Extend

Language Reference Manual

Ishaan Kolluri, Kevin Ye, Jared Samet, Nigel Schuster

October 24, 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:

```
/* Ocamlyacc parser for Extend */

%{
open Ast
%}

%token LSQBRACK RSQBRACK LPAREN RPAREN LBRACE RBRACE HASH
%token COLON COMMA QUESTION GETS ASN SEMI PRECEDES UNDERSCORE
%token SWITCH CASE DEFAULT
%token PLUS MINUS TIMES DIVIDE MOD POWER LSHIFT RSHIFT
%token EQ NOTEQ GT LT GTEQ LTEQ
%token LOGNOT LOGAND LOGOR
%token BITNOT BITXOR BITAND BITOR
%token EMPTY RETURN IMPORT GLOBAL
%token <int> LIT_INT
%token <float> LIT_FLOAT
%token <string> LIT_STRING
%token <string> ID
%token EOF
```

```
%right QUESTION
%left PRECEDES
%left LOGOR
%left LOGAND
%left EQ NOTEQ LT GT LTEQ GTEQ
%left PLUS MINUS BITOR BITXOR
%left TIMES DIVIDE MOD LSHIFT RSHIFT BITAND
%right POWER
%right BITNOT LOGNOT NEG
%left HASH LSQBRACK

%start program
%type <Ast.program> program

%%

program:
    imports globals func_decls EOF { (List.rev $1, List.rev $2, List.rev $3) }

imports:
    /* nothing */ {[]}
  | imports import {$2 :: $1}

import:
    IMPORT LIT_STRING SEMI {$2}

globals:
    /* nothing */ {[]}
  | globals global {$2 :: $1}

global:
    GLOBAL vardecl {$2}

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

func_decl:
    ID LPAREN func_param_list RPAREN LBRACE opt_stmt_list ret_stmt RBRACE
    { {
      name = $1;
      params = $3;
      body = $6;
      ret_val = ((None, None), $7)
    } }
  | ret_dim ID LPAREN func_param_list RPAREN LBRACE opt_stmt_list ret_stmt RBRACE
    { {
      name = $2;
      params = $4;
      body = $7;
      ret_val = ($1, $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:
    var_list SEMI { Vardecl((None, None), List.rev $1) }
  | dim var_list SEMI { Vardecl($1, List.rev $2) }

var_list:
    ID varassign { [ ($1, $2)] }
  | var_list COMMA ID varassign { ($3, $4) :: $1}
```

```
varassign:
    /* nothing */ { None }
  | GETS expr { Some $2 }

assign:
    ID lhs_sel ASN expr SEMI { Assign($1, $2, Some $4) }

expr:
    expr rhs_sel        { Selection($1, $2) }
  | HASH expr           { Selection($2, (None, None)) }
  | op_expr             { $1 }
  | ternary_expr        { $1 }
  | switch_expr         { $1 }
  | func_expr           { $1 }
  | range_expr          { $1 }
  | expr PRECEDES expr  { Precedence($1, $3) }
  | LPAREN expr RPAREN  { $2 }
  | ID                  { Id($1) }
  | LIT_INT             { LitInt($1) }
  | LIT_FLOAT           { LitFlt($1) }
  | LIT_STRING          { LitString($1) }
  | EMPTY               { Empty }

op_expr:
    expr PLUS expr      { BinOp($1, Plus, $3) }
  | expr MINUS expr     { BinOp($1, Minus, $3) }
  | expr TIMES expr     { BinOp($1, Times, $3) }
  | expr DIVIDE expr    { BinOp($1, Divide, $3) }
  | expr MOD expr       { BinOp($1, Mod, $3) }
  | expr POWER expr     { BinOp($1, Pow, $3) }
  | expr LSHIFT expr    { BinOp($1, LShift, $3) }
  | expr RSHIFT expr    { BinOp($1, RShift, $3) }
  | expr LOGAND expr    { BinOp($1, LogAnd, $3) }
  | expr LOGOR expr     { BinOp($1, LogOr, $3) }
  | expr BITXOR expr    { BinOp($1, BitXor, $3) }
  | expr BITAND expr    { BinOp($1, BitAnd, $3) }
  | expr BITOR expr     { BinOp($1, BitOr, $3) }
  | expr EQ expr        { BinOp($1, Eq, $3) }
  | expr NOTEQ expr     { BinOp($1, NotEq, $3) }
  | expr GT expr        { BinOp($1, Gt, $3) }
  | expr LT expr        { BinOp($1, Lt, $3) }
  | expr GTEQ expr      { BinOp($1, GtEq, $3) }
  | expr LTEQ expr      { BinOp($1, LtEq, $3) }
  | MINUS expr %prec NEG  { UnOp(Neg, $2) }
  | LOGNOT expr           { UnOp(LogNot, $2) }
  | BITNOT expr           { UnOp(BitNot, $2) }

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

switch_expr:
    SWITCH LPAREN switch_cond RPAREN LBRACE case_list RBRACE { Switch($3, List.rev $6) }

switch_cond:
    /* nothing */ { None }
  | expr { Some $1 }

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

case_stmt:
    DEFAULT COLON expr SEMI { (None, $3) }
  | CASE case_expr_list COLON expr SEMI { (Some (List.rev $2), $4) }

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

func_expr:
    ID LPAREN opt_arg_list RPAREN { Call($1, $3) }
```

```
range_expr:
    LBRACE row_list RBRACE { LitRange(List.rev $2) }

row_list:
    col_list {[List.rev $1]}
  | row_list SEMI col_list {$3 :: $1}

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

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

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

lhs_sel:
    /* nothing */                         { (None, None) }
  | LSQBRACK lslice RSQBRACK              { (Some $2, None) }
  | LSQBRACK lslice COMMA lslice RSQBRACK { (Some $2, Some $4) }

rhs_sel:
    LSQBRACK rslice RSQBRACK              { (Some $2, None) }
  | LSQBRACK rslice COMMA rslice RSQBRACK { (Some $2, Some $4) }

lslice:
    /* nothing */                         { (None, None) }
  | lslice_val                            { (Some $1, None) }
  | lslice_val COLON lslice_val           { (Some $1, Some $3) }
  | lslice_val COLON                      { (Some $1, Some DimensionEnd) }
  | COLON lslice_val                      { (Some DimensionStart, Some $2) }
  | COLON                                 { (Some DimensionStart, Some DimensionEnd) }

rslice:
    /* nothing */                         { (None, None) }
  | rslice_val                            { (Some $1, None) }
  | rslice_val COLON rslice_val           { (Some $1, Some $3) }
  | rslice_val COLON                      { (Some $1, Some DimensionEnd) }
  | COLON rslice_val                      { (Some DimensionStart, Some $2) }
  | COLON                                 { (Some DimensionStart, Some DimensionEnd) }

lslice_val:
    expr { Abs($1) }

rslice_val:
    expr { Abs($1) }
  | LSQBRACK expr RSQBRACK { Rel($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 { ((None, None), $1) }
  | dim ID { ($1, $2) }

dim:
    LSQBRACK expr RSQBRACK { (Some $2, None) }
  | LSQBRACK expr COMMA expr RSQBRACK { (Some $2, Some $4) }

ret_dim:
    LSQBRACK ret_sin RSQBRACK { ($2, None) }
  | LSQBRACK ret_sin COMMA ret_sin RSQBRACK { ($2,$4) }

ret_sin:
    expr { Some $1 }
  | UNDERSCORE { Some Wild }
```

3. Types and Literals

3.1. Primitive Data Types

Extend has two primitive data types, **numbers** and **ranges**. In the vein of Javascript, numbers are essentially 64-bit floating point numbers in Extend. Ranges are elaborated on in the next section. The user can choose to represent numbers in the following different ways:

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. */
```

4. Expressions

4.1. Operators

4.1.1. Multiplication

4.1.2. Addition

4.1.3. Unary

4.2. Booleans

4.3. Variable Declaration

4.4. Variable Assignment

4.5. Conditionals

5. Functions

Functions lie at Extend's core; however, they are not *first class objects*. Since it can be verbose to write certain operations in Extend, the language will feature a comprehensive number of built-in and standard library function. An important built-in function will be I/O (see section 6).

5.1. Format

Every function in Extend follows the same format, but allows some optional declarations. As in most programming languages the header of the function declares the parameters it accepts and the return type. The simplest function is this:

```
[1,1] foo([1,1] arg) {
  return arg;
}
```

This function simply returns whatever value is passed into it. The leading `[1,1]` marks the return dimensions. `foo` is the function name. In parentheses the function arguments are declared, again with dimensions of the input. The body of the function follows, which in this case is only the return statement.

5.2. Dimension Assignment

Extend will feature gradual typing for function declarations. This will enable users with a weak experience in typing to use the language, but maintains the ability to improve during development.

To avoid specifying the return dimensions, an underscore can be used. This marks a variable range. Thus our function now looks like this:

```
[_,1] foo([5,5] arg1, [1,1] arg2) {
  return arg1[0:arg2 ,0];
}
```

Here we are selecting a range from arg1 that depends on the value of arg2 and can therefore not be known ahead of time.

However Extend will feature even more options to specify ranges. If a certain operation should be applied to a range of numbers of unknown size, the size can be inferred at runtime and match the return size:

```
[m,1] foo([m,1] arg) {
  return arg[0:m-1 ,0] + 1;
}
```

This function will add 1 to each element in arg. Notice, that m is used across the function to apply the operation to the range.

Summarizing, we have 3 ways of specifying a return range:

Type	Symbol Example	Description
Number	3	A number is the simplest descriptor. It specifies the absolute return size
Variable	bar	A variable identifier. To use this, the identifier must also be present as a range descriptor in a function parameter.
Underscore	_	This marker is unique, since it is a wildcard. While the other options aim to be specific, the underscore circumvents declaring the range size.

5.3. Application on Ranges

Extend gives the developer the power to easily apply operations in a functional style on ranges. As outlined in the section above, there are various ways to apply functions to ranges. A feature unique to Extend is the powerful operation on values and ranges. To apply a function on a per cell basis, the corresponding variable needs to be preceded by "#". The following function applies cell wise addition:

```
[m,n] foo([m,n] arg1, [m,n] arg2) {
  [m,n] bar := #arg1 + #arg2;
  return bar;
}
```

If we want to apply a function to the whole range at once we drop the leading symbol. Thus matrix addition takes the following shape:

```
[m,n] foo([m,n] arg1, [m,n] arg2) {
  [m,n] bar := arg1 + arg2;
  return bar;
}
```

While both function above result in the same value, and only show the syntactical difference, more involved functions. If we wanted each cell to to be the square root divided by the sum of the input we have the following:

```
[m] foo([m] arg) {
  [m] bar := sqrt(#arg) / sum(arg);
  return bar;
}
```

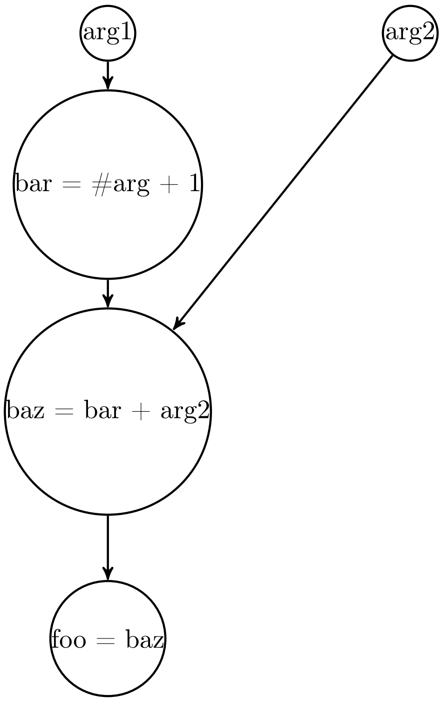
Notice that `arg` is only once preceded by `#`.

5.4. Dependencies Illustrated

The dependency resolution is another asset that sets Extend apart from other languages. Most languages compile ordinarily and execute the given commands sequentially. Extend builds a dependency graph. The advantage of this is that only relevant code segments will be executed. Given the function

```
[m,n] foo([m,n] arg1, [m,n] arg2) {
  [m,n] bar := #arg1 + 1;
  [m,n] faz := #arg1 + 3;
  [m,n] baz := bar + arg2;
  return baz;
}
```

The dependency graph will look like this:



Notice that `faz` does not appear in the graph, because it is not relevant for the return value. Ultimately this graph enables the Extend to find the leaves, evaluate code paths in the best configuration and even in parallel.

6. File I/O

Although the anticipated use cases of Extend generally do not include highly interactive programs, the language has built-in functions that allow the developer to read from and write to files, including standard input, output, and error. These functions are the only part of the language with side effects; as a result, the dependencies between expressions referencing the file I/O functions should be carefully analyzed by the developer to ensure that the program behaves as intended. The precedence operator `->` can be used to create an artificial dependency between expressions to enforce the correct order of evaluation.

6.1. File Pointers

The built-in `fopen` and `fclose` functions open and close file pointers for reading and writing. An attempt to open a nonexistent file, or a file that the user does not have permission to read and write, will result in a runtime error causing the program to halt, as will an attempt to close a file pointer that is not open. The return value of fopen is a range that can be supplied as the file pointer argument to fclose, fread, or fwrite. The return value of fclose is empty. The built-in variables STDIN, STDOUT, and STDERR refer to file pointers that do not need to be opened or closed.

6.2. Reading and Writing

The built-in fread, freadline, and fwrite functions read from and write to an open file pointer. fread takes a maximum number of bytes and a file pointer as arguments and returns a 1-by-n range, where n is the lesser of the number of bytes actually read and the maximum number of bytes requested. If the maximum number of bytes requested is empty, the entire contents of the file are returned. freadline takes a file pointer as argument and returns a 1-by-n range, where n is the number of bytes between the current position of the file pointer and the first newline encountered or EOF, whichever occurs first. The newline, if present, is included in the returned range. The arguments to fwrite are a 1-by-n range and a file pointer and the return value is empty.

6.2.1. Example using the precedence operator

```
bmi() {
  q1 := fwrite("What is your height in inches?\n", STDOUT);
  height := q1 -> parseFloat(freadline(STDIN));
  q2 := height -> fwrite("What is your weight in pounds?\n", STDOUT);
  weight := q2 -> parseFloat(freadline(STDIN));
  return weight * 703 / height ** 2;
}
```

7. Entry Point

7.1. main function

When a compiled Extend program is executed, the main function is evaluated. All computations necessary to calculate the return value of the function are performed, after which the program terminates. If the function declaration includes parameters, the first argument will be a 1-by-n range containing the command line arguments. Any other parameters, if declared, will evaluate to `empty`.

8. Example Program

```
main([1,n] args) {
  seqFP := fopen(args[0]);
  seq1 := freadline(seqFP)[:-2]; // discard newline
  seq2 := seq1 -> freadline(seqFP)[:-2];
  alignment := computeSequenceAlignment(seq1, seq2, 1, -1, -3);
  output := fwrite(alignment[:,0], STDOUT) ->
            fwrite(alignment[:,1], STDOUT) ->
            fclose(seqFP);
  return output;
}

[_,2] computeSequenceAlignment([m,1] seq1, [n,1] seq2,
  matchReward, mismatchPenalty, gapPenalty) {

  [m, n] scoreFromMatch, scoreFromLeft, scoreFromTop;
  [m, n] step, path;
  [1,n] seq2T := transpose(seq2);
  [m+1,n+1] score;

  score[0, 0] = 0;
  score[1:,0] = score[[-1],] + gapPenalty;
  score[0,1:] = score[,[-1]] + gapPenalty;
  score[1:,1:] = nmax(scoreFromMatch[[-1],[-1]],
      nmax(scoreFromLeft[[-1],[-1]], scoreFromTop[[-1],[-1]]));

  scoreFromMatch = #score + ((#seq1 == #seq2T) ? matchReward : mismatchPenalty);
  scoreFromLeft = score[[1],] + gapPenalty;
  scoreFromTop = score[,[1]] + gapPenalty;

  step = (#scoreFromMatch >= #scoreFromLeft) ?
      ((#scoreFromMatch >= #scoreFromTop) ? DDD : TTT) :
      ((#scoreFromLeft  >= #scoreFromTop) ? LLL : TTT);

  path[-1,-1] = 1;
  path[-1,:-1] = (step[,[1]] == LLL && !isEmpty(path[,[1]])) ? 1 + path[,[1]] : empty;
  path[:-1,-1] = (step[[1],] == TTT && !isEmpty(path[[1],])) ? 1 + path[[1],] : empty;
  path[:-1,:-1] = switch () {
    case step[[1],[1]] == DDD && !isEmpty(path[[1],[1]]):
      1 + path[[1],[1]];
    case step[,[1]] == LLL && !isEmpty(path[,[1]]):
      1 + path[,[1]];
    case step[[1],] == TTT && !isEmpty(path[[1],]):
      1 + path[[1],];
  };

  pathLen := path[0,0];
  [m, 1] seq1Positions := pathLen - rmax(path[,:]);
  [1, n] seq2PositionsT := pathLen - rmax(path[:,]);
  [n, 1] seq2Positions := transpose(seq2PositionsT);
  [pathLen, 1] resIdx := colRange(0, pathLength);
  [pathLen, 1] seq1Loc := match(resIdx, seq1Positions);
  [pathLen, 1] seq2Loc := match(resIdx, seq2Positions);

  [pathLength, 2] results;
  results[:,0] = seq1[seq1Loc];
  results[:,1] = seq2[seq2Loc];

  return results;
}
```