Partial Evaluation Based CPS Transformation: An Implementation Case Study

Rajesh Jayaprakash
TCS Research, India
rajesh.jayaprakash@tcs.com

ABSTRACT

We demonstrate the implementation of a partial evaluation based CPS transformation in the context of pLisp, a Lisp dialect and IDE for beginners. The CPS transformation employs a modular technique that unifies the treatment of the language constructs; we illustrate the transformation by explicating the conversion process for a single construct (viz., `if`). To the best of our knowledge, this framework is also novel in that the partial evaluation and CPS transformation techniques are implemented in the implementation language of the system itself (i.e., C), as opposed to bootstrapping from an existing Lisp dialect.

CCS CONCEPTS

• **Software and its engineering → Compilers**;

KEYWORDS

Lisp, partial evaluation, CPS transformation

ACM Reference Format:
Rajesh Jayaprakash. 2020. Partial Evaluation Based CPS Transformation: An Implementation Case Study. In *Proceedings of the 13th European Lisp Symposium (ELS'20).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.3740941

1 INTRODUCTION

Partial evaluation [12] is a well-established technique for optimizing programs. A program is partitioned into a static and a dynamic part, with the static part comprising data and values known beforehand, and the dynamic part comprising data and values not known at compile-time. The compilation or translation process 'executes' the static part so that only a residual program is left over for execution later, thereby resulting in a smaller/faster program.

Continuation Passing Style (CPS) [14] is a style of programming in which every function call is augmented with an additional argument known as a continuation; this continuation embodies the rest of the computation, and the function is expected to perform its computation and then invoke the continuation with the result of the computation. There are a number of advantages to programming in CPS, a few of them being a) simplifying the effort needed at the compiler back-end b) making explicit the semantics of the computation (e.g., order/sequencing of execution primitives) and c) enabling the easier implementation of advanced control structures like non-local control transfers. Transformation of a program to CPS is a step in a typical compiler pipeline that includes steps like assignment conversion, renaming, closure conversion, and lift transformation.

A naive CPS transformation [8] results in quite inefficient code, and these inefficiencies are removed using techniques like inlining. Another option for removing these inefficiencies is to leverage partial evaluation based techniques [6]. The 'static' program fragments that would have been generated by the naive CPS transformation are recognized as such and are executed by the so-called metalanguage interpreter [15] during the transformation process itself, thereby preventing the inefficient code from being generated in the first place.

Figure 1 illustrates the CPS transformation of the expression `(+ x 1)` using both a naive approach (Figure 1a) and a partial evaluation based approach (Figure 1d) [1]. The output of the naive transformation is typically optimized by repeated β-reductions (Figure 1b) and *inlining* (constant propagation of a lambda form followed by a β-reduction; Figure 1c). The quasi-syntactic (and semantic) equivalence of the optimized naive CPS transformation and the partial evaluation based CPS transformation is to be noted, although this equivalence is achieved by different routes; in the case of the partial evaluation approach, the inlining optimization is effected by execution of the relevant code fragment (the let forms in Figure 1b) by the metalanguage interpreter (explained in section 3).

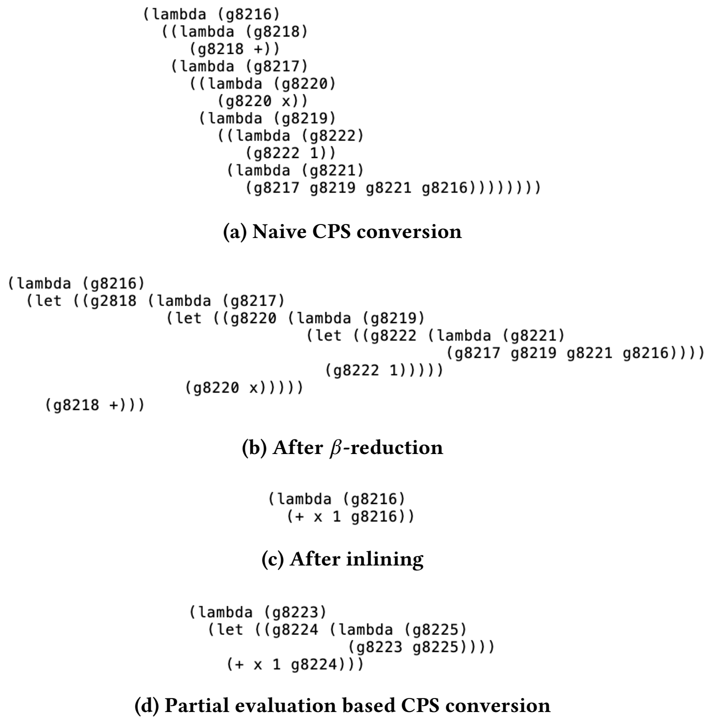
pLisp [11] is a Lisp-1 dialect and an integrated development environment modelled on Smalltalk that targets beginners, with the following features:

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

The native compiler in pLisp implements a partial evaluation based CPS transformation step through a modular and flexible framework, embodying an elegant construct-dependent technique for the creation of the metalanguage closures, which is detailed in this paper.

[1] This code was generated by the pLisp compiler; the names of the binding variables in the abstractions have been shortened to improve readability

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'20, April 27–28 2020, Zurich, Switzerland
© 2020 Copyright held by the owner/author(s).
https://doi.org/10.5281/zenodo.3740941



(a) Naive CPS conversion

(b) After β-reduction

(c) After inlining

(d) Partial evaluation based CPS conversion

Figure 1: Naive and partial evaluation based CPS conversion of '(+ x 1)'

2 COMPILER PIPELINE

The pLisp compiler transforms the Lisp source code to CPS and emits C code, which is then passed to LLVM to produce native code. The compiler does the transformation in the following passes [15]:

- Desugaring/Macro expansion
- Assignment conversion
- Translation
- Renaming
- CPS conversion
- Closure conversion
- Lift transformation
- Conversion to C

These compiler passes produce progressively simpler Lisp dialects, culminating in a version with semantics close enough to C. Figure 2 illustrates this transformation. Since LLVM is used to convert the C code to native code, the pipeline does not include passes like register allocation/spilling.

Desugaring/Macro expansion: This pass performs macro expansion, resulting in a dialect called `plisp_k` ('k' stands for 'kernel') that is shorn of all macro invocations. The backquote construct used in macro-expansion is itself implemented as a macro, so the order of definitions of backquote and its supporting functions is critical. Accordingly, the source file listing the definitions of the core library objects contains the macro-expansion-related infrastructure before the other definitions.

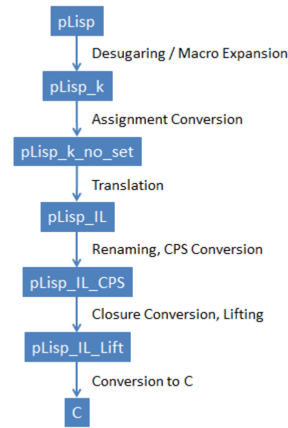


Figure 2: pLisp compiler transformations

Assignment conversion: Assignment conversion replaces all assigned variables with mutable cells, thereby making variable bindings immutable (i.e., the contents of the cell may change over time, but the binding of a cell to a variable will not). In the interests of simplicity and to avoid the introduction of one more object type, mutable cells in pLisp are simulated by CONS objects whose CDR is NIL. Assignment conversion produces code in a dialect called `plisp_k_no_set`, which differs from plisp_k only with respect to the absence of the `set` construct (replaced by the `setcar` primitive).

Translation: This pass produces an intermediate language dialect called `pLisp_IL` (Figure 3) and differs from the previous dialects in that a) it does not have a recursion construct (`letrec`) and b) a new multibinding construct called `let*` is introduced (note: this `let*` is distinct from the similarly-named core library form). This pass also performs syntactic simplifications like removal of empty `let`s, conversion of applications of `lambda` expressions to `let`s, η-reductions and copy propagation.

```
exp  ::=  literal | var
          | (if exp exp exp?)
          | (primop exp+)
          | (lambda (var*) exp)
          | (error exp)
          | (call/cc exp)
          | (exp exp*)
          | (let ((var exp)*) exp)
          | (let* ((var exp)*) exp)
```

Figure 3: pLisp_IL grammar

Renaming: The renaming pass ensures that no two logically distinct variables in the code have the same name, so that variable capture is avoided. Introduction of fresh variable names utilizes the same infrastructure that implements the `gensym` feature in the pLisp core.

CPS Conversion: The CPS conversion pass converts the code to CPS style, and produces the dialect `plisp_IL_CPS` (Figure 4) characterized by restrictions on `let` forms: they can now only have

one binding, and the expressions that can be bound to the `let` identifiers (denoted by `le` in the figure) are restricted to literals, lambda expressions, and primitive operations. The CPS conversion pass also injects code to save the generated continuation object: this is useful for implementing the break/resume functionality for the debugger.

```
exp  ::=  (var val*)
          | (if val exp exp)
          | (let ((var le)) exp)
          | (error exp)
          | (call/cc exp)
val  ::=  literal | var
le   ::=  literal
          | (lambda (var*) exp)
          | (primop val+)
```

Figure 4: pLisp_IL _CPS grammar

Closure conversion: In this pass, functions are converted into closures, so that the free variables referred in the lambda expression body are fetched from the environment that is stored along with the code. The lambda expressions are thus implicitly passed the code/environment pair as their first parameter.

Lift transformation: The lift transformation pass converts all procedures to the top level and thereby linearizes the code. Please note that lift transformation is predicated on the procedures having undergone a closure conversion. This pass results in the `plisp_IL_Lift` dialect, where the code linearization is manifested as further restrictions on `let` bindings (only literals and primitive operations are permitted as `let`able expressions).

Conversion to C: The C conversion pass, in addition to handling the Lisp-to-C syntax translation, also performs additional tasks like decorating the variable names so that they do not violate the C syntax rules for variable names and keywords.

Expressions entered at the top level (i.e., the Workspace in pLisp) are compiled into anonymous closures, and are supplied the identity function as their continuation.

3 PARTIAL EVALUATION BASED CPS TRANSFORMATION

3.1 pLisp Object Representation

In this subsection we briefly describe the pLisp object model, inasmuch as is required to set the context for this work. The following object types are supported by pLisp [11]:

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

Objects are internally represented by `OBJECT_PTR`, a `typedef` for `uintptr_t`, the C language data type used for storing pointer values. The four least significant bits of the value are used to tag the object type (e.g., `0011` for character objects, `0110` for CONS cells, and so on), while the remaining *(n-4)* bits (where *n* is the total number of bits) of the value take on different meanings depending on the object type, i.e., whether the object is a boxed object or an immediate object. If the object is a boxed object, the remaining bits store the referenced memory location. The use of the `GC_posix_memalign()` call (from the Boehm Garbage Collector library) for the memory allocation obviates the loss of the four least significant bits and ensures that the four least significant bits of the returned address are zeros.

3.2 Metalanguage Interpreter

The metalanguage interpreter in pLisp is written in C. While the semantics of the metalanguage interpreter are easier and more natural to represent and implement in Lisp, this option is not available in the present situation: we are implementing pLisp from scratch and therefore do not have a core/kernel Lisp implementation from which to bootstrap. S-expressions which are input to the interpreter are pLisp objects, more specifically linked CONS cells, internally represented as `OBJECT_PTR` values. A subset of the pLisp object types, viz., atoms (excluding types like closures, of course) and CONS cells comprising atoms or other CONS cells, is thus accepted by the interpreter.

The workhorse of the CPS transformation process is the function `mcps()` [2]. This function accepts the source language expression, and depending on its type (i.e., abstraction, application, and so on), creates the corresponding closure *M* which, when invoked, would perform the code transformation for that type.

The CPS transformation process is also predicated on a class of secondary closures *m*, the purpose of which is to transform value expressions (identifiers and literals) in the source dialect (`plisp_IL`) into general expressions in the target dialect (`plisp_IL_CPS`). The closure *M* takes an argument of type *m*.

The partial evaluation semantics are captured in these two closures: the code executed by these closures would have, in a naive CPS transformation, formed a part of the generated code, thereby leading to code bloat and the attendant performance hit.

Listing 1 presents the data structures used by the interpreter. The structures `reg_closure_t` and `metacont_closure_t` are the realizations of the closures *m* and *M* respectively. The first three fields of each structure correspond to the implementation of a closure in a language like C, i.e., the function pointer representing the function and the machinery required to store the closed-over values. The function pointer field in effect specializes the closures: by assigning different functions to this field, we are able to handle the various source language constructs (`if`, `let`, and so on) in a modular way. In addition to these three fields, the structure for *m* has a field called `data`; the need for and usage of this field is explained in the next section.

Figure 5 depicts the object model underpinning the interpreter. In the interests of space, not all the elements corresponding to the language constructs are shown. It is to be noted that there isn't a one-to-one mapping between the language construct entities of *M* and *m* (e.g., there is an entity called `lambda_metacont_fn`,

[2] https://github.com/shikantaza/pLisp/blob/master/src/metacont.c

but no entity called lambda_cont_fn). The absence of such an entity is because it is not always the case that the body of M is of the form ($\mathcal{MCPS}[\![E]\!]\ \lambda V\,.\ \ldots$), which necessitates the presence of the symmetric entity. Also, the dependency or linkage between the transforming entities and the transformed entities is cleanly captured by the OBJECT_PTR reference; the use of this reference enables flexibility and allows us to switch the object representation easily if desired. Finally, please also note the self-referential loop labelled 'data' for reg_closure_t: the self-reference refers to the data field, which, while generic in intent, is used in practice to store only entities of class m (we have explained this further in subsection 4.2).

4 A DETAILED WALKTHROUGH

In order to explain the internals of the translation process and to bring out the mechanics of the interpreter better, we walk through the translation process for the pLisp if construct in this section.

4.1 An Abstract View

At an abstract level, the part of the implementation function mcps() that translates if constructs (more precisely, the function stored in the field mfn of the structure metacont_closure_t, corresponding to M) can be represented [15] by the function shown in Figure 6.

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

Figure 6: Transformation function for *if* construct

We deconstruct the function as follows.

(1) The function $\mathcal{MCPS}$ returns a closure (tagged as M in the previous section). The closed-over values for the returned closure depend on the source language construct in question; in this case, they are E_{test}, E_{then}, and E_{else}.
(2) The argument to this closure is another closure (tagged as m earlier), which handles the conversion of value expressions (literals and identifiers) in the source dialect.
(3) When the closure M is invoked with the argument, it evaluates $\mathcal{MCPS}$ on E_{test}, creates another closure of class m (this is explained below), and applies the former to the latter.
(4) The closure of class m mentioned above builds the target expression in the plisp_IL_CPS dialect. The closed-over values for this closure are E_{then}, E_{else}, and m (the argument to M itself). This is the canonical CPS transformation of if: evaluate the test expression, branch on to the CPS transformation of the consequent or the alternative expression based the truth-value of the test expression.
(5) The body of the closure defined by (λV_{test}.(let.. is an S-expression built up partly with literals like let and fresh symbols, and partly with return values of function applications in the metalanguage.
(6) $id \rightarrow mc$ and $mc \rightarrow exp$ are helper functions; the first converts an identifier to a closure of class m, while the second converts the metalanguage closure m into an abstraction in the target dialect (i.e., the CPS-transformed equivalent expression).

In summary, elements of the source expression are partitioned among multiple closures, and these closures (operating at different stages of the transformation) utilize these elements to recursively build the target expression.

4.2 Implementation

The definitions of the closure functions for transforming if constructs are provided in Listing 2.

The functions if_metacont_fn and if_reg_cont_fn slot into mfn and fn in the respective closure data structures presented in Listing 1. As mentioned earlier, the same data structures are repurposed to handle different constructs by suitably populating these slots as required. Adhering to the convention for closure implementation, the first argument to both these functions are their parent closure data structures themselves.

The computational steps of the abstract function $\mathcal{MCPS}$ are jointly realized in an imperative fashion in these two functions:

if_metacont_fn:

(1) Retrieve the closed-over values from the closure data structure.
(2) Convert the if test expression to a closure M by calling mcps().
(3) Create the regular closure object that would perform the actual transformation.
(4) Invoke the closure created in step (2) on the regular closure object and return the result.

if_reg_cont_fn:

(1) Retrieve the closed-over values from the closure data structure.
(2) Convert the consequent and alternative parts of the if expression to closures by calling mcps().
(3) Build the target expression by splicing together the S-expression from literals like let and fresh symbols, and from results of function calls of the above closures.

A couple of things are to be noted:

(1) The implementation contains calls like gensym() and list(), which are C equivalents of the standard lisp operators.
(2) The slot data in reg_closure_t is also used to store closed-over values (albeit not OBJECT_PTR values). However, such a slot is useful to logically separate such values as originate from the source language expressions (e.g., E_{then}) from values like m for purposes of clarity. There is also an element of type-safety: closed_vals, being of type OBJECT_PTR *, is safer from, e.g., an assignment perspective when compared to data (which is of type void *).

4.3 Another Example

We illustrate the flexibility of the translation framework with a brief look at the machinery for the translation of the let form (Figure 7).

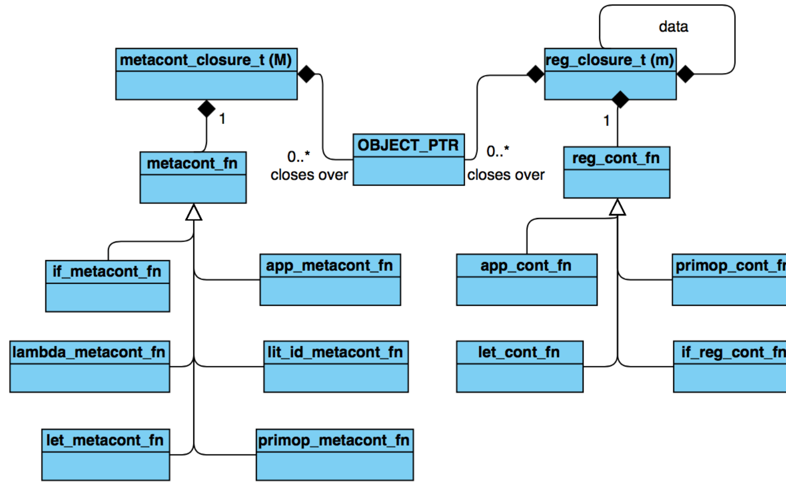


Figure 5: Transformation object model

Listing 1: Data structures used in metalanguage interpreter

```
1  //forward declarations
2  struct reg_closure;
3  struct metacont_closure;
4
5  typedef OBJECT_PTR (*reg_cont_fn)(struct reg_closure *, OBJECT_PTR);
6
7  typedef struct reg_closure
8  {
9    reg_cont_fn fn;
10   unsigned int nof_closed_vals;
11   OBJECT_PTR *closed_vals;
12   void *data;
13 } reg_closure_t;
14
15 typedef OBJECT_PTR (*metacont_fn)(struct metacont_closure *, struct reg_closure *);
16
17 typedef struct metacont_closure
18 {
19   metacont_fn mfn;
20   unsigned int nof_closed_vals;
21   OBJECT_PTR *closed_vals;
22 } metacont_closure_t;
```

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

Figure 7: Transformation function for *let* construct

This is the canonical transformation of `let`: each of the binding expressions E_i undergoes transformation (sequentially), and invokes its respective continuation. Finally the transformation of the `let` body E_{body}, which would have the references to the bindings I_i populated by the `let*` form, is invoked on m, which is the continuation that would have been provided by the whole `let` expression transformation context. Each of the nested closures (also of type m) closes over a part of the `let` binding components. The

Listing 2: Closure function definitions for *if* construct

```
1  OBJECT_PTR if_metacont_fn(metacont_closure_t *mcls, reg_closure_t *cls1)
2  {
3    OBJECT_PTR test_exp = mcls->closed_vals[0];
4    OBJECT_PTR then_exp = mcls->closed_vals[1];
5    OBJECT_PTR else_exp = mcls->closed_vals[2];
6
7    metacont_closure_t *test_mcls = mcps(test_exp);
8
9    reg_closure_t *cls = (reg_closure_t *)GC_MALLOC(sizeof(reg_closure_t));
10
11   cls->fn             = if_reg_cont_fn;
12   cls->nof_closed_vals = 2;
13   cls->closed_vals     = (OBJECT_PTR *)GC_MALLOC(cls->nof_closed_vals * sizeof(OBJECT_PTR));
14
15   cls->closed_vals[0]  = then_exp;
16   cls->closed_vals[1]  = else_exp;
17
18   cls->data = cls1;
19
20   return test_mcls->mfn(test_mcls, cls);
21 }
22
23 OBJECT_PTR if_reg_cont_fn(reg_closure_t *cls, OBJECT_PTR test_val)
24 {
25   OBJECT_PTR i_kif = gensym();
26
27   reg_closure_t *cls1 = (reg_closure_t *)cls->data;
28
29   OBJECT_PTR then_exp = cls->closed_vals[0];
30   OBJECT_PTR else_exp = cls->closed_vals[1];
31
32   metacont_closure_t *then_mcls = mcps(then_exp);
33   metacont_closure_t *else_mcls = mcps(else_exp);
34
35   reg_closure_t *kif_cls = id_to_mc(i_kif);
36
37   return list(3,
38               LET,
39               list(1, list(2, i_kif, mc_to_exp(cls1))),
40               list(4,
41                    IF,
42                    test_val,
43                    then_mcls->mfn(then_mcls, kif_cls),
44                    else_mcls->mfn(else_mcls, kif_cls)));
45 }
```

conversion process lends itself naturally to a recursive implementation, with the final closure (which acts on E_{body}) being the only non-recursive component.

These desired behaviours are captured elegantly in our implementation (Listing 3): the recursive behaviour is realized (at closure creation time) by setting the `fn` slot in `reg_closure_t` to a recursive function (`let_cont_fn_recur`), while the 'tail call' behaviour

is realized by setting the same slot to a non-recursive function (`let_cont_fn_non_recur`). The bindings and the body of the `let` are closed over both these categories of closure.

The same technique—specializing a closure into recursive and non-recursive variants as needed—is used for applications and primitive operations, which, similar to `let`, involve a variable number of sub-expressions that need to be CPS-transformed.

5 RELATED WORK

The number of Lisp/Scheme compilers is quite large; therefore in the interests of space we cover representative ones, highlighting the implementation techniques utilized by them that are of relevance.

The CHICKEN Scheme-to-C compiler [2, 5] employs a CPS-based compilation strategy, and CPS conversion is one of the steps in its compiler pipeline. However, the CPS conversion is written in Scheme itself, as opposed to the implementation language of the compiler (e.g., C). The CPS conversion algorithm is stated to be based on the relatively naive algorithm outlined in [1], and the optimizations induced by partial evaluation are realized at the later (explicit) stages in the compiler pipeline.

Not all Scheme compilers use CPS transforms to provide support for continuations. For example, Bigloo [3] implements `call/cc` by copying the execution context to the heap, while Guile [10] implements continuations by copying the C stack to the heap. A similar mechanism is employed by Gambit [7]. The Stalin Scheme compiler [13] utilizes a lightweight CPS conversion technique that relies on whole-program analysis, whereas normative CPS transformation is syntax-directed and is concerned with local (expression-level) code units.

Blocks [4] are an extension to the C, C++, and Objective-C implementations of Clang that provides a mechanism to create closures in these languages. In contrast, the current work implements closures in ANSI C using the standard mechanisms available in the language. Nested functions [9] are another available mechanism for realizing closures in C. However, the extent of these nested functions is limited to the containing scope, which falls short with respect to the needs imposed by the CPS transformation.

6 CONCLUSION

We presented an implementation of a partial evaluation based CPS transformation in the context of pLisp, a Lisp dialect and integrated development environment for beginners. The object model underpinning the metalanguage interpreter was presented, and the implementation was illustrated with a detailed walkthrough of the transformation for a single source language construct from both an abstract and an implementation perspective. The framework has been implemented in a modular fashion so that it is easy to swap in and swap out implementations of the transformation functions of the individual constructs, and to add support for new constructs. A further improvement to flexibility would be to set up the structure of target expressions in a declarative manner, and to code the transformation functions in such a way that the functions simply fill in the computed values into a pre-built S-expression template (somewhat along the lines of a context object with holes). This is planned to be taken up as future work.

REFERENCES

[1] Andrew W Appel. *Compiling with continuations*. Cambridge University Press, 2007.
[2] Henry G Baker. Cons should not cons its arguments, part ii: Cheney on the mta. *ACM Sigplan Notices*, 30(9):17–20, 1995.
[3] Bigloo Scheme. URL https://www-sop.inria.fr/indes/fp/Bigloo/.
[4] Blocks. URL https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Blocks/Articles/00_Introduction.html.
[5] CHICKEN Scheme. URL https://call-cc.org.
[6] Oliver Danvy and Andrzex Filinski. Representing control: A study of the cps transformation. *Mathematical structures in computer science*, 2(4):361–391, 1992.
[7] M Feeley. Gambit scheme. URL http://gambitscheme.org/wiki/index.php/Main_Page.
[8] Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 237–247, 1993.
[9] GCC Nested Functions. URL https://gcc.gnu.org/onlinedocs/gcc-4.0.4/gcc/Nested-Functions.html#Nested-Functions.
[10] Guile. URL https://www.gnu.org/software/guile/.
[11] Rajesh Jayaprakash. plisp: A friendly lisp ide for beginners. In *Proceedings of the 11th European Lisp Symposium*, 2018.
[12] Neil D Jones, Carsten K Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.
[13] Jeffrey Mark Siskind. Flow-directed lightweight closure conversion. Technical report, Technical Report 99-190R, NEC Research Institute, Inc, 1999.
[14] Guy L Steele Jr. Rabbit: A compiler for scheme. Technical report, Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1978.
[15] Franklyn Turbak, David Gifford, and Mark A Sheldon. *Design concepts in programming languages*. MIT press, 2008.

Listing 3: Closure creation for *let*

```
1  reg_closure_t *create_reg_let_closure(OBJECT_PTR    bindings,
2                                        OBJECT_PTR    full_bindings,
3                                        OBJECT_PTR    body,
4                                        unsigned int  nof_vals,
5                                        OBJECT_PTR    *vals,
6                                        reg_closure_t *cls)
7  {
8    reg_closure_t *let_closure = (reg_closure_t *)GC_MALLOC(sizeof(reg_closure_t));
9
10   if(cons_length(bindings) == 0) //last binding
11     let_closure->fn = let_cont_fn_non_recur;
12   else
13     let_closure->fn = let_cont_fn_recur;
14
15   let_closure->nof_closed_vals = nof_vals + 3;
16   let_closure->closed_vals     = (OBJECT_PTR *)GC_MALLOC(let_closure->nof_closed_vals
17                                                         * sizeof(OBJECT_PTR));
18
19   let_closure->closed_vals[0]  = bindings;
20   let_closure->closed_vals[1]  = full_bindings;
21   let_closure->closed_vals[2]  = body;
22
23   int i;
24   for(i=3; i<let_closure->nof_closed_vals; i++)
25     let_closure->closed_vals[i] = vals[i-3];
26
27   let_closure->data = cls;
28
29   return let_closure;
30 }
```