The BOL IR

Draft of October 21, 2004

1 Introduction

BOL is a normalized extended \(\lambda\)-calculus that serves as the intermediate representation (IR) of the MOBY compiler. It has a weak, but simple, type system that serves as a guide for optimization and code generation. This report describes the dynamic and static semantics of BOL. It is meant to serve as documentation for MOBY compiler.

The following table summarizes the SML types used to represent BOL types and terms and where they are defined and described:

<table>
<thead>
<tr>
<th>Type</th>
<th>Module</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
<td>BOL</td>
<td>BOL variables</td>
<td>5</td>
</tr>
<tr>
<td>exp</td>
<td>BOL</td>
<td>BOL expressions labeled by program points</td>
<td>4.1</td>
</tr>
<tr>
<td>term</td>
<td>BOL</td>
<td>unlabeled expressions</td>
<td>4.1</td>
</tr>
<tr>
<td>lambda</td>
<td>BOL</td>
<td>named function definition</td>
<td>4.2</td>
</tr>
<tr>
<td>rhs</td>
<td>BOL</td>
<td>right-hand-side of binding</td>
<td>4.3</td>
</tr>
<tr>
<td>primop</td>
<td>PrimOps</td>
<td>primitive operations</td>
<td>6</td>
</tr>
<tr>
<td>ppt</td>
<td>ProgPt</td>
<td>program points</td>
<td></td>
</tr>
<tr>
<td>kind</td>
<td>BOLTYPES</td>
<td>kind of a BOL type</td>
<td>2.1</td>
</tr>
<tr>
<td>ty</td>
<td>BOLTYPES</td>
<td>BOL type</td>
<td>2.2 – 2.5</td>
</tr>
<tr>
<td>field</td>
<td>BOLTYPES</td>
<td>field descriptor for BOL struct</td>
<td>2</td>
</tr>
<tr>
<td>c_prototype</td>
<td>BOLTYPES</td>
<td>type of C function</td>
<td>2</td>
</tr>
</tbody>
</table>
2 BOL types

2.1 Kinds

The BOL types are organized into a hierarchy by kind; there are four distinct kinds of BOL types:

1. Word kind (W) types are those that can be stored in a general-purpose machine register on the host processor.
2. Variable kind types (V) are those that can be assigned to a BOL variable.
3. Memory kind types (M) are those types that describe the layout of memory.
4. Type kind types (Type) include all types.

We use $\text{Kind} = \{W, V, M, \text{Type}\}$ for the set of kinds and $\kappa \in \text{Kind}$. The kinds are ordered under set inclusion as follows:

$$W \subset V \subset M \subset \text{Type}$$

i.e., a type of kind $W$ also has kinds $V$, $M$, and $\text{Type}$. A kind environment $KE : (\text{Base} \cup \text{TyVar}) \mapsto \text{Kind}$ maps base types and type variables to kinds. The mapping of base types is architecture (and compiler) specific. For example, the type of 64-bit integers ($\text{long}$) has kind $W$ on 64-bit machines, but kind $V$ on 32-bit machines.

2.2 Kind W types

The following types have $W$ kind and may be mapped to a general-purpose register:

$T_{\text{Any}}$

a word-sized value of unknown type; we use the syntax any to denote this type.

$T_{\text{Bool}}$

a boolean; we use the syntax bool to denote this type.

$T_{\text{Enum}} \{\text{lo} : \text{word}, \text{hi} : \text{word}\}$

a small integer (16-bit) in the range $[\text{lo}, \text{hi}]$. When $\text{lo}$ is equal to $\text{hi}$, then the type is a singleton type. We write $(\text{lo..hi})$ to denote the type $T_{\text{Enum}}\{\text{lo}, \text{hi}\}$.

$T_{\text{Integer}}$

arbitrary precision integers (represented by a pointer); we use the syntax integer to denote this type.
2.3 Kind W or V types

There are five numeric types whose representation (i.e., kind) depends on the target architecture and compiler configuration. These types have either W kind, when they can be mapped into general-purpose registers or V kind when they cannot be so mapped. The types are:
T_Int
32-bit 2’s complement integers; we use int to denote this type.

T_Long
64-bit 2’s complement integers; we use long to denote this type.

T_Float
32-bit IEEE single-precision floating-point numbers; we use float to denote this type.

T_Double
64-bit IEEE double-precision floating-point numbers; we use double to denote this type.

T_Extended
IEEE extended double-precision floating-point numbers; we use extended to denote this type.

2.4 Kind M types

T_Data
a region of memory of unknown size.

T_Object
a region of memory used to represent a MOBY object.

T_Struct of {sz : int, align : int, data : field list}
a region of memory with a known size, alignment, and layout.

T_Vector of {len : int option, elemSz : int, ty : ty}
an immutable vector of elements with the given size and type. When the len field is note NONE, then the length of the vector is known.

T_Array of {len : int option, elemSz : int, ty : ty}
a mutable array of elements with the given size and type. When the len field is note NONE, then the length of the vector is known.

T_Union of ty list
an untagged union of types.

T_TaggedUnion of (int * ty) list
2.5  Kind Type types

T_Void
This type is used to denote the C void type in function prototypes.

3  Representation of Moby types

This section describes how common MOBY types are mapped to BOL types. It is always the case that the BOL type corresponding to a MOBY type will have W kind.

Bool  is represented by the bool type.
Char  is represented by (0..255).
Int   is either represented by wrap(int) or by int.
Long  is either represented by wrap(long) or by long.
Integer is represented by integer.
Float is represented by wrap(float).
Double is represented by wrap(double).
Extended is represented by wrap(extended).

3.1  Sequence types

MOBY sequence types, such as Array and String, have a two-level representation in BOL. There is a two-word header consisting of a 32-bit integer length and a pointer to the data object. 1

3.2  The List type constructor

The MOBY List type constructor is defined as

datatype List(t) { Nil, Cons of (t, List(t)) }  

The Nil value is represented by the value 0, while the Cons values are represented by pointers to two-word pairs. The BOL type for this representation is *\tau + (0.0), where \tau is the type of the list elements (any when the type is unknown).

1On 64-bit machines, there is 32-bits of padding between the length and the data pointer to ensure 64-bit alignment of the data pointer.
4 The BOL representation

Inside the MOBY compiler, BOL expressions are represented using the following datatypes:

\[
\text{datatype exp} = \text{E_Pt of (ProgPt.ppt * term)} \\
\text{and term} = \ldots \\
\text{and rhs} = \ldots
\]

The `exp` type is a term tagged with a unique program point. Program points serve as labels for those analyses that need to track positions in the code. The `rhs` (right-hand-side) type covers terms that cannot appear in a tail context.

4.1 Expression forms

The `term` type has a number of constructors; we call these expression forms (ignoring the lack of a program-point label).

`E_Let of (var list * exp * exp)`

binds the variables to the results of the first expression in the scope of the second expression. The general syntax of this form is

```
let (x₁, ..., xₙ) = e₁; e₂
```

When the number of bound variables is one, we write

```
let x = e₁; e₂
```

and when there are no bound variables, we write

```
do e₁; e₂
```

`E_Stmt of (var list * rhs * exp)`

binds the variables to the results of the right-hand-side in the scope of the expression. The syntax of this form is the same as for `E_Let`.

`E_StackAlloc of (var * int * int * exp)`

binds the variable to reserved space in the stack frame. The first integer specifies the size (in bytes) of the space and the second specifies the alignment. The scope of the binding and the extent of the reserved space is the expression. The syntax for this form is

```
stackalloc x = <sz, align>; e₂
```

`E_Fun of (lambda list * exp)`

binds a collection of mutually recursive function definitions. The scope of the function names includes both the function bodies and the expression. We use the syntax
fun \( f_1 (x_{1,1}, \ldots, x_{1,n_1}) = e_1 \)
\[ \text{and} \ldots \]
\[ \text{and} \quad f_k (x_{k,1}, \ldots, x_{k,n_k}) = e_k; \]
\[ e \]
for the term
\[ \text{E}_\text{Fun}([([f_1, [x_{1,1}, \ldots, x_{1,n_1}], e_1], \ldots, ([f_k, [x_{k,1}, \ldots, x_{k,n_k}], e_k])], e) \]

\text{E}_\text{Cont of (lambda * exp)}
Binds a BOL continuation with the expression as its scope. Note that the lifetime of the
continuation is also its scope!

\text{E}_\text{If of (var * exp * exp)}
tests the variable and if it is true, the evaluate the first expression, otherwise evaluate the
second expression. The syntax for this form is
\[ \text{if } x \text{ then } e_1 \text{ else } e_2 \]

\text{E}_\text{Switch of (var * (int * exp) list * exp option)}
Tests the variable against the integer labels of the list of cases; the third argument is the
optional default case. The cases should be in increasing numeric order and the default case
should be present unless the variable is guaranteed to always have one of the case labels as its
values. We use the syntax
\[ \text{switch } x \{ \text{ case } i_1: e_1 \ldots \text{ case } i_n: e_n \} \]
for the term
\[ \text{E}_\text{Switch}(x, [([i_1, e_1], \ldots, (i_n, e_n])], \text{NONE}) \]
and
\[ \text{switch } x \{ \text{ case } i_1: e_1 \ldots \text{ case } i_n: e_n \text{ default: } e \} \]
for the term
\[ \text{E}_\text{Switch}(x, [([i_1, e_1], \ldots, (i_n, e_n])], \text{SOME}(e)) \]

\text{E}_\text{Apply of (var * var list)}
applies the function named by the first variable to the arguments named by the list of variables.
We use the syntax \textit{call } f (\textit{args}) \textit{ for } E\text{Apply} (f, \textit{args}).

\text{E}_\text{Throw of (var * var list)}
applies the continuation named by the first variable to the arguments named by the list of
variables. We use the syntax \textit{throw } k (\textit{args}) \textit{ for } E\text{Throw} (k, \textit{args}).

\text{E}_\text{Ret of var list}
returns the values bound to the variables. Note that the term “\textit{return}” does not connote
control-flow.
4.2 Lambda abstractions

The type `lambda` is used to represent both functions and continuations. It is defined as:

```
type lambda = (var * var list * exp)
```

where the first variable is the name of the function (there are no anonymous functions in BOL), the list of variables are the formal parameters, and the expression is the function body.

4.3 Right-hand-side forms

- **E_Cast** of `(var * BOLTypes.ty)`
  
  cast the value bound to the variable to the given type (which must have the same kind). We use the notation `(τ) x` for `E_Cast(x, τ)`

- **E_Select** of `(int * var)`
  
  selects the the specified field from the record bound to the variable. We use the notation `x#i` for `E_Select(i, x)`.

- **E_Update** of `(int * var * var)`
  
  updates the specified field from the record bound to the first variable with the value bound to the second variable. This form has no results (i.e., zero-arity). We use the notation `x#i := y` for `E_Update(i, x, y)`.

- **E_Alloc** of `(BOLTypes.ty * var list)`
  
  allocates and initializes a record in the heap. The type specifies the record’s layout and the list of variables provide the initial values for record’s fields.

- **E_AllocObj** of `(BOLTypes.ty * var)`
  
  allocate memory for an object. The type specifies the layout of the object’s fields and the variable is bound to the method suite.

- **E_Wrap** of `var`
  
  wrap (box) the value bound to the variable. We use the syntax `wrap(x)` for `E_Wrap(x)`.

- **E_Unwrap** of `var`
  
  unwrap (unbox) the boxed value bound to the variable. We use the syntax `unwrap(x)` for `E_Unwrap(x)`.

- **E_IConst** of `IntInf.int`
  
  an integer constant.
E_SConst of string
   a string constant. Note that this is the string data and not the representation of a MOBY string literal.

E_FConst of FloatLit.float
   a floating-point constant.

E_BConst of bool
   a boolean constant.

E_StaticAddr of var
   the address of the static location named by the variable.

E_StaticRef of var
   the contents of the static location named by the variable.

E_Prim of var primop
   applies a primitive operator to its arguments. The primitive operators are described in Section 6.

E_Slot of slot_exp

E_DictFieldSel of (var * member_label)

E_DictMethSel of (var * member_label)

E_FieldGet of (var * var)

E_FieldPut of (var * var * var)

E_MethGet of (var * var)

E_ApplyCont of (var * var list)
   Partially apply a continuation to its arguments (but do not transfer control). This operation has the effect of turning a continuation with arguments into one without.

E_ThdCreate of var
E_ThdGetTask

E_ThdGetId of var

E_ThdLockSelf of var

E_ThdEnqueue of (var * var * var)

E_ThdEnqueueSelf of (var * var)

E_ThdDequeue of var

E_ThdTerminate of var

E_CCall of (var * var list)
    calls the C function named by the first variable on the arguments named by the variable list. We use the syntax \texttt{ccall} \texttt{f(args)} for \texttt{E_CCall(f, args)}.

4.4 Creating BOL expressions

The BOL module provides constructor functions for the various expression forms (e.g., \texttt{mkLet} to create an \texttt{E_Let} expression form). These constructor functions take care of labeling the term with a unique program point. The Census module provides similar functions, except that they maintain the additional invariants defined by the census, such as variable binding information.

5 BOL variables

The representation of BOL variables has the SML type \texttt{var}, which is defined in the BOL module as follows:


```haskell
datatype var = V of {
    id : Word.word,
    name : string option,
    src : Var.var option,
    binding : var_binding ref,
    ty : BOLTypes.ty,
    useCnt : int ref,
    props : PropList.holder
}
```

The fields of this representation are used as follows:

- **id**: a unique ID that can be used for identity testing, ordering, or hashing.
- **name**: if present, a symbolic name for the variable.
- **src**: if present, then this BOL variable corresponds to the specified typed AST variable.
- **binding**: the binding that defines this variable.
- **ty**: this variable’s type.
- **useCnt**: the number of times that this variable is used. For functions and continuations, this count includes applications.
- **props**: a holder for name/value pairs (i.e., an association list).

## 6 Primitive operators

Machine-level operations are represented in BOL as “primops” (primitive operations). The `primop` datatype is defined in the `PrimOps` structure. This datatype is type constructor over the type used to represent the primop arguments; the BOL uses this type constructor applied to the `var` type. To ease the addition of new primitive operations, we generate the definition of the `primop` datatype and the various modules that directly work on it (e.g., constant folding, effect analysis, code generation, etc.) from a specification file. The primitive operations can be grouped into the following classes:

- **Boolean operations**: The boolean type serves as the result of conditionals and as the argument of conditionals. There is one operation — logical negation.

- **Integer operations**: There are two fixed-precision integer types in BOL: 32-bit and 64-bit. Each of these types has a complete set of arithmetic and comparison operations; the former are prefixed by “I32,” while the latter are prefixed by “I64.” In addition, there are unsigned comparisons on 32-bit integers (prefixed by “U32”).
Floating-point operations  There are three floating-point types: IEEE 32-bit single-precision numbers, IEEE 64-bit double-precision numbers, and IEEE extended-double-precision numbers. The size of the latter type depends on the target architecture; it is 80-bits on the Intel IA32 (a.k.a. x86) and 64-bits on the PowerPC. Each of these types has a complete set of arithmetic and comparison operations that follow the IEEE semantics. In addition, there are two multiply accumulate instructions that can produce non-IEEE results.

String operations  BOL provides operations for comparison of string data values. Since these values do not have length information (see Section 3), they take a first argument which is a limit on the number of characters to compare.

Pointer testing operations  The translation of higher-level datatypes (e.g., lists) uses the distinction between pointers and small integers (integers in the range $[0, 2^{16} - 1]$) to distinguish between different constructors. In this case, we call the pointer a boxed value and the small integer a unboxed value. BOL provides operations to test for boxed and unboxed values.

Address arithmetic  BOL has a full complement of address arithmetic operations. These are used to support data-level interoperability with foreign code and data structures.

Conversion operations  BOL has conversion operators between the various numeric types. In addition, it has operations to cast between integer and floating-point representations (e.g., to allow one to examine the bits of a floating-point number directly.

Synchronization operations  BOL includes low-level synchronization operations to support spin locks and the like.

The following is a list of the BOL primitive operations with their types and a short description of each operator:

- **BNot**: `Bool` → `Bool`
  - Boolean negation.
- **I32Neg**: `Int` → `Int`
  - 32-bit 2’s complement negation.
- **I32Add**: `(Int, Int)` → `Int`
  - 32-bit 2’s complement addition.
- **I32Sub**: `(Int, Int)` → `Int`
  - 32-bit 2’s complement subtraction.
- **I32Mul**: `(Int, Int)` → `Int`
  - 32-bit 2’s complement multiplication.
- **I32Div**: `(Int, Int)` → `Int`
  - 32-bit 2’s complement division.
- **I32Mod**: `(Int, Int)` → `Int`
  - 32-bit 2’s complement remainder.
I32Not : Int -> Int
32-bit 1’s complement negation.
I32And : (Int, Int) -> Int
32-bit logical and.
I32Or : (Int, Int) -> Int
32-bit logical or.
I32XOr : (Int, Int) -> Int
32-bit logical xor.
I32LSh : (Int, Int) -> Int
32-bit left-shift.
I32RShA : (Int, Int) -> Int
32-bit arithmetic right-shift.
I32RShL : (Int, Int) -> Int
32-bit logical right-shift.
I32Lt : (Int, Int) -> Bool
32-bit 2’s complement less-than comparison.
I32Lte : (Int, Int) -> Bool
32-bit 2’s complement less-than or equal comparison.
I32Gt : (Int, Int) -> Bool
32-bit 2’s complement greater comparison.
I32Gte : (Int, Int) -> Bool
32-bit 2’s complement greater-than or equal comparison.
I64Eq : (Int, Int) -> Bool
64-bit equal test.
I64NEq : (Int, Int) -> Bool
64-bit not-equal test.

U32Lt : (Int, Int) -> Bool
32-bit unsigned less-than comparison.
U32Lte : (Int, Int) -> Bool
32-bit unsigned less-than or equal comparison.
U32Gt : (Int, Int) -> Bool
32-bit unsigned greater comparison.
U32Gte : (Int, Int) -> Bool
32-bit unsigned greater-than or equal comparison.

I64Neg : Int -> Int
64-bit 2’s complement negation.
I64Add : (Int, Int) -> Int
64-bit 2’s complement addition.
I64Sub : (Int, Int) -> Int
64-bit 2’s complement subtraction.
I64Mul : (Int, Int) -> Int
64-bit 2’s complement multiplication.
I64Div : (Int, Int) -> Int
64-bit 2's complement division.

$I64Mod : (\text{Int}, \text{Int}) \rightarrow \text{Int}$
64-bit 2's complement remainder.

$I64Not : \text{Int} \rightarrow \text{Int}$
64-bit 1's complement negation.

$I64And : (\text{Int}, \text{Int}) \rightarrow \text{Int}$
64-bit logical and.

$I64Or : (\text{Int}, \text{Int}) \rightarrow \text{Int}$
64-bit logical or.

$I64XOr : (\text{Int}, \text{Int}) \rightarrow \text{Int}$
64-bit logical xor.

$I64LSh : (\text{Int}, \text{Int}) \rightarrow \text{Int}$
64-bit left-shift.

$I64RShA : (\text{Int}, \text{Int}) \rightarrow \text{Int}$
64-bit arithmetic right-shift

$I64RShL : (\text{Int}, \text{Int}) \rightarrow \text{Int}$
64-bit logical right-shift.

$I64Lt : (\text{Int}, \text{Int}) \rightarrow \text{Bool}$
64-bit 2's complement less-than comparison.

$I64Lte : (\text{Int}, \text{Int}) \rightarrow \text{Bool}$
64-bit 2's complement less-than or equal comparison.

$I64Gt : (\text{Int}, \text{Int}) \rightarrow \text{Bool}$
64-bit 2's complement greater comparison.

$I64Gte : (\text{Int}, \text{Int}) \rightarrow \text{Bool}$
64-bit 2's complement greater-than or equal comparison.

$I64Eq : (\text{Int}, \text{Int}) \rightarrow \text{Bool}$
64-bit equal test.

$I64NEq : (\text{Int}, \text{Int}) \rightarrow \text{Bool}$
64-bit not-equal test.

$F32Neg : \text{Float} \rightarrow \text{Float}$
32-bit IEEE floating-point negation

$F32Add : (\text{Float}, \text{Float}) \rightarrow \text{Float}$
32-bit IEEE floating-point addition

$F32Sub : (\text{Float}, \text{Float}) \rightarrow \text{Float}$
32-bit IEEE floating-point subtraction

$F32Mul : (\text{Float}, \text{Float}) \rightarrow \text{Float}$
32-bit IEEE floating-point multiplication

$F32Div : (\text{Float}, \text{Float}) \rightarrow \text{Float}$
32-bit IEEE floating-point division

$F32Rem : (\text{Float}, \text{Float}) \rightarrow \text{Float}$
32-bit IEEE floating-point remainder

$F32MAdd : (\text{Float}, \text{Float}, \text{Float}) \rightarrow \text{Float}$
32-bit floating-point multiply/add

$F32MSub : (\text{Float}, \text{Float}, \text{Float}) \rightarrow \text{Float}$
32-bit floating-point multiply/subtract

F32Abs : Float -> Float

32-bit IEEE floating-point absolute value

F32CopySign : (Float, Float) -> Float

32-bit IEEE floating-point copy-sign

F32Sqrt : Float -> Float

32-bit IEEE floating-point square root

F32Pow : (Float, Float) -> Float

F32Lt : (Float, Float) -> Bool

32-bit IEEE floating-point less-than comparison.

F32Lte : (Float, Float) -> Bool

32-bit IEEE floating-point less-than or equal comparison.

F32Gt : (Float, Float) -> Bool

32-bit IEEE floating-point greater-than comparison.

F32Gte : (Float, Float) -> Bool

32-bit IEEE floating-point greater-than or equal comparison.

F32Eq : (Float, Float) -> Bool

32-bit IEEE floating-point inequality test.

F32NEq : (Float, Float) -> Bool

32-bit IEEE floating-point equality test.

F32LtGt : (Float, Float) -> Bool

F32ULt : (Float, Float) -> Bool

F32ULte : (Float, Float) -> Bool

F32UGt : (Float, Float) -> Bool

F32UGte : (Float, Float) -> Bool

F32Ordered : (Float, Float) -> Bool

32-bit IEEE floating-point ordered test.

F32Unordered : (Float, Float) -> Bool

32-bit IEEE floating-point unordered test.

F32Finite : Float -> Bool

test for 32-bit IEEE finite number

F32Infinite : Float -> Bool

test for 32-bit IEEE infinite number

F64Neg : Double -> Double

64-bit IEEE floating-point negation

F64Add : (Double, Double) -> Double

64-bit IEEE floating-point addition

F64Sub : (Double, Double) -> Double
64-bit IEEE floating-point subtraction
F64Mul : (Double, Double) -> Double
64-bit IEEE floating-point multiplication
F64Div : (Double, Double) -> Double
64-bit IEEE floating-point division
F64Rem : (Double, Double) -> Double
64-bit IEEE floating-point remainder
F64MAdd : (Double, Double, Double) -> Double
64-bit floating-point multiply/add
F64MSub : (Double, Double, Double) -> Double
64-bit floating-point multiply/subtract
F64Abs : Double -> Double
64-bit IEEE floating-point absolute value
F64CopySign : (Double, Double) -> Double
64-bit IEEE floating-point copy-sign
F64Sqrt : Double -> Double
64-bit IEEE floating-point square root
F64Pow : (Double, Double) -> Double
F64Lt : (Double, Double) -> Bool
64-bit IEEE floating-point less-than comparison.
F64Lte : (Double, Double) -> Bool
64-bit IEEE floating-point less-than or equal comparison.
F64Gt : (Double, Double) -> Bool
64-bit IEEE floating-point greater-than comparison.
F64Gte : (Double, Double) -> Bool
64-bit IEEE floating-point greater-than or equal comparison.
F64Eq : (Double, Double) -> Bool
64-bit IEEE floating-point inequality test.
F64NEq : (Double, Double) -> Bool
64-bit IEEE floating-point equality test.
F64LtGt : (Double, Double) -> Bool
F64ULt : (Double, Double) -> Bool
F64ULte : (Double, Double) -> Bool
F64UGt : (Double, Double) -> Bool
F64UGte : (Double, Double) -> Bool
F64Ordered : (Double, Double) -> Bool
64-bit IEEE floating-point ordered test.
F64Unordered : (Double, Double) -> Bool
64-bit IEEE floating-point unordered test.
F64Finite : Double -> Bool
    test for 64-bit IEEE finite number
F64Infinite : Double -> Bool
    test for 64-bit IEEE infinite number
FXNeg : Extended -> Extended
FXAdd : (Extended, Extended) -> Extended
FXSub : (Extended, Extended) -> Extended
FXMul : (Extended, Extended) -> Extended
FXDiv : (Extended, Extended) -> Extended
FXRem : (Extended, Extended) -> Extended
FXMAdd : (Extended, Extended, Extended) -> Extended
FXMSub : (Extended, Extended, Extended) -> Extended
FXAbs : Extended -> Extended
FXCopySign : (Extended, Extended) -> Extended
FXSqrt : Extended -> Extended
FXPow : (Extended, Extended) -> Extended
FXLt : (Extended, Extended) -> Bool
FXLte : (Extended, Extended) -> Bool
FXGt : (Extended, Extended) -> Bool
FXGte : (Extended, Extended) -> Bool
FXEq : (Extended, Extended) -> Bool
FXNEq : (Extended, Extended) -> Bool
FXLtGt : (Extended, Extended) -> Bool
FXULt : (Extended, Extended) -> Bool
FXULte : (Extended, Extended) -> Bool
FXUGt : (Extended, Extended) -> Bool
FXUGte : (Extended, Extended) -> Bool
FXOrdered : (Extended, Extended) -> Bool
FXUnordered : (Extended, Extended) -> Bool
FXFinite : Extended -> Bool
FXInfinite : Extended -> Bool

StrEq : (Int, String, String) -> Bool
test two strings for equality
StrNEq : (Int, String, String) -> Bool
test two strings for inequality
StrCmp : (Int, String, String) -> Int
compare two strings for order

Boxed : Any -> Bool
test for boxed values
Unboxed : Any -> Bool
test for unboxed values

AdrEq : (Addr, Addr) -> Bool
test addresses for equality
AdrNEq : (Addr, Addr) -> Bool
test addresses for inequality
AdrAdd : (Addr, Int) -> Addr
add an integer to an address
AdrSub : (Addr, Int) -> Addr
subtract an integer from an address
AdrAdd4 : (Addr, Int) -> Addr
add a scaled (by 4) integer to an address
AdrSub4 : (Addr, Int) -> Addr
subtract a scaled (by 4) integer from an address
AdrAdd8 : (Addr, Int) -> Addr
add a scaled (by 8) integer to an address
AdrSub8 : (Addr, Int) -> Addr
subtract a scaled (by 8) integer from an address
AdrLoadI8 : Addr -> Int
load a sign-extended 8-bit integer from memory
AdrStoreI8 : (Addr, Int) → ()
store an 8-bit integer

AdrLoadI16 : Addr → Int
load a sign-extended 16-bit integer from memory

AdrStoreI16 : (Addr, Int) → ()
store a 16-bit integer

AdrLoadI32 : Addr → Int
load a 32-bit integer from memory

AdrStoreI32 : (Addr, Int) → ()
store a 32-bit integer

AdrLoadI64 : Addr → Long
load a 64-bit integer from memory

AdrStoreI64 : (Addr, Long) → ()
store a 64-bit integer

AdrLoadF32 : Addr → Float
load a 32-bit floating-point number from memory

AdrStoreF32 : (Addr, Float) → ()
store a 32-bit floating-point number

AdrLoadF64 : Addr → Double
load a 64-bit floating-point number from memory

AdrStoreF64 : (Addr, Double) → ()
store a 64-bit floating-point number

AdrLoadFX : Addr → Extended
load an extended-precision floating-point number from memory

AdrStoreFX : (Addr, Extended) → ()
store a extended-precision floating-point number

AdrLoadP : Addr → Addr
load an address from memory

AdrStoreP : (Addr, Ptr) → ()
store an address

AdrLoadU8 : Addr → Int
load an unsigned 8-bit integer from memory

AdrLoadU16 : Addr → Int
load an unsigned 16-bit integer from memory

AdrLoad : Addr → Any
load a word from memory

AdrStore : (Addr, Any) → ()
store a word

CvtI32ToI64 : Int → Long
zero-extend a 32-bit integer to a 64-bit integer.

CvtxI32ToI64 : Int → Long
sign-extend a 32-bit integer to a 64-bit integer.

CvtI32ToF32 : Int → Float
convert a 32-bit integer to a 32-bit floating-point number.
CvtI32ToF64 : Int -> Double
convert a 32-bit integer to a 64-bit floating-point number.

CvtI32ToFX : Int -> Extended
convert a 32-bit integer to an extended-precision floating-point number.

CastF32ToI32 : Int -> Float
cast a 32-bit floating-point number to a 32-bit integer.

CastI32ToF32 : Int -> Float
cast a 32-bit integer to a 32-bit floating-point number.

CastF64ToI64 : Double -> Long
cast a 64-bit floating-point number to a 64-bit integer.

CastI64ToF64 : Double -> Long
cast a 64-bit integer to a 64-bit floating-point number.

CvtF32ToF64 : Float -> Double
convert a 32-bit floating-point number to a 64-bit floating-point number.

CvtF32ToFX : Float -> Extended
convert a 32-bit floating-point number to an extended-precision floating-point number.

CvtF64ToFX : Double -> Extended
convert a 64-bit floating-point number to an extended-precision floating-point number.

I32CmpAndSwap : (Addr, Int, Int) -> Bool, Int

I64CmpAndSwap : (Addr, Long, Long) -> Bool, Long