End-to-end language composition

Edd Barrett  Carl Friedrich Bolz  Lukas Diekmann  Laurence Tratt  Naveneetha Krishnan Vasudevan

Software Development Team
2014-04-09
Our problem

We want better programming languages. But better always seems to end up bigger.
We want **better** programming languages
We want better programming languages.

But better always seems to end up bigger.
Language composition

≜
mixing languages together
Underlying language composition challenges
Underlying language composition challenges

Python

Prolog

Unipycation
Underlying language composition challenges

- Python
  - Syntax
  - Runtime
- Prolog
  - Syntax
  - Runtime
- Unipycation
  - Syntax
  - Runtime
Underlying language composition challenges

Language boxes

- Python
  - Syntax
  - Runtime

- Prolog
  - Syntax
  - Runtime

Unipyrcation
  - Syntax
  - Runtime
Underlying language composition challenges

- Syntax: Python, Prolog
- Runtime: Python, Prolog
- Language boxes
- Unipyication
- Composed meta-tracing VMs

Python syntax
Python runtime
Prolog syntax
Prolog runtime
PL X
<grammar>
expr ::= ...
term ::= ...
    | ...
    | ...
func ::= ...
PL X
<program>
for (j : js) {
    doStuff();
}
Parser
Parse Tree
Parsing composition

PL X
<grammar>
expr ::= ...
term ::= ...
    | ...
    | ...
func ::= ...
PL X
<program>
for (j : js) {
    doStuff();
}
.
.
.
LR
Parse Tree

Software Development Team

http://soft-dev.org/
Parsing composition

PL X
<grammar>
expr ::= ...
  term ::= ... |
  term ::= ...
func ::= ...

PL X
<program>
for (j : js) {
    doStuff();
}
.
.
.
LR
Parse Tree
Undefined
Parsing composition

```
<grammar>
expr ::= ...
term ::= ...
| ...
| ...
func ::= ...

<program>
for (j : js) {
    doStuff();
}
```

Generalised Parse Tree

```
PL X
expr ::= ...
term ::= ...
func ::= ...
```

```
for (j : js) {
    doStuff();
}
```
Parsing composition

```plaintext
<grammar>
expr ::= ...
  | ...
  | ...
func ::= ...
<program>
for (j : js) {
    doStuff();
}
```

Generalised Parse Tree

 ambiguity

Parse Tree

Ambiguous
Parsing composition

PL X
<grammar>
expr ::= ... 
term ::= ...
  | ...
  | ...
func ::= ...
PL X
<program>
for (j : js) {
    doStuff();
}
.
.
.
PEG
Parse Tree
Shadows

PL X
<program>
for (j : js) {
    doStuff();
}
.
.
.
PEG
Parse Tree
Shadows
The only choice?
The only choice?

SDE
The challenge

Challenge:
SDE’s power + a text editor feel?
PL X
<grammar>
expr::= ...
term::= ...
    | ...
    | ...
func ::= ...

for (j : js) {
    doStuff();
}
.
.
.
Parser
Parse Tree

PL X
<program>
for (j : js) {
    doStuff();
}
.
.
.
Parser
Parse Tree
PL X
<grammar>
expr ::= ...
term ::= ...
  | ...
  | ...
func ::= ...

Incremental parser
Parse Tree
PL X
<program>
for (j : js) {
    doStuff();
}
.
.
.

9 / 28 http://soft-dev.org/
Eco

PL X
<grammar>
expr ::= ...  
term ::= ...  
| ...  
| ...  
func ::= ...

Incremental parser
Parse Tree

PL X
<program>
for (j : js) {
    doStuff();
}
.
.
.

Abstract Syntax Tree
Runtime composition

PL X
Interpreter

PL Y
Interpreter

C/C++
Runtime composition

PL X
Interpreter

PL Y
Interpreter

C/C++

Too slow
Runtime composition

PL X
JIT Compiler
Interpreter

C/C++

PL Y
JIT Compiler
Interpreter

http://soft-dev.org/
Runtime composition

PL X
JIT Compiler
Interpreter

PL Y
JIT Compiler
Interpreter

C/C++

Too much engineering
Runtime composition

PL X
Interpreter

PL Y
Interpreter

JVM/CLR

JIT Compiler
Runtime composition

Semantic mismatch

PL X
Interpreter

PL Y
Interpreter

JVM/CLR
JIT Compiler

PL X

PL Y

Semantic
mismatch

10 / 28 http://soft-dev.org/
Runtime composition

PL X  PL Y

Interpreters

Glue

Meta-tracing

PL Z

Interpreter

Tracing JIT
Meta-tracing translation with RPython

Interpreter
Meta-tracing translation with RPython

Interpreter

RPython translator
Meta-tracing translation with RPython
Meta-tracing translation with RPython
Adding a JIT to an RPython interpreter

... pc := 0 while 1:

    instr := load_next_instruction(pc)
    if instr == POP:
        stack.pop()
        pc += 1
    elif instr == BRANCH:
        off = load_branch_jump(pc)
        pc += off
    elif ...:
        ...

Observation: interpreters are big loops.
...  
pc := 0  
while 1:  
    jit_merge_point(pc)  
    instr := load_next_instruction(pc)  
    if instr == POP:  
        stack.pop()  
        pc += 1  
    elif instr == BRANCH:  
        off = load_branch_jump(pc)  
        if off < 0: can_enter_jit(pc)  
        pc += off  
    elif ...:  
        ...

Observation: interpreters are big loops.
RPython translation

Interpreter

Optimised Interpreter

JIT

RPython translator
RPython translation

- Interpreter
- Language Interpreter
- Trace Interpreter

RPython translator
User program (lang FL)

```python
if x < 0:
    x = x + 1
else:
    x = x + 2
x = x + 3
```
Tracing JITs

<table>
<thead>
<tr>
<th>User program (lang FL)</th>
<th>Trace when x is set to 6</th>
</tr>
</thead>
</table>
| if x < 0:
  x = x + 1
else:
  x = x + 2
x = x + 3 | guard_type(x, int)
guard_not_less_than(x, 0)
guard_type(x, int)
x = int_add(x, 2)
guard_type(x, int)
x = int_add(x, 3) |
Tracing JITs

<table>
<thead>
<tr>
<th>User program (lang FL)</th>
<th>Optimised trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>if x &lt; 0:</td>
<td>guard_type(x, int)</td>
</tr>
<tr>
<td>x = x + 1</td>
<td>guard_not_less_than(x, 0)</td>
</tr>
<tr>
<td>else:</td>
<td>x = int_add(x, 5)</td>
</tr>
<tr>
<td>x = x + 2</td>
<td></td>
</tr>
<tr>
<td>x = x + 3</td>
<td></td>
</tr>
</tbody>
</table>
Meta-tracing VM components

1. Start
2. Detect hot loop
3. Execute and trace
4. Convert trace to machine code
Meta-tracing JITs

**FL Interpreter**

```python
program_counter = 0; stack = []
vars = {...}
while True:
    jit_merge_point(program_counter)
    instr = load_instruction(program_counter)
    if instr == INSTR_VAR_GET:
        stack.push(
            vars[read_var_name_from_instruction()])
        program_counter += 1
    elif instr == INSTR_VAR_SET:
        vars[read_var_name_from_instruction()]
        = stack.pop()
        program_counter += 1
    elif instr == INSTR_INT:
        stack.push(read_int_from_instruction())
        program_counter += 1
    elif instr == INSTR_LESS_THAN:
        rhs = stack.pop()
        lhs = stack.pop()
        if isinstance(lhs, int) and isinstance(rhs, int):
            if lhs < rhs:
                stack.push(True)
            else:
                stack.push(False)
        else: ...
        program_counter += 1
    elif instr == INSTR_IF:
        result = stack.pop()
        if result == True:
            program_counter += 1
        else:
            program_counter +=
            read_jump_if_instruction()
    elif instr == INSTR_ADD:
        lhs = stack.pop()
        rhs = stack.pop()
        if isinstance(lhs, int)
            and isinstance(rhs, int):
                stack.push(lhs + rhs)
        else: ...
        program_counter += 1
```
**FL Interpreter**

```python
program_counter = 0; stack = []
vars = {...}
while True:
    jit_merge_point(program_counter)
    instr = load_instruction(program_counter)
    if instr == INSTR_VAR_GET:
        stack.push(  
            vars[read_var_name_from_instruction()]  
        )
        program_counter += 1
    elif instr == INSTR_VAR_SET:
        vars[read_var_name_from_instruction()]  
        = stack.pop()
        program_counter += 1
    elif instr == INSTR_INT:
        stack.push(read_int_from_instruction())
        program_counter += 1
    elif instr == INSTR_LESS_THAN:
        rhs = stack.pop()
        lhs = stack.pop()
        if isinstance(lhs, int) and isinstance(rhs, int):  
            if lhs < rhs:
                stack.push(True)
            else:
                stack.push(False)
        else: ...
        program_counter += 1
```
### FL Interpreter

```python
program_counter = 0; stack = []
vars = {...}
while True:
    jit_merge_point(program_counter)
    instr = load_instruction(program_counter)
    if instr == INSTR_VAR_GET:
        stack.push(
            vars[read_var_name_from_instruction()]
        )
        program_counter += 1
    elif instr == INSTR_VAR_SET:
        vars[read_var_name_from_instruction()]
        = stack.pop()
        program_counter += 1
    elif instr == INSTR_INT:
        stack.push(read_int_from_instruction())
        program_counter += 1
    elif instr == INSTR_LESS_THAN:
        rhs = stack.pop()
        lhs = stack.pop()
        if isinstance(lhs, int) and isinstance(rhs, int):
            if lhs < rhs:
                stack.push(True)
            else:
                stack.push(False)
        else: ...
        program_counter += 1
```

### User program (lang FL)

```python
if x < 0:
    x = x + 1
else:
    x = x + 2
x = x + 3
```

if x < 0:
    x = x + 1
else:
    x = x + 2
x = x + 3
Meta-tracing JITs

**FL Interpreter**

```python
program_counter = 0; stack = []
vars = {...}
while True:
    jit_merge_point(program_counter)
    instr = load_instruction(program_counter)
    if instr == INSTR_VAR_GET:
        stack.push(
            vars[read_var_name_from_instruction()]
        )
        program_counter += 1
    elif instr == INSTR_VAR_SET:
        vars[read_var_name_from_instruction()] = stack.pop()
        program_counter += 1
    elif instr == INSTR_INT:
        stack.push(read_int_from_instruction())
        program_counter += 1
    elif instr == INSTR_LESS_THAN:
        rhs = stack.pop()
        lhs = stack.pop()
        if isinstance(lhs, int) and isinstance(rhs, int):
            if lhs < rhs:
                stack.push(True)
            else:
                stack.push(False)
        else: ...
        program_counter += 1
```

**Initial trace**

```plaintext
v0 = <program_counter>
v1 = <stack>
v2 = <vars>
v3 = load_instruction(v0)
guard_eq(v3, INSTR_VAR_GET)
v4 = dict_get(v2, "x")
list_append(v1, v4)
v5 = add(v0, 1)
v6 = load_instruction(v5)
guard_eq(v6, INSTR_INT)
list_append(v1, 0)
v7 = add(v5, 1)
v8 = load_instruction(v7)
guard_eq(v8, INSTR_LESS_THAN)
v9 = list_pop(v1)
v10 = list_pop(v1)
guard_type(v9, int)
guard_type(v10, int)
guard_not_less_than(v9, v10)
list_append(v1, False)
v11 = add(v7, 1)
v12 = load_instruction(v11)
guard_eq(v12, INSTR_IF)
v13 = list_pop(v1)
guard_false(v13)
...
Initial trace in full

\[

t_{0} = \text{<program_counter>}
t_{1} = \text{<stack>}
t_{2} = \text{<vars>}
t_{3} = \text{load_instruction}(t_{0})
guard_{eq}(t_{3}, \text{INSTR_VAR_GET})
t_{4} = \text{dict_get}(t_{2}, "x")
list_append(t_{1}, t_{4})
t_{5} = \text{add}(t_{0}, 1)
t_{6} = \text{load_instruction}(t_{5})
guard_{eq}(t_{6}, \text{INSTR_INT})
list_append(t_{1}, 0)
t_{7} = \text{add}(t_{5}, 1)
t_{8} = \text{load_instruction}(t_{7})
guard_{eq}(t_{8}, \text{INSTR_LESS_THAN})
t_{9} = \text{list_pop}(t_{1})
t_{10} = \text{list_pop}(t_{1})
guard_{type}(t_{9}, \text{int})
guard_{type}(t_{10}, \text{int})
guard_{not_less_than}(t_{9}, t_{10})
list_append(t_{1}, \text{False})
t_{11} = \text{add}(t_{7}, 1)
t_{12} = \text{load_instruction}(t_{11})
guard_{eq}(t_{12}, \text{INSTR_IF})
t_{13} = \text{list_pop}(t_{1})
guard_{false}(t_{13})
t_{14} = \text{add}(t_{11}, 2)
\]

\[

t_{15} = \text{load_instruction}(t_{14})
guard_{eq}(t_{15}, \text{INSTR_VAR_GET})
t_{16} = \text{dict_get}(t_{2}, "x")
list_append(t_{1}, t_{16})
t_{17} = \text{add}(t_{14}, 1)
t_{18} = \text{load_instruction}(t_{17})
guard_{eq}(t_{18}, \text{INSTR_INT})
list_append(t_{1}, 2)
t_{19} = \text{add}(t_{17}, 1)
t_{20} = \text{load_instruction}(t_{19})
guard_{eq}(t_{20}, \text{INSTR_ADD})
list_pop(t_{1})
t_{21} = \text{list_pop}(t_{1})
t_{22} = \text{list_pop}(t_{1})
guard_{type}(t_{21}, \text{int})
guard_{type}(t_{22}, \text{int})
t_{23} = \text{add}(t_{22}, t_{21})
list_append(t_{1}, t_{23})
t_{24} = \text{add}(t_{19}, 1)
t_{25} = \text{load_instruction}(t_{24})
guard_{eq}(t_{25}, \text{INSTR_VAR_GET})
t_{26} = \text{list_pop}(t_{1})
dict_set(t_{2}, "x", t_{26})
t_{27} = \text{add}(t_{24}, 1)
t_{28} = \text{load_instruction}(t_{27})
guard_{eq}(t_{28}, \text{INSTR_VAR_GET})
t_{29} = \text{dict_get}(t_{2}, "x")
list_append(t_{1}, t_{29})
t_{30} = \text{add}(t_{27}, 1)
t_{31} = \text{load_instruction}(t_{30})
guard_{eq}(t_{31}, \text{INSTR_INT})
list_append(t_{1}, 3)
t_{32} = \text{add}(t_{30}, 1)
t_{33} = \text{load_instruction}(t_{32})
guard_{eq}(t_{33}, \text{INSTR_ADD})
t_{34} = \text{list_pop}(t_{1})
t_{35} = \text{list_pop}(t_{1})
guard_{type}(t_{34}, \text{int})
guard_{type}(t_{35}, \text{int})
t_{36} = \text{add}(t_{35}, t_{34})
list_append(t_{1}, t_{36})
t_{37} = \text{add}(t_{33}, 1)
t_{38} = \text{load_instruction}(t_{37})
guard_{eq}(t_{38}, \text{INSTR_VAR_SET})
t_{39} = \text{list_pop}(t_{1})
dict_set(t_{2}, "x", t_{39})
t_{40} = \text{add}(t_{37}, 1)
\]
Removing constants (from \texttt{jit_merge_point})

\begin{verbatim}
v1 = <stack>
v2 = <vars>
v4 = dict_get(v2, "x")
list_append(v1, v4)
list_append(v1, 0)
v9 = list_pop(v1)
v10 = list_pop(v1)
guard_type(v9, int)
guard_type(v10, int)
guard_not_less_than(v9, v10)
list_append(v1, False)
v13 = list_pop(v1)
guard_false(v13)
v16 = dict_get(v2, "x")
list_append(v1, v16)
list_append(v1, 2)
v21 = list_pop(v1)
v22 = list_pop(v1)
guard_type(v21, int)
guard_type(v22, int)
v23 = add(v22, v21)
list_append(v1, v23)
v26 = list_pop(v1)
dict_set(v2, "x", v26)
v29 = dict_get(v2, "x")
list_append(v1, v29)
\end{verbatim}
List folded trace

v1 = <stack>
v2 = <vars>
v4 = dict_get(v2, "x")
guard_type(v4, int)
guard_not_less_than(v4, 0)
v16 = dict_get(v2, "x")
guard_type(v16, int)
v23 = add(v16, 2)
dict_set(v2, "x", v23)
v29 = dict_get(v2, "x")
guard_type(v29, int)
v36 = add(v29, 3)
dict_set(v2, "x", v36)
Optimisation #2 & #3

List folded trace

```
v1 = <stack>
v2 = <vars>
v4 = dict_get(v2, "x")
guard_type(v4, int)
guard_not_less_than(v4, 0)
v16 = dict_get(v2, "x")
guard_type(v16, int)
v23 = add(v16, 2)
dict_set(v2, "x", v23)
v29 = dict_get(v2, "x")
guard_type(v29, int)
v36 = add(v29, 3)
dict_set(v2, "x", v36)
```

Dict folded trace

```
v1 = <stack>
v2 = <vars>
v4 = dict_get(v2, "x")
guard_type(v4, int)
guard_not_less_than(v4, 0)
v23 = add(v4, 2)
guard_type(v23, int)
v36 = add(v23, 3)
dict_set(v2, "x", v36)
```
Type folded trace

v1 = <stack>
v2 = <vars>
v4 = dict_get(v2, "x")

\[
guard\_type(v4, \text{int})
\]

\[
guard\_not\_less\_than(v4, 0)
\]

\[
v23 = \text{add}(v4, 2)
\]

\[
v36 = \text{add}(v23, 3)
\]

\[
dict\_set(v2, "x", v36)
\]
Optimisation #4 & #5

Type folded trace

v1 = <stack>
v2 = <vars>
v4 = dict_get(v2, "x")
guard_type(v4, int)
guard_not_less_than(v4, 0)
v23 = add(v4, 2)
v36 = add(v23, 3)
dict_set(v2, "x", v36)

Arithmetic folded trace

v1 = <stack>
v2 = <vars>
v4 = dict_get(v2, "x")
guard_type(v4, int)
guard_not_less_than(v4, 0)
v23 = add(v4, 5)
dict_set(v2, "x", v23)
### Optimisation #4 & #5

<table>
<thead>
<tr>
<th>Type folded trace</th>
<th>Arithmetic folded trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1 = &lt;stack&gt;</td>
<td>v1 = &lt;stack&gt;</td>
</tr>
<tr>
<td>v2 = &lt;vars&gt;</td>
<td>v2 = &lt;vars&gt;</td>
</tr>
<tr>
<td>v4 = dict_get(v2, &quot;x&quot;)</td>
<td>v4 = dict_get(v2, &quot;x&quot;)</td>
</tr>
<tr>
<td>guard_type(v4, int)</td>
<td>guard_type(v4, int)</td>
</tr>
<tr>
<td>guard_not_less_than(v4, 0)</td>
<td>guard_not_less_than(v4, 0)</td>
</tr>
<tr>
<td>v23 = add(v4, 2)</td>
<td>v23 = add(v4, 5)</td>
</tr>
<tr>
<td>v36 = add(v23, 3)</td>
<td>dict_set(v2, &quot;x&quot;, v23)</td>
</tr>
<tr>
<td>dict_set(v2, &quot;x&quot;, v36)</td>
<td></td>
</tr>
</tbody>
</table>

Trace optimisation: from 72 trace elements to 7.
Runtime composition recap
Runtime composition recap

PL X

Interpreters

Glue

PL Y

Meta-tracing

PL Z

Interpreter

Tracing JIT
Benchmarking VM composition

Warning: draft numbers ahead
### Absolute timing comparison

<table>
<thead>
<tr>
<th>VM</th>
<th>Benchmark</th>
<th>Python</th>
<th>Prolog</th>
<th>Python → Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPython-SWI</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallFunc</td>
<td>0.125s ± 0.006</td>
<td>0.257s ± 0.001</td>
<td>28.893s ± 0.175</td>
<td></td>
</tr>
<tr>
<td>Loop1Arg0Result</td>
<td>2.924s ± 0.215</td>
<td>7.352s ± 0.037</td>
<td>9.310s ± 0.065</td>
<td></td>
</tr>
<tr>
<td>Loop1Arg1Result</td>
<td>4.184s ± 0.028</td>
<td>18.890s ± 0.082</td>
<td>20.865s ± 0.050</td>
<td></td>
</tr>
<tr>
<td>NonDetLoop1Arg1Result</td>
<td>7.531s ± 0.065</td>
<td>18.643s ± 0.159</td>
<td>667.682s ± 5.594</td>
<td></td>
</tr>
<tr>
<td>TermConstruction</td>
<td>264.415s ± 1.815</td>
<td>48.819s ± 0.208</td>
<td>2185.150s ± 14.251</td>
<td></td>
</tr>
<tr>
<td>Lists</td>
<td>9.374s ± 0.046</td>
<td>25.148s ± 0.182</td>
<td>2207.304s ± 12.344</td>
<td></td>
</tr>
<tr>
<td><strong>PyPy-SWI</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallFunc</td>
<td>0.001s ± 0.000</td>
<td>0.256s ± 0.001</td>
<td>4.744s ± 0.062</td>
<td></td>
</tr>
<tr>
<td>Loop1Arg0Result</td>
<td>0.085s ± 0.000</td>
<td>7.358s ± 0.067</td>
<td>7.583s ± 0.103</td>
<td></td>
</tr>
<tr>
<td>Loop1Arg1Result</td>
<td>0.112s ± 0.000</td>
<td>18.988s ± 0.115</td>
<td>18.519s ± 0.111</td>
<td></td>
</tr>
<tr>
<td>NonDetLoop1Arg1Result</td>
<td>0.481s ± 0.007</td>
<td>18.737s ± 0.247</td>
<td>74.833s ± 1.856</td>
<td></td>
</tr>
<tr>
<td>TermConstruction</td>
<td>6.111s ± 0.029</td>
<td>48.897s ± 0.370</td>
<td>166.107s ± 3.218</td>
<td></td>
</tr>
<tr>
<td>Lists</td>
<td>1.102s ± 0.010</td>
<td>25.104s ± 0.189</td>
<td>163.505s ± 3.586</td>
<td></td>
</tr>
<tr>
<td><strong>Unipycaction</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallFunc</td>
<td>0.001s ± 0.000</td>
<td>0.006s ± 0.001</td>
<td>0.001s ± 0.000</td>
<td></td>
</tr>
<tr>
<td>Loop1Arg0Result</td>
<td>0.085s ± 0.000</td>
<td>0.086s ± 0.000</td>
<td>0.087s ± 0.000</td>
<td></td>
</tr>
<tr>
<td>Loop1Arg1Result</td>
<td>0.112s ± 0.000</td>
<td>0.114s ± 0.000</td>
<td>0.115s ± 0.000</td>
<td></td>
</tr>
<tr>
<td>NonDetLoop1Arg1Result</td>
<td>0.500s ± 0.002</td>
<td>0.548s ± 0.064</td>
<td>2.674s ± 0.010</td>
<td></td>
</tr>
<tr>
<td>TermConstruction</td>
<td>6.053s ± 0.218</td>
<td>2.444s ± 0.002</td>
<td>36.069s ± 0.171</td>
<td></td>
</tr>
<tr>
<td>Lists</td>
<td>0.845s ± 0.002</td>
<td>1.416s ± 0.003</td>
<td>5.056s ± 0.026</td>
<td></td>
</tr>
<tr>
<td><strong>Jython-tuProlog</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallFunc</td>
<td>0.088s ± 0.002</td>
<td>3.050s ± 0.036</td>
<td>52.294s ± 0.371</td>
<td></td>
</tr>
<tr>
<td>Loop1Arg0Result</td>
<td>1.078s ± 0.007</td>
<td>206.590s ± 2.884</td>
<td>199.963s ± 1.784</td>
<td></td>
</tr>
<tr>
<td>Loop1Arg1Result</td>
<td>2.145s ± 0.175</td>
<td>293.311s ± 4.270</td>
<td>294.781s ± 4.746</td>
<td></td>
</tr>
<tr>
<td>NonDetLoop1Arg1Result</td>
<td>7.939s ± 0.341</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>TermConstruction</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>Lists</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>VM</td>
<td>Benchmark</td>
<td>Python→Prolog</td>
<td>Python→Prolog</td>
<td>Python→Prolog</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------</td>
<td>---------------</td>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Python</td>
<td>Prolog</td>
<td>Unipycation</td>
</tr>
<tr>
<td>CPython-SWI</td>
<td>SmallFunc</td>
<td>231.770× ±10.154</td>
<td>112.567× ±0.934</td>
<td>27821.079× ±1896.725</td>
</tr>
<tr>
<td></td>
<td>Loop1Arg0Result</td>
<td>3.184× ±0.232</td>
<td>1.266× ±0.011</td>
<td>107.591× ±0.779</td>
</tr>
<tr>
<td></td>
<td>Loop1Arg1Result</td>
<td>4.987× ±0.039</td>
<td>1.105× ±0.006</td>
<td>181.899× ±0.444</td>
</tr>
<tr>
<td></td>
<td>NondetLoop1Arg1Result</td>
<td>88.654× ±1.026</td>
<td>35.814× ±0.389</td>
<td>249.737× ±2.244</td>
</tr>
<tr>
<td></td>
<td>TermConstruction</td>
<td>8.264× ±0.081</td>
<td>44.760× ±0.348</td>
<td>60.583× ±0.487</td>
</tr>
<tr>
<td></td>
<td>Lists</td>
<td>235.459× ±1.742</td>
<td>87.772× ±0.789</td>
<td>436.609× ±3.494</td>
</tr>
<tr>
<td>PyPy-SWI</td>
<td>SmallFunc</td>
<td>5747.739× ±101.546</td>
<td>18.515× ±0.254</td>
<td>4568.213× ±304.275</td>
</tr>
<tr>
<td></td>
<td>Loop1Arg0Result</td>
<td>88.765× ±1.271</td>
<td>1.031× ±0.019</td>
<td>87.636× ±1.316</td>
</tr>
<tr>
<td></td>
<td>Loop1Arg1Result</td>
<td>164.697× ±1.127</td>
<td>0.975× ±0.009</td>
<td>161.446× ±1.006</td>
</tr>
<tr>
<td></td>
<td>NondetLoop1Arg1Result</td>
<td>155.478× ±4.537</td>
<td>3.994× ±0.111</td>
<td>27.990× ±0.705</td>
</tr>
<tr>
<td></td>
<td>TermConstruction</td>
<td>27.179× ±0.522</td>
<td>3.397× ±0.068</td>
<td>4.605× ±0.091</td>
</tr>
<tr>
<td></td>
<td>Lists</td>
<td>148.418× ±3.265</td>
<td>6.513× ±0.142</td>
<td>32.342× ±0.703</td>
</tr>
<tr>
<td>Unipycation</td>
<td>SmallFunc</td>
<td>1.295× ±0.086</td>
<td>0.182× ±0.036</td>
<td>1.000×</td>
</tr>
<tr>
<td></td>
<td>Loop1Arg0Result</td>
<td>1.020× ±0.001</td>
<td>1.012× ±0.002</td>
<td>1.000×</td>
</tr>
<tr>
<td></td>
<td>Loop1Arg1Result</td>
<td>1.025× ±0.002</td>
<td>1.002× ±0.002</td>
<td>1.000×</td>
</tr>
<tr>
<td></td>
<td>NondetLoop1Arg1Result</td>
<td>5.349× ±0.035</td>
<td>4.879× ±0.631</td>
<td>1.000×</td>
</tr>
<tr>
<td></td>
<td>TermConstruction</td>
<td>5.959× ±0.224</td>
<td>14.756× ±0.069</td>
<td>1.000×</td>
</tr>
<tr>
<td></td>
<td>Lists</td>
<td>5.982× ±0.034</td>
<td>3.569× ±0.019</td>
<td>1.000×</td>
</tr>
<tr>
<td>Jython-tuProlog</td>
<td>SmallFunc</td>
<td>592.904× ±14.602</td>
<td>17.143× ±0.259</td>
<td>50354.204× ±3330.993</td>
</tr>
<tr>
<td></td>
<td>Loop1Arg0Result</td>
<td>185.460× ±2.182</td>
<td>0.968× ±0.017</td>
<td>2310.844× ±21.996</td>
</tr>
<tr>
<td></td>
<td>Loop1Arg1Result</td>
<td>137.427× ±11.805</td>
<td>1.005× ±0.022</td>
<td>2569.873× ±41.331</td>
</tr>
<tr>
<td></td>
<td>NondetLoop1Arg1Result</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td></td>
<td>TermConstruction</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td></td>
<td>Lists</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
</tr>
</tbody>
</table>
What can we use this for?
What can we use this for?

COBOL  Java
What can we use this for?

"Big Bang" translation

COBOL  \(\rightarrow\)  Java
What can we use this for?

COBOL  →  COava  →  Java
What can we use this for?

Gradual migration

COBOL  COava  Java

http://soft-dev.org/
Editor: incremental semantic analysis on ASTs and code generation
Editor: incremental semantic analysis on ASTs and code generation

+ VMs: uncovering common idioms
Summary

Unipyrcation

Python
syntax
runtime

Prolog
syntax
runtime

Unipyrcation
syntax
runtime
Summary

- Syntax
- Runtime

Python
- Syntax
- Runtime

Prolog
- Syntax
- Runtime

Unipycation
- Syntax
- Runtime

Language boxes

Composed meta-tracing VMs
Thanks for listening

http://soft-dev.org/