Maxwell Language Design

F. Schuiki (fschuiki@student.ethz.ch)

June 17, 2014

Contents

1 Sequential Expressions aka Blocks

Since everything is an expression in Maxwell, there are no statements. Blocks `{ ... }` in classic C-like languages are statements, i.e. they describe a computational step and do not have a value. In Maxwell, blocks also have a return value, depending on the syntax used for their definition. The following possibilities exist:

Implicit return value. In its simplest form, a block in Maxwell looks the same as in C. A list of semicolon-separated expressions enclosed in curly braces. The return value corresponds to the last statement in the block. `{ expr0; expr1; ... exprn; } = exprn;`

Explicit single return value. The block expression may be prefixed by a return variable name which may be assigned a value inside the block body. The variable may be used like any other variable or function argument, and will be returned as the block's value. `r { ... r = exprn; ... } = r;`

Explicit tuple return value. Instead of a single return variable, the block expression may define a tuple of variables that will be returned as value. `(r,s){ ... r = exprn; s = exprm; ... } = (r,s);`

Refer to listing 1.1 for an example of each type of sequential expression.

Listing 1.1: Sequential Expression examples

```
1  // implicit return value
2  // a = 2, Real
3  var a = { var k = 4; sqrt(k); };
4
5  // explicit single return value
6  // b = 2, Real
7  var b = r { r = sqrt(4); };
8  var b = r Real { r = sqrt(4); };
9
10 // explicit tuple return value
11 // c = (2,4), (Real,Real)
12 var c = (r,s) { r = 2; s = r+2; };
13 var c = (r Real, s Real) { r = 2; s = r+2; };
```

1.1 Future Work

It might actually be possible to unify the definition of variables, function arguments and block return variables into one entity.

2 Functions

2.1 Syntax

Functions represent the core concept of the Maxwell programming language. They map a set of input arguments to an output expression. This expression can be anything from a simple arithmetic operation to a complex algorithm. Unlike most members of the C family of languages, functions in Maxwell are anonymous and need to be assigned to a variable or given a name in order to be called. The syntax for a function definition looks as shown in listing Listing 2.1.

Listing 2.1: Function definition syntax

```
-> expr // no arguments
a0 -> expr // one argument
(a1,a2) -> expr // multiple arguments
```

Functions are objects that may either be defined in global scope (as you would do in classic C), or that may be assigned to a variable which may be called later. The two possibilities are outlined in listing Listing 2.2.

Listing 2.2: Giving names to functions

```
square: x -> x*x;
var f = x -> x*x;
// callable as square(...) and f(...)
```

Note that the concept of return types the reader might be familiar with from the C language, arises from Maxwell's sequential expression. By using a sequential expression as the function's expression, any number of return types may be realized. Refer to the example in listing Listing 2.3.

Listing 2.3: Function return values

```
square: x -> x*x
fibonacci: (x,y) -> x+y
invsquare: x -> r { r = x*x; r = 1/r; }
swapinc: (x,y) -> (s,t) { s = y+1; t = x+1; }
```

2.2 Bound Arguments

Maxwell shall support the notion of Bound Arguments. That is, certain arguments of a function may be set to a constant value, thus yielding a new function which lacks those arguments. This concept is similar to Haskell's *currying*. This feature gives rise to contextual function calls (which C++ calls *member function calls*) and encourages software design that relies on callback functions, as well as functional design in general. The concept is illustrated in an example in listing Listing 2.4.

Listing 2.4: Example of function argument binding

```
add: (a Int, b Int) -> a+b;
var increment = add.bind("b", 1);
// add is a mapping (Int, Int) -> (Int)
// increment is now a mapping (Int) -> (Int)

var x = add(1,2); // x = 3
var y = increment(1); // y = 2
```

2.3 Contextual Calls

Consider C++'s member functions. They are functions which have an implicit function argument representing the object they are called on. For example `A a; B b; a.add(b);` is restructured to `A::add(&a,b)`, implicitly passing `a` as the first function argument. A similar concept is possible in Maxwell, yet in a more generic fashion. Typing `var x = a.add(b);` in Maxwell encompasses an argument bind as well as a function call. The expression `a.add` is restructured to `add.bind(0,a)`, thus creating a new function which has its first argument bound to `a`. Of course this expression by itself may already be assigned to a variable or

given a name in order to call it later. Yet in this example the call expression `...(b)` immediately calls the function and returns its result. See listing Listing 2.5 for an example.

Listing 2.5: Contextual call examples

```
// The following are equivalent:
a.add(b);
add(a,b);
add.bind(0,a)(b);

// Callback functions are easily defined:
var cb = a.add;
cb(5); // actually calls add(a,5)
```

3 Syntax

This section describes the Maxwell language syntax in a modified Backus-Naur form that resembles regular expressions, where

: denotes a definition in the grammar, specifying a set of rules for a symbol;

| separates multiple rules in a definition;

"..." denotes a terminal;

(...) groups parts of a rule;

? suffixes a symbol that is optional;

***** suffixes a symbol that may appear any number of times, including never; and

+ suffixes a symbol that must appear at least once.

3.1 Primary Expression

```
primary_expr  : atomic_expr
              | enclosed_expr

atomic_expr   : ident
              | number_literal
              | string_literal

enclosed_expr : "(" expr ")"
              | tuple_expr
              | array_literal
              | set_literal
              | map_literal
```

A *primary expression* groups two types of expressions together: atomic expressions which consist only of one token, and enclosed expressions which consist of multiple tokens between two grouping symbols such as parentheses. This makes primary epxressions very easy to parse, since they either consist of only one token, or start with a symbol that is matched by another symbol at the end.

3.2 Literals

```
number_literal : /[0-9](\.?[0-9\u'])*/
string_literal : /".*(?!\\)"/

array_literal  : "[" expr ("," expr)* "]"
set_literal    : "{" expr ("," expr)* "}"
map_literal    : "{" primary_expr ":" expr ("," primary_expr ":" expr)* "}"
```

Literals represent values that may be statically derived at compile time.

Number literals start with a digit and may contain digits, periods, apostrophes and any letter unicode character. The number format is not part of the syntax, since it may be extended by the user.

String literals start and end with double quotes, and may contain every unicode character. A backslash is used to escape double quotes.

Array literals start and end with brackets, and contain one or more expressions.

Set literals start and end with braces, and contain one or more expressions.

Map literals start and end with braces, and contain one or more key-value-pairs, where the key is a primary expression and the value is any expression.

There is no empty array, set or map literal since the empty set and map would be ambiguous and would collide with the empty block expression.

3.2.1 Examples

```
1234
123'456.984
"hello"
"He said \"Hello\"!"
[1,2,3,"four",true,nil]
{1,2,3,"four",true,nil}
{1: "one", "two": 2, nil: 492}
```

3.3 Definition

```
definition : name ":" expr
```

expr can be any valid expression and is evaluated at compile time until a function, a type, or a constant value is obtained. If this is not possible, the compiler will stop with an error.

A *definition* is an expression with a label assigned to it. The label allows other parts of the code to refer to the definition. Only definitions are exported out of packages and can be referenced from other code. Multiple definitions with the same label are allowed. When other code refers to that label it is the compiler's responsibility to choose the definition that matches the required type.

3.3.1 Examples

```
(a) A: func x -> x*x
(b) B: func x,y -> x*y
(c) C: type Int range 0 to 7
(d) D: 123
(e) E: A(2)
```

Where **(a)** is a function with one argument, **(b)** is a function with two arguments, **(c)** is a type representing integers from 0 to 7, **(d)** is the integer constant 123, and **(e)** is the integer constant 4.

3.4 Function Expression

```
func_expr    : "func" func_variant ("," func_variant)*
func_variant : pattern_expr "->" expr
```

A *function expression* is a mapping from a pattern to a an expression. In terms of other languages, the pattern may be thought of as the arguments and the expression as the return value. A function may have multiple variants, each of which maps different pattern to an expression — much like function overloading in C++. The value of the expression is the resulting function object which is of function type (`A -> B`).

3.4.1 Examples

```
(a) func x -> x*x
(b) func x,y -> x*y
(c) func nil -> 1234
(d) func (x: Int, y: Int) -> x+y
(e) func x -> { x*x; }
(f) func x -> u,v { u = x; v = x*x; }
(g) func Point(x,y) -> x*x+y*y
(h) func 0 -> "zero",
         x -> "anything"
```

Where **(a)** is a function that maps the value x to its square; **(b)** maps the tuple (u,v) to the result of multiplying u and v; **(c)** maps to the constant integer 1234; **(d)** maps the tuple (x,y) to the sum of x and y, where x and y both are integers; **(e)** maps the value x to a block expression that results in the square of x; **(f)** maps the value x to the tuple (u,v), where u is the same as x and v is the square of x; **(g)** maps a `Point`, whose coordinates are captured as x and y, to the sum of its squared components; and **(h)** demonstrates a function with multiple variants.

3.5 Block Expression

```
block_expr : block_ret? "{" (expr ";")* expr ";"? "}"
block_ret  : ident
           | "(" block_arg ("," block_arg)* ")"
block_arg  : ident (":" type_expr)?
```

A *block expression* is a sequence of expressions that are executed in the order that they appear. The expressions are separated by semicolons, the last of which is optional. The block may define one or more return variables that are joined into a tuple and returned as the resulting value of the block. They behave like every other variable inside the block, allowing expressions to assign values to them.

3.5.1 Behaviour

An *empty* block results in *`nil`*. A block *without return variables* returns its very last expression. A block *with one return variable* returns that variable. A block *with multiple return variables* returns a tuple of these variables. Note that in the presence of return variables, the last expression in the block has no significance.

If the block is a function body and contains one or more *return expressions*, its type is expanded to also include types of all return values, forming a union type if necessary.

If the block is a loop body and contains one or more *yield expressions*, its type is expanded to also include the yield type, forming a union type if necessary. The yield type is the array of the union over all yield values.

3.5.2 Examples

```
// implicit return value
(A) {}
(B) { 123 }
(C) { var x = 4; x*2; }

// explicit return value
(D) x { x = 123 }
(E) (x,y) { x = 12; y = 34; }
(F) (s: String) { s = "abc" }

// return and yield
(G) { return "abc"; return nil; 123 }
(H) { yield "abc"; yield nil; 123 }
(I) { yield "abc"; return nil; 123 }
(J) (x,y) { x = 12; y = 34; return "abc"; yield nil; 123 }
```

Where **(a)** returns nil; **(b)** returns the integer 123; **(c)** returns the integer 8; **(d)** returns an integer; **(e)** returns a tuple of integers; **(f)** explicitly states its return type; **(g)** is of type `String|nil|Int`; **(h)** is of type `Array[String|nil]|Int`; **(i)** is of type `Array[String]|nil|Int`; and **(j)** is of type `(Int,Int)|String|Array[nil]|Int`.

```
// concrete example
r := for (f in files) {
    if (f.exists?)
        yield f.size;
    else
        return Error("file %{f} does not exist");
}
```

In this case the block is the body of the for loop. It contains a yield expression with an integer value and a return expression with an Error value. Thus the variable r is of type `Array[Int]|Error`. If all files exist the yield expression appends each file's size to the result of the for loop, assembling an array of integers in the process. However if a file does not exist, the block returns an error which discards all previous yields and aborts the loop.