Back to the Future in one Week — Implementing a Smalltalk VM in PyPy

Carl Friedrich Bolz[1] Adrian Kuhn[2] Adrian Lienhard[2]
Nicholas D. Matsakis[3] Oscar Nierstrasz[2]
Lukas Renggli[2] Armin Rigo Toon Verwaest[2]

[2]Software Composition Group
University of Bern, Switzerland

[1]Softwaretechnik und Programmiersprachen
Heinrich-Heine-Universität Düsseldorf

[3]ETH Zürich, Switzerland

Workshop on Self-sustaining Systems, May 16 2008

Scope

This talk is about:

- writing a Squeak implementation (called "SPy") in Python
- with eight people
- in five days
- using PyPy

What is PyPy?

- started as a Python implementation in Python
- Open Source project, MIT license
- developed into a general environment for implementing dynamic languages
- supports the language developer with a lot of infrastructure
- most important goal: abstracting over low-level details
- don't fix decisions about low-level details early

PyPy's Approach to VM Construction

- implement an interpreter for the dynamic language in RPython
- translate this interpreter to a low-level language with PyPy toolchain
- a variety of target environment: C, LLVM, JVM, .NET

PyPy's Approach to VM Construction

- implement an interpreter for the dynamic language in RPython
- translate this interpreter to a low-level language with PyPy toolchain
- a variety of target environment: C, LLVM, JVM, .NET

What is RPython?

- a more static subset of Python
- static enough to allow type inference
- still rather expressive: exceptions, single inheritance, dynamic dispatch, interesting builtin types, garbage collection

Translation Aspects

- many low-level details of the final VM are orthogonal to language semantics
- examples: GC strategy, threading model, many object details
- those shouldn't be visible in the interpreter source
- they are inserted during translation

Translation Aspects

- many low-level details of the final VM are orthogonal to language semantics
- examples: GC strategy, threading model, many object details
- those shouldn't be visible in the interpreter source
- they are inserted during translation
- non-trivial translation aspect: auto-generating a dynamic compiler

Translation Aspects

- many low-level details of the final VM are orthogonal to language semantics
- examples: GC strategy, threading model, many object details
- those shouldn't be visible in the interpreter source
- they are inserted during translation
- non-trivial translation aspect: auto-generating a dynamic compiler

Compile-Time Metaprogramming

- PyPy's translation toolchain starts analysis after importing
- arbitrary (non-RPython) code can be executed during import
- this allows metaprogramming of parts of the interpreter

The SPy VM

- really simple, straight-forward Squeak interpreter in RPython
- goal is to fully support loading and running Squeak images
- source code essentially free of low-level details, no GC
- written during a five-day sprint in October in Bern

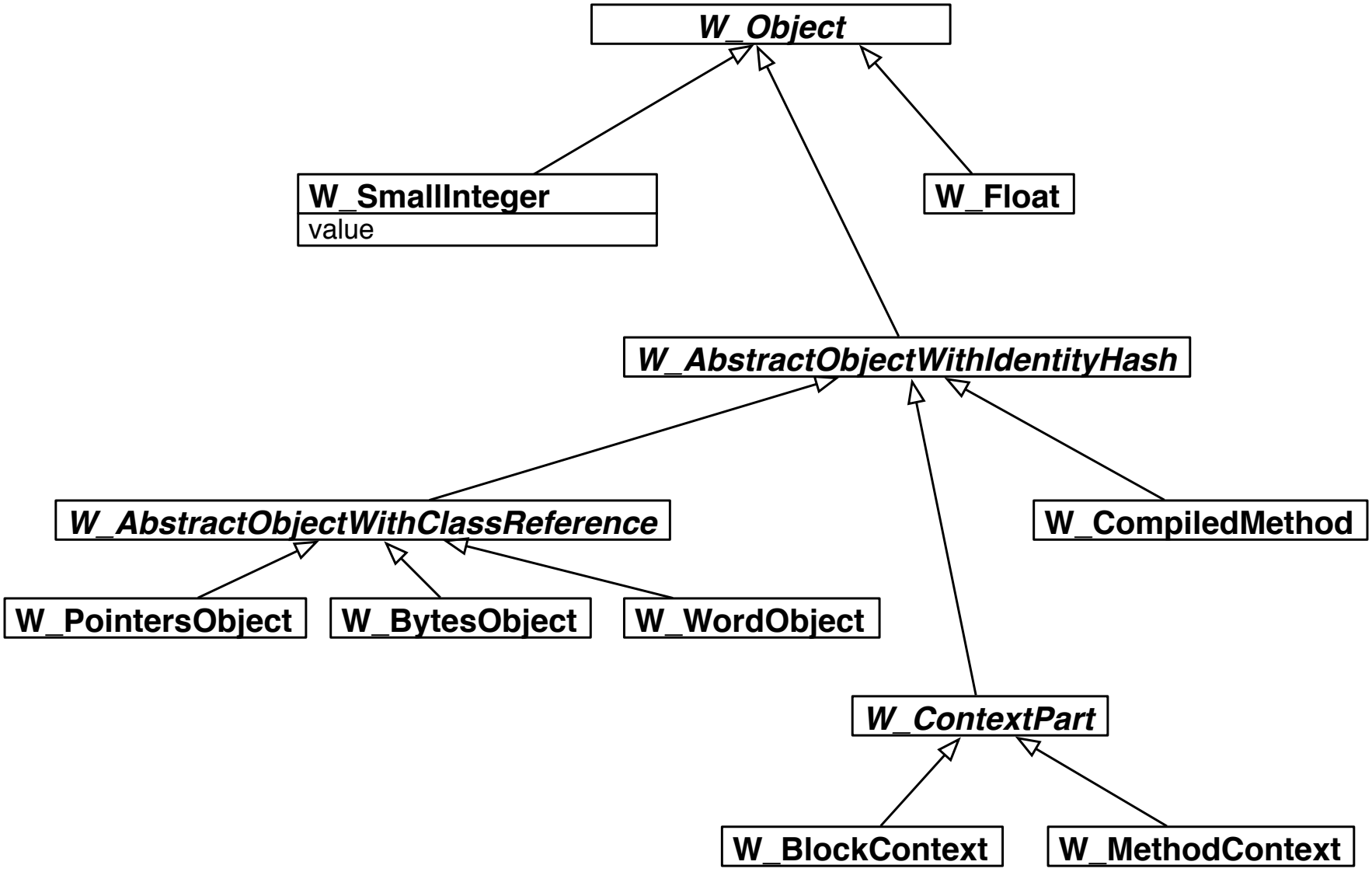
The SPy VM

- really simple, straight-forward Squeak interpreter in RPython
- goal is to fully support loading and running Squeak images
- source code essentially free of low-level details, no GC
- written during a five-day sprint in October in Bern

Status

- bytecodes fully implemented
- many primitives implemented: arithmetic primitives, object primitives
- can load Squeak images by decoding the bit patterns
- runs simple benchmarks

SPy Object Model



Shadows

Problem:

- classes are just objects: can't really say which objects are used as classes
- any object with the right instance fields can be used as a class
- cryptic (bit)fields in the class which the VM needs to decode all the time

Shadows

Problem:

- classes are just objects: can't really say which objects are used as classes
- any object with the right instance fields can be used as a class
- cryptic (bit)fields in the class which the VM needs to decode all the time

Approach

- potentially attach a *shadow* to every object
- the shadow caches VM-internal information
- when the object is changed, the shadow is invalidated
- so far only used for classes, probably more later
- conceptual cleanliness and nicer VM implementation
- still allows to expose low-level view to user code

Tagged Pointers

- Squeak implements small integers as tagged pointers
- doing that is orthogonal to language semantics
- in PyPy implemented as a translation aspect
- one class with exactly one int field can optionally be implemented with tagged pointers
- when enabled, all method calls check for tag first
- usually changing such an implementation choice would be a major effort

Tagged Pointers

- Squeak implements small integers as tagged pointers
- doing that is orthogonal to language semantics
- in PyPy implemented as a translation aspect
- one class with exactly one int field can optionally be implemented with tagged pointers
- when enabled, all method calls check for tag first
- usually changing such an implementation choice would be a major effort

Results of Tagging

- small slowdown due to need to check for flag
- memory advantage not measured
- really easy to do

Primitives

- try to make writing primitives easy
- failure signaled by an exception
- automatic popping from the stack and unwrapping of arguments
- automatic pushing of the result
- using a custom function decorator `expose_primitive`
- compile-time metaprogramming

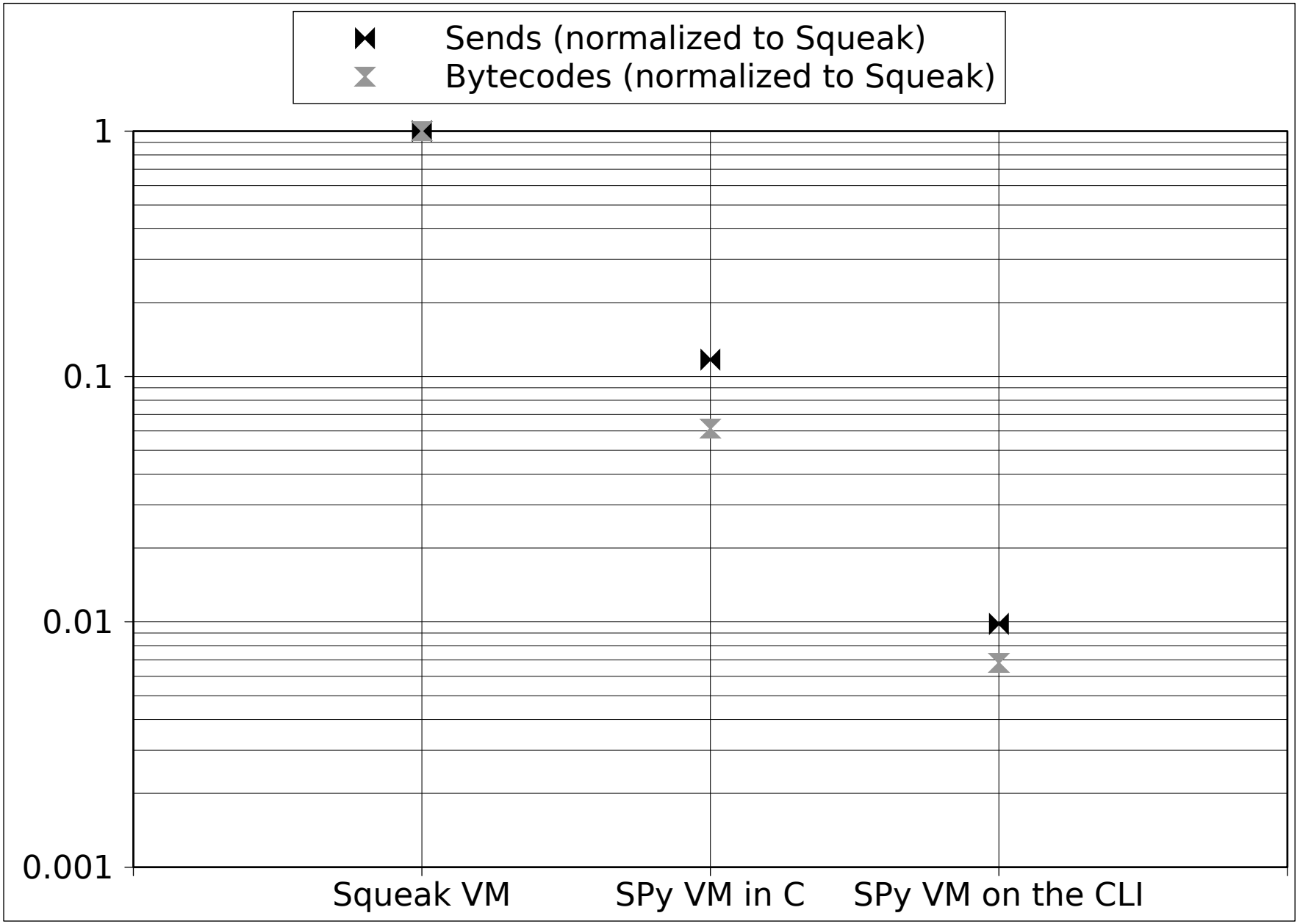
```
bool_ops = [
    (LESSTHAN, lambda x, y: x < y),
    (GREATERTHAN, lambda x, y: x > y),
    (LESSOREQUAL, lambda x, y: x <= y),
    (GREATEROREQUAL,lambda x, y: x >= y),
    (EQUAL, lambda x, y: x == y),
    (NOTEQUAL, lambda x, y: x != y)
    ]


def make_primitive(prim_num, op):

    @expose_primitive(prim_num, unwrap_spec=[int, int])
    def primitive(interp, v1, v2):
        res = op(v1, v2)
        w_res = utility.wrap_bool(res)
        return w_res

for (prim_num, op) in bool_ops:
    make_primitive(prim_num, op)
```

Performance (tiny Benchmark)



Results

Good Points of the Approach:

- simple, understandable, high-level Squeak implementation
- mostly free of low-level details: no GC, no manual pointer tagging
- written in a short amount of time
- runs on various platforms

Results

Good Points of the Approach:

- simple, understandable, high-level Squeak implementation
- mostly free of low-level details: no GC, no manual pointer tagging
- written in a short amount of time
- runs on various platforms

Bad Points of the Approach:

- not really fast (yet)
- RPython isn't Python
- toolchain needs quite some effort (but only once)

Outlook

- do graphical builtins, to actually start the full environment
- Squeak-specific optimizations:
- method-cache (should be easy with shadows)
- JIT (see next slide)
- lessons learned for a "SqueaSquea"?

Join the Sprint!

Saturday - Thursday, C-Base Berlin

JIT Outlook

- PyPy currently explores automatic JIT generation
- 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
- shares ideas with tracing JITs

JIT Outlook

- PyPy currently explores automatic JIT generation
- 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
- shares ideas with tracing JITs

Current status

- basic ideas seem to work, good speedups in prototypes
- a lot of details still left

Questions?

Outlook:

- do graphical builtins, to actually start the full environment
- Squeak-specific optimizations:
- method-cache (should be easy with shadows)
- JIT (automatic generation of dynamic compilers)
- lessons learned for a "SqueaSquea"?

Join the Sprint!

Saturday - Thursday, C-Base Berlin