PyPy

How to not write Virtual Machines for Dynamic Languages

Carl Friedrich Bolz

Institut für Informatik
Heinrich-Heine-Universität Düsseldorf

Vorlesung Dynamische Programmiersprachen, 14. Juli 2010

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

Implementation Proliferation

- restrictions of the original implementation lead to re-implementations, forks
- all implementations need to be synchronized with language evolution
- lots of duplicate effort, compatibility problems

Implementation Proliferation

- restrictions of the original implementation lead to re-implementations, forks
- all implementations need to be synchronized with language evolution
- lots of duplicate effort, compatibility problems

Python Case

- several serious implementations: CPython, Stackless, Psyco, Jython, IronPython, PyPy
- various incompatibilities

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, 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, second 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
- covered by the second part of the talk

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
- 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
- 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

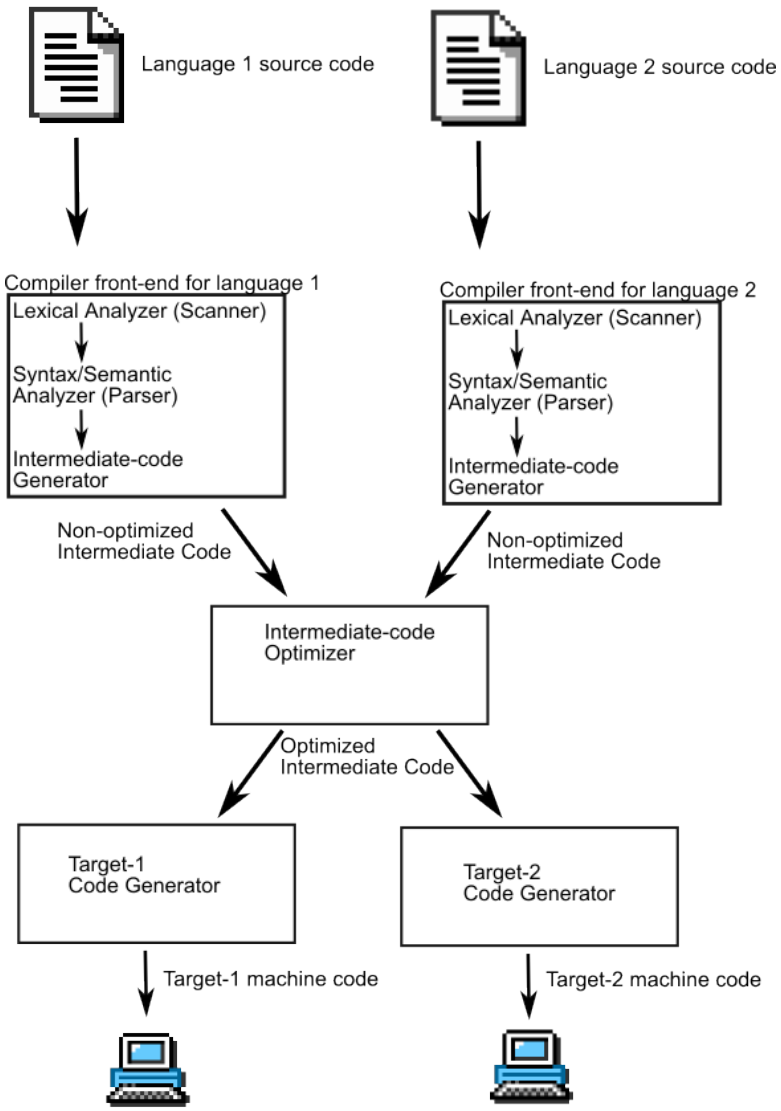
`http://codespeak.net/pypy/`

Demo

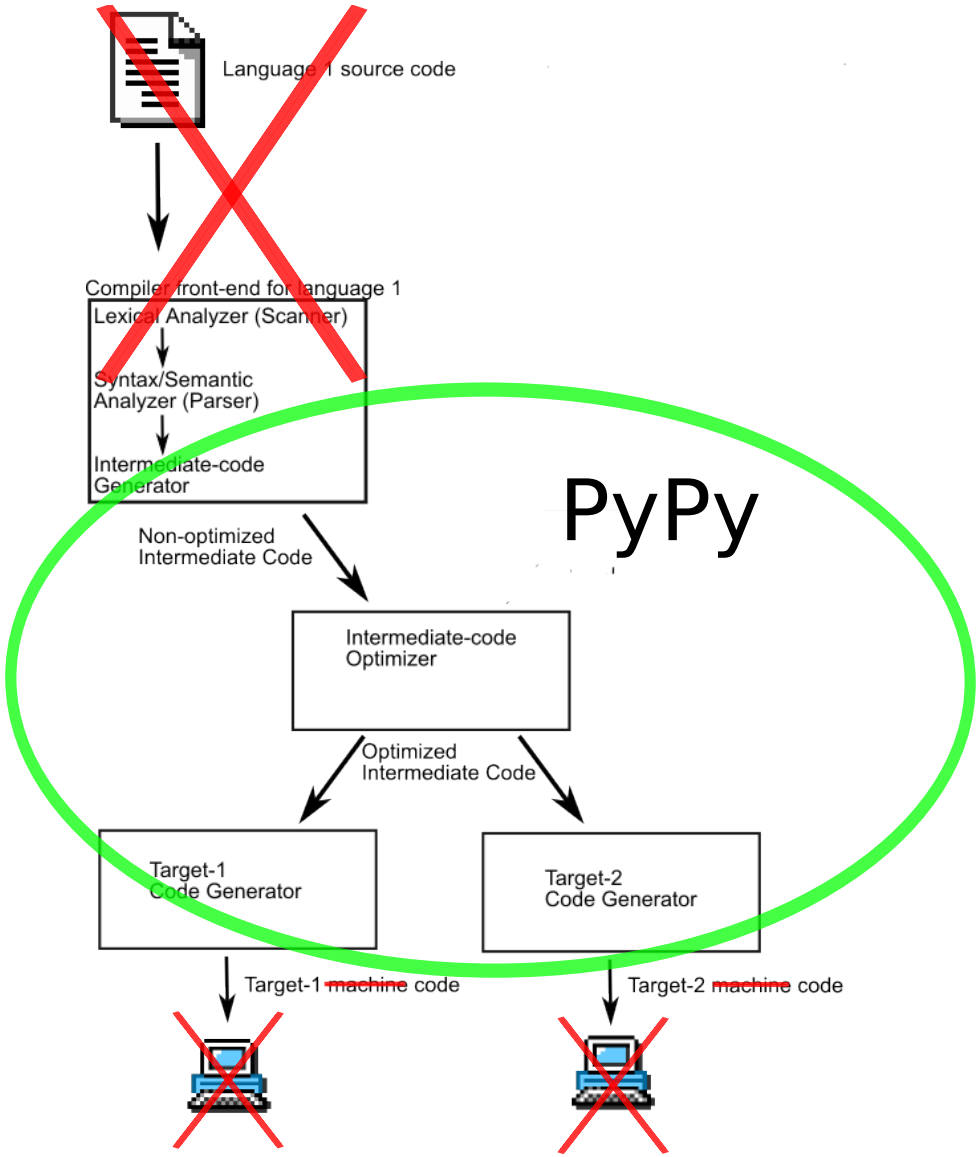
Part 3:

Technical Details of the Translation Toolchain

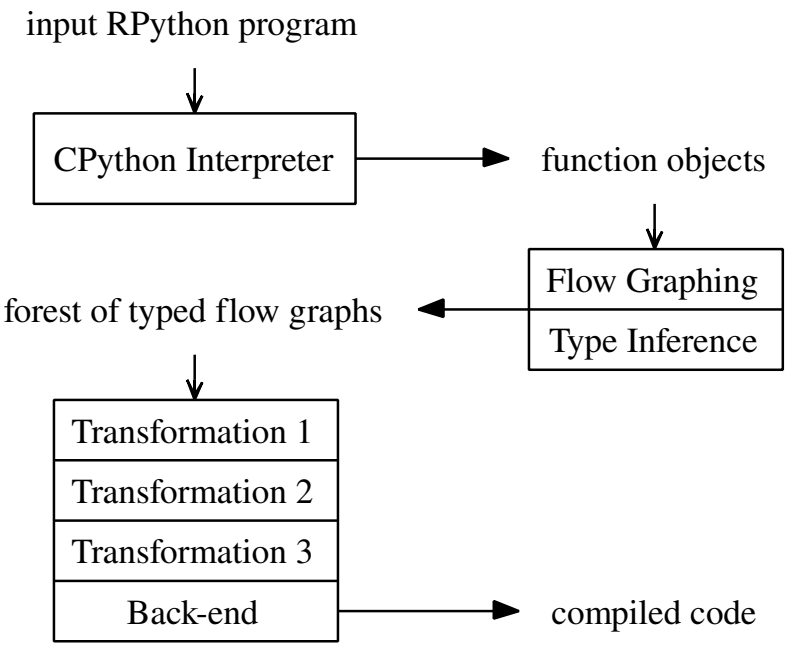
Typical Components of a Compiler



The PyPy Translator

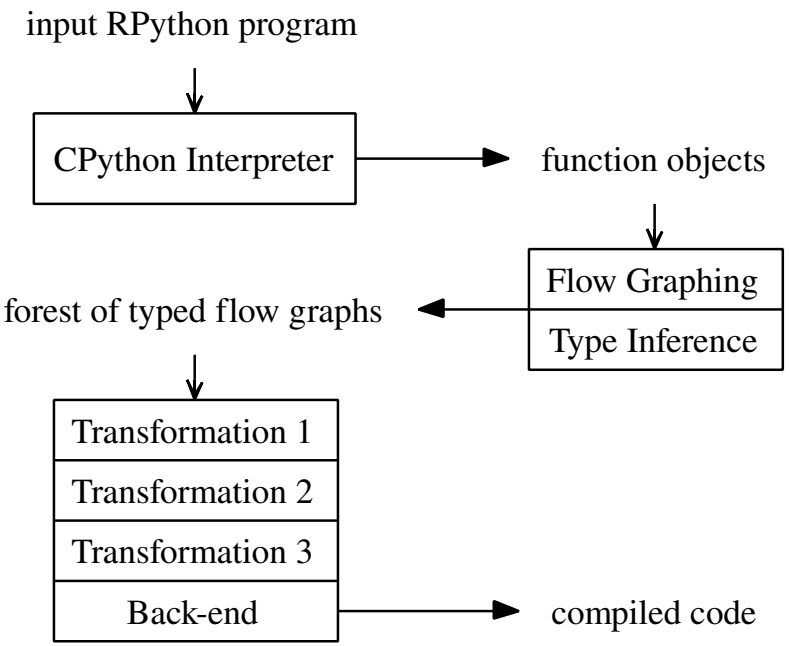


Translation Steps



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

Translation Steps

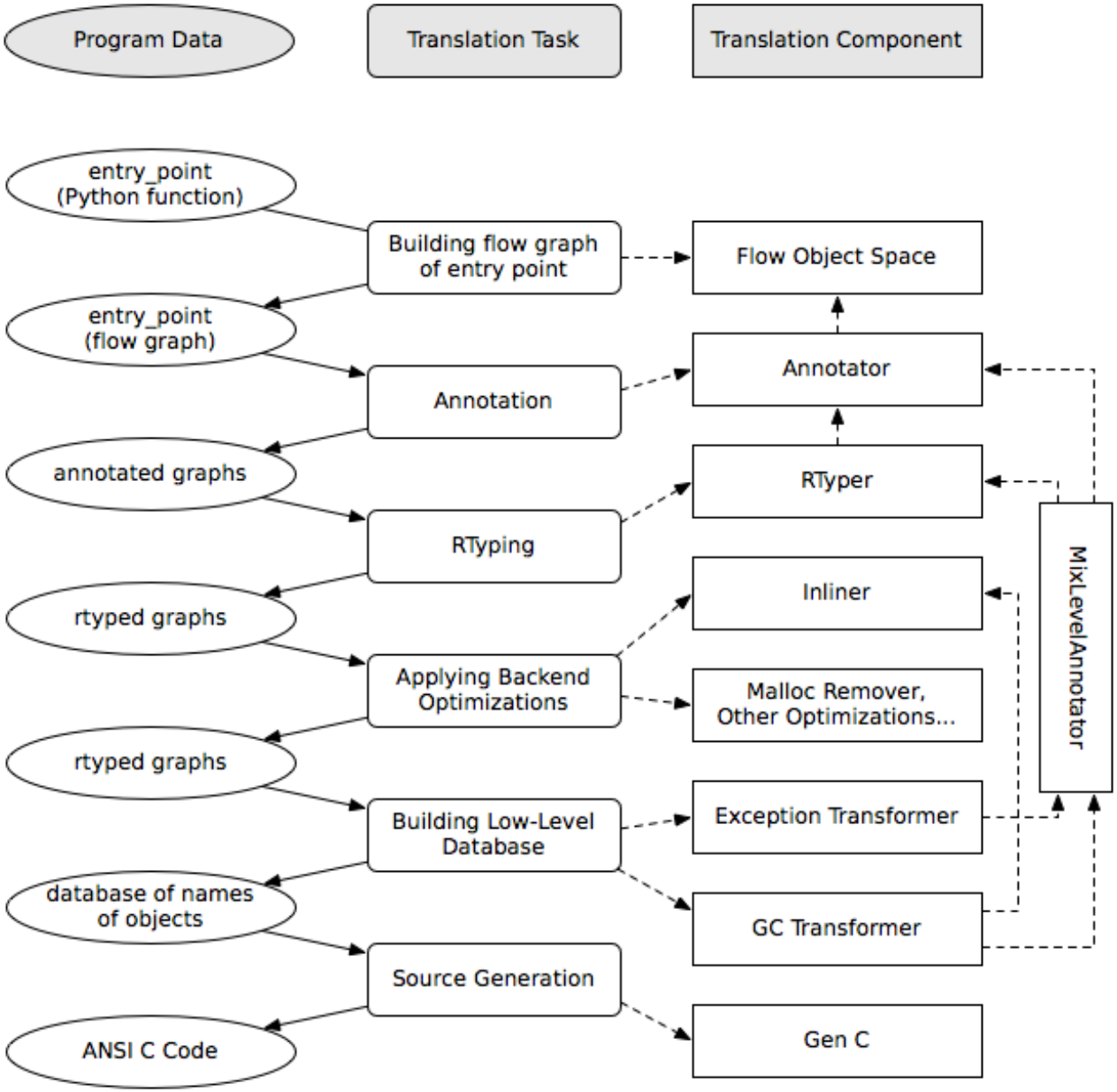


- Generation of a Control Flow Graph
- Type Inference
- Abstraction-Lowering Transformations
- Optimizations
- Code-Generation by the backend

Backends

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

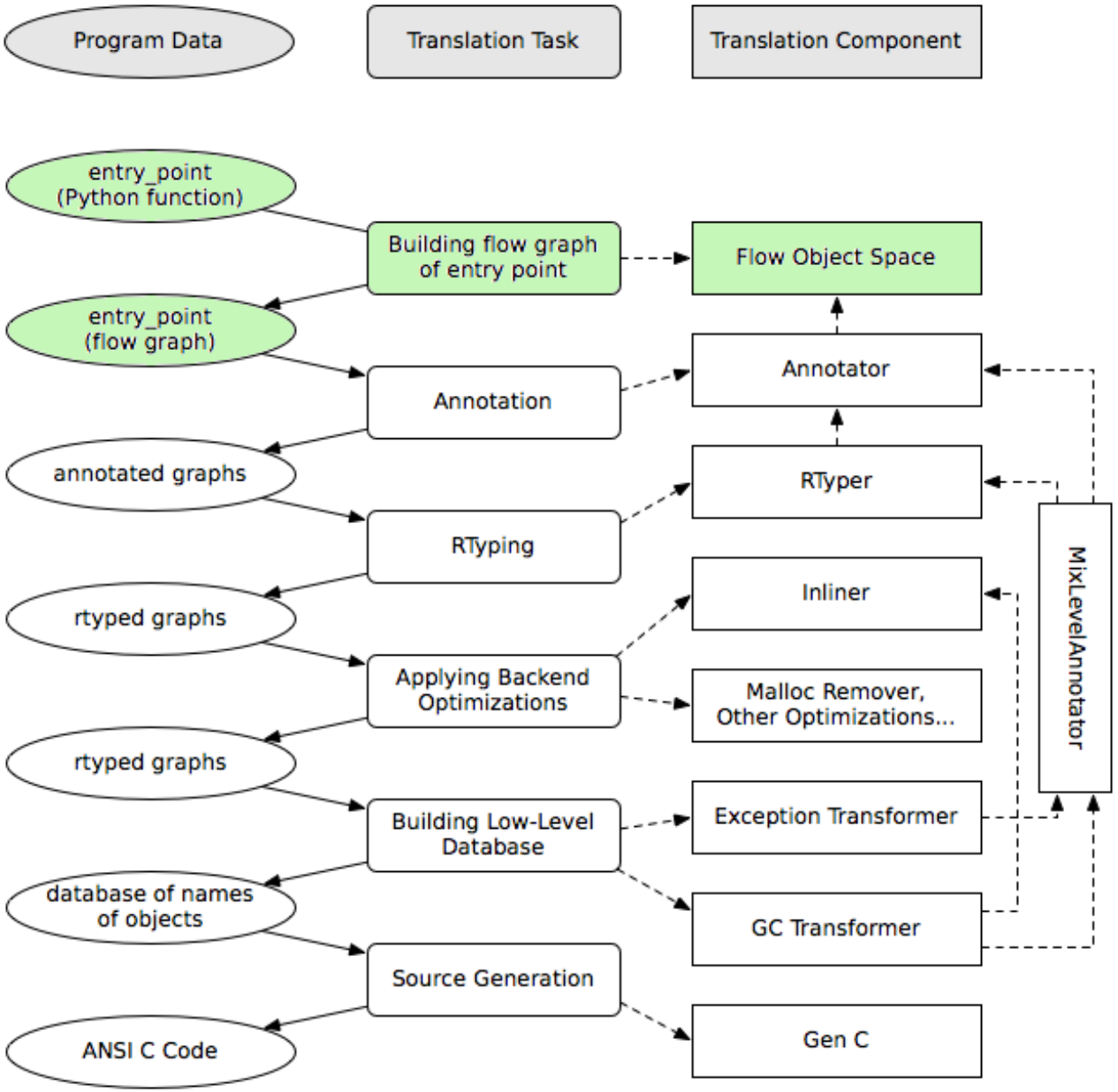
Detailed Overview



Running Example

```
1  def is_prime(n, primes):
2      for num in primes:
3          if n % num == 0:
4              return False
5      return True
6
7
8
9  def primes(n):
10     primes = [2]
11     for i in range(3, n + 1, 2):
12         if is_prime(i, primes):
13             primes.append(i)
14     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)

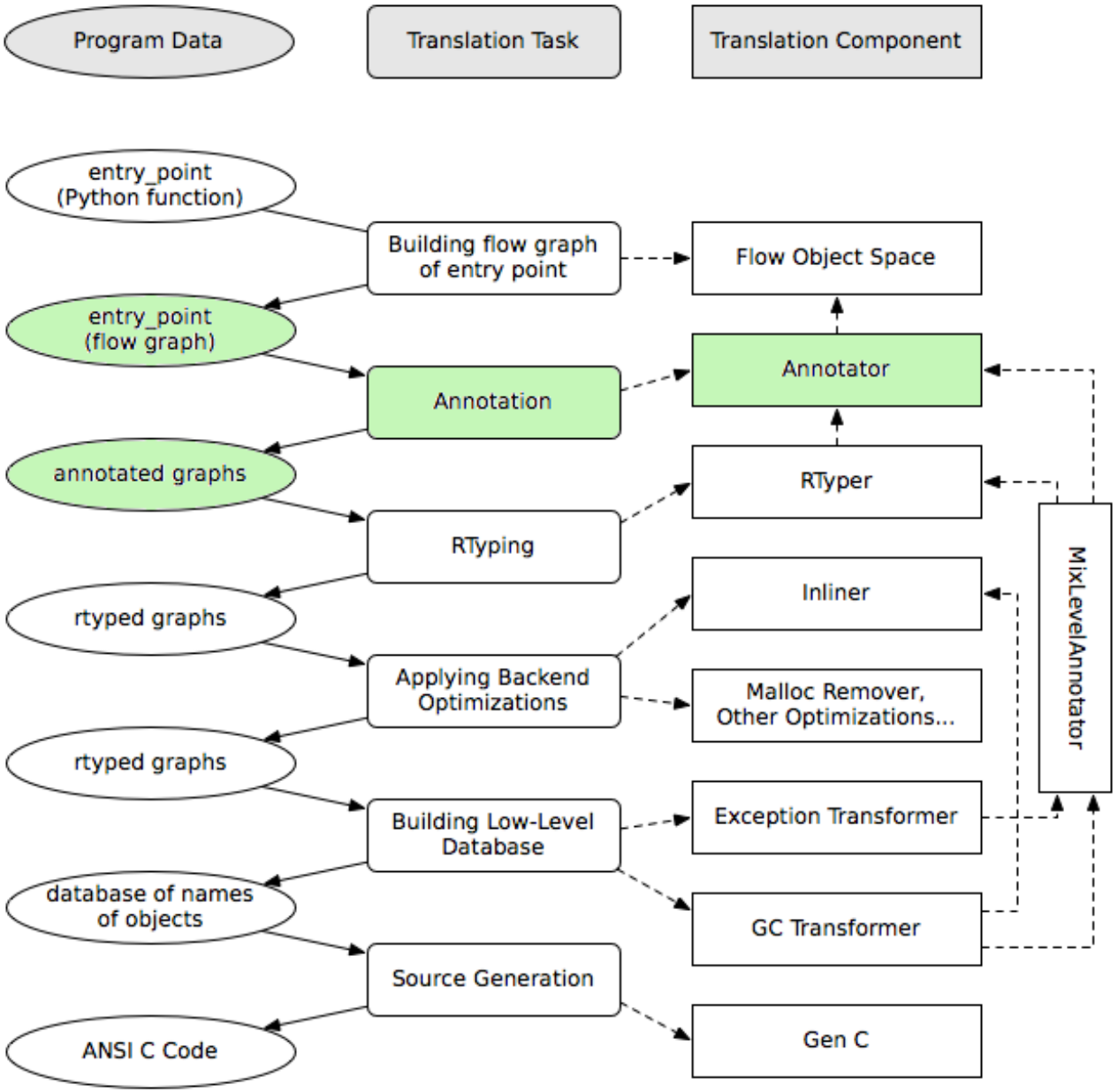
Generation of the Control Flow Graph

- 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

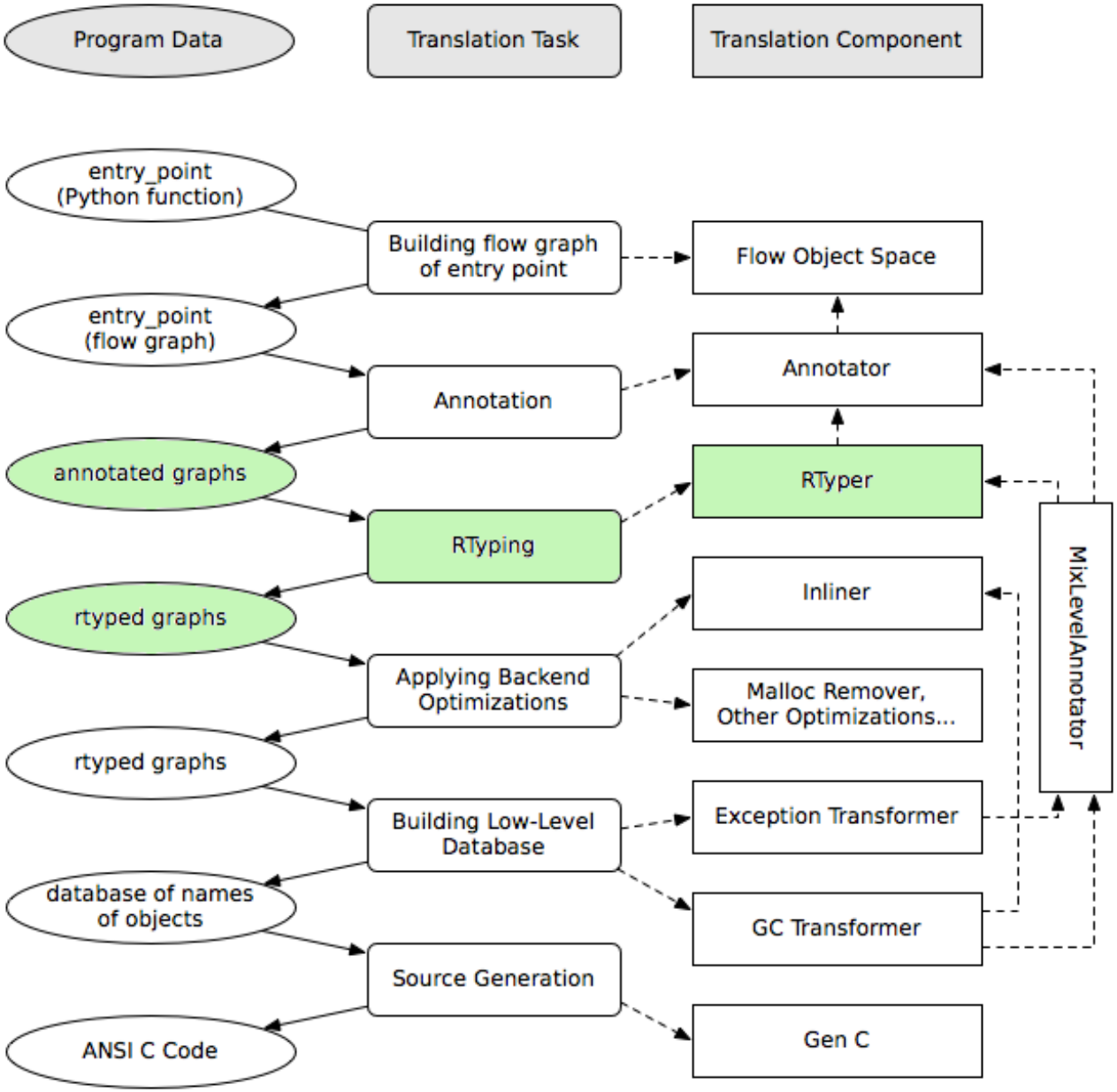
Typinferenz



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

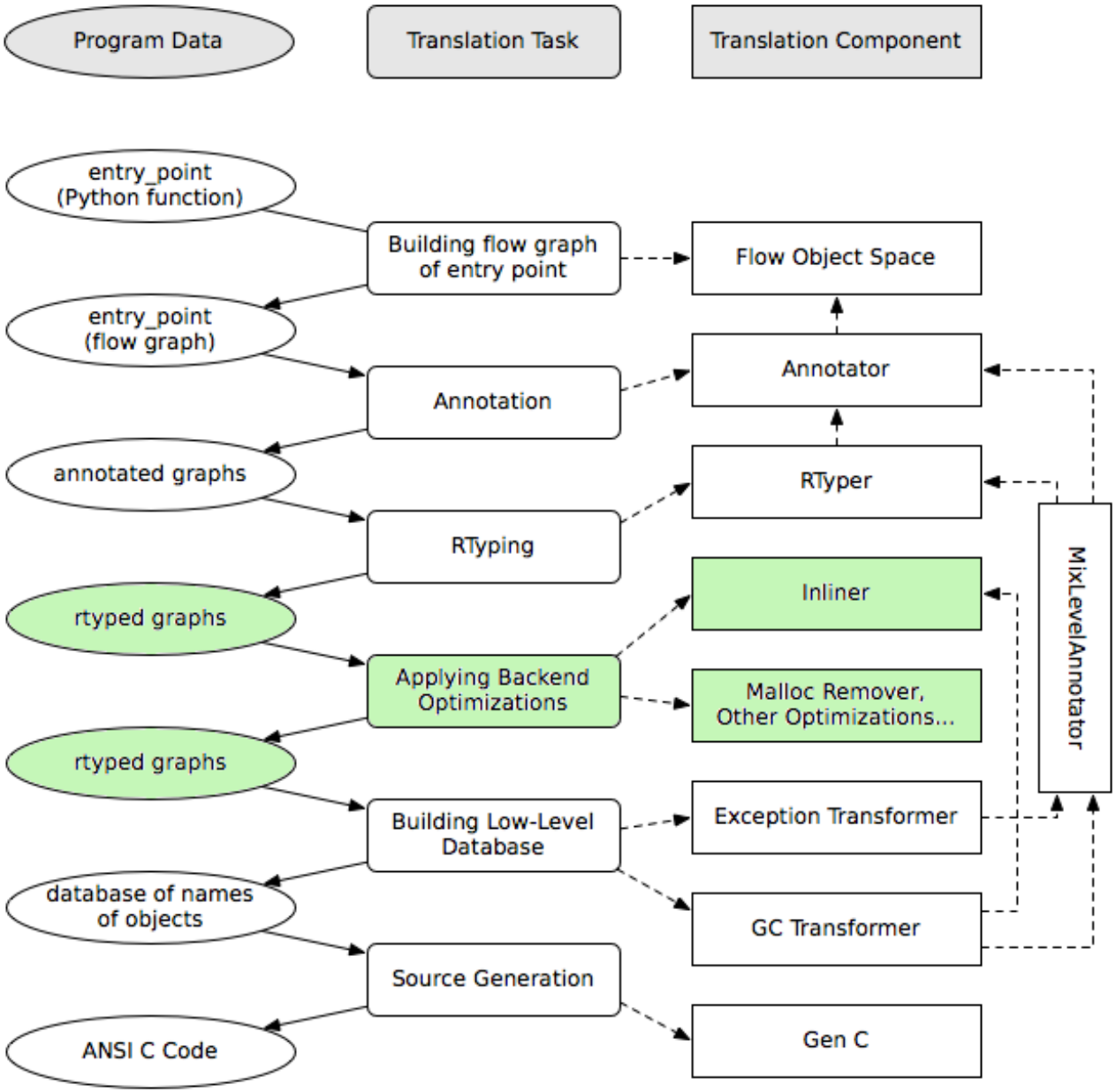
Lowering Of the Abstraction Level



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

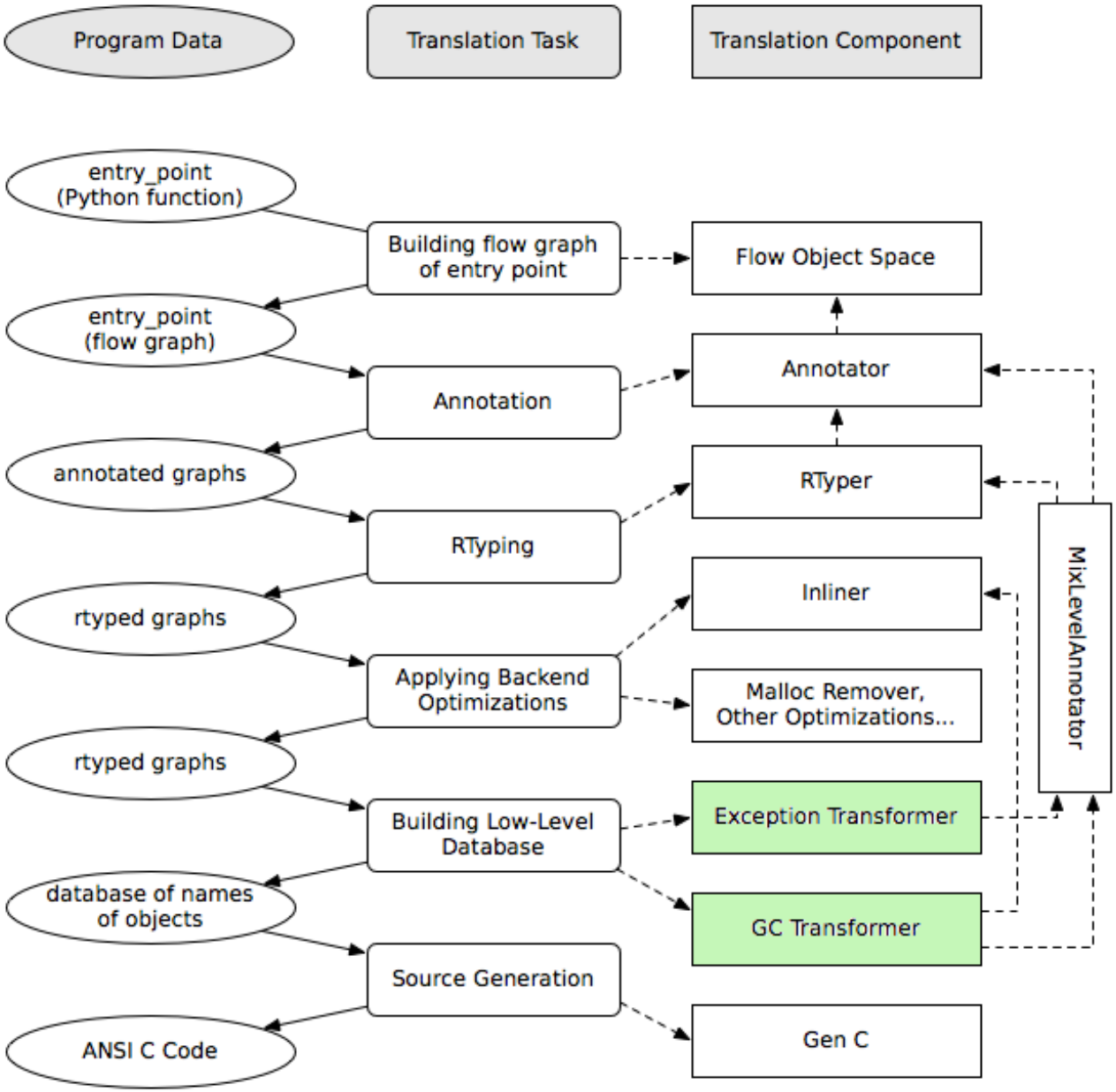
Inlining

- 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

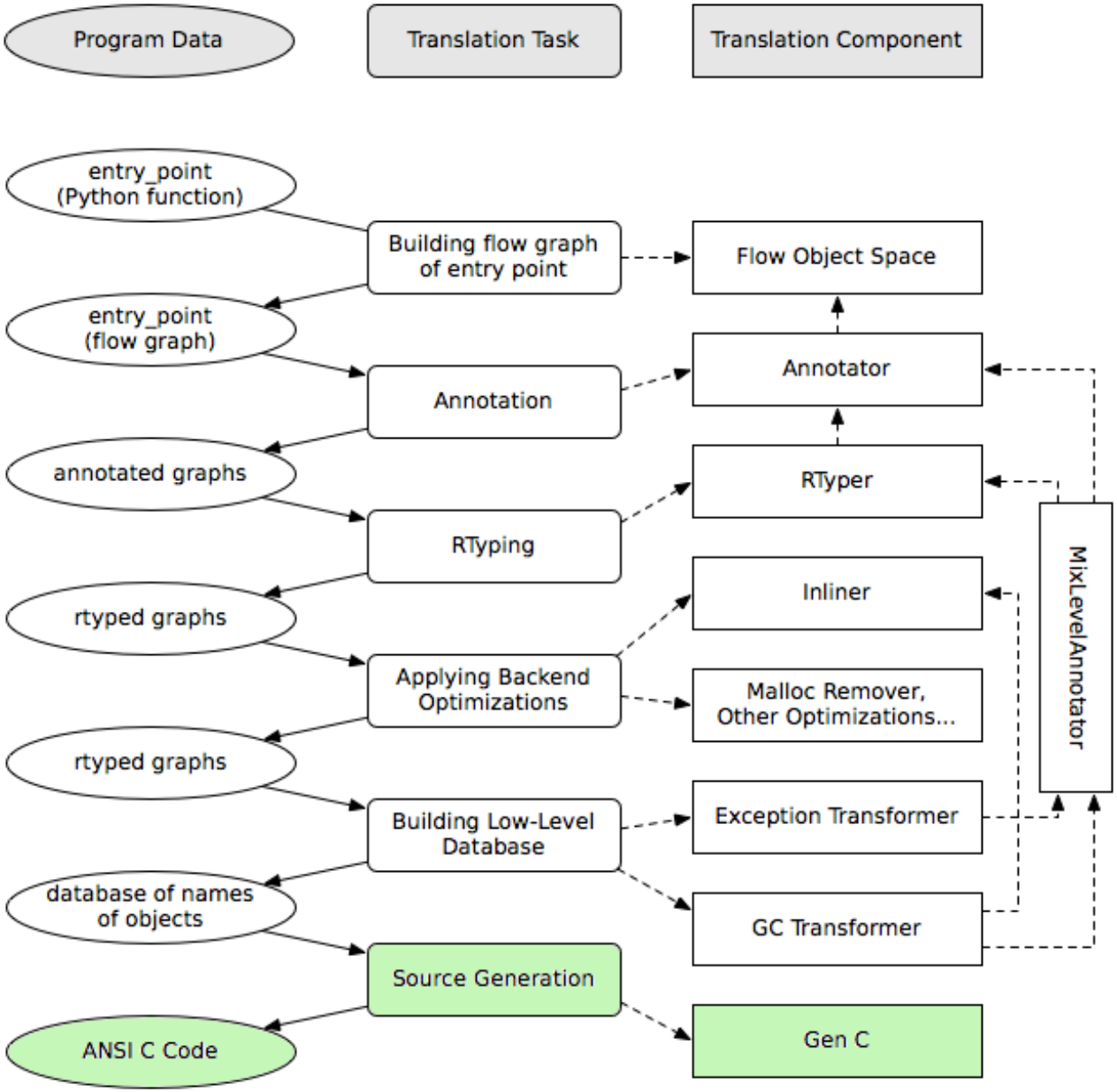
Exceptions and Memory Management



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

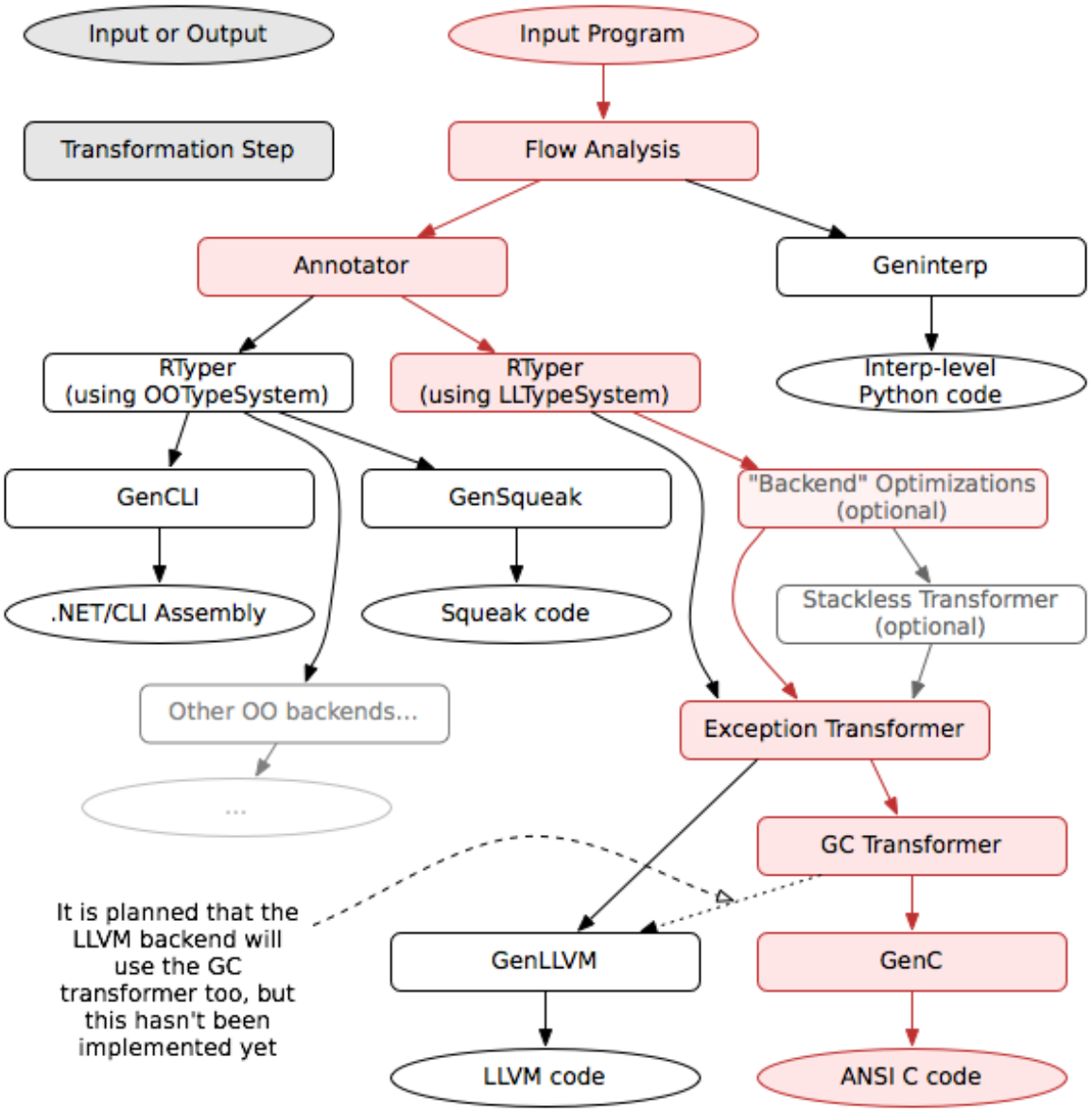
boring

Code Generation

boring

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

Presented Parts of PyPy



Part 2:

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
- what happens when a tracing JIT is applied “one level down”, i.e. to an interpreter
- how to solve the occurring problems

Context: The PyPy Project

- a general environment for implementing dynamic languages
- contains a compiler for a subset of Python (“RPython”)
- interpreters for dynamic languages written in that subset
- various interpreters written with PyPy: Python, Prolog, Smalltalk, Scheme, JavaScript

Context: The PyPy Project

- a general environment for implementing dynamic languages
- contains a compiler for a subset of Python (“RPython”)
- interpreters for dynamic languages written in that subset
- various interpreters written with PyPy: Python, Prolog, Smalltalk, Scheme, JavaScript
- and a GameBoy emulator

Context: The PyPy Project

- a general environment for implementing dynamic languages
- contains a compiler for a subset of Python (“RPython”)
- interpreters for dynamic languages written in that subset
- various interpreters written with PyPy: Python, Prolog, Smalltalk, Scheme, JavaScript
- and a GameBoy emulator
- goal: have a JIT that works for all these languages

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

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)
```

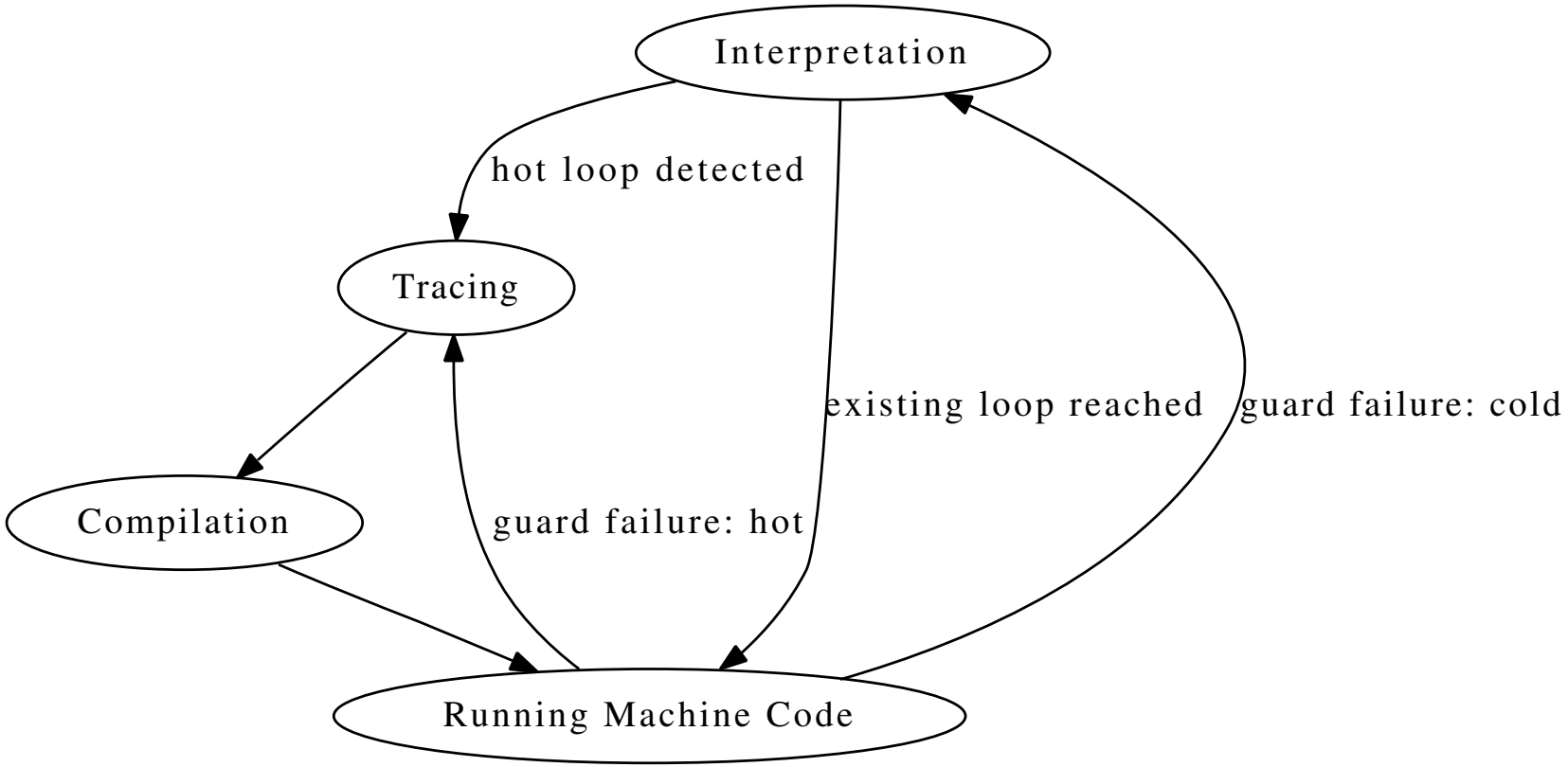
Dealing With Control Flow

- 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

Example

```
1  def f(x):
2      ... some code
3      while cond1:
4          if cond2:
5              if cond3:
6                  ...
7          else:
8              if cond4:
9                  ...
10     while True:
11         if cond5:
12             ...
13         if cond6:
14             ...
15         if cond7:
16             break
```

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

Applying a Tracing JIT to an Interpreter

- Question: What happens if the program is itself a bytecode interpreter?
- the (most important) hot loop of a bytecode interpreter is the bytecode dispatch loop
- Assumption violated: consecutive iterations of the dispatch loop will usually take very different code paths
- what can we do?

Applying a Tracing JIT to an Interpreter

- Question: What happens if the program is itself a bytecode interpreter?
- the (most important) hot loop of a bytecode interpreter is the bytecode dispatch loop
- Assumption violated: consecutive iterations of the dispatch loop will usually take very different code paths
- what can we do?

Terminology

- tracing interpreter: the interpreter that originally runs the program and produces traces
- language interpreter: the bytecode interpreter run on top
- user program: the program run by the language interpreter

```
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_R_A:
            n = ord(bytecode[pc])
            pc += 1
            a = regs[n]
        elif opcode == MOV_A_R:
            ...
        elif opcode == ADD_R_TO_A:
            ...
        elif opcode == DECR_A:
            a -= 1
        elif opcode == RETURN_A:
            return a


#
```

```
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_R_A:
            n = ord(bytecode[pc])
            pc += 1
            a = regs[n]
        elif opcode == MOV_A_R:
            ...
        elif opcode == ADD_R_TO_A:
            ...
        elif opcode == DECR_A:
            a -= 1
        elif opcode == RETURN_A:
            return a
```

```
# Example bytecode
# Square the 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:
JUMP_IF_A   4   #    goto 4
MOV_R_A     2   # return res
RETURN_A
```

#

Trace

Resulting trace when tracing bytecode `DECR_A`:

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

Idea for a Solution

- goal: try to trace the loops in the user program, and not just one iteration of the bytecode dispatch loop
- effectively unrolling the bytecode dispatch loop
- tracing interpreter needs information about the language interpreter
- provided by adding three hints to the language interpreter

Idea for a Solution

- goal: try to trace the loops in the user program, and not just one iteration of the bytecode dispatch loop
- effectively unrolling the bytecode dispatch loop
- tracing interpreter needs information about the language interpreter
- provided by adding three hints to the language interpreter

Hints Give Information About:

- which variables make up the program counter of the language interpreter (together those are called position key)
- where the bytecode dispatch loop is
- which bytecodes can correspond to backward jumps

Interpreter with Hints

```
1  tlrjitdriver = JitDriver(['pc', 'bytecode'])
2
3  def interpret(bytecode, a):
4      regs = [0] * 256
5      pc = 0
6      while True:
7          tlrjitdriver.start_dispatch_loop()
8          opcode = ord(bytecode[pc])
9          pc += 1
10         if opcode == JUMP_IF_A:
11             target = ord(bytecode[pc])
12             pc += 1
13             if a:
14                 pc = target
15                 if target < pc:
16                     tlrjitdriver.backward_jump()
17         elif opcode == MOV_A_R:
18             ... # rest unmodified
```

Modifying Tracing

- goal: try to trace the loops in the user program, and not just one iteration of the bytecode dispatch loop
- change condition when tracing interpreter stops to trace
- tracing interpreter stops tracing only when:
  - it sees a backward jump in the language interpreter
  - the position key of the language interpreter matches an earlier value
- in this way, full user loops are traced

Result When Tracing SQUARE

```
loop_header(a0, regs0, bytecode0, pc0)
# MOV_R_A 0
opcode0 = strgetitem(bytecode0, pc0)
pc1 = int_add(pc0, Const(1))
guard_value(opcode0, Const(2))
n1 = strgetitem(bytecode0, pc1)
pc2 = int_add(pc1, Const(1))
a1 = list_getitem(regs0, n1)
# DECR_A
# MOV_A_R 0
# MOV_R_A 2
# ADD_R_TO_A 1
# MOV_A_R 2
# MOV_R_A 0
...
# JUMP_IF_A 4
opcode6 = strgetitem(bytecode0, pc13)
pc14 = int_add(pc13, Const(1))
guard_value(opcode6, Const(3))
target0 = strgetitem(bytecode0, pc14)
pc15 = int_add(pc14, Const(1))
i1 = int_is_true(a5)
guard_true(i1)
jump(a5, regs0, bytecode0, target0)
```

What Have We Won?

- trace corresponds to one loop of the user program
- trace has the chance to loop for a while until a guard fails

What Have We Won?

- trace corresponds to one loop of the user program
- trace has the chance to loop for a while until a guard fails
- however, most operations are concerned with manipulating bytecode and program counter
- bytecode and program counter are part of the position key
- thus they are constant at the beginning of the loop
- therefore they can and should be constant-folded

Result When Tracing SQUARE With Constant-Folding

```
loop_header(a0, regs0)
# MOV_R_A 0
a1 = list_getitem(regs0, Const(0))
# DECR_A
a2 = int_sub(a1, Const(1))
# MOV_A_R 0
list_setitem(regs0, Const(0), a2)
# MOV_R_A 2
list_getitem(regs0, Const(2))
# ADD_R_TO_A  1
i0 = list_getitem(regs0, Const(1))
a4 = int_add(a3, i0)
# MOV_A_R 2
list_setitem(regs0, Const(2), a4)
# MOV_R_A 0
a5 = list_getitem(regs0, Const(0))
# JUMP_IF_A 4
i1 = int_is_true(a5)
guard_true(i1)
jump(a5, regs0)
```

Results

- almost only computations related to the user program remain
- list of registers is only vestige of language interpreter

Results

- almost only computations related to the user program remain
- list of registers is only vestige of language interpreter

Timing Results Computing Square of 10'000'000

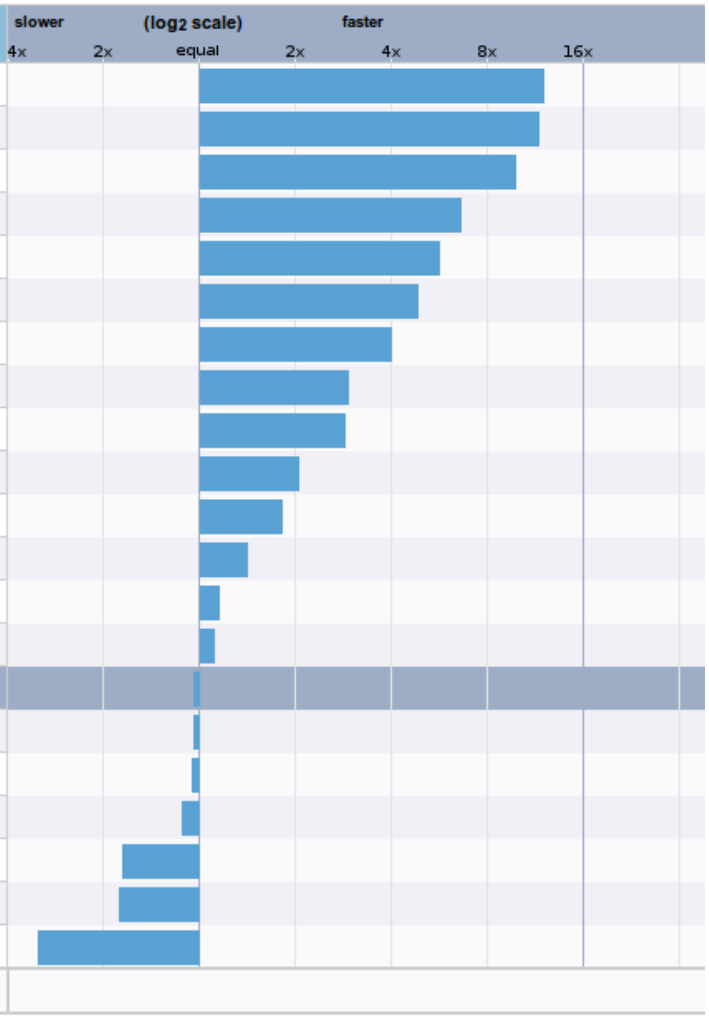
		Time (ms)	speedup
1	No JIT	442.7 ± 3.4	1.00
2	JIT, Normal Trace Compilation	1518.7 ± 7.2	0.29
3	JIT, Unrolling of Interp. Loop	737.6 ± 7.9	0.60
4	JIT, Full Optimizations	156.2 ± 3.8	2.83

Scaling to Large Interpreters?

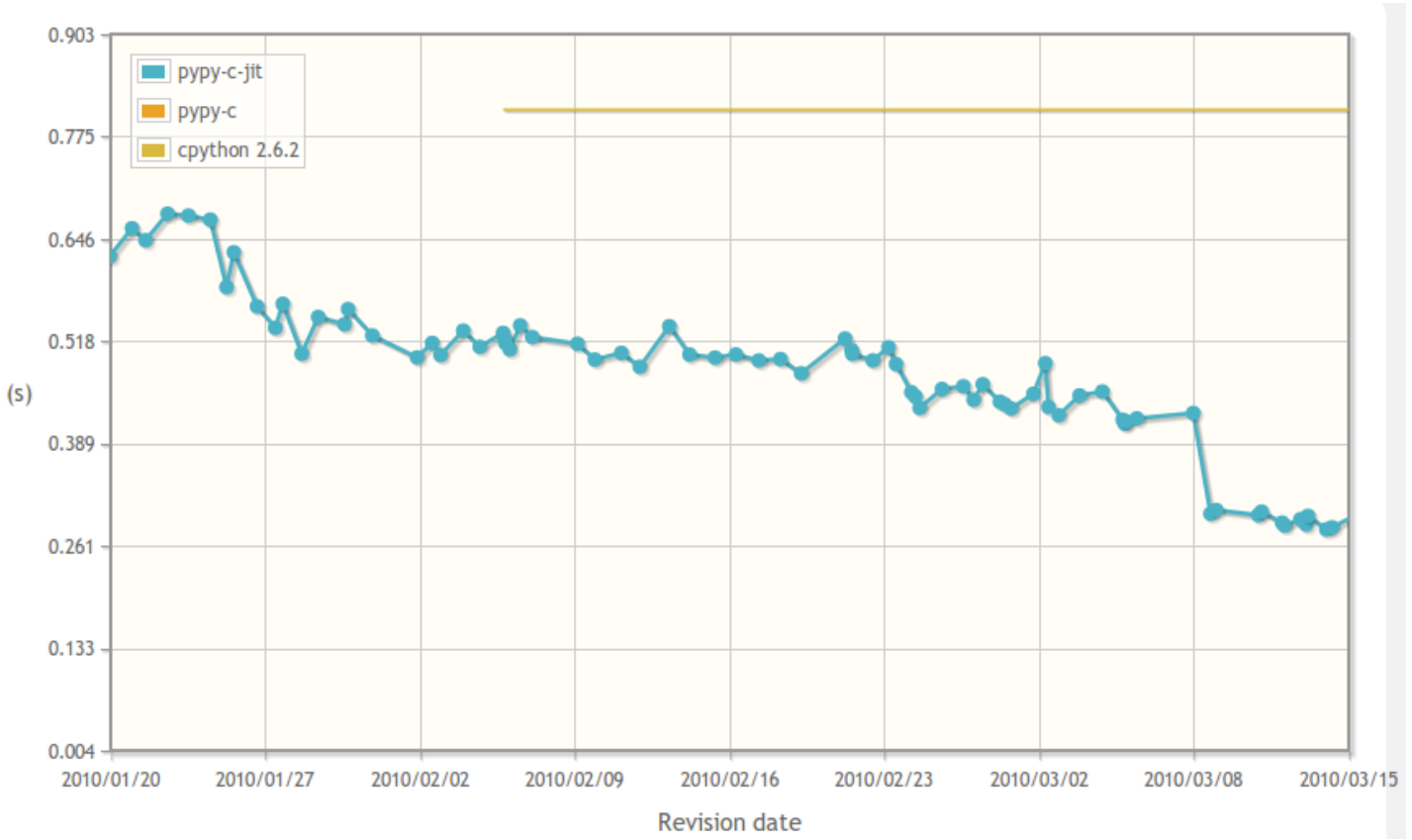
- we can apply this approach to PyPy's Python interpreter (70 KLOC)
- speed-ups very promising (see next slide)
- no Python-specific bugs!

Benchmarks

Benchmark	Time (s)	std dev	Current change	Trend	Times
richards	0.0246	0.0018	0.32%	-0.23%	12.04
spectral-norm	0.0417	0.0060	-4.43%	-3.90%	11.57
chaos	0.0426	0.0327	4.59%	1.31%	9.77
nbody_modified	0.0810	0.0086	2.48%	0.44%	6.63
float	0.0895	0.0285	-1.83%	-2.37%	5.64
fannkuch	0.3754	0.0122	1.64%	0.60%	4.86
twisted_iteration	0.0329	0.0003	0.10%	-0.64%	4.02
telco	0.3382	0.0155	-0.41%	-1.13%	2.95
django	0.2834	0.0071	-0.48%	-1.05%	2.85
spitfire_cstringio	4.1262	0.1360	-7.40%	-1.18%	2.06
twisted_pb	0.0313	0.0003	-0.67%	-2.43%	1.83
twisted_names	0.0063	0.0000	-0.68%	-1.19%	1.41
html5lib	10.1353	1.5026	2.37%	3.04%	1.15
twisted_web	0.1008	0.0079	-0.39%	-1.04%	1.12
rietveld	0.4946	0.2217	-0.87%	0.21%	0.96
spitfire	7.1760	0.1540	-0.82%	2.35%	0.96
ai	0.4504	0.0078	-3.11%	1.08%	0.94
meteor-contest	0.3509	0.0135	-2.62%	-2.47%	0.88
twisted_tcp	1.6573	0.0176	-2.88%	-1.52%	0.57
slowspitfire	1.2217	0.1532	-7.55%	-0.77%	0.56
spambayes	0.8845	0.0646	-3.65%	-0.24%	0.31
Average			-1.25%	-0.53%	



Django Benchmark



Conclusions

- some small changes to a tracing JIT makes it possible to effectively apply it to bytecode interpreters
- result is similar to a tracing JIT for that language
- bears resemblance to partial evaluation, arrived at by different means
- maybe enough to write exactly one tracing JIT?

Conclusions

- some small changes to a tracing JIT makes it possible to effectively apply it to bytecode interpreters
- result is similar to a tracing JIT for that language
- bears resemblance to partial evaluation, arrived at by different means
- maybe enough to write exactly one tracing JIT?

Outlook

- scale to really huge programs
- make the tracing JIT more user-friendly
- are the speedups good enough?

Thank you! Questions?

- some small changes to a tracing JIT makes it possible to effectively apply it to bytecode interpreters
- result is similar to a tracing JIT for that language
- bears resemblance to partial evaluation, arrived at by different means
- maybe enough to write exactly one tracing JIT?

Outlook

- scale to really huge programs
- make the tracing JIT more user-friendly
- are the speedups good enough?