FOAM: A First Order Abstract Machine Version 0.35

Stephen M. Watt
Peter A. Broadbery
Pietro Iglio
Scott C. Morrison
Jonathan M. Steinbach

IBM Thomas J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598

November 1, 1994

1 Purpose

Foam is a programming language intended for use by other computer programs. In particular, foam is generated by the $A^\sharp$ compiler.

This report is a snapshot of a work in progress, and the authors invite comments. There will likely be some changes between the form of Foam described in this document and the form used by the first release of $A^\sharp$.

Foam has been defined with the following goals:

1. it has a well-defined semantics, independent of platform,
2. it has an efficient mapping to Common Lisp and to ANSI C, and
3. its structure allows easy manipulation.

Foam-to-Foam transformations produce equivalent programs with improved performance. These transformations are independent of hardware platform and can be agressively applied even by a cross-compiler.

Having an easy mapping to Common Lisp and to ANSI C is a more nebulous goal. However, the intent is that these mappings be relatively simple and easy to implement efficiently. For example, there is no concept of addresses in Foam since addresses are unpredictable with a Lisp garbage collector. Likewise, Foam does not have self identifying data objects since this is not available in C.

FOAM is not restricted to the precise intersection of C and Lisp. Some aspects are handled by support libraries. Big integer arithmetic is assumed as part of FOAM, and this is provided as a library for C. Also the memory model differs from both C and Lisp in some details: garbage collection is assumed (this is a run time support library in C) and it is possible to make an explicit request to free storage (in Lisp this is ignored).

A FOAM program is comprised of a flat sequence of commands. FOAM types have various sizes and uses. or example, "`Char`" is a text character whereas "`Byte`" is a character sized integer, "`DFlo`" is a double precision floating point, "`Ptr`" can point to an array, record, arbitrary sized integer, etc. Reference instructions contain the kind of reference and the position, e.g., "`Loc 3`" refers to the third local variable of the current function and "`RElt 7` x `2`" indicates the 2nd field of the record x, using the 7th layout format. FOAM operations consist of instructions, such as "`If` b n," which indicates that if b is true then proceed to label n, and builtin operations, e.g., "`HIntLT` a b" is a half-word-integer less-than comparison. The builtin operations are type specific and conversion operations are generally provided.

2 Instructions

$\langle$**Unit**$\rangle$ `::=`

[`Unit` $\langle$**DFmt**$\rangle$ $\langle$**DDef**$\rangle$]

The first 4 slots of DFmt are reserved for the declaration of the globals, constants, lexicals and fluids for the unit, the fifth slot is always the empty format.

$\langle$**DFmt**$\rangle$ `::=`

[`DFmt` $\langle f_0 : \textbf{DDecl}\rangle$ ... $\langle f_{n-1} : \textbf{DDecl}\rangle$]

Layout formats for environment levels, records etc.

$\langle$**DDecl**$\rangle$ `::=`

[`DDecl` $\langle u : \textbf{Byte}\rangle$ $\langle d_0 : \textbf{Decl}\rangle$ ... $\langle d_{n-1} : \textbf{Decl}\rangle$]

Specify formats for globals, parameters, locals, fluids, lexicals and constants. The usage parameter indicates the context in which the ddecl may be used. Note that the Decls may either be `Decl` or `GDecl` instructions.

$\langle$**Decl**$\rangle$ `::=`

[`Decl` $\langle t : \textbf{Type}\rangle$ $\langle s : \textbf{String}\rangle$ $\langle p : \textbf{Byte}\rangle$ $\langle r0 : \textbf{Byte}\rangle$ $\langle r1 : \textbf{Byte}\rangle$]

Declaration of a single parameter, local or lexical. The type is given by t and the user's name for it can be unmangled from s. $r0$ and $r1$ are reserved (and used by the compiler). If t is *Rec* or *Arr*, $r1$ is the format number or type of the slot, respectively.

$\langle$**GDecl**$\rangle$ `::=`

[`Decl` $\langle t : \textbf{Type}\rangle$ $\langle s : \textbf{String}\rangle$ $\langle r0 : \textbf{Byte}\rangle$ $\langle r1 : \textbf{Byte}\rangle$]
$\langle dir : \textbf{Byte}\rangle$ $\langle p : \textbf{Byte}\rangle$ Declaration of a single global. The type is given by t and the user's name for it can be unmangled from s. The language protocol is specified by p. $r0$ and $r1$ are reserved (and used by the compiler). If t is *Rec* or *Arr*, $r1$ is the format number or type of the slot, respectively. The *dir* field indicates whether the declaration is an import or export.

$\langle$**DDef**$\rangle$ `::=`

[`DDef` $\langle v_0 : \textbf{Def}\rangle$ ... $\langle v_{n-1} : \textbf{Def}\rangle$]
Initial values for things.

$\langle$**Def**$\rangle$ `::=`

[`Def` $\langle r : \textbf{Reference}\rangle$ $\langle v : \textbf{Value}\rangle$]
r is typically a global or lexical. v is typically a program or closure.

$\langle$**DEnv**$\rangle$ `::=`

[`DEnv` $\langle e_0 : \textbf{Int}\rangle$ ... $\langle e_{n-1} : \textbf{Int}\rangle$]
List of format numbers for lexical environment levels. The empty environment indicates that the level is never accessed by this function or its children, while a 0 format number indicates that the level is closed over, but is empty.

$\langle$**DFluid**$\rangle$ `::=`

[`DFluid` $\langle i_0 : \textbf{Int}\rangle$ ... $\langle f_{i-1} : \textbf{Int}\rangle$]
Indicies into the unit's fluid list, indicating which are rebound at this level.

$\langle$**Cmd**$\rangle$ `::=` $\langle$**Def**$\rangle$ | $\langle$**Expr**$\rangle$ | one of

[`Seq` $\langle c_0 : \textbf{Cmd}\rangle$ $\langle c_1 : \textbf{Cmd}\rangle$ ... $\langle c_n : \textbf{Cmd}\rangle$]
Defines a sequence of Cmd. Seq represents the body of a program. If the execution of c_i does not cause the transfer of the control, then $c_i + 1$ is executed. The only way to exit a Seq is by a Return instruction. It follows that if the last instruction c_n is reached, then it must transfer the control (typically the last instruction is a Return or a Goto).

[`Goto` $\langle l : \textbf{Label}\rangle$]
Go to label l in current prog.

[`If` $\langle e : \textbf{Expr}\rangle$ $\langle t : \textbf{Label}\rangle$]

If e is BoolTrue, go to label t. Semantic restriction: e must be of type Bool.

[`Select` $\langle e : \textbf{Expr}\rangle$ $\langle l_0 : \textbf{Label}\rangle$ ... $\langle l_{n-1} : \textbf{Label}\rangle$]

e evaluates to an integer. If e is in $[0..n-1]$, goto label l_e in current program. Otherwise, continue. Semantic restrictions: e must be of type SInt.

[`Return` $\langle e : \textbf{Expr}\rangle$]

Return e from the current program.

[`Set` $\langle r : \textbf{Reference}\rangle$ $\langle e : \textbf{Expr}\rangle$]

Update the location given by r to contain the value e. The reference r may be a set of references given by `Values`. A call to a function returning multiple values always looks like: (Set (Values r_1 ... r_n) (MFmt f *call*)) where *call* can be any of the Foam call instructions. The format f describes the type of the returned parameters. This is the only context where `MFmt` can occur.

[`Lose` $\langle r : \textbf{Reference}\rangle$]

Modify the location given by r to point to no structure.

[`PushEnv` $\langle f : \textbf{Int}\rangle$ $\langle e : \textbf{Reference}\rangle$]

Push a new environment with format f onto the stack with e as its parent.

[`PopEnv`]

Pop an environment from the stack.

[`Protect` $\langle e : \textbf{Expr}\rangle$ $\langle p : \textbf{Reference}\rangle$]

Evaluate e, then p, returning the value of e. If a `Throw` occurs and e is abandoned, then p is evaluated and the `Throw` is resumed.

[`Throw` $\langle to : \textbf{Expr}\rangle$ $\langle e : \textbf{Expr}\rangle$]

Throw to the tag *to*, evaluating any intervening protect forms.

[`Halt`]

Terminates the current Foam program.

$\langle\textbf{Expr}\rangle$ `::=` $\langle\textbf{Value}\rangle$ | one of

[`BVal` $\langle n : \textbf{Int}\rangle$]

The n-th builtin value.

[`Label` $\langle n : \textbf{Int}\rangle$]

The n-th command in the current program.

[`Cast` $\langle t : \textbf{Type}\rangle$ $\langle e : \textbf{Expr}\rangle$]
: View value e as being of type t. Types other than DFLo may be cast freely to and from Word without loss of information.

[`ANew` $\langle t : \textbf{Type}\rangle$ $\langle e : \textbf{Expr}\rangle$]
: Form an array of e elements of type t filled with zeroes or nils of the appropriate type. e must be a SInt value.

[`RNew` $\langle f : \textbf{Int}\rangle$]
: Form a record with format f. The elements are filled with zeroes or nils of the appropriate types.

[`TRNew` $\langle f0 : \textbf{Int}\rangle$ $\langle f1 : \textbf{Int}\rangle$ $\langle s : \textbf{Expr}\rangle$]
: Form a record with trailing array whose initial part has format $f0$ and with s elements in its trailing array with element format $f1$.

[`RCopy` $\langle f : \textbf{Int}\rangle$ $\langle e : \textbf{Expr}\rangle$]
: Copy the record e with format f.

[`BCall` $\langle o : \textbf{Int}\rangle$ $\langle e_0 : \textbf{Expr}\rangle$ ... $\langle e_{n-1} : \textbf{Expr}\rangle$]
: Call builtin o on $e_0 ... e_{n-1}$. The expression (BCall o $e_0 ... e_{n-1}$) is equivalent to (OCall t (BVal o) (Env -1) $e_0 ... e_{n-1}$) where t is the return type of (BVal o).

[`CCall` $\langle t : \textbf{Type}\rangle$ $\langle c : \textbf{Expr}\rangle$ $\langle e_0 : \textbf{Expr}\rangle$ ... $\langle e_{n-1} : \textbf{Expr}\rangle$]
: Call closure c on $e_0 ... e_{n-1}$, returning type t. The expression (CCall t c $e_0 ... e_{n-1}$) is equivalent to (OCall t (CProg c') (CEnv c') $e_0 ... e_{n-1}$) where c is possibly a temporary to avoid multiple evaluation.

[`OCall` $\langle t : \textbf{Type}\rangle$ $\langle f : \textbf{Expr}\rangle$ $\langle e : \textbf{Reference}\rangle$ $\langle e_0 : \textbf{Expr}\rangle$ ... $\langle e_{n-1} : \textbf{Expr}\rangle$]
: Call the program f on $e_0 ... e_{n-1}$ in environment e, returning type t. The expression (OCall t f e $e_0 ... e_{n-1}$) is equivalent to (PCall `FOAM_Proto_Foam` t f e $e_0 ... e_{n-1}$)

[`PCall` $\langle p : \textbf{Int}\rangle$ $\langle t : \textbf{Type}\rangle$ $\langle f : \textbf{Expr}\rangle$ $\langle e : \textbf{Reference}\rangle$ $\langle e_0 : \textbf{Expr}\rangle$... $\langle e_{n-1} : \textbf{Expr}\rangle$]
: Call f with environment e and arguments $e_0 .. e_{n-1}$. e will be unused in the case of `FOAM_Proto_C`. according to protocol p, returning type t.

[`MFmt` $\langle f : \textbf{Int}\rangle$ $\langle c : \textbf{Expr}\rangle$]
: Wrapping `MFmt` around a call indicates that the call returns multiple values. See `Set` for more information.

[`Values` $\langle e_0 : \textbf{Expr}\rangle$ ... $\langle e_n : \textbf{Expr}\rangle$]
: Indicates multiple values. It can only occur in a return statement in a program returning multiple values, or on the left hand side of a `Set` instruction, when calling a function returning multiple values. See `Set`.

[`Catch` $\langle tag : \textbf{Name}\rangle$ $\langle e : \textbf{Expr}\rangle$]
: Give *rt* a tag suitable for use with Throw, and evaluate the expression, which is returned. In the case of a throw to the tag, the value given by the throw is returned.

$\langle\textbf{Value}\rangle$ `::=` $\langle\textbf{Reference}\rangle \mid \langle\textbf{Data}\rangle$
$\langle\textbf{Reference}\rangle$ `::=` one of

[`AElt` $\langle t : \textbf{Type}\rangle$ $\langle n : \textbf{Expr}\rangle$ $\langle a : \textbf{Expr}\rangle$]
: The n-th element of the array a, viewed as an array with components of type t.

[`RElt` $\langle f : \textbf{Int}\rangle$ $\langle r : \textbf{Expr}\rangle$ $\langle n : \textbf{Int}\rangle$]
: The n-th field of the record r with the f-th format in the current unit.

[`IRElt` $\langle f : \textbf{Int}\rangle$ $\langle r : \textbf{Expr}\rangle$ $\langle n : \textbf{Int}\rangle$]
: The n-th field of the record with a trailing array r with the f-th format for the initial part of the record in the current unit.

[`TRElt` $\langle f0 : \textbf{Int}\rangle$ $\langle f1 : \textbf{Int}\rangle$ $\langle r : \textbf{Expr}\rangle$ $\langle index : \textbf{Expr}\rangle$ $\langle n : \textbf{Int}\rangle$]
: The n-th field of the *index*-th element of the trailing array of r whose initial format is $f0$ and whose trailing array element has format $f1$ in the current unit.

[`EElt` $\langle l : \textbf{Int}\rangle$ $\langle n : \textbf{Int}\rangle$ $\langle f : \textbf{Int}\rangle$ $\langle e : \textbf{Reference}\rangle$]
: The n-th lexical of the l-th level in the environment e with the f-th format in the current unit.

[`Const` $\langle n : \textbf{Int}\rangle$]
: The n-th constant of the current unit.

[`Glo` $\langle n : \textbf{Int}\rangle$]
: The n-th global of the current unit.

[`Fluid` $\langle n : \textbf{Int}\rangle$]
: The n-th fluid of the current unit.

[`Par` $\langle n : \textbf{Int}\rangle$]
: The n-th parameter of the current function.

[`Loc` $\langle n : \textbf{Int}\rangle$]
: The n-th local of the current function.

[`Lex` $\langle l : \textbf{Int}\rangle$ $\langle n : \textbf{Int}\rangle$]
: The n-th lexical of the l-th level (function or unit).

[`Env` $\langle l : \textbf{Int}\rangle$]
: Environment beginning l levels out in the current prog.

[`EEnv` $\langle l : \textbf{int} \rangle$ $\langle e : \textbf{Reference} \rangle$]
  Environment beginning l levels out from the environment e.

[`CEnv` $\langle c : \textbf{Expr} \rangle$]
  The environment part of a closure.

[`CProg` $\langle c : \textbf{Expr} \rangle$]
  The program part of a closure.

[`EInfo` $\langle e : \textbf{Expr} \rangle$]
  Information related to the environment e. e must be an environment. The result is of type `Word`.

[`PRef` $\langle r : \textbf{Int} \rangle$ $\langle p : \textbf{Expr} \rangle$]
  Information related to the prog p. p must be a prog. r indicates the field of the prog information structure. currently this can only be 0, which is the hashcode of a function.

$\langle$**Data**$\rangle$ `::=` one of

[`Nil`]
  Nothing. Distinguished value of type Nil.

[`Char` $\langle$*char-value*$\rangle$]
  Character. ASCII character set.

[`Bool` $\langle$*bit-value*$\rangle$]
  BoolFalse or BoolTrue.

[`Byte` $\langle$*byte-value*$\rangle$]
  0..255

[`HInt` $\langle$*half-int-value*$\rangle$]
  2's complement 16 bit integer

[`SInt` $\langle$*single-int-value*$\rangle$]
  2's complement 32 bit integer (at least 24 bit?)

[`BInt` $\langle$*bigint-value*$\rangle$]
  signed magnitude big integer, any no of bits.

[`SFlo` $\langle$*single-float-value*$\rangle$]
  IEEE format

[`DFlo` $\langle$*double-float-value*$\rangle$]
  IEEE format

[`Arr` $\langle t : \textbf{Type} \rangle$ $\langle v_0 : \textbf{Value} \rangle$ ... $\langle v_{n-1} : \textbf{Value} \rangle$]

[`Rec` $\langle f : \mathbf{Int}\rangle$ $\langle v_0 : \mathbf{Value}\rangle$ ... $\langle v_{n-1} : \mathbf{Value}\rangle$]

[`Prog` $\langle n : \mathbf{Int}\rangle$ $\langle m : \mathbf{Int}\rangle$ $\langle t : \mathbf{Type}\rangle$ $\langle f : \mathbf{Int}\rangle$ $\langle b : \mathbf{Int}\rangle$ $\langle size : \mathbf{Int}\rangle$ $\langle time : \mathbf{Int}\rangle$ $\langle par : \mathbf{DDecl}\rangle$ $\langle loc : \mathbf{DDecl}\rangle$ $\langle lex : \mathbf{DEnv}\rangle$ $\langle fluid : \mathbf{DFluid}\rangle$ $\langle c_0 : \mathbf{Cmd}\rangle$ $\langle c_1 : \mathbf{Cmd}\rangle$ ... $\langle c_{n-1} : \mathbf{Cmd}\rangle$]
Program of size n bytes, and maximum label m, returning value of type t. If t has the value NOp then the program returns multiple values, where the format f describes the types of the values returned, otherwise f is 0. The integer b contains bits specifying whether: The program is a leaf. The program has side-effects. The program is a generator. The program has optimization info. The program contains OCalls. The program contains Consts. The program must or must not be inlined. The integer *size* is the number of nodes of the program + the *size* info of each Const prog which is referred in program. In other word, this is the growth that is expected inlining this program from another file. The integer *time* is the estimated time cost of the program. Of course, this is an approximation. *par* is the declaration of parameters. *loc* is the declaration of local variables. *lex* is an array of formats for the lexical levels. The commands $c_0..c_{n-1}$ are performed in sequence.

[`Clos` $\langle env : \mathbf{Value}\rangle$ $\langle prog : \mathbf{Value}\rangle$]

[`Ptr` $\langle v : \mathbf{Value}\rangle$]

$\langle$**Type**$\rangle$ `::=` one of

[`Nil`]
Nothing. 1-element type. Distinguished value.

[`Char`]
Character. E.g. `'a'`

[`Bool`]
0 or 1

[`Byte`]
Unsigned integer represented in 8 bits.
Bool can be converted to Byte

[`HInt`]
Half precision integer: Signed int in 16 bits
Byte, Bool can be converted to HInt

[`SInt`]
: Single precision integer: Signed int in 32 bits
HInt, Byte, Bool can be converted to SInt

[`BInt`]
: Signed integer of arbitrary number of bits.

[`SFlo`]
: Single precision floating point.

[`DFlo`]
: Double precision floating point. SFlo can be converted to DFlo

[`Arr`]
: Array (i.e. one dimensional array)

[`Rec`]
: Record

[`Env`]
: Environment.

[`Prog`]
: Program.

[`Clos`]
: Closure.

[`Ptr`]
: Pointer: Nil, BInt, Prog, Clos, Env, Arr, Rec

[`Word`]
: Single precision arbitrary: Ptr, Char, Bool, Byte, Hint, SInt, SFlo.

[`Arb`]
: Arbitrary value: Word, DFlo

[`NOp`]
: Used for multiple types context. See `Prog`.

3 Protocols

A *Protocol* is used to describe the interface through which an object should be called or accessed. The following protocols are currently produced by the $A^\sharp$compiler.

`FOAM_Proto_Foam` Use a natural mapping for Foam objects: for variables this is typically a Lisp or C identifier with a name derived from the *id* field in the declaration.

`FOAM_Proto_Other` Use a natural mapping for objects in the hosting system, for example C or Lisp identifiers with the same name as in the *id* field in the declaration.

`FOAM_Proto_Init` The object is an initializer for a unit, and so it should be called before any other globals from that unit. Foam units initialize those units which they use, but the first one is expected to be called by the hosting environment.

The other protocols (`FOAM_Proto_C, FOAM_Proto_Lisp, FOAM_Proto_Fortran`) indicate that the particular object should be treated as coming from the appropriate language, or that it should be accessible from that language. In these cases, no manipulation of the *id* field occurs.

4 Builtins

These descriptions are in the same order as in the enumeration foam.h. This list is expected to grow somewhat, as needed.

4.1 Operations on type Bool

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.

BoolFalse:	()	→	Bool
BoolTrue:	()	→	Bool
BoolNot:	(Bool)	→	Bool
BoolAnd:	(Bool, Bool)	→	Bool
BoolOr:	(Bool, Bool)	→	Bool
BoolEQ:	(Bool, Bool)	→	Bool
BoolNE:	(Bool, Bool)	→	Bool

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

4.2 Operations on type Char

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.

CharSpace:	()	→	Char
CharNewline:	()	→	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

4.3 Operations on type SFlo

SFlo is single precision floating point. 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. 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.

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
SFloPlus:	(SFlo,SFlo)	→	SFlo
SFloMinus:	(SFlo,SFlo)	→	SFlo
SFloTimes:	(SFlo,SFlo)	→	SFlo
SFloTimesPlus:	(SFlo,SFlo,SFlo)	→	SFlo
SFloDivide:	(SFlo,SFlo)	→	SFlo
SFloSIPower:	(SFlo,SInt)	→	SFlo
SFloBIPower:	(SFlo,BInt)	→	SFlo
SFloRound:	(SFlo)	→	BInt
SFloTruncate:	(SFlo)	→	BInt
SFloFloor:	(SFlo)	→	BInt
SFloCeiling:	(SFlo)	→	BInt

4.4 Operations on type DFlo

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.

DFlo0:	()	→	DFlo
DFlo1:	()	→	DFlo
DFloMin:	()	→	DFlo
DFloMax:	()	→	DFlo
DFloEpsilon:	()	→	DFlo
DFloIsZero:	(DFlo)	→	Bool
DFloIsNeg:	(DFlo)	→	Bool
DFloIsPos:	(DFlo)	→	Bool
DFloEQ:	(DFlo,DFlo)	→	Bool
DFloNE:	(DFlo,DFlo)	→	Bool
DFloLT:	(DFlo,DFlo)	→	Bool
DFloLE:	(DFlo,DFlo)	→	Bool
DFloNegate:	(DFlo)	→	DFlo
DFloPlus:	(DFlo,DFlo)	→	DFlo
DFloMinus:	(DFlo,DFlo)	→	DFlo
DFloTimes:	(DFlo,DFlo)	→	DFlo
DFloTimesPlus:	(DFlo,DFlo,DFlo)	→	DFlo
DFloDivide:	(DFlo,DFlo)	→	DFlo
DFloSIPower:	(DFlo,SInt)	→	DFlo
DFloBIPower:	(DFlo,BInt)	→	DFlo
DFloRound:	(DFlo)	→	BInt
DFloTruncate:	(DFlo)	→	BInt
DFloFloor:	(DFlo)	→	BInt
DFloCeiling:	(SFlo)	→	BInt

4.5 Operations on type 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.

Byte0:	()	→	Byte
Byte1:	()	→	Byte
ByteMin:	()	→	Byte
ByteMax:	()	→	Byte

4.6 Operations on type HInt

Type HInt is used to compactly represent small signed "half precision" integers. This is primarily useful in arrays. Type HInt must be able to represent at least the values -2**15..2**15-1.

HInt0:	()	→	HInt
HInt1:	()	→	HInt
HIntMin:	()	→	HInt
HIntMax:	()	→	HInt

4.7 Operations on type SInt

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. SIntShift is an arithmetic shift. The second argument is the number of bits to shift by. +ve implies shift up. -ve implies shift down. SIntBool(x,i) returns the i'th bit of x.

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
SIntSIPower:	(SInt,SInt)	→	SInt
SIntBIPower:	(SInt,BInt)	→	SInt
SIntPlusMod:	(SInt,SInt,SInt)	→	SInt
SIntMinusMod:	(SInt,SInt,SInt)	→	SInt
SIntTimesMod:	(SInt,SInt,SInt)	→	SInt
SIntLength:	(SInt)	→	SInt
SIntShift:	(SInt,SInt)	→	SInt
SIntBit:	(SInt,SInt)	→	Bool
SIntNot:	(SInt)	→	SInt
SIntAnd:	(SInt,SInt)	→	SInt
SIntOr:	(SInt,SInt)	→	SInt

4.8 Operations on type BInt

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.

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
BIntBit:	(BInt, SInt)	→	Bool

4.9 Operations on type Ptr

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

4.10 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)

4.11 Conversion Operations

SFloToDFlo:	(SFlo)	→	DFlo
DFloToSFlo:	(DFlo)	→	SFlo
ByteToSInt:	(Byte)	→	SInt
SIntToByte:	(SInt)	→	Byte
HIntToSInt:	(HInt)	→	SInt
SIntToHInt:	(SInt)	→	HInt
SIntToBInt:	(SInt)	→	BInt
BIntToSInt:	(BInt)	→	SInt
SIntToSFlo:	(SInt)	→	SFlo
SIntToDFlo:	(SInt)	→	DFlo
BIntToSFlo:	(BInt)	→	SFlo
BIntToDFlo:	(BInt)	→	DFlo
PtrToSInt:	(Ptr)	→	SInt
SIntToPtr:	(SInt)	→	Ptr
ArrToSFlo:	(Arr)	→	SFlo
ArrToDFlo:	(Arr)	→	DFlo
ArrToSInt:	(Arr)	→	SInt
ArrToBInt:	(Arr)	→	BInt
PlatformRTE:	()	→	SInt
PlatformOS:	()	→	SInt
Halt:	(SInt)	→	Nil

5 Semantics of Foam Programs

A Foam program `P` is a set of Units, with the following conditions:

- P contains at least a single unit.
- One unit in P is the starting unit. A Foam program starts with a call to the first Prog in the starting unit, with a null environment.

- Only globals are shared among the units. Globals are unique by name and protocol. It follows that two globals with the same name and protocol must also have the same type. Note: the same global in different units may appear in different positions. Globals with the same name and different protocols may be identified, but this is implementation-defined.
- The order of evaluation for the arguments of call is left undefined. An implementation may specify a particular order.
- The order of evaluation for the arguments of a PCall is related to the specific protocol being be used. In example, if the protocol is `FOAM_Proto_C`, then the C language evaluation order is used.

6 Forms of Foam Code

6.1 Tree format for Foam code

This representation is used by C programs to manipulate Foam code. See the C header file "`foam_c.h`".

6.2 Linear format for Foam code

The purpose of this representation is two-fold:

1. to save foam code compactly in files and
2. to be appropriate for direct interpretation.

The main purpose is (1).
The linear representation is an augmented prefix traversal of a foam tree.
For compactness,

1. only nodes with varying arity indicate the number of descendants ("argc") and
2. nodes which contain an integer index or an argc have multiple representations so the numbers can be be saved in as little space as necessary
3. fields such as builtin operation numbers or type codes are represented as immediate bytes and are understood by context.

For interpretation,

1. Step numbers used in Goto, If, Select are represented as relative offsets into the byte code string.

   If [expr] <label>

Select [expr] $< label_0 > < label_1 > ... < label_{n-1} >$

Goto <label>

2. The same idea is used for progs, but in this case the offset is to step n, i.e. just past the end of the last step. This allows whole program bodies to be skipped when finding/extracting/inflating a subtree in linear format.

   Prog <X:prog size> [F][i][t][b][p][l][x][c_0][c_1] ... [c_{n-1}]

3. All offsets in a given prog are represented in the same format, which is the format of the Prog instruction.

6.3 Instruction formats

For the linear, byte coded version, variant instruction formats are used to help represent the code more compactly. The instructions are divided into groups according to the meaning of the variant formats.

6.3.1 Fixed arity instructions (tree/data args)

NOp
BVal
Ptr
CProg
CEnv
Loose
Kill
Free
Return
Cast
ANew
Clos
Set
Def
AElt
If
Goto
Throw
Catch
Protect
Unit
PushEnv
PopEnv
MFmt
1 form of each instruction 1*24 = 24

6.3.2 Fixed arity (data args)

Nil		(0 data bytes)
Char	char-value	(1 data byte)
Bool	bit-value	(1 data byte)
Byte	byte-value	(1 data byte)
HInt	half-int-value	(2 data bytes)
SInt	single-int-value	(4 data bytes)
SFlo	single-float-value	(4 data bytes)
DFlo	double-float-value	(8 data bytes)
Word	single-precision-arbitrary	(4 data bytes)
Arb	double-precision-arbitrary	(8 data bytes)

1 form of each instruction 1*10 = 10

6.3.3 Fixed arity + Nary ()

Decl
BInt

DDecl
DFluid
DEnv
DDef
DFmt
Rec
Arr
Select
PCall
BCall
CCall
OCall
Seq
Values
Prog

5 forms of each instruction 5*17 = 85

0 $\Rightarrow$ gen argc. (4 bytes)
1 $\Rightarrow$ 1 byte argc.
2 $\Rightarrow$ argc = 0
3 $\Rightarrow$ argc = 1
4 $\Rightarrow$ argc = 2

6.3.4 Fixed arity 1 Int index ()

Par
Loc
Glo
Fluid
Const
Env
EEnv
RNew
PRef
EInfo
RCopy
Label
5 forms of each instruction 5*12 = 60

0 ⇒ gen index (4 bytes)
1 ⇒ 1 byte index
2 ⇒ index = 0
3 ⇒ index = 1
4 ⇒ index = 2

6.3.5 Multi-Int index: (including arity)

Lex
RElt
IRElt
TRNew
TRElt
EElt
5 forms of each instruction 5* 6 = 30

0 ⇒ gen indices (4 bytes each)
1 ⇒ 1 byte indices
2 ⇒ ix1 2 bytes, ix2 1 byte, [ix3 1 byte]
3 ⇒ ix1 1 byte, ix2 2 bytes, [ix3 1 byte]
4 ⇒ ix1 1 byte, ix2 1 byte, [ix3 2 bytes]

Total number of instructions including variant forms = 24+10+85+60+30 = 209

7 Acknowledgements

Foam was preceded by a number of earlier designs, named *SAM* for “Scratchpad Abstract Machine.”

In addition to the authors of this document, many people have contributed to these earlier designs and their implementations. These individuals include Florian Bundshuh, Marc Gaetano, Michael Monagan, Simon Robinson, and Knut Wolf.