Partial Evaluation Based CPS Transformation: An Implementation Case Study

Rajesh Jayaprakash

Tata Consultancy Services

Chennai, India

Overview

- Preliminaries
  - Partial evaluation
  - CPS
- Optimization of Naïve CPS
  - Transformation example
- Compiler Pipeline
- PECPS Implementation
- Conclusion
- Q&A

Partial Evaluation

- Partition a program into static and dynamic parts
- Execute the static part at compile time so that there is less computation to do at run time
- A simplistic, contrived example:

```
int main(int argc,
         char **argv)
{
    long i, a, b, c;
    a = 48594;
    b = 93763;
    c = a + b;
    scanf(“%ld\n”, &i);
    printf(“%ld\n”, i + c);
    return 0;
}
```

```
int main(int argc,
         char **argv)
{
    long i;
    scanf(“%ld\n”, &i);
    printf(“%ld\n”, i + 142357)
    return 0;
}
```

Continuation Passing Style

- Every function is passed one more argument, viz., the rest of the computation, embodied by a continuation function
- The function performs its computation, and invokes the continuation with the result of this computation
- Example (from Paul Graham's "On Lisp"):

```
(/ (- x 1) 2)
```

When (- x 1) is evaluated, the continuation is the function

```
(lambda (v) (/ v 2))
```

Continuation Passing Style (cont.)

- CPS makes all control flow explicit (e.g., order of evaluation of function arguments)
- Easier to introduce non-local control transfers like exceptions to the language

- The output of a CPS transformation is a function that performs the computation of the original expression, and invokes the continuation (passed as argument to the function) on the computation result

Continuation Passing Style (cont.)

```
(if t 1 2)
```

⇓

```
(lambda (k1)
  ((lambda (i1) (i1 t))
   (lambda (test)
     (if test
           ((lambda (i2) (i2 2)) k1)
           ((lambda (i3) (i3 3)) k1)))))
```

Optimizing a Naïve CPS Transform

```
(+ x 1)

(lambda (g8216)
  ((lambda (g8218)
     (g8218 +))
   (lambda (g8217)
     ((lambda (g8220)
        (g8220 x))
      (lambda (g8219)
        ((lambda (g8222)
           (g8222 1))
         (lambda (g8221)
           (g8217 g8219 g8221 g8216)))))))
```

Optimizing a Naïve CPS Transform

```
;after beta-reduction:

(lambda (g8216)
  (let ((g2818 (lambda (g8217)
                ((lambda (g8220)
                   (g8220 x))
                 (lambda (g8219)
                   ((lambda (g8222)
                      (g8222 1))
                    (lambda (g8221)
                      (g8217 g8219 g8221 g8216))))))))
    (g8218 +)))
```

Beta-reduction: (λV.M) N => M[V := N]

Optimizing a Naïve CPS Transform

```
;after one more beta-reduction:

(lambda (g8216)
  (let ((g2818 (lambda (g8217)
                 (let ((g8220 (lambda (g8219)
                                ((lambda (g8222)
                                   (g8222 1))
                                 (lambda (g8221)
                                   (g8217 g8219 g8221 g8216))))))
                   (g8220 x)))))
    (g8218 +)))
```

Beta-reduction: $(\lambda V.M)\ N \Rightarrow M[V := N]$

Optimizing a Naïve CPS Transform

```
;after one more beta reduction:

(lambda (g8216)
  (let ((g2818 (lambda (g8217)
                 (let ((g8220 (lambda (g8219)
                                (let ((g8222 (lambda (g8221)
                                               (g8217 g8219 g8221 g8216))))
                                  (g8222 1)))))
                   (g8220 x)))))
    (g8218 +)))
```

Beta-reduction: $(\lambda V.M)\ N \Rightarrow M[V := N]$

Optimizing a Naïve CPS Transform

```
;after inlining the innermost let (constant propagation followed by beta-reduction):

(lambda (g8216)
  (let ((g2818 (lambda (g8217)
                 (let ((g8220 (lambda (g8219)
                                (g8217 g8219 1 g8216))))
                   (g8220 x)))))
    (g8218 +)))
```

Optimizing a Naïve CPS Transform

```
;after inlining the innermost let (constant propagation followed by beta-reduction):

(lambda (g8216)
  (let ((g2818 (lambda (g8217)
                 (g8217 x 1 g8216))))
    (g8218 +)))
```

Optimizing a Naïve CPS Transform

```
;after inlining the remaining let (constant propagation followed by beta-reduction)

(lambda (g8216)
  (+ x 1 g8216))
```

What is pLisp?

"The only thing left to do is to add whatever is needed
to open a lot of little windows everywhere."
- Christian Queinnec, *Lisp in Small Pieces*

- A Lisp dialect based on Common Lisp
- An integrated development environment
- Platforms
  - Linux, Windows, OS X
- Open source; GPL 3.0 license
- Built using OSS components
  - GTK+, GTKSourceView, libffi, Boehm GC, LLVM, Flex, Bison

https://github.com/shikantaza/pLisp

Motivation for pLisp

- To serve as a friendly environment for beginners to learn Lisp
  - Graduate to Common Lisp and its implementations/environments
- Inspired by Smalltalk environments
  - Workspace/Transcript/System Browser
  - Ability to edit code in all contexts
  - Image based development
    - GUI state part of image

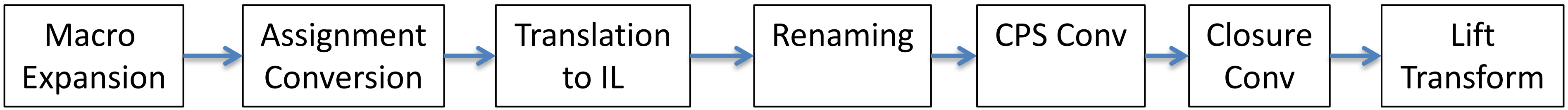
pLisp Features

- Graphical IDE with context-sensitive help, syntax coloring, autocomplete, and auto-indentation
- Native compiler
- User-friendly debugging/tracing
- Image-based development
- Continuations
- Exception handling
- Foreign function interface
- Package/Namespace system

```
(defun fact (n)
  (if (eq n 0)
      1
    (* n (fact (- n 1)))))

(fact 5)
```

pLisp Compiler Pipeline

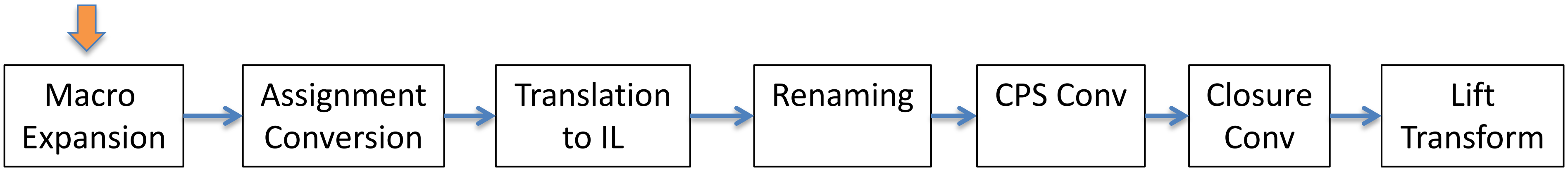


pLisp Compiler Pipeline

Macro Expansion → Assignment Conversion → Translation to IL → Renaming → CPS Conv → Closure Conv → Lift Transform

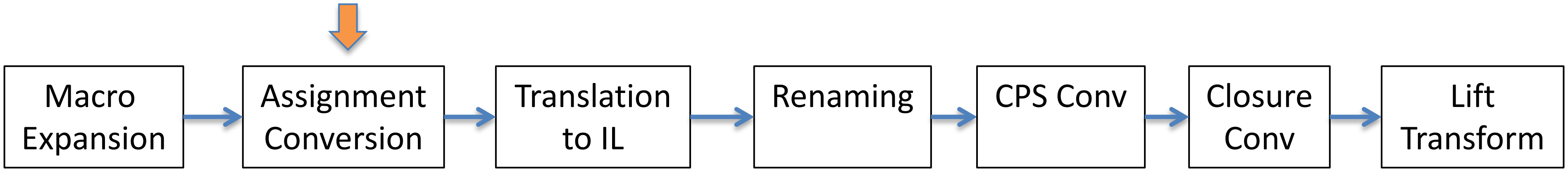
```
(print "Hello World!")
```

pLisp Compiler Pipeline




```
(print "Hello World!")
```

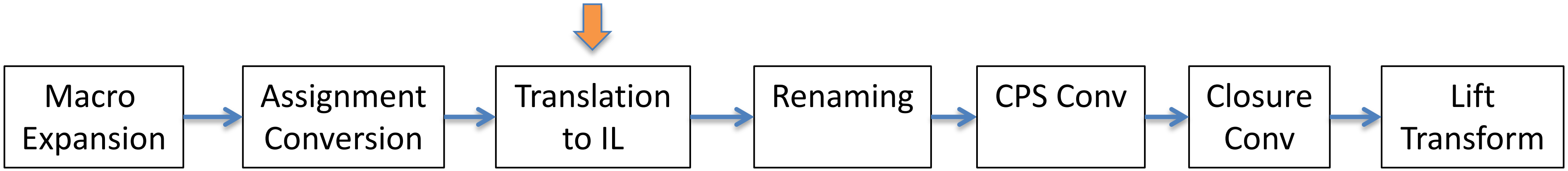
pLisp Compiler Pipeline



Conversion of mutable variables into mutable cells

```
((prim-car print) "Hello World!")
```

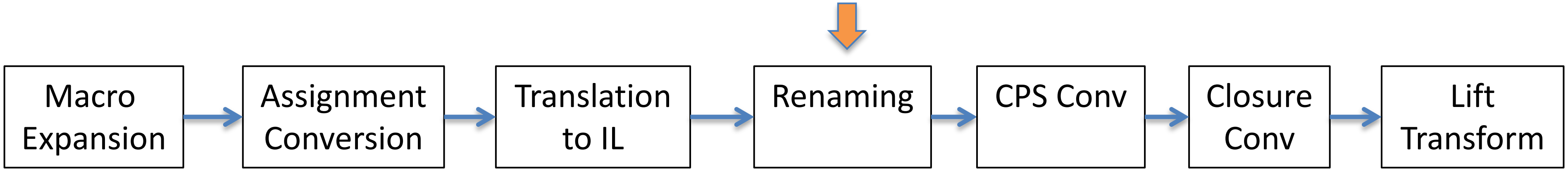
pLisp Compiler Pipeline



Conversion to simple intermediate language without recursive forms

```
((prim-car print) "Hello World!")
```

pLisp Compiler Pipeline



To ensure uniqueness of variable names

```
((prim-car print) "Hello World!")
```

pLisp Compiler Pipeline

Macro Expansion → Assignment Conversion → Translation to IL → Renaming → **CPS Conv** → Closure Conv → Lift Transform

Conversion of code to continuation passing style

```
(lambda (#:g00008073)
  (save-continuation #:g00008073)
  (let ((#:g00008074 (prim-car print)))
    (let ((#:g00008075 (lambda (#:g00008076)
                         (#:g00008073 #:g00008076)))
                         (#:g00008074 "Hello World!" #:g00008075))))))
```

pLisp Compiler Pipeline

Macro Expansion → Assignment Conversion → Translation to IL → Renaming → CPS Conv → **Closure Conv** → Lift Transform

Transformation of all functions to closures

```
(lambda (#:g00008077 #:g00008073)
  (save-continuation #:g00008073)
  (let2 ((print (nth1 1 #:g00008077)))
    (let ((#:g00008074 (prim-car print)))
      (let2 ((#:g00008081 (lambda (#:g00008078 #:g00008076)
                            (let2 ((#:g00008073 (nth1 1 #:g00008078)))
                              (let2 ((#:g00008079 #:g00008073)
                                     (#:g00008080 (extract-native-fn #:g00008079)))
                                (#:g00008080 #:g00008079 #:g00008076)))))
             (#:g00008075 (create-fn-closure 1 #:g00008081 #:g00008073)))
        (let2 ((#:g00008082 #:g00008074)
               (#:g00008083 (extract-native-fn #:g00008082)))
          (#:g00008083 #:g00008082 "Hello World!" #:g00008075))))))
```

pLisp Compiler Pipeline

Macro Expansion → Assignment Conversion → Translation to IL → Renaming → CPS Conv → Closure Conv → Lift Transform

Eliminate function nesting and lifting all functions to the top level

```
(#:g00008084 (lambda (#:g00008077 #:g00008073)
               (save-continuation #:g00008073)
               (let2 ((print (nth1 1 #:g00008077)))
                 (let ((#:g00008074 (prim-car print)))
                   (let2 ((#:g00008081 #:g00008085)
                          (#:g00008075 (create-fn-closure 1 #:g00008081 #:g00008073)))
                     (let2 ((#:g00008082 #:g00008074)
                            (#:g00008083 (extract-native-fn #:g00008082)))
                       (#:g00008083 #:g00008082 "Hello World!" #:g00008075)))))))
(#:g00008085 (lambda (#:g00008078 #:g00008076)
               (let2 ((#:g00008073 (nth1 1 #:g00008078)))
                 (let2 ((#:g00008079 #:g00008073)
                        (#:g00008080 (extract-native-fn #:g00008079)))
                   (#:g00008080 #:g00008079 #:g00008076)))))
```

Regular Vs PE CPS Transformation

$$\begin{aligned}
&\mathcal{SCPS}_{exp}[\![\texttt{(if } E_{test} \;\; E_{then} \;\; E_{else}\texttt{)}]\!] \\
&\quad = \texttt{(abs (}I_k\texttt{)} \;\; ; I_k \text{ fresh} \\
&\qquad\qquad \texttt{(app (}\mathcal{SCPS}_{exp}[\![E_{test}]\!]\texttt{)} \\
&\qquad\qquad\qquad \texttt{(abs (}I_{test}\texttt{)} \;\; ; I_{test} \text{ fresh} \\
&\qquad\qquad\qquad\quad \texttt{(if } I_{test} \\
&\qquad\qquad\qquad\qquad \texttt{(app (}\mathcal{SCPS}_{exp}[\![E_{then}]\!]\texttt{)} \;\; I_k\texttt{)} \\
&\qquad\qquad\qquad\qquad \texttt{(app (}\mathcal{SCPS}_{exp}[\![E_{else}]\!]\texttt{)} \;\; I_k\texttt{)))))}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{MCPS}_{exp}[\![\texttt{(if } E_{test} \;\; E_{then} \;\; E_{else}\texttt{)}]\!] \\
&\quad = (\lambda m \,.\;\; (\mathcal{MCPS}_{exp}[\![E_{test}]\!] \\
&\qquad\qquad (\lambda V_{test} \,.\;\; \texttt{(let ((}I_{kif} \;\; (\text{mc}\rightarrow\text{exp } m)\texttt{))} \;\; ; I_{kif} \text{ fresh} \\
&\qquad\qquad\qquad \texttt{(if } V_{test} \\
&\qquad\qquad\qquad\quad (\mathcal{MCPS}_{exp}[\![E_{then}]\!] \;\; (\text{id}\rightarrow\text{mc } I_{kif})) \\
&\qquad\qquad\qquad\quad (\mathcal{MCPS}_{exp}[\![E_{else}]\!] \;\; (\text{id}\rightarrow\text{mc } I_{kif}))\texttt{))}))))
\end{aligned}$$

Design Concepts in Programming Languages (Turbak et al., 2008)

Regular Vs PE CPS Transformation (cont.)

Regular CPS Transform	PE CPS Transform
CPS-transformed code is an abstraction in the object language	CPS-transformed code is an abstraction in the metalanguage
The abstraction is applied to a continuation in the object language (‘l_k’ in the previous slide)	The abstraction is applied to a continuation in the metalanguage (‘m’ in previous slide)
Made efficient by beta-reductions and inlining in subsequent passes	Application of metalanguage abstraction already generates efficient code

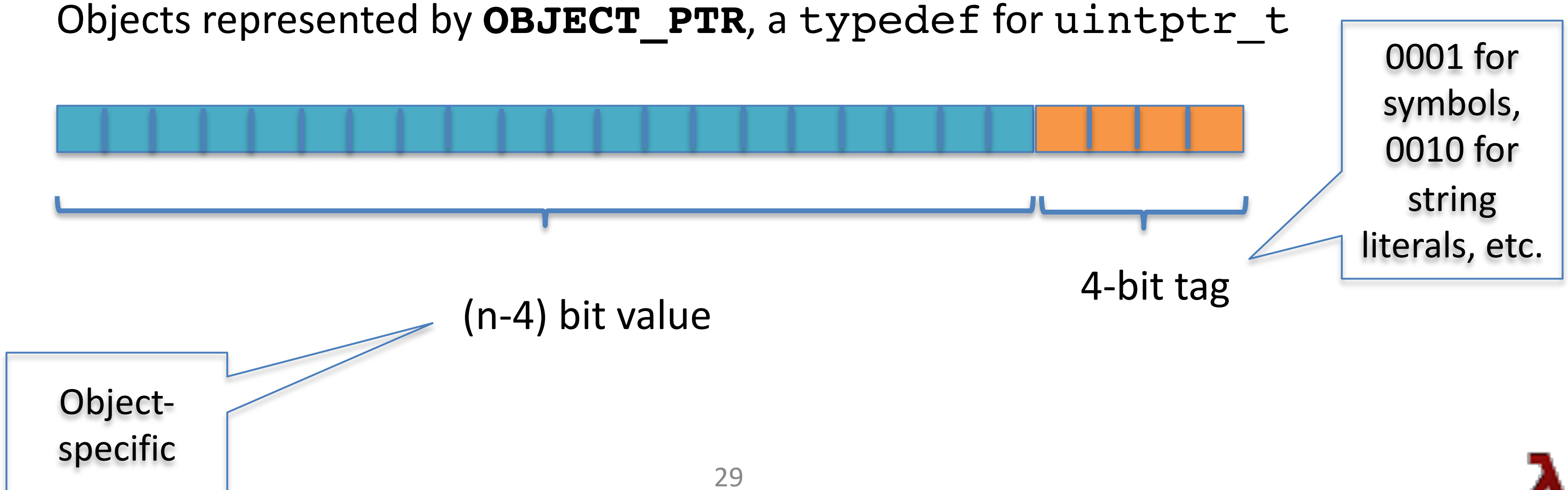
Implementing the PE CPS Pass in pLisp

- pLisp is written in C
  - Imperative
  - FP abstractions (used in the function MCPS) not available
  - Need to mimic OO features to unify the handling of the different language constructs
    - Dispatching to the correct transformation function for each language construct
- Handling transforms involving variable number of sub-expressions (e.g., let, applications, and primops)

pLisp Objects and Representation

- Integers
- Floating point numbers
- Characters
- Strings
- Symbols
- Arrays
- CONS cells
- Closures
- Macros

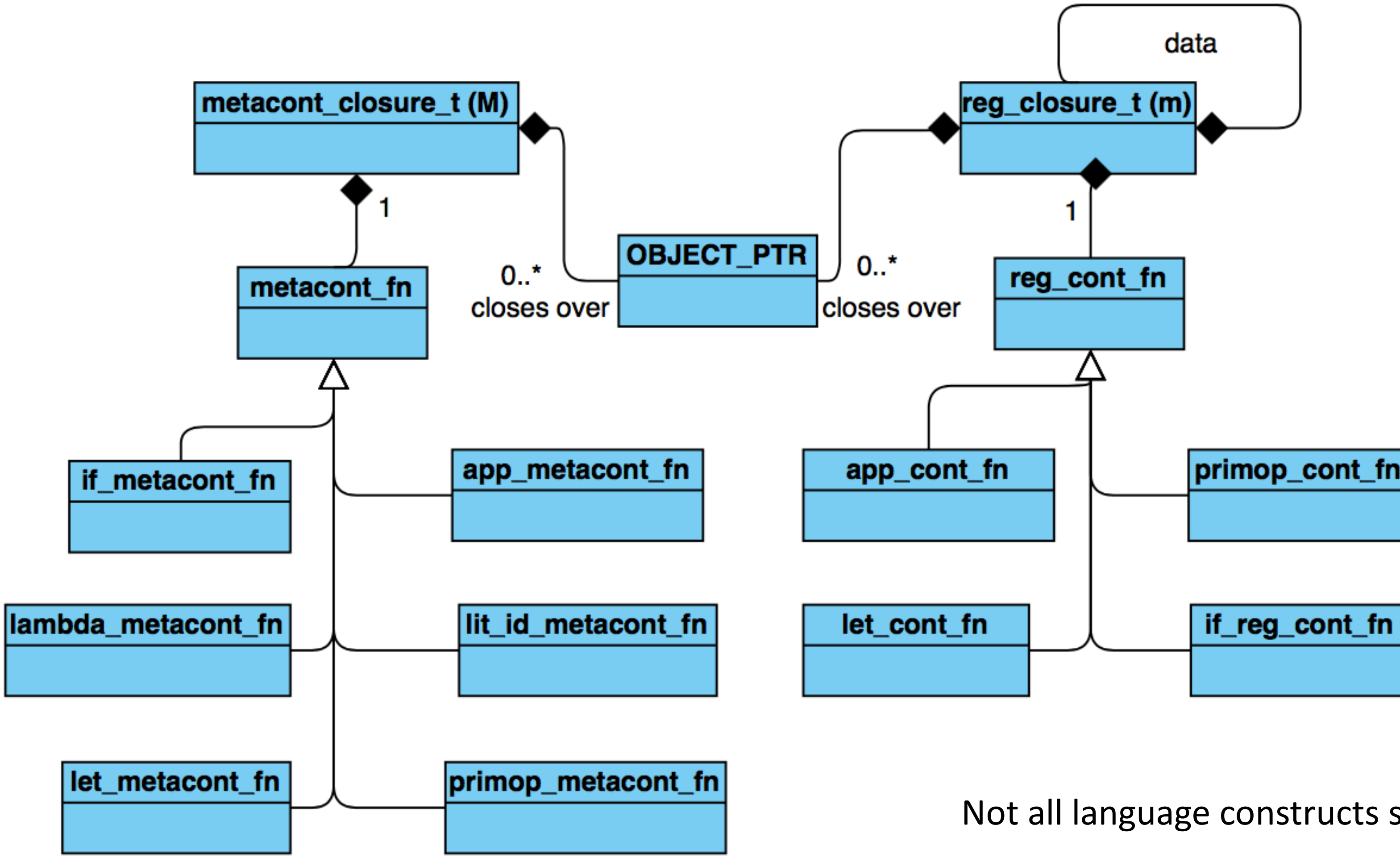
Objects represented by **`OBJECT_PTR`**, a `typedef` for `uintptr_t`



pLisp Objects and Representation (cont.)

Object Type	Object-Specific Value
Integer	Address of allocated integer
Float	Address of allocated floating point number
Character	Numeric representation of ASCII value (e.g. 65 for 'A')
String	Mutable strings are arrays (see below); for immutable strings, value is an index into a global strings array
Symbol	Value is split into a) an index into a global packages array and b) an index into the strings array of the chosen packages array element
Array	Address of segment of size n+1, first element storing the integer object denoting the array size n
CONS cell	Address of first of two contiguous memory locations
Closure	Address of linked list of CONS cells containing the native function object and the closed-over objects
Macro	Similar to above
Native function	Address of native function pointer

Metalanguage Interpreter – Object Model



Not all language constructs shown

Metalanguage Interpreter – Data Structures

```c
//forward declarations
struct reg_closure;
struct metacont_closure;

typedef OBJECT_PTR (*reg_cont_fn)(struct reg_closure *, OBJECT_PTR);

typedef struct reg_closure
{
  reg_cont_fn fn;
  unsigned int nof_closed_vals;
  OBJECT_PTR *closed_vals;
  void *data;
} reg_closure_t;

typedef OBJECT_PTR (*metacont_fn)(struct metacont_closure *, struct reg_closure *);

typedef struct metacont_closure
{
  metacont_fn mfn;
  unsigned int nof_closed_vals;
  OBJECT_PTR *closed_vals;
} metacont_closure_t;
```

PECPS Transform of 'if'

```
if(car_exp == IF)
 {
   metacont_closure_t *mcls = (metacont_closure_t *)
                               GC_MALLOC(sizeof(metacont_closure_t));

   mcls->mfn                = if_metacont_fn;

   mcls->nof_closed_vals = 3;
   mcls->closed_vals        = (OBJECT_PTR *)
                              GC_MALLOC(mcls->nof_closed_vals *
                                        sizeof(OBJECT_PTR));

   mcls->closed_vals[0]  = second(exp);
   mcls->closed_vals[1]  = third(exp);
   mcls->closed_vals[2]  = fourth(exp);

   return mcls;
 }
```

$$\mathcal{MCPS}_{exp}[\![\texttt{(if}\ E_{test}\ \ E_{then}\ \ E_{else}\texttt{)}]\!]$$
$$= (\lambda m\ .\ \ (\mathcal{MCPS}_{exp}[\![E_{test}]\!]$$
$$(\lambda V_{test}\ .\ \ \texttt{(let ((}I_{kif}\ \ (\text{mc}\rightarrow\text{exp}\ \ m)\texttt{))}\ \ ;\ I_{kif}\ \text{fresh}$$
$$\texttt{(if}\ \ V_{test}$$
$$(\mathcal{MCPS}_{exp}[\![E_{then}]\!]\ \ (\text{id}\rightarrow\text{mc}\ \ I_{kif}))$$
$$(\mathcal{MCPS}_{exp}[\![E_{else}]\!]\ \ (\text{id}\rightarrow\text{mc}\ \ I_{kif}))\texttt{)))}))$$

PECPS Transform of 'if' (cont.)

```c
OBJECT_PTR if_metacont_fn(metacont_closure_t *mcls, reg_closure_t *cls1)
{
  OBJECT_PTR test_exp = mcls->closed_vals[0];
  OBJECT_PTR then_exp = mcls->closed_vals[1];
  OBJECT_PTR else_exp = mcls->closed_vals[2];

  metacont_closure_t *test_mcls = mcps(test_exp);

  reg_closure_t *cls = (reg_closure_t *)GC_MALLOC(sizeof(reg_closure_t));

  cls->fn                = if_reg_cont_fn;
  cls->nof_closed_vals   = 2;
  cls->closed_vals       = (OBJECT_PTR *)GC_MALLOC(cls->nof_closed_vals * sizeof(OBJECT_PTR));

  cls->closed_vals[0]    = then_exp;
  cls->closed_vals[1]    = else_exp;

  cls->data = cls1;

  return test_mcls->mfn(test_mcls, cls);
}
```

$$
\begin{aligned}
&\mathcal{MCPS}_{exp}[\![(\texttt{if}\ E_{test}\ E_{then}\ E_{else})]\!] \\
&\quad = (\lambda m\ .\ (\mathcal{MCPS}_{exp}[\![E_{test}]\!] \\
&\qquad\quad (\lambda V_{test}\ .\ (\texttt{let}\ ((I_{kif}\ (\mathrm{mc}\rightarrow\mathrm{exp}\ m)))\ \ ;\ I_{kif}\ \mathit{fresh} \\
&\qquad\qquad\qquad (\texttt{if}\ V_{test} \\
&\qquad\qquad\qquad\quad (\mathcal{MCPS}_{exp}[\![E_{then}]\!]\ (\mathrm{id}\rightarrow\mathrm{mc}\ I_{kif})) \\
&\qquad\qquad\qquad\quad (\mathcal{MCPS}_{exp}[\![E_{else}]\!]\ (\mathrm{id}\rightarrow\mathrm{mc}\ I_{kif})))))))
\end{aligned}
$$

PECPS Transform of 'if' (cont.)

$$
\begin{aligned}
&\mathcal{MCPS}_{exp}[\![\texttt{(if } E_{test}\ E_{then}\ E_{else}\texttt{)}]\!] \\
&\quad = (\lambda m\ .\ (\mathcal{MCPS}_{exp}[\![E_{test}]\!] \\
&\qquad\qquad (\lambda V_{test}\ .\ \texttt{(let ((}I_{kif}\ (\text{mc}\rightarrow\text{exp}\ m)\texttt{))}\ ;\ I_{kif}\ \text{fresh} \\
&\qquad\qquad\qquad \texttt{(if }V_{test} \\
&\qquad\qquad\qquad\quad (\mathcal{MCPS}_{exp}[\![E_{then}]\!]\ (\text{id}\rightarrow\text{mc}\ I_{kif})) \\
&\qquad\qquad\qquad\quad (\mathcal{MCPS}_{exp}[\![E_{else}]\!]\ (\text{id}\rightarrow\text{mc}\ I_{kif}))\texttt{)))})))
\end{aligned}
$$

PECPS Transform of ’if’ (cont.)

```
OBJECT_PTR if_reg_cont_fn(reg_closure_t *cls, OBJECT_PTR test_val)
{
  OBJECT_PTR i_kif = gensym();

  reg_closure_t *cls1 = (reg_closure_t *)cls->data;

  OBJECT_PTR then_exp = cls->closed_vals[0];
  OBJECT_PTR else_exp = cls->closed_vals[1];

  metacont_closure_t *then_mcls = mcps(then_exp);
  metacont_closure_t *else_mcls = mcps(else_exp);

  reg_closure_t *kif_cls = id_to_mc(i_kif);

  return list(3,
              LET,
              list(1, list(2, i_kif, mc_to_exp(cls1))),
              list(4,
                   IF,
                   test_val,
                   then_mcls->mfn(then_mcls, kif_cls),
                   else_mcls->mfn(else_mcls, kif_cls)));
}
```

Handling LET (and similar clauses)

```
reg_closure_t *create_reg_let_closure(OBJECT_PTR    bindings,
                                      OBJECT_PTR    full_bindings,
                                      OBJECT_PTR    body,
                                      unsigned int  nof_vals,
                                      OBJECT_PTR    *vals,
                                      reg_closure_t *cls)
{
  reg_closure_t *let_closure = (reg_closure_t *)GC_MALLOC(sizeof(reg_closure_t));

  if(cons_length(bindings) == 0) //last binding
    let_closure->fn = let_cont_fn_non_recur;
  else
    let_closure->fn = let_cont_fn_recur;

  let_closure->nof_closed_vals = nof_vals + 3;
  let_closure->closed_vals     = (OBJECT_PTR *)GC_MALLOC(let_closure->nof_closed_vals
                                                          * sizeof(OBJECT_PTR));

  let_closure->closed_vals[0]  = bindings;
  let_closure->closed_vals[1]  = full_bindings;
  let_closure->closed_vals[2]  = body;

  int i;
  for(i=3; i<let_closure->nof_closed_vals; i++)
    let_closure->closed_vals[i] = vals[i-3];

  let_closure->data = cls;

  return let_closure;
}
```

$$
\begin{aligned}
&\mathcal{MCPS}[\![(\texttt{let}\ ((I_i\ E_i)_{i=1}^{n})\ E_{body})]\!] \\
&\quad = (\lambda m\,.\ (\mathcal{MCPS}[\![E_1]\!] \\
&\qquad\quad (\lambda V_1\,. \\
&\qquad\qquad \ldots \\
&\qquad\qquad\quad (\mathcal{MCPS}[\![E_n]\!] \\
&\qquad\qquad\qquad (\lambda V_n\,.\ (\texttt{let*}\ ((I_i\ V_i)_{i=1}^{n}) \\
&\qquad\qquad\qquad\qquad (\mathcal{MCPS}[\![E_{body}]\!]\ m))))\ \ldots\ )))
\end{aligned}
$$

Conclusion and Future Work

- PECPS significantly faster than naïve CPS with optimizations
- Metalanguage interpreter is in C
  - Implementing the transform in imperative style takes work (simulating closures, etc.)
  - OO capabilities would have helped
- Explore a declarative style of generating the transforms
  - S-expression templates with context 'holes'

Thank you!

Rajesh Jayaprakash
rajesh.jayaprakash@tcs.com