Defmacro for C

Lightweight, Ad Hoc Code Generation

Kai Selgrad[1], Alexander Lier[1]

Markus Wittman[2], Daniel Lohmann[1], Marc Stamminger[1]

[1] Friedrich-Alexander University Erlangen-Nuremberg

[2] Erlangen Regional Computing Center

European Lisp Symposium

May 6 2014, Paris, France

Proposition

We implemented an **S-Expression syntax for C**, embedded in CL, to **enable meta programming** in application domains where C is strongly established.

Outline

- Background information
- Presentation of syntax
- Implementation details
- Usage examples

Origins

The purpose of our work is to facilitate research in performance critical application domains, e.g.

- Computer Graphics,
- HPC / Simulation, and
- Systems Configuration.

The predominant environment in those areas is C-like.

Computer Graphics

Advanced computer graphics typically relies on Shader programming

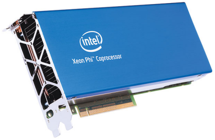
(e.g. using the OpenGL shading language)

or general purpose GPU programming languages.

(e.g. OpenCL, Cuda)

Heterogenous programming is not easily managed.

High Performance Computing



In HPC it is crucial to determine the optimal implementation of an algorithm for the hardware at hand.
The optimal approach for each combination of hardware platfrom

(e.g. Geforce GTX 780 vs Xeon Phi vs Unix cluster)

and algorithms

(e.g. Stencil computation to solve the heat equation)

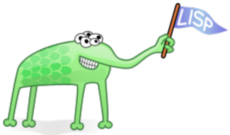
is hard to maintain.
Current findings have to be continuously reevaluated as hardware and algorithms evolve.

Why and how

- Wrote algorithm in GL Shading Language, need Cuda today.
- Wrote 900 ray tracers in 5000 lines of (C++ template) code. Can't change a single one.
- Want to check 7th alternative version of my algorithm,
  – can't read my code for `#ifdef`
  – copy&paste again?

Why and how

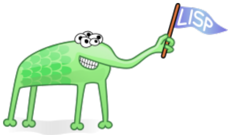
- Wrote algorithm in GL Shading Language, need Cuda today.
- Wrote 900 ray tracers in 5000 lines of (C++ template) code. Can't change a single one.
- Want to check 7th alternative version of my algorithm,
  – can't read my code for `#ifdef`
  – copy&paste again?



Being able to do comprehensive meta programming would be very nice for our applications, too.
S-Exps are easily parsed and manipulated, all the mechanism is provided, too :)

Why and how

- Wrote algorithm in GL Shading Language, need Cuda today.
- Wrote 900 ray tracers in 5000 lines of (C++ template) code. Can't change a single one.
- Want to check 7th alternative version of my algorithm,
  – can't read my code for `#ifdef`
  – copy&paste again?



Being able to do comprehensive meta programming would be very nice for our applications, too.
S-Exps are easily parsed and manipulated, all the mechanism is provided, too :)

We would be (and have been) very happy about feedback from the community.
Our approach is comparable to Parenscript, but for C-like languages.

An S-Exp Syntax for C

```
(function main () -> int
  (decl ((int c)
         (int nl 0))
    (while (!= (set c (getchar)) EOF)
      (if (== c #\newline)
          ++nl))
    (printf "%d\n" nl))
  (return 0))
```

`wc -l`

An S-Exp Syntax for C

```
(function main () -> int
  (decl ((int c)
         (int nl 0))
    (while (!= (set c (getchar)) EOF)
      (if (== c #\newline)
          ++nl))
    (printf "%d\n" nl))
  (return 0))
```

$\boxed{\texttt{wc -l}}$

$\xrightarrow{\text{cgen}}$

```
int main(void) {
    int c;
    int nl = 0;
    while ((c = getchar()) != EOF) {
        if (c == '\n')
            ++nl;
    }
    printf("%d\n", nl);
    return 0;
}
```

Implementation

```
(function main () -> int
  (decl ((int c)
         (int nl 0))
    (while (!= (set c (getchar))
               EOF)
      (if (== c #\newline)
          ++nl))
    (printf "%d\n" nl))
  (return 0))
```

$$\Big\downarrow\ \text{cgen}$$

```
int main(void) {
    int c;
    int nl = 0;
    while ((c = getchar())
           != EOF) {
        if (c == '\n')
            ++nl;
    }
    printf("%d\n", nl);
    return 0;
}
```

- Completely embedded in CL:
  - Case sensitive symbol names via reader.
  - Lisp vs CGen symbols via packages.
  - Destructuring of simple C expressions via reader.

Implementation

```
(function main () -> int
  (decl ((int c)
         (int nl 0))
    (while (!= (set c (getchar))
               EOF)
      (if (== c #\newline)
          ++nl))
    (printf "%d\n" nl))
  (return 0))
```

$$\Big\downarrow \text{cgen}$$

```
int main(void) {
    int c;
    int nl = 0;
    while ((c = getchar())
           != EOF) {
        if (c == '\n')
            ++nl;
    }
    printf("%d\n", nl);
    return 0;
}
```

- Completely embedded in CL:
  - Case sensitive symbol names via reader.
  - Lisp vs CGen symbols via packages.
  - Destructuring of simple C expressions via reader.
- Evaluation of CGen forms builds AST:

  ```
  (* (+ 1 2) x)
  ```

Implementation

```
(function main () -> int
  (decl ((int c)
         (int nl 0))
    (while (!= (set c (getchar))
               EOF)
      (if (== c #\newline)
          ++nl))
    (printf "%d\n" nl))
  (return 0))
```

$$\Big\downarrow\ \text{cgen}$$

```
int main(void) {
    int c;
    int nl = 0;
    while ((c = getchar())
           != EOF) {
        if (c == '\n')
            ++nl;
    }
    printf("%d\n", nl);
    return 0;
}
```

- Completely embedded in CL:
  - Case sensitive symbol names via reader.
  - Lisp vs CGen symbols via packages.
  - Destructuring of simple C expressions via reader.
- Evaluation of CGen forms builds AST:

```
(* (+ 1 2) x)
(* #<arith :op '+ :lhs 1 :rhs 2>
   #<name :name "x">)
```

Implementation

```
(function main () -> int
  (decl ((int c)
         (int nl 0))
    (while (!= (set c (getchar))
               EOF)
      (if (== c #\newline)
          ++nl))
    (printf "%d\n" nl))
  (return 0))
```

$\downarrow$ cgen

```
int main(void) {
    int c;
    int nl = 0;
    while ((c = getchar())
           != EOF) {
        if (c == '\n')
            ++nl;
    }
    printf("%d\n", nl);
    return 0;
}
```

- Completely embedded in CL:
  - Case sensitive symbol names via reader.
  - Lisp vs CGen symbols via packages.
  - Destructuring of simple C expressions via reader.
- Evaluation of CGen forms builds AST:

```
(* (+ 1 2) x)
(* #<arith :op '+ :lhs 1 :rhs 2>
   #<name :name "x">)
#<arith :op '*
        :lhs #<arith :op '+
                     :lhs 1
                     :rhs 2>
        :rhs #<name :name "x">>
```

Demonstrations

We'll show:

- A small DSL.
- A comparison of different approaches to C code generation.
- Direct use of our generated AST is not covered.

See the paper for:

- Numerous small examples.
- A larger evaluation of an example from HPC.

A Simple DSL

Higher level scripting languages usually have nice support for Regular Expressions. Libc is rather verbose.

Using
```
(match text
  ("([^.]*)" (printf "proper list.\n"))
  (".*\."  (printf "improper list.\n")))
```

instead of

```
regex_t reg;
int reg_err;
reg_err = regcomp(&reg, "([^.]*)", REG_EXTENDED);
if (regexec(&reg, text, 0, 0, 0))
    printf("proper list.\n");
else {
    reg_err = regcomp(&reg, ".*\.", REG_EXTENDED);
    if (regexec(&reg, text, 0, 0, 0))
        printf("improper list.\n");
}
```

is easily accomplished using cgen.

A Simple DSL

```
(defmacro match (expression &rest clauses)
  `(macrolet
     ((match-int (expression &rest clauses)
        `(progn
           (set reg_err
                (regcomp &reg ,(caar clauses) REG_EXTENDED))
           (if (regexec &reg ,expression 0 0 0)
               (progn ,@(cdar clauses))
               ,(lisp (if (cdr clauses)
                          `(match-int
                             ,expression
                             ,@(cdr clauses))))))))
     (decl ((regex_t reg)
            (int reg_err))
       (match-int ,expression ,@clauses))))
```

A Simple DSL

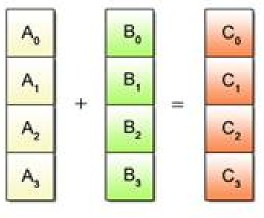
```
(defmacro match (expression &rest clauses)
  `(macrolet
     ((match-int (expression &rest clauses)
        `(progn
           (set reg_err
                (regcomp &reg ,(caar clauses) REG_EXTENDED))
           (if (regexec &reg ,expression 0 0 0)
               (progn ,@(cdar clauses))
               ,(lisp (if (cdr clauses)
                          `(match-int
                             ,expression
                             ,@(cdr clauses)))))))
     (decl ((regex_t reg)
            (int reg_err))
       (match-int ,expression ,@clauses))))
```

A Simple DSL

```
(defmacro match (expression &rest clauses)
  `(macrolet
     ((match-int (expression &rest clauses)
        `(progn
           (set reg_err
                (regcomp &reg ,(caar clauses) REG_EXTENDED))
           (if (regexec &reg ,expression 0 0 0)
               (progn ,@(cdar clauses))
               ,(lisp (if (cdr clauses)
                          `(match-int
                             ,expression
                             ,@(cdr clauses))))))))
     (decl ((regex_t reg)
            (int reg_err))
       (match-int ,expression ,@clauses))))
```

Comparison: Providing a SIMD Notation

In the following we'll describe several variants to add a notation for SIMD operations to C.



```
__m128 x, y, z;
__m128 c_0_5 = _mm_set_ps(0.5);
_mm_mul_ps(
    _mm_add_ps(
        x,
        _mm_add_ps(y, z)),
    c_0_5);
```

This is a nice example of why people like operator overloading, but please bear with me.

SIMD Notation: Engineering

`intrinsify`: A simple preprocessor copying input to output.

- Recognizing //#.
- Parses calculator grammer using Flex and Bison.
- Pretty prints after whole expression is read.

```
__m128d accum, factor;
for (int i = 0; i < N; i++) {
    __m128d curr = _mm_load_pd(base + i);
    //#INT accum = accum + factor * curr;
}
```

```
__m128d accum, factor;
for (int i = 0; i < N; i++) {
    __m128d curr = _mm_load_pd(base + i);
    //#INT accum = accum + factor * curr;
    accum = _mm_add_pd(accum, _mm_mul_pd(factor, curr));
}
```

SIMD Notation: Hacking

We also show a very ad hoc solution using Python (with Mako).

```
__m128d accum, factor;
for (int i = 0; i < N; i++) {
    __m128d curr = _mm_load_pd(base + i);
    ${with_sse(set_var('accum',
                       add('accum',
                           mul('factor', 'curr'))))};
}
```

SIMD Notation: $t \in (0, 1)$

Using our generator:

```
(decl ((__m128d accum)
       (__m128d factor))
  (for ((int i 0) (< i N) i++)
    (intrinsify
      (decl ((mm curr (load-val (aref base i))))
        (set accum (+ accum (* factor curr)))))))
```

SIMD Notation: $t \in (0, 1)$

Using our generator:

```
(decl ((__m128d accum)
       (__m128d factor))
  (for ((int i 0) (< i N) i++)
    (intrinsify
      (decl ((mm curr (load-val (aref base i))))
        (set accum (+ accum (* factor curr))))))))
```

Flex&Bison: 1,500 sloc / String based: 60 sloc / Cgen: 45 sloc

SIMD Notation: $t \in (0, 1)$

Using our generator:

```
(decl ((__m128d accum)
       (__m128d factor))
  (for ((int i 0) (< i N) i++)
    (intrinsify
      (decl ((mm curr (load-val (aref base i))))
        (set accum (+ accum (* factor curr)))))))
```

Flex&Bison: 1,500 sloc / String based: 60 sloc / Cgen: 45 sloc

Extensibility. Convert numbers to SIMD constants:

Flex&Bison: ✓ / String based: – / Cgen: ✓

Conclusion

Limitations

- New target languages require changes at the AST level.
- Late type checking.
- Line Numbers.

Summary

- Lightweight, especially when compared to general purpose code generation for C.
  Lowering the entry barrier to code generation.
- Complete support for CL macros.
- Very simple meta programming targetting C.

Thank you for your attention.

References

Page 3	Stanford dragon, Stanford 3D Scanning Repository.
Page 5	`https://computing.llnl.gov/tutorials/parallel_comp/`, `http://spectrum.ieee.org/semiconductors/processors/what-intels-xeon-phi-coprocessor-means-for-the-future-of-supercomputing`, `http://www.pc-erfahrung.de/hardware/grafikkarte/vga-grafikchips-desk/geforce-gtx-serie/nvidia-geforce-gtx-285-295-gt200b.html`
Page 6	`http://michellewelti.blogspot.de/2010/11/frustration-common-emotional-response.html`
Page 7	`http://www.lisperati.com/logo.html`
Page 8	`http://researchinprogress.tumblr.com/`
Page 10	Sample from “The C Programming Language”, K&R
Page 14	Sample from “The C Programming Language”, K&R
Page 23	`https://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/BasicsOfSIMDProgramming.html`