PyPy

How to not write Virtual Machines for Dynamic Languages

Carl Friedrich Bolz

Institut für Informatik
Heinrich-Heine-Universität Düsseldorf

Dynamische Programmiersprachen, 18. Januar 2012

Scope

This talk is about:

- implementing dynamic languages
  (with a focus on complicated ones)
- in a context of limited resources
  (academic, open source, or domain-specific)

Scope

This talk is about:

- implementing dynamic languages
  (with a focus on complicated ones)
- in a context of limited resources
  (academic, open source, or domain-specific)

Our points

- Do not write virtual machines “by hand”
- Instead, write interpreters in high-level languages
- Meta-programming is your friend

Part 1:

The Why and What of PyPy

Common Approaches to Language Implementation

Using C/C++ (potentially disguised as another language)

- CPython
- Ruby
- Most JavaScript VMs
- but also: Scheme48, Squeak

Common Approaches to Language Implementation

Using C/C++ (potentially disguised as another language)

- CPython
- Ruby
- Most JavaScript VMs
- but also: Scheme48, Squeak

Building on top of a general-purpose OO VM

- Jython, IronPython
- JRuby, IronRuby
- various Prolog, Lisp, even Smalltalk implementations

Implementing VMs in C

When writing a VM in C it is hard to reconcile:

- flexibility, maintainability
- simplicity
- performance

Implementing VMs in C

When writing a VM in C it is hard to reconcile:

- flexibility, maintainability
- simplicity
- performance

Python Case

- CPython is a very simple bytecode VM, performance not great
- Psyco is a just-in-time-specializer, very complex, hard to maintain, but good performance
- Stackless is a fork of CPython adding microthreads. It was never incorporated into CPython for complexity reasons

Fixing of Early Design Decisions

- when starting a VM in C, many design decisions need to be made upfront
- examples: memory management technique, threading model
- such decisions are manifested throughout the VM source
- very hard to change later

Fixing of Early Design Decisions

- when starting a VM in C, many design decisions need to be made upfront
- examples: memory management technique, threading model
- such decisions are manifested throughout the VM source
- very hard to change later

Python Case

- CPython uses reference counting, increfs and decrefs everywhere
- CPython uses OS threads with one global lock, hard to change to lightweight threads or finer locking

Compilers are a bad encoding of Semantics

- to reach good performance levels, dynamic compilation is often needed
- a compiler (obviously) needs to encode language semantics
- this encoding is often obscure and hard to change

Compilers are a bad encoding of Semantics

- to reach good performance levels, dynamic compilation is often needed
- a compiler (obviously) needs to encode language semantics
- this encoding is often obscure and hard to change

Python Case

- Psyco is a dynamic compiler for Python
- synchronizing with CPython's rapid development is a lot of effort
- many of CPython's new features not supported well
- not ported to 64-bit machines, and probably never will
- development stopped

Implementing Languages on Top of OO VMs

- users wish to have easy interoperation with the general-purpose OO VMs used by the industry (JVM, CLR)
- therefore re-implementations of the language on the OO VMs are started
- more implementations!
- implementing on top of an OO VM has its own set of benefits of problems

Implementing Languages on Top of OO VMs

- users wish to have easy interoperation with the general-purpose OO VMs used by the industry (JVM, CLR)
- therefore re-implementations of the language on the OO VMs are started
- more implementations!
- implementing on top of an OO VM has its own set of benefits of problems

Python Case

- Jython is a Python-to-Java-bytecode compiler
- IronPython is a Python-to-CLR-bytecode compiler
- both are slightly incompatible with the newest CPython version

Benefits of implementing on top of OO VMs

- higher level of implementation
- the VM supplies a GC and mostly a JIT
- better interoperability than what the C level provides

Benefits of implementing on top of OO VMs

- higher level of implementation
- the VM supplies a GC and mostly a JIT
- better interoperability than what the C level provides

Python Case

- both Jython and IronPython integrate well with their host OO VM
- both have free threading

The problems of OO VMs

- most immediate problem: it can be hard to map concepts of the dynamic language to the host OO VM
- performance is often not improved, and can be very bad, because of the semantic mismatch between the dynamic language and the host VM

The problems of OO VMs

- most immediate problem: it can be hard to map concepts of the dynamic language to the host OO VM
- performance is often not improved, and can be very bad, because of the semantic mismatch between the dynamic language and the host VM

Python Case

- both Jython and IronPython are quite a bit slower than CPython
- IronPython misses some stack introspection features
- Python has very different semantics for method calls than Java

The problems of OO VMs

- most immediate problem: it can be hard to map concepts of the dynamic language to the host OO VM
- performance is often not improved, and can be very bad, because of the semantic mismatch between the dynamic language and the host VM

Python Case

- both Jython and IronPython are quite a bit slower than CPython
- IronPython misses some stack introspection features
- Python has very different semantics for method calls than Java
- For languages like Prolog it is even harder to map the concepts

The Future of OO VMs?

- The problems described might improve in the future
- JVM consider adding extra support for more languages
- i.e. tail calls, `InvokeDynamic`, ...
- has not really landed yet
- getting good performance needs a huge amount of tweaking

The Future of OO VMs?

- The problems described might improve in the future
- JVM consider adding extra support for more languages
- i.e. tail calls, `InvokeDynamic`, ...
- has not really landed yet
- getting good performance needs a huge amount of tweaking
- But: Microsoft seems to have stopped developing their VM

The Future of OO VMs?

- The problems described might improve in the future
- JVM consider adding extra support for more languages
- i.e. tail calls, `InvokeDynamic`, ...
- has not really landed yet
- getting good performance needs a huge amount of tweaking
- But: Microsoft seems to have stopped developing their VM

Ruby Case

- JRuby tries really hard to be a very good implementations
- can be very fast on the JVM
- took an enormous amount of effort

PyPy's Approach to VM Construction

Goal: achieve flexibility, simplicity and performance together

- Approach: auto-generate VMs from high-level descriptions of the language
- ... using meta-programming techniques and aspects
- high-level description: an interpreter written in a high-level language
- ... which we translate (i.e. compile) to a VM running in various target environments, like C/Posix (and CLR, JVM)

PyPy

- PyPy = Python interpreter written in RPython + translation toolchain for RPython

PyPy

- PyPy = Python interpreter written in RPython + translation toolchain for RPython

What is RPython

- RPython is a subset of Python
- subset chosen in such a way that type-inference can be performed
- still a high-level language (unlike SLang or PreScheme)
- ...really a subset, can't give a small example of code that doesn't just look like Python :-)

Auto-generating VMs

- we need a custom translation toolchain to compile the interpreter to a full VM
- many aspects of the final VM are orthogonal from the interpreter source: they are inserted during translation

Auto-generating VMs

- we need a custom translation toolchain to compile the interpreter to a full VM
- many aspects of the final VM are orthogonal from the interpreter source: they are inserted during translation

Examples

- Garbage Collection strategy
- Threading models (e.g. coroutines with CPS...)
- non-trivial translation aspect: auto-generating a dynamic compiler from the interpreter, third part of the talk

Good Points of the Approach

Simplicity:

- dynamic languages can be implemented in a high level language
- separation of language semantics from low-level details
- a single-source-fits-all interpreter
  - runs everywhere with the same semantics
  - no outdated implementations, no ties to any standard platform
  - less duplication of efforts

Good Points of the Approach

Simplicity:

- dynamic languages can be implemented in a high level language
- separation of language semantics from low-level details
- a single-source-fits-all interpreter
  - runs everywhere with the same semantics
  - no outdated implementations, no ties to any standard platform
  - less duplication of efforts

PyPy

arguably the most readable Python implementation so far

Good Points of the Approach

Flexibility at all levels:

- when writing the interpreter (high-level languages rule!)
- when adapting the translation toolchain as necessary
- to break abstraction barriers when necessary

Good Points of the Approach

Flexibility at all levels:

- when writing the interpreter (high-level languages rule!)
- when adapting the translation toolchain as necessary
- to break abstraction barriers when necessary

Example

- boxed integer objects, represented as tagged pointers
- manual system-level RPython code

Good Points of the Approach

Performance:

- “reasonable” performance
- can generate a dynamic compiler from the interpreter (work in progress, 10x faster on some Python code)

Good Points of the Approach

Performance:

- “reasonable” performance
- can generate a dynamic compiler from the interpreter (work in progress, 10x faster on some Python code)

JIT compiler generator

- almost orthogonal from the interpreter source – applicable to many languages, follows language evolution “for free”
- based on Partial Evaluation techniques
- benefits from a high-level interpreter
- starting to get real fast, and applicable for real world, big applications

Drawbacks / Open Issues / Further Work

- writing the translation toolchain in the first place takes lots of effort (but it can be reused)
- writing a good GC was still necessary, not perfect yet (i.e. not multi-threaded)
- dynamic compiler generation seems to work, but took very long to get right

Conclusion / Meta-Points

- VMs shouldn't be written by hand at a low level
- high-level languages are suitable to implement dynamic languages
- doing so has many benefits
- PyPy's concrete approach is not so important
- it's just one point in a large design space

Conclusion / Meta-Points

- VMs shouldn't be written by hand at a low level
- high-level languages are suitable to implement dynamic languages
- doing so has many benefits
- PyPy's concrete approach is not so important
- it's just one point in a large design space
- let's write more meta-programming toolchains!

Questions about Part 1?

PyPy

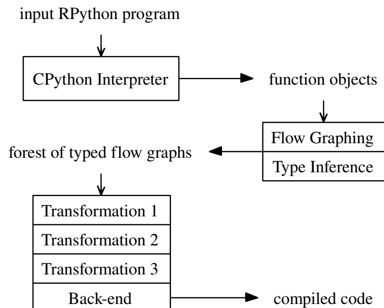
`http://pypy.org`

Demo

Part 2:

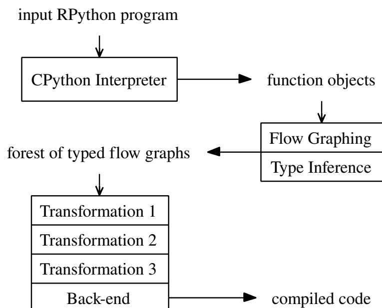
Technical Details of the Translation Toolchain

Translation Steps



- Generation of a Control Flow Graph
- Type Inference
- Abstraction-Lowering Transformations
- Optimizations
- Code-Generation by the backend

Translation Steps

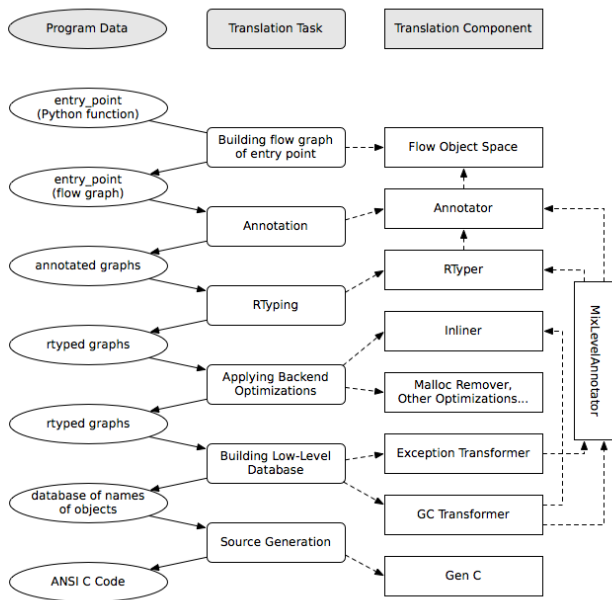


- Generation of a Control Flow Graph
- Type Inference
- Abstraction-Lowering Transformations
- Optimizations
- Code-Generation by the backend

Backends

- Low-level Backend: C
- High-level (object-oriented) backends: Java, .NET

Detailed Overview



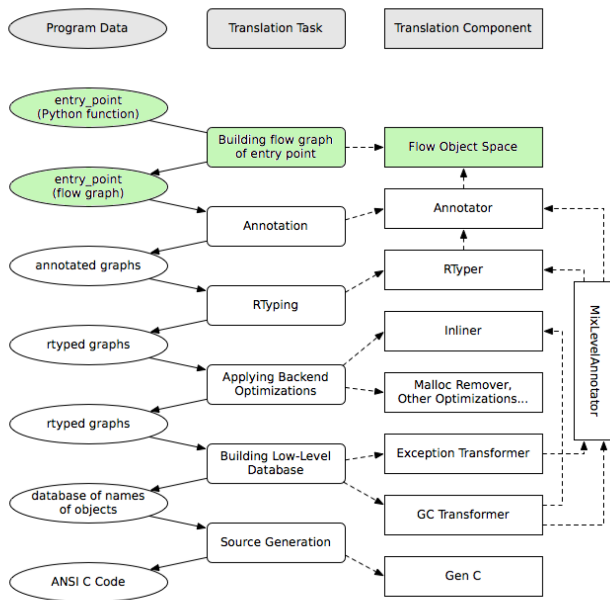
Running Example

```
def is_prime(n, primes):
    for num in primes:
        if n   num == 0:
            return False
    return True



def primes(n):
    primes = [2]
    for i in range(3, n + 1, 2):
        if is_prime(i, primes):
            primes.append(i)
    return primes
```

Control Flow Graph



Generation of the Control Flow Graph

- starts from Python-Bytecode
- result: CFG, made out of basic blocks and links
- "Static Single Information" form (SSI)

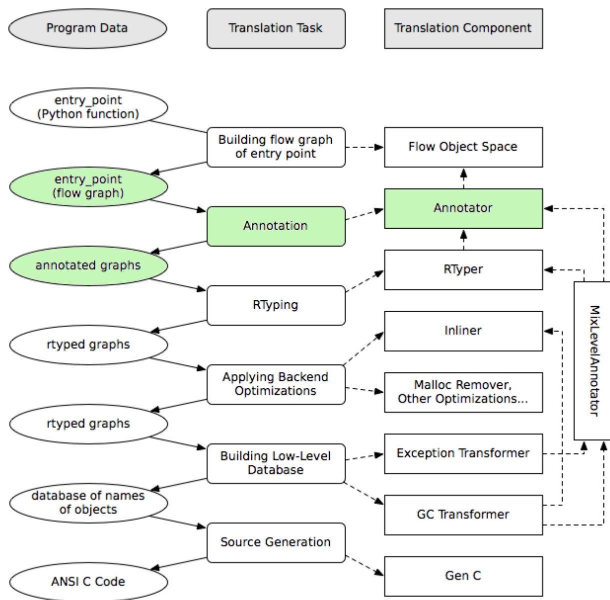
Generation of the Control Flow Graph

- starts from Python-Bytecode
- result: CFG, made out of basic blocks and links
- "Static Single Information" form (SSI)

SSI

- similar to "Static Single Assignment"-form (SSA)
- most modern Compilers use SSA
- SSA: every variable is assigned to only once
- SSI: every variable is used in exactly one block

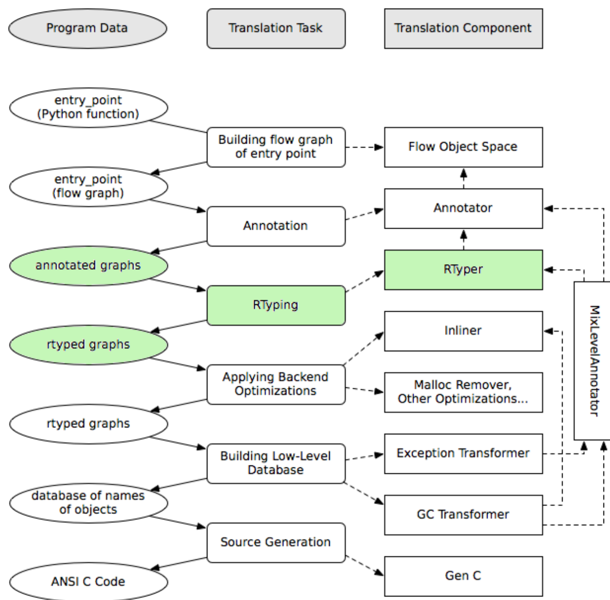
Type Inference



Type Inference

- Python uses dynamic typing
- no type information in the source code
- automatic inference of the type
- can't do bottom-up like Hindley-Milner (ML)
- instead: forward-progagation
- starting from the main function

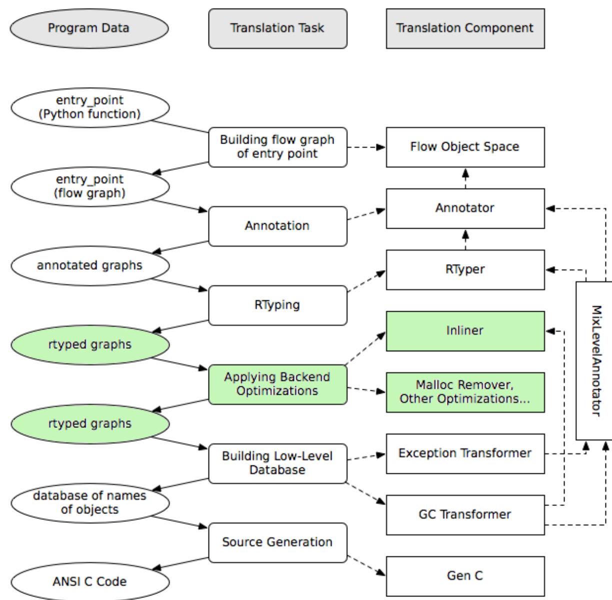
Lowering Of the Abstraction Level



Lowering Of the Abstraction Level

- operations in the graph not supported by target languages
- need to transform graphs
- before the transformation: Python type system
- afterwards: C type system
- primitive Operations in Python become helper function calls

Optimizations



Optimizations

Inlining

- function calls are expensive
- inlining gives more context for other optimizations
- Problem: can make program much larger

Optimizations

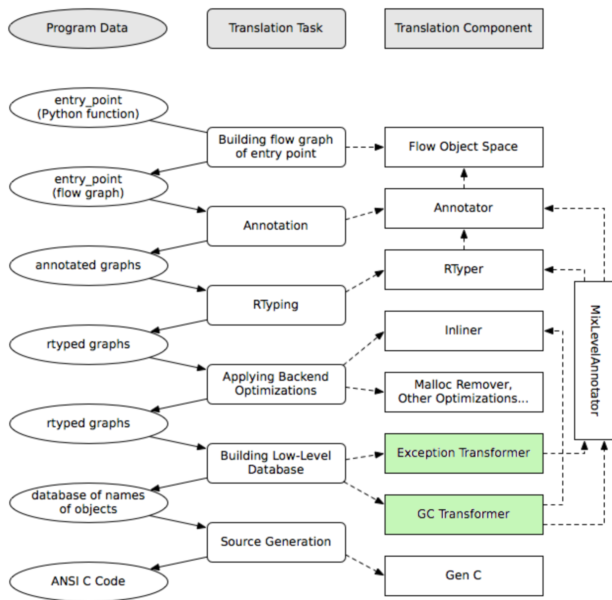
Inlining

- function calls are expensive
- inlining gives more context for other optimizations
- Problem: can make program much larger

Allocation Removal

- memory management takes a lot of time at runtime
- particularly in an object-oriented language
- many short-lived helper objects
- idea: get rid of those helper objects

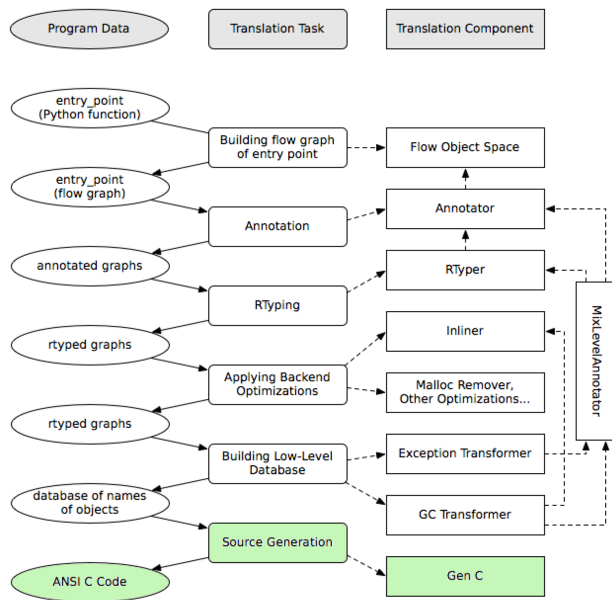
Exceptions and Memory Management



Exceptions and Memory Management

- C does not support exceptions and memory management
- have to be made explicit in the generated Code
- easy for exceptions
- for memory management more complex
- need to insert a garbage collector

Code Generation



Code Generation

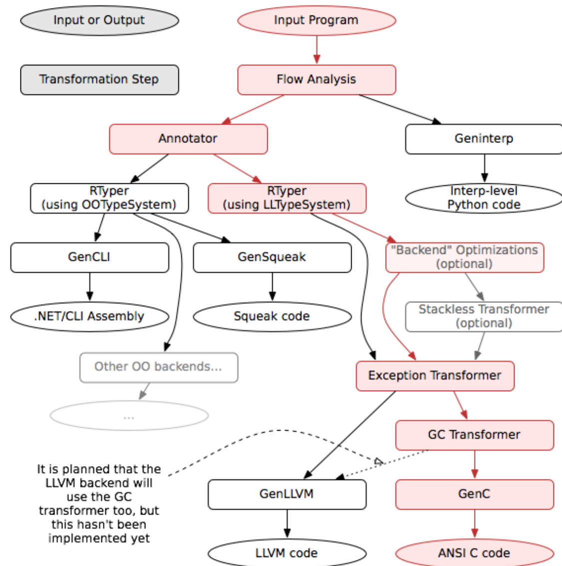
boring

Code Generation

boring

- generate one C function per CFG
- all remaining constructs can be mapped to C easily

Presented Parts of PyPy



Part 3:

PyPy's Tracing JIT Compiler

Motivation

- writing good JIT compilers for dynamic programming languages is hard and error-prone
- tracing JIT compilers are a new approach to JITs that are supposed to be easier

Tracing JIT Compilers

- idea from Dynamo project: dynamic rewriting of machine code
- later used for a lightweight Java JIT
- seems to also work for dynamic languages (see TraceMonkey)

Tracing JIT Compilers

- idea from Dynamo project: dynamic rewriting of machine code
- later used for a lightweight Java JIT
- seems to also work for dynamic languages (see TraceMonkey)

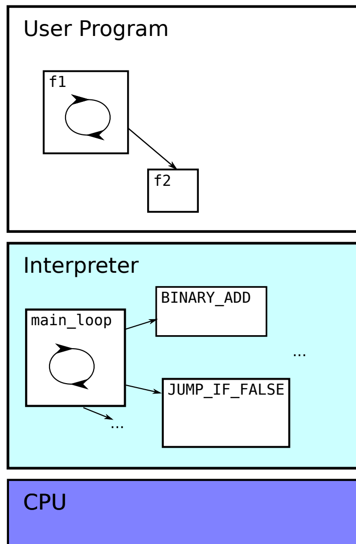
Basic Assumption of a Tracing JIT

- programs spend most of their time executing loops
- several iterations of a loop are likely to take similar code paths

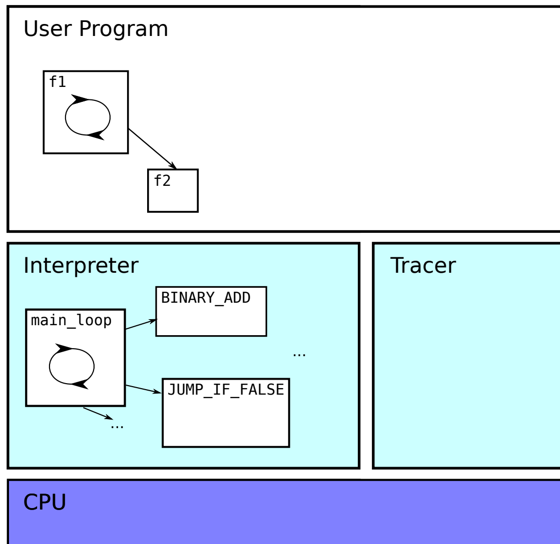
Tracing JIT Compilers

- mixed-mode execution environment
- at first, everything is interpreted
- lightweight profiling to discover hot loops
- code generation only for common paths of hot loops
- when a hot loop is discovered, start to produce a trace

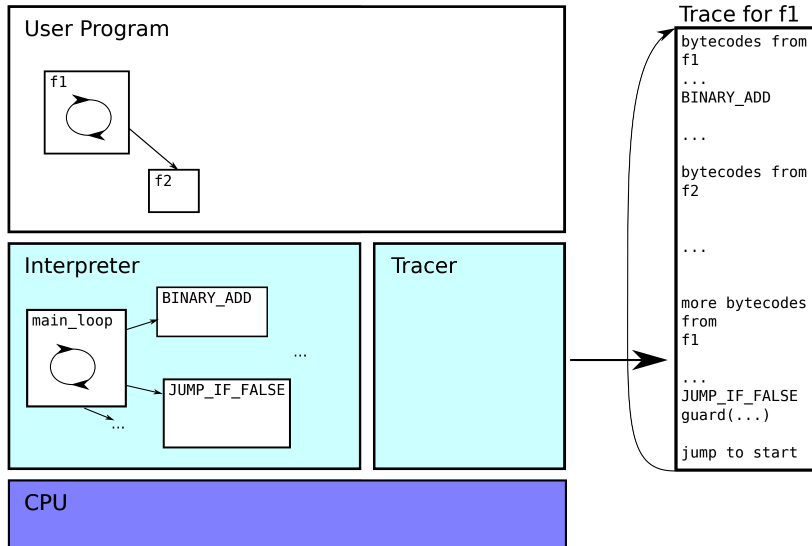
An Interpreter



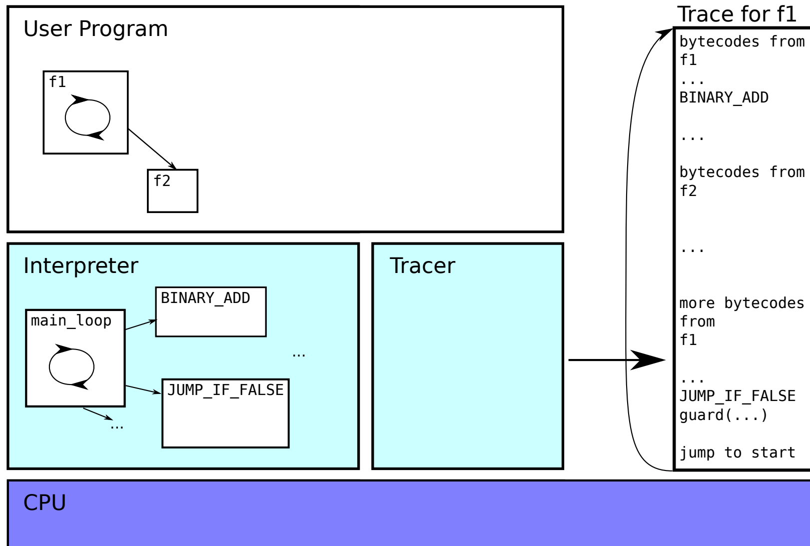
A Tracing JIT



A Tracing JIT



A Tracing JIT



Tracing

- a trace is a sequential list of operations
- a trace is produced by recording every operation the interpreter executes
- tracing ends when the tracer sees a position in the program it has seen before
- to identify these places, the position key is used
- the position key encodes the current point of execution
- a trace thus corresponds to exactly one loop
- that means it ends with a jump to its beginning

Tracing

- a trace is a sequential list of operations
- a trace is produced by recording every operation the interpreter executes
- tracing ends when the tracer sees a position in the program it has seen before
- to identify these places, the position key is used
- the position key encodes the current point of execution
- a trace thus corresponds to exactly one loop
- that means it ends with a jump to its beginning

Guards

- the trace is only one of the possible code paths through the loop
- at places where the path could diverge, a guard is placed

Code Generation and Execution

- being linear, the trace can easily be turned into machine code
- the machine code can be immediately executed
- execution stops when a guard fails
- after a guard failure, go back to interpreting program

Example

```
1  def strange_sum(n):
2      result = 0
3      while n >= 0:
4          result = f(result, n)
5          n -= 1
6      return result
7
8  def f(a, b):
9      if b  46 == 41:
10         return a - b
11     else:
12         return a + b
```

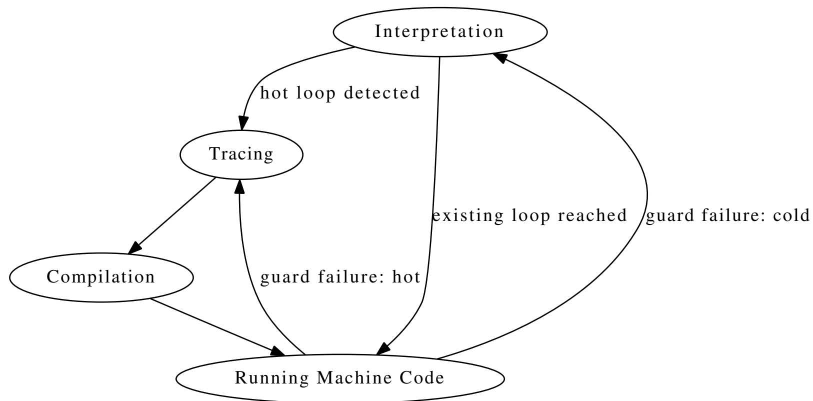
Example

```
1  def strange_sum(n):
2      result = 0
3      while n >= 0:
4          result = f(result, n)
5          n -= 1
6      return result
7
8  def f(a, b):
9      if b % 46 == 41:
10         return a - b
11     else:
12         return a + b
13
14 # trace:
15 # loop_header(result0, n0)
16 # i0 = int_mod(n0, Const(46))
17 # i1 = int_eq(i0, Const(41))
18 # guard_false(i1)
19 # result1 = int_add(result0, n0)
20 # n1 = int_sub(n0, Const(1))
21 # i2 = int_ge(n1, Const(0))
22 # guard_true(i2)
23 # jump(result1, n1)
```

Dealing With Control Flow

- An if statement in a loop is turned into a guard
- if that guard fails often, things are inefficient
- solution: attach a new trace to a guard, if it fails often enough
- new trace can lead back to same loop
- or to some other loop

Stages of Execution



(Dis-)Advantages of Tracing JITs

Good Points of the Approach

- easy and fast machine code generation: needs so support only one path
- interpreter does a lot of the work
- can be added to an existing interpreter unobtrusively
- automatic inlining
- produces very little code

(Dis-)Advantages of Tracing JITs

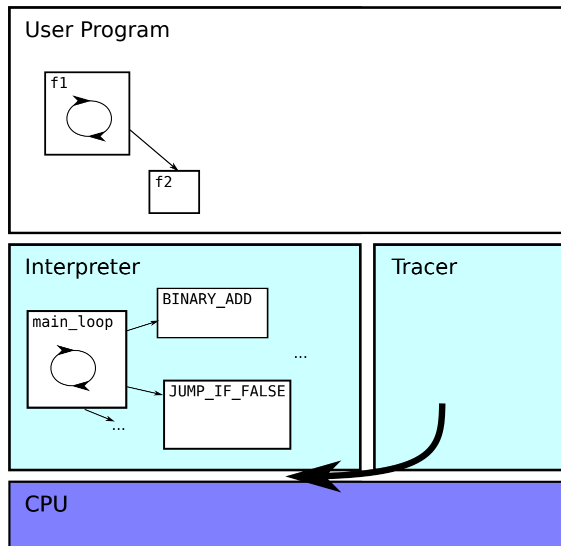
Good Points of the Approach

- easy and fast machine code generation: needs so support only one path
- interpreter does a lot of the work
- can be added to an existing interpreter unobtrusively
- automatic inlining
- produces very little code

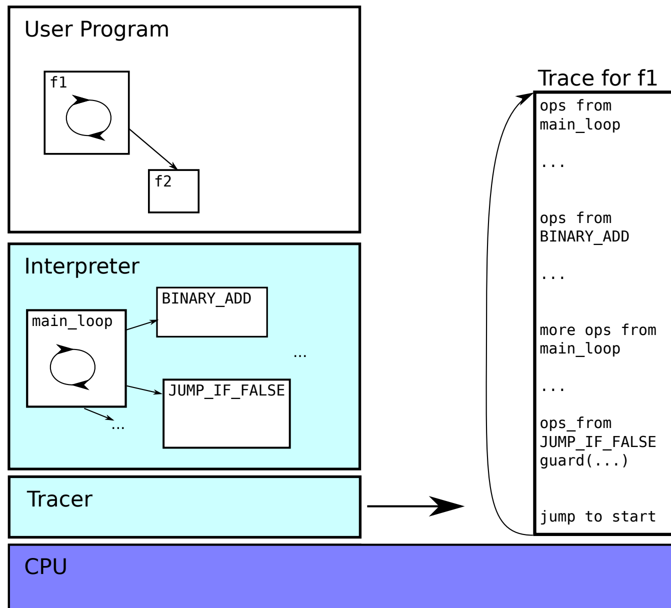
Bad Points of the Approach

- unclear whether assumptions are true often enough
- switching between interpretation and machine code execution takes time
- problems with really complex control flow

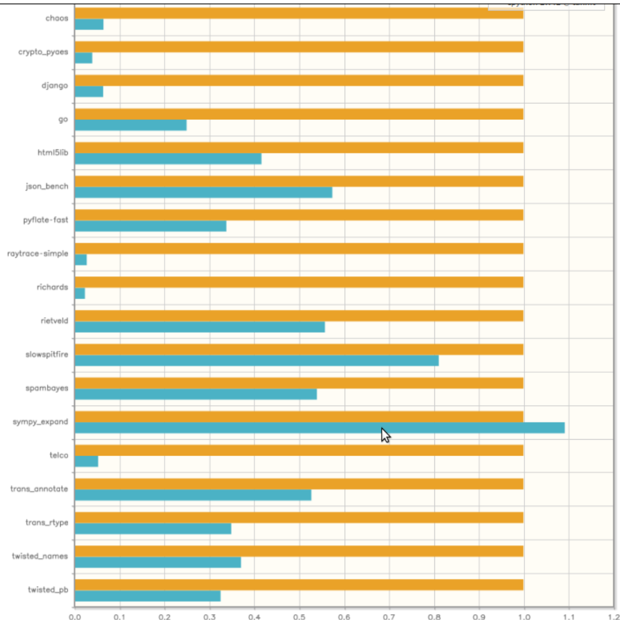
Idea of Meta-Tracing



Meta-Tracing



Benchmarks



Telco Benchmark

telco: A small program which is intended to capture the essence of a telephone company billing application, with a realistic balance between Input/Output activity and application calculations.