CLAM: The Concise Linear Algebra Manipulation Language

Jeremy Andrus and Robert Martin and Kevin Sun and Yongxu Zhang
{jca2119, rdm2128, kfs2110, yz2419}@columbia.edu

Columbia University
COMS W4115: Programming Languages and Translators

November 1, 2011

Language Reference Manual

1 Introduction

The CLAM programming language is a linear algebra manipulation language specifically targeted for image processing. It provides an efficient way to express complex image manipulation algorithms through compact matrix operations. CLAM programs are first compiled into a "C" module which is further compiled into a machine binary by an existing This reference is inspired by the C reference manual [1].

2 Lexical Conventions

2.1 Tokens

The tokens in CLAM are broken down as follows: We have reserved keywords, identifiers, constants, control characters, and operators. The end of a token is defined by the presence of a newline, space, or tab character (whitespace), or by the presence of a character that cannot possibly be part of the current token.

2.2 Comments

Comments are demarcated with an opening /* and closing */, as in C. Any characters inside the comment boundaries are ignored. Comments can be nested.

2.3 Keywords

The reserved keywords in CLAM are:

Image	Import	Int8	Uint8
Kernel	Export	Int16	Uint16
Channel	Angle	Int32	Uint32
Calc	Float		

2.4 Identifiers

Identifiers are composed of an upper or lower-case letter immediately followed by any number of additional letters and/or digits. Identifiers are case sensitive, so “foo” and “Foo” are different identifiers. Identifiers cannot be keywords, and underscores are disallowed.

2.5 Constants

In CLAM there are 3 types of constants: numeric constants, calculation constants, and string literals.

2.5.1 Numeric Constants

Integers are repesented by a series of number characters.

Angles are represented by a series of number characters with an optional period character, followed by a lower-case “a”.

Floats are represented by a series of number characters with an optional period character, followed by a lower-case “f”.

2.5.2 Calculation Constants

Calculation constants are represented by an opening curly brace, followed by a series of *numeric-expressions* separated by whitespace or comma characters. The comma characters represents the division between the rows of the matrix. Each row must have the same number of *numeric-expressions*, but the matrix need not be square.

A calculation constant may also have an optional fraction preceding it, which indicates that every value in the matrix should be multiplied by that fraction. The fraction will be expressed as an opening bracket character, a *numeric-expression* representing the numerator, a forward-slash character, a *numeric-expression* representing the denominator, and a closing bracket character.

```
{ numeric-expr numeric-expr ... , numeric-expr numeric-expr...}
[numeric-expr / numeric-expr ]{ numeric-expr numeric-expr ... , numeric-expr numeric-expr...}
```

The following is an example of a calculation constant.

```
Calc sobelGy := [1 / 9]{1 3 1 , 2 -5 2 , 1 3 1 }
```

2.5.3 String Literals

String constants are demarcated by double quote characters or single quote characters. Consecutive string constants will be automatically appended together into a single string constant.

"*string-constant*"

3 Meaning of Identifiers

3.1 Basic Types

There are four basic types defined by the CLAM language. Type identifiers always begin with an uppercase letter followed by a sequence of zero or more legal identifier characters. The list of built-in types is as follows:

```
Channel
Image
Calc
Kernel
```

3.1.1 Atom Types

The `Channel` and `Calc` built-in types may be further modified to specify individual element, or "atom" types. This specifies either the type of each element of the matrix which makes up the `Channel`, or the type of the resulting calculation performed by a `Calc` object. An *atom-type* identifier is denoted using the `<` and `>` characters immediately following the identifier of the object whose atom type is being specified:

basic-type identifier<*atom-type*>

Legal *atom-types* are as follows:

Uint8
Uint16
Uint32
Int8
Int16
Int32
Angle

3.2 Type Qualifiers

Discuss the use of the @ symbol to denote a special matrix which when used in calculation does not produce in a discrete channel in the result.

Discuss the use of another symbol which tells the matrix composition that calculation must finish before proceeding (i.e. it actively inhibits parallel execution of convolution - probably useful if a subsequent calculation requires a neighborhood of previously calculated values).

4 Objects and Definitions

An *object* in CLAM is either a named collection of `Channels`, called an `Image`, or a named collection of calculation basis, called a `Kernel`. A `Channel` is a mathematical matrix of numeric values whose individual components are not directly accessible via CLAM language semantics – `Channel` values are manipulated via the convolution operator (see 5.5). A calculation basis, known as a `Calc`, is a collection of either calculation constants (see 2.5.2) or calculation expressions (see 5.6), or both.

4.1 `Image` objects

An `Image` is a collection of named `Channels`. `Channels` can be dynamically added using the channel composition operator (see section 5.8.3, or by assigning to a previously undeclared `Channel` name.

For example, to create a gray-scale image from a single, pre-existing `Channel`:

```
Image outImg;
outImg:Red = calcImg:G;
outImg:Green = calcImg:G;
outImg:Blue = calcImg:G;
```

4.2 `Kernel` objects

A `Kernel` is an ordered collection of calculation basis which is used by the convolution operator (see section 5.5). Each calculation basis can be either a calculation constant (see 2.5.2) or a calculation expression (see 5.6). A `Kernel` is composed either using the composition operator (see section 5.4.1), or the `|=` assignment operator (see section 5.8.3).

To see how a `Kernel` is used in calculation, see section 5.5.

5 Expressions

5.1 Primary Expressions

identifiers, constants, strings. The type of the expressions depends on the identifier, constant or string.

5.2 Unary Operators

There are two unary operators in CLAM, and they are only used with a numeric-valued operand such as a numeric constant (see 2.5.1). These expressions are grouped right-to-left:

`+`*numeric-expression*
`-`*numeric-expression*

5.2.1 `+` operator

This operator forces the value of its numeric operand to be positive. The resulting expression is of numeric type with a value equal to the absolute value of the numeric operand.

5.2.2 `-` operator

This operator forces the value of its numeric operand to be negative. The resulting expression is of numeric type with a value equal to the negative of the numeric operand.

5.3 Channel/Calc Expresions

`Channel` and `Calc` types are the basis of `Image` and `Kernel` objects respectively. There are several operators that manipulate `Channel`s and `Calc`s.

5.3.1 : operator

Extract or use an individual `Channel` in an image.

image-identifier : *channel-identifier*

The resulting expression has a type corresponding to the extracted `Channel`.

5.3.2 `$()` operator

This operator forces the evaluation of a previously defined `Image Channel`. It is generally used in the context of a convolution operation.

`$(`*channel-expression*`)`

The resulting expression has a type corresponding to the calculated `Channel`.

5.4 Composition Operators

These operators compose an `Image` from one or more `Channels`. All channel composition operators are left-to-right associative.

5.4.1 `|` operator

Compose two (or more) `Channel`s or `Calc`s. The resulting expression is a *multi-channel-expression*, or *multi-calc* expression, and can be assigned to either an `Image` or a `Kernel` object respectively.

channel-expression `|` *channel-expression*
multi-channel-expression `|` *channel-expression*
calc-expression `|` *calc-expression*
multi-calc-expression `|` *calc-expression*

Note that `Channel`s and `Calc`s are appended in order, and subsequent operations may rely on this order.

5.5 `**` operator

MISSING: Talk about the core of our language. . . the convolution operator

5.6 Escaped "C" Expression

MISSING: Talk about the `#[...]#` operator.

5.7 I/O Expressions

5.7.1 `imgread` expression

The `imgread` expression reads in an `Image` object from a known image format located on the file system. The expression results in an `Image` object which can be assigned using the `=` operator (see section 5.8.1). The resulting `Image` object has 3 `Channel`s named *Red*, *Green*, and *Blue*. Each of the channels correspond to the red, green, and blue image data read into the `Image` object. This expression is invoked as a "C" style function, and expects 1 parameter: the path of the image file to read.

`imgread(` *string-constant* `)`

5.7.2 `imgwrite` expression

The `imgwrite` expression writes out an `Image` object to a known image format. It requires that the `Image` object has at least 3 named `Channels`: *Red*, *Green*, and *Blue*. This expression has no type (null type), and is invoked as a "C" style function. It expects 3 parameters: the first parameter is an `Image` identifier, the second is the image format, and the the third is the path to which the image should be written.

`imgwrite(` *image-identifier* `,` *string-constant* `,` *string-constant* `)`

5.8 Assignment Expressions

5.8.1 `=` assignment operator

Assigns the value of the right operand to the left operand, copying data as necessary. The types of both operands must match.

5.8.2 `:=` assignment operator

Assigns a calculation constant (see section 2.5.2), or escaped "C" expression (see section 5.6) to a `Calc` object.

5.8.3 `|=` assignment operator

Add a `Channel` or a `Calc` object to an `Image` or `Kernel` object. Assignments using this operator are ordered by statement order, and subsequent operations can rely on this order.

Note that a `Calc` object assigned to an `Image` object must be evaluated using the `$()` operator (see section 5.3.2) before being using in calculation or further assignment.

6 Statements

Statements in CLAM always end in a semi-colon. No statement can return a value. All statements either declare a variable, define or modify the definition of a variable, or execute some calculation based on previously declared variables with the result stored in previously declared variables.

7 Program Definition

A program in the CLAM language is simply a sequence of statements which are executed in order.

8 Scope Rules

All identifiers in the CLAM language are global, except for the identifiers prefixed with an @ symbol, which can only be accessed by their own calculation.

In an escaped C block that defines a channel, the existing channels for an image will be in scope when the block is executed. Because this block will be executed on every pixel, the name of the channel will bind to the current pixel value for that channel. These bindings will be resolved when the channel is calculated; not when it is defined.

9 Declarations

All variables must be declared before they can be used. However, variable declarations can be made at any point in a program. Variables become usable after the end of the semi-colon of the statement in which its contained.

10 Grammar

MISSING: explicitly spell out all the above rules in a concise way.

11 Examples

The following example implements a Sobel image filter using the CLAM language.

```
/* read an image into the 'srcimg' variable */
Image srcimg = imgread("someimage.jpg");

/* define a luninance channel for this image
 * (Red, Green, and Blue channels are implicit from imgread)
 * No type specification with <> defaults to Uint8 */
Calc Lum := #[(3*Red + 6*Green + 1*Blue)/10]#;
srcimg |= Lum;

/* Kernel definitions are ordered i.e. the channels
 * are calculated in the order they are defined */
Calc sobelGx<Uint8>     := [1 / 1]{ -1  0 +1 , -2  0 +2 , -1  0 +1 };
Calc sobelGy<Uint8>     := [1 / 1]{ +1 +2 +1 ,  0  0  0 , -1 -2 -1 };
Calc sobelG<Uint8>      := #[sqrt(sobelGx*sobelGx + sobelGy*sobelGy)]#;
Calc sobelTheta<Angle>  := #[arctan(sobelGy/sobelGx)]#;

Kernel sobel = @sobelGx | @sobelGy | sobelG;
sobel |= sobelTheta;

/* Convolution - resulting image will have the same number
 * of channels as the filtering kernel. */
```

```
Image edges = $(srcimg:Lum) ** sobel;

/* compose an output image which is a grayscale of
 * edge gradient magnitude */
Image output;
output:Red   = edges:sobelG;
output:Green = edges:sobelG;
output:Blue  = edges:sobelG;

imgwrite( output, "jpg", "edges_of_someimage.jpg");
```

References

[1] B. W. Kernighan and D. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.