Maxwell Language Design

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

February 19, 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 for an example of each type of sequential expression.

1.1 Future Work

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

```
// implicit return value
// a = 2, Real
var a = { var k = 4; sqrt(k); };

// explicit single return value
// b = 2, Real
var b = r { r = sqrt(4); };
var b = r Real { r = sqrt(4); };

// explicit tuple return value
// c = (2,4), (Real,Real)
var c = (r,s) { r = 2; s = r+2; };
var c = (r Real, s Real) { r = 2; s =
    r+2; };
```

Listing 1: Sequential Expression examples

2 Functions

Functions represent the core concept of the Maxwell programming language. A function maps a set of input arguments to an arbitrary output expression.

3 Miscellaneous

This section acts as a placeholder for random information.

3.1 Array Literals

An array literal is a constant, static array defined in source code. It is not intended to be modified and supports only a rather simplistic interface:

- get (index Int) -> A
- length -> Int

This corresponds to the *ConstArray* interface, which declares a function to access individual elements, and a function to obtain the overall length of the array.

3.2 Iterators

An iterator is a structure which walks through elements of a collection; i.e. an array, set, map or string. Walking through a linear collection such as

```
type Iterator interface {
  next (iterator * @) -> nil;
  get (iterator @) -> any;
  end (iterator @) -> Bool;
}
```

Listing 2: Iterator interface definition

a vector array is simple, as the elements may be accessed by a continuously increasing index. In case of more complex structures, such as sets or maps, stepping from one element to another is not as simple anymore. Iterators provide a way to abstract this operation, allowing each collection to implement their most efficient iteration technique.

The language shall define an *Iterator* interface as shown in listing 2. The *next* function advances the iterator to the next element, or does nothing if the iterator went past the last element of its collection. The *get* function provides access to the element the iterator is currently pointing at.

Listings