Extend

Language Reference Manual

Ishaan Kolluri, Kevin Ye, Jared Samet, Nigel Schuster

October 20, 2016

Contents

1. Introduction to Extend

Extend is a domain-specific programming language used to designate ranges of cells as reusable functions. It abstracts dependencies between cells and models a dependency graph during compilation. In order to offer great performance for any size of datasets, Extend compiles down to LLVM.

Extend's syntax is meant to provide clear punctuation and easily understandable cell range access specifications, while maintaining the look of modern functional programming languages. Given Extend's functionality resonates well with spreadsheets, it borrows syntactical elements from programs such as Microsoft Excel.

2. Structure of an Extend Program

Extend is predominantly composed of function declarations. In order to run the program, the **main** function will be executed. To illustrate the scope of the language, the OCaml grammar is attached below:

```
program:
    func_decls EOF { List.rev $1 }


func_decls:
    /* nothing */ {[]}
  | func_decls func_decl {$2 :: $1}


func_decl:
    ID LPAREN func_param_list RPAREN LBRACE opt_stmt_list ret_stmt RBRACE
```

```
    { ((1,1), $1, $3, $6, $7) }
  | ret_dim ID LPAREN func_param_list RPAREN LBRACE opt_stmt_list ret_stmt RBRACE
    { ($1,    $2, $4, $7, $8) }

opt_stmt_list:
    /* nothing */ { [] }
  | stmt_list { List.rev $1 }

stmt_list:
    stmt { [$1] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    vardecl { $1 } |  assign { $1 }

ret_stmt:
    RETURN expr SEMI {$2}

vardecl:
    ID varassign SEMI {($1, $2)}
  | dim ID varassign SEMI {($1, $2, $3)}

varassign:
    /* nothing */ {}
  | GETS expr {($2)}

assign:
    ID lhs_sel EQ expr SEMI { ($1,$2,$4) }
```

```
expr:
    ID rhs_sel {($1,$2)}
  | op_expr { $1 }
  | ternary_expr { $1 }
  | switch_expr { $1 }
  | func_expr { $1 }
  | LPAREN expr RPAREN { $2 }
  | LIT_INT { $1 }
  | LIT_FLOAT { $1 }
  | EMPTY { Empty }

op_expr:
    expr PLUS expr { ($1, $3) }
  | expr MINUS expr { ($1, $3) }
  | expr TIMES expr { ($1, $3) }
  | expr DIVIDE expr { ($1, $3) }
  | expr MOD expr { ($1, $3) }

ternary_expr:
  /* commented out optional part for now */
    expr QUESTION expr COLON expr %prec QUESTION { ($1, $3, $5) }

switch_expr:
    SWITCH switch_cond LBRACE case_list RBRACE { ($2, List.rev $4) }

switch_cond:
    /* nothing */ { True }
```

```
  | expr { $1 }


case_list:
    case_stmt { [$1] }
  | case_list case_stmt { $2 :: $1 }


case_stmt:
    DEFAULT COLON expr SEMI { $3 }
  | CASE case_expr_list COLON expr SEMI { (List.rev $2, $4) }


case_expr_list:
    expr { [$1] }
  | case_expr_list COMMA expr { $3 :: $1 }


func_expr:
    ID LPAREN opt_arg_list RPAREN { $3 }


opt_arg_list:
    /* nothing */ {[]}
  | arg_list { List.rev $1 }


arg_list:
    expr {[$1]}
  | arg_list COMMA expr {$3 :: $1}


lhs_sel:
    /* nothing */ { [0,0] }
  | LSQBRACK lslice COMMA lslice RSQBRACK { ($2,$4) }
```

```
  | LSQBRACK lslice RSQBRACK { ($2) }


rhs_sel:
    /* nothing */ { [0,0] }
  | LSQBRACK rslice COMMA rslice RSQBRACK { ($2,$4) }
  | LSQBRACK rslice RSQBRACK { ($2) }


lslice:
    /* nothing */ { 0 }
  | lslice_val { $1 }
  | lslice_val COLON lslice_val { ($1,$3) }


rslice:
    /* nothing */ { 0 }
  | rslice_val { $1 }
  | rslice_val COLON rslice_val { ($1,$3) }


lslice_val:
    expr { $1 }


rslice_val:
    expr { $1 }
  | LSQBRACK expr RSQBRACK { ($2) }


func_param_list:
    /* nothing */ { [] }
  | func_param_int_list { List.rev $1 }
```

```
func_param_int_list:
    func_sin_param { [$1] }
  | func_param_int_list COMMA func_sin_param { $3 :: $1 }

func_sin_param:
    ID { ($1) }
  | dim ID { ($1, $2) }

dim:
    LSQBRACK lslice_val RSQBRACK { $2 }
  | LSQBRACK lslice_val COMMA lslice_val RSQBRACK { ($2,$4) }

ret_dim:
  LSQBRACK ret_sin COMMA ret_sin RSQBRACK { ($2,$4) }

ret_sin:
    LIT_INT { $1 }
  | ID { $1 }
  | UNDERSCORE {}
```

3. Types and Literals

3.1. Primitive Data Types

Extend's basic primitives are *integers*, *floats*, *char* literals, and *string* literals. They are all internally represented as numbers or a range of numbers. They are as follows:

Char

A **char** literal is essentially a size 1 numerical range. At evaluation, the number in the range will be compared with its ASCII equivalent.

String

A **string** literal is a range of numbers of size n, where n is the length of the string. The string 'hello' can be represented internally as [104, 101, 108, 108, 111].

Integer

A **integer** can be represented as a size 1 numerical range as well. However, it retains its numerical value upon evaluation.

Float

A **float**, like Javascript numbers, can be represented as 64 bit, where the fraction is stored in bits 52 to 62.

Below is a snippet illustrating programmatic declarations for each of the above types.

```
/* Integer */
num = 5;
/* Char */
chr = 'A'
/* String */
str = 'Hello'
/* Float */
num = 1.5;
```

3.2. Ranges

Ranges are a data type unique to the Extend language. It borrows conceptually from spreadsheets; a range is a group of cells with dimensions represented as rows and columns. Each range is either one or two-dimensional. A range is composed of cells, and cells are comprised of functions that can have dependencies on the values of other cells. A range is written as follows:

```
/* This is a left-handed range, used to assign a value. */
[1,2]foo; /*Range with 1 row and 2 columns */
```

3.2.1. Range Slicing

Extend somewhat mimics Python in its range slicing syntax; however, it offers the ability to slice a range in both absolute and relative terms.

```
foo[1,2] /* This evaluates to the cell value at row 1, column 2. */
foo[1,] /* Evaluates to the range of cells in row 1. */
foo[,2] /* Evaluates to the range of cells in column 2.*/
foo[,[1]] /* The internal brackets denote RELATIVE notation.
In this case, 1 column right of the one currently being operated on. */
foo[5:, 7:] /* 5th row down, and 7th column from the absolute origin.
foo[[1:2], [5:7]]
/* Selects the rows between the 1st and 2nd row from current row */
/* Selects the columns between 5th and 7th column from current column */
```

3.2.2. The Underscore Symbol

The underscore(_) symbol allows the dimension of the range to have an unspecified size. For example, in function signatures, using the underscore allows the return value to have various possible dimensions. An example is illustrated below:

```
[1,_]foo;
/* A range with 1 row and an unknown, variable number of columns. */
```