META-TRACING JUST-IN-TIME COMPILATION FOR RPYTHON

Inaugural-Dissertation

zur
Erlangung des Doktorgrades der
Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von

Carl Friedrich Bolz

September 2012

ABSTRACT

Many dynamic languages are implemented using traditional interpreters because implementing just-in-time (JIT) compilers for them is too complex. This limits their performance and restricts their applicability. This thesis describes *meta-tracing*, a technique that makes it possible to apply a single tracing JIT compiler across a wide variety of languages. Meta-tracing is flexible enough to incorporate typical JIT techniques such as run-time type feedback and unboxing. With the help of several case studies, meta-tracing is applied to various language implementations. Performance measurements of using meta-tracing show improvements by up to factor 50 for implementations of production languages. Meta-tracing can thus simplify the implementation of high-performance virtual machines for dynamic languages significantly, making these languages more practical in a wider context.

ZUSAMMENFASSUNG

Viele dynamische Programmiersprachen sind durch klassische Interpreter implementiert, weil das Implementieren von Just-in-Time-Compilern für sie zu komplex ist. Dies schränkt ihre Geschwindigkeit und ihre Anwendbarkeit ein. In dieser Arbeit wird *Meta-Tracing* beschrieben, eine Methode, die es möglich macht, einen einzigen Tracing-JIT-Compiler auf eine große Bandbreite an Sprachen anzuwenden. Der Ansatz ist flexibel genug, typische JIT-Techniken zu ermöglichen, z.B. Laufzeit-Typ-Feedback und Unboxing. Mit Hilfe mehrerer Fallstudien wird Meta-Tracing evaluiert, indem es auf verschiedene Sprachimplementierungen angewendet wird. Geschwindigkeitsmessungen zeigen Verbesserungen von bis zu einem Faktor 50 für Implementierungen von Produktionssprachen. Meta-Tracing kann deshalb die Implementierung von schnellen virtuellen Maschinen für dynamische Programmiersprachen signifikant erleichtern und diese Sprachen so für einen breiteren Kontext anwendbar machen.

ACKNOWLEDGMENTS

During the work on this thesis, I received an unbelievable amount of support and help from many people. Most importantly I'd like to thank my advisor Michael Leuschel for his work of encouraging, supporting, and challenging me over the last four years during my graduate studies and the years before that as a student at the Heinrich-Heine-Universität Düsseldorf. His STUPS group has been a wonderful and fertile home for my work, and I'd like to thank my colleagues David Schneider, Jens Bendisposto, Daniel Plagge, Armin Rigo, and Stefan Hallerstede for many fruitful discussions.

The other important environment that supported me and that I owe gratitude to is Robert Hirschfeld and his group at the Hasso-Plattner-Institute in Potsdam, particularly Malte Appeltauer, Michael Haupt, Jens Lincke, Bastian Steinert, Michael Perscheid, as well as Martin von Löwis. I had the pleasure of visiting the HPI seven times during my graduate studies.

The work of this thesis was only possible due to the big and wonderful community of the PyPy project. When I joined the project in 2005, Holger Krekel warmly welcomed me into the community and helped me make my first contributions. He, Armin Rigo, Samuele Pedroni, Michael Hudson-Doyle, Maciej Fijałkowski, Antonio Cuni, Laura Creighton, Bea Düring, and Jacob Hallén guided and mentored me in various ways over many years and I am extremely thankful for the cooperation we enjoy on a technical level but also (and more importantly) for their friendship. Other important persons in the project that I had and have the pleasure to work with are Amaury Forgeot d'Arc, Alex Gaynor, Christian Tismer, Håkan Ardö, Benjamin Peterson, David Schneider, Eric van Riet Paap, Anders Chrigström, Richard Emslie, Wim Lavrijsen, David Edelsohn, and many more.

On the more academic side I want to thank Kenichi Asai and Olivier Danvy for their wise words, the encouragement they gave me and the discussions we had (and table football!); David Edelsohn, Peng Wu and other people from the IBM Watson research center for inviting me and discussing tracing JITs and dynamic languages; Nikolai Tillman and colleagues for inviting me to Microsoft Research Redmond for a week and telling me many details about the SPUR project.

I had interesting discussions with Hidehiko Masuhara about partial evaluation, reflection, Ruby, and many other things. Theo d'Hondt invited me to present to his group in Brussels leading to more interesting discussions. I'm also thankful to the recurring participants of the "Programmiersprachen und Rechenkonzepte" yearly meeting in Bad Honnef, in particular Bernd Brassel, Jan Christiansen, Sebastian Fischer, Frank Huch, and Baltasar Trancón y Widemann. The regular expression matcher described in Chapter 9 was developed at such a meeting.

Many parts of the thesis appeared as papers in one form or the other. I would like to thank my co-authors Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, Armin Rigo, David Schneider, Laurence Tratt, and reviewers of these papers: Toon Verwaest, Peng Wu, David Edelsohn, Laura Creighton, Stefan Hallerstede, David Schneider, Thomas Stiehl, Sven Hager, and the anonymous reviewers of ICOOOLPS'09, ICOOOLPS'11, PEPM'11, and PPDP'10.

A number of people have given me useful feedback on drafts of this thesis: Stefan Marr, Robert Hirschfeld, Olivier Danvy, Armin Rigo, David Schneider, David Edelsohn, Jens Bendisposto, Michael Leuschel, Jana Tereick, Sven Hager, Maciej Fijałkowski, Laurence Tratt.

This thesis was partially supported by the BMBF funded project PyJIT (nr. 01QE0913B; Eureka Eurostars).

Last not least I'd like to thank my family and friends, in particular my wife Jana Tereick.

CONTENTS

Part I

INTRODUCTION AND MOTIVATION

1 INTRODUCTION

Dynamically typed languages[1] have seen a steady rise in popularity in the last decade. JavaScript is increasingly used to implement full-scale applications, which run within a browser. Dynamic languages are also used for the server side of many Web applications, as well as in various other areas, such as scientific programming, finance, desktop applications, and many more.

The performance penalties their implementations impose is commonly cited [Tra09] as one of the drawbacks of dynamically typed languages. Often they are slower than implementations of statically typed languages. Advanced techniques for improving the performance of dynamic languages are not as widely used as one would expect. Many dynamic language implementations use bytecode-interpreters without advanced implementation techniques like just-in-time (JIT) compilation (which, among other techniques, has been investigated in the Self project [Hö94]).

There are several reasons for the limited use of JIT compilers. Most are due to the inherent complexities of compilation. Interpreters are simple to implement, understand, extend, and port, whereas writing a JIT compiler is an error-prone task that is made harder by the dynamic features of a language. The features that make implementing a dynamic language hard include late binding of name lookups, run-time type dispatching, and boxing of primitive types. Popular languages such as JavaScript, Ruby, PHP, and Python have very complex core semantics with many corner cases, which makes it difficult to implement them efficiently. A JIT compiler for such a language needs to correctly handle the (often unexpected) interactions of all its language features while still generating efficient code for the common case.

The solution proposed in this thesis is *meta-tracing*. Meta-tracing is a language implementation technique that makes it easier to efficiently execute dynamic languages without having to write a dedicated JIT compiler for each of them.

Tracing JITs are a recent approach to write JIT-compilers for dynamic languages with relative ease [GPF06, CBY$^+$07]. Contrary to most other existing tracing JITs, a meta-tracing JIT does not trace the execution of user programs but the execution of the interpreter itself. This property makes meta-tracing JITs applicable to interpreters of different languages. The process of applying a meta-tracing JIT to an

[1] In the rest of the thesis the term *dynamic language* will be used synonymously.

interpreter of a new language is not fully automatic; it needs a small number of hints from the author of the interpreter.

After these basic hints are added, the meta-tracing JIT compiler can map the control flow of the dynamic language to machine code. By adding further hints to the object model supported by the particular interpreter, meta-tracing can also be made especially useful for optimizing object model operations, such as method dispatch. This is usually one of the hardest parts of implementing an object-oriented dynamic language well, made more difficult by the mentioned complexity of the core object semantics of many dynamic languages. Since the meta-tracer traces the execution of the interpreter, the object model implementation is transparent to the tracer and its optimizations. Therefore the semantics of the dynamic language does not have to be replicated or even considered in the JIT. However, bare meta-tracing lacks detailed knowledge about how to optimize the specifics of the object model of the language at hand.

This problem is solved by making two more annotations available to the language implementor. Conceptually, the significant speed-ups that can be achieved with dynamic compilation depend on feeding into compilation values observed at run-time and exploiting them. The annotations provided make it possible to implement such feedback and exploitation in a meta-tracing context. They can be used to express many classic implementation techniques used for object models of dynamic languages, such as run-time type feedback and maps.

Due to the dynamic typing, variables in dynamic languages can potentially store all sorts of objects, even integers, floats, booleans, in addition to instances of user-defined classes. On the implementation side this makes it necessary to give all these types a uniform representation in memory. Therefore those primitive types are usually *boxed*, meaning that a small heap-structure is allocated for them that contains the actual value. Boxing primitive types can be very costly, because a lot of common operations, particularly all arithmetic operations, have to allocate new boxes, in addition to the actual computation.

Type dispatching is the process of finding the concrete implementation that is applicable to the objects at hand when performing a generic operation on them. An example is the addition of two objects: For addition the types of the concrete objects need to be checked and the proper implementation chosen. Type dispatching is a very common operation in modern[2] dynamic languages because types are unknown at compile time. Therefore all operations need it, which makes optimizing them particularly important.

[2] For languages in the Lisp family, basic arithmetic operations are not always overloaded; even in Smalltalk, type dispatching is much simpler than in Python or JavaScript.

The overhead of boxing primitive types and of type dispatching are therefore two further performance problems that many dynamic language implementations have. These are two problems that are important for the efficient execution of dynamically typed languages and that are usually not present or at least less severe in statically typed languages. The third part of the meta-tracing approach is thus a very powerful language-independent optimization to remove the overhead of boxing and type dispatching. The optimization takes traces produced by the meta-tracer and removes unneeded operations from them.

These three elements – enabling meta-tracing, making it possible to express language-specific object semantics and optimizing boxing and type dispatching overhead – are fully implemented in *MetaJIT*. MetaJIT is our implementation of meta-tracing in the context of the PyPy project.

PyPy is trying to find approaches to ease the implementation of dynamic languages. It started as an implementation of Python in Python, but has now extended its goals to be useful for implementing other dynamic languages as well. The general approach is to implement interpreters for the languages in RPython, a restricted subset of Python. This subset is restricted in such a way that programs written in it can be compiled into a C program.

How well MetaJIT and the techniques it implements work, how hard they are to apply, and what effects they have is evaluated in this thesis with several experiments. For these experiments, meta-tracing is applied to three interpreters of different size and for different languages. On the one hand, meta-tracing is applied to a regular expression engine to show a full and useful interpreter in its entirety and how meta-tracing improves it. On the other hand, interpreters for two production languages are presented, one for Prolog and one for full Python. These two dynamic languages are very different: Python is object-oriented and imperative while Prolog is a logical language with unification and backtracking. The complexity of using meta-tracing on these interpreters is described and the effect of meta-tracing on performance is evaluated.

1.1 CONTRIBUTIONS

The main contributions of this thesis are:

- A retargetable JIT compilation approach based on meta-tracing of interpreters,
- user-customizable run-time-feedback within this meta-tracing framework based on annotations that the interpreter author can give, and

- an efficient and effective optimization that removes allocations that do not escape within a trace, based on partial-evaluation techniques.

This thesis was written within the context of the PyPy project. Most of the ideas and the technical work are the result of a group effort by me and a number of other people. This thesis now takes this technological base and evaluates it within a scientific context. Some chapters of this thesis have been published in earlier venues. The publication history of these chapters can be found in Appendix B.

1.2 STRUCTURE OF THE THESIS

Part I of the thesis introduces dynamic languages, the PyPy project, and tracing JITs in Chapter 2.

Part II of the thesis presents meta-tracing and MetaJIT, its implementation within in the RPython framework. The explanations are accompanied by example interpreters of increasing complexity.

- A description of how to apply a tracing JIT compiler to an interpreter to achieve meta-tracing is given in Chapter 4.
- Chapter 5 gives a description of two hints that can be applied to improve the object model of a language by giving the user control over run-time feedback, together with examples for how classical VM techniques are expressed with these hints.
- Chapter 6 contains a description of a practical, efficient, and effective optimization that can be applied to traces to remove object allocations. The algorithm is characterized as partial evaluation.

Part III of the thesis evaluates meta-tracing in various ways:

- A simple case study to apply meta-tracing to a regular expression matcher is given in Chapter 9.
- A thorough examination of the effects of meta-tracing on PyPy's Python interpreter, as well as the optimizations performed on it, is given in Chapter 10.
- A Prolog interpreter written in RPython to answer the question whether meta-tracing is also applicable to non-imperative languages is described in Chapter 11.
- Chapter 12 contains a comparison of tracing and meta-tracing with partial evaluation. This is done by writing executable models of both for a trivial imperative language in Prolog and comparing them conceptually.

Part IV presents related work and concludes the thesis.

2 BACKGROUND

2.1 DYNAMIC LANGUAGES AND THEIR IMPLEMENTATION

While not everyone agrees about what constitutes a dynamic language [Tra09], commonly agreed on elements include dynamic (but strong) typing, garbage collection (GC), late-binding and reflection. Notable examples of dynamic languages are Smalltalk [Gol83], Lisp [McC60], Self [US87], Python,[1] PHP,[2] Ruby,[3] JavaScript [ECM99], R [IG96], and others.

Dynamic languages have recently shown a boost of popularity, due to their widespread use in Web programming. This is true both for the client in the form of JavaScript in the browser, as well as on the server. Other areas where dynamic languages are used are scientific programming, finance, desktop applications, scripting, and many more.[4]

One dynamic language that will be particularly important in this thesis is Python. Python was invented in the late 1980s by Guido van Rossum at the National Research Institute for Mathematics and Computer Science in Amsterdam.[5] Python is a dynamically (but strongly) typed object-oriented imperative language. It supports multiple inheritance using the C3 algorithm for superclass linearization [BCH+96], uses a simple metaclass model very similar to that of ObjVlisp [Coi87] and has powerful container types built directly into the language.

Implementing dynamic languages efficiently is historically a hard problem. Many dynamic languages such as Lisp, Smalltalk, and Self have been designed for purity and consistency of the feature-set, not for speed. Some of the problems for efficiency when implementing dynamic languages are:

LATE BINDING Most dynamic languages use late binding, which means that most name lookups can only be done at run-time. This makes it very hard to know at compile-time what to or where a name will be bound.

DISPATCHING An important special case of late binding is dispatching. Dispatching is the process of finding the right implementation for a generic operation on objects. Dynamic languages have

[1] http://python.org
[2] http://www.php.net/
[3] http://www.ruby-lang.org/en/
[4] http://python.org/about/apps/
[5] http://python-history.blogspot.com/2009/01/brief-timeline-of-python.html

complex dispatching rules, often with user-changeable operation implementations which makes it hard to know in advance which implementation to choose.

BOXING Due to the dynamic typing of dynamic languages, it is very hard to know in advance what the type of an object will be. Therefore all objects need a compatible representation in heap memory, even primitive data types such as integers and floating point numbers. This means that all operations on such primitives produce new objects on the heap, which increases GC pressure significantly.[6]

Due to these problems it has taken a lot of research to bring dynamic languages to acceptable performance levels. Early results for Smalltalk [DS84] and Lisp [SJG93] were followed by intense research around the Self language [CUE89, HU94, Hö94], which then got applied to the Hotspot VM [PVC01]. All these results use a JIT compiler [Ayc03] to compile and recompile parts of the program at runtime, adapting the generated code to the situation and use-case at hand. This run-time adaption is necessary due to the inherent nature of dynamic languages. Attempts to get interesting static information for languages like Python have produced limited results [Can05] or are quite complex and don't scale well [Sal04].

However, implementing a JIT compiler is a really hard problem, particularly when the language to be compiled is itself complex. In the Python case, this can be seen very well by looking at the Psyco project [Rig04]. Psyco is a specializing JIT compiler for Python. It was first released when Python 2.2 was the current version and has been adapted to Python 2.3 – 2.6. Then, development could no longer be sustained due to the increasing speed of Python development, therefore Python 2.7 and the 3.x series are not supported by Psyco. Furthermore, Psyco never produced consistent speedup over a variety of Python programs.

Since the industry widely adopted either the JVM [Gos05] or the CLR [ECM10] as virtual machines, a lot of attempts have been made to use these VMs as an implementation substrate for dynamic languages as well. For example, both Python and Ruby have implementations on both of these VMs: Jython[7] and JRuby[8] are implementations on top of the JVM whereas IronPython[9] and IronRuby[10] are implementations on top of the CLR.

[6] A different approach for representing primitive types is using a tagged representation [Gud93]. Tagging representations have their own set of advantages and disadvantages. Usually only one primitive data type (for example integers or floats) can be tagged in a VM. Tagging is ignored in this thesis and will be subject of a later study.

[7] http://www.jython.org/

[8] http://jruby.org/

[9] http://ironpython.net/

[10] http://ironruby.net/

This work has been done mostly for interoperability reasons, but in theory the approach should give good performance as well. The underlying VM provides an extremely well-tuned GC and JIT. However, it has turned out that implementing a dynamic language efficiently on top of these VMs is a non-trivial task. Most dynamic languages have semantics that cannot be directly mapped to that of the underlying VM, which means that the JIT often does not improve the performance of the language much [SS02]. The introduction of the `invokedynamic` bytecode on the JVM [Ros09] might change that, but it remains to be seen how much it helps in practice.

2.2 THE PYPY PROJECT AND RPYTHON

The PyPy project[11] [RP06, BR07] is a framework that supports the writing of flexible implementations of dynamic languages. The project was started in 2003 by Holger Krekel, Christian Tismer, and Armin Rigo, with Laura Creighton, Michael Hudson and others soon joining (I joined the project in fall 2005). In July 2005 the Python interpreter was bootstrapped successfully for the first time. PyPy 1.0 was released in March 2007, which contained the first version of the JIT compiler (not based on tracing). In March 2010, the first release containing the current meta-tracing JIT compiler MetaJIT was published, PyPy 1.2.[12] At the time of writing, PyPy 1.9 is the last released version.

To implement a dynamic language with PyPy, an interpreter for that language has to be written in RPython [AACM07]. RPython ("Restricted Python") is a subset of Python for which type inference can be performed. The language interpreter can then be translated with the help of the RPython translation toolchain into various target environments, most importantly C/Posix (but also the CLI and the JVM, experimentally).

By writing VMs in a high-level language, the implementation of the language is kept free of low-level details, such as memory management strategy, threading model, or object layout. These features are orthogonal to the language semantics and are automatically woven into the generated code during the translation process. This process starts by performing control flow graph construction and type inference, which are followed by a series of transformational steps. Each step lowers the abstraction level of the intermediate representation until C code can be generated directly. The first transformation step makes details of the RPython object model explicit in the intermediate representation, later steps introduce garbage collection and other low-level details.

[11] `http://pypy.org`

[12] `http://morepypy.blogspot.de/2010/12/we-are-not-heroes-just-very-patient.html`

One of the low-level details that can be inserted during translation is MetaJIT, a language-independent tracing JIT. Describing how this works is the core contribution of this thesis.

The advantage of using a generic JIT-compiler is that writing an interpreter is much easier and less error prone than writing a compiler for every language by hand. Similarly, writing in a high level language such as RPython is easier than writing in C. Also, RPython being a subset of Python significantly eases testing. Usually an interpreter is first written as a normal Python program and can be unit-tested during development. Only when the interpreter is sufficiently complete translation to C is attempted.

A number of languages have been implemented with RPython, most importantly a full Python implementation (see Chapter 10), but also a Prolog interpreter (see Chapter 11) and some experiments (see Chapter 9 for one of them), such as a Smalltalk VM [BKL^{+}08], a GameBoy emulator [BV09], a PHP implementation,[13] and an R interpreter.[14] Of these, the Python interpreter is the biggest and the most complete. It will be used throughout the thesis to study the effectiveness of the described techniques.

An overview of the translation process can be seen in Figure 1, using the Prolog interpreter described in Chapter 11 as an example. The Prolog interpreter is written in RPython (1), which can then be translated into C code (2). During the translation, MetaJIT will be inserted into the resulting VM. If a Prolog program (3) is executed by this interpreter, the interpreter will run it and MetaJIT will emit machine code (4) for parts of the program. Those machine code snippets will be executed at run-time, and can transfer control to each other, as well as back to the interpreter.

2.2.1 *The language RPython*

As mentioned before, the language that is used within PyPy to implement the language interpreters is called *RPython*, *Restricted Python* [AACM07]. It is a subset of the Python language, chosen in such a way that type inference on RPython programs is possible. As its main restriction, in RPython it is not allowed to mix types at the same location in the program. For example, it would not be valid RPython to have a function that accepts both integers and strings as its first argument. In addition, RPython forbids run-time reflection (like changing methods of classes at run-time), full multiple inheritance and the use of most operator overloading. Despite the restrictions RPython is still quite an expressive object-oriented high-level language, supporting garbage collection, exceptions, single inheritance with mixins [BC90], dynamic dispatch, and good built-in

[13] https://bitbucket.org/fijal/hippyvm/

[14] https://bitbucket.org/cfbolz/rapydo/

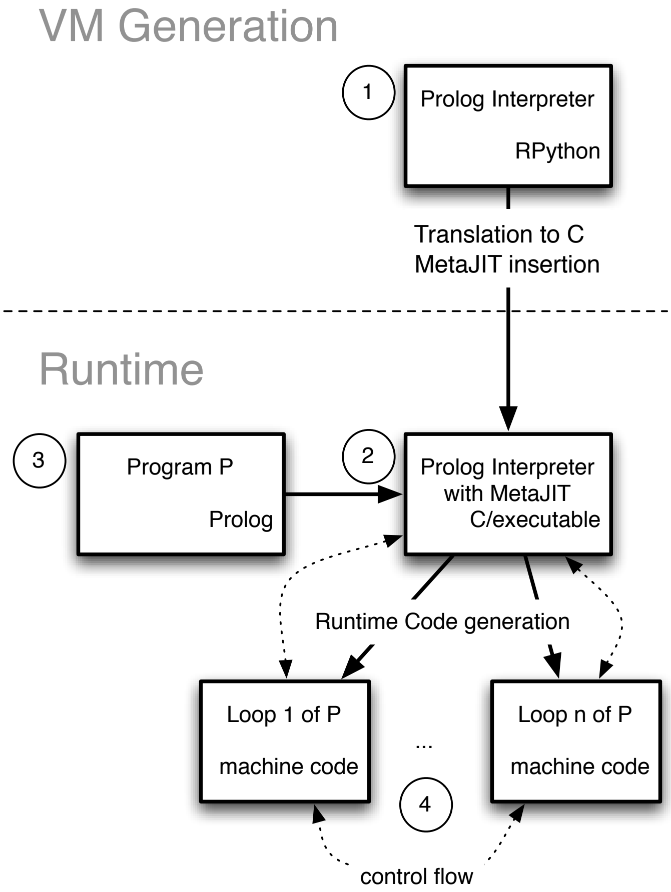


Figure 1: Building a VM with RPython

data-structures. This makes it very different from Slang, a Smalltalk dialect used for the Squeak Smalltalk implementation [IKM+97]) and PreScheme, a Scheme dialect used for the Scheme48 Scheme implementation [KR94]). These languages have semantics very close to that of C, with Smalltalk and Scheme syntax respectively.

Since type inference can be performed on RPython programs, it is possible to translate them into an efficient C program. The C program can then be turned into an executable and be run. Many aspects of the final executable are not apparent in the original interpreter. Since RPython has automatic memory management and C does not, a garbage collector needs to be inserted during the translation process.

2.2.2 *Garbage collection*

The most effective garbage collector used by RPython is a generational copying collector [JHM12]. Objects are first allocated in a nursery generation. If they survive a minor collection, they are moved to an old object generation. This old generation is collected using mark-and-sweep, so that long-living objects do not have to be copied at every collection. References from older to younger objects are detected with the help of a write barrier.

Since the collector is using a nursery, allocation is extremely fast, essentially just incrementing and comparing a pointer. The GC is also very efficient at dealing with high allocation rates, as long as most objects die very quickly.

RPython's garbage collector is itself written in RPython [RP06]. This is similar to how MMTk [BCM04] allows to write garbage collectors in Java for the Jikes RVM virtual machine [AAB+00]. This approach is sometimes called *high-level low-level programming* [DSP+09].

2.3 TRACING JITS

This thesis is about inserting the MetaJIT aspect into the final VM during translation. The details of how this works will be subject of the rest of this thesis. MetaJIT is a tracing JIT compiler. This section will now give a general introduction into what tracing JITs are.

Tracing optimizations were initially explored by the Dynamo project [BDB00] to dynamically optimize machine code at run-time. Its techniques were then successfully used to implement JIT compilers for Java VMs [GPF06, GF06, BCW+10, IHWN11]. Subsequently these tracing JITs were discovered to be a relatively simple way to implement JIT compilers for dynamic languages [CBY+07]. The technique was used by both Mozilla's TraceMonkey JavaScript VM [GES+09] and has been tried for Adobe's Tamarin ActionScript VM [CSR+09]. Probably the fastest tracing JIT for a dynamic lan-

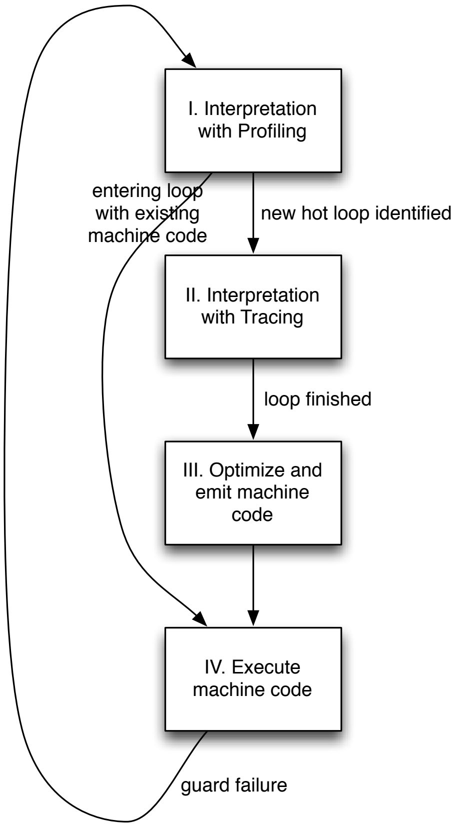

I. Interpretation with Profiling
entering loop with existing machine code
new hot loop identified
II. Interpretation with Tracing
loop finished
III. Optimize and emit machine code
IV. Execute machine code
guard failure

Figure 2: The stages of execution when using a tracing JIT compiler

guage is LuaJIT,[15] an open source VM for the Lua language written by Mike Pall.

Tracing JITs are built on the following basic assumptions:

- Programs spend most of their running time in loops.
- Several iterations of the same loop are likely to take similar code paths.

The basic approach of a tracing JIT is to only generate machine code for the hot code paths of commonly executed loops and to interpret the rest of the program. The code for those common loops however is highly optimized, including aggressive inlining and path splitting.

A VM with a tracing JIT is typically a mixed-mode execution environment, containing both an interpreter and a JIT compiler.[16] Executing code in such a VM leads to several execution stages, which are shown in Figure 2. In the beginning (phase I), all code is executed by the interpreter, which also performs some lightweight profiling to identify hot loops of the program. This lightweight profiling is

[15] `http://luajit.org`

[16] VMs where the interpreter is replaced by a simple compiler have also been described [BCW$^+$10, BBF$^+$10] but the fundamental mechanisms are the same.

usually done by having a counter on each backward jump instruction that counts how often this particular backward jump is executed. Since loops need a backward jump somewhere, this method looks for loops in the user program.

If a hot loop is found, the interpreter enters a special *tracing* mode (phase II), where all operations that the interpreter performs during the execution of that loop are recorded. The recorded operations are stored in a linear list called the *trace*. Tracing continues until the interpreter has recorded the execution of one iteration of the hot loop. To decide when this is the case, the trace is repeatedly checked as to whether the interpreter is at a position in the program where it had been earlier. This recording automatically inlines functions: when a function call is encountered the operations of the called functions are simply put into the trace of the caller too.

Such a trace can be used to generate efficient machine code (phase III). This generated machine code is immediately executable, and can be used in the next iteration of the loop (phase IV). The generated machine code is also cached and if the interpreter later wants to execute the same loop again, it will switch to running the compiled code instead.

Being sequential, the trace represents only one of the many possible paths through the code. To ensure correctness, the trace contains a *guard* at every possible point where the control flow could have followed another path, for example at conditions and indirect or virtual calls. When generating the machine code, every guard is turned into a quick check to guarantee that the path that is being executed is still valid. When the generated machine code is later executed and a guard fails, machine code execution is stopped and execution continues by falling back to interpretation. These guards are the only mechanism to stop the execution of a trace, the loop end condition also takes the form of a guard.

If one specific guard fails often enough, another trace is generated starting from that specific point in the program and the existing machine code is patched to jump to that new trace [GF06]. These and related mechanisms of selecting traces and connecting them to each other [HHS05, WHIN11] are mostly orthogonal to the issues discussed in this thesis and won't be discussed further.

It is important to understand how the tracer recognizes that the trace it recorded so far corresponds to a loop. This happens when the *position key* is the same as at an earlier point in the trace. The position key describes the position of the execution of the program, i.e., usually contains things like the function currently being executed and the program counter position of the interpreter. The tracer does not need to check all the time whether the position key already occurred earlier, but only at instructions that are able to change the position key to an earlier value, for example a backward branch instruction. Note

```
def f(a, b):
    if b % 46 == 41:
        return a - b
    else:
        return a + b
def strange_sum(n):
    result = 0
    while n >= 0:
        result = f(result, n)
        n -= 1
    return result

# corresponding trace:
[$result_0$, $n_0$]
# inside result = f(result, n)
    $i_0$ = int_mod($n_0$, 46)
    $i_1$ = int_eq($i_0$, 41)
    guard_false($i_1$)
    $result_1$ = int_add($result_0$, $n_0$)
$n_1$ = int_sub($n_0$, 1)
$i_2$ = int_ge($n_1$, 0)
guard_true($i_2$)
jump($result_1$, $n_1$)
```

Figure 3: A simple RPython function and the recorded trace

that this is already the second place where backward branches are treated specially: during interpretation they are the place where the profiling is performed and where tracing is started or already existing machine code executed; during tracing they are the place where the check for a closed loop is performed.

One risk that a tracing JIT faces is that in theory an exponential number of paths through one loop can exist (for example if the loop contains n consecutive `if` statements, there are 2^n paths). In practice, this occurs rarely and is only a problem if all the paths are equally likely to be executed, because otherwise only the common paths will be traced.

2.3.1 *A tracing example*

As a small example, take the (somewhat contrived) RPython code in Figure 3. The tracer interprets these functions in a bytecode format that is an encoding of the intermediate representation of the RPython translation toolchain after type inference has been performed. When the profiler discovers that the `while` loop in `strange_sum` is executed often the tracing JIT will start to trace the execution of that loop. The trace would look as in the lower half of Figure 3.

The trace contains all the operations that were executed. The operations are in SSA-form [CFR+91], meaning every variable is assigned to only once. The variables $result_0$ and n_0 that are live when entering the trace are given in square brackets. They are also called the arguments of the trace. The trace ends with a jump operation that gets as arguments the new values of these arguments. The operations in the sequence are operations of the above-mentioned intermediate representation (for example the generic modulo and equality operations in the function above have been recognized to always take integers as arguments and are thus rendered as `int_mod` and `int_eq`). The trace forms an endless loop that can only be left via a guard failure. The call to `f` is inlined into the trace, the resulting operations are indented in the figure. The trace contains only the hot `else` case of the `if` test in `f`, with a guard marking the existence of the other path. After the trace has been recorded, it can then be converted into machine code and executed.

2.3.2 *Optimization and code generation*

Before sending the trace to the backend to produce actual machine code, it is optimized. The optimizer uses a number of techniques to remove or simplify the operations in the trace. Most of these are well known compiler optimization techniques, with the difference that it is easier to apply them in a tracing JIT because they only have to deal with linear traces, not arbitrary control flow. Among the common techniques are constant folding, common subexpression elimination, allocation removal (see Chapter 6), store/load propagation, loop invariant code motion [ABF12, MM97].

The fact that traces are linear pieces of code without control flow joins (except at the beginning of the trace) makes many optimizations a lot more tractable, and the inlining that happens gives the optimizations automatically more context to work with.[17]

After optimization, machine code can be generated. This is also a very straightforward process again due to the linearity, which greatly simplifies register allocation and instruction selection. When generating machine code, every guard is turned into a quick check to see whether the assumption still holds.

[17] This is similar to how some classical compiler optimization techniques use extended basic blocks as their optimization scope. "Inside an [extended basic block] the compiler can use facts discovered in earlier basic blocks to improve code in later blocks. Superlocal methods can treat the individual paths through an [extended basic block] as if they were in a single block." [CT04, p. 405]

Part II

META-TRACING

3 A HIGH-LEVEL VIEW OF META-TRACING

This chapter gives a brief overview of meta-tracing. Subsequent chapters provide more detailed descriptions of the techniques.

When using RPython to implement a dynamic language, there are two interpreters involved. First, there is the interpreter for the dynamic language to be implemented, the *language interpreter*. The program that the language interpreter executes is the *user program* (from the point of view of a VM author, the "user" is a programmer using the VM). Second, there is the interpreter used by MetaJIT to perform tracing, the *tracing interpreter*. Similarly, since tracing JITs are concerned with loops it is important to distinguish loops at two different levels: *interpreter loops* are loops inside the language interpreter and *user loops* are loops in the user program.

The techniques of normal tracing JITs described in the previous chapter cannot be used directly in the RPython context but need to be adapted. A normal tracing JIT is explicitly written for one specific language. It directly traces the execution of the user program with the help of an extra component which is part of the language interpreter. Therefore the language's semantics is hard-coded into such a tracing JIT. This makes it necessary to write a tracing JIT specifically for every language to be implemented.

A schematic diagram of a normal tracing JIT is given on the left side of Figure 4. It shows a tracing JIT for a language *L*. The CPU executes the interpreter and tracer, which in turn execute and trace a program in *L* written by a programmer, the user. The tracer is a component that was specifically written for the *L* language as part of the VM. Tracing the *L* program yields a trace consisting of operations on the level of the bytecode of the interpreter for *L*, which are specific to that interpreter. The operations correspond to the bytecodes of the loop in function `f1` that was traced, together with bytecodes from `f3` and `f4` which were inlined into the trace.

RPython is intended to be a general framework for implementing dynamic languages, therefore the approach of RPython has to be different. RPython contains a language-independent tracing JIT compiler called MetaJIT. MetaJIT is a (modified) tracer for programs in RPython. It traces the execution of the *language interpreter* while the interpreter is running the program. The traces consist of RPython-level operations of the interpreter that were used to execute the user program. This approach is therefore called *meta-tracing*, because the tracer operates on the implementation level.

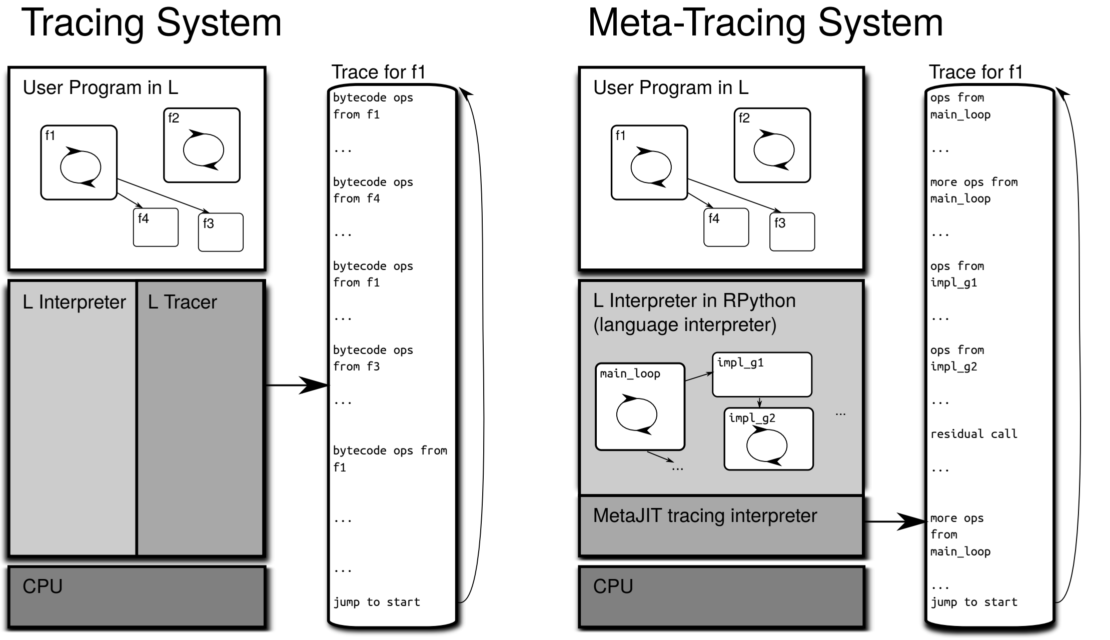


Figure 4: The components involved in tracing and meta-tracing of a language *L*

The right side of Figure 4 shows the components involved in the meta-tracing process. The CPU executes MetaJIT. MetaJIT contains an interpreter that executes and traces the interpreter for the language *L*, which consists of functions written in RPython. The trace consists of the RPython operations in the interpreter for *L*.

If MetaJIT is used, it traces operations of the language interpreter. Tracing continues until a full loop in the user program is traced. Tracing the execution of an interpreter has many advantages. It makes the tracer, its optimizers, and backends applicable to a variety of languages. The language semantics does *not* need to be hard-coded into MetaJIT. Instead, MetaJIT produces traces that behave as the interpreter does. Therefore MetaJIT by construction supports the full language that the interpreter implements. Also, MetaJIT can be used for different languages by applying it to different language interpreters.

For tracing an interpreter it is not enough to write a normal tracing JIT for the RPython language. The bytecode dispatch loop is usually executed significantly more often than all other interpreter loops. One iteration of the bytecode dispatch loop corresponds to the execution of one bytecode. Tracing one single iteration of the bytecode dispatch loop would therefore not produce useful traces, because usually the same bytecode instruction is not executed several times in a row. A meta-tracer must instead trace many such iterations, until the trace corresponds to one user loop. To do that, the tracer needs information provided by the interpreter author, which is given in the form of explicit hints in the source of the interpreter. The details of these hints will be discussed in the next chapter .

Meta-tracing solves many of the problems of dynamic language implementation. By observing what actually happens at run-time, the tracer automatically compiles only the few common paths through the complex late binding and dispatching semantics of a dynamic language. The rules for late binding and dispatching are transparent to the tracer because it only traces the actions of the interpreter. Sometimes it is desirable to influence the tracing of late binding and dispatching, which can be done with another set of hints. These are discussed in Chapter 5.

A tracer splits control flow paths very aggressively. Control flow merges can happen only at the beginning of a loop. This has two effects. On the one hand, subsequent dispatches on the same object are automatically optimized. On the other hand, boxing overhead can be reduced. This optimization will be discussed in Chapter 6.

4 META-TRACING INTERPRETERS

4.1 APPLYING A TRACING JIT TO AN INTERPRETER

MetaJIT, RPython's tracing JIT is atypical in that it is not applied to the user program, but to the interpreter running the user program. In this section the problems this brings will be explored, and approaches how to solve them (at least partially) will be presented.

A tracing JIT compiler finds the hot loops of the program it is compiling. In the case of MetaJIT, this program is the language interpreter. This chapter assumes that the language interpreter is bytecode-based (but that is not a fundamental restriction, as will be seen in Chapter 9 and Chapter 11). The most important hot loop in the interpreter is the bytecode dispatch loop (for many simple interpreters it is also the only hot loop). One iteration of this loop corresponds to the execution of one opcode. This means that the assumption made by tracing JITs – that several iterations of a hot loop take the same or similar code paths – is wrong in this case. It is very unlikely that the same particular opcode is executed many times in a row.

An example interpreter that will be used to explain the principles of meta-tracing is given in Figure 5. It shows the code of a very simple bytecode interpreter with 256 registers and an accumulator. The `bytecode` argument is a string of bytes, all register and the accumulator are integers.[1] A program for this interpreter that computes the square of the accumulator is shown in Figure 6.

If the tracing interpreter traces the execution of the `DECR_A` opcode (whose integer value is 7), the trace would look as in Figure 7. Because of the guard on $opcode_0$, the code compiled from this trace will be useful only when executing a long series of `DECR_A` opcodes. For all the other operations the guard will fail and the trace is left, which will mean that performance is not improved at all.

To improve this situation, the tracing JIT could trace the execution of several opcodes, thus effectively unrolling the bytecode dispatch loop. Ideally, the bytecode dispatch loop should be unrolled exactly so much that the unrolled version corresponds to one *user loop*. User loops occur when the program counter that the *language interpreter* uses has the same value several times. This program counter is typically stored in one or several variables in the language interpreter, for example the bytecode object of the currently executed function of the user program and the position of the current bytecode within

[1] The chain of `if`, `elif`, ... instructions checking the opcodes is turned into a `switch` statement by one of RPython's optimizations. Python lacks a `switch` statement.

```
def interpret(bytecode, a):
    regs = [0] * 256
    pc = 0
    while True:
        opcode = ord(bytecode[pc])
        pc += 1
        if opcode == JUMP_IF_A:
            target = ord(bytecode[pc])
            pc += 1
            if a:
                pc = target
        elif opcode == MOV_A_R:
            n = ord(bytecode[pc])
            pc += 1
            regs[n] = a
        elif opcode == MOV_R_A:
            n = ord(bytecode[pc])
            pc += 1
            a = regs[n]
        elif opcode == ADD_R_TO_A:
            n = ord(bytecode[pc])
            pc += 1
            a += regs[n]
        elif opcode == DECR_A:
            a -= 1
        elif opcode == RETURN_A:
            return a
```

Figure 5: A very simple bytecode interpreter with registers and an accumulator

```
MOV_A_R     0   # i = a
MOV_A_R     1   # copy of ’a’

# 4:
MOV_R_A     0   # i--
DECR_A
MOV_A_R     0

MOV_R_A     2   # res += a
ADD_R_TO_A  1
MOV_A_R     2

MOV_R_A     0   # if i!=0: goto 4
JUMP_IF_A   4

MOV_R_A     2   # return res
RETURN_A
```

Figure 6: Example bytecode: Compute the square of the accumulator

```
[a0, regs0, bytecode0, pc0]
opcode0 = strgetitem(bytecode0, pc0)
pc1 = int_add(pc0, 1)
guard_value(opcode0, 7)
a1 = int_sub(a0, 1)
jump(a1, regs0, bytecode0, pc1)
```

Figure 7: Trace when executing the DECR_A opcode

that. In the example above, the program counter is represented by the bytecode and pc variables.

Since MetaJIT cannot know which variables of the language interpreter are the program counter, the author of the language interpreter needs to mark the relevant variables with the help of a *hint*. The tracing interpreter will then effectively add the values of these variables to the position key. This means that the loop will only be considered to be closed if these variables that are making up the program counter at the language interpreter level are the same a second time. Loops found in this way are, by definition, user loops.

The program counter used in the language interpreter can only be the same a second time after an instruction in the user program sets it to an earlier value. This happens only at backward jumps in the language interpreter. That means that the tracing interpreter needs to check for a closed loop only when it encounters a backward jump in the language interpreter. Again MetaJIT cannot know which part of the language interpreter implements backward jumps, so the author of the language interpreter needs to indicate this with the help of another hint.

The language interpreter uses a similar technique to detect *hot user loops*: the profiling is done at the backward branches of the user program. Every seen value of the program counter of the language interpreter gets a counter that is incremented when that backward branch is executed. A loop is deemed hot when that counter grows over a certain threshold.

The condition for reusing existing machine code also needs to be adapted to this new situation. In a classical tracing JIT there is no or one piece of machine code per loop of the jitted program, which in our case is the language interpreter. When applying MetaJIT to the language interpreter as described so far, *all* pieces of machine code correspond to the bytecode dispatch loop of the language interpreter. However, they correspond to different paths through the loop and different ways to unroll it. To find out which of them to use when trying to enter machine code again, the program counter of the language interpreter needs to be checked. If it corresponds to the position key of one of the pieces of machine code, then this machine code can be executed. This check again only needs to be performed at the backward branches of the language interpreter.

```
tlrjitdriver = JitDriver(greens = [’pc’, ’bytecode’],
                         reds   = [’a’, ’regs’])

def interpret(bytecode, a):
    regs = [0] * 256
    pc = 0
    while True:
        tlrjitdriver.jit_merge_point()
        opcode = ord(bytecode[pc])
        pc += 1
        if opcode == JUMP_IF_A:
            target = ord(bytecode[pc])
            pc += 1
            if a:
                if target < pc:
                    tlrjitdriver.can_enter_jit()
                pc = target
        elif opcode == MOV_A_R:
            ... # rest unmodified
```

Figure 8: Simple bytecode interpreter with hints applied

Let’s look at how hints would need to be applied to the example interpreter from Figure 5. Figure 8 shows the relevant parts of the interpreter with hints applied. One needs to instantiate `JitDriver` by listing all the variables of the bytecode loop. The variables are classified into two groups, “green” variables and “red” variables. The green variables are those that MetaJIT should consider to be part of the program counter of the language interpreter. In the case of the example, the `pc` variable is obviously part of the program counter; however, the `bytecode` variable is also counted as green, since the `pc` variable is meaningless without the knowledge of which bytecode string is currently being interpreted. All other variables are red.

In addition to the classification of the variables, there are two methods of the `JitDriver` instance that need to be called. The first one is `jit_merge_point` which needs to be put at the beginning of the body of the bytecode dispatch loop. The other, more interesting one, is `can_enter_jit`. This method needs to be called at the end of any instruction that can set the program counter of the language interpreter to an earlier value.[2] For the example this is only the `JUMP_IF_A` instruction, and only if it is actually a backward jump. Here is where the language interpreter performs profiling to decide when to start tracing or whether to jump to an already existing piece of machine code. It is also the place where MetaJIT checks whether a loop is closed. This is considered to be the case when the values of the green variables are the same as at an earlier call to the `can_enter_jit` method.[3]

[2] The hints need to be written a bit differently in the actual implementation for purely technical reasons. Examples of this will be shown in the case studies, Chapters 9 and 11.

[3] For convenience, the `can_enter_jit` hint can be left out as well, in which case the meta-tracer considers it to be directly in front of the `jit_merge_point`.

For the small example the hints look like a lot of work. However, the number of hints that need to be put into the interpreter source remains small, which makes the extra work negligible for larger interpreters.

When executing the square function of Figure 6, the profiling will identify the loop in the square function to be hot, and start tracing. It traces the execution of the interpreter running the loop of the square function for one iteration, thus unrolling the interpreter loop of the example interpreter eight times, corresponding to the eight bytecodes in the loop of the square function. The resulting trace can be seen in Figure 9.

4.2 CONSTANT FOLDING PARTS OF THE TRACE

The critical problem of tracing the execution of just one opcode has now been solved, the loop corresponds exactly to the loop in the square function. However, the resulting trace is not yet very optimal. Most of its operations are not doing any computation that is part of the square function. Instead, they manipulate the data structures of the language interpreter. This effect is to be expected, given that the tracing interpreter looks at the execution of the language interpreter. However, to reach good performance it is necessary to remove some of these operations.

The simple insight on how to improve the situation is that most of the operations in the trace are concerned with manipulating the bytecode string and the program counter. Those are stored in variables that are green, meaning that they are part of the position key. This means that the tracer checks that those variables have some fixed value at the beginning of the loop (they may well change over the course of the loop, though). In the example of Figure 9 the check would be that at the beginning of the trace the `pc` variable is `4` and the `bytecode` variable is the bytecode string corresponding to the square function. Therefore it is possible to constant-fold computations on them away, as long as the operations are pure. Since strings are immutable in RPython, it is possible to constant-fold the `strgetitem` operation. The `int_add` operations are additions of the green variable `pc` and a constant number, so they can be folded away as well.

With this optimization enabled, the trace looks as in Figure 10. Now much of the language interpreter is gone from the trace and what is left corresponds very closely to the loop of the square function. The only vestige of the language interpreter is the fact that the register list is still used to store the state of the computation. This can be removed by some other optimization, which is not described here. Once the optimizer has optimized the trace, the JIT backend generates the corresponding machine code for it.

```
[a0, regs0, bytecode0, pc0]
# MOV_R_A 0
opcode0 = strgetitem(bytecode0, pc0)
pc1 = int_add(pc0, 1)
guard_value(opcode0, 2)
n1 = strgetitem(bytecode0, pc1)
pc2 = int_add(pc1, 1)
a1 = list_getitem(regs0, n1)
# DECR_A
opcode1 = strgetitem(bytecode0, pc2)
pc3 = int_add(pc2, 1)
guard_value(opcode1, 7)
a2 = int_sub(a1, 1)
# MOV_A_R 0
opcode2 = strgetitem(bytecode0, pc3)
pc4 = int_add(pc3, 1)
guard_value(opcode2, 1)
n2 = strgetitem(bytecode0, pc4)
pc5 = int_add(pc4, 1)
list_setitem(regs0, n2, a2)
# MOV_R_A 2
...
# ADD_R_TO_A 1
opcode4 = strgetitem(bytecode0, pc7)
pc8 = int_add(pc7, 1)
guard_value(opcode4, 5)
n4 = strgetitem(bytecode0, pc8)
pc9 = int_add(pc8, 1)
i0 = list_getitem(regs0, n4)
a4 = int_add(a3, i0)
# MOV_A_R 2
...
# MOV_R_A 0
...
# JUMP_IF_A 4
opcode7 = strgetitem(bytecode0, pc13)
pc14 = int_add(pc13, 1)
guard_value(opcode7, 3)
target0 = strgetitem(bytecode0, pc14)
pc15 = int_add(pc14, 1)
i1 = int_is_true(a5)
guard_true(i1)
jump(a5, regs0, bytecode0, target0)
```

Figure 9: Trace when executing the square function of Figure 6, with the corresponding bytecodes as comments.

```
[$a_0$, $regs_0$]
# MOV_R_A 0
$a_1$ = list_getitem($regs_0$, 0)
# DECR_A
$a_2$ = int_sub($a_1$, 1)
# MOV_A_R 0
list_setitem($regs_0$, 0, $a_2$)
# MOV_R_A 2
$a_3$ = list_getitem($regs_0$, 2)
# ADD_R_TO_A  1
$i_0$ = list_getitem($regs_0$, 1)
$a_4$ = int_add($a_3$, $i_0$)
# MOV_A_R 2
list_setitem($regs_0$, 2, $a_4$)
# MOV_R_A 0
$a_5$ = list_getitem($regs_0$, 0)
# JUMP_IF_A 4
$i_1$ = int_is_true($a_5$)
guard_true($i_1$)
jump($a_5$, $regs_0$)
```

Figure 10: Trace when executing the square function of Figure 6, with constant-folding of operations on green variables enabled

4.3 metajit implementation issues

This section describes some of the practical issues of implementing MetaJIT, particularly those of integrating MetaJIT with the language interpreter. These issues can be considered implementation details of MetaJIT and are not inherent properties of the meta-tracing approach.

The first integration problem is how to *not* integrate MetaJIT at all. It is possible to choose when the language interpreter is translated to C whether MetaJIT should be built in or not. If the JIT is not enabled, all the hints that have possibly been put into the interpreter source are just ignored by the translation process.

If MetaJIT is enabled, things are more interesting.[4] A classical tracing JIT will interpret the program it is running until a hot loop is identified, at which point tracing and ultimately machine code generation starts. However, MetaJIT is operating on the language interpreter, which is itself written in RPython. But RPython programs are statically translatable to C anyway. It is clearly desirable to only incur this double-interpretation overhead when that is absolutely necessary, which is during tracing. Otherwise, the meta-tracing approach would be less practical, because before warmup the user program would be extremely slow.

This is achieved by running the language interpreter as a C program, until a hot loop in the user program is found. To identify loops, the C version of the language interpreter is generated in such a way

[4] At the moment the JIT can only be enabled when translating the interpreter to C, but Cuni [Cun10] explored techniques for lifting this restriction.

that at the place that corresponds to the `can_enter_jit` hint profiling is performed using the program counter of the language interpreter. Apart from this bit of profiling, the language interpreter behaves in just the same way as without a JIT.

After a hot user loop has been identified in this way, tracing is started. The tracing interpreter is invoked to start tracing the language interpreter of the running user program. Of course the tracing interpreter cannot trace the execution of the C representation of the language interpreter. Instead it takes the state of the execution of the language interpreter and starts tracing using a bytecode representation of the RPython code of the language interpreter. That means that the language interpreter is embedded in the final executable of the VM in two formats: on the one hand it is there as executable machine code, on the other hand as bytecode for the tracing interpreter. This also means that tracing is slow, because of the double interpretation overhead.

From then on things proceed as described in Section 4.1. The tracing interpreter traces until it has traced a full loop of the user program. After it has done that, it will produce machine code for that loop and this machine code will be immediately executed. The machine code is executed until a guard fails. Then the execution should fall back to normal interpretation by the language interpreter. This falling back is possibly a complex process, since the guard failure can have occurred arbitrarily deep inside an inlined helper function of the language interpreter. That would make it hard to rebuild the state of the language interpreter and let it run from that point as it would involve building a potentially deep C stack.

Instead the falling back is achieved by a special *fallback interpreter* which runs the language interpreter and the user program from the point of the guard failure. The fallback interpreter is essentially a variant of the tracing interpreter that does not keep a trace. The fallback interpreter runs until execution reaches a safe point where it is easy to let the C version of the language interpreter resume its operation. Such a safe point is a place between the executions of single bytecode instructions of the language interpreter. This means that the fallback interpreter executes up to one bytecode operation of the language interpreter and then falls back to the C version of the language interpreter. After this, the whole process of profiling may start again. This complex architecture is part of the price paid for being able to do meta-tracing.

Machine code production is done via a well-defined interface to a machine code backend. This allows easy porting of the tracing JIT to various architectures; at the time when the thesis was written, MetaJIT contains backends for 32-bit and 64-bit Intel-x86 machines as well as for ARM processors [Sch11].

		time (s)	speedup over interpreter
1	Compiled to C, no JIT	2.35 ± 0.00	1.00
2	Normal Trace Compilation	2.65 ± 0.00	0.89
3	Unrolling of Interpreter Loop	0.41 ± 0.13	5.76
4	JIT, Full Optimization	0.13 ± 0.00	18.63
5	Profiling Overhead	5.63 ± 0.01	0.42

Figure 11: Benchmark results of example interpreter computing the square of 100 000 000

4.4 EVALUATION

In this section we evaluate the work done so far by looking at some benchmarks for the example interpreter. The benchmarking method, the involved soft- and hardware, and links to all software repositories are given in Appendix A.

The first round of benchmarks (Figure 11) are timings of the example interpreter given in Figure 5 computing the square of 100 000 000 using the bytecode of Figure 6.[5] The timing results can be seen in Figure 11. The following five configurations were measured:

BENCHMARK 1: The interpreter translated to C without including MetaJIT.

BENCHMARK 2: MetaJIT is enabled, but no interpreter-specific hints are applied. This corresponds to the trace in Figure 7. The threshold when to consider a loop to be hot is 1039 iterations.[6] As expected, this is not faster than the previous number. It is even a bit slower, probably due to the overheads, as well as non-optimal generated machine code.

BENCHMARK 3: MetaJIT is enabled and hints as in Figure 8 are applied. This means that the interpreter loop is unrolled so that it corresponds to the loop in the square function. Constant folding of operations on green variables is disabled, therefore the resulting machine code corresponds to the trace in Figure 9. This alone brings a significant improvement over the previous case and is more than five times faster than interpretation.

BENCHMARK 4: Same as before, but with constant folding enabled. This corresponds to the trace in Figure 10. This speeds up

[5] The result will overflow, but for smaller numbers the running time is not long enough to sensibly measure it.

[6] This is the default threshold that was arrived at by trying various thresholds on PyPy's Python interpreter.

the square function considerably, making it more than 18 times faster than the pure interpreter.

BENCHMARK 5: Same as before, but with the threshold set so high that the tracer is never invoked. In this way the overhead of the profiling is measured. For this interpreter it seems to be rather large, being 2.4 times slower than the interpreter without profiling. This is because the interpreter is small and the opcodes simple. For larger interpreters like PyPy's Python interpreter the overhead will likely be less significant.

The numbers for this tiny interpreter are not too representative. The interpreter is small, the language it runs is simple, deals with only one data type and is statically typed. However, the numbers give an idea of the speedups that can be achieved. A more realistic interpreter will be studied in Chapter 10.

4.5 CONCLUSION

In this chapter we have seen techniques for improving the results when applying a tracing JIT to an interpreter. This is absolutely necessary for reaching meta-tracing, but on its own does not yield too good results when implementing complex languages. However, the first benchmarks presented in this chapter indicate that these techniques work well on small interpreters.

To push the approach further more hints to guide the meta-tracing process are needed (presented in the next chapter), and a novel optimization to deal with the heavy allocation-rate of dynamic languages (presented in Chapter 6).

5 RUN-TIME FEEDBACK

The last chapter explained how to trace an interpreter with a generic tracer running on the level below the interpreter. This chapter presents hints that allow the interpreter author to fine-tune this process. The big speed-ups that JIT compilation can bring come from effectively feeding back and exploiting run-time information into the compilation process. In particular, if there are values which change very slowly, it is possible to compile multiple specialized versions of the same code, one for each actual value. To make use of run-time feedback, the implementation code and data need to be structured so that many such slow-changing values are available.

The most important application of this run-time feedback is the observation of the actual types used in a program. These are not manifest in the source code of the user program due to dynamic typing. While the types could vary during the execution of a program, in practice they rarely do [HH09, CRTR11, RLBV10]. In hand-written JITs this process is manually encoded in the JIT compiler. In the meta-tracing approach, the process is guided by hints that the interpreter author can place in the source code of the interpreter. This works by utilizing the constant folding pass of MetaJIT. The first of these hints has the effect of increasing the amount of constants available in the trace. The second allows the declaration of constant-foldable functions. The combined effect is that a lot of operations in the trace can be constant-folded.

5.1 MOTIVATING EXAMPLE

As the running example of this chapter a very simple and bare-bones object model will be used that just supports classes and instances, without any inheritance or other advanced features. The RPython implementation can be seen in Figure 12. In the model every class contains methods (line 4). Every instance has a class (line 18) and a number of attributes, or fields (line 19). When looking up an attribute of an instance (line 27), the instance's attributes are searched (line 28). If the attribute is not found there, the class' methods are searched (line 30).[1]

In this straightforward implementation the methods and attributes are just stored in dictionaries (hash maps) on the classes and in-

[1] In this example the "methods" are just numbers, just as the attributes. In a more realistic language they would be actual method objects, but the cost of the lookup would stay the same.

```python
class Class(object):
    def __init__(self, name):
        self.name = name
        self.methods = {}

    def instantiate(self):
        return Instance(self)

    def find_method(self, name):
        return self.methods.get(name, None)

    def write_method(self, name, value):
        self.methods[name] = value


class Instance(object):
    def __init__(self, cls):
        self.cls = cls
        self.attributes = {}

    def getfield(self, name):
        return self.attributes.get(name, None)

    def write_attribute(self, name, value):
        self.attributes[name] = value

    def getattr(self, name):
        result = self.getfield(name)
        if result is None:
            result = self.cls.find_method(name)
            if result is None:
                raise AttributeError
        return result
```

Figure 12: Original version of a simple object model

```
# inst1.getattr("a")                                      27
attributes1 = inst1.attributes                            22
result1 = dict.get(attributes1, "a")                      22
guard(result1 is not None)                                29

# inst1.getattr("b")                                      27
attributes2 = inst1.attributes                            22
v1 = dict.get(attributes2, "b", None)                     22
guard(v1 is None)                                         29
cls1 = inst1.cls                                          30
methods1 = cls1.methods                                   10
result2 = dict.get(methods1, "b", None)                   10
guard(result2 is not None)                                31
v2 = result1 + result2                                    -1

# inst1.getattr("c")                                      27
attributes3 = inst1.attributes                            22
v3 = dict.get(attributes3, "c", None)                     22
guard(v3 is None)                                         29
cls2 = inst1.cls                                          30
methods2 = cls2.methods                                   10
result3 = dict.get(methods2, "c", None)                   10
guard(result3 is not None)                                31

v4 = v2 + result3                                         -1
return(v4)                                                -1
```

Figure 13: Trace through the object model

stances, respectively. While this object model is very simple, it already contains most hard parts of Python's object model. Both instances and classes can have arbitrary fields, and they are changeable at any time. Moreover, instances can change their class after they have been created.

When using this object model in an interpreter, a large amount of time will be spent doing lookups in these dictionaries. Assuming we have created a class and an instance as follows:

```
cls = Class("A")
cls.write_method("b", 9)
cls.write_method("c", 21)
inst = cls.instantiate()
inst.write_attribute("a", 11)
```

Let us look at what happens when the following code that sums three attributes, is traced:

```
inst.getattr("a") + inst.getattr("b") + inst.getattr("c")
```

The trace would look like in Figure 13. In this example, the attribute `a` is found on the instance, but the attributes `b` and `c` are found on the class. The line numbers in the trace correspond to the line numbers in Figure 12 where the traced operations come from. The trace is in SSA form. Note how all the guards in trace correspond to an `if` condition in the original code. The trace contains five calls to `dict.get`, which is a slow operation. To make the language efficient using a tracing JIT, these dictionary lookups need to be removed in some way. How to achieve this will be the topic of Section 5.3. The key observation of doing so will be that the classes' methods as well as the layout of the instances change rarely.

5.2 HINTS FOR CONTROLLING OPTIMIZATION

This section describes two hints that allow the interpreter author to increase the optimization opportunities for constant folding. If applied correctly these techniques can give big speedups by pre-computing parts of what happens at run-time. However, if applied incorrectly they might lead to code bloat, thus actually making the resulting program slower. Note that these hints never have to be put into the user program, only into the interpreter by the interpreter author.

For constant folding to work, two conditions need to be met: the arguments of an operation actually need to all be constant, i.e. statically known by the optimizer and the operation needs to be *constant-foldable*, i.e. always yield the same result given the same arguments. There is one kind of hint for both of these conditions.

5.2.1 *Where do all the constants come from?*

It is worth clarifying what a "constant" is in this context. A variable of the trace is said to be constant if its value is statically known by the optimizer. The simplest example of constants are literal values in the source code, such as `1`. However, the optimizer can statically know the value of a variable even if it is not a constant in the original source code. For example, consider the following fragment of RPython code. If the following fragment is traced with `x` being `4`:

```
if x == 4:
    y = y + x
```

Then this trace is produced:

```
guard(x1 == 4)
y2 = y1 + x1
```

A guard is a run-time check. The above trace will run to completion only when x_1 `== 4`. If the check fails, execution of the trace is stopped and the interpreter continues to run. Therefore, the value of x_1 is statically known to be `4` after the guard.

In some interpreters there are places in which it would be useful to turn an arbitrary variable into a constant value because that would open a lot of optimization opportunities. This is the case if there is a lot of computation depending on the value of one variable. This is made possible by a process called *promotion*. Promotion is essentially a tool for trace specialization and very similar to an old idea in partial evaluation (it is called "The Trick" [JGS93] there). The technique is substantially more powerful in a JIT compiler than in the static setting of classic partial evaluation (the connection between meta-tracing and partial evaluation will be explored in detail in Chapter 12).

The approach of promotion is to artificially introduce an equality guard for a variable like the one above but without having an `if` statement in the code. Additionally, the run-time value of that variable during tracing is chosen as the concrete value that the variable is compared against by the guard. This makes it possible to use promotion for variables where the likely run-time values are not knowable in advance.

To make this more concrete, let's assume that a call to the following function (written in RPython) is traced:

```
def f1(x, y):
    z = x * 2 + 1
    return z + y
```

Then this trace is produced:

```
v1 = x1 * 2
z1 = v1 + 1
v2 = z1 + y1
return(v2)
```

Observe how the first two operations could be constant-folded if the value of x_1 were known. Let us further assume that the value of `x` in the RPython code can vary, but does so rarely, i.e. only takes a few different values at run-time. If this is the case, a hint to promote `x` can be added, like this:

```
def f1(x, y):
    promote(x)
    z = x * 2 + 1
    return z + y
```

The hint indicates that `x` is likely a run-time constant and that Meta-JIT should try to perform run-time specialization on it in the code that follows. When just running the code, the `promote` function has no effect. When tracing, some extra work is done. Let us assume that this changed function is traced with the arguments `4` and `8`. The trace will be the same, except for one operation at the beginning:

```
guard(x1 == 4)
v1 = x1 * 2
z1 = v1 + 1
v2 = z1 + y1
return(v2)
```

The promotion is turned into a `guard` operation in the trace. The guard captures the run-time value of `x` as it was during tracing, which can then be exploited by the optimizer. The introduced guard specializes the trace, because it only works if the value of x_1 is `4`. For the optimizer, this guard is not different from the one produced by the `if` statement in the first example. After the guard, it can be assumed that x_1 is equal to `4`, meaning that the optimizer will turn this trace into:

```
guard(x1 == 4)
v2 = 9 + y1
return(v2)
```

Notice how the first two arithmetic operations were constant folded. The assumption is that the guard is executed quicker than the multiplication and the addition that was now optimized away.

If this trace is executed with values of x_1 other than `4`, the guard will fail, and execution will continue in the interpreter. If the guard fails often enough, a new trace will be started from the guard. This other trace will capture a different value of x_1. If it is for example `2`, then the optimized trace looks like this:

```
guard(x1 == 2)
v2 = 5 + y1
return(v2)
```

This new trace will be attached to the guard instruction of the first trace. If x_1 takes on even more values, a new trace will eventually be made for all of them, linking them into a chain. This is clearly not desirable, so only variables that do not vary very much should be promoted. However, adding a promotion hint will never produce wrong results. It might just lead to too much machine code being generated.

Promoting integers, as in the examples above, is not used that often. However, the internals of dynamic language interpreters often have values that are variable but vary little in the context of parts of a user program. An example would be the types of variables in a user function, which rarely change in a dynamic language in practice (even though they could) [HH09, CRTR11, RLBV10]. In the interpreter, these user-level types are values. Thus promoting them will lead to type-specialization on the level of the user program. Section 5.3 will present a complete example of how this works.

5.2.2 *Declaring new foldable operations*

The previous section presented a way to turn arbitrary variables into constants. Foldable operations that operate only on constant arguments will be optimized away by the constant-folding optimization. This works well for constant folding of primitive types, e.g. integers. Unfortunately, in the context of an interpreter for a dynamic language, most operations manipulate objects, not primitive types. The operations on objects are often not foldable and might even have side-effects. If one reads a field out of a constant reference to an object this cannot necessarily be folded away because the optimizer has to assume that that the object could be mutated. Therefore, another hint is needed.

This hint can be used to mark functions as *trace-elidable*. A function is termed trace-elidable, if, during the execution of the program, successive calls to the function with identical arguments always return the same result. In addition the function needs to have no side effects or idempotent side effects.[2] From this definition follows that a call to a trace-elidable function with constant arguments in a trace can be replaced with the result of the call seen during tracing.

As an example, take the following class:

```
class A(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def f(self, val):
        self.y = self.c() + val

    def c(self):
        return self.x * 2 + 1
```

Tracing the call `a.f(10)` of some instance of `A` yields the this trace (note how the call to `c` is inlined):

```
# inside A.c
    $x_1$ = $a_1$.x
    $v_1$ = $x_1$ * 2
    $v_2$ = $v_1$ + 1
$v_3$ = $v_2$ + $val_1$
$a_1$.y = $v_3$
```

In this case, adding a promote of `self` in the `f` method to get rid of the computation of the first few operations does not help. Even if a_1 is a constant reference to an object, reading the `x` field does not necessarily always yield the same value. To solve this problem, there is another annotation, which lets the interpreter author communicate invariants to the optimizer. In this case, she could decide that the `x` field of instances of `A` is immutable, and therefore `c` is a trace-elidable function. To communicate this, there is an `@elidable` decorator. If the code in `c` should be constant-folded away, the class would have to be changed as follows:

```
class A(object):
    ...

    def f(self, val):
        promote(self)
        self.y = self.c() + val

    @elidable
    def c(self):
        return self.x * 2 + 1
```

[2] This property is less strict than that of a pure function, because it is only about actual calls during execution. All pure functions are trace-elidable though.

Leading to this result:

```
guard($a_1$ == 0xb73984a8)
$v_1$ = A.c($a_1$)
$v_2$ = $v_1$ + $val_1$
$a_1$.y = $v_2$
```

Here, `0xb73984a8` is the address of the instance of `A` that was used during tracing. When a call to a trace-elidable function is seen during tracing, that function is not inlined, so that the optimizer sees it as a call, not as its constituent operations. Since the `A.c` method is marked as trace-elidable, and its argument is a constant reference, the call will be removed by the optimizer. The final trace looks like this (assuming that the value of the field `x` is `4`):

```
guard($a_1$ == 0xb73984a8)
$v_2$ = 9 + $val_1$
$a_1$.y = $v_2$
```

On the one hand, the `@elidable` annotation is very powerful. It can be used to constant-fold arbitrary parts of the computation in the interpreter. However, the annotation also gives the interpreter author ample opportunity to introduce bugs. If a function is annotated to be trace-elidable, but is not really, the optimizer can produce subtly wrong code. Therefore, a lot of care has to be taken when using this annotation.[3] RPython contains a preliminary debugging mode which checks whether the annotation is applied incorrectly to mitigate this problem, which is so far extremely slow but can be used for finding bad elidable annotations.

5.3 IMPROVING THE EXAMPLE OBJECT MODEL

In this section the simple object model from Section 5.1 will be made efficient using the hints described in the previous section. The object model of Figure 12 is typical for many dynamic languages (such as Python, Ruby, and JavaScript) as it relies heavily on hash-maps to implement its objects.

5.3.1 *Faster instance attributes with maps*

The first step in making `getattr` faster in the object model is to optimize away the dictionary lookups on the instances. The hints of the previous section do not seem to be applicable to how the object model is currently implemented. There is no trace-elidable function to be seen, and the instance is not a candidate for promotion, because there tend to be many instances.

[3] The most common use case of the `@elidable` annotation is indeed to declare the immutability of fields. Because it is so common, RPython has special syntactic sugar for it.

```python
class Map(object):
    def __init__(self):
        self.indexes = {}
        self.other_maps = {}

    @elidable
    def getindex(self, name):
        return self.indexes.get(name, -1)

    @elidable
    def add_attribute(self, name):
        if name not in self.other_maps:
            newmap = Map()
            newmap.indexes.update(self.indexes)
            newmap.indexes[name] = len(self.indexes)
            self.other_maps[name] = newmap
        return self.other_maps[name]

EMPTY_MAP = Map()

class Instance(object):
    def __init__(self, cls):
        self.cls = cls
        self.map = EMPTY_MAP
        self.storage = []

    def getfield(self, name):
        map = self.map
        promote(map)
        index = map.getindex(name)
        if index != -1:
            return self.storage[index]
        return None

    def write_attribute(self, name, value):
        map = self.map
        promote(map)
        index = map.getindex(name)
        if index != -1:
            self.storage[index] = value
            return
        self.map = map.add_attribute(name)
        self.storage.append(value)

    def getattr(self, name):
        ... # as before
```

Figure 14: Simple object model with maps

```
# inst1.getattr("a")
map1 = inst1.map
guard(map1 == 0xb74af4a8)
index1 = Map.getindex(map1, "a")
guard(index1 != -1)
storage1 = inst1.storage
result1 = storage1[index1]

# inst1.getattr("b")
map2 = inst1.map
guard(map2 == 0xb74af4a8)
index2 = Map.getindex(map2, "b")
guard(index2 == -1)
cls1 = inst1.cls
methods1 = cls1.methods
result2 = dict.get(methods1, "b", None)
guard(result2 is not None)
v2 = result1 + result2

# inst1.getattr("c")
map3 = inst1.map
guard(map3 == 0xb74af4a8)
index3 = Map.getindex(map3, "c")
guard(index3 == -1)
cls2 = inst1.cls
methods2 = cls2.methods
result3 = dict.get(methods2, "c", None)
guard(result3 is not None)

v4 = v2 + result3
return(v4)
```

Figure 15: Unoptimized trace after the introduction of maps

This is a common problem when trying to apply hints. Often, the interpreter needs a small rewrite to expose the trace-elidable functions and nearly-constant objects that are implicitly there. In the case of instance fields this rewrite is not entirely obvious. The basic idea is as follows. In theory instances can have arbitrary fields. In practice however many instances share their layout (i.e. their set of keys) with many other instances.

Therefore it makes sense to factor the layout information out of the instance implementation into a shared object, called the *map*. Maps are a well-known technique to efficiently implement instances. They stem from the Self project [CUE89] and are also used by many JavaScript implementations, such as V8, where they are called hidden classes. The rewritten `Instance` class using maps can be seen in Figure 14.

In this implementation instances no longer use dictionaries to store their fields. Instead, they have a reference to a map, which maps field names to indexes into a storage list. The storage list contains the actual field values. Maps are shared between different instances, therefore they have to be immutable, which means that their `getindex` method is a trace-elidable function. When a new attribute is added to an instance, a new map needs to be chosen, which is done with the `add_attribute` method on the previous map. This function caches all new instances of `Map` that it creates, to make sure that objects with the same layout have the same map. This caching makes its side effects idempotent and the function trace-elidable. Now that maps have been introduced, it is safe to promote the map everywhere, because it is safe to assume that the number of different instance layouts is small.

With this adapted instance implementation, the trace of Section 5.1 changes to that of Figure 15. There `0xb74af4a8` is the memory address of the `Map` instance that has been promoted. Operations that can be optimized away are grayed out, their results will be replaced with fixed values by the constant folding.

The calls to `Map.getindex` can be optimized away, because they are calls to a trace-elidable function and they have constant arguments. That means that $index_{1/2/3}$ are constant and the guards on them can be removed. All but the first guard on the map will be optimized away too, because the map cannot have changed in between. This trace is already much better than the original one. Now only two out of five dictionary lookups are left.

The technique to make instance lookups faster is applicable in more general cases. A more abstract view of maps is that of splitting a data-structure into an immutable part (the map) and a part that changes (the storage list). All the computation on the immutable part is trace-elidable so that only the manipulation of the quick-changing part remains in the trace after optimization.

5.3.2 *Versioning of classes*

In the previous section it was assumed that the total number of different instance layouts is small compared to the number of instances. For classes an even stronger assumption can be made. For classes it is very rare that they change at all [CRTR11], or only after initialization [HH09]. This is not always reasonable (sometimes classes contain counters or similar things) but for this simple example it is good enough.[4]

It would be best if the `Class.find_method` method were trace-elidable. But it cannot be, because it is always possible to change the methods of the class. Every time the methods are changed, `find_method` can potentially return a new value.

Therefore, every class gets a version object, which is changed every time the class's methods are changed. This means that the result of calls to `methods.get()` for a given `(name, version)` pair will always be

[4] There is a more complex variant of the presented technique that can accommodate quick-changing class fields a lot better. It is presented in Chapter 10.

```
class VersionTag(object):
    pass

class Class(object):
    def __init__(self, name):
        self.name = name
        self.methods = {}
        self.version = VersionTag()

    def find_method(self, name):
        promote(self)
        version = self.version
        promote(version)
        return self._find_method(name, version)

    @elidable
    def _find_method(self, name, version):
        assert version is self.version
        return self.methods.get(name, None)

    def write_method(self, name, value):
        self.methods[name] = value
        self.version = VersionTag()
```

Figure 16: Versioning of classes

the same, i.e. it is a trace-elidable operation. To make MetaJIT detect this case, this is factored out into the helper method `_find_method`, which is marked as `@elidable`. The refactored `Class` can be seen in Figure 16.

The two promotions in `find_method` encode two different assumptions: The first promote encodes the assumption that in the context of a specific piece of code the receiver's class will likely not vary much. The second promote encodes the assumption that a class is not changed often.

What is interesting here is that `_find_method` takes the `version` argument but it does not use it at all. Its only purpose is to make the call trace-elidable, because when the version object changes, the result of the call might be different from the previous one. The function is not pure, however.

The trace with this new class implementation can be seen in Figure 17. The calls to `Class._find_method` can now be optimized away, also the promotion of the class and the version, except for the first one. The final optimized trace can be seen in Figure 18.

The index `0` that is used to read out of the `storage` list is the result of the constant-folded `getindex` call. The constants `41` and `17` are the results of the folding of the `_find_method` calls. This final trace is now very efficient. It no longer performs any dictionary lookups. Instead it contains several guards. The first guard checks that the map is still the same. This guard will fail if the same code is executed with an instance that has another layout. The second guard checks that the

```
# inst1.getattr("a")
map1 = inst1.map
guard(map1 == 0xb74af4a8)
index1 = Map.getindex(map1, "a")
guard(index1 != -1)
storage1 = inst1.storage
result1 = storage1[index1]

# inst1.getattr("b")
map2 = inst1.map
guard(map2 == 0xb74af4a8)
index2 = Map.getindex(map2, "b")
guard(index2 == -1)
cls1 = inst1.cls
guard(cls1 == 0xb7aaaaf8)
version1 = cls1.version
guard(version1 == 0xb7bbbb18)
result2 = Class._find_method(cls1, "b", version1)
guard(result2 is not None)
v2 = result1 + result2

# inst1.getattr("c")
map3 = inst1.map
guard(map3 == 0xb74af4a8)
index3 = Map.getindex(map3, "c")
guard(index3 == -1)
cls2 = inst1.cls
guard(cls2 == 0xb7aaaaf8)
version2 = cls2.version
guard(version2 == 0xb7bbbb18)
result3 = Class._find_method(cls2, "c", version2)
guard(result3 is not None)

v4 = v2 + result3
return(v4)
```

Figure 17: Unoptimized trace after introduction of versioned classes

```
# inst1.getattr("a")
map1 = inst1.map
guard(map1 == 0xb74af4a8)
storage1 = inst1.storage
result1 = storage1[0]

# inst1.getattr("b")
cls1 = inst1.cls
guard(cls1 == 0xb7aaaaf8)
version1 = cls1.version
guard(version1 == 0xb7bbbb18)
v2 = result1 + 41

# inst1.getattr("c")
v4 = v2 + 17
return(v4)
```

Figure 18: Optimized trace after introduction of versioned classes

class of inst is still the same. It will fail if the trace is executed with an instance of another class. The third guard checks that the class did not change since the trace was produced. It will fail if somebody calls the write_method method on the class.

The first and the second guard could be merged by storing the class of the instance in the map as well. How this works will be discussed in Chapter 10, which also evaluates the speed benefits of using maps and version tags to implement classes and instances of Python.

5.4 CONCLUSION

In this chapter two hints, promote and elidable were presented that can be used in the source code of an interpreter written with RPython. They give control over run-time feedback and optimization to the language implementor. They are expressive enough for building well-known virtual machine optimization techniques, such as maps and inline caches. They are flexible enough to express a wide variety of language semantics efficiently, which will be studied and explained in more detail in the case studies in Chapter 10 and Chapter 11.

The hints described in this chapter are good for operations on user-defined classes and objects. The next chapter will show how a generic optimization can be used to optimize away the overhead of implementing primitive objects, such as integers and floats in dynamic languages.

6 ALLOCATION REMOVAL

As seen in the previous two chapters, the use of a meta-tracing JIT can remove the overhead of bytecode dispatch, of the interpreter data structures and of operations in the object model. In this chapter a further optimization for tracing JITs is presented that removes some of the further overhead more closely associated with dynamic languages, such as boxing overhead and type dispatching. As opposed to the previous two chapters, this optimization is completely language-independent and does not need any hints to function. Section 6.1 shows an example object model that showcases many of the problems related to boxing primitive types. Section 6.2 analyzes the problem to be solved more closely.

The core of this trace optimization technique can be viewed as partial evaluation [Fut99, JGS93]. Partial evaluation (PE), also called specialization, is a program manipulation technique. PE takes an input program and transforms it into a (hopefully) simpler and faster output program. It does this by assuming that some variables in the input program are constants. These are called *static* variables. All operations that act only on static variables can be folded away. All other operations need to remain in the output program (called residual program). Thus the partial evaluator proceeds much like an interpreter, just that it cannot actually execute all of the operations. Also, its output is not just a value, but a list of remaining operations that could not be optimized away.

The optimization described in the chapter performs a form of escape analysis [Bla03] and scalar replacement [KM05] on the traces. This technique is informally described in Section 6.3; a more formal description is given in Section 6.4, which also shows the relationship between the optimization and partial evaluation. The introduced techniques are evaluated in Section 6.6 using PyPy's Python interpreter.

The technique described in this chapter is the workhorse of the optimizer chain of MetaJIT. It is also representative of how optimizations that are usually too expensive to do at run-time can become tractable by applying them to traces.

6.1 RUNNING EXAMPLE

In this chapter, a tiny interpreter for a dynamic language is used with a very simple object model, that just supports an integer and a float type. The objects support only two operations, `add`, which adds two objects (promoting ints to floats in a mixed addition) and `is_positive`, which returns whether the number is greater than zero. The implementation of `add` uses double-dispatching. The classes can be seen in Figure 19 (written in RPython).

Using these classes to implement arithmetic shows the basic problem of a dynamic language implementation. The language supports both an integer and a float type. However, RPython does not allow the mixing of these two primitive types. Therefore the interpreter needs to be represent the numbers as instances of either `BoxedInteger` or `BoxedFloat`, two classes that share a common base class. Now numbers within the tiny dynamic languages consume space on the heap. Every arithmetic operation needs to examine the types of the arguments and also allocate a new instance of one of these classes for the result. Performing many arithmetic operations thus produces lots of garbage quickly, putting pressure on the garbage collector. Using double dispatching to implement the numeric tower needs two RPython method calls per arithmetic operation, which is costly due to the method dispatch.

Let us now consider a simple "interpreter" function `f` that uses the object model (see the bottom of Figure 19). The loop in `f` iterates `y` times, and computes something in the process. Simply running this function is slow, because there are lots of virtual method calls inside the loop, one for each `is_positive` and even two for each call to `add`. These method calls need to check the type of the involved objects repeatedly and redundantly. In addition, a lot of objects are created when executing that loop, many of these objects are short-lived. The actual computation that is performed by `f` is simply a sequence of float or integer additions.

If the function is executed using MetaJIT, with `y` being a `BoxedInteger`, the produced trace looks like the one of Figure 20. The trace corresponds to one iteration of the while-loop in `f`.

The operations in the trace are indented corresponding to the stack level of the function that contains the traced operation. The trace is in single-assignment form, meaning that each variable is assigned a value exactly once. The arguments p_0 and p_1 of the loop correspond to the live variables `y` and `res` in the while-loop of the original function.

The operations in the trace correspond to the operations in the RPython program in Figure 19:

- `new` creates a new object.
- `get` reads a field of an object.
- `set` writes to a field of an object.
- `guard_class` is a precise type check. It always precedes an (inlined) method call and is followed by the trace of the called method.

```
class Base(object):
   pass

class BoxedInteger(Base):
   def __init__(self, intval):
      self.intval = intval

   def add(self, other):
      return other.add__int(self.intval)

   def add__int(self, intother):
      return BoxedInteger(intother + self.intval)

   def add__float(self, floatother):
      floatvalue = floatother + float(self.intval)
      return BoxedFloat(floatvalue)

   def is_positive(self):
      return self.intval > 0

class BoxedFloat(Base):
   def __init__(self, floatval):
      self.floatval = floatval

   def add(self, other):
      return other.add__float(self.floatval)

   def add__int(self, intother):
      floatvalue = float(intother) + self.floatval
      return BoxedFloat(floatvalue)

   def add__float(self, floatother):
      return BoxedFloat(floatother + self.floatval)

   def is_positive(self):
      return self.floatval > 0.0


def f(y):
   res = BoxedInteger(0)
   while y.is_positive():
      res = res.add(y).add(BoxedInteger(-100))
      y = y.add(BoxedInteger(-1))
   return res
```

Figure 19: An “interpreter” for a tiny dynamic language written in RPython

```
[$p_0$, $p_1$]
# inside f: res.add(y)
guard_class($p_1$, BoxedInteger)
    # inside BoxedInteger.add
    $i_2$ = get($p_1$, intval)
    guard_class($p_0$, BoxedInteger)
        # inside BoxedInteger.add__int
        $i_3$ = get($p_0$, intval)
        $i_4$ = int_add($i_2$, $i_3$)
        $p_5$ = new(BoxedInteger)
            # inside BoxedInteger.__init__
            set($p_5$, intval, $i_4$)

# inside f: BoxedInteger(-100)
$p_6$ = new(BoxedInteger)
    # inside BoxedInteger.__init__
    set($p_6$, intval, -100)

# inside f: .add(BoxedInteger(-100))
guard_class($p_5$, BoxedInteger)
    # inside BoxedInteger.add
    $i_7$ = get($p_5$, intval)
    guard_class($p_6$, BoxedInteger)
        # inside BoxedInteger.add__int
        $i_8$ = get($p_6$, intval)
        $i_9$ = int_add($i_7$, $i_8$)
        $p_{10}$ = new(BoxedInteger)
            # inside BoxedInteger.__init__
            set($p_{10}$, intval, $i_9$)

# inside f: BoxedInteger(-1)
$p_{11}$ = new(BoxedInteger)
    # inside BoxedInteger.__init__
    set($p_{11}$, intval, -1)

# inside f: y.add(BoxedInteger(-1))
guard_class($p_0$, BoxedInteger)
    # inside BoxedInteger.add
    $i_{12}$ = get($p_0$, intval)
    guard_class($p_{11}$, BoxedInteger)
        # inside BoxedInteger.add__int
        $i_{13}$ = get($p_{11}$, intval)
        $i_{14}$ = int_add($i_{12}$, $i_{13}$)
        $p_{15}$ = new(BoxedInteger)
            # inside BoxedInteger.__init__
            set($p_{15}$, intval, $i_{14}$)

# inside f: y.is_positive()
guard_class($p_{15}$, BoxedInteger)
    # inside BoxedInteger.is_positive
    $i_{16}$ = get($p_{15}$, intval)
    $i_{17}$ = int_gt($i_{16}$, 0)
# inside f
guard_true($i_{17}$)
jump($p_{15}$, $p_{10}$)
```

Figure 20: An unoptimized trace of the example interpreter

- int_add and int_gt are integer addition and comparison ("greater than"), respectively.

- guard_true checks that a boolean is true.

Method calls in the trace are preceded by a guard_class operation, to check that the class of the receiver is the same as the one that was observed during tracing.[1] These guards make the trace specific to the situation where y is a BoxedInteger. When the trace is turned into machine code and afterwards executed with BoxedFloat, the first guard_class instruction will fail and execution will continue using the interpreter.

The trace shows the inefficiencies of f clearly, if one looks at the number of new, set/get and guard_class operations. The number of guard_class operation is particularly problematic, not only because of the time it takes to run them. All guards also have additional information attached that makes it possible to return to the interpreter, should the guard fail. This means that too many guard operations also consume a lot of memory [SB12].

In the rest of the chapter we will see how this trace can be optimized using partial evaluation.

6.2 OBJECT LIFETIMES IN A TRACING JIT

To understand the problems that this chapter is trying to solve in more detail, we first need to understand the cases of object lifetimes that can occur in a tracing JIT compiler.

Figure 21 shows a trace before optimization, together with the lifetime of various kinds of objects created in the trace. It is executed from top to bottom. At the bottom, a jump is used to execute the same loop another time (for clarity, the figure shows two iterations of the loop). The loop is executed until one of the guards in the trace fails, and the execution is aborted and interpretation resumes.

Some of the operations within this trace are new operations, each of which creates a new instance of some class. These instances are used for some time within the trace, by reading and writing their fields as well as by calling methods on them, which are inlined into the trace. Some of these instances *escape*, which means that they are stored in some globally accessible place or are passed into a non-inlined function via a residual call. This means that they cannot be tracked precisely any more.

Together with the new operations, the figure shows the lifetimes of the created objects. The objects that are created within a trace using new fall into one of several categories:

[1] guard_class performs a precise class check, not checking for subclasses.

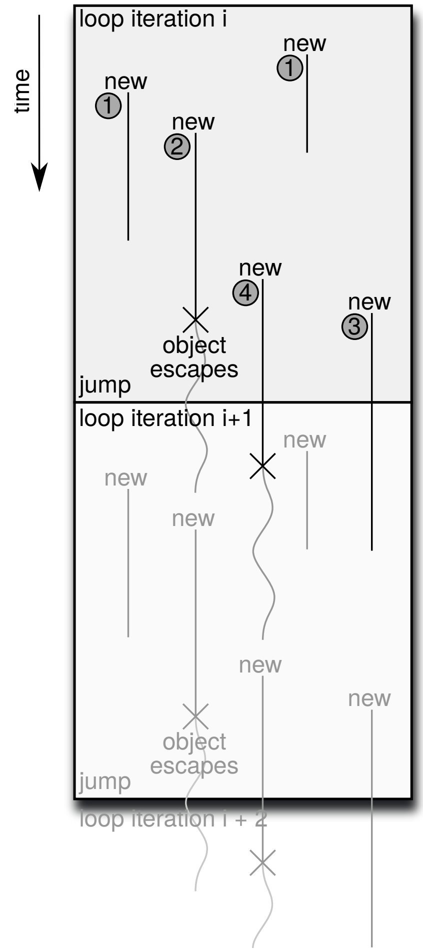


Figure 21: Object lifetimes in a trace

1. Objects that live for some time, and are then just not used any more afterwards.

2. Objects that live for some time and then escape.

3. Objects that live for some time, survive across the jump to the beginning of the loop, and are then not used any more.

4. Objects that live for some time, survive across the jump, and then escape. Objects that live across several jumps and then either escape or stop being used are also in this category.

The objects that are allocated in the example trace in Figure 20 fall into categories 1 and 3. Objects stored in p_5, p_6, p_{11} are in category 1, objects in p_{10}, p_{15} are in category 3.

The creation of objects in category 1 is removed by the optimization described in Sections 6.3 and 6.4. Objects in the other categories are partially optimized by this approach as well.[2]

6.3 ALLOCATION REMOVAL IN TRACES

The main insight to improve the code shown in Section 6.1 is that objects in category 1 do not survive very long – they are used only inside the loop and there is no other outside reference to them. The optimizer identifies objects in category 1 and removes the allocation of these objects, and all operations manipulating them.

This is a process that is usually called *escape analysis* [GP90]. In this chapter escape analysis will be done using partial evaluation. The use of partial evaluation is a bit peculiar in that it receives no static input arguments for the trace, but it is only used to optimize operations within the trace. This section will give an informal account of this process by examining the example trace in Figure 20. The final trace after optimization can be seen in Figure 22 (the line numbers are the lines of the unoptimized trace where the operation originates).

To optimize the trace, it is traversed from beginning to end and an output trace is produced. Every operation in the input trace is either removed or copied into the output trace. Sometimes new operations need to be produced as well. The optimizer can only remove operations that manipulate objects that have been allocated within the trace, while all other operations are copied to the output trace unchanged.

Looking at the example trace of Figure 20, the operations in lines 3–9 are manipulating objects which existed before the trace and that are passed in as arguments: therefore the optimizer just copies them into the output trace.

The following operations (lines 10–17) are more interesting:

[2] We also started to work on optimizing objects in category 3 [ABF12].

```
p5 = new(BoxedInteger)                                  10
set(p5, intval, i4)                                     12
p6 = new(BoxedInteger)                                  15
set(p6, intval, -100)                                   17
```

When the optimizer encounters any new, it removes it optimistically, and assumes that the object is in category 1.[3] If later the optimizer finds that the object escapes, it will be allocated at that point. The optimizer needs to keep track of the state of the object that the operation would have created. This is done with the help of a *static object*.[4] The static object describes the shape of the object that would have been allocated, meaning the type of the object and where the values that would be stored in the fields of the allocated object come from.

In the snippet above, the two new operations are removed and two static objects are constructed. The set operations manipulate static objects, therefore they can be removed as well; their effect is remembered in the static objects.

After the operations the static object associated with p_5 would store the knowledge that it is a BoxedInteger whose intval field contains i_4; the one associated with p_6 would store that it is a BoxedInteger whose intval field contains the constant -100.

The subsequent operations (line 20–26) in Figure 20, which use p_5 and p_6, can then be optimized using that knowledge:

```
guard_class(p5, BoxedInteger)                           20
i7 = get(p5, intval)                                    22
guard_class(p6, BoxedInteger)                           23
i8 = get(p6, intval)                                    25
i9 = int_add(i7, i8)                                    26
```

The guard_class operations can be removed, since their arguments are static objects with the matching type BoxedInteger. The get operations can be removed as well, because each of them reads a field out of a static object. The results of the get operation are replaced with what the static object stores in these fields: all the occurrences of i_7 and i_8 in the trace are just replaced by i_4 and -100. The only operation copied into the optimized trace is the addition:

```
i9 = int_add(i4, -100)                                  26
```

The rest of the trace from Figure 20 is optimized in a similar vein. The operations in lines 27–34 produce two more static objects and are removed. Those in lines 37–39 are just copied into the output trace because they manipulate objects that are allocated before the trace. Lines 40–42 are removed because they operate on a static object. Line 43 is copied into the output trace. Lines 44–46 produce a new

[3] Note that this is works for arbitrary classes and not just for simple boxes like the class used as the running example.

[4] Here "static" is meant in the sense of partial evaluation, i.e., known at partial evaluation time, not in the sense of "static allocation" or "static method".

static object and are removed, lines 49–51 manipulate that static object and are removed as well. Lines 52–54 are copied into the output trace.

The last operation (line 55) is an interesting case. It is the jump operation that passes control back to the beginning of the trace. The two arguments to this operation at this point are static objects. However, because they are passed into the next iteration of the loop they live longer than the trace and therefore cannot remain static. They need to be turned into dynamic (run-time) objects before the actual jump operation. This process of turning a static object into a dynamic one is called *lifting*.

Lifting a static object puts new and set operations into the output trace. Those operations produce an object at run-time that has the shape described by the static object. This process is a bit delicate, because the static objects could form an arbitrary graph structure. In our example it is simple, though:

```
$p_{15}$ = new(BoxedInteger)                                   44
set($p_{15}$, intval, $i_{14}$)                                   46
$p_{10}$ = new(BoxedInteger)                                   27
set($p_{10}$, intval, $i_9$)                                    29
jump($p_{15}$, $p_{10}$)                                          55
```

Observe how the operations for creating these two instances have been moved to a later point in the trace. This is worthwhile even though the objects have to be allocated in the end because some get operations and guard_class operations on the lifted static objects could be removed.

More generally, lifting needs to occur if a static object is used in any operation apart from get, set, and guard. It also needs to occur if set is used to store a static object into a non-static one.

The final optimized trace of the example can be seen in Figure 22. The optimized trace contains only two allocations, instead of the original five, and only three guard_class operations, compared to the original seven.

The BoxedInteger instances in the example are only ever written to once, after creation. While this property is very common, it is not a fundamental limitation to the approach. As long as an object is static even repeated writes to it will be optimized away and the static object updated.

6.4 FORMAL DESCRIPTION OF THE ALGORITHM

In this section a formal description of the semantics of the traces and of the optimizer is given. The optimization is characterized as partial evaluation. The focus here is on the operations for manipulating heap allocated objects, as those are the only ones that are actually optimized. Only objects with two fields L and R are considered in this section, generalizing to arbitrary many fields is straightforward.

```
[p0, p1]                                1
guard_class(p1, BoxedInteger)           3
i2 = get(p1, intval)                    5
guard_class(p0, BoxedInteger)           6
i3 = get(p0, intval)                    8
i4 = int_add(i2, i3)                    9
i9 = int_add(i4, -100)                  26

guard_class(p0, BoxedInteger)           37
i12 = get(p0, intval)                   39
i14 = int_add(i12, -1)                  43

i17 = int_gt(i14, 0)                    52
guard_true(i17)                         54

p15 = new(BoxedInteger)                 44
set(p15, intval, i14)                   46
p10 = new(BoxedInteger)                 27
set(p10, intval, i9)                    29

jump(p15, p10)                          55
```

Figure 22: Resulting trace after allocation removal

Traces are lists of operations. The operations considered here are `new`, `get`, `set` and `guard_class`.

The values of all variables are locations (pointers). Locations are mapped to objects, which are represented by triples (T, l_1, l_2) of a type T, and two locations that represent the fields of the object. When a new object is created, the fields are initialized to null, but we require that they are initialized to a real location before being read, otherwise the trace is malformed (this condition is guaranteed by how the traces are generated by the RPython toolchain).

Some abbreviations are used here when dealing with object triples. To read the type of an object, $\text{type}((T, l_1, l_2)) = T$ is used. Reading a field F from an object is written $(T, l_1, l_2)_F$ which either is l_1 if $F = L$ or l_2 if $F = R$. To set field F to a new location l, the notation $(T, l_1, l_2)!_F l$ is used, which yields a new triple (T, l, l_2) if $F = L$ or a new triple (T, l_1, l) if $F = R$.

Figure 23 shows the (small-step) operational semantics for traces. The interpreter formalized there executes one operation at a time. Its state is represented by an environment E and a heap H, which may be changed by the execution of an operation. The environment is a partial function from variables to locations and the heap is a partial function from locations to objects. Note that a variable can never be null in the environment, otherwise the trace would have been malformed. The environment could not directly map variables to objects, because several variables can point to the *same* object, due to aliasing.

new $$\frac{l \text{ fresh}}{v = \texttt{new}(T), E, H \overset{\text{run}}{\Longrightarrow} E\left[v \mapsto l\right], H\left[l \mapsto (T, \text{null}, \text{null})\right]}$$

get $$\frac{}{u = \texttt{get}(v, F), E, H \overset{\text{run}}{\Longrightarrow} E\left[u \mapsto H\left(E\left(v\right)\right)_F\right], H}$$

set $$\frac{}{\texttt{set}\left(v, F, u\right), E, H \overset{\text{run}}{\Longrightarrow} E, H\left[E\left(v\right) \mapsto \left(H\left(E\left(v\right)\right)!_F E(u)\right)\right]}$$

guard $$\frac{\text{type}(H(E(v)) = T}{\texttt{guard\_class}(v, T), E, H \overset{\text{run}}{\Longrightarrow} E, H}$$

$$\frac{\text{type}(H(E(v)) \neq T}{\texttt{guard\_class}(v, T), E, H \overset{\text{run}}{\Longrightarrow} \bot, \bot}$$

Object Domains:

$u, v \in V$	variables in trace
$T \in \mathfrak{T}$	run − time types
$F \in \{L, R\}$	fields of objects
$l \in L$	locations on heap

Semantic Values:

$E \in V \rightharpoonup L$	Environment
$H \in L \rightharpoonup \mathfrak{T} \times (L \cup \{\text{null}\}) \times (L \cup \{\text{null}\})$	Heap

Figure 23: The operational semantics of simplified traces

The following notation for updating partial functions is used: $E[v \mapsto l]$ denotes the environment which is just like E, but maps v to l.

The new operation creates a new object $(T, \text{null}, \text{null})$ on the heap under a fresh location l and adds the result variable to the environment, mapping it to the new location l.

The get operation reads a field F out of an object, and adds the result variable to the environment, mapping it to the read location. The heap is unchanged.

The set operation changes field F of an object stored at the location that variable v maps to. The new value of the field is the location in variable u. The environment is unchanged.

The guard_class operation is used to check whether the object stored at the location that variable v maps to is of type T. If that is the case, then execution continues without changing heap and environment. Otherwise, execution is stopped.

6.4.1 *Optimizing traces*

To optimize the simple traces of the last section, online partial evaluation is used. The partial evaluator optimizes one operation of a trace at a time. Every operation in the unoptimized trace is replaced by a list of operations in the optimized trace. This list is empty if the operation can be optimized away. The optimization rules can be seen in Figure 24. Operation lists are written using angular brackets, $\langle\,\rangle$, list concatenation is expressed using two colons, $\text{ops}_1 :: \text{ops}_2$.

The state of the optimizer is stored in an environment E and a *static heap* S. Each step of the optimizer takes an operation, an environment and a static heap and produces a list of operations, a new environment and a new static heap.

The environment is a partial function from variables in the unoptimized trace V to variables in the optimized trace V^* (which are themselves written with a * for clarity). The reason for introducing new variables in the optimized trace is that several variables that appear in the unoptimized trace can turn into the same variables in the optimized trace. The environment of the optimizer serves a function similar to that of the environment in the semantics: to express sharing.

The static heap is a partial function from V^* into the set of static objects, which are triples of a type and two elements of V^*. The object referenced by a variable v^* is static, if v^* is in the domain of the static heap S. The object $S(v^*)$ describes what is statically known about the object, its type and its fields. The fields of objects in the static heap are also elements of V^* (or null, for short periods of time).

When the optimizer sees a new operation, it optimistically removes it and assumes that the resulting object can stay static. The optimiza-

new

$$\frac{v^* \text{ fresh}}{v = \texttt{new}(T), E, S \overset{\text{opt}}{\Longrightarrow} \langle\,\rangle, E\left[v \mapsto v^*\right], S\left[v^* \mapsto (T, \text{null}, \text{null})\right]}$$

get

$$\frac{E(v) \in \text{dom}(S)}{u = \texttt{get}(v, F), E, S \overset{\text{opt}}{\Longrightarrow} \langle\,\rangle, E\left[u \mapsto S(E(v))_F\right], S}$$

$$\frac{E(v) \notin \text{dom}(S), \;\; u^* \text{ fresh}}{u = \texttt{get}(v, F), E, S \overset{\text{opt}}{\Longrightarrow} \langle u^* = \texttt{get}(E(v), F)\rangle, E\left[u \mapsto u^*\right], S}$$

set

$$\frac{E(v) \in \text{dom}(S)}{\texttt{set}\left(v, F, u\right), E, S \overset{\text{opt}}{\Longrightarrow} \langle\,\rangle, E, S\left[E\left(v\right) \mapsto \left(S(E(v))!_F E(u)\right)\right]}$$

$$\frac{E(v) \notin \text{dom}\left(S\right), \left(E(u), S\right) \overset{\text{lift}}{\Longrightarrow} (\text{ops}, S')}{\texttt{set}\left(v, F, u\right), E, S \overset{\text{opt}}{\Longrightarrow} \text{ops} :: \langle \texttt{set}\left(E(v), F, E(u)\right)\rangle, E, S'}$$

guard

$$\frac{E(v) \in \text{dom}(S), \text{type}(S(E(v))) = T}{\texttt{guard\_class}(v, T), E, S \overset{\text{opt}}{\Longrightarrow} \langle\,\rangle, E, S}$$

$$\frac{E(v) \notin \text{dom}(S) \vee \text{type}(S(E(v))) \neq T, \; (E(v), S) \overset{\text{lift}}{\Longrightarrow} (\text{ops}, S')}{\texttt{guard\_class}(v, T), E, S \overset{\text{opt}}{\Longrightarrow} \text{ops} :: \langle \texttt{guard\_class}(E\left(v\right), T)\rangle, E, S'}$$

lifting

$$\frac{v^* \notin \text{dom}(S)}{v^*, S \overset{\text{lift}}{\Longrightarrow} \langle\,\rangle, S}$$

$$\frac{v^* \in \text{dom}(S), \; (v^*, S) \overset{\text{liftfields}}{\Longrightarrow} (\text{ops}, S')}{v^*, S \overset{\text{lift}}{\Longrightarrow} \langle v^* = \texttt{new}\left(\text{type}\left(S\left(v^*\right)\right)\right)\rangle :: \text{ops}, S'}$$

$$\frac{\left(S\left(v^*\right)_L, S \setminus \{v^* \mapsto S\left(v^*\right)\}\right) \overset{\text{lift}}{\Longrightarrow} \left(\text{ops}_L, S'\right), \left(S\left(v^*\right)_R, S'\right) \overset{\text{lift}}{\Longrightarrow} \left(\text{ops}_R, S''\right)}{v^*, S \overset{\text{liftfields}}{\Longrightarrow} \text{ops}_L :: \text{ops}_R :: \langle \texttt{set}\left(v^*, L, S\left(v^*\right)_L\right), \texttt{set}\left(v^*, R, S\left(v^*\right)_R\right)\rangle, S''}$$

Object Domains:

$u, v \in V$	variables in trace
$u^*, v^* \in V^*$	variables in opt. trace
$T \in \mathfrak{T}$	run − time types
$F \in \{L, R\}$	fields of objects

Semantic Values:

$E \in V \rightharpoonup V^*$	Environment
$S \in V^* \rightharpoonup \mathfrak{T} \times (V^* \cup \{\text{null}\}) \times (V^* \cup \{\text{null}\})$	Static Heap

Figure 24: Optimization rules

tion for all further operations is split into two cases. One case is for when the involved variables are in the static heap, which means that the operation can be performed at optimization time and can be removed from the trace. These rules mirror the execution semantics closely. The other case is for when not enough is known about the variables, and the operation has to be residualized.

If the argument v of a get operation is mapped to something in the static heap, the get can be performed at optimization time. Otherwise, the get operation needs to be residualized. Residualizing the operation means simply to insert a copy of the operation into the output operation list. As the first argument of the residualized operation, the variable with a star corresponding to the first argument of the original operation is chosen. The result variable of the residualized operation is a new variable with star, which is identified with the old result variable in the environment.

If the first argument v to a set operation is mapped to something in the static heap, then the set can be performed at optimization time which updates the static heap just as the corresponding rule of the operational semantics. Otherwise the set operation needs to be residualized. This needs to be done carefully, because the new value for the field, from the variable u, could itself be static, in which case it needs to be lifted first.

If a guard_class is performed on a variable that is in the static heap, the type check can be performed at optimization time, which means the operation can be removed if the types match. If the type check fails statically or if the object is not in the static heap, the guard_class is residualized. This also needs to lift the variable on which the guard_class is performed.

Lifting takes a variable and turns it into a dynamic variable. If the variable is already dynamic, nothing needs to be done. If it is in the static heap, operations are emitted that construct an object with the shape described there, and the variable is removed from the static heap.

Lifting a static object needs to recursively lift its fields. Some care needs to be taken when lifting a static object, because the structures described by the static heap can be cyclic. To make sure that the same static object is not lifted twice, the liftfield operation removes it from the static heap *before* recursively lifting its fields.

As an example for lifting, consider the static heap

$$\{v^* \mapsto (T_1, w^*, v^*), w^* \mapsto (T_2, u^*, u^*)\}$$

which contains two static objects. If v^* needs to be lifted, the following residual operations are produced:

```
$v^*$ = new($T_1$)
$w^*$ = new($T_2$)
set($w^*$, $L$, $u^*$)
set(w*, R, u*)
set(v*, L, w*)
set(v*, R, v*)
```

After the lifting the static heap is the empty set, because both static objects were lifted. If we had lifted w^* instead of v^*, then the following operations would have been produced:

```
w* = new(T2)
set(w*, L, u*)
set(w*, R, u*)
```

In this case, the static heap afterwards would be:

$$\{v^* \mapsto (T_1, w^*, v^*)\}$$

6.4.2 *Analysis of the algorithm*

While this thesis does not contain a formal proof of it, it can be argued informally that the algorithm presented above is sound: it works by delaying (and often completely removing) some operations. The algorithm runs in a single pass over the list of operations. Although recursively lifting a static object is not a constant-time operation, the algorithm only takes a total time linear in the length of the trace. This is due to the fact that the size of the static heap is bounded by the length of the trace and lifting always removes elements from it. The algorithm itself is not particularly complex. However, *in the context of tracing JITs* it is possible to find a simple enough algorithm that performs well.

The optimization completely removes the manipulation of objects in category 1 (those that do not escape); moreover, objects in category 2 (escaping) are still partially optimized: all the operations in between the creation of the object and the point where it escapes that involve the object are removed. Objects in category 3 and 4 are also partially optimized, their allocation is delayed till the end of the trace.

The optimization is particularly effective for chains of operations. For example, it is typical for an interpreter to generate sequences of writes-followed-by-reads, where one interpreted opcode writes to some object's field and the next interpreted opcode reads it back, possibly dispatching on the type of the object created just before. A typical example would be a chain of arithmetic operations.

6.5 metajit implementation

The allocation removal technique described in this chapter was implemented in the optimizer of MetaJIT. The optimization is independent of which interpreter the JIT is applied to. There are some practical issues beyond the techniques described in this chapter. The actual implementation needs to deal with more operations than described in Section 6.4, for example to also support static arrays in addition to static objects. The implementation of this optimization is about 400 lines of RPython code. Furthermore, on a guard failure the static objects need to be reconstructed before leaving to the interpreter, which requires some code to store a description of the static objects in a space-efficient way for every guard [SB12].

A further complication is that most interpreters written with RPython use reified heap-allocated frame objects to store local variables. Those severely hinder the effectiveness of allocation removal, because every time an object is stored into a local variable in the user program, it is stored into the frame object, which makes it escape. We implemented a technique to treat such frames objects in a special way to solve this problem: While machine code is executed, the frame object is not updated all the time but its content can exist as variables in the trace (i.e., in the CPU registers and on the CPU stack). Only when the frame is actually accessed by code outside of the trace is the frame object filled. This is a common approach in VM implementations [Mir99, GES^{+}09]; the novelty of our approach is that we generalized it enough to be usable for different interpreters.

6.6 EVALUATION

To evaluate the allocation removal algorithm, its effectiveness when used in PyPy's Python interpreter is analyzed (this interpreter will be described in more detail in Chapter 10).

The benchmarks used are small-to-medium Python programs, some synthetic benchmarks but most of them (parts of) real applications.[5] The benchmarks are listed and described in Appendix A.

The allocation removal algorithm is evaluated along two lines: how many allocations and other operations can it optimize away and how much are the run times of the benchmarks improved by that. For that, two versions of PyPy's Python interpreter are used, one that does not includes the allocation removal optimization (`opt1`), the other does (`opt2`).[6]

As the first step, the number of the occurring operations were counted in all generated traces before and after the optimization phase for all benchmarks using `opt2`. Figure 26 shows the numbers of `new`, `get` and `set` operations and how many of them are removed by the allocation removal optimization (no optimization in `opt1` removes

[5] All the benchmarks and the scripts to run them can be found at: `http://cfbolz.de/phdthesis/`
There is also a website that monitors PyPy's performance nightly at: `http://speed.pypy.org/`

[6] Both versions disable the heap optimization pass and the loop invariant code motion pass [ABF12]. These passes disturb the measurements of the allocation removal optimizations. They can remove some of the `set` and `get` operations that the allocation removal gets rid of as well. This means that the fastest version of PyPy is faster than the performance numbers presented here, see Chapter 10 for more details.

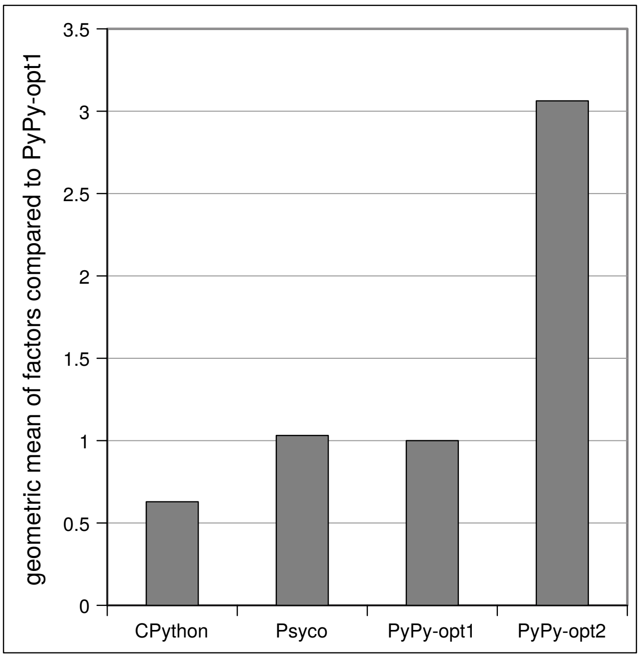


Figure 25: Overview of how much faster than PyPy without allocation removal the variants are (higher is better)

any of them). The optimization removes between 74% and 92% of allocation operations in the traces of the benchmarks. All benchmarks taken together, the optimization removes 86% percent of allocation operations. The numbers look similar for the writing of fields, reading of fields is affected less.

Figure 27 shows the effect on guards, numeric operations and the remaining types of operations that the two optimization configurations have. The table shows the number of operations for each category, what percentage of these are removed when using `opt1` and what percentage are removed when using `opt2`. The table shows that allocation removal removes an additional number of guards for all benchmarks. It also shows that allocation removal can give more optimization opportunities to other optimizations. Even though allocation removal itself does not remove operations from the numeric and rest categories, enabling it leads to the additional removal of these kinds of operations.

In addition to the count of operations Figures 25 and 28 presents some time measurements. For each implementation the table also reports the speedup over PyPy without allocation removal. With the optimization turned on, PyPy's Python interpreter outperforms CPython in all benchmarks. All benchmarks are improved by the allocation removal optimization, by at least 32% and by as much as a factor of 8.84.

	num loops	new	opt2	get	opt2	set	opt2
chaos	38	1966	88%	2327	22%	9213	90%
crypto_pyaes	35	1385	83%	1360	0%	9807	89%
django	36	1036	84%	2221	15%	6681	89%
go	804	47331	89%	71740	23%	306087	91%
pyflate-fast	144	5769	85%	11437	45%	35948	92%
raytrace-simple	114	6891	87%	6203	8%	43696	91%
richards	50	1808	92%	2547	20%	15338	94%
spambayes	584	14880	80%	27735	17%	103139	75%
sympy_expand	272	10838	74%	17818	18%	61604	83%
telco	95	5708	91%	7776	19%	31924	91%
twisted_names	298	16305	84%	30417	17%	103357	88%
total	2470	113917	85%	181581	21%	726794	88%

Figure 26: Number of new/get/set operations and percentage removed by optimization

	guard	opt1	opt2	numeric	opt1	opt2	rest	opt1	opt2
chaos	4780	15%	33%	1666	8%	24%	4306	43%	46%
crypto_pyaes	2977	28%	36%	1550	30%	32%	3064	25%	27%
django	4009	22%	34%	571	18%	33%	3188	46%	48%
go	103249	24%	37%	17722	23%	52%	86602	46%	48%
pyflate-fast	17519	15%	42%	3819	20%	54%	15883	39%	43%
raytrace-simple	13996	16%	28%	2385	18%	24%	15479	50%	54%
richards	5269	19%	29%	697	31%	60%	5449	52%	55%
spambayes	41504	17%	32%	13183	21%	35%	34066	36%	40%
sympy_expand	33873	20%	34%	4366	17%	32%	27280	48%	52%
telco	16084	21%	32%	2065	21%	36%	13264	57%	59%
twisted_names	52621	24%	35%	9763	24%	42%	37535	46%	48%
total	295881	21%	35%	57787	22%	42%	246116	45%	48%

Figure 27: Effect of optimization on other operations

	Python [ms]	Psyco [ms]	PyPy opt1 [ms]	PyPy opt2 [ms]
chaos	364.68 ± 4.56	147.47 ± 1.13	122.80 ± 2.76	30.33 ± 0.64
	0.34 ×	0.83 ×	1.00 ×	4.05 ×
crypto_pyaes	2098.80 ± 3.69	67.66 ± 4.73	819.42 ± 21.17	189.36 ± 20.19
	0.39 ×	12.11 ×	1.00 ×	4.33 ×
django	753.89 ± 4.03	827.37 ± 2.32	290.40 ± 12.69	90.91 ± 9.75
	0.39 ×	0.35 ×	1.00 ×	3.19 ×
go	724.85 ± 5.80	423.63 ± 5.03	964.61 ± 246.69	349.73 ± 298.83
	1.33 ×	2.28 ×	1.00 ×	2.76 ×
pyflate-fast	2418.29 ± 20.28	1157.32 ± 9.84	1711.35 ± 60.01	992.97 ± 53.81
	0.71 ×	1.48 ×	1.00 ×	1.72 ×
raytrace-simple	1910.54 ± 9.06	981.73 ± 2.88	782.14 ± 30.64	93.41 ± 29.13
	0.41 ×	0.80 ×	1.00 ×	8.37 ×
richards	256.23 ± 2.51	63.52 ± 1.78	154.39 ± 2.27	17.46 ± 1.43
	0.60 ×	2.43 ×	1.00 ×	8.84 ×
spambayes	242.61 ± 2.08	249.32 ± 2.56	277.64 ± 182.60	223.48 ± 221.97
	1.14 ×	1.11 ×	1.00 ×	1.24 ×
sympy_expand	1133.99 ± 11.55	12691.42 ± 7934.46	1326.91 ± 84.87	1008.60 ± 122.12
	1.17 ×	0.10 ×	1.00 ×	1.32 ×
telco	957.50 ± 8.60	676.50 ± 10.46	508.23 ± 24.66	116.61 ± 22.57
	0.53 ×	0.75 ×	1.00 ×	4.36 ×
twisted_names	7.88 ± 0.03	8.00 ± 0.03	5.86 ± 0.27	3.58 ± 0.46
	0.74 ×	0.73 ×	1.00 ×	1.64 ×
geom. mean	0.63	1.04	1.00	3.07

Figure 28: Benchmark times in milliseconds, together with factor over PyPy without allocation removal

6.7 CONCLUSION

In this chapter, an optimization based on online partial evaluation is described that optimizes away allocations and type guards in the traces of a tracing JIT. This optimization can reduce the overhead of boxing primitive types significantly and therefore bring the performance of dynamic languages closer to that of statically typed ones for applications that perform many arithmetic or other allocation-heavy operations. The approach is language-independent and needs no changes to the language interpreter to work. However, sometimes changes to the language interpreter make objects escape less often, which can increase the effectiveness of the optimization.

In the meta-tracing context a simple optimization based on partial evaluation gives good results. This is due to the fact that the tracing JIT itself is responsible for all control issues, which are usually the hardest part of partial evaluation: the tracing JIT selects the parts of the program that are worthwhile to optimize, and extracts common linear paths through them, inlining functions as necessary. What is left to optimize are only those linear paths. The relation of meta-tracing and partial evaluation will be explored in more detail in Chapter 12.

A similar approach to optimization as described in this chapter for allocation removal can likely be taken for other optimizations. Many optimizations have the property that they work well on linear code paths but require a complex (often whole program) analysis phase for making them work in larger scopes than a basic block. Tracing gives such optimizations a lot more context to work with by selecting interesting linear code paths and by aggressively inlining called functions. For example, MetaJIT also contains a store-load propagation optimization [BGS99] with a very simple type-based alias analysis [DMM98]. Of course a trace still does not give truly global information. For example it is not possible to know about anything outside of the trace. However, for many optimization such knowledge is not necessary.

7 SUMMARY

In this first half of the thesis the foundations of meta-tracing have been described. To enable meta-tracing, two hints, `jit_merge_point` and `can_enter_jit` are needed that control the unrolling of the interpreter loop to get traces that correspond to the loops in the user program. Two further optional hints, `promote` and `elidable` were described in Chapter 5. These hints can be used to improve the traces produced by the meta-tracer to get better performance. These hints guide the feedback of run-time information into the trace, and the exploitation of such static information. They make it possible to express object model optimizations and are the main tool for language implementors to improve the performance of their interpreter when using meta-tracing.

Complementing the application of these hints is a language-independent optimization that removes operations that operate on short-lived heap objects. Whereas the annotations of Chapter 5 are typically used to optimize operations on complex objects, such as instances and classes, the optimization of Chapter 6 is most typically useful for removing the overhead of primitive types, such as integers and floats.

Another view of meta-tracing is that it separates language performance concerns into several layers:

- The language semantics are specified by the language interpreter.
- Language-specific optimizations are given by adding hints to the interpreter.
- Optimizations that are independent of the language and the target machine are part of the meta-tracing JIT.
- Target-specific code is generated by the architecture-specific backend.

This separation can be likened to the traditional architecture of (ahead-of-time) compilers that are split into frontend, optimizations and backend. The difference is that in the meta-tracing model the frontend takes the form of an interpreter, which is easier to write and understand.

Together the meta-tracing techniques should be able to remove most of the overheads of dynamic languages. How much they can improve the performance is explored in the next half of this thesis, where they are applied to three interpreters, one for regular expressions, one for Python, and one for Prolog.

Part III

EVALUATION

8 INTRODUCTION

The first half of this thesis showed a number of techniques for improving the performance of interpreters by tracing through their implementation. This second half evaluates these techniques using three interpreters.

The first one, presented in Chapter 9, is a small interpreter for regular expressions. It is small enough to be fully shown in the chapter but still executes an interesting language.

The second interpreter is at the other extreme on the complexity scale. Chapter 10 presents the Python interpreter that gives PyPy its name in more detail. Python is an object-oriented imperative dynamic language with very complex object semantics. The chapter describes the rewrites of the interpreter that are necessary to speed this language up in interesting ways. This interpreter is also used to do a thorough evaluation of the speed effects of the techniques presented in Part II.

The third interpreter is a Prolog interpreter presented in Chapter 10. Prolog is a logical declarative language featuring unification, nondeterminism and backtracking. It is neither imperative nor object-oriented. The chapter evaluates how well the meta-tracing approach works for such languages.

After looking at concrete implementations, Chapter 12 places meta-tracing into a wider context. The chapter compares meta-tracing to partial evaluation, which is the traditional approach for improving the performance of interpreters without writing a compiler instead. The chapter tries to illuminate the differences and commonalities of the two approaches by presenting executable models for both, written in Prolog.

9 CASE STUDY: REGULAR EXPRESSIONS

This chapter presents the first case study for MetaJIT, a simple regular expression interpreter. There are two typical approaches to implement regular expression. A naive one is to use a back-tracking implementation, which can lead to exponential matching times for certain regular expressions.

The other, more complex one, is to transform the regular expression into a non-deterministic finite automaton (NFA) [RS59] and then transform the NFA into a deterministic finite automaton (DFA). A DFA can be used to efficiently match a string, the problem of this approach is that turning an NFA into a DFA can lead to exponentially large automatons. This can be seen by looking at the regular expression `(a|b)*a(a|b){`n`}a(a|b)*` where n is a natural number. This regular expression matches strings of `a` or `b` characters that contain two `a` with exactly n characters between them. The DFA for this regular expression needs to be have at least 2^n states to remember at which of the last n positions an `a` was found.

Given this problem of potential size explosion, a more sophisticated approach to matching is to not construct the full DFA, but instead use the NFA for matching. This requires some care, because it is necessary to keep track of which set of states the automaton is in (it is not just one state, because the automaton is non-deterministic).

The algorithm implemented in this chapter is essentially equivalent to that approach, however it does not need an intermediate NFA and instead represents a state of the corresponding DFA as a marked regular expression (represented as a tree of nodes). The algorithm described here follows a paper by Fischer et al. [FHW10]. For many details about alternative approaches to implement regular expressions efficiently, see Russ Cox's comprehensive article collection.[1]

9.1 THE ALGORITHM

In the algorithm the regular expression is represented as a tree of nodes. The leafs of the nodes can match exactly one character (apart from the epsilon node, which matches the empty string). The inner nodes of the tree combine other nodes in various ways, like alternative, sequence or repetition. Every node in the tree can potentially have a mark. The meaning of the mark is that a node is marked, if that sub-expression matches the string seen so far.

[1] `http://swtch.com/~rsc/regexp/`

```python
class Regex(object):
    def __init__(self, empty):
        # empty denotes whether the regular expression
        # can match the empty string
        self.empty = empty
        # mark that is shifted through the regex
        self.marked = False

    def reset(self):
        """ reset all marks in the regular expression """
        self.marked = False

    def shift(self, c, mark):
        """ shift the mark from left to right, matching character c."""
        marked = self._shift(c, mark)
        self.marked = marked
        return marked

    def _shift(self, c, mark):
        raise NotImplementedError("abstract base class")

    def match(self, s):
        """ check whether regular expression matches a string."""
        if not s:
            return self.empty
        # shift a mark in from the left
        result = self.shift(s[0], True)
        for c in s[1:]:
            # shift the internal marks around
            result = self.shift(c, False)
        self.reset()
        return result
```

Figure 29: Base class of all regular expression classes and matching function

```
class Char(Regex):
    def __init__(self, c):
        Regex.__init__(self, False)
        self.c = c

    def _shift(self, c, mark):
        return mark and c == self.c
```

Figure 30: Matching characters

```
class Epsilon(Regex):
    def __init__(self):
        Regex.__init__(self, empty=True)

    def _shift(self, c, mark):
        return False
```

Figure 31: Matching empty strings

The basic approach of the algorithm is that for every character of the input string the regular expression tree is walked and some of the nodes in the regular expression are marked. At the end of the string, if the top-level node is marked, the string matches. Otherwise it does not. At the beginning of the string, one mark gets shifted into the regular expression from the top, and then the marks that are in the regular expression already are shifted around for every additional character.

Let's start looking at some code, and an example to make this clearer.[2] The base class of all regular expressions is shown in Figure 29.

The most important subclass of `Regex` is `Char`, which matches one concrete character, see Figure 30. Shifting the mark through `Char` is easy: a `Char` instance retains a mark that is shifted in when the current character is the same as that in the instance.

Another easy case is that of the empty regular expression `Epsilon`, see Figure 31. `Epsilons` never get a mark, but they can match the empty string.

9.1.1 *Alternative*

Now the more interesting cases remain. First we define an abstract base class `Binary` for the case of composite regular expressions with two children, and then the first subclass `Alternative` which matches if either of two regular expressions matches the string, see Figure 32. The usual regular expressions syntax for alternatives is `a|b`.

[2] The source code together with the benchmarks can be found here: `http://cfbolz.de/phdthesis/`

```
class Binary(Regex):
    def __init__(self, left, right, empty):
        Regex.__init__(self, empty)
        self.left = left
        self.right = right

    def reset(self):
        self.left.reset()
        self.right.reset()
        Regex.reset(self)

class Alternative(Binary):
    def __init__(self, left, right):
        empty = left.empty or right.empty
        Binary.__init__(self, left, right, empty)

    def _shift(self, c, mark):
        marked_left  = self.left.shift(c, mark)
        marked_right = self.right.shift(c, mark)
        return marked_left or marked_right
```

Figure 32: Alternative

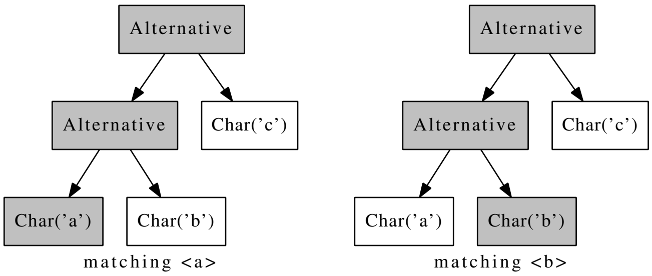


Figure 33: Matching a simple alternative with input-strings 'a' and 'b'

An `Alternative` can match the empty string if either of its children can. Similarly, shifting a mark into an `Alternative` shifts it into both its children. If either of the children are marked afterwards, the `Alternative` is marked as well.

As an example, consider the regular expression `a|b|c`, which would be represented by the following objects:
`Alternative(Alternative(Char('a'), Char('b')), Char('c'))`
Figure 33 shows the marks that the objects would get when matching the strings `"a"` and `"b"` (gray nodes are marked, white ones are unmarked). At the start of the process, no node is marked. Then the first char is matched, which adds a mark to the `Char('a')` node, and the mark will propagate up the two `Alternative` nodes.

```
class Repetition(Regex):
    def __init__(self, re):
        Regex.__init__(self, empty=True)
        self.re = re

    def _shift(self, c, mark):
        return self.re.shift(c, mark or self.marked)

    def reset(self):
        self.re.reset()
        Regex.reset(self)
```

Figure 34: Repetition

9.1.2 *The Kleene star*

The two remaining classes are slightly trickier. The Kleene star is used to match a regular expression any number of times. The usual syntax for is is `a*`. It is implemented in the class `Repetition`, see Figure 34.

A `Repetition` can always match the empty string. The mark is shifted into the child, but if the `Repetition` is already marked, this will be shifted into the child as well, because the `Repetition` could match another time.

Figure 35 shows as an example the regular expression `(a|b|c)*` matching the string `abcbac`. For every character, one of the alternatives matches, thus the repetition matches as well.

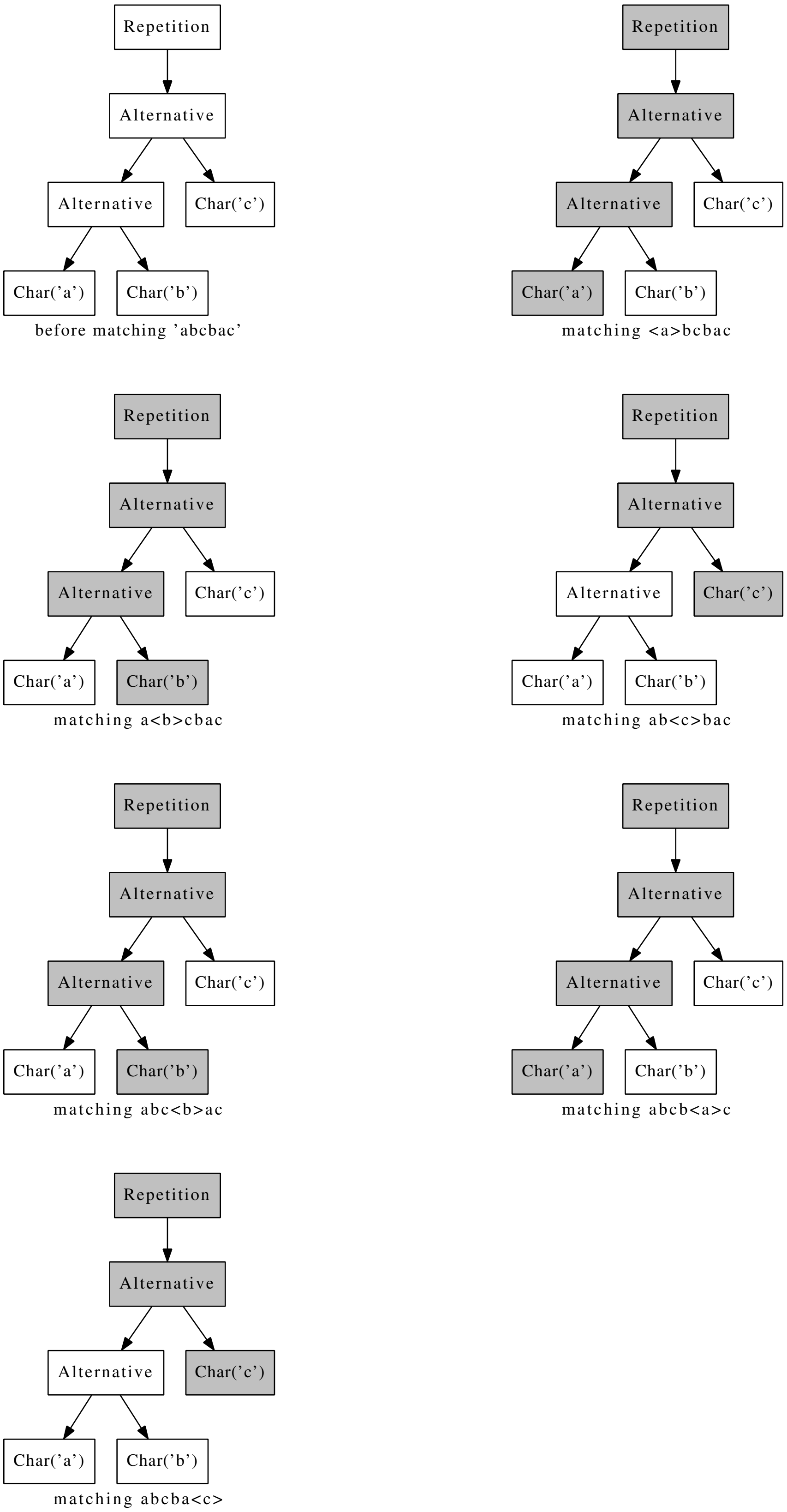
9.1.3 *Sequence*

The only missing class is that for sequences of expressions, `Sequence`. The code can be seen in Figure 36. It's usual regular expressions syntax is `ab`.

A `Sequence` can be empty only if both its children are empty. The mark handling is slightly more complicated than that of the other classes. If a mark is shifted in, it will be shifted to the left child regular expression. The right child gets a mark shifted in either if the left child can match the empty string, or if the left child is already marked *before the shift*.

The whole sequence matches (meaning that it is marked) in two cases. On the one hand, it matches if the right child is marked. On the other, it also matches if the left child is marked after the shift and the right child can match the empty string.

Consider the regular expression `abc` matching the string `abcd`, shown in Figure 37. For the first three characters, the marks wander from left to right. When the character `d` is reached, the matching fails.



before matching 'abcbac'

matching <a>bcbac

matching a<b>cbac

matching ab<c>bac

matching abc<b>ac

matching abcb<a>c

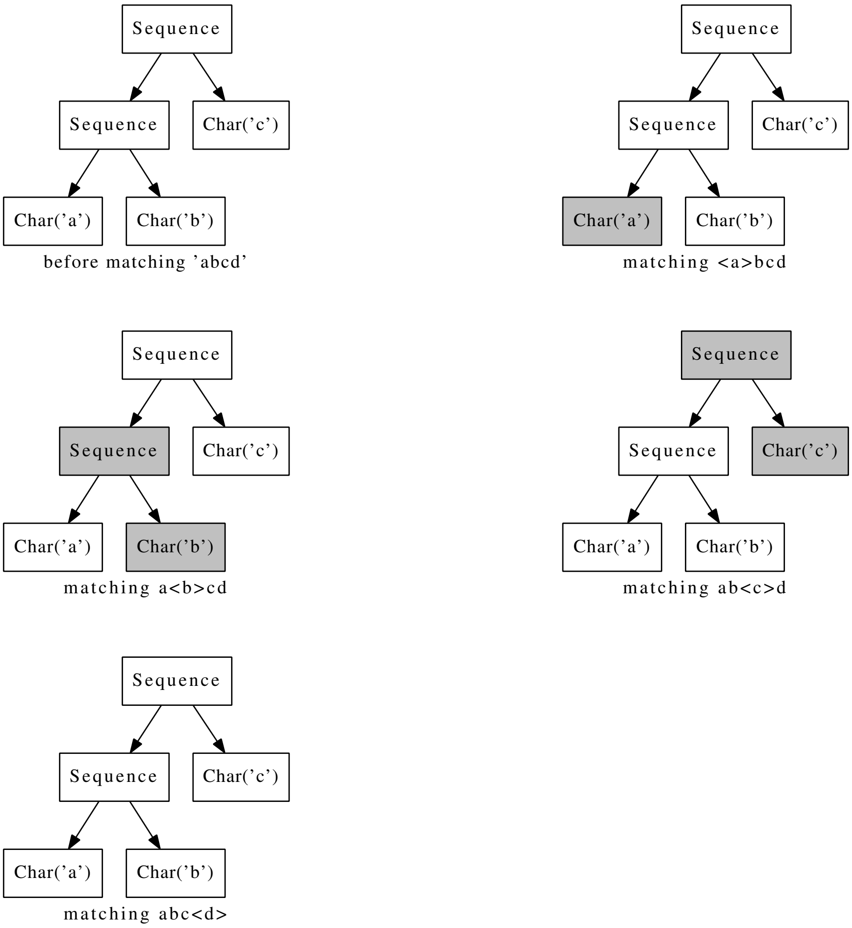
matching abcba<c>

Figure 35: Matching a repetition

```
class Sequence(Binary):
    def __init__(self, left, right):
        empty = left.empty and right.empty
        Binary.__init__(self, left, right, empty)

    def _shift(self, c, mark):
        old_marked_left = self.left.marked
        marked_left = self.left.shift(c, mark)
        marked_right = self.right.shift(
            c, old_marked_left or (mark and self.left.empty))
        return (marked_left and self.right.empty) or marked_right
```

Figure 36: Sequence



before matching 'abcd'

matching <a>bcd

matching a<b>cd

matching ab<c>d

matching abc<d>

Figure 37: Matching a sequence

9.1.4 *A more complex example*

As a more complex example, Figure 38 shows the expression `((ab)*|(abc))(c|e)` matching the string `abababe`. Note how the two branches of the first alternative match the first `ab` in parallel, until it becomes clear that only the left alternative `(ab)*` can work.

9.1.5 *Conclusion*

The code shown in this section is a compact implementation of regular expressions. The `match` function loops over the entire string without going back and forth. Each iteration goes over the whole tree every time. Thus the complexity of the algorithm is $O(mn)$ where m is the size of the regular expression and n is the length of the string. Therefore, no pathological cases exist. By itself, the Python code shown is not terribly efficient. In the next section meta-tracing will be used to make the matcher significantly faster.

9.2 META-TRACING THE REGULAR EXPRESSION MATCHER

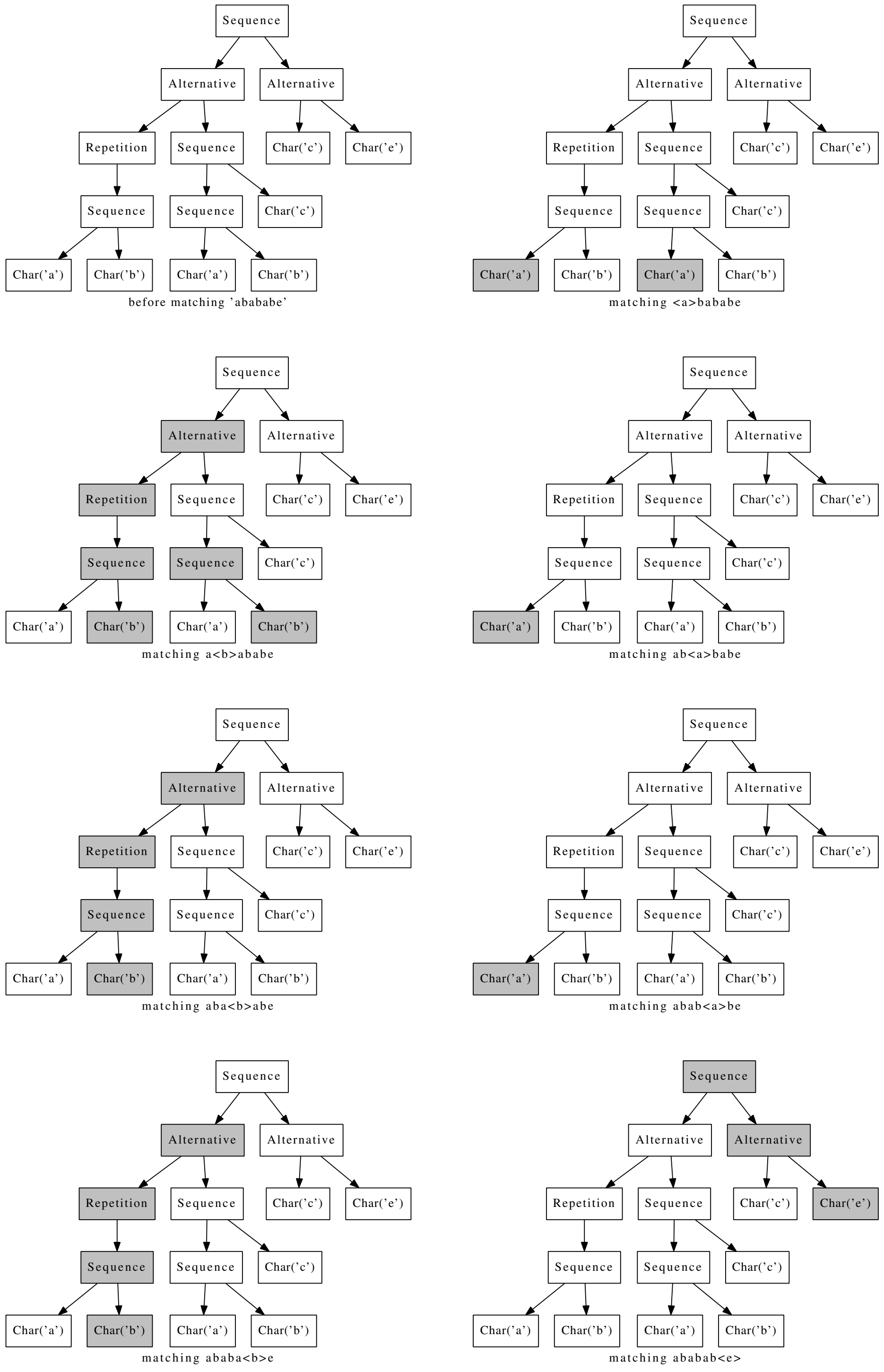
This section describes how meta-tracing can be used to turn the short but not particularly fast regular expression matcher into a rather fast implementation. The speed will be evaluated by comparing the matcher against other regular expression implementations.

9.2.1 *Reference performance numbers*

The regular expression `(a|b)*a(a|b){20}a(a|b)*` is used as an example. It matches all strings consisting of `a` and `b` characters that have two `a` with exactly 20 characters between them. Converting this regular expression to a DFA would be impractical, since the DFA needs 2^{21} states. As an input string, a random string (of varying lengths) that does *not* match the regular expression is used. The unit to measure the speed of a regular expression implementation is the number of chars matched per second. While this is not a particularly typical regular expression, it should still be possible to get some rough numbers for the speeds of various implementations – as we will see, the differences between implementations are big anyway.

To get started, the CPython `re` module[3] (which is implemented about 4000 lines of C and quite optimized) can match 1 950 000 chars/s. On the other end of the performance scale is the pure-Python code from the last section running on CPython. It can match only 15 800 chars/s and is thus 123 times slower than the `re` module.

[3] `http://docs.python.org/library/re.html`



before matching 'abababe'

matching <a>babe

matching a<b>ababe

matching ab<a>babe

matching aba<b>abe

matching abab<a>be

matching ababa<b>e

matching ababab<e>

Figure 38: Matching a complex regular expression

9.2.2 *Translating the matcher*

The code described in the last section is not only normal Python code, but also valid RPython code. Nothing dynamic is going on in the code, thus it can be translated with the RPython translation toolchain to C code. The resulting binary is considerably faster and can match 1 055 000 chars/s, 66 times faster than the untranslated version.

Another approach is to write equivalent versions of the algorithms in lower level languages. This has been done for C++ by Sebastian Fischer and for Java by Baltasar Trancón y Widemann. The code presented in this chapter an be mapped very closely to both of these languages. The Python code above is about 90 lines long, the C++ version 100 and the Java version about 200 lines. The C++ version is a little bit faster than the RPython version translated to C, at 1 099 000 chars/s. This is not very surprising, given their similarity. The Java version is almost ten times faster, with 10 080 000 chars/s. Apparently the HotSpot Java JIT compiler is a lot better at optimizing the method calls in the algorithm or does some other optimizations. One reason for this could be that the Java JIT can assume that the classes it sees are all there are (and it will invalidate the generated machine code if more classes are loaded), whereas the C++ compiler needs to generate code that works even in the presence of more regular expression classes.[4]

9.2.3 *Adding JIT hints*

To apply MetaJIT to the regular expression matcher the code for string matching needs to be extended by the hints described in Chapter 4. This works well, because the regular expression matcher can be seen as an interpreter for regular expressions. Then the `match` function corresponds to the dispatch loop of a traditional interpreter.

The regular expression matcher is a very peculiar interpreter in that the matcher works by running exactly one loop (the one in `match`) as many times as the input string is long, irrespective of the "program", i.e. the particular regular expressions. In addition, within the loop there are no conditions (e.g. `if` statements) at all, it is just linear code. This makes it very well-suited for a tracing JIT, as no control flow at all needs to be taken into consideration: There is exactly one loop per regular expression, without any guards in them, except one to end the loop.

The hints that are needed for the `match` function of Figure 29 can be seen in Figure 39. The `self` variable is listed as the only green local variable, making the currently matched regular expression the program of the interpreter (which adds it to the position key as per

[4] These numbers are using Java 7. The achieved performance was still a lot worse with Java 6, meaning that the speedup is from a recent optimization.

```
jitdriver = jit.JitDriver(reds=["i", "s"], greens=["self"])

class Regex(object):
    ...

    def match(self, s):
        if not s:
            return self.empty
        # shift a mark in from the left
        result = self.shift(s[0], 1)
        i = 1
        while i < len(s):
            jitdriver.jit_merge_point(i=i, s=s, self=self)
            # shift the internal marks around
            result = self.shift(s[i], 0)
            i += 1
        self.reset()
        return result
```

Figure 39: Applying JIT hints to the matching function

```
class Char(Regex):
    _immutable_fields_ = ["c"]
    def __init__(self, c):
        ...
```

Figure 40: Declaring the `Char` class to be immutable

Chapter 4). There is no part of the regular expression matcher that corresponds to a backward jump bytecode, so the `can_enter_jit` hint is simply left out. The `jit_merge_point` hint is put into the loop.

In addition to these hints we also declare that all of the fields of the subclasses of `Regex` except `marked` are immutable. For example for the `Char` class this is expressed as in Figure 40. These hints allow MetaJIT to constant-fold reads out of the immutable fields in some situations, as described in Chapter 5.

9.2.4 *Adaptations to the original code*

In the introduction above it was stated that the code within the loop in `match` uses no conditions. It is indeed true that none of the `_shift` methods have an `if` statement or similar. However, there are some hidden conditions because the `and` and `or` boolean operators are used, which are short-circuiting. Therefore the JIT-version of the code needs to be adapted to use the non-short-circuiting operators `&` and `|`.

9.2.5 *Generated code example*

As an example of how the generated machine code looks like, consider the regular expression `(a|b)*`. As regular expression objects

```
[$i_0$, $result_0$, $s_0$]                                                   1
char = $s_0$[$i_0$] # read the character                                     2
# read the current mark:                                                     3
$i_5$ = ConstPtr(ptr_repetition).marked                                      4
$i_7$ = char == 'a' # is the character equal to 'a'                          5
$i_8$ = $i_5$ & $i_7$                                                        6
$i_{10}$ = char == 'b' # is the character equal to 'b'                       7
$i_{11}$ = $i_5$ & $i_{10}$                                                  8
# write new mark                                                             9
ConstPtr(ptr_chara).marked = $i_8$                                           10
$i_{13}$ = $i_8$ | $i_{11}$                                                  11
# write new mark                                                             12
ConstPtr(ptr_charb).marked = $i_{11}$                                        13
# write new mark                                                             14
ConstPtr(ptr_alternative).marked = $i_{13}$                                  15
# increment the index                                                        16
$i_{17}$ = $i_0$ + 1                                                         17
$i_{18}$ = len($s_0$)                                                        18
# write new mark                                                             19
ConstPtr(ptr_repetition).marked = $i_{13}$                                   20
# check that index is smaller than the length of the string                  21
guard($i_{17}$ < $i_{18}$)                                                   22
jump($i_{17}$, $i_{13}$, $s_0$) # start from the top again                   23
```

Figure 41: Trace for matching `(a|b)*`

this would be:
`Repetition(Alternative(Char('a'), Char('b')))`
The trace that is generated looks as in Figure 41.

The various `ConstPtr(ptr_*)` denote constant addresses of parts of the regular expression tree:

- `ptr_repetition` is the `Repetition`
- `ptr_chara` is `Char('a')`
- `ptr_charb` is `Char('b')`
- `ptr_alternative` is the `Alternative`

Essentially the machine code reads the next char out of the string (line 2), the current mark out of the `Repetition` (line 4). The char is compared to `a` and `b` (lines 5 and 7) and then some boolean operations are performed on the previous mark and the result of the comparison (lines 6, 8, 11). Then the newly computed marks are stored back into the regular expression objects (lines 10, 13, 15, 20). Afterwards there is a guard that the end of the string is not reached, then the trace ends with a jump. Note how the generated machine code does not need to do any method calls to `shift` and `_shift` and that most field reads out of the regular expression classes have been optimized away, because the fields are immutable. Therefore the machine code does not need to deconstruct the tree of regular expression objects at all, it just knows where in memory the parts of it are, and encodes that directly into the code.

	chars per second	speedup vs. Python
Pure Python code	15 830	1
Python re module	1 949 000	123
RPython implementation compiled to C	1 018 000	64
C++ implementation	1 097 000	69
Java implementation	10 370 000	655
RPython implementation with MetaJIT	7 526 000	475

Figure 42: Matching speed of various regular expression implementations

9.2.6 *Performance results with MetaJIT*

With the regular expression matcher translated to C and with MetaJIT, the regular expression performance increases significantly. Our running example can match 7 377 000 chars/s, which is almost four times faster than the `re` module. This is not an entirely fair comparison, because the `re` module can give more information than just "matches" or "doesn't match", but it is still interesting to see. A more relevant comparison is that between the program with and without the JIT: Using MetaJIT speeds the matcher up by more than 7 times. All the results that appeared in this chapter can be seen in Figure 42.

The matcher without MetaJIT is slower, because for every character of the input string it needs to walk over the whole regular expression tree, reading fields out of the inner tree nodes and calling the `_shift` method of their children. Every method call is expensive, because it needs to dispatch to the right implementation. Also, during the matching the current character is compared many times to `'a'` and `'b'`. MetaJIT gets rid of all these sources of inefficiency by inlining all `_shift` methods and optimizing the trace, as seen in Figure 41.

9.3 CONCLUSION

The simple and slow regular expression matching algorithm described in the first part of the section was translated to C and was sped up significantly. The real win however is gained by using the MetaJIT for the matcher, which can be regarded as an interpreter. The resulting matcher is quite fast. Adding the hints for this simple interpreter is very easy, only the hints from Chapter 4 and some immutability information are needed. This shows that MetaJIT can be used on small interpreters without much effort, but giving large speed improvements.

10 CASE STUDY: PYTHON

After the small example interpreter from the last chapter, this chapter will present PyPy's Python interpreter, and how its operations were optimize by adding hints. Particular attention will be given to common object operations and how they can be supported well. The chapter also contains a detailed performance evaluation of MetaJIT and the various techniques described in Part II of this thesis as used in the Python interpreter.

PyPy's Python interpreter is fully written in RPython, in about 93 500 lines of code. It can be roughly split into four parts:

- the bytecode interpreter (about 8000 lines of code)
- the library of built-in objects, called *object space* (about 18 000 lines of code)
- the parser and bytecode compiler (about 7500 lines of code)
- the library of built-in modules (about 60 000 lines of code split into about 70 modules)

Of these, only the bytecode interpreter and the built-in object implementation interact with MetaJIT, so only they will be described.

10.1 THE BYTECODE INTERPRETER AND META-TRACING

The bytecode interpreter supports the control flow constructs of the Python language with a straightforward stack-based bytecode instruction set. PyPy's Python interpreter uses the same instruction set as CPython, with a few minor changes to support specific optimization. The instruction set contains opcodes for:

- variable handling: `LOAD_FAST`, `STORE_FAST`, `LOAD_CONST`, `LOAD_GLOBAL`, ...
- stack manipulation: `POP_TOP`, `DUP_TOP`, ...
- control flow: `JUMP_FORWARD`, `POP_JUMP_IF_FALSE`, `FOR_ITER`, ...
- unary and binary operations on objects: `UNARY_NOT`, `BINARY_ADD`, `BINARY_MUL`, ...
- object manipulation: `LOAD_ATTR`, `STORE_ATTR`, `BUILD_CLASS`, ...

The bytecode interpreter is implemented in a very simple, direct way. The only interesting property of the design of PyPy's Python interpreter is that it tries to cleanly separate the bytecode dispatching and the implementation of all the opcodes that are concerned with operations on objects. To that end, the bytecode interpreter treats all Python objects that it handles as black boxes. All operations that it performs on them are dispatched to the object space (see next subsection).

CPython (and thus also PyPy) is according to Brunthaler's terminology [Bru09] a *high abstraction level virtual machine*. This means that "operation implementation requires *significantly* more native machine instructions than for low abstraction level [virtual machines]" where "operation implementation can be directly translated to a few native machine instructions" [Bru09] [emphasis in the original]. In other words, many bytecodes within the Python instruction set have rich behaviour with complex dispatching rules to find the right implementation for the types at hand. An example is `BINARY_ADD` corresponding directly to the syntax `a + b` which can add numbers (which are themselves split into different types, such as integers, floats, long integers, complex numbers, ...) but also concatenate strings and lists as well as do user-defined operations.[1]

Figure 43 shows the hints from Chapter 4 in the context of the source code of the Python interpreter.[2] The JIT driver lists the local variables of the main bytecode dispatch loop. Of these, three make up the green variables, which signify the position in the program that is currently being executed. These are `pycode` which is the bytecode object of the current function, `next_instr` which is the program counter and `is_being_profiled` which is a flag that specifies whether profiling is turned on. Because this flag is green, the meta-tracer produces different traces for when a function is run without profiling and for when it is run with profiling.

The `jit_merge_point` is at the beginning of the `while` loop of the `dispatch` method. This method calls `handle_bytecode` which executes the next bytecode instruction, returning the new program counter position. It can also raise an `ExitFrame` exception, when the function execution is finished.

The `can_enter_jit` hint is placed in the implementation of the `jump_absolute` bytecode, which is the only way a backward jump happens.

[1] A consequence of this high abstraction level is that most classical techniques that reduce the bytecode dispatch overhead of interpreters, such as threaded code [Bel73] and superinstructions [Pro95] don't help much in the case of Python. Python's bytecode dispatch overhead is low compared to the complex dispatching that goes on within the bytecode [Bru09].

[2] The code is lightly edited to leave out some implementation details.

```
pypyjitdriver = JitDriver(
    reds = ['frame', 'ec'],
    greens = ['next_instr', 'is_being_profiled', 'pycode'])


class PyFrame(eval.Frame):
    ...

    def dispatch(self, pycode, next_instr, ec):
        is_being_profiled = self.is_being_profiled
        try:
            while True:
                pypyjitdriver.jit_merge_point(ec=ec,
                    frame=self, next_instr=next_instr, pycode=pycode,
                    is_being_profiled=is_being_profiled)
                co_code = pycode.co_code
                self.valuestackdepth = promote(self.valuestackdepth)
                next_instr = self.handle_bytecode(
                    co_code, next_instr, ec)
                is_being_profiled = self.is_being_profiled
        except ExitFrame:
            return self.popvalue()

    ...

    def jump_absolute(self, jumpto, _, ec=None):
        pypyjitdriver.can_enter_jit(
            frame=self, ec=ec, next_instr=jumpto,
            pycode=self.getcode(),
            is_being_profiled=self.is_being_profiled)
        return jumpto
```

Figure 43: The main Python bytecode loop and the `JUMP_ABSOLUTE` Bytecode

10.2 OBJECT IMPLEMENTATION AND OPTIMIZATIONS

The object space contains the implementation of all the kinds of objects built into the Python language. It is a library containing only types that are in some way directly integrated into the language (for example by having special syntactic support).

The built-in Python types that are implemented in the object space are:

- Numerical types: `int`, `long`, `float`, `complex`
- String types: `str` and `unicode`
- Container types: `list`, `tuple`, `dict`, `set`, and `frozenset` (an immutable set type)
- Types relevant to the class system: `object` and `type`

All these types have rich operations on them built into the language and implemented in the object space. Particularly the container types each have a large number of methods for manipulating them.[3] Python's object system supports multiple inheritance and meta-classes, following the ObjVLisp model [Coi87].

All the primitive objects such as integers, floats, complex numbers and partially strings are already sufficiently optimized by the allocation removal techniques described in Chapter 6. Optimizations on container types have been described by Diekmann [Die12]. What remains are the important building blocks for larger programs, instances, classes, and modules. Those are optimized using more advanced variants of the techniques described in Chapter 5. In general the design goals of these optimizations were to make the common cases of non-reflective access as fast as possible. All additional flexibility (reflection, using dynamic access patterns, mutating the exposed reflective data structures) is allowed to cost additional memory and time. Of course the semantics of the Python language always has to be preserved. It is never allowed to have operations that give incorrect results or raise an exception even in obscure corner-cases.

10.2.1 *Optimizing instances*

Instances of user-defined classes are used quite often as soon as a Python program is larger than just a small script. At the same time, Python's object model is rather complex. Every instances can store several types of information, as described below.

[3] See for example the dictionary documentation: `http://docs.python.org/library/stdtypes.html#dict`

Every instance knows which class it belongs to. This information is accessible via the `__class__` attribute. It can also be changed to other (compatible enough) classes by writing to that attribute.

Every instance also stores an arbitrary number of attributes (also called instance variables). The instance variables used can vary per instance, which is not the case in most other class-based languages: traditionally (for example in Smalltalk [Gol83] or Java [Gos05]) the class describes the shape of its instances, which means that the set of admissible instance variable names is the same for all instances of a class.

In Python on the other hand, it is possible to add arbitrary attributes to an instance at any point. The instance behaves like a dictionary mapping attribute names (as strings) to the attribute values. In this way Python behaves more like a prototype-based language such as Self [US87] or JavaScript [ECM99]. The class is only there for sharing behaviour between instances.

In CPython instances are implemented by giving every instance a reference to a dictionary that stores all the attributes of the instance. This dictionary can be reached via the `__dict__` attribute. For additional complexity, the dictionary can also be *replaced* by writing to that attribute. This implementation decision is costly in terms of memory, as dictionaries are not small data structures, and seems to defeat many reasonable optimisations.

The implementation of instances in PyPy is optimized in such a way that the direct access of instance attributes is as fast as possible. To reach this goal, the attributes of instances are represented using maps as described in Section 5.3. All the features of Python's object model need to work within that conceptual framework.

A very important optimization beyond the code in Section 5.3 is that in practice the shape of an instance is correlated with its class. Therefore it does not makes sense to let them vary independently. In PyPy's Python interpreter the class of an instance is therefore stored on its map. In that way, when promoting the map of an instance, its class is also known directly. The approach needs one less promotion as the code in Figure 18. Since promotions turn into guards in a trace, this produces smaller traces. It also has the benefit of making objects one word smaller.

Although performance is PyPy's most obvious goal, it also attempts to save memory when that is not in direct conflict with performance. One example of this is PyPy's compact representation of instances. Observance of real systems showed that most objects have five or fewer slots. PyPy therefore preallocates space for five slots, freeing it from the need to allocate an arbitrarily sized list to store slots in most cases (which, when all its parts are taken account of, is up to 40% more memory needed to store five slots). Only when more

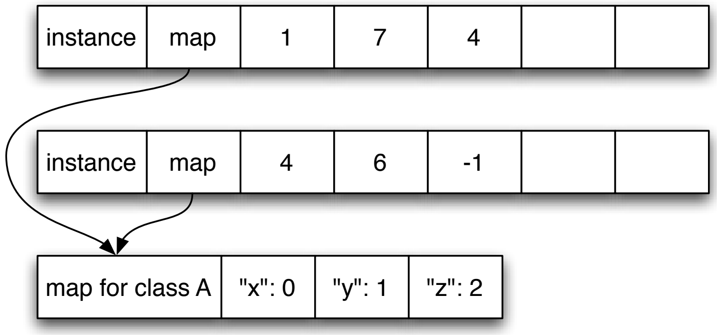


Figure 44: Two instances of class A sharing the same map

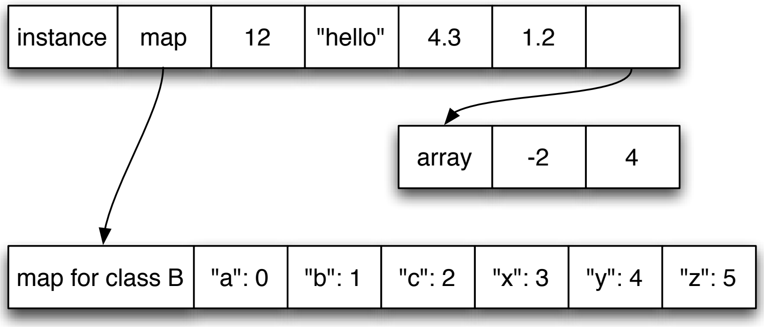


Figure 45: An instance of class B with six attributes

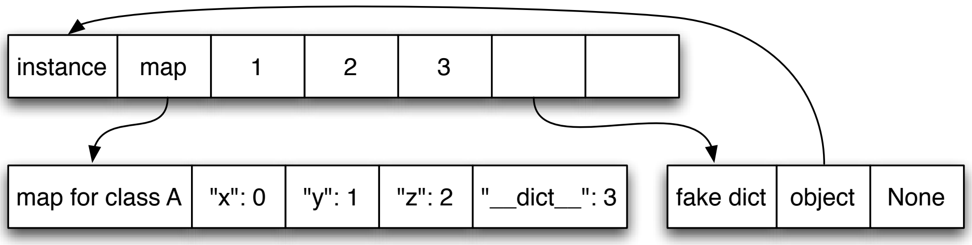


Figure 46: An instance implemented with a map, and its dictionary

than five slots are assigned to an instance is an arbitrarily sized list created and referenced from the object.

Figure 44 shows PyPy's layout scheme for two instances of class A, each instance using the same slot names. Since the instances have only three slots, the content of the slots can be stored in the preallocated slots. Figure 45 shows an instance with six slots. Two of the fields have to be stored in an extra array allocated for that use. Note how the last preallocated slot of the instance is used for that indirection.

When the .__dict__ attribute of an instance is accessed a simple solution would be to stop using a map to implement the instance storage and switch over to storing everything in a dictionary again. Doing so would mean that any reflective access of the dictionary would substantially slow down subsequent use of that instance. Since the dictionary is mostly used for reading and writing slots, this would slow down many real programs. Therefore, in PyPy, requesting an instance's dictionary returns a fake dictionary. This is indistinguishable from a real dictionary, and transparently redirects all reads and write to keys and values to the underlying instance.[4] Using this technique, performance for normal accesses after simple reflective use of the object remains as fast as the standard case.

Figure 46 shows an instance that has its data stored with a map, together with the fake dictionary that redirects all accesses back to the instance. Note that the instance needs to keep a reference to the dictionary once it has been requested to ensure that the expected object identity invariants are maintained.

However, when the programmer uses more of Python's dynamic features – in particular, writing a new dictionary to the __dict__ slot – even this tactic is no longer viable. In such cases, PyPy stops using maps for the instance and stores its instances in a real dictionary (as

[4] A proper solution to implement this sort of reflective access would be to use mirrors [BU04]. This would make it necessary to change the semantics of the Python language. Indeed, the .__dict__ attribute can already be seen as a mirror on the attributes of an instances. However, the dictionary gives stronger guarantees than mirrors, because it is supposed to stay the same object when repeatedly accessing it.

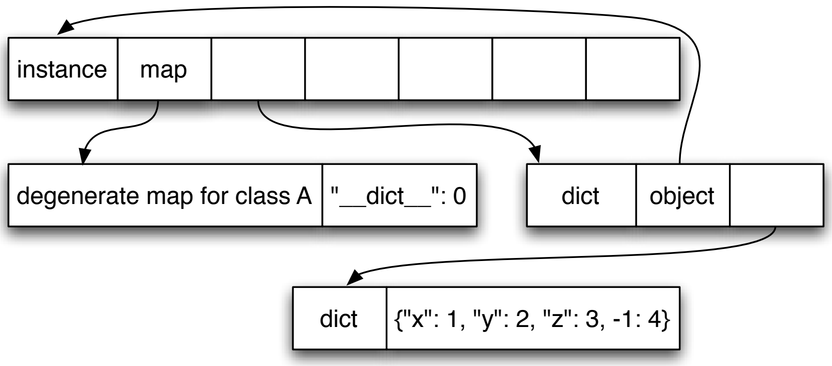


Figure 47: An instance that has its attributes stored in a dictionary

shown in Figure 47). Fortunately such uses are rare, so few programs suffer the consequent slowdowns.

10.2.2 *Optimizing classes*

The goal when optimizing classes was to make a non-reflective method lookup as fast as possible. Python's class model supports multiple inheritance. The C3 algorithm [BCH$^+$96] is used to linearize the parent classes into a method-resolution order for every class. Therefore looking up a method in a class needs to consider all the classes in the whole method resolution order. This makes the versioning of classes as described in Section 5.3 more complex because classes cannot be versioned in isolation. If class `A` is changed, its version changes. At the same time, the version of every class that has `A` in its method resolution order needs to be changed as well. To make this possible, a class keeps a list of weak references to its subclasses.[5] This makes class changes expensive, but they should be rare. The advantage of this approach is that a lookup in a class hierarchy is replaced by the checking of one class's version. After the version check, the correct result is already known. Thus every lookup has the same cost, no matter how deep the class hierarchy is.[6]

One problem with this approach is that while most classes never change, there are some classes that have attributes that change often. A common example is that of storing a counter on the class to give every instance a unique number. In that case, every time the counter is increased, the class version (and that of all subclasses) changes, which means a lot of guards that check for the previous version start

[5] The reference needs to be weak to make it possible to garbage collect classes.

[6] There is a further optimization in place in RPython to remove this check entirely. The version field of the class object is declared to be "quasi-immutable". This means that it is supposed to change rarely. Reading such a field out of a class produces no operation in the trace at all, instead a dependency of the trace on the field's value is recorded. When that value changes the trace is invalidated.

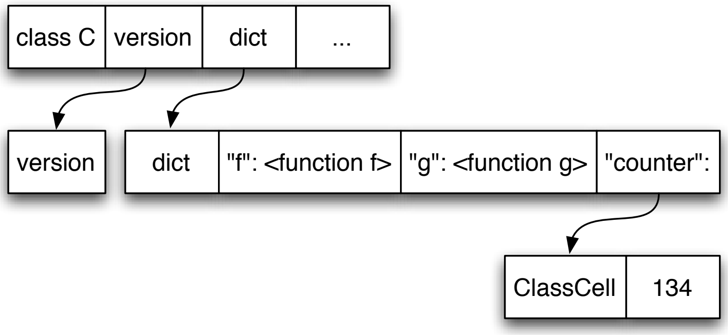


Figure 48: Class C with two methods and a counter

failing and new traces would have to be generated. Clearly this is not desirable.

Therefore an extension was made to address class changes. When a class attribute is changed for the first time to another value, an extra level of indirection is introduced. That means the dictionary of the class doesn't store the value of the attribute directly, but stores a reference to a small intermediate object that contains the real value. When the value of the class attribute is changed subsequently, only the content of that intermediate object is changed, not the class's attribute. Thus for such changes the type version of the class does not need to be changed, making writing to it efficient. After the first time, writing to such a field thus causes relatively little slowdown, while reading from it needs an extra memory read (including when accessed via subclasses). While slightly less efficient, this seems like a reasonable balance between fast general performance and reasonable performance in the rarer case.

Figure 48 shows a class with two methods `f` and `g` and a `counter` field. The counter is stored via an indirection to a `ClassCell`, meaning that changing it does not update the version of the class. On reading the `counter` attribute, an extra pointer dereference is needed.

This approach is similar to Smalltalk's mechanism for handling global variables [Gol83, p. 599]. A global variable[7] is a reference to an *association* object, which corresponds to PyPy's class cells. To read it, the value of the association is read. To write to the global variable, the value of the association object is set. In Smalltalk, this indirection is always used, not just for commonly changed variables. More generally, class versions can be related to the invalidation of method caches when a new method is compiled in some Smalltalk systems [DS84].

[7] The same is true for class variables and pool variables.

10.2.3 *Optimizing modules*

Modules are conceptually similar to classes, with both providing namespaces for storing functions/methods and values. However, modules are simpler in that there can be no inheritance between them. Similar to classes, a module's content typically stays the same after initialization. However, there are also cases in which a module contains a few fields that are changed all the time. This is the case when code is executed on the global level or when a variable is declared to be `global` within a function's scope. Thus a similar solution as for classes applies: Every module gets a version. When the module is mutated, the version is changed. When the same name in the module is mutated a second time, an indirection from the module dictionary to a cell is introduced. The lookups for this name are slightly slower, but changing that same name again in the future will be fast.

10.3 BENCHMARKS

This section will evaluate the performance effect of meta-tracing and its techniques as described in the first part of this thesis. To achieve that, the benchmarks that were already used in Chapter 6 are used again. They are described in Appendix A.[8]

To evaluate how successful the described techniques are, six versions of the PyPy Python interpreter were built and are compared against each other. The starting point is the purely interpretative version of the Python interpreter. Then step by step various features are enabled. The versions are described in Figure 49. For all the versions the full optimizations of the RPython tracing system are enabled, including that of Chapter 6. The first four of these versions can be used to gauge the effectiveness of basic meta-tracing, the last two of run-time feedback and object optimizations.

As a baseline, CPython[9] is used, the standard Python implementation in C. It uses a bytecode-based interpreter. Furthermore we compared against Psyco [Rig04], a (hand-written) extension module to CPython which is a just-in-time compiler that produces machine code at run-time. It is not based on traces and it also does not preserve the language semantics fully, for example it disables the use of the debugger.

All the results are summarized in Figure 50. It shows the geometric mean over all benchmarks results as a factor over CPython. Higher results are better.

Figure 51 shows the detailed results and comparison between CPython, PyPy-interp and PyPy-full which shows the full range of

[8] All the benchmarks and the scripts to run them can be found at: `http://cfbolz.de/phdthesis/`

[9] `http://python.org`

PYPY-INTERP is a pure interpreter.

PYPY-TRACING enables classical tracing of the interpreter, not meta-tracing. The bytecode loop is not unrolled during tracing. It corresponds to Figure 7 in Chapter 4.

PYPY-META enables meta-tracing. Like in Figure 9, the bytecode loop is unrolled to correspond to user loops. However, the operations on the bytecode and program counter are not folded away.

PYPY-FOLD is the same as PyPy-meta, but also folds operations on bytecode and program counter. It corresponds to Figure 10. Then hints needed to achieve this are shown in Figure 43.

PYPY-FRAME adds the frame optimization that are briefly described in Section 6.5.

PYPY-FULL adds promote and elidable hints of Chapter 5 as described in Section 10.2 of this chapter. This is the "normal" PyPy Python interpreter that is shipped in releases. It contains all operations known to be effective.

Figure 49: The versions of PyPy used in benchmarks

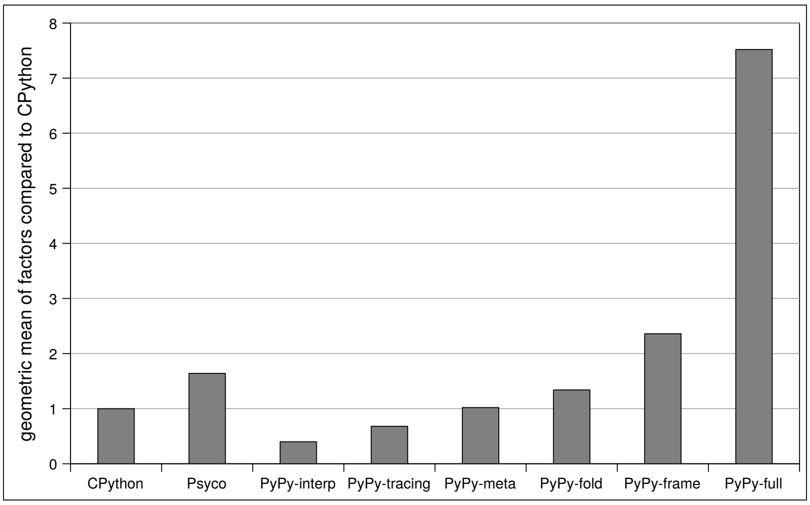


Figure 50: Overview of how much faster than CPython the variants are (higher is better)

	CPython [ms]	PyPy-interp [ms]	PyPy-full [ms]
chaos	364.68 ± 4.56	853.35 ± 1.68	21.16 ± 1.20
	2.34 ×	1.00 ×	40.34 ×
crypto_pyaes	2098.80 ± 3.69	4633.95 ± 20.47	119.24 ± 19.45
	2.21 ×	1.00 ×	38.86 ×
django	753.89 ± 4.03	1595.27 ± 9.37	74.87 ± 9.56
	2.12 ×	1.00 ×	21.31 ×
go	724.85 ± 5.80	2998.54 ± 14.23	171.58 ± 74.50
	4.14 ×	1.00 ×	17.48 ×
pyflate-fast	2418.29 ± 20.28	5541.48 ± 24.63	904.71 ± 17.09
	2.29 ×	1.00 ×	6.13 ×
raytrace-simple	1910.54 ± 9.06	3789.66 ± 8.27	41.35 ± 5.69
	1.98 ×	1.00 ×	91.64 ×
richards	256.23 ± 2.51	562.26 ± 3.02	6.93 ± 1.99
	2.19 ×	1.00 ×	81.11 ×
spambayes	242.61 ± 2.08	566.55 ± 13.00	131.06 ± 84.99
	2.34 ×	1.00 ×	4.32 ×
sympy_expand	1133.99 ± 11.55	3281.73 ± 31.62	948.74 ± 42.38
	2.89 ×	1.00 ×	3.46 ×
telco	957.50 ± 8.60	2821.28 ± 4.69	70.70 ± 9.62
	2.95 ×	1.00 ×	39.90 ×
twisted_names	7.88 ± 0.03	18.46 ± 0.22	3.18 ± 0.13
	2.34 ×	1.00 ×	5.80 ×
geom. mean	2.47	1.00	18.60

Figure 51: CPython, PyPy interpreter and full JIT

performance. PyPy-interp is about 2 to 4.4 times slower than CPython. PyPy-full achieves speedups between 3.7 and almost 90 times faster than the interpreted version.

Figure 52 shows the comparison of the full JIT against Psyco. PyPy-full is quite competitive with Psyco, there is only one benchmark, crypto_pyaes, where Psyco is about two times faster than PyPy. Apart from this benchmark, PyPy-full is often significantly faster than Psyco.

In the following subsections we will see from which optimizations these performance-improvements come from.

	Psyco [ms]	PyPy-full [ms]
chaos	147.47 ± 1.13	21.16 ± 1.20
	1.00 ×	6.97 ×
crypto_pyaes	67.66 ± 4.73	119.24 ± 19.45
	1.00 ×	0.57 ×
django	827.37 ± 2.32	74.87 ± 9.56
	1.00 ×	11.05 ×
go	423.63 ± 5.03	171.58 ± 74.50
	1.00 ×	2.47 ×
pyflate-fast	1157.32 ± 9.84	904.71 ± 17.09
	1.00 ×	1.28 ×
raytrace-simple	981.73 ± 2.88	41.35 ± 5.69
	1.00 ×	23.74 ×
richards	63.52 ± 1.78	6.93 ± 1.99
	1.00 ×	9.16 ×
spambayes	249.32 ± 2.56	131.06 ± 84.99
	1.00 ×	1.90 ×
sympy_expand	12691.42 ± 7934.46	948.74 ± 42.38
	1.00 ×	13.38 ×
telco	676.50 ± 10.46	70.70 ± 9.62
	1.00 ×	9.57 ×
twisted_names	8.00 ± 0.03	3.18 ± 0.13
	1.00 ×	2.52 ×
geom. mean	1.00	4.58

Figure 52: Psyco and PyPy-Full

10.3.1 *The effect of (meta-)tracing*

To measure the effects of the meta-tracer (Chapter 4) various versions of the PyPy Python interpreter are compared with each other, employing different ways to do tracing. The results can be seen in Figure 53. Surprisingly, classical tracing gives performance benefits for some of the benchmarks, between 26% and more than 3 times faster. This is opposed to the results gotten for the small stack-based interpreter in Section 4.4. The reason is that Python's bytecodes contain a lot of dynamic dispatches so even just picking common paths in them is sometimes useful. Two benchmarks crashed with PyPy-trace. Because of the general limitations of PyPy-trace this was not investigated further.

Further speedups can be achieved by enabling meta-tracing, unrolling the bytecode loop until the trace corresponds to the loops on the Python level. Together with the folding of the operations on bytecode on program counter the speedup over PyPy-interp is between 50% and a factor of almost 9. This corresponds to getting rid of all the bytecode dispatch overhead and doing optimizations on primitive data types, such as integers and floats, with the help of the allocation removal optimization.

10.3.2 *The effect of run-time feedback*

To improve the performance even more, optimizations of objects are needed. To that end, the optimizations of instances, classes, and modules as described in Section 10.2 are enabled. The results are seen in Figure 54. The figure shows that the object optimizations give a further speedup over PyPy-fold of 50% up to a factor of more than 50. Some of this effect is due to optimizing reified frames, which on its own already gives an improvement of 18% up to a factor of 3.

10.3.3 *Warmup times*

To better understand the warmup times that MetaJIT needs to reach full speed, the cold run times of the benchmarks were measured. They are taken by measuring the time of the first iteration of the benchmark after starting the VM. The time for the first iteration typically includes the tracing and compilation time. The numbers can be seen in Figure 55. For MetaJIT, the results are mixed. While in some benchmarks the JIT seems to help even within the first iteration, in others the first iteration is slower than CPython. Psyco's compilation overhead and warmup times are a lot lower than that of MetaJIT: for Psyco, all benchmarks except chaos and crypto_pyaes have almost their full speed in the first iteration. However, for MetaJIT this is only true for twisted_names.

	PyPy-interp [ms]	PyPy-trace [ms]	PyPy-meta [ms]	PyPy-fold [ms]
chaos	853.35 ± 1.68	448.94 ± 10.71	408.23 ± 11.29	225.91 ± 3.05
	1.00 ×	1.90 ×	2.09 ×	3.78 ×
crypto_pyaes	4633.95 ± 20.47	2528.03 ± 29.38	881.01 ± 21.44	521.93 ± 23.65
	1.00 ×	1.83 ×	5.26 ×	8.88 ×
django	1595.27 ± 9.37	1591.44 ± 21.38	488.53 ± 23.40	417.59 ± 11.90
	1.00 ×	1.00 ×	3.27 ×	3.82 ×
go	2998.54 ± 14.23	908.12 ± 19.00	713.93 ± 56.86	487.94 ± 32.47
	1.00 ×	3.30 ×	4.20 ×	6.15 ×
pyflate-fast	5541.48 ± 24.63	3985.06 ± 53.54	2500.42 ± 34.04	2198.21 ± 28.73
	1.00 ×	1.39 ×	2.22 ×	2.52 ×
raytrace-simple	3789.66 ± 8.27	2161.27 ± 232.26	1673.84 ± 15.72	1422.96 ± 14.45
	1.00 ×	1.75 ×	2.26 ×	2.66 ×
richards	562.26 ± 3.02	446.99 ± 8.45	419.28 ± 5.25	367.68 ± 3.83
	1.00 ×	1.26 ×	1.34 ×	1.53 ×
spambayes	566.55 ± 13.00	300.31 ± 32.35	243.10 ± 106.60	210.25 ± 101.02
	1.00 ×	1.89 ×	2.33 ×	2.69 ×
sympy_expand	3281.73 ± 31.62	2206.15 ± 81.45	1626.77 ± 65.49	1429.57 ± 71.01
	1.00 ×	1.49 ×	2.02 ×	2.30 ×
telco	2821.28 ± 4.69	—	1070.77 ± 10.41	696.14 ± 11.30
	1.00 ×		2.63 ×	4.05 ×
twisted_names	18.46 ± 0.22	—	8.47 ± 0.32	7.12 ± 0.21
	1.00 ×		2.18 ×	2.59 ×
geom. mean	1.00	1.67	2.53	3.31

Figure 53: Comparing various ways to do tracing

	PyPy-fold [ms]	PyPy-frame [ms]	PyPy-full [ms]
chaos	225.91 ± 3.05	92.61 ± 2.36	21.16 ± 1.20
	1.00 ×	2.44 ×	10.68 ×
crypto_pyaes	521.93 ± 23.65	172.56 ± 20.43	119.24 ± 19.45
	1.00 ×	3.02 ×	4.38 ×
django	417.59 ± 11.90	250.63 ± 9.79	74.87 ± 9.56
	1.00 ×	1.67 ×	5.58 ×
go	487.94 ± 32.47	302.99 ± 50.39	171.58 ± 74.50
	1.00 ×	1.61 ×	2.84 ×
pyflate-fast	2198.21 ± 28.73	1411.50 ± 23.37	904.71 ± 17.09
	1.00 ×	1.56 ×	2.43 ×
raytrace-simple	1422.96 ± 14.45	523.86 ± 4.20	41.35 ± 5.69
	1.00 ×	2.72 ×	34.41 ×
richards	367.68 ± 3.83	245.30 ± 2.25	6.93 ± 1.99
	1.00 ×	1.50 ×	53.04 ×
spambayes	210.25 ± 101.02	177.47 ± 93.53	131.06 ± 84.99
	1.00 ×	1.18 ×	1.60 ×
sympy_expand	1429.57 ± 71.01	1172.30 ± 71.42	948.74 ± 42.38
	1.00 ×	1.22 ×	1.51 ×
telco	696.14 ± 11.30	339.42 ± 8.97	70.70 ± 9.62
	1.00 ×	2.05 ×	9.85 ×
twisted_names	7.12 ± 0.21	5.30 ± 0.18	3.18 ± 0.13
	1.00 ×	1.34 ×	2.24 ×
geom. mean	1.00	1.76	5.62

Figure 54: Enabling frame and object optimizations

	CPython [ms]	Psyco [ms]	Psyco [ms]	PyPy-full [ms]	PyPy-full [ms]
			cold		cold
chaos	364.68 ± 4.56	147.47 ± 1.13	149.81	21.16 ± 1.20	292.97
	1.00 ×	2.47 ×	2.45 ×	17.24 ×	1.25 ×
crypto_pyaes	2098.80 ± 3.69	67.66 ± 4.73	68.49	119.24 ± 19.45	345.08
	1.00 ×	31.02 ×	30.26 ×	17.60 ×	6.01 ×
django	753.89 ± 4.03	827.37 ± 2.32	826.51	74.87 ± 9.56	306.72
	1.00 ×	0.91 ×	0.91 ×	10.07 ×	2.46 ×
go	724.85 ± 5.80	423.63 ± 5.03	470.57	171.58 ± 74.50	2766.88
	1.00 ×	1.71 ×	1.54 ×	4.22 ×	0.26 ×
pyflate-fast	2418.29 ± 20.28	1157.32 ± 9.84	1143.60	904.71 ± 17.09	1540.96
	1.00 ×	2.09 ×	2.11 ×	2.67 ×	1.57 ×
raytrace-simple	1910.54 ± 9.06	981.73 ± 2.88	991.28	41.35 ± 5.69	1285.51
	1.00 ×	1.95 ×	1.92 ×	46.20 ×	1.48 ×
richards	256.23 ± 2.51	63.52 ± 1.78	67.40	6.93 ± 1.99	449.73
	1.00 ×	4.03 ×	3.79 ×	36.96 ×	0.57 ×
spambayes	242.61 ± 2.08	249.32 ± 2.56	256.57	131.06 ± 84.99	563.90
	1.00 ×	0.97 ×	0.96 ×	1.85 ×	0.44 ×
sympy_expand	1133.99 ± 11.55	12691.42 ± 7934.46	1435.63	948.74 ± 42.38	2598.10
	1.00 ×	0.09 ×	0.79 ×	1.20 ×	0.44 ×
telco	957.50 ± 8.60	676.50 ± 10.46	680.00	70.70 ± 9.62	1232.08
	1.00 ×	1.42 ×	1.41 ×	13.54 ×	0.78 ×
twisted_names	7.88 ± 0.03	8.00 ± 0.03	7.97	3.18 ± 0.13	3.18
	1.00 ×	0.98 ×	1.00 ×	2.48 ×	2.50 ×
geom. mean	1.00	1.64	1.97	7.52	1.08

Figure 55: Cold run times

	Python [MiB]	Psyco [MiB]	PyPy-interp [MiB]	PyPy-full [MiB]
chaos	7.02	12.70	37.50	38.78
crypto_pyaes	2.93	8.05	41.08	45.03
django	9.98	21.70	50.48	59.54
go	6.78	15.50	38.21	44.63
pyflate-fast	11.31	17.87	49.39	62.32
raytrace-simple	2.59	8.39	38.27	38.63
richards	2.62	8.00	23.06	33.86
spambayes	6.27	21.67	44.33	47.56
sympy_expand	14.68	385.70	50.42	52.94
telco	2.85	10.64	38.58	48.40
twisted_names	6.98	15.76	39.97	52.75

Figure 56: Memory usage

10.3.4 *Memory usage*

To understand the effects on memory usage that MetaJIT has on PyPy's Python interpreter the memory usage was tracked during a run of every benchmark. The results can be seen in Figure 56. They show that even PyPy without MetaJIT has a large memory overhead over CPython, of up to more than 10 times larger for crypto_pyaes. The JIT can add another factor of 2, for example for richards. The memory overhead of Psyco over CPython is comparable to that of PyPy-full over PyPy-interp, except for sympy_expand, where Psyco seems to generate a huge amount of machine code (sympy_expand is also the only benchmark where using Psyco makes the program slower). The memory overhead of MetaJIT is justified by the speedup it gives but work should be done to reduce it further. However, the memory overhead of PyPy-interp itself compared to other languages seems unacceptably large in some cases.

10.3.5 *Comparison with other languages*

To understand how the performance of the Python interpreter compares with other JIT implementations this subsection presents the results of a different benchmark set, that of the Computer Language Benchmarks Game.[10] It contains implementations of a number of benchmarks in various languages. The benchmarks mostly do string and numeric computations and are small (about 20-300 lines of code). The benchmarks implementations in the different languages perform

[10] http://shootout.alioth.debian.org/

the same function, but do not have to use the same implementation approach. Some of the benchmarks were slightly modified to make the results comparable. All the CPython-specific optimizations from the Python versions were removed. Furthermore, all Java versions were modified to use only one thread.

The JITs chosen to compare against are V8,[11] Google's JavaScript VM used in the Chrome browser; LuaJIT,[12] an open source Lua tracing JIT by Mike Pall and Hotspot[13], the Oracle Java Virtual Machine.

The results of the comparison can be seen in Figure 57. All shootout benchmarks were run 30 times, starting a new process for every run. Every benchmark is run with two input sizes, one that finishes quickly and one that runs longer. This is done to estimate the effects of the warmup times. Hotspot is the fastest VM, followed by LuaJIT, V8, and PyPy. On average, LuaJIT is almost two times faster than PyPy, while V8 is still 60% faster. In individual benchmarks PyPy is even a lot slower. Hotspot is rather slow for regexdna, which exercises the regular expression engines of the VMs. Both V8 and PyPy have JITs for theirs, which explains the good results.

The results still show that a meta-tracing can achieve results in the same ballpark as manually tuned virtual machines. A certain slowdown compared to a hand-written VM is to be expected, due to the generality of the meta-tracing approach.

The benchmarks don't show a clear preference when comparing tracing to method-based JIT compilers. LuaJIT and PyPy use tracing, while Hotspot and V8 are method-based JITs. While Hotspot is the fastest JIT, it is for a statically typed language. LuaJIT, a tracing JIT, is the fastest VM for a dynamic language of the ones benchmarked.

10.4 conclusion

In this chapter PyPy's Python interpreter and the object model optimizations of it were presented. With these optimizations, instances, classes, and modules perform well in the Python VM. Instances are stored almost as compactly in memory as in, for example, Smalltalk, with equally efficient attribute access times, despite keeping enough information to implement the more dynamic behaviour that Python supports. Classes are optimized for the common case (an inheritance hierarchy where methods in classes are not changed). Module globals have most of their lookup overhead removed. The optimizations speed up the execution of the presented benchmark programs, sometimes significantly so, up to a factor of 80 times. More importantly, the Python interpreter with MetaJIT also beats Psyco in many benchmarks, showing that MetaJIT can compete with the only hand-written

[11] http://code.google.com/p/v8/

[12] http://luajit.org

[13] http://openjdk.java.net/groups/hotspot/

	HotSpot	LuaJIT	V8	PyPy
binarytrees(14)	0.27 ± 0.01	1.14 ± 0.04	0.21 ± 0.00	2.22 ± 0.03
	8.29 ×	1.95 ×	10.58 ×	1.00 ×
binarytrees(19)	5.62 ± 0.15	52.43 ± 0.37	13.23 ± 0.14	53.86 ± 0.37
	9.58 ×	1.03 ×	4.07 ×	1.00 ×
fannkuchredux(10)	0.58 ± 0.01	0.53 ± 0.00	0.47 ± 0.00	3.76 ± 0.08
	6.50 ×	7.09 ×	8.00 ×	1.00 ×
fannkuchredux(11)	6.29 ± 0.16	6.91 ± 0.01	5.98 ± 0.15	45.15 ± 1.03
	7.18 ×	6.53 ×	7.55 ×	1.00 ×
fasta(5000000)	1.15 ± 0.18	2.42 ± 0.06	5.30 ± 0.55	3.04 ± 0.19
	2.64 ×	1.26 ×	0.57 ×	1.00 ×
fasta(50000000)	10.59 ± 1.74	24.12 ± 0.15	51.49 ± 1.85	28.43 ± 2.39
	2.69 ×	1.18 ×	0.55 ×	1.00 ×
knucleotide(1000000)	2.40 ± 0.02	2.19 ± 0.02	12.86 ± 0.20	5.41 ± 0.07
	2.25 ×	2.47 ×	0.42 ×	1.00 ×
knucleotide(10000000)	21.11 ± 0.29	18.69 ± 0.21	123.80 ± 2.28	52.97 ± 0.81
	2.51 ×	2.83 ×	0.43 ×	1.00 ×
mandelbrot(500)	0.12 ± 0.00	0.04 ± 0.00	0.35 ± 0.02	0.34 ± 0.02
	2.83 ×	8.50 ×	0.97 ×	1.00 ×
mandelbrot(5000)	2.75 ± 0.01	4.33 ± 0.01	36.22 ± 1.10	27.48 ± 2.72
	10.00 ×	6.35 ×	0.76 ×	1.00 ×
nbody(2500000)	0.83 ± 0.05	1.45 ± 0.05	1.68 ± 0.14	4.97 ± 0.08
	6.02 ×	3.44 ×	2.96 ×	1.00 ×
nbody(25000000)	7.28 ± 0.01	14.41 ± 0.29	17.31 ± 0.41	49.10 ± 1.01
	6.74 ×	3.41 ×	2.84 ×	1.00 ×
regexdna(1000000)	5.94 ± 0.54	8.92 ± 0.06	0.64 ± 0.01	1.95 ± 0.02
	0.33 ×	0.22 ×	3.06 ×	1.00 ×
regexdna(10000000)	58.52 ± 1.34	104.62 ± 0.77	6.31 ± 0.11	23.50 ± 0.14
	0.40 ×	0.22 ×	3.72 ×	1.00 ×
revcomp(1000000)	0.26 ± 0.01	0.38 ± 0.01	0.53 ± 0.01	0.56 ± 0.01
	2.15 ×	1.50 ×	1.07 ×	1.00 ×
revcomp(10000000)	1.66 ± 0.04	3.74 ± 0.06	5.00 ± 0.13	5.55 ± 0.10
	3.34 ×	1.48 ×	1.11 ×	1.00 ×
spectralnorm(500)	0.20 ± 0.00	0.08 ± 0.00	0.09 ± 0.01	0.15 ± 0.00
	0.75 ×	1.88 ×	1.65 ×	1.00 ×
spectralnorm(5000)	8.52 ± 0.01	8.40 ± 0.01	8.59 ± 0.01	8.48 ± 0.01
	0.99 ×	1.01 ×	0.99 ×	1.00 ×
geom. mean	2.81	1.92	1.73	1.00

Figure 57: Comparison with other JITs

JIT compiler for Python. The comparison with hand-written JITs for other languages shows that MetaJIT does not consistently reach the same performance levels yet. PyPy's performance with MetaJIT is up to 11 times slower than that of V8 and LuaJIT, but sometimes also surpasses them.

The optimizations presented in this chapter give examples of how RPython VM authors need to consider which usage patterns are the most frequent and therefore should be made as efficient as possible. They must then (re)arrange the interpreter and data structures so that, in conjunction with the trace optimizer, small traces with little code and few guards are produced. There is, of course, a tension between making common cases fast while not making less common cases unusably slow. VM authors need to understand their languages and intended use cases well. However, as often the case with performance issues, it is not realistic to do so purely intellectually: real programs must be analysed to determine which cases need to be focused on. Different benchmarks (synthetic or not) can change the perception of the most important areas substantially, and must be carefully chosen.

As this suggests, it is impossible to design a perfectly optimal interpreter up-front. Analyzing the traces produced by real programs often shows places where they can be improved. These places can be surprising and are not always obvious in advance. Each pinch-point identified in the interpreter can be addressed either by adding hints for MetaJIT, or by rewriting the interpreter. Doing so is often not a trivial task, particularly as the interpreter becomes larger and more complex. It requires careful thought about the goals of the optimization, the trade-offs involved (including to code readability), and how to reach these goals.

Thus, while naively applying a meta-tracing JIT is easy, achieving very good performance using it is no small task. That said, nearly all optimizations are understandable at the level of the interpreter itself: one need never look within the JIT compiler itself. The interpreter thus still expresses the language semantics correctly (albeit somewhat strangely when optimizations require changing its structure), and many optimizations improve the performance of the language interpreter as well as helping the meta-tracing JIT. For example maps are a memory optimization if only an interpreter is used, and class versions can be used for a method cache within a purely interpreted system.

While promoting and eliding are direct features of MetaJIT, class versions are an idiom of use. This can understate their importance: they are a powerful way to constant-fold arbitrary functions on large data structures. The versions need to be updated carefully every time the result of a function on a structure can change. Therefore this technique is only applicable on data structures which change slowly or which (as PyPy's approach to optimizing classes in Section 10.2.2 showed) can be made to change slowly.

I believe that the manual rewriting of parts of the interpreter is a key part of the meta-tracing approach. Many of the optimizations rely on in-depth knowledge of the language the interpreter implements. The rewrites expose not only properties of the language semantics (which are already present in the interpreter) but also expectations about patterns of language use (which are not). While an "optimally smart" meta-tracing compiler might be able to deduce some optimizations, most rely on the wider context which only a human can bring to the table.

11 CASE STUDY: PROLOG

The two case studies so far were for a minimal example, a regular expression matcher, and a full-featured big language, Python. This chapter examines how far the tracing concept can be pushed outside of the context that it was initially invented for. The concept of tracing JITs was invented for Java [GPF06], an object-oriented imperative language. Later it was extended for a number of dynamic imperative languages.

Prolog [ISO95, CR96] is a non-deterministic logical language with terms and logical variables as its main data structures. It is neither imperative nor object-oriented but relies on pattern matching and backtracking to implement its control flow. It is also dynamically typed and allows run-time introspection and changes to the currently running program.

The chapter presents Pyrolog, an interpreter for Prolog written in RPython, which can be compiled to C and which is traced at runtime using MetaJIT. The performance results are surprisingly good. It is faster than state-of-the-art Prolog VMs on specific benchmarks. This shows the potential of the JIT approach for Prolog systems, and that future Prolog implementations should consider integrating a JIT compiler.

Section 11.1 discusses typical approaches to Prolog implementation. This is contrasted with Pyrolog, the details of which, its object model and continuation-based execution, are described in Section 11.2. How MetaJIT is applied to the Prolog interpreter is presented in Section 11.3, which also discusses how the allocation removal optimization of Chapter 6 applies to Pyrolog. Section 11.4 presents and analyzes some benchmarks measuring performance and memory consumption and compares it to two other Prolog implementations.

11.1 PROLOG IMPLEMENTATIONS

Most high-performance implementations of the Prolog language in common use today are implemented using an extension of the Warren Abstract Machine (WAM) [War83, AK91]. The WAM is a very successful instruction set that made high-speed Prolog execution possible [vR94]. However, it is also a very low-level instruction set that is predominantly useful when implementing in a low-level language operating close to the machine level. Apart from WAM-based approaches, there are some Prolog implementations written in object-oriented high-level languages, such as Java or the .NET VM [PBOR08, Coo04]. These often have flexible and extensible architectures, and integrate well with their host virtual machine, but are typically significantly slower than low-level VMs.

11.2 STRUCTURE OF THE INTERPRETER

The goal in implementing the Prolog interpreter in RPython was to have a simple, high-level object oriented implementation of Prolog. The semantics of Prolog should be mirrored closely by the structure of the interpreter. High-level optimizations should be incorporated into the interpreter, but the mapping to machine code and the handling of low-level details is left to MetaJIT.

The resulting interpreter fulfills many of these goals. It uses a simple structure copying approach, has a straightforward data model (Section 11.2.1) and uses continuation objects for the interpretation core (Section 11.2.2). So far it does not contain many optimizations, for example there is no indexing implemented yet. In addition to the interpreter core, some built-ins are implemented (see Section 11.2.3).

The interpreter is about 12 500 lines of RPython code, of which 2000 lines are implementing built-ins and 6500 are tests. It can be translated to C using RPython's translation toolchain and MetaJIT can be used for it (see Section 11.3). When translating to C without MetaJIT, the translation toolchain generates about 550 000 lines of C code, the compiled binary is 1.7 MiB large. When inserting MetaJIT, about 1 450 000 lines of C code are generated (much of it support code for MetaJIT) resulting in a binary of 6.4 MiB. The translation time for producing an executable without a JIT is about six minutes. When MetaJIT is also translated the time grows to nearly 15 minutes.

11.2.1 *Data model of the interpreter*

To represent Prolog terms the interpreter uses a straightforward object-oriented design of the Prolog concepts. Prolog objects are modelled by instances of subclasses of the `PrologObject` base class. Simple non-variable terms are represented by their own class, such as `Atom`, `Number` and `Float` (which are just boxes around a string, an integer and a floating point number respectively). Logic variables are represented by instances of a class `Var`. This class has a `binding` attribute which is initialized to `None` to signify that the variable is not bound. If the variable gets bound, the `binding` attribute gets set to the bound value. Compound terms are represented by a class `Compound`, which has a string attribute specifying the functor and an array of more `PrologObjects`, for the arguments.

Unification is implemented in an object-oriented style: all `PrologObjects` have a `unify` method, which takes a second object as

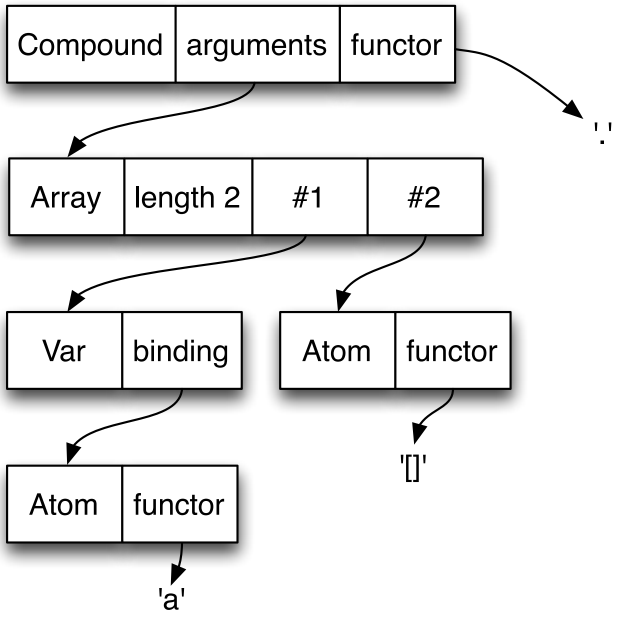


Figure 58: Representation of Prolog object `[X]` where `X` was bound to `a`

the argument, as well as a `Trail` object. The `unify` method calls itself recursively on the arguments of compound terms.

When a variable is bound, it needs to be trailed to be able to undo the binding should backtracking occur. This is done with the help of a `Trail` object. `Trail` objects are connected as a linked list, one `Trail` instance per choice point, each pointing to its next-oldest predecessor. If a variable is bound, it is stored into an array in the current, newest trail object, which holds all variables that will need to get their bindings undone when backtracking occurs.

As an optimization *variable shunting* [LH90] is implemented. If a variable is created and immediately bound, without a choice point being created in the meantime, it does not need to be trailed. On backtracking the variable will cease to exist anyway. This is achieved by making each `Var` objects point to the trail object in which it was created. If it is bound while this trail is still being used, the variable can be replaced by its binding value.

Most low-level Prolog engines use a tagged pointer representation [Gud93] for commonly-used compound terms, typically for cons cells (terms with functor `./2`). The RPython translation toolchain does not provide this level of control over low-level representation of objects. However, terms with common functors are still optimized, by representing them with their own class. This means that at least the array and the explicit reference to the functor can be saved. For an illustration of the concept, see Figure 58 for a representation of an unoptimized compound term and Figure 59 for the same term using the optimized classes.

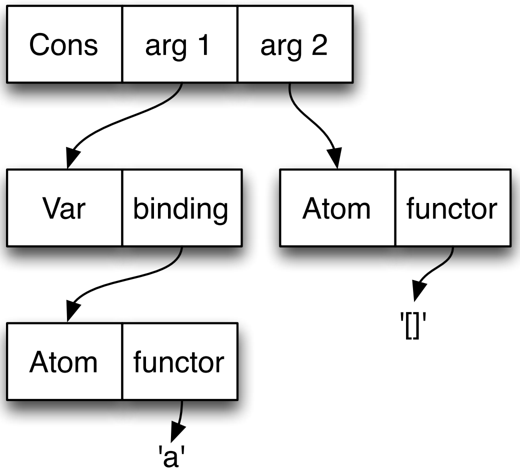


Figure 59: Representation using a specialized class for cons cells

11.2.2 *Continuation-based interpretation*

Compared to the Python object model, the Prolog object model is significantly simpler. Apart from variables, Prolog data structures are immutable and therefore easily optimized by the JIT. No complex rewriting of the interpreter was necessary. In contrast, to support the complex execution semantics of Prolog, much more work than for Python's execution semantics is needed. The execution of Prolog predicates does not follow the stack discipline of imperative languages. On the one hand, Prolog eliminates tail calls and models iteration as recursion. On the other hand, Prolog predicates can produce more than one solution by introducing choice points and backtracking. These properties make the state of the execution that the interpreter needs to keep track of quite complex. This makes it important to find a good abstraction for the interpreter state. The hope is that MetaJIT can reduce the cost of these abstractions and thereby yield an efficient implementation.

The abstraction chosen in Pyrolog to represent the execution state internally is that of *continuations*. Several Prolog systems have been based on continuations already [Tar92, Neu95, Lin94, Tar12]. The basic approach in the Pyrolog implementation is that all the state of the interpreter is encapsulated in two (possibly nested) continuation objects, a *success continuation* and a *failure continuation*. All continuations are instances of one of the subclasses of a `Continuation` class. A continuation thus contains state as well as behaviour.[1] The success continuation contains the still to be executed "rest of the program", the failure continuation contains the code that is to be executed if

[1] Indeed, on systems that implement Prolog on Scheme or Lisp [KC84], a continuation is usually just represented by a closure. This was not possible here, because RPython supports neither closures nor does it optimize tail calls.

```
def interpret(sc, fc, trail):
    while not sc.is_done():
        try:
            sc, fc, trail = sc.activate(fc, trail)
        except UnificationFailed:
            sc, fc, trail = fc.fail(trail)
```

Figure 60: The main interpreter loop

backtracking needs to happen. Calling a continuation typically consumes it, and potentially replaces the current continuations by new ones. Interpretation proceeds by calling the current success continuation until the computation is finished. If calling a continuation fails, the current failure continuation is called instead.

Whenever a non-deterministic choice is reached, the interpreter creates a new trail object. It then builds a failure continuation that backtracks to the previous state and then continues with the other option.

The main loop of the interpreter (in slightly simplified form) can be seen in Figure 60. The loop has three local variables: `sc` is the success continuation, `fc` the failure continuation and `trail` the current trail object. As long as there is still something to do, the `activate` method of the current continuation is called, which returns a new set of continuations. If activating the continuation fails, it will raise an `UnificationFailed` exception. If that happens, the failure continuation will get its `fail` method called.

The types of continuation used by the interpreter are (the continuations are given here as Prolog terms, in the actual implementation each sort of continuation is simply a class with the arguments of the terms as attributes):

- `call(Goal, Next)` which will call the goal, when activated
- `restore(Trail, Next, FailureContinuation)` which will backtrack the bindings done up to the point specified by `Trail`
- `apply(Rule, Goal, Next)` which applies a specific rule of a predicate to the goal
- `true` which signifies that the computation is finished

All continuations have a `Next` continuation, which will be called after the current continuation has been executed. In addition to the continuations listed, there are specific continuations used for built-ins that can have more than one solution (for example `arg/3`). Only `restore` continuations can appear as failure continuations, the other continuations can be success continuations only. The `FailureContinuation` argument of `restore` gives the failure continuation that will be used after the `restore` continuation has been activated. It contains the previous choice point.

Database: `f(a). f(b).`

Continuations:

1	`sc`	`call(f(X), true)`
	`fc`	`true`
	`trail`	`<trail1>`
2	`sc`	`apply(<f rule 1>, f(X), true)`
	`fc`	`restore(<trail1>, apply(<f rule 2>, f(X), true), true)`
	`trail`	`<trail2>`
3	`sc`	`true`
	`fc`	`restore(<trail1>, apply(<f rule 2>, f(X), true), true)`
	`trail`	`<trail2: X=a>`

Figure 61: Continuations when calling a predicate `f(X)` with two rules

For an example see Figure 61. The figure shows the success continuations `sc`, failure continuations `fc` and trail objects that are constructed when calling a predicate `f(X)` which has two rules in the database. When applying the first rule, a new trail is created and the failure continuation set to a `restore`, which can potentially undo the changes done by the first rule and continue with the second rule, if backtracking occurs later.

The overhead of constantly creating these continuation objects is kept small by the good GC support that the RPython toolchain gives (see Section 2.2.2). Since most of the continuations are very short-lived they are collected extremely efficiently by the generational GC.

11.2.3 *Implementing built-ins*

In addition to the core Prolog execution model a number of built-ins have been implemented. Most built-ins are rather straightforward to implement using the continuation-based model. built-ins that always have at most one solution are trivial, built-ins that can have many solutions need some more work, because they typically need a new type of continuation. There are some built-ins that need some care, because they manipulate the current continuations in more complex ways.

The negation built-in `\+ Goal` performs a call to `Goal` but swaps failure and success continuation when executing the call.[2] Indeed, if `Goal` fails the whole construct succeeds, and vice versa.

If-then-else `A -> B ; C` needs to remove those failure continuations that were introduced during the execution of `A`. Removing a failure continuation also means that the trail object which corresponds to the removed choice is merged with its predecessor. Similarly the cut `!` needs to remove all failure continuations that were introduced during the execution of the current predicate. It is not trivial to figure out the extent of the cut, since the continuations are not marked by which predicate they were created for. Therefore a special marker continuation is needed if a predicate that contains a cut is called. In this regard, if-then-else is a much cleaner concept.

The `findall` built-in needs a special sort of success continuation which, when activated, collects the found solution and then forces backtracking.

The exception handling built-ins work as follows. A call to `catch` will insert a special catching success continuation, which will not do anything when actually activated. When `throw` is called, it will walk the chain of success continuations until it finds a matching catching continuation and continue by calling the recovery goal of the `catch` call.

11.3 APPLYING METAJIT TO PYROLOG

In this section the application of MetaJIT to the Prolog interpreter is described. The first task in doing so is correctly placing hints of Chapter 4 and Chapter 5 in the source code of the interpreter:

- In the Prolog interpreter the `jit_merge_point` hint that indicates the interpreter's main loop is put into the `interpret` loop shown in Figure 60. The currently executed Prolog rule is marked as green.

- The `can_enter_jit` hint to indicate the code of the interpreter that is responsible for closing a loop. In an imperative language this hint is placed in the implementation of the bytecode which performs backward jumps. This one is the hardest in Prolog, since there is no explicit loop construct, only tail calls. Therefore the hint was placed in the code that is responsible for applying one specific rule (see Section 11.2.2).

- Before a predicate rule is applied, it is promoted.

- Many classes in the interpreter are marked as *immutable*. Examples for such classes are all the classes implementing Prolog

[2] Together with some extra code to remove the bindings done during the execution of the goal.

```
nrev([], []).
nrev([X|Y], Z) :- nrev(Y, Z1), append(Z1, [X], Z).

append([], L, L).
append([H | T1], T2, [H | T3]) :- append(T1, T2, T3).
```

Figure 62: Code of naive reverse and append

terms, except `Var`; the class that represents Prolog rules (but not predicates, because `assert` and `retract` can change predicate objects).

11.3.1 *Loops in Prolog code*

Since the inserted JIT uses tracing and focuses on producing good code for loops, it is important to discuss when a loop occurs in Prolog. Despite Prolog not having an explicit loop construct, there are still a number of cases in which MetaJIT will detect a loop. A loop for the JIT is simply a situation where the same rule of a predicate will be applied repeatedly (potentially with other rule applications in between).

The most straightforward sort of loop is a loop with *tail calls*, like a list-append where the first argument is instantiated, or an arithmetic loop. However, it is not necessary for the call in the loop to be in a tail position. If one takes the implementation of naive reverse in Figure 62, the second rule of `nrev` will repeatedly call itself (constructing a continuation that calls `append` at every iteration). Thus the JIT will also detect it as a loop. After the base case is reached, those continuations will be activated one after another, which is yet another loop (which is distinct from the loop of `append` itself). Further ways to get loops are failure-driven loops or all-solution predicates, such as `findall`.

11.3.2 *Optimizations by the JIT*

This section describes how the optimizations of Chapter 5 and Chapter 6 help the Prolog interpreter.

A number of classes are marked as immutable. Since a Prolog rule is immutable (even in the presence of assert and retract), all the reads out of it are constant-folded away. This applies in particular to reading the head and body of the rule. The head and body of the rule are themselves immutable terms (since they are usually not variables), thus the JIT will recursively optimize away most of those reads. This means that for the unification of a rule head with a calling term, all of the operations acting on the rule head are constant-folded away.

```
[scont, $i_1$, fcont, trail]

# Check whether the base case applies
$i_2$ = int_eq($i_1$, 0)
guard_false($i_2$)

# X > 0
$i_3$ = int_gt($i_1$, 0)
guard_true($i_3$)

# $X_0$ is X - 1
$i_4$ = int_sub($i_1$, 1)

# recursive call to iterate(Y)
# Check whether assert or retract was
# used on the loop/1 function:
$p_2$ = read_field(<address loop/1>, 'first_rule')
guard_value($p_2$, <address 1st rule of loop/1>)

jump(scont, $i_4$, fcont, trail)
```

Figure 63: The intermediate code for the generated machine code of the `iterate/1` function

Allocation removal can often remove all overhead of using continuations in the interpreter. If a continuation object is created, it will often just be activated quickly afterwards and then not be used anymore. In this case the continuation object will be fully removed by the optimizer. Only in the case when a choice point is created or the continuation chain grows, can the allocation not be removed (for example this is the case for naive reverse in Figure 62). The same applies to trail objects.

In addition to the removal of continuations, allocations of Prolog objects can be avoided by this optimization. When standardizing apart before the application of a rule a copy of the rule body is created. Some parts of the copied body will be immediately deconstructed again, thus they don't need to be allocated at all. In this way, unification is compiled into the trace.

The guards that the tracer automatically inserts into the trace lead to mode and type specialization [TS98]. Due to the inlining the tracer does, called Prolog predicates as well as built-ins are inlined into the trace.

As an example of what the optimizations can achieve in the best case, let's look at what happens when the Prolog interpreter executes a simple arithmetic iteration (see the predicate `iterate/1` in Figure 64 for the code). At first, the interpreter will normally run the `iterate` loop, keeping count of which predicates are executed often. After a few iterations, it will identify the `iterate` predicate as a likely candidate and start tracing it. The generated trace will then be optimized as described above.

Most of the operations in the trace are removed by the optimization step. The resulting trace can be seen in Figure 63. This trace will then be turned into machine code by the machine code backend and can then be executed.

In this simple example the optimizer of MetaJIT was able to remove all the allocations in the trace, since the continuations that are created are immediately activated and do not escape anywhere. The same is true for the copied body of the `iterate` predicate. In addition, even the `Number` object that is used to box the integer value of the loop variable is removed, since each of these objects survives for one iteration of the loop only. Thus the generated machine code can keep the loop index in a machine integer, which can be kept in a CPU register. All the `int_*` operations are just simple machine instructions.

The `jump` instruction at the end of the trace jumps to the beginning again. Thus the trace by itself is an infinite loop. It can only be left via one of the guard instructions. Those guards check that the assumptions of the trace are not violated. If the machine code is executed and the iteration count reaches zero, the first guard will fail and execution will fall back to using the interpreter again.

11.4 EVALUATION

In this section the performance and memory behaviour of the Prolog system when translated to C and then compiled to an executable is evaluated, with and without inserting MetaJIT.[3]

The performance of Pyrolog is compared against that of Sicstus Prolog and SWI-Prolog. Sicstus is run in both its interpreted mode and its compiled mode, using `load_files/2` with the `compilation_mode(consult)` and `compilation_mode(compile)` flags respectively.

11.4.1 *Iteration benchmarks*

The first set of benchmarks uses various methods to implement iteration in Prolog (see Figure 64 for the code). This is completely untypical Prolog code. They are still useful to gauge the maximum speedup the technology can give, as they are the sort of code that a tracing JIT is best at.

The results of running these iteration benchmarks each with 10 million iterations can be seen in Figure 65. For two of the benchmarks, SWI-Prolog failed to finish the run, as it was running out of memory. The interpreter without a JIT seems to be about a bit faster than Sicstus in interpreted mode and is quite a bit slower than the other implementations. With the JIT Pyrolog is faster than all other im-

[3] All the benchmarks and the scripts to run them can be found at: `http://cfbolz.de/phdthesis/`

basic iteration:

```
iterate(0).
iterate(X) :- Y is X - 1, iterate(Y).
```

iteration with call:

```
iterate_call(X) :- c(X, c).
c(0, _).
c(X, Pred) :-
    Y is X - 1, C =.. [Pred, Y, Pred], call(C).
```

iteration with a cut:

```
iterate_cut(0).
iterate_cut(X) :- Y is X - 1, !, iterate_cut(Y).
iterate_cut(X) :- Y is X - 2, iterate_cut(Y).
```

iteration with exceptions:

```
e(0).
e(X) :- X > 0, X0 is X - 1, throw(continue(X0)).
iterate_exception(X) :-
    catch(e(X), continue(X0), iterate_exception(X0)).
```

iteration with a failure-driven loop:

```
g(X, Y, Out) :- Out is X - Y.
g(X, Y, Out) :- Y > 0, Y0 is Y - 1, g(X, Y0, Out).

iterate_failure(X) :- g(X, X, A), fail.
iterate_failure(_).
```

iteration using findall:

```
iterate_findall(X) :-
    findall(Out,
            (g(X, X, Out), 0 is Out mod 50),
            _).
```

iteration with if-then-else:

```
equal(0, 0). equal(X, X).
iterate_if(X) :- equal(X, 0) -> true ;
                 Y is X - 1, iterate_if(Y).
```

Figure 64: Iteration benchmarks

	Sicstus-interp	Sicstus-comp	SWI-Prolog	Pyrolog-interp	Pyrolog-JIT
iterate	13255.00 ± 24.71	260.50 ± 38.43	558.25 ± 7.54	8331.00 ± 10.63	12.50 ± 0.99
	1.00 ×	50.88 ×	23.74 ×	1.59 ×	1060.40 ×
iterate_call	21416.25 ± 22.05	2461.75 ± 39.92	—	20865.50 ± 20.38	12.65 ± 2.24
	1.00 ×	8.70 ×		1.03 ×	1692.98 ×
iterate_cut	16354.25 ± 24.26	375.75 ± 42.78	801.75 ± 10.77	15261.98 ± 60.59	2317.20 ± 366.81
	1.00 ×	43.52 ×	20.40 ×	1.07 ×	7.06 ×
iterate_exception	37547.50 ± 25.01	12651.75 ± 17.69	—	31872.78 ± 301.94	6792.77 ± 144.94
	1.00 ×	2.97 ×		1.18 ×	5.53 ×
iterate_failure	28908.00 ± 22.72	813.25 ± 10.30	4676.75 ± 20.04	22231.67 ± 29.84	75.65 ± 1.69
	1.00 ×	35.55 ×	6.18 ×	1.30 ×	382.13 ×
iterate_findall	36889.00 ± 109.42	8109.25 ± 9.30	8014.75 ± 67.60	23324.03 ± 45.12	296.73 ± 25.52
	1.00 ×	4.55 ×	4.60 ×	1.58 ×	124.32 ×
iterate_if	19943.25 ± 16.84	705.25 ± 32.00	1580.00 ± 0.00	18273.12 ± 29.03	16.85 ± 0.84
	1.00 ×	28.28 ×	12.62 ×	1.09 ×	1183.58 ×
geom. mean	1.00	11.28	7.47	1.21	89.00

Figure 65: Benchmark times for iteration benchmarks

plementations, sometimes significantly, apart from when the cut is involved. For the cut the JIT is unable to remove the overhead of the creation of the (immediately removed again) choice point.

In the case of `iterate_call` the JIT is able to show its full strength. The JIT can optimize the `call` built-in by optimistically assuming that the target predicate will stay the same. This is simply ensured with a guard, which would fail if the assumption turns out to be false later.

The `iterate_exception` benchmark is not idiomatic Prolog code (hopefully nobody writes actual code like this). However, it showcases that the JIT optimizes all features that the interpreter implements, without the interpreter author having to do extra work.

11.4.2 *Classical Prolog benchmarks*

In addition to the unrealistic micro-benchmarks from the last section the Prolog implementations were also measured against some larger programs, most of them well-known benchmarks. Many of these benchmarks execute so quickly that they had to be run many times to get sensible measurements. The following benchmarks were each run 500 times: `chat_parser`, `crypt`, `deriv`, `qsort` sorting a list of 50 elements, `reducer`, `zebra`. The sizes of the benchmarks can be seen in Figure 66.

Name	lines of code
arithmetic	148
boyer	390
chat_parser	1177
crypt	94
deriv	53
meta_nrev	18
nrev	15
primes	28
qsort	45
queens	22
reducer	380
tak	23
zebra	48

Figure 66: Lines of code of Prolog benchmarks

In addition the following benchmarks were used: boyer, tak, nrev which uses naive reverse to reverse a list of 1700 elements, queens solving the queens puzzle with 11 queens, primes searching for all primes up to 10 000. arithmetic is a declarative arbitrary-precision arithmetic implementation using lists of bits to represents the numbers. The benchmark computes 14! and is derived from code in "The Reasoned Schemer" [FBK05].

The results of these benchmarks can be seen in Figure 67. The Prolog interpreter without the JIT is significantly slower for these more realistic Prolog programs. It is between 5 times slower and 50% faster than Sicstus in interpreted mode, and significantly slower than the other Prolog implementations.

If the JIT is also inserted, the execution times always improves over interpretation. The JIT gives a speedup of more up to 20 times, which makes it competitive with Sicstus in compiled mode for the benchmarks queens and arithmetic.

I believe that the bad cut performance of Pyrolog (see Section 11.4.1) is the biggest problem of the implementation. Indeed, all benchmarks except arithmetic use the cut in some way.

11.4.3 *Warmup time*

As with Python in Section 10.3.3 the warmup times are measured by taking the time of the first iteration of the benchmark after starting the VM. The results can be seen in Figure 68. Only nrev is warmed

	Sicstus-interp	Sicstus-comp	SWI-Prolog	Pyrolog-interp	Pyrolog-JIT
arithmetic	3557.50 ± 10.64	464.00 ± 9.72	941.75 ± 9.81	4903.07 ± 9.26	240.38 ± 11.51
	1.00 ×	7.67 ×	3.78 ×	0.73 ×	14.80 ×
boyer	423.25 ± 9.30	33.75 ± 9.61	97.50 ± 8.60	652.20 ± 14.39	82.70 ± 15.95
	1.00 ×	12.54 ×	4.34 ×	0.65 ×	5.12 ×
chat_parser	21092.25 ± 11.30	4630.00 ± 10.87	9149.50 ± 9.87	47921.80 ± 112.51	7739.80 ± 105.62
	1.00 ×	4.56 ×	2.31 ×	0.44 ×	2.73 ×
crypt	2117.25 ± 9.91	176.25 ± 9.61	592.50 ± 8.60	1377.65 ± 2.53	170.10 ± 2.07
	1.00 ×	12.01 ×	3.57 ×	1.54 ×	12.45 ×
deriv	1065.75 ± 9.81	173.25 ± 9.30	539.75 ± 11.30	3525.25 ± 19.91	663.10 ± 15.42
	1.00 ×	6.15 ×	1.97 ×	0.30 ×	1.61 ×
meta_nrev	2006.25 ± 11.48	385.50 ± 9.87	837.75 ± 15.04	2810.00 ± 16.39	434.52 ± 5.77
	1.00 ×	5.20 ×	2.39 ×	0.71 ×	4.62 ×
nrev	853.00 ± 9.10	48.00 ± 7.94	274.50 ± 10.83	1115.00 ± 17.43	232.95 ± 8.79
	1.00 ×	17.77 ×	3.11 ×	0.77 ×	3.66 ×
primes	1961.00 ± 5.95	64.50 ± 9.87	451.50 ± 12.20	2148.97 ± 14.76	484.55 ± 7.93
	1.00 ×	30.40 ×	4.34 ×	0.91 ×	4.05 ×
qsort	1223.75 ± 9.61	143.75 ± 9.61	544.75 ± 10.86	3154.12 ± 18.49	295.05 ± 10.28
	1.00 ×	8.51 ×	2.25 ×	0.39 ×	4.15 ×
queens	8334.75 ± 10.86	411.00 ± 7.43	1934.25 ± 11.65	7601.95 ± 11.42	388.85 ± 4.65
	1.00 ×	20.28 ×	4.31 ×	1.10 ×	21.43 ×
reducer	5188.00 ± 10.12	730.75 ± 9.30	2774.00 ± 17.64	18423.25 ± 17.29	2850.50 ± 28.80
	1.00 ×	7.10 ×	1.87 ×	0.28 ×	1.82 ×
tak	635.50 ± 9.87	18.00 ± 7.94	87.50 ± 9.67	813.88 ± 17.88	169.07 ± 18.59
	1.00 ×	35.31 ×	7.26 ×	0.78 ×	3.76 ×
zebra	2453.00 ± 9.10	980.00 ± 4.44	2176.00 ± 9.72	10513.33 ± 13.26	4270.68 ± 9.94
	1.00 ×	2.50 ×	1.13 ×	0.23 ×	0.57 ×
geom. mean	1.00	8.47	2.73	0.61	3.72

Figure 67: Benchmark times for classical Prolog benchmarks

	Sicstus-comp	Pyrolog-interp	Pyrolog-JIT	Pyrolog-JIT cold
arithmetic	464.00 ± 9.72	4903.07 ± 9.26	240.38 ± 11.51	444.00
boyer	33.75 ± 9.61	652.20 ± 14.39	82.70 ± 15.95	508.00
chat_parser	4630.00 ± 10.87	47921.80 ± 112.51	7739.80 ± 105.62	28337.00
crypt	176.25 ± 9.61	1377.65 ± 2.53	170.10 ± 2.07	418.00
deriv	173.25 ± 9.30	3525.25 ± 19.91	663.10 ± 15.42	1065.00
meta_nrev	385.50 ± 9.87	2810.00 ± 16.39	434.52 ± 5.77	457.00
nrev	48.00 ± 7.94	1115.00 ± 17.43	232.95 ± 8.79	239.00
primes	64.50 ± 9.87	2148.97 ± 14.76	484.55 ± 7.93	578.00
qsort	143.75 ± 9.61	3154.12 ± 18.49	295.05 ± 10.28	399.00
queens	411.00 ± 7.43	7601.95 ± 11.42	388.85 ± 4.65	477.00
reducer	730.75 ± 9.30	18423.25 ± 17.29	2850.50 ± 28.80	8041.00
tak	18.00 ± 7.94	813.88 ± 17.88	169.07 ± 18.59	219.00
zebra	980.00 ± 4.44	10513.33 ± 13.26	4270.68 ± 9.94	4543.00

Figure 68: Warmup times

up in one iteration, for all other benchmarks the final speed is higher. However, the JIT already helps after one iteration.

11.4.4 *Memory footprint*

To measure the overhead of having an object-oriented object model and of representing the interpreter state in continuation objects, the memory footprint of each Prolog interpreter was also measured, by running each benchmark and continuously sampling the physical memory the process used. The numbers are reported in Figure 69 and are the maximum amount of memory each benchmark used during the run.

For most benchmarks the memory footprint of the interpreter is about 4–40 times larger than that of the other interpreters. The size difference that the `nrev` benchmarks exhibits shows that continuations have a large memory overhead, because `nrev` builds a chain of continuations as large as the reversed list has elements. In some benchmarks using the JIT adds memory overhead, which is to be expected because the generated code takes memory. Interestingly the JIT can also *save* memory, for example for `chat_parser` and `reducer`. The reason is that allocation removal can remove the allocation of some continuations and store the information more compactly on the CPU stack.

	Sicstus-interp	Sicstus-comp	SWI-Prolog	Pyrolog-interp	Pyrolog-JIT
arithmetic	3.7 MiB	3.4 MiB	3.2 MiB	37.9 MiB	39.7 MiB
boyer	4.8 MiB	4.8 MiB	3.2 MiB	38.4 MiB	42.3 MiB
chat_parser	3.6 MiB	3.5 MiB	2.5 MiB	38.5 MiB	88.3 MiB
crypt	3.4 MiB	3.4 MiB	2.5 MiB	20.9 MiB	17.3 MiB
deriv	4.4 MiB	3.4 MiB	2.5 MiB	38.6 MiB	41.4 MiB
meta_nrev	35.6 MiB	23.3 MiB	28.2 MiB	38.4 MiB	53.4 MiB
nrev	4.8 MiB	4.8 MiB	2.9 MiB	39.3 MiB	56.7 MiB
primes	23.6 MiB	15.5 MiB	14.4 MiB	51.4 MiB	53.1 MiB
qsort	3.9 MiB	3.4 MiB	2.5 MiB	38.6 MiB	40.2 MiB
queens	3.4 MiB	3.4 MiB	2.5 MiB	38.0 MiB	26.1 MiB
reducer	4.7 MiB	3.4 MiB	2.5 MiB	38.6 MiB	52.1 MiB
tak	4.8 MiB	3.4 MiB	2.5 MiB	38.4 MiB	40.3 MiB
zebra	3.5 MiB	3.4 MiB	2.5 MiB	38.2 MiB	40.9 MiB

Figure 69: Memory footprint for classical Prolog benchmarks

11.5 CONCLUSION

In this chapter a simple Prolog interpreter written in RPython was presented, which can be compiled into a C-level VM with the RPython translation toolchain, optionally also using MetaJIT in the process. The resulting VM is reasonably efficient and can be very fast in cases where the meta-tracing JIT works well. The experiments with Pyrolog show that meta-tracing can also be used for languages that are not imperative and object-oriented. Doing so requires to carefully design the representation of the interpreter state as continuations in the interpreter.

The experiments also show, that employing well-tuned general purpose technologies can beat Prolog-specific implementations under certain circumstances. However, the MetaJIT integration needs more care and thought, because no loops are readily available in Prolog. Also, some features of Prolog seem hard to support efficiently in a high-level model, such as the cut. Prolog is not really a simple language, even though it seems to be.

At the moment there are also some disadvantages to the approach. The memory usage of the resulting interpreter can be very bad, due to the overhead of using many objects and the lack of low-level control. In addition, the way MetaJIT works is not always very transparent, sometimes making it hard to know why certain Prolog code is compiled efficiently to machine code and other code is not. Sometimes the JIT compiler itself can take too much time to be profitable.

To the best of my knowledge, Pyrolog is the first Prolog implementation that defers all compilation to run-time. Prolog can greatly benefit from JIT compilation techniques, given its dynamic nature. To make a really efficient JIT for Prolog it might be necessary to write the JIT by hand to be able to carefully adapt it to the peculiarities of Prolog.

12 COMPARISON TO PARTIAL EVALUATION

In this chapter meta-tracing will be compared to partial evaluation. This will be done with the help of a functional model of partial evaluation and of tracing for a simple imperative flow-graph language. This model takes the form of a Prolog program. Section 12.1 defines the semantics of that flow-graph language with the help of an interpreter for it. Section 12.2 presents a naive polyvariant online partial evaluator for the language and Section 12.3 presents a tracer for the language. It also explains the semantics of traces with an interpreter for executing them. Afterwards an extension to the language and the tracer are considered by introducing the `promote` hint of Chapter 5 into the language. The hint can be used to freeze values of certain run-time variables into the trace. This information is exploited in Section 12.5 to optimize traces, a process which is similar to partial evaluation – a similarity that already discussed in Chapter 6.

12.1 EXECUTABLE MODELS IN PROLOG

To capture the essential differences between partial evaluation and meta-tracing, we will look at a minimal implementation of both in Prolog.[1] In this way, an *executable model* is given for both partial evaluation and tracing. The two techniques will be applied to the flow-graph language.

The language is conceptionally similar to RPython's intermediate representation flow graphs of the translation toolchain, but a bit more restricted. It does not have function calls, only labelled basic blocks that consist of a series of linearly executed operations, followed by a conditional jump. To get an unconditional jump, a conditional jump with a constant condition can be used. Every operation assigns a value to a variable, which is computed by applying an operation to some arguments. The flow-graph language is similar to that used in a formalization of tracing by Guo and Palsberg [GP11] to show that tracing and normal execution behave in the same way.

A program to raise `x` to the `yth` power in that language is shown in Figure 70. Lines 1–13 give the program in a pseudo-syntax. In the actual code below, the program is expressed as Prolog facts. Every fact of `block` declares one basic block of the program by first giving the label of the block followed by the code, which is a series of `op` statements terminated by an `if` or a `print_and_stop`. Operations are

[1] Prolog is chosen as an implementation language because of its succinctness due to pattern matching. No non-determinism is used in the code.

```
% power:
%     res = 1
%     c = y >= 0
%     if c goto power_rec else goto power_done

% power_rec:
%     res = res * x
%     y = y - 1
%     c = y >= 1
%     if c goto power_rec else goto power_done

% power_done:
%     print_and_stop res

block(power, op(res, assign, const(1), _,
             if(var(y), power_rec, power_done))).
block(power_rec, op(res, mul, var(res), var(x),
                 op(y, sub, var(y), const(1),
                 if(var(y), power_rec, power_done)))).
block(power_done, print_and_stop(var(res))).
```

Figure 70: Raising x to the yth in the flow graph language

performed by `op` statements which take two arguments[2] and are of the form:
`op(result_var, operation, argument1, argument2, next_statement)`
Arguments can be either variables in the form `var(name)` or constants in the form `const(value)`. Operations like assignment that only need one argument simply ignore the second one. Conditions are of the form `if(argument, label1, label0)`. When the value of the argument is 1, execution continues at `label1`; if it is 0, at `label0`. If the value is neither, the program is not well-formed.

The code of the interpreter, partial evaluator and tracer of the language need some helper functionality, which can be seen in Figure 71. The first few helper functions are concerned with the handling of environments. Environments are lists of `name/value` pairs. They are the data structures the interpreter uses to map variable names occurring in the program to the variables' current values.

The `lookup` function finds a key in an environment list, the `write_env` function adds a new key/value pair to an environment, `remove_env` removes a key. The `resolve` function is used to take either a constant or a variable and return its value. If it is a constant, the value of that constant is returned, if it is a variable, it is looked up in the environment.

The figure also defines the `plookup` and `presolve` variants, which are used for partial evaluation. The difference to the normal function is that they deal with names not being found in the environment.

[2] The whole language can be trivially extended to also support operations with more arguments.

```
lookup(X, [], _) :- throw(key_not_found(X)).
lookup(Name, [Name/Value | _], Value) :- !.
lookup(Name, [_ | Rest], Value) :- lookup(Name, Rest, Value).

write_env([], Name, Value, [Name/Value]).
write_env([Name/_ | Rest], Name, Value, [Name/Value | Rest]) :- !.
write_env([Pair | Rest], Name, Value, [Pair | NewRest]) :-
    write_env(Rest, Name, Value, NewRest).

remove_env([], _, []).
remove_env([Name/_ | Rest], Name, Rest) :- !.
remove_env([Pair | Rest], Name, [Pair | NewRest]) :-
    remove_env(Rest, Name, NewRest).

resolve(const(X), _, X).
resolve(var(X), Env, Y) :- lookup(X, Env, Y).

plookup(Name, [], var(Name)).
plookup(Name, [Name/Value | _], const(Value)) :- !.
plookup(Name, [_ | Rest], Value) :- plookup(Name, Rest, Value).

presolve(const(Value), _, const(Value)).
presolve(var(Name), PEnv, X) :- plookup(Name, PEnv, X).
```

Figure 71: Helper functions

```
interp(op(ResultVar, Op, Arg1, Arg2, Rest), Env) :-
    interp_op(ResultVar, Op, Arg1, Arg2, Env, NEnv),
    interp(Rest, NEnv).

interp(if(Arg, L1, L0), Env) :-
    resolve(Arg, Env, RArg),
    (RArg == 0 ->
        L = L0
    ;
        L = L1
    ),
    block(L, Code),
    interp(Code, Env).

interp(print_and_stop(Arg), Env) :-
    resolve(Arg, Env, Val),
    print(Val), nl.

interp_op(ResultVar, Op, Arg1, Arg2, Env, NEnv) :-
    resolve(Arg1, Env, RArg1),
    resolve(Arg2, Env, RArg2),
    do_op(Op, RArg1, RArg2, Res),
    write_env(Env, ResultVar, Res, NEnv).

do_op(assign, L, _, L).
do_op(mul, X, Y, Z) :- Z is X * Y.
do_op(add, X, Y, Z) :- Z is X + Y.
do_op(sub, X, Y, Z) :- Z is X - Y.
do_op(eq, X, Y, Z) :- X == Y -> Z = 1; Z = 0.
do_op(ge, X, Y, Z) :- X >= Y -> Z = 1; Z = 0.
```

Figure 72: Interpreter source code

To execute a program, an interpreter in the form of an `interp` predicate is defined. It takes as its first argument the operation to execute, and as its second argument the current environment. The code for the interpreter can be seen in Figure 72.

To execute an operation (line 24) a helper function `interp_op` is used (line 42). It resolves the operation's arguments into values, then the operation is executed using the `do_op` predicate. Afterwards the result is written back into the environment. Then `interp` is called on the rest of the program.

To execute `print_and_stop` (line 38) the argument is resolved, printed and then execution stops.

The conditional jump (line 28) is only slightly more difficult. First the variable is resolved using the environment. If the variable is zero, execution continues at the second block, otherwise it continues at the first block.

Given this interpreter, we can execute the example program from Figure 70 like this, on a Prolog console:

```
?- block(power, Code), interp(Code, [x/10, y/10]).
10000000000
```

12.2 PARTIAL EVALUATION OF THE FLOWGRAPH LANGUAGE

Now that we have seen the interpreter for the flow graph language, a polyvariant [JGS93, p. 130] online partial evaluator for it is presented. The partial evaluator is quite straightforward, its control mechanisms are rudimentary and it does not ensure termination.

The partial evaluator cannot use normal environments, because unlike the interpreter not all variables' values are known to it. It will therefore work on partial environments, which store just the known variables. For these partial environments, some new helper functions are needed, which are also in Figure 71.

The function `plookup` (line 18) takes a variable and a partial environment and returns either `const(Value)` if the variable is found in the partial environment or `var(Name)` if it is not. Equivalently, `presolve` (line 22) is like `resolve`, except that it uses `plookup` instead of `lookup`.

The partial evaluator for the language can be seen in Figure 73. The `pe` predicate takes a partial environment, the current operations and potentially returns a new operation. The potentially generated residual operation is stored into the output argument `Residual`. The output argument of the recursive call is the last argument of the newly created residual operation, which will then be filled by the recursive call.

To partially evaluate a simple operation (line 55), a helper predicate `pe_op` is used (line 78). It resolves the arguments of the operation using the partial environment. If both arguments are constants, the

```
pe(op(ResultVar, Op, Arg1, Arg2, Rest), PEnv, Residual) :-
    pe_op(ResultVar, Op, Arg1, Arg2,
          PEnv, NEnv, Residual, RestResidual),
    pe(Rest, NEnv, RestResidual).

pe(print_and_stop(Arg), Env, print_and_stop(RArg)) :-
    presolve(Arg, Env, RArg).

pe(if(Arg, L1, L0), PEnv, Residual) :-
    presolve(Arg, PEnv, RArg),
    (RArg = const(C) ->
        (C = 0 -> L = L0 ; L = L1),
        do_pe(L, PEnv, LR),
        Residual = if(const(1), LR, LR)
    ;
        RArg = var(V),
        write_env(PEnv, V, 1, NEnvTrue),
        do_pe(L1, NEnvTrue, L1R),
        write_env(PEnv, V, 0, NEnvFalse),
        do_pe(L0, NEnvFalse, L0R),
        Residual = if(RArg, L1R, L0R)
    ).

pe_op(ResultVar, Op, Arg1, Arg2, PEnv,
                 NEnv, Residual, RestResidual) :-
    presolve(Arg1, PEnv, RArg1),
    presolve(Arg2, PEnv, RArg2),
    (RArg1 = const(C1), RArg2 = const(C2) ->
        do_op(Op, C1, C2, Res),
        write_env(PEnv, ResultVar, Res, NEnv),
        RestResidual = Residual
    ;
        remove_env(PEnv, ResultVar, NEnv),
        Residual = op(ResultVar, Op, RArg1, RArg2, RestResidual)
    ).

do_pe(L, PEnv, LR) :-
    (code_cache(L, PEnv, LR) -> true ;
        gensym(L, LR),
        assert(code_cache(L, PEnv, LR)),
        block(L, Code),
        pe(Code, PEnv, Residual),
        assert(block(LR, Residual))
    ).
```

Figure 73: Partial evaluation rules

operation can be executed, and no new operation is produced. Otherwise, a new residual operation needs to be produced, which is exactly like the operation currently looked at. Also, in that case the result variable needs to be removed from the partial environment, because even though it might have been known before the operation, the operation overwrites it with an unknown value. This rule is where the main optimization in the form of constant folding happens.

Note how the first case of this helper predicate is just like interpretation. The second case doesn't really do anything, it just produces a residual operation. This relationship between normal evaluation and partial evaluation is very typical. In a partial evaluator every rule for executing an operation is split into two parts. One part just residualizes the operation if not enough information is available. If enough information is known, the operation is executed during partial evaluation like in the interpreter.

Partially evaluating `print_and_stop` (line 60) is easy as well, it is just turned into another `print_and_stop` statement.

Conditional jumps (line 63) are more interesting. The residual code of a conditional jump is always a conditional jump itself. The target label of that residual jump is computed by asking the partial evaluator to produce residual code for the labels `L1` and `L0` with the given partial environment. There is an optimization that checks whether the condition variable is in the static environment, to only produce residual code for one path in that case.

This rule is the one that causes the partial evaluator to potentially do much more work than the interpreter, because after an `if` sometimes both paths need to be explored. In the worst case this process never stops, so a real partial evaluator would need to ensure somehow that it terminates in all cases. This problem is simply ignored in this partial evaluator.

The whole process of partially evaluating a block with a certain partial environment is started by the `do_pe` predicate (line 91). The `do_pe` predicate makes sure that the same block is not partially evaluated twice. This is achieved by memoizing code that was already partially evaluated in the past by keeping a mapping of `Label, Partial Environment` to `Label of the residual code`. If this code cache indicates that label `L` was already partially evaluated with partial environment `PEnv`, then the previous residual code label `LPrevious` is returned. Otherwise, a new label is generated with `gensym`, the code cache is informed of that new label with `assert`, then the block is partially evaluated and the residual code is added to the database. This predicate is what makes the partial evaluator polyvariant: For every label of the original program, many labels can be produced in the residual program. The partial evaluator never throws away information from the partial environment, which means it never generalizes the residual code.

```
?- do_pe(power, [y/5], Label), block(Label, Code),
   interp(Code, [x/10]).
100000
Label = power1,
Code = jump(power_rec1) .

?- listing(code_cache).

code_cache(power, [y/5], power1).
code_cache(power_rec, [y/5, res/1], power_rec1).
code_cache(power_rec, [y/4], power_rec2).
code_cache(power_rec, [y/3], power_rec3).
code_cache(power_rec, [y/2], power_rec4).
code_cache(power_rec, [y/1], power_rec5).
code_cache(power_done, [y/0], power_done1).

?- listing(block).
... all the user-written rules of the program
block(power_done1, print_and_stop(var(res))).
block(power_rec5, op(res, mul, var(res), var(x),
                        if(const(1), power_done1, power_done1))).
block(power_rec4, op(res, mul, var(res), var(x),
                        if(const(1), power_rec5, power_rec5))).
block(power_rec3, op(res, mul, var(res), var(x),
                        if(const(1), power_rec4, power_rec4))).
block(power_rec2, op(res, mul, var(res), var(x),
                        if(const(1), power_rec3, power_rec3))).
block(power_rec1, op(res, mul, const(1), var(x),
                        if(const(1), power_rec2, power_rec2))).
block(power1, if(const(1), power_rec1, power_rec1)).
```

Figure 74: Partially evaluating the power function

With this code we can look at the "Hello World" of partial evaluation. We can ask the partial evaluator to compute a power function where the exponent y is a fixed number, e.g. 5, and the base x is unknown. This is done in Figure 74.

The code_cache tells which residual labels correspond to which original label under which partial environments. For example, power1 contains the code of power under the assumption that y is 5. Looking at the block listing, the label power1 corresponds to code that simply multiplies res by x five times without using the variable y at all. The loop that was present in the original program has been fully unrolled, the loop variable y has disappeared. Hopefully this is faster than the original program.

12.2.1 *Control in realistic partial evaluators*

The control algorithm of the partial evaluator in this section is very simple. It never throws away any information and does not guarantee termination. This approach has been taken by very early partial evaluators and also more recently by hybrid partial evaluation [SC11].

However, most partial evaluators use more sophisticated control algorithms. The control of a partial evaluation system needs to ensure that the process of partial evaluation terminates even if the input program does not. Furthermore, commonly executed paths of the original program should be chosen for partial evaluation.

After the goal of ensuring termination, the most important task of a control algorithm is to try to find the right balance between over-specialization and under-specialization [LB02]. Over-specialization occurs when the generated residual code is specialized too much, which leads to a lot of similar code being generated without additional performance improvements. Under-specialization occurs when the residual code is too general, which means it does not remove enough of the interpretative overhead to speed up the program in significant ways. In its most extreme form, under-specialization can lead to the program not being specialized at all.

The control algorithms used in many partial evaluators try to reach a balance between over- and under-specialization by complex heuristics that have good theoretical properties and handle the specialization of a wide variety of cases well. They need to ensure that all reachable paths in the specialized program are covered while still ensuring termination. Many of these heuristics are based on well-quasi orders, a very popular one being the homeomorphic embedding [Leu02].

12.2.2 *Conclusion*

In this section we saw a partial evaluator for the flow graph language. The partial evaluator essentially duplicates every rule of the interpreter. If all the arguments of the current operation are known, it acts like the interpreter, otherwise it simply copies the operation into the residual code.

12.3 A TRACER FOR THE FLOW GRAPH LANGUAGE

This section presents a tracer for the same language and how it relates to both execution and to partial evaluation. The idea of a tracer is to do completely normal interpretation but at the same time keep a log of all the normal operations (i.e. non-control-flow operations) that were performed. This continues until the tracer executes the code block where it started at, in which case the trace corresponds to a closed loop. Then tracing stops and the last operation is replaced by a jump to the start of the trace. After tracing has ended, the trace can be executed, optionally optimizing it before that.

The tracer presented here abstracts over a real tracing system in several ways. The tracer does not cache its generated traces for later reuse. The model also does not explain how tracing is started in a real system (typically by doing profiling and starting tracing after a

```
trace(op(ResultVar, Op, Arg1, Arg2, Rest), Env,                        99
      op(ResultVar, Op, Arg1, Arg2, T), TraceAnchor) :-                100
    interp_op(ResultVar, Op, Arg1, Arg2, Env, NEnv),                   101
    trace(Rest, NEnv, T, TraceAnchor).                                 102

trace(print_and_stop(V), Env, _, _) :-                                 104
    resolve(V, Env, Val),                                              105
    print(Val), nl.                                                    106

trace(if(Arg, L1, L0), Env,                                            108
      guard(Arg, Val, if(const(1), OL, OL), T), TraceAnchor) :-        109
    resolve(Arg, Env, Val),                                            110
    (Val == 0 ->                                                       111
        L = L0, OL = L1                                                112
    ;                                                                  113
        L = L1, OL = L0                                                114
    ),                                                                 115
    trace_jump(L, Env, T, TraceAnchor).                                116

trace_jump(L, Env, loop, traceanchor(L, FullTrace)) :-                 118
    do_optimize(FullTrace, OptTrace),                                  119
    runtrace(OptTrace, Env, OptTrace).                                 120

trace_jump(L, Env, T, TraceAnchor) :-                                  122
    block(L, Code),                                                    123
    trace(Code, Env, T, TraceAnchor).                                  124

do_trace(L, Env) :-                                                    126
    block(L, StartCode),                                               127
    trace(StartCode, Env, ProducedTrace,                               128
          traceanchor(L, ProducedTrace)).                              129
```

Figure 75: A tracer for the flow graph language

threshold is reached). Also, the stop condition of the tracer is simplified. However, the code still explains the actual tracing and execution of traces.

To write a tracer, we start from the rules of the interpreter, rename the predicate to trace and add some extra arguments. The code of the tracer can be seen in Figure 75.

To trace an op statement (line 99), the helper predicate interp_op of the interpreter is reused. With respect to the effect on the environment, the tracer acts exactly like the interpreter. The meaning of the arguments of trace is as follows: The first and second argument are the operation currently executed and the environment, like in the interpreter. The argument after that is an output argument that collects the currently traced operation. In the first rule it is a copy of the operation that was executed. TraceAnchor is additional information about the trace that is being built right now, most of the time it is just handed on to the recursive call of trace. We will see soon what it contains.

The rule for print_and_stop (line 104) is very simple, as execution (and therefore also tracing) simply stops there.

Left is the rule for the control flow operation `if` (line 108). A trace linearizes one execution path, it contains no jumps. Therefore an `if` statement needs special treatment, because it is where the control flow can diverge from the trace. The trace is a linear list of operations, therefore it can only record one of the two possible paths. When later executing the trace, it is possible for the other path to be taken. Therefore we need to make sure that the same conditions which were true or false during tracing are still true or false during the execution of the trace. This is done with a guard operation, which checks for this condition. The arguments of the guard are: The argument that is being guarded, the value that this argument had during tracing, compensation code that the interpreter needs to execute when the guard fails, and the rest of the trace. The compensation code just contains a jump to the other path that was not taken during tracing (more operations will be added there later).

After the guard has been reached, the tracer needs to decide whether tracing should be continued. This is done with the `trace_jump` helper function. When a jump to the starting label is reached, tracing should stop. Therefore, the implementation of `trace_jump` contains two cases.

In the first rule, we see what `TraceAnchor` is (line 118). It is a term of the form `traceanchor(StartLabel, FullTrace)`. `StartLabel` is a label in the program where tracing started (and where it should end as well, when the loop is closed). The argument `FullTrace` contains the full trace that is being built right now.

The first rule (line 118) matches when the target-label `L` is the same as the one stored in the trace anchor. If that is the case, tracing stops. The produced operation in the trace is `loop`, which jumps back to the beginning of the trace when executed. Afterwards the trace is printed, optimized, and then run, using the `FullTrace` part of the `traceanchor`. If the label we jump to is *not* the `StartLabel` (line 122) we simply continue tracing without recording any operation.

Finally, `do_trace` (line 126) is a small helper predicate that can be used to conveniently start tracing. The predicate takes a label and an environment and executes the code at the label with the given environment by first producing a trace, then executing the trace and eventually jumping back to interpretation, if a guard fails. It does this by reading the code at label `L` with the `block` statement, and then calling trace with an unbound variable `ProducedTrace` to hold the trace and a trace anchor that contains the label where tracing started and the produced trace variable.

12.3.1 *Executing traces*

In a real tracing system, the traces would be turned into machine code and executed by the CPU. In our small model, we will simply write

```
runtrace(op(ResultVar, Op, Arg1, Arg2, Rest), Env, WholeTrace) :-       130
    interp_op(ResultVar, Op, Arg1, Arg2, Env, NEnv),                    131
    runtrace(Rest, NEnv, WholeTrace).                                   132

runtrace(guard(Arg, C, CompensationCode, Rest), Env, WholeTrace) :-     134
    resolve(Arg, Env, Val),                                             135
    (Val == C ->                                                        136
        runtrace(Rest, Env, WholeTrace)                                 137
    ;                                                                   138
        interp(CompensationCode, Env)                                   139
    ).                                                                  140

runtrace(loop, Env, WholeTrace) :-                                      142
    runtrace(WholeTrace, Env, WholeTrace).                              143
```

Figure 76: Executing traces

another interpreter for them. This interpreter is very small because traces can contain only `op`, `guard`, and `loop` statements. The code is shown in Figure 76.

The rule for `op` is equivalent to the corresponding `interp` rule, it reuses the same helper function. An extra argument `WholeTrace` is needed, which is always just handed over to the recursive call of `runtrace`. The `WholeTrace` argument is used in the execution of the `loop` statement to simply start from the beginning (line 142).

The remaining question is what to do when encountering a guard (line 134). In that case the guard condition needs to be checked. If the guard succeeds, executing the trace can continue. Otherwise the trace is aborted and the regular interpreter executes the compensation code.

12.3.2 *Control in realistic tracing systems*

Compared to the often complex control algorithms in partial evaluators, the control algorithms of realistic tracers is often significantly simpler. Tracing is usually fully online, with no upfront analysis of the interpreter. Since tracing generally focuses on loops, control decisions of a tracing systems are restricted to the question of whether the end of the currently traced loop is reached, which is simple to decide.[3]

Ensuring termination in a tracing system is easier than in a partial evaluator. A partial evaluator typically runs ahead of time and it always needs to terminate. A tracer always runs the executed program, therefore the tracer only needs to terminate if the program terminates. Thus the termination behaviour of a tracer that never stops tracing because the program itself is not terminating would still be correct.

[3] It is slightly harder in languages that model iteration as recursion, such as Prolog or Scheme.

Tracing indefinitely is of course not desirable, due to the growing memory needed for the built trace and the fact that no machine code is ever generated. To solve this problem, most tracing JITs impose a limit on the length of the trace that is currently being recorded. When that limit is hit, tracing stops, the trace is discarded and normal interpretation resumes. Many tracing JITs employ no other heuristics apart from this one.

In dealing with over- and under-specialization, tracing JITs have a significant advantage over classical partial evaluator, the fact that they operate at run-time. Since tracing only starts after a profiling phase has identified a commonly executed loop, it is very unlikely that the code generated for that loop is not actually needed, as would be the case in over-specialization. On the other hand, under-specialization can again be solved by making use of the run-time nature of tracing: When the value of a variable would be needed to generate good code, the tracer can simply observe which value the running program actually uses. How this is done is shown in the next section. This gaining of additional information at the place where it is needed is not easily possible for classical partial evaluation.

Another advantage of tracing JITs is the way they deal with rare cases. An ahead-of-time classical partial evaluator needs to generate code that deals with all possible contingencies. Of these there are typically many in dynamic languages, most of which never occur. This alone is a reason why under-specialization often occurs when partially evaluating an interpreter for a dynamic language. Using tracing, all these possible extra control flow paths are simply ignored and expressed via guards. When a guard fails, which is hopefully a rare occurrence, execution can always fall back to interpretation. If a guard starts to fail often, a new trace for that path can always be generated.

12.3.3 *Conclusion*

In this section we have seen a very minimalistic tracer and an interpreter for the produced traces. The tracer is very much like the original interpreter, it just additionally keeps track of which operations were executed, in addition to executing the program. Tracing stops when a loop is closed, then the trace can be optimized and run. Running a trace continues until a failing guard is hit. At that point, execution goes back to the normal interpreter (and, in this very simple implementation, stays there).

In the next section the flow graph language will be extended with a hint that makes it possible to do run-time feedback of information into the trace.

```
interp(promote(_, L), Env) :-                                   144
    block(L, Code),                                             145
    interp(Code, Env).                                          146

trace(promote(Arg, L), Env,                                     148
      guard(Arg, Val, if(const(1), L, L), T), TraceAnchor) :-   149
    resolve(Arg, Env, Val),                                     150
    trace_jump(L, Env, T, TraceAnchor).                         151
```

Figure 77: Adding promotion

12.4 introducing promotion

As it is, the tracer does not actually add much to the interpreter. It linearizes control flow following the executed path, but it is not entirely clear yet, what tracing adds over interpretation. This section demonstrates how to add *promotion* (Chapter 5) to the tracer. Promotion is a crucial but simple to implement extension to the control flow language that allows the tracer to do type feedback in a way not possible for a partial evaluator.

As described in Section 5.2.1, promotion is basically a hint that the programmer can add to her control flow graph program. A promotion in the control flow graph language is an operation `promote(Arg, L)` at the end of a basic block that takes an argument `Arg` and a label `L`. Figure 77 shows the needed new rules for `interp` and `trace`. When the interpreter runs this statement, it simply jumps to the label `L` and ignores the variable.

However, the tracer does something much more interesting. For the tracer (line 148), the `promote` statement is a hint that it would be very useful to know the value of `Arg` and that the rest of the trace should keep that value as a constant. Therefore, when the tracer encounters a promotion, it inserts a guard.

The inserted `guard` is an interesting operation, because it freezes the current value `Val` of argument `Arg` into the trace. When the trace is executed, the guard checks that the current value of the variable and the frozen value are the same. If yes, execution continues, if not, the trace is aborted. This can be compared to how a guard has been used so far. Normally a guard marks a control flow decision by freezing the boolean value of the variable used in the `if` statement.

What can this operation be used for? It's a way to communicate to the tracer that argument `Arg` is not changing very often and that it is therefore useful to freeze the current value into the trace. This can be done even without knowing the value of `Arg` in advance.

Figure 78 shows a slightly contrived example. It is a loop that counts down in steps of `x * 2 + 1`, whatever `x` might be, until `i >= 0` is no longer true. Assuming that `x` doesn't change often, it is worth to

```
% l:
%     c = i >= 0
%     if c goto b else goto l_done

% l_done:
%     print_and_stop i

% b:
%     promote(x, b2)

% b2:
%     x2 = x * 2
%     x3 = x2 + 1
%     i = i - x3
%     goto l

block(l, op(c, ge, var(i), const(0),
         if(var(c), b, l_done))).
block(l_done, print_and_stop(var(i))).

block(b, promote(var(x), b2)).
block(b2, op(x2, mul, var(x), const(2),
          op(x3, add, var(x2), const(1),
          op(i, sub, var(i), var(x3),
          if(const(1), l, l))))).
```

Figure 78: A promotion example

promote it to be able to constant-fold `x * 2 + 1` to not have to redo it every iteration. This is done with the promotion of x.[4]

To trace this, we can run the following query:

```
?- do_trace(b, [i/100, x/3]).
trace
  guard(var(x),3,if(const(1),b2,b2),
  op(x2,mul,var(x),const(2),
  op(x3,add,var(x2),const(1),
  op(i,sub,var(i),var(x3),
  guard(const(1),1,if(const(1),l,l),
  op(c,ge,var(i),const(0),
  guard(var(c),1,if(const(1),l_done,l_done),
  loop)))))))
...
```

After the first `guard`, the operations performed on x could be constant-folded away, because the guard ensures that x is 3 in the rest of the trace. To actually do the constant-folding, we would need some optimization component that optimizes traces. This will be done in the next section.

Promotion is something that an ahead-of-time partial evaluator cannot do in all situation. There are weaker versions of promotion possible in a classical partial evaluator. One of them is "the trick" of partial evaluation [JGS93, page 93]. The trick is a way to get to know

[4] Of course optimizing this loop with loop invariant code motion would work as well, because x doesn't actually change during the loop.

the value of a dynamic, unknown variable that can take on n values by splitting the control flow into n paths using `if` statements. Using the trick means to turn code like this:

```
% apply trick on variable x here, which can range from 0 to 255
<code>
```

into this:

```
if x == 0:
    <copy of code>
if x == 1:
    <copy of code>
...
if x == 255:
    <copy of code>
```

When partially evaluating the second version, in each copy of the code the value of `x` is known. This only works if n is not too large, because the code after the trick is duplicated n times. The trick also doesn't work well if the variable in question holds a complex data structure.

Tracing does not actually need to do the code transformation above. Instead, when tracing the untransformed code, the right variant is picked by looking at the run-time value of `x`. Thus tracing can be seen as a lazy variant of the trick [BLR09]. Only when a run-time value is seen during tracing, the corresponding version of the code is traced. Some partial evaluators work at run-time, such as DyC [GMP$^+$00], which also supports a concept similar to promotion (called dynamic-to-static promotion).

The next section shows how to optimize traces before executing them and how the optimizer for traces is related to partial evaluation.

12.5 optimizing traces of the flow graph language

In the last two sections we saw how to produce a linear trace with guards by interpreting a control flow graph program in a special mode. A trace always ends with a `loop` statement, which jumps to the beginning. The tracer is just logging the operations that are done while interpreting, so the trace can contain superfluous operations. On the other hand, the trace also contains some of the run-time values through promotions and some decisions made on them which can be exploited by optimization. An example for this is the trace produced by the promotion example from Figure 78.

After the `guard(var(x), 3, ...)` operation, `x` is know to be `3`: If it isn't `3`, execution falls back to the interpreter. Therefore, operations on `x` after the guard can be constant-folded. To do that sort of constant-folding, an extra optimization step is needed. That optimization step walks along the trace, remembers which variables are constants and what their values are using a partial environment. The

```
optimize(op(ResultVar, Op, Arg1, Arg2, Rest), PEnv, NewTrace) :-       153
    pe_op(ResultVar, Op, Arg1, Arg2, PEnv, NEnv, NewTrace, RestTrace), 154
    optimize(Rest, NEnv, RestTrace).                                   155

optimize(loop, PEnv, T) :-                                             157
    generate_assignments(PEnv, loop, T).                               158

optimize(guard(Arg, C, CompensationCode, Rest), PEnv, NewTrace) :-     160
    presolve(Arg, PEnv, Val),                                          161
    (Val = const(C) ->                                                 162
        NewTrace = RestTrace,                                          163
        NEnv = PEnv                                                    164
    ;                                                                  165
        Val = var(V),                                                  166
        generate_assignments(                                          167
            PEnv, CompensationCode, NCompensationCode),                168
        NewTrace = guard(Arg, C, NCompensationCode, RestTrace),        169
        write_env(PEnv, V, C, NEnv)                                    170
    ),                                                                 171
    optimize(Rest, NEnv, RestTrace).                                   172

generate_assignments([], LastOp, LastOp).                              174
generate_assignments([Var/Val | Tail], LastOp,                         175
                     op(Var, assign, const(Val), const(0), T)) :-      176
    generate_assignments(Tail, LastOp, T).                             177

do_optimize(Trace, OptimizedTrace) :-                                  179
    optimize(Trace, [], OptimizedTrace).                               180
```

Figure 79: Optimizing traces

optimizer removes operations that have only constant arguments and leaves the others in the trace. This process is remarkably similar to partial evaluation: Some variables are known to be constants, operations on only constant arguments are optimized away, the rest remains.

The optimizer source code can be seen in Figure 79. It reuses the helper functions `presolve` from the partial evaluator and also uses a partial environment `PEnv`. The rule for optimizing `op` operations look exactly like those of the partial evaluator and reuses the `pe_op` helper function. When the arguments of the operation are known constants in the partial environment, the operation can be executed at optimization time and removed from the trace. Otherwise, the operation has to stay in the output trace.

Now we need to deal with guards in the trace (line 160). When the variable that is being guarded is actually known to be a constant, the guard can be removed. Note that it is not possible that the guard of that constant fails: The tracer recorded the operation while running with real values, therefore the guards *have* to succeed for values the optimizer discovers to be constant.

If the guard cannot be removed the compensation code of the guard operation needs to be changed. So far, the compensation code has always been a jump. Now it needs to be extended with a number of assignments, one for every entry in the partial environment (line 174). The reason is that the optimizer has removed operations with constant results from the trace. Since an operation is also always an assignment, the environment when running the trace will lack some values. If the interpreter is to continue execution, it needs these variables to have the correct value again, which is achieved by the assignments in the compensation code.

As an example of how `generate_assignments` assignments works, let's look at the following example. When the partial environment is `[x/5, y/10]`, the following assignments are generated:

```
?- generate_assignments([x/5, y/10], if(const(1), l, l), T).
T = op(x, assign, const(5), const(0),
       op(y, assign, const(10), const(0), if(const(1), l, l))).
```

Guards are the only way the optimizer gains entries into the partial environment, which it then exploits to do constant-folding on later operations. After the guard, the optimizer knows that if the guard has succeeded, the value of the variable is the constant frozen in the trace (line 170). This is a chief difference from partial evaluation: There, the optimizer knows the value of some variables from the start. When optimizing traces, at the beginning the value of no variable is known. Knowledge about some variables is only later gained through guards.

The only thing left to describe is what happens with the `loop` statement. In principle, it is turned into a `loop` statement again. However, as in the changes to the compensation code of guards, it is necessary to introduce a number of assignments that are executed before the next iteration of the loop can start.

With this machinery in place, we can optimize the trace from the promotion example above, see Figure 80. As intended, the operations on `x` after the `guard` have all been removed. However, some additional assignments (to `x, x2, x3`) at the end have been generated as well. The assignments look superfluous, but the optimizer does not have enough information to easily recognize this. That can be fixed, but only at the cost of additional complexity.[5]

12.5.1 *Conclusion*

In this section we have seen how to optimize traces by applying the partial evaluation principle: Perform all the operations that have only constant arguments, leave the others alone. However, optimizing traces is much simpler than partially evaluating programs because

[5] A real system would transform the trace into single static assignment form [CFR+91] to answer such questions.

```
?- do_trace(b, [i/100, x/3]).
trace: trace
  guard(var(x),3,if(const(1),b2,b2),
  op(x2,mul,var(x),const(2),
  op(x3,add,var(x2),const(1),
  op(i,sub,var(i),var(x3),
  guard(const(1),1,if(const(1),l,l),
  op(c,ge,var(i),const(0),
  guard(var(c),1,if(const(1),l_done,l_done),
  loop)))))))

opttrace
  guard(var(x),3,if(const(1),b2,b2),
  op(i,sub,var(i),const(7),
  op(c,ge,var(i),const(0),
  guard(var(c),1,
      op(x,assign,const(3),const(0),
      op(x2,assign,const(6),const(0),
      op(x3,assign,const(7),const(0),
      if(const(1),l_done,l_done)))),
  op(x,assign,const(3),const(0),
  op(x2,assign,const(6),const(0),
  op(x3,assign,const(7),const(0),
  op(c,assign,const(1),const(0),
  loop))))))))

-10
```

Figure 80: Optimizing the promotion example

no control flow is involved in the former. The partial evaluator needs to deal with control flow statements and with making sure that code is reused if the same block is partially evaluated with the same constant variables.

When optimizing traces, these complexities have already been solved. The tracer has already flattened the control flow and replaced it with guards and one `loop` operation at the end. Thus, the optimizer can simply do one pass over the operations, removing some (with some extra care around the `loop` statement).

These findings are in agreement with those in Chapter 6, which showed that a powerful allocation removal optimization can be seen as a partial evaluator on traces, also without any complex control component.

12.6 CONCLUSION

The observations made in this chapter give the opportunity to make some concluding high-level thoughts about the similarities of tracing and partial evaluation: Tracing and partial evaluation try to tackle a similar problem, namely that of automatically reducing the interpreter overhead. Their approaches are slightly different though.

Partial evaluation takes as input a program and parts of the inputs of the program. It produces a residual, more efficient program by evaluating those parts of the program that operate on the known input.

Tracing is very close to normal evaluation, only keeping some extra information in the process. But then, the optimizer that is used in a tracer is again very similar in structure to a partial evaluator. The task of the optimizer is much simpler though, because it does not need to deal with control flow at all, just a linear list of operations.

So in a sense tracing is taking those parts of partial evaluation that work (the "just evaluate those things that you can, and leave the others") and replacing the parts that are hard (controlling unfolding) by a much more pragmatic mechanism. That mechanism observes actual execution runs of the program to choose control flow paths that are typical. At the same time, the tracer's focus is on loops, because they are where most programs spend significant amounts of time.

Another point of view of tracing is that it is a form of partial evaluation that replaces the control components of a partial evaluator with an oracle (the actual execution runs) that provide the information which paths to look at.

13 SUMMARY

This second half of the thesis described the application of meta-tracing to three interpreters on different points of the complexity scale. An interpreter for regular expressions was given as a first example. It achieves good performance compared to other regular expression implementations, which is not an entirely even comparison, because most of them support larger classes of (non-regular) languages.

At the other end of the scale is Python. PyPy's Python interpreter is an almost fully compatible implementation of a large complex imperative object oriented dynamic language. The object model of Python can be efficiently implemented using the hints that RPython's meta-tracer supports. Since RPython was designed for imperative object-oriented languages, it is not surprising that performance of Python is good. This is mirrored by other results in the PyPy project, for example for a PHP interpreter written in RPython.[1] As the performance evaluation showed the performance of the Python interpreter is between 60% and 4 times slower than that of manually written just-in-time compilers.

Pyrolog is the attempt to apply the meta-tracing technique to a logical language with very different execution semantics than Python. Even there meta-tracing yields good performance improvements. Pyrolog even beats some existing Prolog implementations for certain tasks. One conclusion from this is that Prolog is a language that could benefit from run-time compilation – which so far no Prolog implementation uses.

Chapter 12 relates partial evaluation to meta-tracing. Both have similar goals, but meta-tracing seems to work better in practice than partial evaluation. One possible explanation is that the control problem of partial evaluation is simply too hard and cannot be robustly solved with heuristics. Meta-tracing replaces control with a much simpler mechanism but retains the useful parts of partial evaluation.

[1] http://morepypy.blogspot.com/2012/07/hello-everyone.html

Part IV

RELATED WORK AND CONCLUSION

14 RELATED WORK

This chapter presents additional related work that has not already been discussed in the respective chapters.

14.1 META-TRACING

Applying a trace-based optimizer to an interpreter and adding hints to help the tracer produce better results has been tried before in the context of the DynamoRIO project [SBB$^+$03], which has been a great inspiration. They achieve the same unrolling of the interpreter loop so that the unrolled version corresponds to the loops in the user program by adding hints very similar to those of Chapter 4. However the approach is greatly hindered by the fact that they trace on the machine code level and thus have no high-level information available about the interpreter. This makes it necessary to add quite a large number of hints, because at the machine code level many pieces of information are already lost. For example, it is not really visible anymore that a bytecode string is immutable. For that reason more advanced optimizations like allocation removal would not be possible with that approach.

Yermolovich et al. [YWF09] describe the use of the Tamarin JavaScript tracing JIT as a meta-tracer for a Lua interpreter. They compile the normal Lua interpreter in C to ActionScript bytecode. Again, the interpreter is annotated with some hints that indicate the main interpreter loop to the tracer. No further hints are described in the paper. There is no comparison of the performance of their system to that of the original Lua VM in C, which makes it hard to judge the effectiveness of the approach.

SPUR [BBF$^+$10] is a tracing JIT for CIL bytecode, which is then used to trace through a JavaScript implementation written in C#. This makes it a meta-tracing JIT. SPUR does not trace through the execution of interpreters. Instead, the JavaScript implementation compiles JavaScript to CIL bytecode. That bytecode contains calls into an implementation of the JavaScript object model. Since the JavaScript implementation is not interpreter-based, it is not necessary to unroll the main interpreter loop. Instead, loops on the JavaScript are mapped to loops on the CIL level by the JavaScript-to-CIL compiler which the tracer then traces.

The JavaScript object model uses maps and inline caches to speed up operations on objects. SPUR contains two hints that can be used to influence the tracer: one to prevent tracing of a C# function and one to force unrolling of a loop.[1] The object model does not contain hints that are equivalent to promotion and elidability. A similar effect is achieved with a powerful loop-invariant code-motion optimization and inline caches that the JavaScript compiler explicitly introduces into the generated CIL code.

SPUR in general relies much more on code generation than MetaJIT. For every CIL method, several machine code versions can be generated: one that does profiling to determine common loops, one that traces the method. MetaJIT relies on interpreters to perform the same functions.

There are quite a number of approaches that try to minimally enhance interpreters to generate code at run-time without actually writing a native compiler by hand. The goal of these is to get rid of dispatch overhead of typical interpreters while retaining ease of implementation. Piumarta and Riccardi [PR98] propose to copy fragments of the interpreter together for commonly occurring bytecode sequences to reduce dispatch overhead. However, dispatching is still needed to jump between such sequences and also when non-copyable bytecodes occur. Ertl and Gregg [EG04] go further and get rid of all dispatch overhead by stitching together the concatenated sequences by patching the copied machine code. Both techniques can speed up interpreters with large dispatch overhead a lot. However they will help less if the bytecodes themselves do a lot of work (as is the case with Python [Bru09]) and the dispatch overhead is lower. On the other hand, the meta-tracing approach can do a better job by tracing inside the implementation of those bytecode and inlining common paths.

To help the performance of interpreters that have complex bytecodes, various techniques have been proposed. Examples are quickening [Bru10a] and inline caching for interpreters [Bru10b].

An early attempt at building a general environment for easily writing efficient implementations of dynamic languages is described by Wolczko et al. [WAD99]. They implement Java and Smalltalk on top of the Self VM by compiling the languages to Self. The Self JIT is good enough to optimize the compiled code very well. This approach is restricted to languages that are similar enough to Self as there were no mechanisms to control the underlying compiler.

Somewhat relatedly, the proposed "invokedynamic" bytecode [Ros09] added to the JVM is supposed to make the implementation of dynamic languages on top of JVMs easier. The bytecode gives the user access to generalized inline caches. It requires of course compilation to JVM bytecode instead of writing an interpreter.

The traditional approach for automatically producing a compiler for a programming language given an interpreter for it is that of

[1] MetaJIT has equivalent hints, but they are beyond the scope of this thesis

partial evaluation [Fut99, JGS93]. There are conceptual similarities to meta-tracing, many of which have been explored in Chapter 12. In partial evaluation some arguments of the interpreter function are known (static) while the rest are unknown (dynamic). This separation of arguments is related to MetaJIT's separation of variables into those that should be part of the position key and the rest.

Classical partial evaluation has failed to be useful for dynamic languages for many of the same reasons why ahead-of-time compilers cannot compile them to efficient code. If the partial evaluator knows only the program, it simply does not have enough information to produce good code. Therefore some work has been done to do partial evaluation at run-time. One of the earliest works on run-time specialization is Tempo for C [CN96, CHN+96]. However, it is essentially a normal partial evaluator "packaged as a library"; decisions about what can be specialized and how are pre-determined. Another work in this direction is DyC [GMP+00], another run-time specializer for C. Both of these projects have a problem similar to that of DynamoRIO. Targeting the C language makes higher-level specialization difficult.

There have been some attempts to do *dynamic partial evaluation*, which defers partial evaluation completely to run-time to make it more useful for dynamic languages. This concept was introduced by Sullivan [Sul01] who implemented it for a small dynamic language based on lambda-calculus. It is also again related to Psyco [Rig04]. I have also explored dynamic partial evaluation for Prolog [Bol08]. There also were experiments within the PyPy project to use dynamic partial evaluation for automatically generating JIT compilers out of interpreters [RP07, Cun10]. Those have not been as successful as was hoped, so they were supplanted with the work on tracing JITs described here.

Another work of run-time partial evaluation was done within the context of the Jikes RVM by Shankar et al. [SSBS05]. Their partial evaluator is integrated into the Jikes JIT compiler and fully transparent to the user. It uses heap analysis at run-time to identify values on the heap that are constant and uses that information to generate specialized code. The generated code is invalidated if the objects that it depends on get mutated. The heap analysis makes it possible to use their approach without any annotations from the user and still allows them to optimize interpreters well.

14.2 RUN-TIME FEEDBACK

Promotion was also already used in other contexts, such as in earlier versions of PyPy's JIT as well as in a Prolog partial evaluator [BLR09]. Promotion is also heavily used by Psyco [Rig04] (promotion is called "unlifting" in this paper). Promotion is quite similar to run-time type feedback (and also inline caching) techniques which were first used in Smalltalk [DS84] and Self [HU94] implementations. Promotion is more general because any information can be fed back into compilation, not just types.

The approach of rewriting interpreters to make them behave specially after partial evaluation is quite common. It has been called the *interpretive approach* [GJ94]. As an example using imperative languages, Debois [Deb08] shows how to achieve strength reduction and loop-invariant code motion by rewriting an interpreter and then applying partial evaluation to it. As in Chapter 10, the rewrites still express the semantics correctly, but somewhat strangely.

14.3 ALLOCATION REMOVAL

There exists a large number of works on escape analysis, which is a program analysis that tries to find an upper bound for the lifetime of objects allocated at specific program points [GP90, PG92, CGS$^+$99, Bla03]. This information can then be used to decide that certain objects can be allocated on the stack, because their lifetime does not exceed that of the stack frame they are allocated in. The difference to the algorithm presented in Chapter 6 is that escape analysis is split into an analysis and an optimization phase. The analysis can be a lot more complex than the presented one-pass optimization. Also, stack-allocation reduces garbage-collection pressure but does not optimize away the actual accesses to the stack-allocated object. In our case, an object is not needed at all any more.[2]

Kotzmann and Mössenböck [KM05, KM07] also achieve scalar replacement using escape analysis in the Java HotSpot client compiler. If an allocated object never leaves a single function at all it can be replaced by its constituent fields. The optimization also relies on inlining to make it more likely that objects stay local to a single method. This approach still has some problems when used in the context of a dynamic language, as for most operations there exist escaping paths that are rarely executed. This defeats the escape analysis but is handled well by a tracing JIT.

Chang et al. describe a tracing JIT for JavaScript running on top of a JVM [CBY$^+$07]. They mention in passing an approach to allocation removal that moves the allocation of an object of type 1 out of the loop to only allocate it once, instead of every iteration. No details are given for this optimization. Also, the approach only removes the allocations, but not the reads out of and writes into the object.

SPUR also seems to be able to remove allocations in a similar way to the approach described here, as hinted at in their paper [BBF$^+$10]. However, no details for the approach and its implementation are given.

[2] If the tracer also has a good load-store forwarding optimization, the effect is almost the same as allocation removal.

Psyco [Rig04] also implements a more ad-hoc version of the allocation removal described here. Our static objects could be related to what are called *virtual* objects in Psyco. Earlier versions of RPython's JIT (that were not based on tracing) also used virtual objects [RP07]. Historically, MetaJIT can be seen as a successor of Psyco for a general context (Indeed, MetaJIT's source code uses the term "virtual" as well).

The original Self JIT compiler [CUE89] used an algorithm for forward-propagating the types of variables as part of its optimizations. This makes it possible to remove all but the first type checks on a variable. The optimization does not deal with removing the full object, if it is short-lived, but the type check removals are similar to what our optimization achieves.

A related optimization is also that of deforestation [Wad88, GLPJ93] which removes intermediate lists or trees in functional languages. A more general approach is boxing analysis [Jø96] which optimizes pairs of calls to box/unbox in a functional language. Similarly, "dynamic typing" [Hen94] tries to remove dynamic type coercions in a dynamically typed lambda-calculus. All these optimizations work by analyzing the program before execution, which makes them unsuitable for dynamic languages like Python, where almost nothing can be inferred purely by looking at the source code.

Partially known data structures are built directly into Prolog (via unbound logic variables) and thus the treatment of partially static data structures was part of partial evaluation of Prolog programs from the early stages [LS91]. One effect of unfolding in Prolog is that terms that are constructed and immediately matched again, completely disappear in the residual program. This is similar to what RPython's allocation removal optimization does for an imperative language. In functional programming this idea was introduced as constructor specialization by Mogensen [Mog93].

An example of an optimization that – like allocation removal – becomes much more tractable when applying it to traces is trace scheduling [FERN84]. Trace scheduling is used in the code generation stage of a static compiler to schedule the generated machine code to make use of the functional units of the CPU in a most efficient way. This is done by reordering the machine instructions. Since doing that on general control flow graphs is very hard, trace scheduling does it on execution traces. This gives the optimization linear code with side exits to work with, making the algorithms tractable. This is comparable to allocation removal, which also works well because of the linear nature of traces. It also shows that traces are in general a good approach to scale optimizations that work on basic blocks up to larger code regions.

14.4 REGULAR EXPRESSIONS

Thompson [Tho68] describes one of the first JIT compilers ever built. It takes a regular expression and turns it into IBM 7094 machine code at run-time. The algorithm has some degenerate cases where the resulting code loops endlessly, such as `a**` or `a*a*a*a*`.

Many current browsers have regular expression JITs, such as Apple Webkit[3] and Google Chrome.[4] Their engines are much more complex than the one described in Chapter 9 because they need to support Perl-style regular expression supporting groups and back-references. These are not really regular expressions in the computer-science sense as they allow the matching of non-regular languages.[5] This makes their implementation much harder and backtracking necessary.

14.5 PROLOG

Continuations have been used in various cases as the basis for implementing a Prolog system. BinProlog [Tar92, Tar12] uses a transformation to continuation-passing-style for all Prolog clauses and then uses a simplified WAM to execute those. However, it uses only a success continuation and thus doesn't make the choice points explicit. Some more work has been done to use this single success-continuation passing style for optimizations [Dem90, Neu95].

Lindgren [Lin94] proposes to use a continuation-passing style as an intermediate language before code generation. In contrast to the approaches mentioned so far, he uses both a success *and* a failure continuation, thus moving all control decisions to the source level. In the implementation of the Prolog interpreter of Chapter 11, transformation to continuation-passing style is not used as a preprocessing step. Instead, continuations are used at run-time to represent the interpreter state.

There have been a number of attempts at writing high-level object oriented Prolog interpreters. tuProlog is a Prolog implementation running on top of a Java virtual machine which was written with good object-oriented design in mind [DOR01]. It uses a state machine to execute Prolog programs [PBOR08], whose states can be related to the kinds of continuations of the interpreter presented in Section 11.2.2. Furthermore there is Prolog Café [BTI06], a Prolog to Java bytecode translator and P#, a Prolog integrated into the .NET environment with good integration with C# libraries.

Costa et al. [SCSL07] have modified YAP to perform demand-driven indexing. Their technique analyses and modifies the WAM

[3] `http://www.webkit.org/blog/214/introducing-squirrelfish-extreme/`

[4] `http://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html`

[5] As an example, the following Perl regular expression matches strings that have a non-prime length greater than one using a backreference: `(..+?)\1+`

bytecode of a predicate at run-time if it looks like the predicate could benefit from indexing on other arguments than the first. Thus they avoid heuristics or costly upfront analysis to find out on which argument indexing should be performed. This is a great example of how run-time techniques can improve the performance of Prolog systems.

15 SUMMARY AND OUTLOOK

Meta-tracing is a language implementation technique that makes it easier to efficiently implement dynamic languages without having to manually write a JIT compiler for them. Meta-tracing consists of several sub-components, presented in the first part of the thesis: The actual tracing component which traces through the execution of an interpreter, hints for user-controlled run-time feedback, and a generic allocation removal optimization.

The second part of the thesis applied meta-tracing to a regular expression engine, a Python interpreter, and a Prolog interpreter. For these languages the obtained benchmark results show that meta-tracing is useful, is easy to apply, and performance can be very good when using it. The fact that the implemented languages are on very different ends of the language spectrum supports the hypothesis that the approach works for very different language semantics.

In the future, meta-tracing should be evaluated with even more and different languages, for example a functional one like ML. This could establish to which kind of languages meta-tracing can be successfully applied. Another area of future work is to improve the warmup times of MetaJIT, which so far are rather long due to the double interpretation overhead. One option to do that would be to use the cogen approach of partial evaluation [HL91].

In conclusion, meta-tracing often makes it unnecessary to write a language specific JIT compiler. Instead, an interpreter of the language is enriched by applying some hints and maybe restructuring some parts of the interpreter to get reasonably good performance with little effort. This reduces the effort of language experimentation and makes it possible for small teams to write high-performance implementations of dynamic languages.

A

BENCHMARKING ENVIRONMENT AND SOURCE CODE REPOSITORIES

This appendix gives details of the hard- and software environment that was used to run the benchmarks, as well as pointers to the software repositories containing all source code.

Benchmarking was done on an otherwise idle Intel Core2 Duo P8400 processor with 2.26 GHz and 3072 KB of cache on a machine with 3GB RAM running in 32bit mode. Unless otherwise mentioned, all benchmarks were run 50 times within the same process. The first 10 runs are ignored to give the JIT time to warm up. The final numbers are reached by computing the average of all other runs, the confidence intervals were computed using a 95% confidence level [GBE07].

Installed and used software was:

- Ubuntu 12.04
- Linux 3.2.0-29-generic-pae #46-Ubuntu SMP
- GCC 4.6.3-1ubuntu5
- CPython 2.7.3
- Psyco 1.6 under CPython 2.6.7
- Java Hotspot 1.7.0_03
- LuaJIT Git revision 4c882fe7
- V8 SVN revision 12379
- SICStus Prolog 4.2.0 (x86-linux-glibc2.7)
- SWI-Prolog Version 5.10.4, Multi-threaded, 32 bits

A.1 PYTHON BENCHMARKS

CHAOS: A Chaosgame implementation creating a fractal.

CRYPTO_PYAES: An AES implementation.

DJANGO: The templating engine of the Django Web framework.[1]

GO: A Monte-Carlo Go AI.[2]

PYFLATE_FAST: A BZ2 decoder.

RAYTRACE_SIMPLE: A ray tracer.

RICHARDS: The Richards benchmark.

SPAMBAYES: A Bayesian spam filter.[3]

SIMPY_EXPAND: A computer algebra system.

TELCO: A Python version of the Telco decimal benchmark,[4] using a pure Python decimal floating point implementation.

TWISTED_NAMES: A DNS server benchmark using the Twisted networking framework.[5]

A.2 SOURCE CODE REPOSITORIES

All the repositories of the systems described in this thesis, including all benchmarks and the scripts to run them can be found at:
`http://cfbolz.de/phdthesis/`

[1] http://www.djangoproject.com/

[2] http://shed-skin.blogspot.com/2009/07/disco-elegant-python-go-player.html

[3] http://spambayes.sourceforge.net/

[4] http://speleotrove.com/decimal/telco.html

[5] http://twistedmatrix.com/

B

PUBLICATION HISTORY

Earlier versions of parts of this thesis have already been published in other venues.

- Chapter 4 has appeared at ICOOOLPS 2009 [BCFR09].
- Chapter 5 has appeared at ICOOOLPS 2011 [BCF$^+$11b]
- Chapter 6 has appeared at PEPM 2011 [BCF$^+$11a].
- Chapter 9 has appeared as two blog posts on the PyPy status blog[1] in May and June 2010.
- Parts of Chapter 10 were submitted to publication to Science of Computer Programming [BT12].
- Chapter 11 has appeared at PPDP 2010 [BLS10].

[1] http://morepypy.blogspot.com

LIST OF FIGURES

BIBLIOGRAPHY

[AAB+00] Bowen Alpern, C. Richard Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Mark F. Mergen, Ton Ngo, James R. Russell, Vivek Sarkar, Mauricio J. Serrano, Janice C. Shepherd, Stephen E. Smith, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeno virtual machine. *IBM Systems Journal*, 39(1):211–, 2000. (Cited on page 14.)

[AACM07] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *DLS*, Montreal, Quebec, Canada, 2007. ACM. (Cited on pages 11 and 12.)

[ABF12] Håkan Ardö, Carl Friedrich Bolz, and Maciej Fijałkowski. Loop-aware optimizations in PyPy's tracing JIT. In *DLS*, October 2012. Accepted for publication. (Cited on pages 18, 55, and 64.)

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, August 1991. (Cited on page 111.)

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003. (Cited on page 10.)

[BBF+10] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: a trace-based JIT compiler for CIL. In *OOPSLA*, Reno/Tahoe, Nevada, USA, 2010. ACM. (Cited on pages 15, 153, and 156.)

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, Ottawa, Canada, 1990. ACM. (Cited on page 12.)

[BCF+11a] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation removal by partial evaluation in a tracing JIT. In *PEPM*, Austin, Texas, USA, 2011. (Cited on page 165.)

[BCF+11b] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Runtime feedback in a meta-tracing JIT for efficient dynamic languages. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '11, page 9:1–9:8, New York, NY, USA, 2011. ACM. (Cited on page 165.)

[BCFR09] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *ICOOOLPS*, pages 18–25, Genova, Italy, 2009. ACM. (Cited on page 165.)

[BCH+96] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for Dylan. *SIGPLAN Not.*, 31(10):69–82, October 1996. (Cited on pages 9 and 96.)

[BCM04] Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley. Oil and water? high performance garbage collection in java with MMTk. *Proceedings of the 26th International Conference on Software Engineering*, page 137–146, 2004. ACM ID: 999420. (Cited on page 14.)

[BCW+10] Michael Bebenita, Mason Chang, Gregor Wagner, Andreas Gal, Christian Wimmer, and Michael Franz. Trace-based compilation in execution environments without interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pages 59–68, Vienna, Austria, 2010. ACM. (Cited on pages 14 and 15.)

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000. (Cited on page 14.)

[Bel73] James R Bell. Threaded code. *Communications of the ACM*, 16:370–372, June 1973. ACM ID: 362270. (Cited on page 90.)

[BGS99] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Load-reuse analysis: design and evaluation. *SIGPLAN Not.*, 34(5):64–76, May 1999. (Cited on page 68.)

[BKL+08] Carl Friedrich Bolz, Adrian Kuhn, Adrian Lienhard, Nicholas Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo, and Toon Verwaest. Back to the future in one week — implementing a Smalltalk VM in PyPy. In *Self-Sustaining Systems*, pages 123–139. 2008. (Cited on page 12.)

[Bla03] Bruno Blanchet. Escape analysis for Java: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, 2003. (Cited on pages 49 and 156.)

[BLR09] Carl Friedrich Bolz, Michael Leuschel, and Armin Rigo. Towards just-in-time partial evaluation of Prolog. In *Proceedings of the 19th international conference on Logic-Based Program Synthesis and Transformation*, LOPSTR'09, page 158–172, Berlin, Heidelberg, 2009. Springer-Verlag. (Cited on pages 143 and 155.)

[BLS10] Carl Friedrich Bolz, Michael Leuschel, and David Schneider. Towards a jitting VM for Prolog execution. In *PPDP*, Hagenberg, Austria, 2010. ACM. (Cited on page 165.)

[Bol08] Carl Friedrich Bolz. *Automatic JIT Compiler Generation with Runtime Partial Evaluation*. Master thesis, Heinrich-Heine-Universität Düsseldorf, 2008. (Cited on page 155.)

[BR07] Carl Friedrich Bolz and Armin Rigo. How to not write a virtual machine. In *Proceedings of the 3rd Workshop on Dynamic Languages and Applications*, 2007. (Cited on page 11.)

[Bru09] Stefan Brunthaler. Virtual-machine abstraction and optimization techniques. *Electronic Notes in Theoretical Computer Science*, 253(5):3–14, December 2009. (Cited on pages 90 and 154.)

[Bru10a] Stefan Brunthaler. Efficient interpretation using quickening. In *Proceedings of the 6th symposium on Dynamic languages*, pages 1–14, Reno/Tahoe, Nevada, USA, 2010. ACM. (Cited on page 154.)

[Bru10b] Stefan Brunthaler. Inline caching meets quickening. In *ECOOP 2010 – Object-Oriented Programming*, pages 429–451. 2010. (Cited on page 154.)

[BT12] Carl Friedrich Bolz and Laurence Tratt. The impact of meta-tracing on VM design and implementation. *Submitted to Science of Computer Programming*, March 2012. (Cited on page 165.)

[BTI06] Mutsunori Banbara, Naoyuki Tamura, and Katsumi Inoue. Prolog Cafe : A Prolog to Java translator system. In *Declarative Programming for Knowledge Management*, pages 1–11. 2006. (Cited on page 158.)

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th annual ACM SIGPLAN* *conference on Object-oriented programming, systems, languages, and applications*, pages 331–344, Vancouver, BC, Canada, 2004. ACM. (Cited on page 95.)

[BV09] Camillo Bruni and Toon Verwaest. PyGirl: generating whole-system VMs from high-level prototypes using PyPy. In Will Aalst, John Mylopoulos, Norman M Sadeh, Michael J Shaw, Clemens Szyperski, Manuel Oriol, and Bertrand Meyer, editors, *Objects, Components, Models and Patterns*, volume 33 of *Lecture Notes in Business Information Processing*, pages 328–347. Springer Berlin Heidelberg, 2009. 10.1007/978-3-642-02571-6_19. (Cited on page 12.)

[Can05] Brett Cannon. *Localized type inference of atomic types in Python*. Master thesis, California Polytechnic State University, 2005. (Cited on page 10.)

[CBY+07] Mason Chang, Michael Bebenita, Alexander Yermolovich, Andreas Gal, and Michael Franz. Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference. Technical Report ICS-TR-07-10, Donald Bren School of Information and Computer Science, University of California, Irvine, 2007. (Cited on pages 5, 14, and 156.)

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991. (Cited on pages 17 and 145.)

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. *SIGPLAN Not.*, 34(10):1–19, 1999. (Cited on page 156.)

[CHN+96] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi. A uniform approach for compile-time and run-time specialization. *Dagstuhl Seminar on Partial Evaluation*, pages 54—72, 1996. (Cited on page 155.)

[CN96] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, United States, 1996. ACM. (Cited on page 155.)

[Coi87] Pierre Cointe. Metaclasses are first class: The ObjVlisp model. *SIGPLAN Not*, 22(12):156–162, 1987. (Cited on pages 9 and 92.)

[Coo04] Jonathan J. Cook. P#: a concurrent Prolog for the .NET framework. *Softw. Pract. Exper.*, 34(9):815–845, 2004. (Cited on page 112.)

[CR96] Alain Colmerauer and Philippe Roussel. The birth of Prolog. In *History of programming languages—II*, pages 331–367. ACM, 1996. (Cited on page 111.)

[CRTR11] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How developers use the dynamic features of programming languages: the case of Smalltalk. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 23–32, New York, NY, USA, 2011. ACM. (Cited on pages 35, 40, and 45.)

[CSR$^+$09] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for Web 3.0: Trace compilation for the next generation web applications. In *VEE*, pages 71–80, Washington, DC, 2009. ACM. (Cited on page 14.)

[CT04] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 1st edition, 2004. (Cited on page 18.)

[CUE89] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *OOPSLA*, volume 24, 1989. (Cited on pages 10, 44, and 157.)

[Cun10] Antonio Cuni. *High performance implementation of Python for CLI/.NET with JIT compiler generation for dynamic languages.* PhD thesis, Dipartimento di Informatica e Scienze dell'Informazione, University of Genova, 2010. Technical Report DISI-TH-2010-05. (Cited on pages 31 and 155.)

[Deb08] Søren Debois. Imperative-program transformation by instrumented-interpreter specialization. *Higher-Order and Symbolic Computation*, 21(1):37–58, June 2008. (Cited on page 156.)

[Dem90] Bart Demoen. On the transformation of a Prolog program to a more efficient binary program. Technical Report 130, K.U. Leuven, December 1990. (Cited on page 158.)

[Die12] Lukas Diekmann. *Memory Optimizations for Data Types in Dynamic Languages.* Master thesis, Heinrich-Heine-

Universität Düsseldorf, Düsseldorf, February 2012. (Cited on page 92.)

[DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. *SIGPLAN Not.*, 33(5):106–117, May 1998. (Cited on page 68.)

[DOR01] Enrico Denti, Andrea Omicini, and Alessandro Ricci. tuProlog: a light-weight Prolog for internet applications and infrastructures. In *Practical Aspects of Declarative Languages*, pages 184–198. 2001. (Cited on page 158.)

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *POPL*, Salt Lake City, Utah, 1984. ACM. (Cited on pages 10, 97, and 156.)

[DSP$^+$09] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin Garner, David P. Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying magic: High-level low-level programming. In *VEE 2009: The 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2009. (Cited on page 14.)

[ECM99] ECMA International. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, December 1999. (Cited on pages 9 and 93.)

[ECM10] ECMA International. *Standard ECMA-335: Common Language Infrastructure (CLI)*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, December 2010. (Cited on page 10.)

[EG04] M. Anton Ertl and David Gregg. Retargeting JIT compilers by using C-compiler generated executable code. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 41–50. IEEE Computer Society, 2004. (Cited on page 154.)

[FBK05] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. MIT Press, July 2005. (Cited on page 123.)

[FERN84] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel processing: a smart compiler and a dumb machine. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, SIGPLAN '84, page 37–47, New York, NY, USA, 1984. ACM. (Cited on page 157.)

[FHW10] Sebastian Fischer, Frank Huch, and Thomas Wilke. A play on regular expressions: functional pearl. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 357–368, Baltimore, Maryland, USA, 2010. ACM. (Cited on page 75.)

[Fut99] Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. (Cited on pages 49 and 155.)

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. *SIGPLAN Notices*, 42(10):57–76, 2007. (Cited on page 163.)

[GES$^+$09] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, PLDI '09, New York, New York, 2009. ACM. ACM ID: 1542528. (Cited on pages 14 and 64.)

[GF06] Andreas Gal and Michael Franz. Incremental dynamic code generation with trace trees. Technical Report ICS-TR-06-16, Donald Bren School of Information and Computer Science, University of California, Irvine, November 2006. (Cited on pages 14 and 16.)

[GJ94] Robert Glück and Jesper Jørgensen. Generating optimizing specializers. In *Proceedings of the 1994 International Conference on Computer Languages, 1994*, pages 183 –194, May 1994. (Cited on page 156.)

[GLPJ93] Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, page 223–232, New York, NY, USA, 1993. ACM. (Cited on page 157.)

[GMP$^+$00] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000. (Cited on pages 143 and 155.)

[Gol83] Adele Goldberg. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Series in Computer Science. Addison-Wesley, 1983. (Cited on pages 9, 93, and 97.)

[Gos05] James Gosling. *The Java language specification Previous ed.: 2000*. Addison-Wesley, Boston, Mass., 2005. (Cited on pages 10 and 93.)

[GP90] B. Goldberg and Y. G. Park. Higher order escape analysis: optimizing stack allocation in functional program implementations. In *Proceedings of the third European symposium on programming on ESOP '90*, pages 152–160, Copenhagen, Denmark, 1990. Springer-Verlag New York, Inc. (Cited on pages 55 and 156.)

[GP11] Shu-yu Guo and Jens Palsberg. The essence of compiling with traces. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, page 563–574, New York, NY, USA, 2011. ACM. (Cited on page 129.)

[GPF06] Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *VEE*, Ottawa, Ontario, Canada, 2006. ACM. (Cited on pages 5, 14, and 111.)

[Gud93] David Gudeman. Representing type information in dynamically-typed languages. Technical Report TR93-27, University of Arizona at Tucson, 1993. (Cited on pages 10 and 113.)

[Hen94] Fritz Henglein. Dynamic typing: syntax and proof theory. In *Selected papers of the symposium on Fourth European symposium on programming*, page 197–230, Amsterdam, The Netherlands, The Netherlands, 1994. Elsevier Science Publishers B. V. (Cited on page 157.)

[HH09] Alex Holkner and James Harland. Evaluating the dynamic behaviour of Python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, pages 19–28, Wellington, New Zealand, 2009. Australian Computer Society, Inc. (Cited on pages 35, 40, and 45.)

[HHS05] David Hiniker, Kim Hazelwood, and Michael D Smith. Improving region selection in dynamic optimization systems. *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, page 141–154, 2005. ACM ID: 1100546. (Cited on page 16.)

[HL91] C. K. Holst and John Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming,* pages 210–218, Skye, Scotland, 1991. Glasgow University. (Cited on page 161.)

[HU94] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI*, pages 326–336, Orlando, Florida, United States, 1994. ACM. (Cited on pages 10 and 156.)

[Hö94] Urs Hölzle. Adaptive optimization for SELF: reconciling high performance with exploratory programming. Technical report, Stanford University, 1994. (Cited on pages 5 and 10.)

[IG96] Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996. (Cited on page 9.)

[IHWN11] H. Inoue, H. Hayashizaki, Peng Wu, and T. Nakatani. A trace-based Java JIT compiler retrofitted from a method-based compiler. In *2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 246–256. IEEE, April 2011. (Cited on page 14.)

[IKM$^+$97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, page 318–326, New York, NY, USA, 1997. ACM Press. (Cited on page 14.)

[ISO95] ISO. *ISO\IEC 13211-1:1995: Information technology — Programming languages — Prolog — Part 1: General core*. 1995. (Cited on page 111.)

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, 1993. (Cited on pages 39, 49, 132, 142, and 155.)

[JHM12] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Press, Boca Raton, FL, 2012. (Cited on page 14.)

[Jø96] Jesper Jørgensen. *A Calculus for Boxing Analysis of Polymorphically Typed Languages*. Ph.D., University of Copenhapen Technical Report 96/28, 1996. (Cited on page 157.)

[KC84] Kenneth M. Kahn and Mats Carlsson. How to implement Prolog on a LISP machine. In *Implementations of Prolog*, pages 117–134. 1984. (Cited on page 114.)

[KM05] Thomas Kotzmann and Hanspeter Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, page 111–120, New York, NY, USA, 2005. ACM. (Cited on pages 49 and 156.)

[KM07] Thomas Kotzmann and Hanspeter Mössenböck. Run-time support for optimizations based on escape analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, page 49–60, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 156.)

[KR94] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp Symb. Comput.*, 7(4):315–335, December 1994. (Cited on page 14.)

[LB02] Michael Leuschel and Maurice Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory Pract. Log. Program.*, 2(4-5):461–515, July 2002. (Cited on page 136.)

[Leu02] Michael Leuschel. Homeomorphic embedding for online termination of symbolic methods. In Torben Mogensen, David Schmidt, and I. Sudborough, editors, *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 379–403. Springer Berlin / Heidelberg, 2002. (Cited on page 136.)

[LH90] Serge Le Huitouze. A new data structure for implementing extensions to Prolog. In *Programming Language Implementation and Logic Programming*, pages 136–150. 1990. (Cited on page 113.)

[Lin94] Thomas Lindgren. A continuation-passing style for Prolog. In *Symposium on Logic Programming*, pages 603–617, 1994. (Cited on pages 114 and 158.)

[LS91] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *J. Log. Program.*, 11(3-4):217–242, 1991. (Cited on page 157.)

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, April 1960. (Cited on page 9.)

[Mir99] Eliot Miranda. Context management in VisualWorks 5i. Technical report, ParcPlace Division, CINCOM, Inc., 1999. (Cited on page 64.)

[MM97] Steven S. Muchnick and Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, September 1997. (Cited on page 18.)

[Mog93] Torben Æ Mogensen. Constructor specialization. In *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 22–32, Copenhagen, Denmark, 1993. ACM. (Cited on page 157.)

[Neu95] Ulrich Neumerkel. Continuation Prolog: A new intermediary language for WAM and BinWAM code generation. *Post-ILPS'95 Workshop on Implementation of Logic Programming Languages. F16G*, 1995. (Cited on pages 114 and 158.)

[PBOR08] Giulio Piancastelli, Alex Benini, Andrea Omicini, and Alessandro Ricci. The architecture and design of a malleable object-oriented Prolog engine. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 191–197, Fortaleza, Ceara, Brazil, 2008. ACM. (Cited on pages 112 and 158.)

[PG92] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. *SIGPLAN Not.*, 27(7):116–127, 1992. (Cited on page 156.)

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. *SIGPLAN Not.*, 33(5):291–300, 1998. (Cited on page 154.)

[Pro95] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, page 322–332, New York, NY, USA, 1995. ACM. (Cited on page 90.)

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium on Java Virtual Machine Research and Technology Symposium - Volume 1*, Monterey, California, 2001. USENIX Association. (Cited on page 10.)

[Rig04] Armin Rigo. Representation-based just-in-time specialization and the Psyco prototype for Python. In *PEPM*, Verona, Italy, 2004. ACM. (Cited on pages 10, 98, 155, and 157.)

[RLBV10] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–12, Toronto, Ontario, Canada, 2010. ACM. (Cited on pages 35 and 40.)

[Ros09] John R Rose. Bytecodes meet combinators: invokedynamic on the JVM. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, VMIL '09, 2009. (Cited on pages 11 and 154.)

[RP06] Armin Rigo and Samuele Pedroni. PyPy's approach to virtual machine construction. In *DLS*, Portland, Oregon, USA, 2006. ACM. (Cited on pages 11 and 14.)

[RP07] Armin Rigo and Samuele Pedroni. JIT compiler architecture. Technical Report D08.2, PyPy, May 2007. (Cited on pages 155 and 157.)

[RS59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114 –125, April 1959. (Cited on page 75.)

[Sal04] Michael Salib. *Starkiller: a static type inferencer and compiler for Python*. Master thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2004. (Cited on page 10.)

[SB12] David Schneider and Carl Friedrich Bolz. The efficient handling of guards in the design of RPython's tracing JIT. In *VMIL, accepted for publication*, 2012. (Cited on pages 53 and 64.)

[SBB$^+$03] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic native optimization of interpreters. In *Workshop on Interpreters, virtual machines and emulators*, San Diego, California, 2003. ACM. (Cited on page 153.)

[SC11] Amin Shali and William R. Cook. Hybrid partial evaluation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, page 375–390, New York, NY, USA, 2011. ACM. (Cited on page 135.)

[Sch11] David Schneider. *An ARM Backend for PyPy's Tracing JIT*. Master thesis, Heinrich-Heine-Universität Düsseldorf, Düsseldorf, March 2011. (Cited on page 32.)

[SCSL07] Vítor Santos Costa, Konstantinos Sagonas, and Ricardo Lopes. Demand-driven indexing of Prolog clauses. In *Logic Programming*, pages 395–409, 2007. (Cited on page 158.)

[SJG93] Guy L. Steele Jr. and Richard P. Gabriel. The evolution of Lisp. In *The second ACM SIGPLAN conference on History of programming languages*, HOPL-II, page 231–270, New York, NY, USA, 1993. ACM. (Cited on page 10.)

[SS02] Bernard Paul Serpette and Manuel Serrano. Compiling Scheme to JVM bytecode: a performance study. *SIGPLAN Not.*, 37(9):259–270, 2002. (Cited on page 11.)

[SSBS05] Ajeet Shankar, S. Subramanya Sastry, Rastislav Bodík, and James E. Smith. Runtime specialization with optimistic heap analysis. *SIGPLAN Not.*, 40(10):327–343, 2005. (Cited on page 155.)

[Sul01] Gregory T. Sullivan. Dynamic partial evaluation. In *Proceedings of the Second Symposium on Programs as Data Objects*, pages 238–256. Springer-Verlag, 2001. (Cited on page 155.)

[Tar92] Paul Tarau. BinProlog: a continuation passing style Prolog engine. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science, page 479–480. Springer, August 1992. poster. (Cited on pages 114 and 158.)

[Tar12] Paul Tarau. The BinProlog experience: Architecture and implementation choices for continuation passing Prolog and first-class logic engines. *Theory and Practice of Logic Programming*, 12(Special Issue 1-2):97–126, 2012. (Cited on pages 114 and 158.)

[Tho68] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. (Cited on page 158.)

[Tra09] Laurence Tratt. Dynamically typed languages. In Marvin V. Zelkowitz, editor, *Advances in Computers*, volume 77 chapter 5, pages 149–184. Elsevier, 2009. (Cited on pages 5 and 9.)

[TS98] Simon Taylor and Zoltan Somogyi. Optimization of Mercury programs. Technical report, University of Melbourne, November 1998. (Cited on page 119.)

[US87] David Ungar and Randall B Smith. Self: The power of simplicity. *ACM SIGPLAN Notices*, 22:227–242, December 1987. ACM ID: 38828. (Cited on pages 9 and 93.)

[vR94] Peter van Roy. 1983-1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming*, 19:385–441, 1994. (Cited on page 111.)

[Wad88] Philip Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, page 231–248, Amsterdam, The Netherlands, The Netherlands, 1988. North-Holland Publishing Co. (Cited on page 157.)

[WAD99] Mario Wolczko, Ole Agesen, and David Ungar. Towards a universal implementation substrate for object-oriented languages. In *OOPSLA workshop on Simplicity, Performance, and Portability in Virtual Machine Design*, 1999. (Cited on page 154.)

[War83] D. H. D. Warren. An abstract Prolog instruction set. Technical report, SRI International, 1983. (Cited on page 111.)

[WHIN11] Peng Wu, Hiroshige Hayashizaki, Hiroshi Inoue, and Toshio Nakatani. Reducing trace selection footprint for large-scale Java applications without performance loss. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, page 789–804, New York, NY, USA, 2011. ACM. (Cited on page 16.)

[YWF09] Alexander Yermolovich, Christian Wimmer, and Michael Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. In *DLS*, pages 79–88, Orlando, Florida, USA, 2009. ACM. (Cited on page 153.)