MetaC

David McAllester

June 5, 2018

Abstract

MetaC is a C extension supporting a read-eval-print loop (REPL) capable of incremental procedure redefinition (dynamic linking) and the general look-and-feel of python programming. MetaC also extends C with symbolic programming features for writing sophisticated preprocessors (compilers) as packages of computed macros. Computed macros can do sophisticated type inference or data-flow analysis as part of macro expansion. MetaC supports computed macros with the high level symbolic progrmming features of pattern matching and pattern instantiation. Pattern instantiation is done with backquote — a language feature of Lisp. Generating C code using backquote (pattern instantiation) is facilitated by the use of a universal algebraic syntax (UAS). UAS is a "universal" correspondence between character strings and expression trees.

1 Introduction and Overview

We start with hello world.

1.1 Hello World

Once one has cloned the MetaC repository (git@github.com:mcallester/MetaC.git) one can cd to the repository directory and type

```
bash$ make MC
...

bash$ MC

MC>
```

We can then evaluate a hello world expression.

```
MC> ‘{hello world}

hello world

MC>
```

In the above example the backquote expression given to the REPL macro-expands to a C expression which evaluates to the expression “hello world”. Backquote is described in more detail below.

1.2 C Statements, C expressions, and UAS Expressions

We can also declare global variables and execute statements from the REPL.

```
MC> int x[10];

done

MC> for(int i = 0; i < 10; i++)x[i] = i;

done

MC> for(int i = 0; i < 10; i++)fprintf(stdout,"%d",x[i]);

0123456789done

MC>int_exp(x[5])

5

MC>
```

In C syntax one distinguishes C expressions from C statements. A C statement is executed for effect while a C expression is evaluated for value. The REPL can be given either a statement or an expression. When given a C statement, the REPL simply prints “done” after executing the statement. When given a C expression the REPL computes and prints the value. The REPL assumes that the value of a C expression is a UAS tree (also called a UAS expression giving a second, and confusing, sense of the term “expression”). The procedure `int_exp` above converts an integer to a UAS expression. UAS expressions are abstract syntax trees but can viewed as representations of strings.

If a C statement executes a `return` outside of any procedure then that value is returned to the REPL and printed. For example, the above session can be extended with the following.

```
MC>{int sum = 0; for(int i = 0; i < 10; i++)sum += x[i]; return int_exp(sum);}

45

MC>
```

1.3 Definitions of Types and Procedures

One can also define new data types and procedures from the REPL.

```
MC> typedef struct expliststruct{
      expptr car;
      struct expliststruct * cdr;
    } expliststruct, *explist;

done

MC> explist mycons(expptr x, explist l){
      explist cell = malloc(sizeof(expliststruct));
      cell->car = x;
      cell->cdr = l;
      return cell;}

done

MC> expptr list_exp(explist l){
     if(l == NULL) return NULL;
     return `{ ${l->car} ${list_exp(l->cdr)} };
}

done

MC> list_exp(mycons(`{foo},mycons(`{bar},NULL)))

foo bar

MC>
```

Procedures can also be redefined at the REPL provided that the signature

(argument types and return type) remains the same. In the current implementation (June 5, 2018) changing a procedure signature requires restarting the MetaC REPL. This restricting ensures meaningful C type checking in the presence of dynamic linking.

1.4 The Global Variable Array Restriction

To simplify dynamaic linking, all global data variables must be arrays. One can always use single element arrays to represent non-array variables. To make this more convenient we allow single element arrays to be declared with an initial value in a manner similar to non-array data variables. For example, we can declare and assign a variable `y` as follows.

```
MC> int y[0] = 2;

done

MC> y[0] += 1;

done

MC> int_exp(y[0])

3

MC>
```

Here assignments to $y[0]$ are allowed but assignments to y are not — assignment to array variables are not allowed in C. As noted above, this restriction greatly simplifies dynamic linking of data variables.

1.5 The Single Symbol Type Restriction

To simplify the implementation of MetaC all type expressions appearing in procedure signatures and global array declarations must be single symbols. Arbitrary types can be given single symbol names using `typedef`.

2 Backquote and UAS Expression Trees

We now consider backquote and UAS expressions in more detail. Backquote can be used in a manner analogous to string formatting.

```
MC> expptr friend[0] = `{Bob Givan};

done

MC> int height[0] = 6;

done

MC> `{My friend ${friend[0]} is ${int_exp(height[0])} feet tall.}

My friend Bob Givan is 6 feet tall.

MC>
```

While UAS expressions can be viewed as representations of strings, UAS expressions are actually abstract syntax trees. The tree structure plays an important role in pattern matching. The tree structure is more apparent in the following example.

```
MC> expptr x[0] = `{a+b};

done

MC> `{bar(${x[0]})}

bar(a+b)

MC>
```

The pattern `foo(!x)` will match the expression `foo(a+b)` with `x` bound to the expression (tree) `a+b`. Pattern matching is described in more detail below.

2.1 Pattern Matching

MetaC provides a pattern matching case statement. The general form of the case construct is the following.

```
ucase{e;
  {<pattern1>}:{<body1>}
   ...
  {<patternn>}:{<bodyn>}}
```

Variables are marked in patterns by the symbol tags `!` and `?`. Variables tagged with `!` can bind to any UAS expression while variables tagged with `?` can only bind to atomic symbols. For example we might write the following.

```
int value(expptr e){
  ucase{e;
   {!x+!y}:{return value(x)+value(y);}
   {!x*!y}:{return value(x)*value(y);}
   {(!x)}:{return value(x);}
   {?z}:{return symbol_int(z);}}
return 0;
}
```

In many cases the choice of the particular tree structure imposed by the MetaC reader is unimportant. For example, the expression `a+b+c` will match the pattern `!x+!y+!z` independent of whether + is left associative or right associative. But the tree structure does matter in other cases. The pattern `!x*!y` will not match the string `a+b*c` because `+` is above (outer to) `*` in the abstract syntax tree. The pattern `!x*!y` does match `(a+b)*c`.

As of June 2018 repeated variables, such as in the pattern {`!x + !x`}, are not supported (and not checked for). If a variable is repeated it will be matched at its last occurrences under depth-first left to right traversal.

2.2 Gensym and Macro Definitions

MetaC supports computed macros which can exploit pattern matching, backquote pattern instantiation, and other (bootstrapped) high level language features. As a very simple example we can define a `dolist` macro at the REPL.

```
MC> umacro{mydolist(?x, !L){!body}}{
     expptr rest = gensym(‘{rest});
     return ‘{for(explist ${rest} = ${L};
                  ${rest} != NULL;
                  ${rest} = ${rest}->cdr;)
               {expptr ${x} = ${rest}->car; ${body}}}}

done

MC>macroexpand(‘{dolist(item,list){f(item);}})

for(explist _mcgen_rest1=list; _mcgen_rest1 !=NULL; _mcgen_rest1=_mcgen_rest1->cdr;)
    {expptr item=_mcgen_rest1->car;
```

```
    f(item);}

MC>
```

The general form of a macro definition is

```
umacro{<pattern>}{<body>}
```

where instances of ¡pattern¿ in MetaC code are to be replaced by the value of ¡body¿ under the variable bindings determined by the match. The procedure `macroexpand` takes a UCS expression an returns the result of repeatedly expanding macros until no more macro expansion is possible. Macro expansion can generate effects as well as return an expansion. The REPL performs macro expansion on the given expression and also performs the effects of that expansion. `umacro` is itself a MetaC macro and we can feed the above macro definition to the REPL.

The macro expansion of the above macro definition defines a procedure to compute the macro expansion and installs that procedure on a macro property of the symbol `dolist`. In general a macro patterns must either be an application expression or a binary connection expressions. The procedure for macro expansion is attached to either the head symbol of the application or the binary operator of the binary connection expression.

3 Universal Algebraic Syntax (UAS)

It is not obvious how to implement light weight expression quotation and expression pattern matching for C expressions. The syntax of C is complex. We bypass this complexity by introducing universal algebraic syntax (UAS). UAS is a compromise between the extreme syntactic simplicity of Lisp and the desire for a syntax that is more similar to C. While not as minimal as Lisp, UAS is still simple — we have the following eight syntactic categories.

- A symbol. A symbol represents a string of alpha-numeric characters (upper and lower case letters of the alphabet, plus the decimal numerals, plus underbar). For example `foo_bar1`.
- A tagged symbol. A tagged symbol is either a symbol, a symbol-tag followed by a symbol, or a symbol tag followed by the NULL expression. The symbol tag characters are `$`, \, `‘`, `!`, `?`, `#` and `%`. For example we have the tagged symbols `?foo` and `!foo`.
- A quoted string. For example `"!:&?;"` or `’a’`. In UAS there is no distinction between `"foo"` and `’foo’` other than the choice of the quotation

character (which must match). To avoid confusion newline characters are not allowed in strings (the two character sequence `/n` is ok).

- A parenthesis expression. This is an expression of the form (e), $\{e\}$, or $[e]$ where e is an expression. In MetaC (e) is always a different expression from e.

- An application expression. This is a tagged symbol (possibly untagged) followed by a parenthesis expression or an application expression followed by a parenthesis expression. For example `foo(x)` or `foo(x)(y)` or `‘{a}`. The formation of application expressions is tighter than (higher precedence than) the formation of binary connective expressions or semicolon-terminated expressions.

- A binary connective expression. For example `x=3` is the binary connective expression with connective `=` and symbol arguments `x` and `3`. Binary conectives are described in more detail below.

- A semicolon-terminated expression. For example `x=3;` or `f(x);`. All binary connectives are tighter than (higher precedence than) the formation of semicolon-terminated expressions.

- The null expression and the null connective. In order to ensure that the UAS reader can read any parenthesis-balanced string we can use the null expression as arguments and the null connective to combine pairs of adjacent expressions. This is described in more detail below.

Universal algebraic syntax is “universal” in that it has the following three properties.

1. UAS trees are semantics-free. They are simply trees and can be viewed as representations of character strings.

2. Any parenthesis-balanced character string can be represented (up to white space equivalence) by a UAS tree. Here parenthesis-balanced means that every open parenthesis, brace or bracket has a matching closing character and that string quotations are properly closed. Two strings are white space equivalent if they are the same after removing all white space characters (space, tab and return) but leaving a single space character after each symbol (to separate symbols).

3. The printer inverts the reader. If we read a string s into a UAS expression e and then print e back into a string s' we have that s and s' are white-space equivalent. This implies that s and s' both read to the same expression e.

These universality properties ensure that one can assign semantics to expressions based on the strings that they represent. We can assign C semantics to

an expression by printing the expression and passing the resulting string to a C compiler. This string-based semantics is not always compositional with respect to three structure of UAS trees. However, the tree structure imposed by the reader is designed to approximate the compositional structure of C syntax. In most cases pattern matching on UAS expressions recovers substructure that is semantically meaningful under C semantics.

The above discussion ignores the presence of C comments. The MetaC reader first preprocessing the input string to remove comments. The Meta preprocessor also divides long character strings, such as entire code files, into segments where each segment is read as a separate expression. The partitioning into segments is different in file inputs than in REPL inputs. For file inputs a new segment starts at the beginning of any line whose first character is not a space, tab or close character (right parenthesis, brace or bracket). A quoted return — the character '\\' followed by '\n' — does not start a new line. This is important for "multi-line" preprocessor definitions. A file must be parenthesis-balanced within each segment. For REPL inputs the segment ends at the first return character outside of parenthesization independent of further input characters. This allows the REPL to respond immediately to the top level return character.

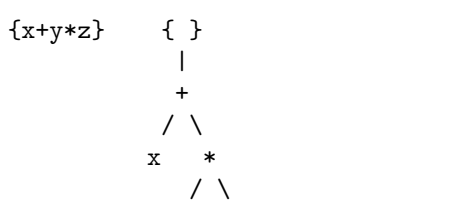
3.1 Binary Connectives

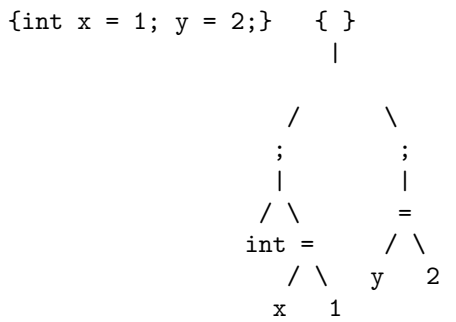
The binary connectives consist of all non-empty strings over the characters comma, {=, ∼, <, >}, {|, &}, {+, -}, {*, /}, ., @, and ^. Single-character connectives are outermost to innermost in the order given (low precedence to high precedence) where set brackets indicate same-precedence. All binary connectives are infix. The last four single-chracter connectives are left-associative and the others are right-associative. All multi-character binary connectives have the same precedence which is higher than (innermost to) all single character connectives other than :. Multi-character connectives are left associative. The null connective, described below, is right associative and is outermost to (lower precedence) than all other binary connectives except ; and comma. This precedence of the null connective causes C argument lists and C statement sequences to have friendly tree structure. Expressions can also be connected by the null connective — a connective which prints as the empty string. Connection by the null connective is higher priority than all binary connectives other than comma. Making null connection higher priority than comma allows proper pattern matching of C procedure argument lists.

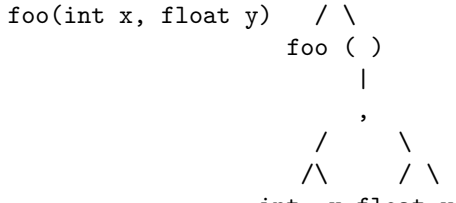
3.2 Examples

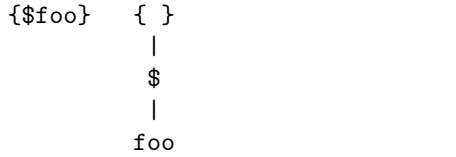
We now give a set of examples of character stings and their corresponding representation as UAS expression trees.

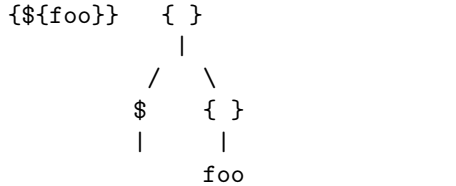

(foo bar)
( )
foo
bar

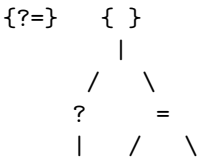

{x+y*z}
{ }
+
x
*
y
z


{int x = 1; y = 2;}
{ }
;
;
int =
=
x
1
y
2


foo(int x, float y)
foo ( )
,
int x float y

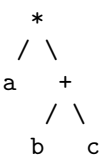

{$foo}
{ }
$
foo


{${foo}}
{ }
$
{ }
foo

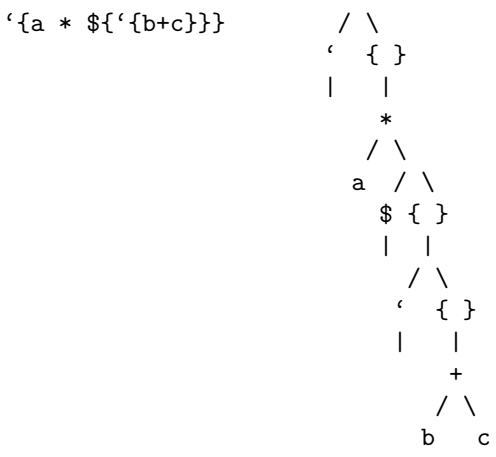


3.3 A More Precise Semantics of Backquote

There are UAS trees that cannot be represented by a string but can be constructed by backquote. For example we have that `+` is lower precedence than `*` and hence the tree



cannot be represented by a string. It can, however, be created with backquote. Backquote and pattern matching both operate on trees rather than character strings. The above tree can be created as the value of `‘{a * ${‘{b+c}}}`. In general a subexpression of the form `${e}` inserts the value of e at the node of the syntax tree for the backquote expression where the `${e}` subexpression occurs. The backquote expression has the following tree.



The general form of a backquote expression is `‘{<exp1>}` where the value of this is the given expression `<exp1>` with the nodes in the tree for `<exp1>` at which subexpressions of the form `${<exp2>}` appear replaced with the computed

C value of `<exp2>`. Under this evaluation rule the value of above backquote expression is the preceding tree which has no string representation.

Unfortunately the above simple evaluation rule for backquote expressions is incomplete. One of the most confusing situations is where the expansion of a macro contains a backquote. Writing such a macro typically involves nested backquotes. While nested backquotes are confusing, and should be avoided when possible, nested backquotes are supported. We consider a series of backquote expressions each of which evaluates to the previous one.

First we have

$$`\{a + b + \$\{z\}\} \tag{1}$$

If the value of variable `z` is the expression `c` then the value of expression (1) is the expression `a+b+c`. The unary operator `$` can be included in an expression constructed with backquote by quoting it. This gives our second expression.

$$`\{`\{a + \$\{y\} + \backslash\$\{z\}\}\} \tag{2}$$

If the value of variable `y` is the expression b then the value expression of (2) is expression (1). Quotation expression can be included in the expression by adding another layer of quotation as in the following.

$$`\{`\{`\{\$\{x\} + \backslash\$\{y\} + \backslash\backslash\$\{z\}\}\}\} \tag{3}$$

If the value of variable `x` is the expression `a` then the value of expression (3) is expression (2).

4 Interning, Property Lists, and Memory Management

4.1 Expression Interning

4.2 Expression Properties

4.3 Undo Stacks and Undo Memory Management

4.4 The MetaC Execution Stack for Debugging

4.5 Allocating Memory from the MetaC Execution Stack

5 System Features

5.1 The Emacs Interactive Development Environment

5.2 MetaC Source Code File Management

MetaC macro-expander executables macro-expand MetaC source code files .mc into a C source file .c. The MetaC macro-expander applications are written in MetaC — MetaC is bootstrapped on itself. The expanders are defined in layers where each layer has a set of active macros. Each layer has both an expander executable and a REPL with the same set of macros pre-installed. A new layer with more installed macros can be constructed using the previous layer to expand files with additional code including additional macro definitions.