The Highs and Lows of Macros in a Modern Language

Laurence Tratt

Software Development Team
2016-08-09
A perfect programming language
Solution

A new programming language
Reality
Reality

Another imperfect programming language
What to expect from this talk

1. What happens when you put macros into a modern programming language?
1. What happens when you put macros into a modern programming language?

2. If it doesn’t work out well, is there an alternative?
Part I

Defining the area
What is a macro?

It's complicated...

'...et simplification'

A calculation that happens at compile-time.
It’s complicated...
What is a macro?

It’s complicated...

Let’s simplify to “a calculation that happens at compile-time”.
This C fragment:

```c
#define sq2(y) ((y) * (y))

int main() {
    printf("%d\n", sq2(3));
}
```

is preprocessed to:

```c
int main() {
    printf("%d\n", ((3) * (3)));
}
```

and then compiled.
Some clever (and useful) things are possible e.g.:

```c
#define TRY { \
    jmp_buf _env; \
    if (setjmp(_env) == 0) { \
        add_exception_frame(_env);
#define CATCH(v) \n    remove_exception_frame(); \n    } \n    else { \n        (v) = read_and_reset_exception();
#define TRY_END } }
```
Some clever (and useful) things are possible e.g.:

```c
#define TRY { \
    jmp_buf _env; \
    if (setjmp(_env) == 0) { \
        add_exception_frame(_env); \
    } 
#define CATCH(v) \n    remove_exception_frame(); \
    } \nelse { \
    (v) = read_and_reset_exception(); 
#define TRY_END } 
```

can be used – fairly naturally – for:

```c
Exception *e;
TRY {
    ...
} CATCH (e) {
    ...
}
TRY_END
```
What does the following print out?

```c
#define sq2(y) (y * y)
int main() {
    printf("%d\n", sq2(3));
    printf("%d\n", sq2(1+2));
}
```
What does the following print out?

```c
#define sq2(y) (y * y)
int main() {
    printf("%d\n", sq2(3));
    printf("%d\n", sq2(1+2));
}
```

*Obviously 9, 5?!
What does the following print out?

```c
#define sq2(y) (y * y)
int main() {
    printf("%d\n", sq2(3));
    printf("%d\n", sq2(1+2));
}
```

*Obviously* 9, 5?! What about:

```c
#define sq2(y) ((y) * (y))
typedef struct { int y; } C;
int main() {
    C x;
    x.y = 3;
    printf("%d\n", sq2(++x.y));
}
```
What does the following print out?

```c
#define sq2(y) (y * y)
int main() {
    printf("%d\n", sq2(3));
    printf("%d\n", sq2(1+2));
}
```

**Obviously** 9, 5?!

What about:

```c
#define sq2(y) ((y) * (y))
typedef struct { int y; } C;
int main() {
    C x;
    x.y = 3;
    printf("%d\n", sq2(++x.y));
}
```

**Obviously** 20?!
What does the following print out?

```c
#define sq2(y) (y * y)
int main() {
    printf("%d\n", sq2(3));
    printf("%d\n", sq2(1+2));
}
```

Obviously 9, 5?!

What about:

```c
#define sq2(y) ((y) * (y))
typedef struct { int y; } C;
int main() {
    C x;
    x.y = 3;
    printf("%d\n", sq2(++x.y));
}
```

Obviously 20?!

There are other problems too, but you get the idea...
**Heterogeneous:** where the meta-programming language/system (e.g. the C preprocessor) is different than the main language/system (e.g. C).

**Homogeneous:** where the two are the same.

Crudely: heterogeneous is powerful, but difficult to use, and unsafe; homogeneous is safe(r) and easier to use.

[See Sheard 2003 ‘Accomplishments and Research Challenges in Meta-programming’]
The ‘Lisp’ family is huge.
The ‘Lisp’ family is huge. In a typical-ish Lisp, one might do:

```
(defun sq2 (e)
  (list '+ e e))
```

```
(print (macroexpand '(* (+ 1 2) (+ 1 2))))
(print (macroexpand '(sq2 (+ 1 2))))
```

which will print:

```
(* (1 2) (1 2))
(* (1 2) (1 2))
```

Note: everything is done on trees.
For decades, macro research was Lisp. Why?
For decades, macro research was Lisp. Why?

Brackets (maybe); homoiconicity (definitely).
The tumbleweed years

For decades, macro research was Lisp. Why?

Brackets (maybe); homoiconicity (definitely).

Until MetaML and successors, including Template Haskell.
Part II

What happens when you put macros into a modern programming language?
import sys

func main():
    sys::println("hello world")
Summary: Python + TH-esque macros

```python
import Sys
func main():
    Sys::println("hello world")
```
Code (as trees, not text) is programatically generated.
Compile-time Meta-programming / Macros

Code (as trees, not text) is programatically generated.

| Expression | 2 + 3 | evaluates to 5 (as one expects). |
| Splice     | $<x>$ | evaluates $x$ at compile-time; the AST returned overwrites the splice. |
| Insertion  | [ | 2 + ${x}$ | ] | ‘inserts’ the AST $x$ into the AST being created by the quasi-quotes. |
When do things execute?

When are $x$ and $y$ evaluated?

```go
func main()
    y
```
We want:

\[ \text{power3} := \text{mk\_power}(3) \]

to be compiled to:

\[ \text{power3} := \text{func (x)}:\]
\[ \quad \text{return } x \times x \times x \times 1 \]

How to do it?
The `printf` Function
What can we use this stuff for?

1. Very little syntax
2. A (restrictive) static type system.

Either is true in modern dynamically typed languages.

Do macros have uses?

HTTP://SOFT-DEV.ORG/
IMHO, macros are useful if your language has:

1. very little syntax
IMHO, macros are useful if your language has:

1. very little syntax and/or
2. a (restrictive) static type system.
IMHO, macros are useful if your language has:

1. very little syntax and/or
2. a (restrictive) static type system.

Neither is true in modern dynamically typed languages.
What can we use this stuff for?

IMHO, macros are useful if your language has:
1. very little syntax and/or
2. a (restrictive) static type system.

Neither is true in modern dynamically typed languages.

Do macros have uses?
Embedding DSLs

**Splice** $\langle x \rangle$ evaluates $x$ at compile-time; the AST returned overwrites the splice.

**Quasi-quote** [ | 2 + 3 | ] evaluates to a *hygienic* AST representing $2 + 3$.

**Insertion** [ | 2 + $\{x\}$ | ] ‘inserts’ the AST $x$ into the AST being created by the quasi-quotes.
**Embedding DSLs**

- **Splice** \(\$\langle x\rangle\)
evaluates \(x\) at compile-time; the AST returned overwrites the splice.

- **Quasi-quote** \([| 2 + 3 |]\)
evaluates to a *hygienic* AST representing \(2 + 3\).

- **Insertion** \([| 2 + \{x\} |]\)‘inserts’ the AST \(x\) into the AST being created by the quasi-quotes.

- **DSL blocks** \(\$\langle\langle x\rangle\rangle: \ y\)
pass the string \(\ y\) to the function \(x\) at compile-time.
Building a DSL
We normally assume that compilers are perfect
DSL debugging

We normally assume that compilers are perfect

DSL compilers are probably imperfect
We normally assume that compilers are perfect

DSL compilers are probably imperfect

Are errors due to the user or the compiler?
Static error reporting
Src infos are a triple: \((file\ ID, char\ offset, char\ span)\)

Threaded throughout the compiler:
1. Each token/lexeme has one src info
2. Each parse tree has more than one src info
3. Each bytecode has more than one src info
Dynamic scoping is dangerous.
Dynamic scoping is dangerous.

Can it be made safe?
Three *relative* meta-levels describe everything:

<table>
<thead>
<tr>
<th>Meta-level</th>
<th>Description</th>
</tr>
</thead>
</table>

[HTTP://SOFT-DEV.ORG/](http://soft-dev.org/)
Three *relative* meta-levels describe everything:

<table>
<thead>
<tr>
<th>Meta-level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Normal compilation</td>
</tr>
</tbody>
</table>
Three *relative* meta-levels describe everything:

<table>
<thead>
<tr>
<th>Meta-level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Splicing ($&lt;\ldots&gt;$)</td>
</tr>
<tr>
<td>0</td>
<td>Normal compilation</td>
</tr>
</tbody>
</table>
Three *relative* meta-levels describe everything:

<table>
<thead>
<tr>
<th>Meta-level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Splicing ($&lt;\ldots&gt;$)</td>
</tr>
<tr>
<td>0</td>
<td>Normal compilation</td>
</tr>
<tr>
<td>+1</td>
<td>Quasi-quoting ([</td>
</tr>
</tbody>
</table>
src infos make debugging possible.
1. src infos make debugging possible.
2. rename enables building huge, name-safe trees.
What works well?

1. src infos make debugging possible.
2. rename enables building huge, name-safe trees.
3. DSL layers work and are useful.
What works well?

1. src infos make debugging possible.
2. rename enables building huge, name-safe trees.
3. DSL layers work and are useful.
4. The compiler is surprisingly simple.
What works well?

1. src infos make debugging possible.
2. rename enables building huge, name-safe trees.
3. DSL layers work and are useful.
4. The compiler is surprisingly simple (though calculations with names make my head hurt).
1. Delimiters are *far* too ugly for repeated use.
What doesn’t work?

1. Delimiters are *far* too ugly for repeated use.
2. Macro evaluation is top-to-bottom. DSLs can’t validate e.g.:

   $<<SQL>>\text{SELECT } c1 \text{ FROM } t$
   $<<SQL>>\text{CREATE TABLE } t ( \text{c2 STR } )$
What doesn’t work?

1. Delimiters are *far* too ugly for repeated use.

2. Macro evaluation is top-to-bottom. DSLs can’t validate e.g.:
   
   ```
   $<<SQL>>SELECT c1 FROM t
   $<<SQL>>CREATE TABLE t ( c2 STR )
   ```

3. Syntax composition is nearly impossible.
What doesn’t work?

1. Delimiters are *far* too ugly for repeated use.
2. Macro evaluation is top-to-bottom. DSLs can’t validate e.g.:
   \[
   \$\langle\langle SQL \rangle\rangle \langle SELECT c1 FROM t \rangle
   \]
   \[
   \$\langle\langle SQL \rangle\rangle \langle CREATE TABLE t ( c2 STR ) \rangle
   \]
3. Syntax composition is nearly impossible.
4. Performance for mildly complex DSLs is poor.
Where do we go from here?
Part III

A different way
Language composition: two levels of challenge
Language composition: two levels of challenge

Tooling
Language composition: two levels of challenge

Tooling

Language friction
Tooling challenges

Python

PHP

PyHyp
Tooling challenges

- Python: Syntax, Runtime
- PHP: Syntax, Runtime
- PyHyp: Syntax, Runtime
Tooling challenges

Language boxes

Python
  syntax
  runtime

PHP
  syntax
  runtime

PyHyp
  syntax
  runtime

32 / 59 HTTP://SOFT-DEV.ORG/
Tooling challenges

Language boxes

Composed
meta-tracing
VMs

Syntax

- Python
  - runtime

Syntax

- PHP
  - runtime

Syntax

PyHyp
- runtime
Syntax composition

\[ PL \ X \]
\begin{verbatim}
<grammar>
expr ::= ... 
term ::= ...
    | ...
    | ...
func ::= ...
\end{verbatim}

\[ PL \ Y \]
\begin{verbatim}
<program>
for (j : js) {
    doStuff();
}
.
.
.
\end{verbatim}
Syntax composition

Parser

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

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

3yNTAxCOMPOSITION

3yNTAxCOMPOSITION
Syntax composition

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

**PLY Y**
<program>
for (j : js) {
    doStuff();
} 
.
.
.

Parser

Parse Tree
Syntax composition

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

PLY Y
<program>
for (j : js) {
    doStuff();
}
.
.
.
LR
Parse Tree
Syntax composition

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

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

LR Parse Tree

Undefined
Syntax composition

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

**PLY Y**
<program>
for (j : js) {
    doStuff();
}

Generalised Parse Tree

Parse Tree
Syntax composition

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

Generalised Parse Tree
Ambiguous

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

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

Parse Tree
Syntax composition

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

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

PEG
Parse Tree
Syntax composition

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

**PLY Y**
<program>
for (j : js) {
    doStuff();
}

Parse Tree
The only choice?
The only choice?

SDE
Challenge: SDE’s power + a text editor feel?
Runtime composition
Runtime composition

PL X
Interpreter

PL Y
Interpreter

C/C++
Runtime composition

PL X

Interpreter

Too slow

PLY

Interpreter

C/C++
Runtime composition

PL X
Interpreted
C/C++
JIT Compiler

PL Y
Interpreted
C/C++
JIT Compiler
Runtime composition

Too much engineering
Runtime composition

PL X
Interpreter

PL Y
Interpreter

JVM/CLR
JIT Compiler
Runtime composition

Semantic mismatch

PL X

Interpreted

PL Y

Interpreted

JVM/CLR

JIT Compiler

Semantic mismatch

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

- PL X
- PL Y
- Interpreters
- Glue
- Meta-tracing
- PL Z
- Interpreter
- Tracing JIT

RPython
Interpreter
Tracing JIT
PL Y
Interpreters
Glue
PL Z
Meta-tracing
Meta-tracing translation with RPython

Interpreter
Meta-tracing translation with RPython
Meta-tracing translation with RPython

Interpreter
Optimised Interpreter
JIT

http://soft-dev.org/
Meta-tracing translation with RPython

Interpreter
Optimised
Interpreter
JIT
RPython
translator

you write this

Optimised Interpreter

JIT

http://soft-dev.org/
Meta-tracing translation with RPython

you write this

you get this for free

Optimised Interpreter

JIT

you get this for free
Adding a JIT compiler 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
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>
<tbody>
<tr>
<td>if x &lt; 0:</td>
<td>guard_type(x, int)</td>
</tr>
<tr>
<td></td>
<td>guard_not_less_than(x, 0)</td>
</tr>
<tr>
<td>else:</td>
<td>guard_type(x, int)</td>
</tr>
<tr>
<td>x = x + 2</td>
<td>x = int_add(x, 2)</td>
</tr>
<tr>
<td>x = x + 3</td>
<td>guard_type(x, int)</td>
</tr>
<tr>
<td></td>
<td>x = int_add(x, 3)</td>
</tr>
</tbody>
</table>
## User program (lang FL) | Optimised trace
--- | ---
if $x < 0$:
    $x = x + 1$
else:
    $x = x + 2$
$x = x + 3$

`guard_type(x, int)`

`guard_not_less_than(x, 0)`

$x = \text{int_add}(x, 5)$
Meta-tracing VM components

1. Start
2. Detect hot loop
3. Execute and trace
4. Convert trace to machine code
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
```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
```
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
```

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

User program (lang FL)

43 / 59
**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**

```python
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

\[
\begin{align*}
v_0 &= \text{<program_counter>} \\
v_1 &= \text{<stack>} \\
v_2 &= \text{<vars>} \\
v_3 &= \text{load_instruction}(v_0) \\
guard_eq(v_3, \text{INSTR_VAR_GET}) \\
list_append(v_1, v_2) \\
v_5 &= \text{add}(v_0, 1) \\
v_6 &= \text{load_instruction}(v_5) \\
guard_eq(v_6, \text{INSTR_INT}) \\
list_append(v_1, 0) \\
v_7 &= \text{add}(v_5, 1) \\
v_8 &= \text{load_instruction}(v_7) \\
guard_eq(v_8, \text{INSTR_LESS_THAN}) \\
v_9 &= \text{list_pop}(v_1) \\
v_{10} &= \text{list_pop}(v_1) \\
guard_type(v_9, \text{int}) \\
guard_type(v_{10}, \text{int}) \\
guard_not_less_than(v_9, v_{10}) \\
list_append(v_1, \text{False}) \\
v_{11} &= \text{add}(v_7, 1) \\
v_{12} &= \text{load_instruction}(v_{11}) \\
guard_eq(v_{12}, \text{INSTR_IF}) \\
v_{13} &= \text{list_pop}(v_1) \\
guard_false(v_{13}) \\
v_{14} &= \text{add}(v_{11}, 2) \\
v_{15} &= \text{load_instruction}(v_{14}) \\
guard_eq(v_{15}, \text{INSTR_VAR_GET}) \\
v_{16} &= \text{dict_get}(v_2, "x") \\
list_append(v_1, v_{16}) \\
v_{17} &= \text{add}(v_{14}, 1) \\
v_{18} &= \text{load_instruction}(v_{17}) \\
guard_eq(v_{18}, \text{INSTR_INT}) \\
list_append(v_1, 2) \\
v_{19} &= \text{add}(v_{17}, 1) \\
v_{20} &= \text{load_instruction}(v_{19}) \\
guard_eq(v_{20}, \text{INSTR_ADD}) \\
v_{21} &= \text{list_pop}(v_1) \\
v_{22} &= \text{list_pop}(v_1) \\
guard_type(v_{21}, \text{int}) \\
guard_type(v_{22}, \text{int}) \\
v_{23} &= \text{add}(v_{22}, v_{21}) \\
list_append(v_1, v_{23}) \\
v_{24} &= \text{add}(v_{19}, 1) \\
v_{25} &= \text{load_instruction}(v_{24}) \\
guard_eq(v_{25}, \text{INSTR_VAR_SET}) \\
v_{26} &= \text{list_pop}(v_1) \\
dict_set(v_2, "x", v_{26}) \\
v_{27} &= \text{add}(v_{24}, 1) \\
v_{28} &= \text{load_instruction}(v_{27}) \\
guard_eq(v_{28}, \text{INSTR_VAR_GET}) \\
v_{29} &= \text{dict_get}(v_2, "x") \\
v_{30} &= \text{add}(v_{27}, 1) \\
v_{31} &= \text{load_instruction}(v_{30}) \\
guard_eq(v_{31}, \text{INSTR_INT}) \\
list_append(v_1, 3) \\
v_{32} &= \text{add}(v_{30}, 1) \\
v_{33} &= \text{load_instruction}(v_{32}) \\
guard_eq(v_{33}, \text{INSTR_ADD}) \\
v_{34} &= \text{list_pop}(v_1) \\
v_{35} &= \text{list_pop}(v_1) \\
guard_type(v_{34}, \text{int}) \\
guard_type(v_{35}, \text{int}) \\
v_{36} &= \text{add}(v_{35}, v_{34}) \\
list_append(v_1, v_{36}) \\
v_{37} &= \text{add}(v_{32}, 1) \\
v_{38} &= \text{load_instruction}(v_{37}) \\
guard_eq(v_{38}, \text{INSTR_VAR_SET}) \\
v_{39} &= \text{list_pop}(v_1) \\
dict_set(v_2, "x", v_{39}) \\
v_{40} &= \text{add}(v_{37}, 1)
\end{align*}
\]
Removing constants (from `jit_merge_point`)

```
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)
list_append(v1, 3)
v34 = list_pop(v1)
v35 = list_pop(v1)
guard_type(v34, int)
guard_type(v35, int)
v36 = add(v35, v34)
list_append(v1, v36)
v39 = list_pop(v1)
dict_set(v2, "x", v39)
```
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)
List folded trace

\begin{align*}
  v1 &= \text{<stack>}
  v2 &= \text{<vars>}
  v4 &= \text{dict_get(v2, "x")}
  \text{guard_type(v4, int)}
  \text{guard_not_less_than(v4, 0)}
  v16 &= \text{dict_get(v2, "x")}
  \text{guard_type(v16, int)}
  v23 &= \text{add(v16, 2)}
  \text{dict_set(v2, "x", v23)}
  v29 &= \text{dict_get(v2, "x")}
  \text{guard_type(v29, int)}
  v36 &= \text{add(v29, 3)}
  \text{dict_set(v2, "x", v36)}
\end{align*}

Dict folded trace

\begin{align*}
  v1 &= \text{<stack>}
  v2 &= \text{<vars>}
  v4 &= \text{dict_get(v2, "x")}
  \text{guard_type(v4, int)}
  \text{guard_not_less_than(v4, 0)}
  v23 &= \text{add(v4, 2)}
  \text{guard_type(v23, int)}
  v36 &= \text{add(v23, 3)}
  \text{dict_set(v2, "x", v36)}
\end{align*}
Type folded trace

```python
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)
```
<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>
### Type folded trace

<table>
<thead>
<tr>
<th>v1 = stack</th>
<th>v2 = vars</th>
<th>v4 = dict_get(v2, &quot;x&quot;)</th>
<th>guard_type(v4, int)</th>
<th>guard_not_less_than(v4, 0)</th>
<th>v23 = add(v4, 2)</th>
<th>v36 = add(v23, 3)</th>
<th>dict_set(v2, &quot;x&quot;, v36)</th>
</tr>
</thead>
</table>

### Arithmetic folded trace

<table>
<thead>
<tr>
<th>v1 = stack</th>
<th>v2 = vars</th>
<th>v4 = dict_get(v2, &quot;x&quot;)</th>
<th>guard_type(v4, int)</th>
<th>guard_not_less_than(v4, 0)</th>
<th>v23 = add(v4, 5)</th>
<th>dict_set(v2, &quot;x&quot;, v23)</th>
</tr>
</thead>
</table>

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

PL X → Interpreters → Glue → Meta-tracing → PL Z

- PL X: Interpreters
- PL Y: Glue
- PL Z: Interpreter, Tracing JIT

RPython
Interpreter
Tracing JIT

Glue

47 / 59
http://soft-dev.org/
Runtime composition recap

- PyPy
- Hippy
- Interpreters
- Glue
- Meta-tracing
- PyHyp
- Interpreter
- Tracing JIT
## Composed Richards vs. other VMs

<table>
<thead>
<tr>
<th>Type</th>
<th>VM</th>
<th>Time (µs) ± Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono</td>
<td>CPython 2.7.7</td>
<td>9.475 ± 0.0127</td>
</tr>
<tr>
<td></td>
<td>HHVM 3.4.0</td>
<td>4.264 ± 0.0386</td>
</tr>
<tr>
<td></td>
<td>HippyVM</td>
<td>0.250 ± 0.0008</td>
</tr>
<tr>
<td></td>
<td>PyPy 2.4.0</td>
<td>0.178 ± 0.0006</td>
</tr>
<tr>
<td></td>
<td>Zend 5.5.13</td>
<td>9.070 ± 0.0361</td>
</tr>
</tbody>
</table>
## Composed Richards vs. other VMs

<table>
<thead>
<tr>
<th>Type</th>
<th>VM</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono</td>
<td>CPython 2.7.7</td>
<td>9.475 ± 0.0127</td>
</tr>
<tr>
<td></td>
<td>HHVM 3.4.0</td>
<td>4.264 ± 0.0386</td>
</tr>
<tr>
<td>Mono</td>
<td>HippyVM</td>
<td>0.250 ± 0.0008</td>
</tr>
<tr>
<td></td>
<td>PyPy 2.4.0</td>
<td>0.178 ± 0.0006</td>
</tr>
<tr>
<td></td>
<td>Zend 5.5.13</td>
<td>9.070 ± 0.0361</td>
</tr>
<tr>
<td>Composed</td>
<td>PyHyp</td>
<td>0.335 ± 0.0012</td>
</tr>
</tbody>
</table>
Datatype conversion

PHPRoot

PHPObject  PHPInt  PHPFunc
Datatype conversion

PHPRoot

PHPObject PHPInt PHPFunc

PyRoot

PyObject PyInt PyFunc

49 / 59

http://soft-dev.org/
Datatype conversion: primitive types

PHP

Python
Datatype conversion: primitive types

PHP

2 : PHPInt

Python
Datatype conversion: primitive types

PHP

2: PHPInt

Python

2: PyInt
Datatype conversion: user types

PHP

Python
Datatype conversion: user types

PHP

Python

\texttt{\texttt{o : PHPObject}}
Datatype conversion: user types

PyRoot

PyObject

PyInt

PyFunc
Datatype conversion: user types

PyRoot

PyObject  PyInt  PyFunc

PyPHPAdapter
Datatype conversion: user types

Diagram:

PyRoot

PyObject

PyInt

PyFunc

PyPHPAdapter

php_obj : PHPObjec
Datatype conversion: user types

PHP

```
o : PHPObject
```

Python
Datatype conversion: user types

PHP

\( o : \text{PHPObject} \)

Python

\( : \text{PyPHPAdapter} \)
Datatype conversion: user types

PHP

o : PHPObj ect

:PyPHPAdapter

php_obj

Python
Datatype conversion: user types

PHP

o : PHPObject

Python

:PyPHPAdapter

php_obj

Immutable field
A good composition needs to reduce *friction*.
A good composition needs to reduce *friction*. Some examples:

- Lexical scoping (or lack thereof) in PHP and Python (semantic friction)
A good composition needs to reduce friction. Some examples:

- Lexical scoping (or lack thereof) in PHP and Python (semantic friction)
- PHP datatypes are immutable except for references and objects; Python’s are largely mutable (semantic and performance friction)
A good composition needs to reduce friction. Some examples:

- Lexical scoping (or lack thereof) in PHP and Python (semantic friction)
- PHP datatypes are immutable except for references and objects; Python’s are largely mutable (semantic and performance friction)
- PHP has only dictionaries; Python has lists and dictionaries (semantic friction)
Unipyycation

PyPy
Hippy
Interpreters

Glue

Meta-tracing

PyHyp
Interpreter
Tracing JIT

53 / 59 HTTP://SOFT-DEV.ORg/
Unipycation

PyPy
Pyrolog
Interpreters
Glue

Meta-tracing

Interpreter
Tracing JIT

Unipycation

Software Development Team
http://soft-dev.org/
### 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</td>
<td>±0.007</td>
<td>0.257s</td>
<td>±0.002</td>
</tr>
<tr>
<td>L1A0R</td>
<td>2.924s</td>
<td>±0.284</td>
<td>7.352s</td>
<td>±0.048</td>
</tr>
<tr>
<td>L1A1R</td>
<td>4.184s</td>
<td>±0.038</td>
<td>18.890s</td>
<td>±0.111</td>
</tr>
<tr>
<td>NdL1A1R</td>
<td>7.531s</td>
<td>±0.080</td>
<td>18.643s</td>
<td>±0.197</td>
</tr>
<tr>
<td>TCons</td>
<td>264.415s</td>
<td>±2.250</td>
<td>48.819s</td>
<td>±0.252</td>
</tr>
<tr>
<td>Lists</td>
<td>9.374s</td>
<td>±0.059</td>
<td>25.148s</td>
<td>±0.221</td>
</tr>
<tr>
<td><strong>Unipycan</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallFunc</td>
<td>0.001s</td>
<td>±0.000</td>
<td>0.006s</td>
<td>±0.001</td>
</tr>
<tr>
<td>L1A0R</td>
<td>0.085s</td>
<td>±0.000</td>
<td>0.086s</td>
<td>±0.000</td>
</tr>
<tr>
<td>L1A1R</td>
<td>0.112s</td>
<td>±0.000</td>
<td>0.114s</td>
<td>±0.000</td>
</tr>
<tr>
<td>NdL1A1R</td>
<td>0.500s</td>
<td>±0.003</td>
<td>0.548s</td>
<td>±0.085</td>
</tr>
<tr>
<td>TCons</td>
<td>6.053s</td>
<td>±0.288</td>
<td>2.444s</td>
<td>±0.003</td>
</tr>
<tr>
<td>Lists</td>
<td>0.845s</td>
<td>±0.002</td>
<td>1.416s</td>
<td>±0.003</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</td>
<td>±0.003</td>
<td>3.050s</td>
<td>±0.053</td>
</tr>
<tr>
<td>L1A0R</td>
<td>1.078s</td>
<td>±0.009</td>
<td>206.590s</td>
<td>±3.846</td>
</tr>
<tr>
<td>L1A1R</td>
<td>2.145s</td>
<td>±0.232</td>
<td>293.311s</td>
<td>±5.691</td>
</tr>
<tr>
<td>NdL1A1R</td>
<td>7.939s</td>
<td>±0.457</td>
<td>1857.687s</td>
<td>±5.169</td>
</tr>
<tr>
<td>TCons</td>
<td>543.347s</td>
<td>±3.289</td>
<td>8014.477s</td>
<td>±17.710</td>
</tr>
<tr>
<td>Lists</td>
<td>5.661s</td>
<td>±0.046</td>
<td>6981.873s</td>
<td>±18.795</td>
</tr>
</tbody>
</table>
### Relative timing comparison

<table>
<thead>
<tr>
<th>VM</th>
<th>Benchmark</th>
<th>Python→Prolog Python</th>
<th>Python→Prolog Prolog</th>
<th>Python→Prolog Unipycation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPython-SWI</strong></td>
<td>SmallFunc</td>
<td>231.770× ±13.136</td>
<td>112.567× ±1.242</td>
<td>27821.079× ±2331.665</td>
</tr>
<tr>
<td></td>
<td>L1A0R</td>
<td>3.184× ±0.300</td>
<td>1.266× ±0.014</td>
<td>107.591× ±0.995</td>
</tr>
<tr>
<td></td>
<td>L1A1R</td>
<td>4.987× ±0.049</td>
<td>1.105× ±0.007</td>
<td>181.899× ±0.590</td>
</tr>
<tr>
<td></td>
<td>NdL1A1R</td>
<td>88.654× ±1.368</td>
<td>35.814× ±0.554</td>
<td>249.737× ±2.922</td>
</tr>
<tr>
<td></td>
<td>TCons</td>
<td>8.264× ±0.101</td>
<td>44.760× ±0.453</td>
<td>60.583× ±0.637</td>
</tr>
<tr>
<td></td>
<td>Lists</td>
<td>235.459× ±2.314</td>
<td>87.772× ±1.017</td>
<td>436.609× ±4.415</td>
</tr>
<tr>
<td><strong>Unipycation</strong></td>
<td>SmallFunc</td>
<td>1.295× ±0.105</td>
<td>0.182× ±0.054</td>
<td>1.000×</td>
</tr>
<tr>
<td></td>
<td>L1A0R</td>
<td>1.020× ±0.002</td>
<td>1.012× ±0.002</td>
<td>1.000×</td>
</tr>
<tr>
<td></td>
<td>L1A1R</td>
<td>1.025× ±0.002</td>
<td>1.002× ±0.003</td>
<td>1.000×</td>
</tr>
<tr>
<td></td>
<td>NdL1A1R</td>
<td>5.349× ±0.045</td>
<td>4.879× ±0.924</td>
<td>1.000×</td>
</tr>
<tr>
<td></td>
<td>TCons</td>
<td>5.959× ±0.282</td>
<td>14.756× ±0.092</td>
<td>1.000×</td>
</tr>
<tr>
<td></td>
<td>Lists</td>
<td>5.982× ±0.045</td>
<td>3.569× ±0.026</td>
<td>1.000×</td>
</tr>
<tr>
<td><strong>Jython-tuProlog</strong></td>
<td>SmallFunc</td>
<td>592.904× ±19.517</td>
<td>17.143× ±0.338</td>
<td>50354.204× ±4341.413</td>
</tr>
<tr>
<td></td>
<td>L1A0R</td>
<td>185.460× ±2.818</td>
<td>0.968× ±0.021</td>
<td>2310.844× ±28.093</td>
</tr>
<tr>
<td></td>
<td>L1A1R</td>
<td>137.427× ±14.537</td>
<td>1.005× ±0.028</td>
<td>2569.873× ±52.847</td>
</tr>
<tr>
<td></td>
<td>NdL1A1R</td>
<td>250.776× ±14.666</td>
<td>1.072× ±0.009</td>
<td>744.699× ±6.726</td>
</tr>
<tr>
<td></td>
<td>TCons</td>
<td>15.096× ±0.106</td>
<td>1.023× ±0.004</td>
<td>227.409× ±1.592</td>
</tr>
<tr>
<td></td>
<td>Lists</td>
<td>985.149× ±8.674</td>
<td>0.799× ±0.003</td>
<td>1103.206× ±8.338</td>
</tr>
</tbody>
</table>

[56/59] http://soft-dev.org/
What can we use this for?
First-class languages
What can we use this for?

First-class languages

Language migration
Thanks to our funders

- EPSRC: COOLER and Lecture.
- Oracle: various.
Thanks for listening!

http://soft-dev.org/