreter as described previously. It is important to note that this compilation usually happens at runtime, as needed (For example, CPython will compile each module the first time it is imported.). This has the major advantage of allowing a significant amount of static inference (both CPython and GraalPy will e.g. number local variables rather than resolving them by name). The actual compilation target can also be something other than bytecode as described prior, which is usually helpful for non-procedural languages, for example interaction nets [15] for functional programming languages, or even just transforming the AST in some way, but the core idea remains the same: an interpreter can be made faster by taking some compilation steps, even without generating native code. While it is possible to optimise code at this level, it is usually not done to a great extent since this compilation should be quite fast to minimise latency – how long it takes to go from source code to the result of running the program, leaving fairly little time for complex optimisation passes.

However, optimizing the code is still desirable, even in interpreters. For this purpose, a technique called just-in-time (hereafter called `JIT`) compilation can be used, where runtime profiling is employed to find the most performance-limiting paths of the program and using a compiler separate from the bytecode compiler to compile (usually) the bytecode directly into optimised native code. Some of the fastest interpreters, such as `V8` of JavaScript [12], LuaJit [18], as well the `JVM`, employ this technique. An interesting example here is `Julia` [7], a dynamic language that compiles each function directly into optimized native code for each combination of types it is called with, allowing for remarkably desirable performance characteristics for certain use cases, notably scientific computation. This approach does, however, lead to fairly high latency, since generating optimized native code is a slow and complex operation, and losing the default option to execute in an interpreter hurts.

There is an in-progress project to employ JIT compilation in CPython [43], though at the time of this work, the performance benefits are not present. There are other Python implementations that do in fact employ this technique, though not in the same manner as was previously described. One such example is Numba [16], a JIT compiler for Python utilizing LLVM [17], which requires the programmer to explicitly state which functions should be compiled. While Numba is capable of compiling all Python code, it is primarily used in so-called *nopython* mode, where it forgoes compiling all valid Python code for the sake of performance, instead compiling a statically-typed, object-free, subset, thus all but eliminating the usual interpreter overhead. An interesting standout is PyPy, an implementation of Python written in RPython [2], a restricted subset of Python designed for more efficient interpretation, using RPython to get behaviour akin to JIT compilation without having to explicitly write a native compiler. GraalPy does also feature JIT compilation, as is described in Section 2.4.

2.2 Graal Compiler

The Graal compiler is a native compiler for the `JVM`, that is, it produces optimised native assembly from `JVM` bytecode. GraalVM is a `JVM` distribution which includes this compiler as the `JIT` compiler, as well as a suite of other tools built upon it. Unlike previous such compilers, Graal is written in Java, rather than C++, meaning it has a greater degree of flexibility without extensive interop efforts. This flexibility is what allows frameworks like Truffle (see section 2.3), as well as tools like Native Image [45], an ahead-of-time compiler for `JVM` applications, to function. Several Truffle types are special-cased in the Graal compiler, providing the framework with compiler intrinsics, allowing for features such

as value types [26] (that is, a composite type that is not an object, but rather a primitive), something usually inexpressible under the `JVM`. Of course, the type is a not in fact a primitive in the `JVM` bytecode, but once the Graal compiler ends up compiling it into native code, it will be treated as if it were a primitive.

2.3 Truffle

Truffle is a language implementation framework designed for writing efficient interpreters, built on the Graal compiler and the Java Virtual Machine. Using Truffle, it is possible to achieve fairly impressive results, especially when peak performance is measured. For example Sulong, a Truffle LLVM bitcode implementation is on average only 40% slower compared to Clang, one of the state-of-the-art ahead-of-time C compilers, in C benchmarks [39]. A big part of this performance is accomplished by using the aforementioned compiler intrinsics provided to Truffle by the Graal compiler. It is possible to control the `JIT` compilation using domain knowledge of the interpreter that a conventional `JIT` compiler would not be aware of, giving the language developer further control of the performance of their languages compared to using another interpreted language, without the inconvenience of using a language without a traditional interpreter runtime and thus having to write one for languages that desire such a runtime [46]. Truffle thus introduces two terms [47]:

A *host VM* as the name of the runtime Truffle itself runs on, some variant of the `JVM`, notably the traditional OpenJDK distribution, where Truffle is delegated to a traditional `JIT` interpreter, as the Graal compiler is not present in such a distribution at this time, GraalVM, where the aforementioned compiler intrinsics can be used for further performance, and SubstrateVM [23], the runtime used by the aforementioned Native Image ahead-of-time compiler.

A *guest VM* is the runtime of the language implemented by Truffle, providing whatever additional features are needed that the host runtime lacks. For example, in the case of Python, the guest runtime implements thread-local state, codecs for parsing source code at runtime, etc. The advantage over writing a whole new runtime as one would in a traditional interpreter written in C is that certain features can simply be delegated to the host runtime [47], for example garbage collection and dynamic dispatch.

At the core of Truffle is the `Node` class. A `Node` is a Truffle abstraction for creating abstract syntax trees which can change their code depending on their usage, effectively doing `JIT` compilation on the `AST` directly, allowing for better performance than is usual for tree-walking interpreters, often matching bytecode interpreters [47].

This is accomplished by rewriting the tree, using several Truffle tools, of which the most common is a so-called `Specialization`, where each `Node` has multiple correct implementations of its behaviour for various types, for example `a + b` for two `int` types, but `a.add(b)` for `LangObject` types (where `LangObject` can also end up being some boxed integer type). Truffle then tracks the types used in the `Node`, and picks the optimal `Specialization` for each program point, adjusting as the program runs further.

2.4 GraalPy

GraalPy is an implementation of the Python programming language for the JVM using the aforementioned Truffle language implementation framework. Unlike a traditional Truffle tree-walking interpreter, GraalPy uses a bytecode interpreter instead, which allows greater

performance even with Truffle, while also making it easier to match CPython behavior in many cases.

Efficient implementation of Python is difficult, since a significant part of the value of Python is its library ecosystem, which relies heavily on C extensions. Faster interpreters, notably PyPy as well as GraalPy, will slow down heavily when having to emulate the C API, often being beaten by the CPython interpreter, despite its lack of runtime native compilation. Additionally, Python is a dynamic language with no primitives, with mutable types and a fully dynamic import system, meaning determining when the tree rewriting of Truffle should happen is difficult.

As is typical, the core of the bytecode interpreter is a loop with a large switch, dispatching on the current opcode and choosing the proper logic to run for a given instruction. For GraalPy, this is implemented in `PBytecodeRootNode`. This node then adopts a wide variety of other nodes depending on the bytecode it is executing, allowing each opcode to have its own specialization, even if they use the same node. For example, a program that calls `+` twice would get a separate addition node twice, generated the first time either operator is run. `Truffle` does support this sort of runtime „AST“ generation, though it does require each such operation to invalidate previous compilations. Since it is common for some code paths to run exactly once, spending all this time invalidating compilations would be unproductive. Thus, so-called uncached nodes are used for the first run of a code path, that is, singleton instances of the regular Truffle nodes – no `Node`s are added to tree. They are also not subject to any of Truffle's tree rewriting for their children, and are always used in the interpreter, rather than compiled. If a given code path runs multiple times, it is rewritten to use the process described previously for additional performance.

Just having a plain bytecode interpreter is still quite slow, no matter how clever the Java `JIT` compiler tries to be. However, writing an optimizing compiler from GraalPy bytecode to native code would be all-but-impossible, since GraalPy is largely limited to running in the Java Virtual machine. Thus, the compilation is done entirely by the Graal compiler, which only knows how to compile Java code. On its own, this is more or less a strictly worse version of having the bytecode interpreter written in a regular compiled language such as C, due to those compilers having more time to compile the code on the account of doing so ahead-of-time.

Instead, a technique called *partial evaluation* [9, 11] is used to help improve the compilation. The `PBytecodeRootNode` holds a constant byte array representing the bytecode, as well as the information required to evaluate the bytecode. This allows Truffle to partially evaluate and unroll the core interpreter loop, allowing further compiler passes to optimise entire sequences of instructions, rather than each one individually (see Figure 2.1). A similar technique is employed in CPython, where each instruction predicts the following instruction, and runs it directly if it predicts right, without going through an entire interpreter loop iteration.

2.5 Testing GraalPy

When testing programming languages, it is typical to write a large portion of the test suite in the language being tested, for example [30, 38]. Notably, the CPython test suite is largely written in Python, and since GraalPy is an implementation of Python, it is possible to reuse the tests. This is fully intended by the CPython test suite and it does make an effort at separating CPython implementation details from Python behaviour. For various reasons (differing error messages, reliance on garbage collector events, feature not yet implemented,

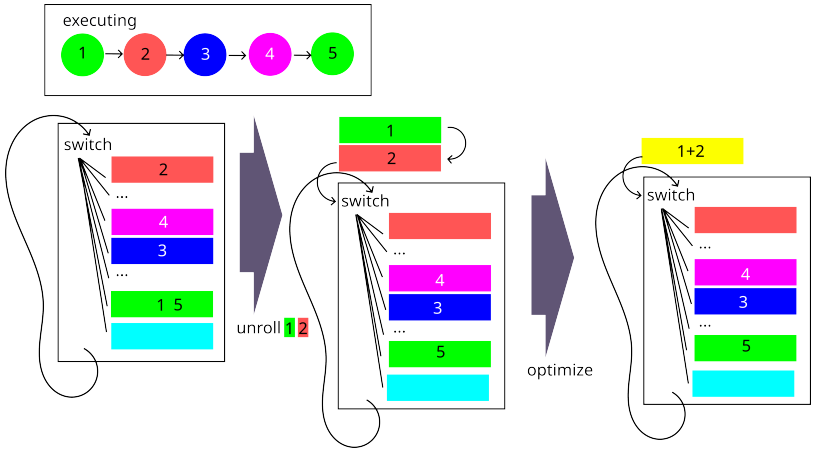

Figure 2.1: Partially unrolling the interpreter loop with the knowledge of the bytecode that will be run, allowing the optimizing compiler to manipulate a sequence of instructions at once. Colors correspond to opcodes, circles correspond to instructions as encoded in bytecode, numbers and arrows correspond to execution order, and rectangles to implementations of instructions. Notice how the switch branches of unrolled instructions remain in case control flow returns to them later, for example when the unrolling happens to the body of a loop.

etc.), not every single test passes, meaning some system for determining which tests are supposed to be passing is needed.

Since the CPython test suite is structured into various modules, the solution is fairly simple, storing a `.txt` file for each module with the passing tests of that module. These tests then run for every pull request. This is a fairly slow process, making it necessary to split the tests into several batches (15 as of 4/2024) which can then run in parallel, reducing the time to around 3 hours per test run. This is the primary way in which the result of this work is tested.

Unfortunately, as was mentioned before, the Python library ecosystem does not always make the same effort to separate CPython implementation details from core Python features, making just the CPython test suite inadequate, even if it is patched in places to include tests CPython considers implementation details. Thus, it becomes necessary to run the individual test suites for each library. These tests run weekly since they are slow, producing reports on which packages regressed or improved [13].

Finally, there are certain features of GraalPy which are unique to GraalPy specifically, notably anything related to Java interop, which require testing via more traditional Java unit tests. This is also where GraalPy-specific implementation details are tested, such as checking for memory leaks. This suite of tests also runs on every pull request, but is much faster, making it more useful for quickly checking if a pull request has a chance at passing the greater CPython test suite.

Chapter 3

Improving Support for Async in GraalPy

In recent years, asynchronous programming has seen a rise in popularity within the Python ecosystem, primarily for web-related tasks. The inclusion of `asyncio` into the standard library has allowed the ecosystem to grow to the point async support is expected in just about every relevant library. This has shifted the situation from async being a weird gimmick for specialised use cases to being the state-of-the-art way to do web development in Python. Since running web servers is a usecase of GraalPy, supporting these libraries is valuable.

As GraalPy gets better at supporting the Python ecosystem, it gets more and more common to run into popular libraries which do not work due to lacking async support. The primary goal of this chapter is to eliminate this as a cause for incompatibility. Since the current stubs can often hard crash or have other undesirable behaviours, running test suites can get complicated, rather than the tests simply not passing, potentially hiding other bugs in GraalPy. Following these contributions, async features tend to work with no additional changes if the remainder of the package does. A current example is `Jinja2`, a mostly synchronous templating library with a couple extra async features.

There are async libraries which could work on GraalPy without needing the explicit async support of this work, as they predate the relevant Python features, have no native dependencies, and are almost entirely pure Python. They, however, tend to lack ecosystem support to the extent `asyncio` does, so they are insufficient for GraalPy. An interesting standout is `Gevent` [8], as it can „convert“ any library into its own kind of async using a lot of monkey patching and rewriting the stdlib. It uses a few too many implementation details to work correctly without additional effort, though supporting it would certainly be a future goal of GraalPy.

3.1 Background on Relevant Technologies

3.1.1 Glossary

There are several terms relevant to this topic with differing meanings when talking about Python in various contexts, use this section as a reference for their meaning in this work:

- **Duck-typing** – a notion of polymorphism where any object that can operate as another object is considered to share a type with such an object, for example, a Cat is any object that can meow, including a e.g. VoiceActor. The name comes from the phrase „If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.“
- **ASGI** – async server gateway interface - an API to allow separating http-related server tasks from web framework logic [3], that is, any ASGI server and any ASGI web framework can be used together.
- **Iterable** – any object fitting the iterable protocol, that is, calling the `__iter__` method returns an Iterator.
- **Iterator** – any object fitting the iterator protocol, that is, supports `__next__` and `__iter__` returns itself [32]. All iterators are iterables.
- **generator** – An instance of the built-in generator type.
- **generator-like** – An object duck-typing compatible with a generator.
- **generator function** – A function returning a generator.
- **coroutine** – an instance of the built-in coroutine type.
- **iterable coroutine** – a generator special-cased to be allowed in await statements, not awaitable.
- **coroutine-like** – an object duck typing compatible with a coroutine.
- **coroutine function** – a function returning a coroutine, coroutine methods also exist.
- **awaitable** – an object with an `__await__` method returning a generator-like object, for example a coroutine.
- **async iterator** – any object fitting the async iterator protocol, that is, supporting the coroutine methods `__anext__` and `__aiter__`.
- **Future** – a representation of an ongoing async computation to be done later.
- **Task** – a `Future` subclass representing an executing coroutine-like object.
- **async generator** – an instance of the builtin async generator, supports two key methods, `asend` and `athrow`, and is an `async iterator`, where `__anext__` is equivalent `asend(None)`.

3.1.2 Language-level Support

It is important to make the distinction between `asyncio` – the library providing event loops as an abstraction around IO multiplexing, and the Python features it uses for the high-level API. `asyncio` can be used without `async/await` (and in fact, predates them), and the language features can be used without `asyncio`, as many alternative libraries do, for example Trio [44].

The very basic abstraction used here is the iterator pattern. In Python, as with most languages, iteration has language-level support [32], such as for-each loops. For an object to be an iterator in Python, two methods must be present:`__iter__` and `__next__` – where `__iter__` must always return self, and `__next__` either returns the next value of the iterator, or raises `StopIteration` if the iterator has been exhausted. It is allowed to call `__next__` even on an exhausted iterator, which should raise StopIteration again.

Writing iterators by hand is annoying and error-prone, however. Therefore, generator functions can be used. Their Python type is still `function`, but they have a special bit set in the function flags – a bitset of various attributes a function can have. A generator function is created by including the `yield` or `yield from` keyword in its body, even if it is in an unreachable position, such as after a return – this allows for creating empty generators. When a generator function is called, it creates a generator object, but does not actually execute any of its body, that will only happen when the generator is iterated over.

A generator is an iterator, but it has features beyond the basic protocol. It is its own type, and can only be created by generator functions (or generator comprehensions, but those are not relevant for this topic, since they cannot contain `yield from`). When another element is requested via `__next__`, the generator executes up to the next `yield`, resuming from where it left off, then pauses, producing the value that was the argument to the `yield`. However, generators have two additional methods, `send` and `throw`, where `__next__` is equivalent to `send(None)`. Since `yield` is an expression, the `send` argument determines the result of the `yield` that produced the previous value. Consider the following generator:

```
def foo():
    bar = yield 1
    baz = yield 2
    return bar + baz
```

The first value that is sent to any generator must be `None`. `foo` has yet to `yield` anything, and thus there is no expression which would result in the `send` argument. This `send` would return `1`, pausing the generator, and bar is yet to be assigned. The next `send` may have a non-`None` argument, and that argument will be assigned to `bar`, say `12` in this example. `foo` will continue evaluating until the *next* `yield`, and its operand becomes the return value for `send` – in this case `2`. Once again, `baz` is yet to be assigned and `foo` is no longer evaluating.

Finally, the next `send` call, say with the argument `13`, will assign that argument to `baz`. But now, there is no further `yield` to evaluate to. Thus, the `send` call will raise `StopIteration`, with one argument – the return value of the generator, in this case `12+13=25`.

More interesting is `throw`, which will take the `yield` the generator is paused at, and raise an exception at that location. The crucial feature here is that any exception handling around the `yield` will trigger for this exception. This is used to report errors back to the coroutine, as will be described later.

```
def process(resp):
    ...

async def get():
    client = Client()
    await client.open()
    resps = await client.get(URL)
    data = []
    while await resps.has_next():
        data.append(process(await resps.next()))

    await client.close()
    return data

asyncio.run(get())
```

Figure 3.1: Colored functions – note how await is needed for async calls.

The final piece of the puzzle is the `yield from` keyword. Intuitively, the keyword delegates to another iterator, obtained from the passed iterable. Any `throw` calls will propagate up if not handled in the delegated-to generator. And since `yield from` is an expression, the return value of the delegated-to generator can be obtained the result of this expression. This allows composing generators and is the state in which asyncio was originally designed.

While these features are sufficient for a lot of use cases, there are several issues. Notably, to quote Python Enhancement Proposal (PEP) 492 [42]:

- It is easy to confuse coroutines[1] with regular generators, since they share the same syntax; this is especially true for new developers.
- Whether or not a function is a coroutine is determined by a presence of `yield` or `yield from` statements in its body, which can lead to unobvious errors when such statements appear in or disappear from function body during refactoring.
- Support for asynchronous calls is limited to expressions where yield is allowed syntactically, limiting the usefulness of syntactic features, such as `with` and `for` statements.

In order to resolve these issues, a fairly elaborate syntactic extension was introduced via the aforementioned PEP 492. These require fairly minor changes to libraries built atop the previous abstractions, and are explicitly designed to allow freely mixing old-style and new-style code [41]. As is typical of Python, minor changes to old code are required (adding `@types.coroutine` to every generator that should be used as a coroutine)

The basis for the new abstractions are coroutine functions, created via `async def`. These are then allowed to use the remainder of async-specific language features, which are listed further. Calling a coroutine function is fairly similar to calling a generator function in that none of the function code is run until the returned coroutine is evaluated. This evaluation is what the async library of choice is responsible for. This situation where one needs to be inside a coroutine to call coroutines is what leads to so-called colored async – there is the async world and the normal world, and the normal world cannot touch the async world, as shown in Figure 3.1 (except for a single entrypoint). While this approach has disadvantages, it does make concurrency easier, since unlike with colorless async (e.g. OS

[1] In this quote, coroutine refers to a generator used for async programming, not yet the Python feature.

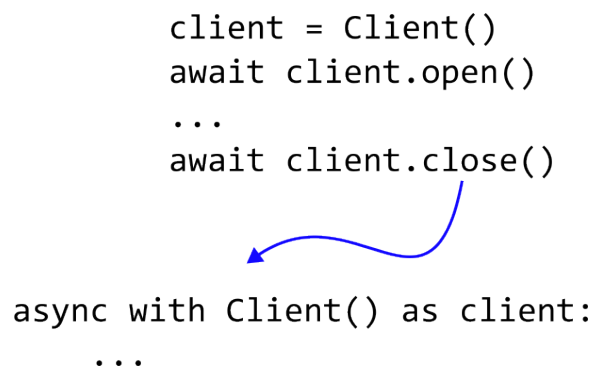

Figure 3.2: Using an async context manager to manage an async resource.

threads), it is very predictable where a context switch happens – wherever there is an `await` or some other async keyword. A perhaps less obvious consequence is that doing anything that does not properly yield back to the event loop will block every single coroutine until it finishes. `asyncio` has a diagnostic to report such a situation at runtime.

A coroutine is similar to a generator, however, unlike a generator, it is not an iterator, since it doesn't really make sense to iterate over it. However, it is possible to create so-called iterable coroutines from generators via the aforementioned `@types.coroutine`. An iterable coroutine is a generator with a special flag set to make it possible to use it in `await` expressions despite not supporting `__await__`. A coroutine does however still support `send` and `throw`, with just about the same semantics. An important difference is that while requesting the next element from an exhausted generator repeatedly simply raises `StopIteration` (however, only the first will have the return value, the rest will just have `None`), a coroutine will raise a `RuntimeError` – this is to avoid accidental coroutine reuse.

The semantics of `await` are very similar to `yield from`, since it is meant to be a one-to-one replacement for `yield from` in these new coroutines, with some notable exceptions:

- Instead of taking iterables, it takes awaitables (and iterable coroutines) and uses `__await__`.
- It can only occur inside async functions.
- It sets the origin if the awaited object is a coroutine, allowing for easier debugging.

With async being a first-class citizen in the language, it allows adding new syntax to simplify common patterns in async code – notably `async for` and async iterateables/iterators, as well as `async with`.

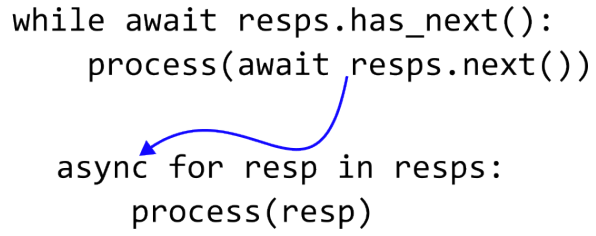

Figure 3.3: Using an async for loop to iterate over an async iterator.

Async context managers, used via the `async with` keywords, await the `__aenter__` coroutine method, passing its result to its optional binding, then runs its body, finally awaiting `__aexit__` with information on the exception that occured in its body if one did, making it an error handling facility, similar to a regular context manager. `async with` is quite crucial, since an object initializer can't be async – making async context managers the only way to sanely initialize async resources, such as database connections (and close them later, of course) – see Figure 3.2. This often has the consequence of creating deeply nested code if several such async resources needs to be setup, but it is adequate enough for most use cases.

Async iteration is similar to regular iteration, `__aiter__` is a function (not a coroutine function, though it used to be one in older Python versions) that returns an async iterator, and `__anext__`, a coroutine functions which can be awaited to get the next element of async iterable, or raises `StopAsyncIteration` if there are no more elements. It is impossible to reuse `StopIteration` since that is used to signal end of iteration in async internals. Thus, the iterator pattern has language support even in async usecases (see Figure 3.3).

`await` and `async for` are also allowed in comprehensions [28], for example `[await foo(i) async for i in bar]` and will work as one would expect – `async for` iterating over the iterable and producing one item at a time, `await` awaiting its operand. As one would expect, they are only allowed in async functions (exempting async generator comprehensions, as described later).

The same issue as before exists, however. Writing an async iterator by hand is annoying and error-prone. And so, coming back full circle , async generators are a way to conveniently define async iterators, made by including the `yield` keyword in the body of an `async def` function, or via an async generator comprehension. Async generators have the coroutine methods `asend` and `athrow` on top of being an async iterable – they work as one would expect, though the internals of how they accomplish this are non-trivial [35].

3.1.3 Asyncio Internals

At the core of asyncio stands an event loop. This event loop notably tracks the following:

- Callbacks to run in the next iteration.
- The time at which other callbacks should run.
- File descriptors to watch for reading or writing, and callbacks to run once they become readable/writable.

An event loop works in iterations. Starting by computing the arguments to a selector, an operating system primitive for waiting for multiple events at once (usually one of several sockets becoming readable) – for example `epoll` [1] on Linux. The timeout is computed as the time to the earliest callback. Any new callbacks added as a consequence of a callback are only run on the next iteration, even if it means running a selector with 0 timeout – which can still result in some callbacks being added if the selector found a readable/writable file descriptor.

This alone would be quite unwieldy, so the `Future` class can be used to represent a result that will eventually be set by an event loop callback. A `Future` has a reference to the event loop it belongs to, and will schedule callbacks that should run when it is done. It can either end up with a result, or with an exception.

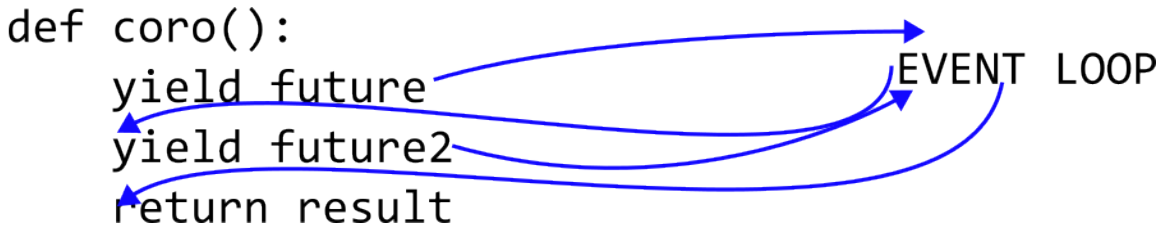

Figure 3.4: Coroutine receives control flow once the yielded future is done, as set up by the `Task` it is wrapped in.

The way this whole set of abstractions connects to coroutines is via the `Task` class. The coroutine should `yield Future`s (which can also be further `Task`s), and it will receive control flow once that future is done, meaning that it also needs to setup the event loop to complete the yielded future. The `Task` is responsible for taking these future yields and setting up the event loop to either `send(None)` if the future was successful, or `throw` the exception if it failed, thus creating the flow shown in Figure 3.4.

For example, `asyncio.sleep(1)`, a coroutine that simply waits one second and returns None, creates a `Future`, sets up a callback in the event loop to finish that `Future` in one second, then yields the future. A `Task` running that coroutine then adds a callback to that future that will then call a method of this `Task` that will eventually call `send(None)` on the coroutine. After one second, the callback that finishes the `Future` runs, thus triggering the `done` callback of the `Future`, which then calls the `Task`, which finally runs the sleep coroutine up to the next yield, or, in this case, the return of None.

3.1.4 State of Async in GraalPy

Prior to this work, generators and `yield from` worked correctly, so despite the different internals, the implementation is largely a task of replicating the CPython steps that added the new async features. This was nevertheless met with some interesting issues, as is described in Section 3.2.

A generator in GraalPy is effectively an array of Truffle `CallTarget`s, which one can think of as lower-level „functions“. Every yield gets its own `CallTarget`, and thus the correct place to continue is simply the corresponding `CallTarget`. This is completely different from CPython's solution where frame objects are kept around and have a state machine attached to track which should execute, but it does result in the same language feature. `yield from` is then implemented as per PEP380 [10] , except entirely in GraalPy bytecode – there are specialised bytecodes `SEND` and `THROW` for this purpose (not to be confused with `RAISE`, which simply raises an exception).

Unfortunately, the function flags described earlier were not properly implemented in GraalPy, meaning quite a bit of runtime information that is critical for asyncio and the syntactic elements to function correctly was not present. Complicating matters further, unlike CPython, the flags are not the „source of truth“ for these attributes, instead being computed at runtime when requested to allow certain libraries to use them to some extent.

3.1.5 Task-local State, the contextvars Module and Hash-Array Mapped Tries

It is often useful to have variables that are *local* to each unit of concurrency, that is `Task`, but which are still *global* in scope, similar to thread-local variables when dealing with OS threads. The standard Python contextvars module is ideal for this usecase, as it allows setting a thread-local namespace called context to an instance of type `Context` for a given function call, and any accesses to so-called context variables within that call will refer to that context. Thus, asyncio can ensure it tracks the context of each coroutine and set the appropriate context.

It is important to note that when a new `Task` is spawned from a coroutine, it should inherit its context, but any changes to this context cannot be allowed to propagate back. Thus, the context must be copied. However, since changing a context variable is much less common than starting a new task, it is desirable for copies to be constant-time and space. Context variables are also accessed by their string name.

These requirements limit the choice of data structure quite heavily. Constant time copies all but require copy-on-write, but copying the entire data structure on each write is also undesirable. Since the variables are string-keyed, and most contexts only have few variables, the data structure of choice will be a hash map of sorts, and to minimize the amount of copies per write, the data structure should have a tree-like structure. Thus, the Hash-Array Mapped Trie [4] (hereafter called `HAMT`) is used in CPython, and in GraalPy following this work as well. Trees are convenient for copy-on-write since each write only needs to copy the parent nodes of the modified leaf, that is, a $O(logn)$ amount of memory, given a balanced tree. Of course, maintaining that balance can often be problematic. In a HAMT, the hash function is fully responsible for this, and having a poor one will significantly hinder performance – it must distribute items largely evenly across the whole hash space. Of course, alternative copy-on-write associate array data structures do exist, notably the various kinds of search trees, for example the standard Haskell `Data.Map` library. However, a HAMT leads to shallower trees, thus allowing for lower overhead in smaller trees, often fitting them in one or two nodes.

There are certain adaptations needed to use a `HAMT` in this work, as described in Section 3.2.1.

3.2 The Implementation Process

3.2.1 Implementing a Hash-Array Mapped Trie

`HAMT`s are fairly well explored, and are remarkably popular as the primary associative array type in functional programming languages (notably Clojure) for their ability to efficiently operate as an immutable data structure. The basic idea is setting up a tree, and using some number of bits from the hash of the key to describe a step on the path through this tree. For GraalPy, the perfect number of bits turns out to be 5, same as CPython and the original paper [4], yielding at most a 6 deep tree using 30 bits of the hash, that is, the first 5 bits are used for the root node, the next 5 for the appropriate child, etc. A naive implementation would simply store 32 pointers per node, however, the memory inefficiencies there would be quite glaring. Instead, 4 node types are used, as well as `null`.

Leaf nodes store a key-value pair and have no children.

Array nodes are the naive option mentioned before, always having 32 children, some of which are potentially $\perp$. When a node has over 16 children, this node is used, and indexed by the corresponding 5 bits for its depth.

The magic happens with the so-call bitmap nodes, which store all children in a smaller, contiguous, array. The issue here is of course figuring out how to determine at which index is a given child. To do this, we use a 32-bit bitmap, where each set bit corresponds to a child at that „index“ – that is, the 5bit slice of the hash. Then, to compute the index in the array from the hash slice, a mask is used to pick all bits before the corresponding bit, and then the popcount operation is done (counting the number of ones in an integer). Thus, the very first set bit will yield 0, the first item in the array, and the later set bits will yield further items. This node is used for all nodes which have fewer than 16 children.

Additionally, since unlike the original paper, the GraalPy hash is only 32bits long, and cannot be extended further to distinguish any two unequal keys, a way to handle collisions is required. Colliding Leaf nodes are simply stored in a flat array, in a so-called Collision node.

Accessing a specific context variable in a HAMT is done via the algorithm in Function 1. The key operation for such access is the calculation $hash(k) \gg 5 \cdot d \,\&\, 0\text{x}1\text{F}$, which takes the 5-bit slice of the hash appropriate for the current tree depth. The logic then depends on the type of the HAMT node – note that a subtree of a HAMT is not a HAMT, due to starting the hash at a different depth, which is why depth is tracked across recursive calls.

Given a $\perp$ node, the HAMT node is empty and, meaning $\perp$ is returned, as on line 1.

Given a Leaf node, the HAMT node has one value, if the keys line up, the value was found and is returned (line 2), otherwise return $\perp$ (line 3).

Given an Array node, The appropriate child index is computed and Lookup is recursively called on the child (line 4). If the child is $\perp$, the recursive call with produce $\perp$ as well.

The interesting case is the Bitmap node. The index is computed as before, but now, accessing the item at that index first requires checking whether that index is present – there is no array with a $\perp$ in this case (line 5). If it is, all bits after the bit at that index, and that bit itself, are masked off, then running popcount is run on that result, yielding the index of the child (line 6).

A Collision node is largely a generalization of the Leaf node. If any leaf of the Collision Node has the looked-up key, the corresponding value is returned (line 7), otherwise, $\perp$ is returned (line 8).

Within the two algorithms in this chapter, the following notation is used

- $\perp$ – the empty node
- $hash(k)$ – the 30bit hash of the key k,
- $a \gg b$ – a logically (leftmost bits are 0-padded) bitwise shifted right by b places,
- $a \ll b$ – a bitwise shifted left by b places,
- $a \,\&\, b$ – the bitwise and of a and b,
- $a \mid b$ – the bitwise or of a and b.

Function 1 – Lookup(HAMT node N, key k, depth $d = 0$)

```
   Output: The value associated with k, or ⊥
   begin
       if N = ⊥ then
1          return ⊥
       else if N is a Leaf node then
           if key of N = k then
2              return value of N
           else
3              return ⊥
       else if N is an Array node then
           i ⟵ (hash(k) ≫ 5 · d)&0x1F
           N′ ⟵ child i of N
4          return Lookup(N′, k, d + 1)
       else if N is a Bitmap node then
           o ⟵ (hash(k) ≫ 5 · d)&0x1F
           b ⟵ the bitmap of N
           if b & (1 ≪ o) = 0 then
5              return ⊥
           i ⟵ popcount(b & ((1 ≪ o) − 1))
           N′ ⟵ child i of N
6          return Lookup(N′, k, d + 1) /* Bitmap nodes cannot have ⊥        */
       else if N is a Collision node then
           for leaf L in N do
               if key of L = k then
7                  return value of L
8          return ⊥
```

While accessing a key in a HAMT is a fairly straightforward, driven almost entirely by the shape of each node, insertion is a bit more complicated, as seen in Function 2. As was described previously, rather than to mutate the tree, a new HAMT is returned, which may share certain nodes with the previous HAMT, implementing copy-on-write behaviour. Nevertheless, one part remains the same, the action taken depends on the type of node.

Adding an item to the $\perp$ node is straightforward, simply return the Leaf node that should be added.

For adding to a Leaf node, there are four possible cases:

- The new node and the existing node share keys – the new node is returned (line 1).
- The new and existing nodes share hashes, but not keys – a new Collision node is created for them (line 2).
- The two nodes have distinct hashes at a different depth – they are combined at a lower depth, and stored at a 1-child bitmap node (line 3).
- The two nodes have distinct hashes at this depth – They are combined at into a two-child bitmap ndoe. It is important that the node with the lower corresponding hash slice is the first child of the Bitmap node (line 4).

Adding to an Array node is much more straightforward, the new Leaf is recursively added to the child at the appropriate index, and the entire node is shallow-copied with that new child in place. (line 5)

Adding to a bitmap node is probably the most complex case here. There are once again three cases:

- If there is already a child at that index, as determined by the corresponding bit being set in the bitmap, the process is the same as with Array nodes (line 6).
- If adding this new child would result in a bitmap node with over 16 items, it is fully replaced with an Array node (line 7).
- Otherwise, insert a new node at the appropriate position, and add it to the bitmap by setting the appropriate bit to 1 (line 8).

Adding to a Collision node is once again a generalisation of the 3 Leaf node cases.

Function 2 – With(HAMT node N, Leaf node L, depth $d = 0$)

Output: A new HAMT node that contains L

begin

$k \longleftarrow$key of L
$v \longleftarrow$value of L
$o \longleftarrow (hash(keyofL) \gg 5 \cdot d)$&0x1F
if $N = \perp$ **then**
  return L
else if N *is a Leaf node* **then**
  $k' \longleftarrow$ key of N
  if $k = k'$ **then**
1   **return** L
  else if $hash(k) = hash(k')$ **then**
2   **return** a Collision node with two children, N and L
  $b \longleftarrow (1 \ll o)|(1 \ll o')$ $o' \longleftarrow (hash(k') \gg 5 \cdot d)$&0x1F
  else if $o = o'$ **then**
    $N' \longleftarrow With(N, L, d + 1)$
3   **return** a Bitmap node with one child N' and bitmap b
  $C_{0,1} \longleftarrow \begin{cases} L, N & o < o' \\ N, L & o > o' \end{cases}$
4 **return** a Bitmap node with two children C_0 and C_1, and bitmap b
else if N *is an Array node* **then**
  $N' \longleftarrow$child o of N
  $N'' \longleftarrow With(N', k, v, d + 1)$
5 **return** shallow copy of N, with child o replaced with N''
else if N *is a Bitmap node* **then**
  $b \longleftarrow$ bitmap of N
  $i \longleftarrow popcount(b\&((1 \ll o) - 1))$
  if $(b \gg o)\&1 = 1$ **then**
    $N' \longleftarrow$ child i of N
    $N'' \longleftarrow With(N', k, v, d + 1)$
6   **return** shallow copy of N with child i replaced with N''
  else if $popcount(b) \geq 15$ **then**
    $j \longleftarrow popcount(b) - 1$
    for $j' \leftarrow 0$ *to* 31 **do**
      if $(b \gg j')\&1 = 1 \wedge j \geq 0$ **then**
        $A_{j'} \longleftarrow$ child j of N
        $j \longleftarrow j - 1$
      else
        $A_{j'} \longleftarrow \perp$
    $A_o \longleftarrow L$
7   **return** new Array node with children $A_{0...31}$
  else
    $b' \longleftarrow b|(1 \ll o)$
8   **return** shallow copy of N with a new child L *inserted* at index i
else if N *is a Collision node* **then**
  left as an exercise to the reader

3.2.2 Replicating the `_asyncio` Module

The native `_asyncio` module is used by the python `asyncio` module to improve performance and pass the test suite – there are some minor differences when internal APIs are monkey-patched. It also contains native implementations of the `Task` and `Future` classes. These are however not implemented in GraalPy since they only matter for performance, and functionality was a priority. These are mostly straightforward, with the added importance of not losing track of `None` and `null`. The implementation always stores `null` in Java code, and transforms it to `None` explicitly. This allows Java tooling to keep better track of things, as well as making it more explicit where `None` is and is not allowed.

3.2.3 Iterable Coroutines

While most of the implementation was simply about translating CPython code to GraalPy, iterable coroutines have their own share of complexity. The first obvious problem was that generators inherit flags (notably `CO_ITERABLE_COROUTINE`) from the `CodeUnit` rather than the Python code object. A `CodeUnit` is a product of the compiler, and as such its flags are not affected by `@types.coroutine`. The solution ends up passing the appropriate information explicitly to the constructor, rather than trying to reconstruct it into flags. This does mean flags are incorrect in some places, but it does work and will likely be fixed fully in the future, avoiding the need for this workaround.

`await` then of course needs to be aware of this possibility, but that is fairly straightforward with Truffle specialisations.

A related note is compatibility in the opposite direction, that is, `yield from` must be able to work with a coroutine if inside an iterable coroutine. This was solved simply – `yield from` always works with coroutines. The bytecode interpreter does not have access to the python object that holds its code and it would require a major refractor to give it that awareness.

3.2.4 Other Async Keywords

While `await` can reuse code of `yield from` by replacing a single instruction to get the iterator to delegate to, both `async for` and `async with` are exception handlers – `async with` passes any exception in its body to `__aexit__`, and `async for` has to catch the `StopAsyncIteration` exception to find where the async iterable ends. `async with` is relatively straightforward, since it behaves roughly the same as `with`, just awaiting rather than calling.

`async for` however is quite complex, since it requires keeping a significant amount of state on the stack, as well as catching `StopAsyncIteration` in the exact right subsequence of bytecode. This gets even more complex in async comprehensions, where the nesting requires special care.

Another interesting note with async comprehensions is that generator comprehensions are quite special. As was said before, `await` is only allowed in coroutine functions. This is however not quite true. Both `await` and `async for` are allowed in generator comprehensions, and their presence turns such a comprehension into an asynchronous generator comprehension, producing an async generator. Since they are expressions that produce an async generator, they are allowed outside async functions, and their result is not implicitly awaited unlike with a usual comprehension. Thus, for a comprehension to be compiled, one must check if it contains any async syntax, then check if it allowed to contain this syntax,

and produce the appropriate code, implicitly awaiting the result for list, set, and dictionary comprehensions, but not for generator comprehensions.

3.2.5 Async Generators

Async generators are one of the more involved pieces, since they have to differentiate between `yield` as in some awaitable being awaited just yielded something, and `yield` as in a `yield` expression used in the body. In the case of GraalPy, this is done by generating a special bytecode instruction `ASYNCGEN_WRAP`, which wraps a value in a special `asyncgen_wrapper` object. This object is unreachable from regular Python code, since async generators do not have `.send` and `.throw`. The only way to interact with them is via the two new awaitables, which must be implemented in Java for GraalPy and are implemented in C for CPython, allowing them to send and throw despite them not being exposed to Python code.

These awaitables are created via the `.asend` and `.athrow` methods, and are used to send/throw something to a `yield` in the async generator body. The GraalPy implementation is largely a one-to-one port of the CPython implementation, though since the implementation used `goto` heavily in C, rewriting in Java ended up being fairly distinct.

3.3 Conclusions

A basic implementation of the standard Python async APIs was merged into GraalPy. Certain features are missing, notably origin tracing, asyncgen finalizers, as well as warnings on unawaited coroutines. Since almost all of these are either impossible, or debugging tools, they are not a priority, though it would be nice to support them in the future.

Following this work, 121 out of 163 CPython tests for the async syntax pass as-is. The non-passing tests are largely differences in error messages and similar minor things, though some do indeed fail due to the missing features.

While a large amount of the asyncio CPython test suite does pass, the asyncio test suite is remarkably extensive, which, combined with asyncio having features for threads and processes, something GraalPy does not handle as effectively as CPython, means that running all of them is all but impossible. However, a reasonable subset has been enabled to avoid regressions.

Currently, the httpx asgi client and fastAPI asgi web framework work fully, but the lack of an ASGI server makes it hard to put this support into practice. However, the limitation on the support for ASGI server is no longer async, but rather a variety of other, more minor issues, creating a clear path towards compatibility.

This work is part of the GraalPy 23.0.2 release, and has been used to improve compatibility with the Python ecosystem.

Chapter 4

Python Debugger API Support in GraalPy

A general goal of GraalPy is allowing Java programs to access the Python ecosystem, as the ecosystem has certain libraries developed to a greater degree than a possible Java alternative, e.g. numpy [20] for manipulating multi-dimensional arrays, and PyTorch [6] for machine learning. Therefore, it is useful to maintain behaviour as close as possible to CPython, as, although Python does make an effort at separating implementation details from parts of the language, most of the ecosystem is only tested for CPython, and thus will not work by default on non-CPython implementations. One of these implementation details is tracing, a CPython API for implementing debuggers and similar tools. The goal of this part of the work is supporting this API in GraalPy.

The primary goal is to create a baseline from which to improve support for libraries such as coverage.py [5], a tool for measuring test coverage, and pdb [34], the standard python debbuger, as well as the integrated debuggers in PyCharm [14] and VSCode [19]. This should allow significantly easier work with GraalPy in those editors, and saves work on porting Truffle tooling to each of the editors, which will never be quite as good as the integrated debuggers.

GraalPy does support a debugger, coverage, etc. even outside of CPython's tracing, via the relevant Truffle APIs [25]. However, these are hard to hook into from Python, and behave differently enough that it is more or less impossible to port existing packages that depend on tracing to the Truffle APIs. They do however have a number of advantages, notably in performance and handling of Java-related exceptional cases, such as being able to manipulate Java stack traces, thus developers targeting GraalPy are likely to want to use them if possible. Nevertheless, the convenience of just being able to run CPython-targeted tooling the user may already be familiar with makes this implementation worthwhile.

Due to the recent switch to the bytecode interpreter in GraalPy, it is possible to implement tracing in a very similar fashion as in CPython. However, line numbers are handled differently between CPython and GraalPy, requiring a more complex computation on the GraalPy, since only source offsets are stored. It is impossible to get identical behaviour, since the bytecode is quite different in places, as will be described later. Nevertheless, most programs using the tracing API should work, unless they rely on opcode events – that is, the ability to trace execution at the CPython bytecode level, since GraalPy does not use CPython bytecode.

4.1 Background on Relevant Technologies

4.1.1 CPython Tracing API

The tracing API uses so-called tracing functions to report the currently executing code to the user. A tracing function takes 3 arguments:

1. frame – the frame object representing the frame entered by the event.
2. event – One of the strings `'call'`, `'line'`, `'return'`, `'exception'`, or `'opcode'`, describing the event that caused a tracing function call.
3. argument – An extra argument for certain events, `None` if unused.

The tracing entry point is the `sys.settrace` function [36]. It sets the so-called **global** tracing function, stored in the thread-local state of the interpreter. This is in contrast to **local** tracing functions, which shall be discussed later. The global tracing function is called whenever a new scope is entered (that is, a new frame is created and a code object is executed, generally due to a function being executed). Of course, one cannot trace the inside of a tracing function, as that would recurse infinitely. The event used here is `'call'`, which is the only event possible for the global tracing function. The global tracing function then should return a tracing function to be set as the local tracing function for the newly created frame.

The local tracing function is stored on the frame object under the attribute `f_trace`. It is updated every time it is called to the return value of the past tracing function (see Figure 4.1) and is called for the other 4 events:

- `'line'` – Traced whenever a new line of code is executed. It can be disabled via setting the frame attribute `f_trace_lines` to `False`. It is the most complex event to handle sensibly, since the definition of what is an executing line is not quite exact. In both CPython and GraalPy, it is mostly defined by the generated bytecode, which, albeit not perfectly consistent, makes the implementation comprehensible and works well enough for most tools.
- `'return'` – Traced whenever the scope we are currently in exits, be it due to a `yield`, `return`, or an exception. The argument parameter is set to the return value, or `None` if no return value is applicable (e.g. due to an exception). The return value is ignored, since there is no further execution in the frame to trace.
- `'exception'` – Traced whenever an exception is raised in the current scope. It is followed by the `'return'` event unless this trace function call returns `None`. The argument holds the `(exception, value, traceback)` exception triple.
- `'opcode'` – Traced for every bytecode instruction executed by the CPython bytecode interpreter. Not implemented for GraalPy, as the bytecode numbers are not compatible, making it more or less impossible to actually use without special casing for GraalPy. Disabled by default in CPython, can be enabled by setting the frame attribute `f_trace_opcodes` to `True`.

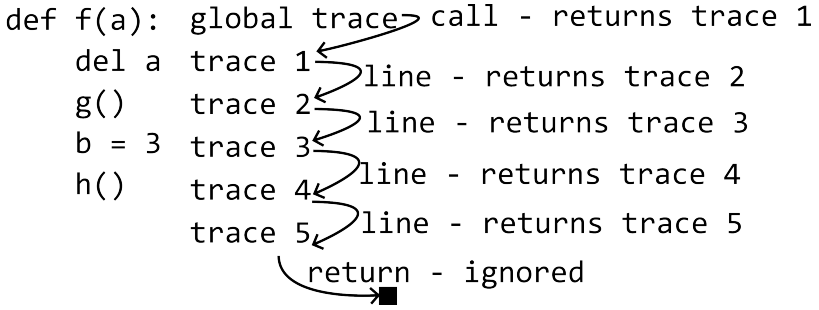

Figure 4.1: A diagram showing how each invocation of the trace function sets the local trace function for the rest of the scope. The arrow labels show the event passed to the trace function and their return value.

4.1.2 Relevant Aspects of GraalPy

With the version 22.3, the implementation of GraalPy switched to a bytecode interpreter, instead of the original AST interpreter, which Truffle was originally designed for. This has allowed for greater control over performance, as well as making it easier to track control flow, which allowed the implementation of tracing in this work.

The bytecode used by GraalPy is stack-based, that is, intermediate values are stored on a stack, and most instructions either push or pop from the stack. This follows CPython and is easier to generate and interpret than the often better-performing register-based bytecode. It is important to ensure that the stack does not underflow, that is, no instruction pops more from the stack than is on it. In order to avoid this entire category of bugs, the GraalPy bytecode is generated so that it is possible to statically check the stack depth at each instruction.

Another consideration when analysing the stack depths is exception handling. GraalPy features zero-cost exceptions, that is, if no exception is raised, the bytecode interpreter executes no exception handling related code. This is done via a so-called exception handler table. Each entry in the table holds 4 values, the start of the exception handling area , the end of the exception handling area, the start of the handler, all in byte indexes, and the stack depth to unwind to at the handler. This table is generated so that the handlers never overlap, and so that there are always enough stack items for the handler. The semantics are fairly straightforward, if an exception happens and the current byte index is within a handler, pop 0 or more stack items until the stack depth matches the depth in the handler, then push the exception object and continue execution from the byte index of the handler.

4.2 Tracing via Bytecode Execution

At the high level, the logic for tracing is the same in CPython and in GraalPy. However, due to extra optimisations and the lack of a line number table in GraalPy, certain additional considerations must be made. A description of how each event is detected in GraalPy follows.

Call Events

Whenever a PBytecodeRootNode is executed, a call event is traced, with one exception – On Stack Replacement (OSR) loops. OSR is a technique to allow recompiling a loop

between its iterations. It is critical to be able to efficiently interpret code such as the following.

```
a, b = 0, 1
for i in range(100000):
    a, b = b, a + b
```

Otherwise, it would be impossible to specialise the inner sum if the function containing it is only called once, so even if a more optimal version of the code were to be created, it would not be used. However, the way OSR is implemented means that it will execute the PBytecodeRootNode again, from the correct bytecode index, allowing further specialization.

Checking if the bytecode index is 0 at the start may seem like a solution, but generators also execute the node at a non-zero index, continuing from after the last yield, and call events should be emited in that case. Instead, the PBytecodeRootNode is informed whether it was called from OSR or normally, and uses this to decide whether a call event should be traced.

Additionally, this logic is responsible for setting the local trace function, stored on the frame object. This has a severe performance penalty, since materialising a full frame object is avoided in GraalPy to a maximal degree, but if tracing is enabled (that is, a global tracing function is set to a non-`None` value), it needs to be created every time. This cost is not avoidable while tracing, but can be avoided when not tracing – see Section 4.4.

Return Events

In CPython, goto is used to ensure a return event happens without having to explicitly enumerate all possible exits from a scope [31]. This is not possible in GraalPy, instead, each bytecode which can lead to a scope exit is handled separately, and exceptions are handled in a `catch` block over the main interpreter loop. To be more specific, there are 2 possible scope exits outside of exceptions:

`RETURN_VALUE` is a bytecode instruction that is genereted whenever a function returns a value, be it via the `return` statement or returning `None` implicitly. When this opcode executes, a return event is traced on the frame's local tracing function, and its argument is set to the value returned.

`YIELD_VALUE` is treated in the same manner as `RETURN_VALUE`, though it is used for generator yields rather than return values. An interesting edge case arises here – the return value is ignored, despite the frame continuing to exist, as a new local trace function will be generated when the generator resumes.

With exceptions, the situation is a little more complicated, since GraalPy can have Java exceptions which are in no way represented or can be manipulated from the Python side, e.g. due to native interop errors from Sulong. However, these exceptions generally create an unconditional interpreter crash, so it is fine to not create a return event. It would of course be ideal to allow a debugger to inspect the Python state during the crash, but this is one of the scenarios where the Truffle-specific tooling is better equipped to help.

If it is a simple Python exception or an exception which it makes sense to wrap as a Python exception (e.g. calling an interop JS function that errors out), and it is not handled within the same scope – e.g. via `try ...catch`, a return event is called with None as the argument. Its return value is also ignored.

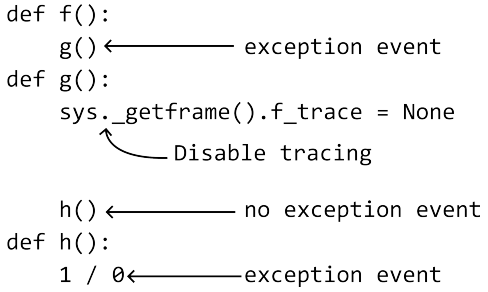

Figure 4.2: Propagation of exception events up the call stack.

Exception Events

As was alluded to before, exceptions are not quite as straightforward as in CPython, since the same Java exception can carry many kinds of failures, ranging from someone dividing by zero in a Python program to an internal Truffle bug. Therefore, an exception event is only traced if the exception can be represented in Python. It is important to note that unlike e.g. `with` statement exits, it is impossible to handle an exception in a traced exception event.

The exception event is traced even if the exception is handled within the same scope, and once for every scope. That is, if a function 4-call-levels-deep raises an error, an exception event will be traced at every call level (unless tracing is disabled for the frame, see Figure 4.2). If the exception is handled, the return value indicates the new local trace function to be used. If it is not, the return value is ignored.

The argument is the (exception, value, traceback) tuple, where the exception is the type of the exception, the value is the exception object itself, and the traceback object stores the stack trace. This allows debuggers comprehensive access to the details of the exception, making it simple for the user to figure out what is going on.

Line Events

At the most basic level, a line event should be traced whenever the previous bytecode index has a different line number than the current index. However, this turns out to be insufficient for jumps due to the way certain common control flow mechanisms are compiled. Additionally, it can create certain odd edge cases around `RESUME_YIELD`. It is also interesting to note that expressions spanning multiple lines do not get multiple line trace events, as the compiler simply assigns them the same line number. Interestingly, the GraalPy logic ends up fairly similar to the old CPython logic, before the new line number format, as described in the notes for the old format [33].

To be more specific regarding jumps, a backward jump must always be traced, even if it ends up on the same line. This is due to programs like `while f(): del y[-1]`, where a debugger wants to control the execution of the condition separately from the body, even though they are on the same line. Additionally since, unlike CPython, GraalPy bytecode generates no instructions for `pass`, instead of the single CPython bytecode `NOOP`, it is impossible to trace lines with only `pass` in them. This is not all that much of an issue outside of breaking a number of CPython tests, however. For the GraalPy tests, `del x` was used, since it is only a single bytecode instruction (`DELETE_NAME`).

In contrast, a forward jump must only be traced if it is a jump to the first bytecode instruction of a line. This is to ensure we do not create line events where execution is not

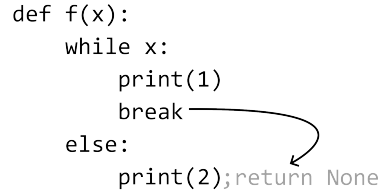

Figure 4.3: A forward jump instruction to a line that is never executed. In grey, note the implicit return of every python function which the jump is actually targeting. Example adapted from [33]

.

happening in certain edge cases, such as else branches on loops. Given the program in Figure 4.3 with the argument `True`, the break generates a jump to after the else branch, where `LOAD_NONE` and `RETURN_VALUE` are. They have their line number set to 6, however, which could make it seem like the `print(2)` being executed. In other words, if other instructions on the same line are generated before the jump target, it is likely the jump is to block cleanup of sorts, rather than user-written code on that line.

In this same example, we can also see a need to rewrite line numbers of `return` events. It can be confusing to see a return event on the last line of a function/block, rather than on the last line that was executed – and thus traced via a line event. Therefore, the frame object passed to the return event does not get its line number set by the line number bytecode table, but instead by wherever the last line event was.

4.3 Debugger Jumps

During debugging, it is occasionally useful to arbitrarily move the execution to a specific line. To allow this, the tracing API allows setting the line number of the frame object inside a trace function. This can be used to e.g. forcibly exit a loop. There are of course several issues with this concept – not every such jump (not to be confused with a jump opcode in bytecode) can work. For example, jumping into the body of a for-each loop should not work, since there is no iterable for further iterations of the loop to iterate over (such jumps will be called unsound hereafter), however, jumping from one for-each loop to another is fine, since there is an iterator on the stack already (such jumps will be called sound hereafter).

In order to determine whether a jump is sound, it is necessary to analyze the bytecode, determining what the types on the stack are for each instruction. Four types are considered:

- *Obj* – Generic objects, produced by most instructions.
- *Iter* – Iterators, produced by compiled `for` loops and comprehensions.
- *Ctx* – Context managers, produced by compiled `with` statements.
- *Exc* – Exceptions, produced by exception handling.

A jump is sound if the destination stack is a prefix of the source stack, that is, it is possible to pop items from the source until they are both equal. The analysis is done as per Algorithm 3, adapted from the CPython algorithm for the same task. The basic idea for the analysis is to start with an empty stack of types for each instruction, then go over each instruction

Algorithm 1: Analysis of bytecode for the purpose of determining whether a jump is allowed

Data: bytecode byte array B, set H of exception handlers (s, e, h, l) catching exception from indexes s to e and handling them at index h at stack level l

Result: function relation R of (i, S) tuples, where stack S corresponds to the types $Obj|Iter|Ctx|Exc$ of the runtime stack before executing the instruction at index i. $\times$ is used as a placeholder for ignored value.

begin

- $Q \longleftarrow \{0\}$
- $R \longleftarrow \{(0, \bot)\}$
- **while** $Q \neq \emptyset$ **do**
  - $i \longleftarrow pop(Q)$
  - 1 $assert\ \forall S'.iRS' \Rightarrow S = S'$
  - 2 **if** $(i, \times, h, l) \in H$ **then**
    - $assert\ l \leq length(S)$
    - create S' by removing items from S until $length(S) = l$
    - $S' \longleftarrow (Exc, S')$
    - $R \longleftarrow \{(h, S')\} \cup R$
    - $Q \longleftarrow Q \cup \{h\}$
  - 3 **if** $op(B, i) =$ `SWAP` **then**
    - $(a, (b, S)) \longleftarrow S$
    - $R \longleftarrow \{(next(B, i), (b, (a, S)))\} \cup R$
    - $Q \longleftarrow Q \cup \{next(B, i)\}$
  - 4 **else if** $op(B, i) =$ `GET_ITER` **then**
    - $R \longleftarrow \{(next(B, i), (Iter, S))\} \cup R$
    - $Q \longleftarrow Q \cup \{next(B, i)\}$
  - **else if** ... **then**
    - other operations with special handling
  - **else if** $hasNext(B, i)$ **then**
    - create S' by removing $nRemoved(B, i)$ items from S
    - **for** $\times \leftarrow 1$ **to** $nAdded(B, i)$ **do**
      - $S' \leftarrow (Obj, S')$
    - 5 $R \longleftarrow \{(next(B, i), S')\} \cup R$
    - $Q \longleftarrow Q \cup \{next(B, i)\}$
    - **if** $hasJump(B, i)$ **then**
      - create S' by removing $nRemovedWithJump(B, i)$ items from S
      - **for** $\times \leftarrow 1$ **to** $nAddedWithJump(B, i)$ **do**
        - $S' \leftarrow (Obj, S')$
      - 6 $R \longleftarrow \{(nextWithJump(B, i), S')\} \cup R$
      - $Q \longleftarrow Q \cup \{nextWithJump(B, i)\}$
- **return** R

in the order they would be evaluated in, removing the appropriate amount of elements from the stack and pushing the appropriate amount of generic object entries, using the resulting stack for that instruction, as done in Line 5. To support this analysis, three functions are defined:

- $next(B, i)$ provides the byte index of the instruction following the instruction starting at $B[i]$. For unconditional jumps, this is the jump destination. Conditional jumps are explained later.
- $nRemoved(B, i)$ provides the non-negative number of stack items popped by the instruction.
- $nAdded(B, i)$ provides the non-negative number of stack items pushed by the instruction.

It may seem that it should suffice to operate on the stack difference, rather than on the number of added/removed items directly. However, since instructions can pop an item of one type and push back a generic object, so not explicitly popping the item and pushing back a generic object could lead to different outcomes depending on code path. This alone is inadequate for sufficiently complete analysis, there are several groups of instructions which must be handled in a special way.

The first group are instructions that do conditional jumps, such as *jump if true*. Here the analysis is split into two, and continues from both the next instruction sequentially, and from the jump destination. Usually, attempting such analysis would require placing some limit, such as up to a fixed point, and even then, no such fixed point may exist – the GraalPy bytecode is after all, Turing complete. Here is where the aforementioned generating instructions such that the stack depth can be analyzed statically comes in – each instruction only ever has one possible stack, that is, if it is reachable from multiple locations, it must have the same stack from all code paths, as checked in Line 1. This allows the analysis to finish in roughly linear time. To support the analysis of conditional jumps, four more functions are defined:

- $hasJump(B, i)$ holds if the instruction at $B[i]$ is a conditional jump.
- $nextWithJump(B, i)$ provides the jump destination of the conditional jump at $B[i]$.
- $nRemovedWithJump(B, i)$ same as $nRemoved(B, i)$, just in the case the conditional jump does happen.
- $nAddedWithJump(B, i)$ same as $nAdded(B, i)$, just in the case the conditional jump does happen.

The previous three functions ($next, nRemoved, nAdded$) provide their values if the conditional jump does not happen. Fortunately, other than the next, hasJump and nextWithJump, these operations were already present in GraalPy and used in the compiler, meaning it was only needed to enumerate all conditional jumps, rather than the stack effect of all instructions.

The second group is where having multiple types on the stack comes in. A jump between an exception handler and a for-each loop is just as unsound as a jump into a for-each loop from the outside. Thus, instructions that generate values used by such statements must be special-cased to pop and push not only the correct number of values, but also the correct

types. There is no shortcut here, every relevant instruction must simply be listed, for example `GET_ITER` on Line 4, providing the iterator when compiling a `for` loop.

The final group are stack manipulation instructions, for example swap. These are used by the compiler in a wide variety of cases and must be handled in a special manner, since they are more or less fundamental operations of stack-based bytecodes and there is not enough of them to make attempts at generalization worthwhile. An example of this is on Line 3. Notice how there is no need to handle the case where a swap is used with fewer than 2 values on the stack. This is once again because the GraalPy compiler will not generate invalid bytecode. While it is possible to handwrite bytecode in CPython and GraalPy, it is acceptable to crash in those cases. Handwriting invalid bytecode is in fact one of the very few ways to trigger a segmentation fault in CPython without explicit C interop.

Another interesting aspect is handling bytecode exception handlers. As described previously, every bytecode array carries with it an exception handler table. The correct way to deal with this table is to check if the currently analyzed bytecode starts an exception catching area, and if so, treat it as a conditional jump to the corresponding handler, as done in Line 2. Since the start of an exception catching defines the shallowest point of the stack for the area, it is the correct place to determine the stack state in the handler. Trying to handle all exception handlers at the start does not work, since not all exception handlers are reachable. It may seem like it would be possible to handle them at once at the end of the non-exception-related analysis, but since an exception handler could only be reachable from another exception handler, it would require further care to ensure correct behaviour, and avoid the complexity of looking up an exception handler repeatedly. This does bring up one limitation of this style of analysis, however.

Jumping to an unreachable bit of code is unsound, since the analysis also doesn't reach these instructions, making it impossible to determine what the state the stack should be in when executing that unreachable code. Even in the compiler itself, exception handlers in unreachable code paths will not have an unwind depth set, thus making them impossible to execute, even if such a jump were to be allowed.

Another problem that arises has to do with the performance in GraalPy. A key factor of the GraalPy interpreter performance is the aforementioned partial unrolling of the interpreter loop. Of course, since the debugger can choose to jump at any time, non-deterministically, that optimization is broken. The solution of choice is to pretend that no jump can happen and discard the compiled node if it does, running the remainder in the interpreter. This way, performance while simply debugging is unaffected by the option to perform these non-deterministic jumps.

4.4 Avoiding performance penalties

As was already mentioned, creating a full frame object for every single frame is slow, and avoided in GraalPy. However, if the global trace function is `None`, the operation can be skipped. Unfortunately, even just checking whether the global trace function is set has severe overhead, as it has to be done for every single bytecode interpreter loop iteration to determine whether to run the (even more expensive) tracing logic. Any line of code could end up invoking `sys.settrace`, and the current frame could have `f_trace` already set.

Fortunately, Truffle is well equipped to deal with these kinds of APIs. The tool of choice here is `Assumption` [24]. The Graal compiler will treat the Assumption as being **valid** and constant fold it as `true`, and Truffle will remember that a given piece of compiled code was compiled with a given `Assumption` in mind. When an `Assumption` is **invalidated**,

Truffle will discard all compiled code depending on the `Assumption`, starting the compilation process over again, this time constant folding it as `false`. This way, compiled code can avoid the overhead of a check if it can be guaranteed to hold up to a certain point.

In the case of tracing, the invalidation criteria is fairly straightforward: the first call to `sys.settrace` with a non-`None` argument invalidates the assumption. It may seem like this would invalidate a large part of the already compiled code, but most of the actual compilation-heavy work happens outside of the `PBytecodeRootNode` itself, inside one of its many child nodes. Nevertheless, the performance while tracing is less than ideal, though performance when not tracing is unaffected.

Unfortunately, since a static path to the `Assumption` must exist, the `Assumption` is shared between all Python interpreters running on the same JVM (more specifically, all Python interpreters using the same `PythonLanguage` instance, which is a singleton). This means that any one of those interpreters starting tracing slows down all the other interpreters. However, the typical use case of multiple interpreters in a single JVM is running multiple workers in parallel, avoiding the GIL, so it is not a severe issue.

4.5 Conclusions

An implementation of the CPython tracing API was implemented for GraalPy, requiring changes to the core interpreter loop, taking advantage of the recent transition to a bytecode interpreter. Rather than to strive for pedantic compatibility with the reference implementation, the effort was placed on keeping the implementation understandable and maintainable. Moreover, no additional overhead is introduced by the inclusion of this API to programs that do not use it, which is critical, since the vast majority of programs are not actively being debugged during their usual operation.

Providing more concrete numbers, this work makes 230 CPython tracing tests pass [37], of the total 308 (excluding 68 OpCode-related tests). Most of the failures happen due to different exception handling bytecode in either implementation, as was described above, or differences in error messages. `PDB`, the standard debugger for python, is made usable with these changes, and has been used in GraalPy development and by GraalPy users.

A tracing API feature was added that allows debuggers and other tools to arbitrarily move control flow to another line, a so-called debugger jump. It is important to ensure these jumps do not break the interpreter, for which an algorithm was adapted from CPython, and 12 instructions which require special handling were identified. Additionally, 13 jump instructions now support accessing the jump destination without evaluating them.

This work has been included in the release 24.0 of GraalPy.

Further work in this area will focus on ensuring individual tools run on GraalPy, for example coverage and the various integrated debuggers in editors. Additionally, the tracing API could be expanded to the new monitoring API, which fixes several defects of the old API.

Chapter 5

Overall Conclusion

In this work, two major features were implemented into GraalPy, both being part of a release and already having had a notable impact on the usability and compatibility of GraalPy. They are also the last major Python features that were previously missing in GraalPy, meaning further work on compatibility will be more focused on individual bugs, rather than implementing major features.

The first of these features was async programming, a way of writing concurrent code that avoids certain pitfalls of true parallelism. This required changing the internals of how generators work in GraalPy, implementing bytecode compilation for new syntax, and porting some parts of the `_asyncio` module, originally written in C, into Java. As is, the Python features and `asyncio` module work enough for 3rd party packages with async features to either work, or reveal further defects in GraalPy that prevent them from working. Some examples are FastAPI and httpx working fully as of 4/2024, but since 3rd party packages are not tested for every change, the set of packages that are supported changes fairly regularly.

The second of these features was the tracing API, the CPython API for writing debuggers, test coverage tools etc. This requires analysis of the bytecode during runtime to determine when a line is executed, requiring adapting the CPython algorithm to GraalPy bytecode. Additionally, in order to support the debugger being able to arbitrarily move control flow, another algorithm for analysis of bytecode is needed, this time tracking the types on the data stack. This algorithm was also adapted from CPython. This has allowed most tools using this API to work in GraalPy as they should, notably `coverage.py`, the primary tool for measuring test coverage, as of 4/2024.

Overall, this work has succeeded in its goals, and has allowed GraalPy to meaningfully expand its support for the Python ecosystem, as well as find further areas of improvement in GraalPy.

Bibliography

[1] *Epoll - I/O event notification facility* [website]. Available at: https://man7.org/linux/man-pages/man7/epoll.7.html.

[2] Ancona, D., Ancona, M., Cuni, A. and Matsakis, N. D. RPython: a step towards reconciling dynamically and statically typed OO languages. In: *Proceedings of the 2007 Symposium on Dynamic Languages*. New York, NY, USA: Association for Computing Machinery, 2007, p. 53–64. DLS '07. DOI: 10.1145/1297081.1297091. ISBN 9781595938688. Available at: https://doi.org/10.1145/1297081.1297091.

[3] ASGI team. *ASGI documentation* [sphinx readthedocs]. Accessed 2023-06-07. Available at: https://asgi.readthedocs.io/en/latest/.

[4] Bagwell, P. *Ideal Hash Trees*. École Polytechnique Fédérale de Lausanne, 2000.

[5] Batchelder, N. *Coverage.py* [source code]. Accessed 2024-01-28. Available at: https://github.com/nedbat/coveragepy.

[6] Batchelder, N. *Coverage.py* [homepage]. Accessed 2024-04-27. Available at: https://pytorch.org/.

[7] Bezanson, J., Karpinski, S., Shah, V. B. and Edelman, A. Julia: A Fast Dynamic Language for Technical Computing. *CoRR*. 2012, abs/1209.5145. Available at: http://arxiv.org/abs/1209.5145.

[8] Bilenko, D. *Gevent* [PyPi package]. Accessed 2024-01-28. Available at: https://pypi.org/project/gevent/.

[9] *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*. Berlin, Heidelberg: Springer-Verlag, 1998. ISBN 3540667105.

[10] Ewing, G. *Syntax for Delegating to Subgenerators* [rst document (english)]. Accessed 2023-06-07. Available at: https://peps.python.org/pep-0380/#formal-semantics.

[11] Futamura, Y. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*. Dec 1999, vol. 12, no. 4, p. 381–391. DOI: 10.1023/A:1010095604496. ISSN 1573-0557. Available at: https://doi.org/10.1023/A:1010095604496.

[12] Google. *V8* [website]. Accessed 2024-01-28. Available at: https://v8.dev/.

[13] GraalPy Team. *GraalPy: Package Compatability* [website]. Accessed 2024-04-27. Available at: https://www.graalvm.org/python/compatibility/.

[14] JetBrains. *PyCharm* [website]. Accessed 2024-04-06. Available at: https://numpy.org/.

[15] Lafont, Y. Interaction nets. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1989, p. 95–108. POPL '90. DOI: 10.1145/96709.96718. ISBN 0897913434. Available at: https://doi.org/10.1145/96709.96718.

[16] Lam, S. K., Pitrou, A. and Seibert, S. Numba: a LLVM-based Python JIT compiler. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. New York, NY, USA: Association for Computing Machinery, 2015. LLVM '15. DOI: 10.1145/2833157.2833162. ISBN 9781450340052. Available at: https://doi.org/10.1145/2833157.2833162.

[17] Lattner, C. and Adve, V. LLVM: a compilation framework for lifelong program analysis & transformation. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, p. 75–86. DOI: 10.1109/CGO.2004.1281665.

[18] LuaJIT team. *LuaJIT project* [website]. Accessed 2024-04-06. Available at: https://luajit.org/.

[19] Microsoft. *Visual Studio Code* [website]. Accessed 2024-04-06. Available at: https://code.visualstudio.com/.

[20] NumPy team. *NumPy* [website]. Accessed 2024-04-06. Available at: https://numpy.org/.

[21] Oracle. *GraalJS* [source code]. Accessed 2024-04-26. Available at: https://github.com/oracle/graaljs.

[22] Oracle. *GraalJS* [source code]. Accessed 2024-04-26. Available at: https://github.com/oracle/truffleruby.

[23] Oracle. *SubstrateVM* [website]. Accessed 2024-04-26. Available at: https://docs.oracle.com/en/graalvm/enterprise/20/docs/reference-manual/native-image/SubstrateVM/.

[24] Oracle. *Truffle Assumption documentation* [javadoc (english)]. Accessed 2023-01-31. Available at: https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/Assumption.html.

[25] Oracle. *Truffle Debugger documentation* [javadoc (english)]. Accessed 2023-01-31. Available at: https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/debug/Debugger.html.

[26] Oracle. *ValueType* [documentation]. Accessed 2024-01-28. Available at: https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/CompilerDirectives.ValueType.html.

[27] Oracle Labs. *GraalPy* [source code]. Accessed 2023-06-07. Available at: https://github.com/oracle/graalpython.

[28] Python Foundation. *Comprehension documentation* [rst document (english)]. Accessed 2023-06-07. Available at: https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions.

[29] Python Foundation. *CPython* [source code]. Accessed 2023-06-07. Available at: https://github.com/python/cpython.

[30] Python Foundation. *CPython test suite* [source code]. Accessed 2024-04-26. Available at: https://github.com/python/cpython/tree/main/Lib/test.

[31] Python Foundation. *Dispatching the tracing return event* [source code]. Accessed 2023-01-31. Available at: https://github.com/python/cpython/blob/3.10/Python/ceval.c#L4524-L4537.

[32] Python Foundation. *Iterator python documentation* [rst document (english)]. Accessed 2023-06-07. Available at: https://docs.python.org/3/library/stdtypes.html#typeiter.

[33] Python Foundation. *Notes on line number tables and tracing* [plaintext]. Accessed 2023-01-31. Available at: https://github.com/python/cpython/blob/3.10/Objects/lnotab_notes.txt.

[34] Python Foundation. *PDB* [rst document (english)]. Accessed 2024-04-06. Available at: https://docs.python.org/3/library/pdb.html.

[35] Python Foundation. *Source code of asend and athrow* [source code]. Accessed 2023-06-07. Available at: https://github.com/python/cpython/blob/main/Objects/genobject.c#L1744-L2370.

[36] Python Foundation. *Sys.settrace documentation* [rst document (english)]. Accessed 2023-01-31. Available at: https://docs.python.org/3.10/library/sys.html#sys.settrace.

[37] Python Foundation. *Test suite for sys.settrace* [source code]. Accessed 2023-01-31. Available at: https://github.com/python/cpython/blob/3.10/Lib/test/test_sys_settrace.py.

[38] Raku team. *Roast* [source code]. Accessed 2024-04-26. Available at: https://github.com/Raku/roast.

[39] Rigger, M., Grimmer, M., Wimmer, C., Würthinger, T. and Mössenböck, H. Bringing Low-Level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In: *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*. New York, NY, USA: Association for Computing Machinery, 2016, p. 6–15. VMIL 2016. DOI: 10.1145/2998415.2998416. ISBN 9781450346450. Available at: https://doi.org/10.1145/2998415.2998416.

[40] Ronacher, A. *How Python was Shaped by leaky Internals*. PyCon Russia. Available at: https://www.youtube.com/watch?v=qCGofLIzX6g.

[41] Rossum, G. van. *PEP 492 vs. PEP 3152, new round* [email (english)]. Accessed 2023-06-07. Available at: https://mail.python.org/pipermail/python-dev/2015-April/139503.html.

[42] Selivanov, Y. *PEP 492 - Coroutines with async and await syntax* [rst document (english)]. Accessed 2023-06-07. Available at: https://peps.python.org/pep-0492/.

[43] Selivanov, Y. *PEP 744 – JIT Compilation* [rst document (english)]. Accessed 2024-04-27. Available at: https://peps.python.org/pep-0744/.

[44] The Trio collective. *Trio* [PyPi package]. Accessed 2024-01-28. Available at: https://pypi.org/project/trio/.

[45] Wimmer, C., Stancu, C., Hofer, P., Jovanovic, V., Wögerer, P. et al. Initialize once, start fast: application initialization at build time. *Proc. ACM Program. Lang.* New York, NY, USA: Association for Computing Machinery. oct 2019, vol. 3, OOPSLA. DOI: 10.1145/3360610. Available at: https://doi.org/10.1145/3360610.

[46] Würthinger, T., Wimmer, C., Wöss, A., Stadler, L., Duboscq, G. et al. One VM to rule them all. In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. New York, NY, USA: Association for Computing Machinery, 2013, p. 187–204. Onward! 2013. DOI: 10.1145/2509578.2509581. ISBN 9781450324724. Available at: https://doi.org/10.1145/2509578.2509581.

[47] Würthinger, T., Wöss, A., Stadler, L., Duboscq, G., Simon, D. et al. Self-optimizing AST interpreters. In: *Proceedings of the 8th Symposium on Dynamic Languages*. New York, NY, USA: Association for Computing Machinery, 2012, p. 73–82. DLS '12. DOI: 10.1145/2384577.2384587. ISBN 9781450315647. Available at: https://doi.org/10.1145/2384577.2384587.