# FOAM

Pete Huerter
Stephen Watt
Department of Computer Science
The University of Western Ontario
huerter0@csd.uwo.ca
watt@csd.uwo.ca

January 18, 2001

# FOAM

Pete Huerter (huerter0@csd.uwo.ca) Stephen Watt (watt@csd.uwo.ca)

## 1 Introduction

...

Aldor has many interesting features.

- Types are first class values. Aldor is statically typed, but types are values. Programs may be parameterized by types provided at execution time.
  - How are domains represented at the FOAM level?
  - How are categories represented at the FOAM level?
- *Post facto extensions* to libraries.
- Functions are first class values. Aldor provides mechanisms for composing and manipulating functions in usefull ways.
- Dependent types. A Dependent type is a type T in which the type of one subexpression of T depends on the value of another.
- *Control abstraction* via generators and type-specific tests.

It helps, in understanding the FOAM, to consider how these features are represented at a lower level. Such features require a unique and powerfull IR. Indeed, many of the characteristics of the Aldor language are visible in its FOAM representation. The FOAM is an HIR, or *high-level intermediate representation.* The functionality of its instruction set mirrors that of *Aldor.*

In the pages to follow, we will stip away the syntactic sugar of Aldor revealing its core, the FOAM.

...

# 2 Terminology

**lexical binding**
A binding in a lexical environment.

**lexical closure**
A function that, when invoked on arguments, executes the body of a lambda expression in the lexical environment that was captured at the time of the creation of the lexical closure, augmented by bindings of the function's parameters to the corresponding arguments.

**lexical environment**
That part of the environment that contains bindings whose names have lexical scope. A lexical environment contains, among other things: ordinary bindings of variable names to values, lexically established bindings of function names to functions, macros, symbol macros, blocks, tags, and local declarations (see declare).

**lexical scope**
Scope that is limited to a spatial or textual region within the establishing object. Implies that the only variables visible at a given point in a program are those that have been created locally or imported into scopes surrounding the current point.

**lexical variable**
A variable the binding for which is in the lexical environment.

**environment**
Environments are essentially dictionaries that tie names to values. In Aldor, an environment is called a *domain*. Domains can be created and manipulated dynamically in Aldor. An environment describes the denotation or meaning of an arbitrary term.

**closure**
Functions may depend on externally defined values. In the FOAM, functions are first class values. A closure dynamically captures the creation environment of a new function value.

When a function expression (anonymous function, lambda expression) is evaluated, it captures the lexical environment in which it appears, creating a lexical closure. The values of the variables which are visible in the scope of the function expression are then avaiable when it is eventually applied to a set of arguments. Function expressions evaluate to functions.

$$(s1 : S1, ..., sn : Sn) : T == E \text{ is shorthand for}$$
$$f : (s1 : S1, ..., sn : Sn) - > T == (s1 : Sn, ..., sn : Sn) : T + - > E$$

**fluid variables**
A fluid variable exists throughout the lifetime of a program, and its value is always the most recent extant binding of the variable.

A fluid declaration declares that the given identifiers should be treated as having dynamic, as opposed to lexical scope. The declaration is enforced within the lexical scope containing the declaration.

See rules in Aldor User Guide (pg. 115...)

**protocol**
A Protocol is used to describe the interface through which an object should be called or accessed.

**environment**
A set of bindings.

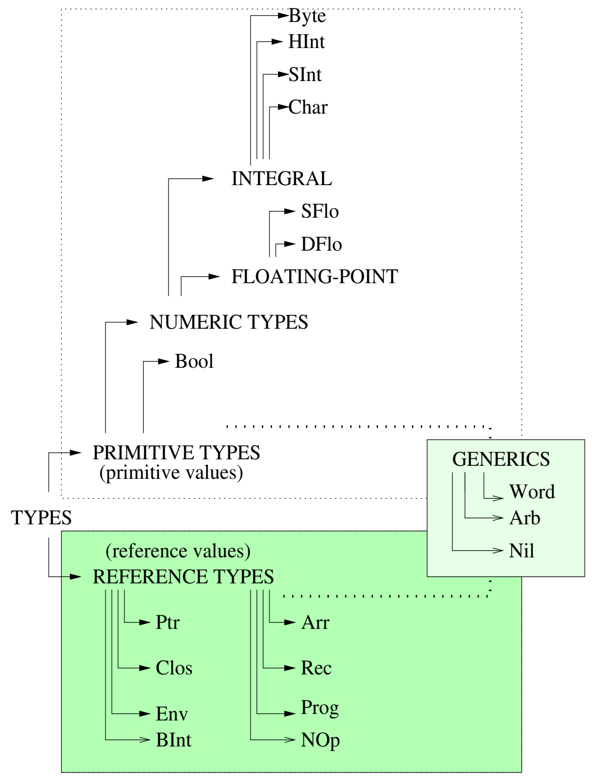
**environment object**
An object representing a set of lexical bindings, used in the processing of a form to provide meanings for names within that form.

**binding**
An association between a name and that which the name denotes. "A lexical binding is a lexical association between a name and its value."

## 3 Data Types, Values, Built-in Operations

A *data type* declares or restricts the domain of an object declared of that type to the set of *values* associated with that type. Types are a classification mechanism for values. Each data type has a set of values associated with it.

The data types of the FOAM fall into three main categories, *primitive* data types, *reference* data types, and *generic* data types. The primitive data types are further sub-divided into the type *bool*, the *integral* data types and the *floating-point* data types which together form the numeric types of the FOAM. This classification is illustrated in Figure __ below.



Values associated with the primitive types, or *basic types*, can be used freely and efficiently in imperitive languages. Operations on basic values often correspond to single machine instructions or short sequences of machine instructions, so the operations are implemented efficiently. At the HIR level, these instructions are simulated by the FOAM in the form of *built-in operations*. The type *Bool* in the above classification is treated separately, as there is typically no Boolean basic type on modern architectures. It is commonly encoded using an integral basic type, and so its classification is primitive. The other primitive types are supported by modern architectures.

### Reference types

or *non-primitive* types, why called reference, list 'em. The *Nil* data type ... Nothing. 1-element type. Distinguished value.

### Generic types

*Arb* and *Word* are defined according to size. There is no specific value set associated with each ...

Types are first class values.

Functions are first class values.

**Plan:**

- discuss how they fit into the big picture.
- for each type:
  - introduce the actual values and/or use of each type with its built-ins. ieee fp, 2's-complement stuff here.
  - specific *uses* of each type in the AM. This sets the stage for how these things can be used.
- Talk about how the types work together (ie arithmetic through single integer, casting and type promotion etc... . there is a cast instruction but there are also many built-ins dedicated to casting).

## 3.1 Primitive Data Types

The primitive data types of the FOAM are listed below, along with their intended representation. The ANSI C-like *long* data type is replaced with the generic *BInt* (*big integer*) reference data type which represents integers of arbitrary size (discuss some issues like efficiency and why this decision was made, perhaps a hidden long imp. subset of BInt implementation).

## 3.2 Representation of Primitives

The following table describes the intended representation of each of the primitive data types of the FOAM.

| Primitive Type | Representation |
|---|---|
| *Byte* | Unsigned integer represented in 8-bits. |
| *HInt* | Half precision integer. Signed 2's complement integer in 16-bits. |
| *SInt* | Single precision integer. Signed 2's complement integer in 32-bits. |
| *SFlo* | Single precision floating point. IEEE format. Maps to the ANSI C *float* type. |
| *DFlo* | Double precision floating point. IEEE format. Maps to the ANSI C *double* type. |
| *Char* | ASCII Character, 8-bit representation. |
| *Bool* | Boolean value, 0 or 1, 8-bit representation. |

**Table ?**
The primitive data types of the FOAM.

## 3.3 Primitive Values

Since the AM is designed to be platform independent the *intended* values of each type are defined. The intended values of a primitive data type are of course dependent on the intended representation of that primitive data type as outlined above. The intended representation of each primitive data type must be capable of physically representing the indended value set of that data type.

The actual values are dependent upon the accuracy of the implementation of the AM and perhaps on platform. This manual discusses the FOAM independent of platform. FOAM code is currently targeted to ANSI C and Common Lisp (...). A FOAM interpreter written in C has also been implemented. The type correspondence between the C language and the FOAM is outlined for each primitive type.

The following table describes the intended value set of each of the primitive data types of the FOAM.

| Primitive Type | Values |
|---|---|
| *Byte* | At least the positive integers $0..2^7 - 1$. Capable of $0..2^8 - 1$. |
| *HInt* | Integral values $-2^{15}..2^{15} - 1$ inclusive. |
| *SInt* | At least the values $-2^{23}..2^{23} - 1$. Capable of values $-2^{31}..2^{31} - 1$. |
| *SFlo* | IEEE 754 32-bit single precision value set $*$. |
| *DFlo* | IEEE 754 64-bit double precision value set $*$. |
| *Char* | $0..2^7 - 1$ inclusive. The ASCII character set. |
| *Bool* | Integral values 0 or 1. |

**Table ?**
The value sets of the primitive types of the FOAM.

$*$ As specified in IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985 (IEEE, New York).

## 3.4 Built-In Operations on Primitive Types

### Byte

Type Byte is used to compactly represent small positive integers. This is primarily useful in arrays. To compute with Byte values, convert them to SInt first. Type Byte must be able to represent at least the values $0..2^7 - 1$. Bytes are used for numeric data and are never subject to character set conversion.

| FOAM instruction | | | | Description |
|---|---|---|---|---|
| Byte0: | () | $\rightarrow$ | Byte | |
| Byte1: | () | $\rightarrow$ | Byte | |
| ByteMin: | () | $\rightarrow$ | Byte | |
| ByteMax: | () | $\rightarrow$ | Byte | |

**Table ?**
Byte size integer built-in operations

### Short Integer

Type HInt is used to compactly represent small signed "half precision" integers. This is primarily useful in arrays. To compute with Byte values, convert them to SInt first. Type HInt must be able to represent at least the values $-2^{15}..2^{15} - 1$.

| FOAM instruction | | | | Description |
|---|---|---|---|---|
| HInt0: | () | $\rightarrow$ | HInt | |
| HInt1: | () | $\rightarrow$ | HInt | |
| HIntMin: | () | $\rightarrow$ | HInt | |
| HIntMax: | () | $\rightarrow$ | HInt | |

**Table ?**
Short integer built-in operations

### Integer

Type SInt is used to represent signed single precision integers. Type SInt must be able to represent at least the values $-2^{23}..2^{23} - 1$.

The values behave as if represented in two's complement for the logical operations (Bool, Not, And, Or). If arithmetic operations overflow, the result is not defined and may or may not equal the true value modulo $2^{machine-wordsize}$.

The operations SIntPlusMod, SIntMinusMod, SIntTimesMod require their first 2 arguments to be in the range $0..m-1$, for m = third argument. Otherwise the result is not defined.

The operation SIntLength is the number of bits required to represent the number in two's complement and in particular can be less than the word size.

| FOAM instruction | | | | Description |
|---|---|---|---|---|
| SInt0: | () | → | SInt | |
| SInt1: | () | → | SInt | |
| SIntMin: | () | → | SInt | |
| SIntMax: | () | → | SInt | |
| SIntIsZero: | (SInt) | → | Bool | |
| SIntIsNeg: | (SInt) | → | Bool | |
| SIntIsPos: | (SInt) | → | Bool | |
| SIntIsEven: | (SInt) | → | Bool | |
| SIntIsOdd: | (SInt) | → | Bool | |
| SIntEQ: | (SInt,SInt) | → | Bool | |
| SIntNE: | (SInt,SInt) | → | Bool | |
| SIntLT: | (SInt,SInt) | → | Bool | |
| SIntLE: | (SInt,SInt) | → | Bool | |
| SIntNegate: | (SInt) | → | SInt | |
| SIntPrev: | (SInt) | → | SInt | |
| SIntNext: | (SInt) | → | SInt | |
| SIntPlus: | (SInt,SInt) | → | SInt | |
| SIntMinus: | (SInt,SInt) | → | SInt | |
| SIntTimes: | (SInt,SInt) | → | SInt | |
| SIntTimesPlus: | (SInt,SInt,SInt) | → | SInt | |
| SIntMod: | (SInt,SInt) | → | SInt | |
| SIntQuo: | (SInt,SInt) | → | SInt | |
| SIntRem: | (SInt,SInt) | → | SInt | |
| SIntDivide: | (SInt,SInt) | → | (SInt,SInt) | |
| SIntGcd: | (SInt,SInt) | → | SInt | |
| SIntPlusMod: | (SInt,SInt,SInt) | → | SInt | |
| SIntMinusMod: | (SInt,SInt,SInt) | → | SInt | |
| SIntTimesMod: | (SInt,SInt,SInt) | → | SInt | |
| SIntTimesModInv: | (SInt, SInt, SInt, DFlo) | → | SInt | |
| SIntLength: | (SInt) | → | SInt | |
| SIntShiftUp: | (SInt,SInt) | → | SInt | |
| SIntShiftDn: | (SInt,SInt) | → | SInt | |
| SIntBit: | (SInt,SInt) | → | Bool | |
| SIntNot: | (SInt) | → | SInt | |
| SIntAnd: | (SInt,SInt) | → | SInt | |
| SIntOr: | (SInt,SInt) | → | SInt | |
| SIntXOr: | (SInt,SInt) | → | SInt | |

**Table ?**
Integer built-in operations

## Single Precision Floating-point

SFlo is single precision floating point. This type is used primarily for storing large quantities of floating pt data. In the tree form of Foam, SFlo values are represented in a machine-dependent single precision floating point format. The linear representation presently uses IEEE single precision format, however, this will change to extended single precision format.

SFloMax is the largest positive number. SFloEpsilon is the smallest positive number which can be represented. SFloMin is the most negative number which can be represented.

| FOAM instruction | | | | Description |
|---|---|---|---|---|
| SFlo0: | () | → | SFlo | |
| SFlo1: | () | → | SFlo | |
| SFloMin: | () | → | SFlo | |
| SFloMax: | () | → | SFlo | |
| SFloEpsilon: | () | → | SFlo | |
| SFloIsZero: | (SFlo) | → | Bool | |
| SFloIsNeg: | (SFlo) | → | Bool | |
| SFloIsPos: | (SFlo) | → | Bool | |
| SFloEQ: | (SFlo,SFlo) | → | Bool | |
| SFloNE: | (SFlo,SFlo) | → | Bool | |
| SFloLT: | (SFlo,SFlo) | → | Bool | |
| SFloLE: | (SFlo,SFlo) | → | Bool | |
| SFloNegate: | (SFlo) | → | SFlo | |
| SFloPrev: | (SFlo) | → | SFlo | |
| SFloNext: | (SFlo) | → | SFlo | |
| SFloPlus: | (SFlo,SFlo) | → | SFlo | |
| SFloMinus: | (SFlo,SFlo) | → | SFlo | |
| SFloTimes: | (SFlo,SFlo) | → | SFlo | |
| SFloTimesPlus: | (SFlo,SFlo,SFlo) | → | SFlo | |
| SFloDivide: | (SFlo,SFlo) | → | SFlo | |
| SFloRPlus: | (SFlo,SFlo,SInt) | → | SFlo | |
| SFloRMinus: | (SFlo,SFlo,SInt) | → | SFlo | |
| SFloRTimes: | (SFlo,SFlo,SInt) | → | SFlo | |
| SFloRTimesPlus: | (SFlo,SFlo,SFlo,SInt) | → | SFlo | |
| SFloRDivide: | (SFlo,SFlo,SInt) | → | SFlo | |
| SFloDissemble: | (SFlo) | → | (SInt,SInt,Word) | |
| SFloAssemble: | (SInt,SInt,Word) | → | SFlo | |
| SFloRound: | (SFlo) | → | BInt | |
| SFloTruncate: | (SFlo) | → | BInt | |
| SFloFraction: | (SFlo) | → | SFlo | |

**Table ?**
Single precision floating-point built-in operations

## Double Precision Floating-point

DFlo is double precision floating point. In the tree form of Foam, DFlo values are represented in a machine-dependent double precision floating point format. The linear representation presently uses IEEE double precision format, however, this will change to extended double precision format.

| FOAM instruction | | | | Description |
|---|---|---|---|---|
| DFlo0: | () | $\to$ | DFlo | |
| DFlo1: | () | $\to$ | DFlo | |
| DFloMin: | () | $\to$ | DFlo | |
| DFloMax: | () | $\to$ | DFlo | |
| DFloEpsilon: | () | $\to$ | DFlo | |
| DFloIsZero: | (DFlo) | $\to$ | Bool | |
| DFloIsNeg: | (DFlo) | $\to$ | Bool | |
| DFloIsPos: | (DFlo) | $\to$ | Bool | |
| DFloEQ: | (DFlo,DFlo) | $\to$ | Bool | |
| DFloNE: | (DFlo,DFlo) | $\to$ | Bool | |
| DFloLT: | (DFlo,DFlo) | $\to$ | Bool | |
| DFloLE: | (DFlo,DFlo) | $\to$ | Bool | |
| DFloNegate: | (DFlo) | $\to$ | DFlo | |
| DFloPrev: | (DFlo) | $\to$ | DFlo | |
| DFloNext: | (DFlo) | $\to$ | DFlo | |
| DFloPlus: | (DFlo,DFlo) | $\to$ | DFlo | |
| DFloMinus: | (DFlo,DFlo) | $\to$ | DFlo | |
| DFloTimes: | (DFlo,DFlo) | $\to$ | DFlo | |
| DFloTimesPlus: | (DFlo,DFlo,DFlo) | $\to$ | DFlo | |
| DFloDivide: | (DFlo,DFlo) | $\to$ | DFlo | |
| DFloRPlus: | (DFlo,DFlo,SInt) | $\to$ | DFlo | |
| DFloRMinus: | (DFlo,DFlo,SInt) | $\to$ | DFlo | |
| DFloRTimes: | (DFlo,DFlo,SInt) | $\to$ | DFlo | |
| DFloRTimesPlus: | (DFlo,DFlo,DFlo,SInt) | $\to$ | DFlo | |
| DFloRDivide: | (DFlo,DFlo,SInt) | $\to$ | DFlo | |
| DFloDissemble: | (DFlo) | $\to$ | (SInt,SInt,Word,Word) | |
| DFloAssemble: | (SInt,SInt,Word,Word) | $\to$ | DFlo | |
| DFloRound: | (DFlo) | $\to$ | BInt | |
| DFloTruncate: | (DFlo) | $\to$ | BInt | |
| DFloFraction: | (DFlo) | $\to$ | DFlo | |

**Table ?**
Double precision floating-point built-in operations

## 3.5 Boolean Operations

Although the Java virtual machine defines a boolean type, it only provides very limited support for it. There are no Java virtual machine instructions solely dedicated to operations on boolean values. Instead, expressions in the Java programming language that operate on boolean values are compiled to use values of the Java virtual machine *int* data type.

Type Bool contains the values 'false' and 'true'. Values of this type are used to control the sequence of program evaluation. In a C implementation the values can be represented as the integers 0 and 1. In a Lisp implementation the values can be represented as Nil and T.

| FOAM instruction | | | | Description |
|---|---|---|---|---|
| BoolFalse: | () | $\rightarrow$ | Bool | |
| BoolTrue: | () | $\rightarrow$ | Bool | |
| BoolNot: | (Bool) | $\rightarrow$ | Bool | |
| BoolAnd: | (Bool, Bool) | $\rightarrow$ | Bool | |
| BoolOr: | (Bool, Bool) | $\rightarrow$ | Bool | |
| BoolEQ: | (Bool, Bool) | $\rightarrow$ | Bool | |
| BoolNE: | (Bool, Bool) | $\rightarrow$ | Bool | |

**Table ?**
Boolean operations

BoolAnd and BoolOr are not conditional, that is both the arguments are evaluated in every case.

## 3.6 Character

## Character operations

Type Char contains letters, numerals and other text constituents. Char Data may need to be converted to a native character set (e.g. EBCDIC) for an implementation. CharLower and CharUpper convert the case of letters and do not modify other character values. CharOrd converts a character to a small integer and CharNum does the reverse.

| FOAM instruction | | | | Description |
|---|---|---|---|---|
| CharSpace: | () | → | Char | |
| CharNewline: | () | → | Char | |
| CharTab: | () | → | Char | |
| CharMin: | () | → | Char | |
| CharMax: | () | → | Char | |
| CharIsDigit: | (Char) | → | Bool | |
| CharIsLetter: | (Char) | → | Bool | |
| CharEQ: | (Char,Char) | → | Bool | |
| CharNE: | (Char,Char) | → | Bool | |
| CharLT: | (Char,Char) | → | Bool | |
| CharLE: | (Char,Char) | → | Bool | |
| CharLower: | (Char) | → | Char | |
| CharUpper: | (Char) | → | Char | |
| CharOrd: | (Char) | → | SInt | |
| CharNum: | (SInt) | → | Char | |

**Table ?**
Character built-in operations

## 3.7 Reference Types

The reference types of the FOAM are listed below. They are categorized as reference types because they are not directly supported by modern architectures like primitive types.

1. *BInt* Represented as a 32-bit pointer. Implemented in /aldor/lib/libfoam/links/bigint.{h,c} as :

   ```
   typedef struct bint {
           Bool    isNeg;          /* Sign of number. Is it negative?   */
           Length  placea;         /* No. of slots allocated for placev */
           Length  placec;         /* No. of slots used in placev       */
           BIntS   placev[NARY];   /* Digits in radix representation    */
   } *BInt;
   ```

   Values of BInt are "demoted" to primitive values for efficiency if the size permits and recast back to their BInt representation after the computation using the following functions

   ```
   #define BIntToInt(b)    ((IInt) UnImmed(ptrToLong(b)))
   #define IntToBInt(n)    ((BInt) ptrFrLong(MkImmed(n)))
   ```

- /aldor/lib/libfoam/links/foam_i.{h,c} implements a BInt library for the C back-end.
- does Lisp and Scheme back-ends already have an implicit BInt?

2. *Arr* Represented as a 32-bit pointer.
3. *Rec* Represented as a 32-bit pointer.
4. *Env* Represented as a 32-bit pointer.
5. *Prog* Represented as a 32-bit pointer.
6. *Clos* Represented as a 32-bit pointer.
7. *Ptr* Represented as a 32-bit pointer.
8. *NOp* ???

## 3.8 Reference Values

## 3.9 Built-In Operations on Reference Types

### Big Integer

Type BInt is used to represent integers which may be arbitrarily large. The operations on BInt require dynamic memory allocation and garbage collection. BIntIsSmall tests whether a value could be represented as a SInt. Operations have the same meaning as for SInt but will never overflow.

| FOAM instruction | | | | Description |
|---|---|---|---|---|
| BInt0: | () | → | BInt | |
| BInt1: | () | → | BInt | |
| BIntIsZero: | (BInt) | → | Bool | |
| BIntIsNeg: | (BInt) | → | Bool | |
| BIntIsPos: | (BInt) | → | Bool | |
| BIntIsEven: | (BInt) | → | Bool | |
| BIntIsOdd: | (BInt) | → | Bool | |
| BIntIsSingle: | (BInt) | → | Bool | |
| BIntEQ: | (BInt, BInt) | → | Bool | |
| BIntNE: | (BInt, BInt) | → | Bool | |
| BIntLT: | (BInt, BInt) | → | Bool | |
| BIntLE: | (BInt, BInt) | → | Bool | |
| BIntNegate: | (BInt) | → | BInt | |
| BIntPrev: | (BInt) | → | BInt | |
| BIntNext: | (BInt) | → | BInt | |
| BIntPlus: | (BInt, BInt) | → | BInt | |
| BIntMinus: | (BInt, BInt) | → | BInt | |
| BIntTimes: | (BInt, BInt) | → | BInt | |
| BIntTimesPlus: | (BInt, BInt, BInt) | → | BInt | |
| BIntMod: | (BInt, BInt) | → | BInt | |
| BIntQuo: | (BInt, BInt) | → | BInt | |
| BIntRem: | (BInt, BInt) | → | BInt | |
| BIntDivide: | (BInt, BInt) | → | (BInt, BInt) | |
| BIntGcd: | (BInt, BInt) | → | BInt | |
| BIntSIPower: | (BInt, SInt) | → | BInt | |
| BIntBIPower: | (BInt, BInt) | → | BInt | |
| BIntLength: | (BInt) | → | SBInt | |
| -BIntShift: | (BInt, SBInt) | → | BInt | |
| +BIntShiftUp: | | | | |
| +BIntShiftDn: | | | | |
| +BIntShiftRem: | | | | |
| BIntBit: | (BInt, SInt) | → | Bool | |

**Table ?**
Big integer built-in operations

### Ptr

| | | | |
|---|---|---|---|
| PtrNil: | () | → | Ptr |
| PtrIsNil: | (Ptr) | → | Bool |
| PtrEQ: | (Ptr, Ptr) | → | Bool |
| PtrNE: | (Ptr, Ptr) | → | Bool |

## 3.10 Generic Types

The generic types of the FOAM are listed below. They are categorized as generic types because a variable declared to be of a generic type can take on either primitive or reference values subject to some constraints.

*Word*: Single precision arbitrary: Ptr, Char, Bool, Byte, HInt, SInt, SFlo.
*Arb*: Arbitrary value: Word, DFlo.

*Nil*:

## 3.11 Generic Values

There is no specific value set associated with either of the generic types Word or Arb. Nil is the distinguished value of type Nil. It represents the "nothing" value. The Type Promotion and Casting section below will discuss interpreting the value of a variable declared to be of generic type.

### Word

| FOAM instruction | Description |
|---|---|
| +WordTimesDouble | |
| +WordDivideDouble | |
| +WordPlusStep | |
| +WordTimesStep | |

**Table ?**
Word built-in operations.

## 3.12 Type Promotion and Casting

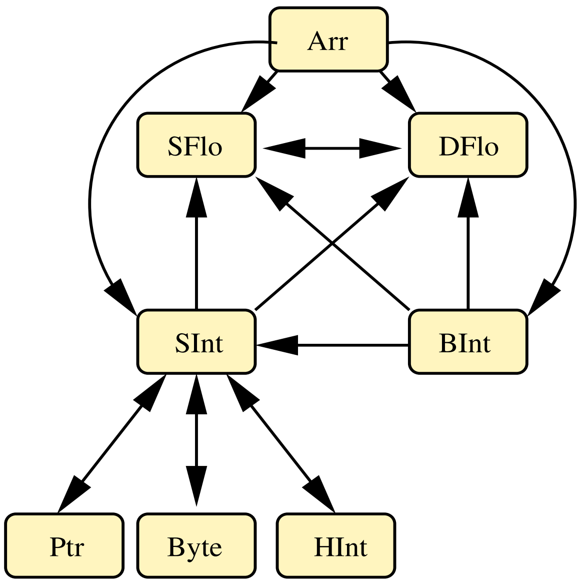
A *type conversion* is an operation which changes a value from one type to a value of another type to which the original value would not otherwise belong.

In Aldor, primitive type conversion is accomplished with the *pretend* conversion operation. Most type conversions are performed by functions. By convention, type conversion is most often performed by a *coerce* function, exported by either the source type or the destination type.

Bool can be converted to Byte. Byte, Bool can be converted to HInt. HInt, Byte, Bool can be converted to SInt. Word can be converted to and from Ptr, Char, Bool, Byte, HInt, SInt, SFlo. Arb can be any primitive value (Word or DFlo). ...

| FOAM instruction | | | | Description |
|---|---|---|---|---|
| SFloToDFlo: | (SFlo) | $\rightarrow$ | DFlo | |
| DFloToSFlo: | (DFlo) | $\rightarrow$ | SFlo | |
| ByteToSInt: | (Byte) | $\rightarrow$ | SInt | |
| SIntToByte: | (SInt) | $\rightarrow$ | Byte | |
| HIntToSInt: | (HInt) | $\rightarrow$ | SInt | |
| SIntToHInt: | (SInt) | $\rightarrow$ | HInt | |
| SIntToBInt: | (SInt) | $\rightarrow$ | BInt | |
| BIntToSInt: | (BInt) | $\rightarrow$ | SInt | |
| SIntToSFlo: | (SInt) | $\rightarrow$ | SFlo | |
| SIntToDFlo: | (SInt) | $\rightarrow$ | DFlo | |
| BIntToSFlo: | (BInt) | $\rightarrow$ | SFlo | |
| BIntToDFlo: | (BInt) | $\rightarrow$ | DFlo | |
| PtrToSInt: | (Ptr) | $\rightarrow$ | SInt | |
| SIntToPtr: | (SInt) | $\rightarrow$ | Ptr | |
| ArrToSFlo: | (Arr) | $\rightarrow$ | SFlo | |
| ArrToDFlo: | (Arr) | $\rightarrow$ | DFlo | |
| ArrToSInt: | (Arr) | $\rightarrow$ | SInt | |
| ArrToBInt: | (Arr) | $\rightarrow$ | BInt | |

**Table**
Casting the Promotion



## 3.13 Additional Built-in Operations

### Text operations

FormatXxx takes a value of type Xxx, a character array and an integer index. The operation formats the value into the character array starting at the position given by the integer. The result is the number of characters placed in the array.

ScanXxx is the opposite of FormatXxx. It produces a value of type Xxx from the contents of the character array. The SInt argument is the index of the array element to start at and the SInt return value is the index of the first unscanned array element following.

| | | | |
|---|---|---|---|
| FormatSFlo: | (SFlo,Arr,SInt) | → | SInt |
| FormatDFlo: | (DFlo,Arr,SInt) | → | SInt |
| FormatSInt: | (SInt,Arr,SInt) | → | SInt |
| FormatBInt: | (BInt,Arr,SInt) | → | SInt |

| | | | |
|---|---|---|---|
| ScanSFlo: | (Arr, SInt) | → | (SFlo, SInt) |
| ScanDFlo: | (Arr, SInt) | → | (DFlo, SInt) |
| ScanSInt: | (Arr, SInt) | → | (SInt, SInt) |
| ScanBInt: | (Arr, SInt) | → | (BInt, SInt) |

| | | | |
|---|---|---|---|
| PlatformRTE: | () | → | SInt |
| PlatformOS: | () | → | SInt |
| Halt: | (SInt) | → | Nil |

**BInt?**

## 4 The FOAM Instruction Set.

**Values**

**Lexical Environment Levels**

## 5 Grammar

## 6 The Semantics of FOAM Programs

## 7 The FOAM and the IEEE 754 Standard

Summarize relavent information about the IEEE 754 standard and develop the context in terms of the AM.

### 7.1 IEEE 754 Rounding Modes

The FOAM fully supports all of the rounding modes mandated by the IEEE 754 standard. The rounding modes of the AM are represented through the interface provided by the following built-in operations.

| Rounding Mode | *SInt* value |
|---|---|
| RoundZero | 0 |
| RoundNearest | 1 |
| RoundUp | 2 |
| RoundDown | 3 |
| RoundDontCare | 4 |

**Table**
Casting the Promotion

The *RoundDontCare* rounding mode is not part of the IEEE 754 standard. It allows for the faster execution of rounding mode sensitive operations, saving a rounding mode change where appropriate. Currently the result of the rounding mode insensitive operation

$$SFloPlus(SFlo, SFlo) \rightarrow SFlo$$

is equivalent to its rounding mode sensitive counterpart under mode *RoundDontCare*

$$SFloRPlus(SFlo, SFlo, \mathit{RoundDontCare}() \rightarrow SInt) \rightarrow SFlo.$$

Likewise for all other senstive SFlo built-in operations and DFlo built-in operations to their rounding mode insensitive counterparts.

The rounding mode sensitive operations of the FOAM are listed below. The desired rounding mode is passed into each using the interface built-in functions defined above. There are no other rounding mode sensitive operations in the FOAM. Currently the rounding mode functionality is explicitely coupled with arithmetic instructions and is never alone.

| Type | Rounding Mode Sensitive Operation | | | |
|---|---|---|---|---|
| SFlo | $SFloRPlus$ | $(SFlo, SFlo, SInt)$ | $\rightarrow$ | $DFlo$ |
| | $SFloRMinus$ | $(SFlo, SFlo, SInt)$ | $\rightarrow$ | $DFlo$ |
| | $SFloRTimes$ | $(SFlo, SFlo, SInt)$ | $\rightarrow$ | $DFlo$ |
| | $SFloRTimesPlus$ | $(SFlo, SFlo, SFlo, SInt)$ | $\rightarrow$ | $DFlo$ |
| | $SFloRDivide$ | $(SFlo, SFlo, SInt)$ | $\rightarrow$ | $DFlo$ |
| | $SFloRound$ | $(SFlo)$ | $\rightarrow$ | $BInt*$ |
| DFlo | $DFloRPlus$ | $(DFlo, DFlo, SInt)$ | $\rightarrow$ | $DFlo$ |
| | $DFloRMinus$ | $(DFlo, DFlo, SInt)$ | $\rightarrow$ | $DFlo$ |
| | $DFloRTimes$ | $(DFlo, DFlo, SInt)$ | $\rightarrow$ | $DFlo$ |
| | $DFloRTimesPlus$ | $(DFlo, DFlo, DFlo, SInt)$ | $\rightarrow$ | $DFlo$ |
| | $DFloRDivide$ | $(DFlo, DFlo, SInt)$ | $\rightarrow$ | $DFlo$ |
| | $DFloRound$ | $(DFlo)$ | $\rightarrow$ | $BInt*$ |

**Table ?**
The rounding mode sensitive operations of the FOAM.

* BInt can be cast directly or indirectly to and from every other integral primitive data type supported by the FOAM.

See Appendix ? for more information on the implementation of rounding modes in the FOAM.

## 7.2 IEEE 754 Exceptions

How are exceptions handled in the FOAM? trap? signal? Does the FOAM support signalling floating point comparisons? Comment on these issues.

### 7.3 IEEE 754 Extended Value Set Formats

Does the FOAM support the single-extended abd double-extended floating-point value set formats of IEEE 754?

## 8 Examples

## 9 Appendix A

Below is a partial tour of the source code.

- **foam_c.h , foam_c.c**
  Run time support for the C version of the abstract machine. The operations in this file provide an implementation of FOAM for use with C code. This file includes an implemetation of the built-in operations described in Section ?
- **foam_cfp.h , foam_cfp.c**
  Currently, rounding mode changes are not implemented in hardware. They are implemented using the functions
  - FiSFlo fiSFloRId(FiSlfo, FiSInt) which uses functions fiSFloNext(...), fiSFloPrev(...) to *simulate* rounding, and
  - FiDFlo fiDFloRId(FiSlfo, FiSInt) which uses functions fiDFloNext(...), fiDFloPrev(...) to *simulate* rounding

  in the file *aldor/src/foam_c.*{*h,c*}.

  Platform dependent ieee support support is implemented in *aldor/src/foam_cfp.*{*h,c*}. Rounding support is implemented by the functions
  - FiWord fiIeeeGetRoundingMode(FiWord s)
  - FiWord fiIeeeSetRoundingMode(FiWord s).

  The interpreter (*aldor/src/fint.c*) uses the implementation of rounding modes provided by *foam_c.**, as illustrated by the following code excerpt. Built-in function fiSFloRPlus is implemented with a call to function fiSFloRId (described above).*aldor/src/foam_cfp.*{*h,c*} appear to go unused.

  ...

```
case FOAM_BVal_SFloRPlus:
          fintEval(&expr1);
          fintEval(&expr2);
          fintEval(&expr3);

          retDataObj->fiSFlo =
          fiSFloRPlus(expr1.fiSFlo,expr2.fiSFlo, expr3.fiSInt);

...
```

- The signitures for all FOAM instructions are located in a sequence of tables at the end of **foam.c**.
- ...

## 10 Runtime System

- noOperation: () -¿ (); ++ Do nothing — used to clobber initialisation fns.

  extendMake: DomainFun(DomainRep) -¿ Domain; ++ extendMake(fun) creates a new lazy extend domain object; extendFill!: (DomainRep, Array Domain) -¿ (); ++ adds the extendee, extender pair to an extension domain lazyGetExport!: (Domain, Hash, Hash) -¿ LazyImport; ++ creates a lazy function to retrieve the export lazy-ForceImport: LazyImport-¿Value; ++ forces a get on the lazy value rtConstSIntFn: SingleInteger-¿(()-¿SingleInteger); ++ Save on creating functions. rtAddStrings: (Array Hash, Array String) -¿ (); ++ Adds more strings to the list of known exports domainPrepare!: Domain -¿ (); ++ initializes a domain.
- * * A procedure stack frame consists in:

```
 *   - header (starting at bp, see below)
 *   - parameters, if any (starting at bp + PAR_OFFSET)
 *   - locals, if any (locValues points to (Loc 0))
 *   - fluids, if any (fluidValues points to (Fluid 0))
 *
 * * Stack Chaining
 * In order to provide a virtually infinite stack, the interpreter stack is
 * organized as a list of stack. Every element has size STACK_SIZE. The
 * starting stack is <headStack>. If a stackFrameAlloc or stackAlloc operation
```

```
* needs X bytes and such amount is not available in the current stack, then
* a new stack of size STACK_SIZE is dynamically allocated and is chained to
* the previous stack.
```

## A

| `BVal` *index* | *builtin name* | *signature* |
|---|---|---|
| FOAM_BVAL_START=0 | BoolFalse | |
| 1 | BoolTrue | |
| 2 | BoolNot | |
| 3 | BoolAnd | |
| 4 | BoolOr | |
| 5 | BoolEQ | |
| 6 | BoolNE | |
| 7 | CharSpace | |
| 8 | CharNewline | |
| 9 | CharTab | |
| 10 | CharMin | |
| 11 | CharMax | |
| 12 | CharIsDigit | |
| 13 | CharIsLetter | |
| 14 | CharEQ | |
| 15 | CharNE | |
| 16 | CharLT | |
| 17 | CharLE | |
| 18 | CharLower | |
| 19 | CharUpper | |
| 20 | CharOrd | |
| 21 | CharNum | |
| 22 | SFlo0 | |
| 23 | SFlo1 | |
| 24 | SFloMin | |
| 25 | SFloMax | |
| 26 | SFloEpsilon | |
| 27 | SFloIsZero | |
| 28 | SFloIsNeg | |
| 29 | SFloIsPos | |
| 30 | SFloEQ | |
| 31 | SFloNE | |
| 32 | SFloLT | |
| 33 | SFloLE | |
| 34 | SFloNegate | |
| 35 | SFloPrev | |
| 36 | SFloNext | |
| 37 | SFloPlus | |
| 38 | SFloMinus | |
| 39 | SFloTimes | |
| 40 | SFloTimesPlus | |
| 41 | SFloDivide | |
| 42 | SFloRPlus | |
| 43 | SFloRMinus | |
| 44 | SFloRTimes | |
| 45 | SFloRTimesPlus | |
| 46 | SFloRDivid | |