Implementing FL in C

Alex Aiken Brennan Gaunce Brian Murphy

June 22, 1991

1 Purpose

This is a design document for a C implementation of FL. Decisions were made keeping two principles in mind:

- The generated C code should be 100% portable.
- The EFL target for C should make as much explicit about the implementation as possible.

The benefits of portability are obvious. Pursuing the second goal has two benefits. First, the runtime system (i.e., the extra stuff needed to support C execution that is not part of the program) is minimized. Second, more rewriting can be done.

2 Implementation Problems

The following is a list of major problems that must be solved. The order corresponds roughly to the order in which the implementation addresses the problems.

1. How are exceptions implemented?
2. How are the primitive data types implemented?
3. What are the primitive functions of the implementation?
4. How are function calls implemented?
5. What is the target language? (Are there any restrictions on use of primitives of the implementation?)
6. What is the algorithm to map an FL program to a program expressed in terms of the primitives of the implementation?
7. What does the code generator do?
8. How can we do separate compilation?
9. How is memory allocation/garbage collection handled?
10. What about I/O?

Number (1) is really a special case of (2); however, exceptions are different enough that we implement them differently.

We briefly outline the overall design of the C implementation. FL itself serves as the intermediate language, and C code is generated directly from FL. There are a number of new EFL functions specifically for this implementation. These are the primitives of the implementation and are translated directly to C by the code generator. At a high level, the implementation has the following phases:

1. Make explicit the representations of all primitive data types (except exceptions).
2. Introduce definitions that implement the environment PF in terms of lower-level implementation primitives that work on the representation introduced in step 2.
3. General rewriting.
4. Use existing prog strategy to eliminate tupling and compositions.
5. Map result of progging to a program that maps stores to stores (a "cprog"), introducing store analogs of the forb, reduceb, prog, and app combining forms. Also add explicit call and return code for defined functions.
6. Generate code.

The following sections address each of the problems above. Each section consists of two parts: a description of a proposed solution, and a "Notes" section for anything interesting but tangential to the main issue (a core dump).

3 Representations

For the most part, implementing FL in C is a study in standard program transformation—taking one program and transforming it to an equivalent, but better, program. However, there is one different idea in this design, and that is representing a program by a program that does *not* have the same meaning. We do this to make details of the implementation explicit, and thus available for optimization.

What does it mean for a "representation" of a program to be correct? Roughly speaking, given an expression e and its representation $R(e)$, we want to be able to recover the meaning $\mu(e)$ from the meaning of the representation $\mu(R(e))$. The following definition formalizes this idea.

Definition 3.1 (Representations) Let R be a function from FL expressions to FL expressions and let D be the domain of R. Then R is a *representation function* for D if there exists a 1-1 function f such that:

$$\forall e \in D \; \mu(e) = f(\mu(R(e)))$$

The following lemma shows that a representation function preserves meaning; that is, two expressions have the same meaning if and only if they have the same meaning in the representation.

Lemma 3.2 Let R be a representation function. Then for all e and e' in the domain of R

$$\mu(e) = \mu(e') \Leftrightarrow \mu(R(e)) = \mu(R(e'))$$

Proof: For the $\Rightarrow$ direction:

$$\begin{array}{rcll} \mu(e) & = & \mu(e') & \text{assumption} \\ f(\mu(R(e))) & = & f(\mu(R(e'))) & \text{apply def of } f \text{ to both sides} \\ \mu(R(e)) & = & \mu(R(e')) & f \text{ is 1-1} \end{array}$$

For the $\Leftarrow$ direction:

$$\begin{array}{rcll} \mu(R(e)) & = & \mu(R(e')) & \text{assumption} \\ f(\mu(R(e))) & = & f(\mu(R(e'))) & f \text{ is a function} \\ \mu(e) & = & \mu(e') & \text{apply def of } f \text{ to both sides} \end{array}$$

□

4 Implementing Exceptions

An exception is a tagged pair with a tag of zero. All FL functions are strict with respect to exceptions. In Lisp this semantics is implemented using the Lisp primitives catch and throw; in C we can use the analogous operations of setjump and longjump. However, some care is required, because longjump does not necessarily restore the state of local variables. This is discussed further in Section 9.

Another problem with implementing exceptions is correctly handling overflow and underflow of arithmetic operations. For integer operations, we do explicit bit-wise operations before and/or after an arithmetic operation to detect overflow and underflow. In the event of an arithmetic exception, the arguments are converted to bignums and the operation repeated. Thus, no integer arithmetic operation (except division by zero) produces a runtime exception. For floating point arithmetic, there is no graceful solution. For the moment, we do nothing; thus underflows and overflows of floating operations are not detected in the C implementation. Many C compilers generate a non-number for overflows and set underflows to zero. We may exploit this later.

4.1 Notes

The problems with arithmetic operations would disappear if we generated native code.

5 Implementing Primitive and User-Defined Types

We define representations for each of the primitive data types. The primitive data types of FL are: integers, reals, characters, booleans, vectors, sequences, tagged pairs, functions, and exceptions. Exceptions are handled specially and are not considered here. For efficiency, the other primitive types are rearranged somewhat at the implementation level. The actual set of primitive data types in the implementation is: small integers, big integers ("bigints"), floating point numbers, specials (the characters plus the constants **true**, **false**, and $\langle\rangle$), vectors, cons cells, tagged pairs, and closures. This is not the whole story, however. Some of these data types have variable size layouts (e.g., bigints, closures, vectors), some need to incorporate pointers to other data types (e.g., cons cells, tagged pairs), and, for portability, we cannot assume that we know the machine representation of floating point numbers (although we do assume we know the size). So, we also have pointers to bigints, pointers to floating point numbers, pointers to vectors, pointers to cons cells, pointers to tagged pairs, and pointers to closures as primitive data types.

Finally, it is necessary to distinguish types during execution, both to perform normal computation (i.e., conditionals discriminating on type) and to do garbage collection. In particular, the garbage collector

low order bits	*data type*
000	short integer
001	special
010	ptr to float
011	ptr to vector
100	ptr to cons cell
101	ptr to bigint
110	ptr to tagged pair
111	ptr to closure

Figure 1: Bit patterns for pointers, ints, and specials.

must be able to distinguish one kind of pointer from another. To accomplish this, we use the low-order three bits of a word to tag each pointer and immediate value (small integers and specials) with its data type. These bit patterns are given in Figure 1; note that each of these data types consumes exactly one word.

The choice of bit patterns is not completely arbitrary. Using "000" for short integers allows integer arithmetic to be implemented efficiently. Because we use three bits to identify pointers, all objects begin at addresses ending in "100" (that is, every object contains an even number of words). Pointers to cons cells (presumably the most common pointer) then automatically address an object. To deference pointers other than pointers to cons cells, some simple address arithmetic is required.

Bigints, cons cells, vectors, closures, and floating point numbers are implemented by contiguous blocks of memory. We name such entities *tuples* and write them using a sequence-like notation $\{x_1, \ldots, x_n\}$. The following table gives a description of the C representation of each data type:

type	*C representation*	*comment*
bigint	$\{\mathtt{n}, \mathtt{i}_1, \ldots, \mathtt{i}_n\}$	**n** is the length of the tuple
floating point number	$\{\ldots\}$	fixed length tuple (machine dependent)
cons cell	{**hd**, **consptr**|**NULL**}	pair of words
vector	$\{\mathtt{n}, \mathtt{x}_1, \ldots, \mathtt{x}_n\}$	**n** is the length of the tuple
function	$\{\mathtt{fn}, \mathtt{maxarg}, \mathtt{togo}, \mathtt{a}_1, \ldots, \mathtt{a}_i\}$	**maxarg** − **togo** = **i**

The representation of bigints is not yet specified. The representation of floating point numbers is machine-dependent, but we assume that on any machine it is a fixed, known number of words. The representation of closures contains a function name (**fn**), the number of arguments required before the function is applied (**maxarg**), and the number of arguments missing (**togo**).

For each of the primitive data types of the implementation, we choose an appropriate FL representation and convert FL programs to use this representation. The following table gives the representation of each type:

type	*FL representation*
small integer	integer
special	character, **true**, **false**, $\prec$ **null**, **0** $\succ$
ptr to bigint	$\prec$ **bigint**, $\langle$ **n**, $\mathtt{x}_1, \ldots, \mathtt{x}_n \rangle \succ$
ptr to float	real
ptr to cons cell	$\prec$ **cons**, $\langle$ **x**, **y** $\rangle \succ$
ptr to vector	$\langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle$
ptr to tagged pair	$\prec$ **utype**, $\langle$ **x**, **y** $\rangle \succ$
ptr to closure	$\prec$ **clos**, $\langle$ **fn**, **maxarg**, **togo**, $\mathtt{a}_1, \ldots, \mathtt{a}_i \rangle \succ$

The representation of characters, booleans, and reals is unchanged. Vectors are represented by sequences. The empty sequence is represented by the constant $\prec$ **null**, **0** $\succ$ and non-empty sequences are represented as Lisp lists (see the notes). Integers are represented either by integers (if the integer is less than some maximum) or by a tagged pair representing a bigint. Functions are represented by an abstract type of closures.

All of the primitive functions must have implementations that handle this representation; we postpone that discussion until Section 7. Note that because functions are not represented by functions, the meaning of application must change. The function $\boldsymbol{R}$ converts an FL program into one that works with this representation.

$$
\begin{aligned}
\mathtt{R(f{:}x)} &= \mathtt{pfapply{:}\langle R(f), R(x)\rangle} \\
\mathtt{R}(\langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle) &= \prec \mathtt{cons}, \langle \mathtt{R}(\mathtt{x}_1), \mathtt{R}(\langle \mathtt{x}_2, \ldots, \mathtt{x}_n \rangle) \rangle \succ \\
\mathtt{R}(\langle\rangle) &= \prec \mathtt{null}, \mathtt{0} \succ \\
\mathtt{R}('\mathtt{a}) &= '\mathtt{a} \\
\mathtt{R(i)} &= \mathtt{i} \text{ if } \mathtt{i} \leq \mathtt{MAXSMALLINT} \\
\mathtt{R(i)} &= \prec \mathtt{bigint}, \ldots \succ \text{ if } \mathtt{i} > \mathtt{MAXSMALLINT} \\
\mathtt{R(r)} &= \mathtt{r} \\
\mathtt{R(true)} &= \mathtt{true} \\
\mathtt{R(false)} &= \mathtt{false} \\
\mathtt{R}(\prec \mathtt{tag, e} \succ) &= \prec \mathtt{utype}, \langle \mathtt{R(tag)}, \mathtt{R(e)} \rangle \succ \\
\mathtt{R(ufn)} &= \mathtt{ufn} \\
\mathtt{R(fn)} &= \prec \mathtt{clos}, \langle \mathtt{pffn}, 1, 1 \rangle \succ
\end{aligned}
$$

In order to show that $\boldsymbol{R}$ satisfies Definition 3.1, we must exhibit the "pfx" function for each primitive function "x" in the environment PF. See Section 7. The following is a list of optimizations to $\boldsymbol{R}$. Keep in mind that primitives on the right-hand side of the "=" are implementation primitives.

$$
\begin{aligned}
\mathtt{R(f \circ g)} &= \begin{cases} \mathtt{c\_mkclos} \circ [\tilde{}\mathtt{fn}, \tilde{}\mathtt{j}, \tilde{}(\mathtt{i}-1), \tilde{}\mathtt{x}_1, \ldots, \tilde{}\mathtt{x}_n, \mathtt{R(g)}] \\ \quad \text{if } \mathtt{R(f)} = \prec \mathtt{clos}, \langle \mathtt{fn, j, i}, \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle \succ, \mathtt{i} \neq 1 \\ \mathtt{fn} \circ [\tilde{}\mathtt{x}_1, \ldots, \tilde{}\mathtt{x}_n, \mathtt{R(g)}] \\ \quad \text{if } \mathtt{R(f)} = \prec \mathtt{clos, fn, j, i}, \mathtt{x}_1, \ldots, \mathtt{x}_{j-1} \rangle, \mathtt{i} = 1 \end{cases} \\
\mathtt{R}([\mathtt{f}_1, \ldots, \mathtt{f}_n]) &= \mathtt{c\_mkcons} \circ [\mathtt{R}(\mathtt{f}_1), \mathtt{R}([\mathtt{f}_2, \ldots, \mathtt{f}_n])] \\
\mathtt{R}([]) &= \tilde{} \prec \mathtt{null}, \mathtt{0} \succ
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{R(p \rightarrow q;r)} &= \mathtt{c\_isfalse} \circ \mathtt{R(p)} \rightarrow \mathtt{R(r);R(q)} \\
\mathtt{R(K{:}e)} &= \mathtt{K{:}R(e)} \\
\mathtt{R(GC{:}i{:}fn{:}x_1{:}\ldots{:}x_j)} &= \prec \mathtt{clos}, \langle \mathtt{fn}, \mathtt{i}, \mathtt{i-j}, \mathtt{R(x_1)}, \ldots, \mathtt{R(x_j)} \rangle \succ \\
\mathtt{R(for{:}\langle e, b \rangle)} &= \mathtt{for{:}\langle R(e) \circ c\_mkcons \circ [s1, c\_mkcons \circ [s2, \tilde{} \prec null, 0 \succ]], R(b) \rangle}
\end{aligned}
$$

In the last optimization, there is probably a good chance that the rewriter (or progging) can eliminate the explicit sequence.

5.1 Notes

This representation implements sequences as Lisp lists. This seems unimaginative, but at this we don't have a convincing alternative.

We discussed having two different representations for functions, one for plain function names and another for closures, functions with some (but not enough) arguments. The purpose was to have a more efficient call mechanism for non-closures. The rewriter can do this optimization for us (or prog can do it), so this isn't a good reason to complicate things at this level. Was there another reason?

The representation of closures should have the list of arguments built in, rather than represented as a pointer to a sequence or a list. This means that whenever a new closure is built, a copy must be made of the old one, including all the arguments. This is better than the alternatives, because the number of arguments is generally very small (2).

6 The Implementation Primitives

We must decide which functions are implemented directly in C by the code generator. All others are implemented using these functions in FL. We must also choose a target language for the interface with the code generator. In this section, we give the implementation primitives and the target language. Subsequent sections give implementations for the rest of the primitive functions, and an algorithm to map FL (after representations are explicit) to the target.

In keeping with the goal of making as much explicit as possible, the implementation primitives are simple, each translating into a few C instructions. Only untupled versions of these functions are implemented; this is made explicit by the prog phase. Figure 2 lists the implementation primitives and gives a rough semantics.

The semantics of the easy implementation primitives is given in Figures 4 and 5. The control constructs are somewhat complex, so we describe them in more detail. The function **cprog** is a function from stores to stores, where a store is the usual FL abstract type.

$$
\mathtt{c\_prog{:}\langle x_1, \ldots, x_n \rangle{:}s_0} = \mathtt{s_n} \text{ where } \begin{cases}
\mathtt{s_i} = \mathtt{s_{i-1}[j \leftarrow f{:}\langle s_{i-1}(y_1), \ldots, s_{i-1}(y_m) \rangle]} & \text{if } \mathtt{x_i} = \mathtt{\langle j, f, y_1, \ldots, y_m \rangle} \\
\mathtt{s_i} = \mathtt{x_i{:}s_{i-1}} & \text{if } \mathtt{x_i} \text{ is a function} \\
\mathtt{s_i} = \mathtt{\perp_{err}} & \text{otherwise}
\end{cases}
$$

The list of $\mathtt{x_i}$ corresponds to a list of statements. The two kinds of statements correspond to assignment and control statements. The assignments read some locations from the store and compute a value that is put back into the store. The control statements themselves map stores to stores (such as conditionals, loops, etc.).

The function **c_func** is similar to **c_prog** , but instead of mapping stores to stores, it maps a sequence of values to a value. The purpose of **c_func** is to model a C function of some arguments. In the following

category	*function name*	*semantics*
arithmetic	addi2	adds two (raw) integers
	c_muli2	multiplies two integers
	c_subi2	subtracts two integers
	c_divi2	divides two integers (does not check for / 0)
	c_addr2	adds two reals
	c_mulr2	multiplies two reals
	c_subr2	subtracts two reals
	c_divr2	divides two reals (does not check for / 0)
	c_floor	maps real to next smaller integer
	c_ceiling	maps real to next larger integer
	c_float	maps integer to real
predicate	c_gettag	three low-order bits of representation as integer
	c_isnull	1 for $\prec$ ***null***, 0 $\succ$
	c_istrue	1 for true
	c_isfalse	1 for false
comparison	c_eqi2	equality on two integers; 1 if true, 0 if false
	c_lti2	less-than on two integers; 1 if true, 0 if false
	c_eqc2	equality on two characters; 1 if true, 0 if false
	c_ltc2	less-than on two characters; 1 if true, 0 if false
	c_eqr2	equality on two reals; 1 if true, 0 if false
	c_ltr2	less-than on two reals; 1 if true, 0 if false

Figure 2: Implementation Primitives (part 1)

category	*function name*	*semantics*
constructors	c_mkcons	2-vector to cons
	c_mkutype	2-vector to tagged pair
	c_mkclos	vector to closure
	c_mkbigint	vector to bigint
	cons	
selectors	c_utype_tag	returns tag of tagged pair
	c_utype_val	returns value of tagged pair
	c_cons_car	returns car of cons cell
	c_cons_cdr	returns cdr of cons cell
	c_clos_fn	returns function of closure
	c_clos_num	returns num of args expected in closure
	c_clos_togo	return num of args to go
	c_clos_sel	selects *i*th arg in closure
	c_bigint_len	selects size of bigint
	c_bigint_sel	selects *i*th component of bigint
	len	
	sel	
	s:i	curried sel
	al	
	ar	
conversions	c_char2int	character to integer
	c_int2char	integer to character
	c_seq2vect	sequence (nested cons cells) to vector
	c_vect2seq	vector to sequence (nested cons cells)
control	c_prog	block of statements
	c_func	user-defined function body
	c_for	for loop, produces a sequence
	c_vfor	for loop, produces a vector
	c_reduce	reduction loop
	p → q; r	conditional
	c_catch	catch
	signal	signal
	c_push	save arguments on a stack
	c_pop	restore arguments from a stack
	c_apply	apply a function to some arguments
miscellaneous	c_io	I/O interface
	gamma	suppress I/O
	id	
	K	
	apply	

Figure 3: Implementation Primitives (part 2)

$$
\begin{aligned}
\texttt{c\_addi2:}\langle \texttt{i}_1, \texttt{i}_2 \rangle &= \texttt{i}_1 + \texttt{i}_2 \\
\texttt{c\_muli2:}\langle \texttt{i}_1, \texttt{i}_2 \rangle &= \texttt{i}_1 * \texttt{i}_2 \\
\texttt{c\_subi2:}\langle \texttt{i}_1, \texttt{i}_2 \rangle &= \texttt{i}_1 - \texttt{i}_2 \\
\texttt{c\_divi2:}\langle \texttt{i}_1, \texttt{i}_2 \rangle &= \texttt{i}_1 \texttt{divi}_2 \\
\texttt{c\_addr2:}\langle \texttt{r}_1, \texttt{r}_2 \rangle &= \texttt{r}_1 + \texttt{r}_2 \ \texttt{or} \ \textit{exception?} \\
\texttt{c\_mulr2:}\langle \texttt{r}_1, \texttt{r}_2 \rangle &= \texttt{r}_1 * \texttt{r}_2 \ \texttt{or} \ \textit{exception?} \\
\texttt{c\_subr2:}\langle \texttt{r}_1, \texttt{r}_2 \rangle &= \texttt{r}_1 - \texttt{r}_2 \ \texttt{or} \ \textit{exception?} \\
\texttt{c\_divr2:}\langle \texttt{r}_1, \texttt{r}_2 \rangle &= \texttt{r}_1 / \texttt{r}_2 \ \texttt{or} \ \textit{exception?} \\
\texttt{c\_gettag:x} &= \texttt{tag bits of pointer or immediate as int} \\
\texttt{c\_isnull} &= \texttt{1 if x} \prec \texttt{null}, \ldots \succ, \texttt{0 otherwise} \\
\texttt{c\_istrue} &= \texttt{1 if x} = \texttt{true}, \texttt{0 otherwise} \\
\texttt{c\_isfalse} &= \texttt{1 if x} = \texttt{false}, \texttt{0 otherwise} \\
\texttt{c\_floor:r} &= \max \texttt{i s.t. i} \leq \texttt{r} \\
\texttt{c\_ceiling:r} &= \min \texttt{i s.t. i} \geq \texttt{r} \\
\texttt{c\_float:i} &= \texttt{r s.t. r``=''i} \\
\texttt{c\_eqi2:}\langle \texttt{i}_1, \texttt{i}_2 \rangle &= \begin{cases} \texttt{1} & \text{if } \texttt{i}_1 = \texttt{i}_2 \\ \texttt{0} & \text{otherwise} \end{cases} \\
\texttt{c\_lti2:}\langle \texttt{i}_1, \texttt{i}_2 \rangle &= \begin{cases} \texttt{1} & \text{if } \texttt{i}_1 < \texttt{i}_2 \\ \texttt{0} & \text{otherwise} \end{cases} \\
\texttt{c\_eqc2:}\langle \texttt{c}_1, \texttt{c}_2 \rangle &= \begin{cases} \texttt{1} & \text{if } \texttt{c}_1 = \texttt{c}_2 \\ \texttt{0} & \text{otherwise} \end{cases} \\
\texttt{c\_ltc2:}\langle \texttt{c}_1, \texttt{c}_2 \rangle &= \begin{cases} \texttt{1} & \text{if } \texttt{c}_1 < \texttt{c}_2 \\ \texttt{0} & \text{otherwise} \end{cases} \\
\texttt{c\_eqr2:}\langle \texttt{i}_1, \texttt{i}_2 \rangle &= \begin{cases} \texttt{1} & \text{if } \texttt{r}_1 = \texttt{r}_2 \\ \texttt{0} & \text{otherwise} \end{cases} \\
\texttt{c\_ltr2:}\langle \texttt{r}_1, \texttt{r}_2 \rangle &= \begin{cases} \texttt{1} & \text{if } \texttt{r}_1 < \texttt{r}_2 \\ \texttt{0} & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4: Semantics of implementation primitives (part 1)

$$
\begin{aligned}
\texttt{c\_mkcons:}\langle x, y\rangle &= \prec \texttt{cons}, \langle x, y\rangle \succ \\
\texttt{c\_mkutype :}\langle x, y\rangle &= \prec \texttt{utype}, \langle x, y\rangle \succ \\
\texttt{c\_mkclos:}\langle \texttt{fn}, j, i, x_1, \ldots, x_k\rangle &= \prec \texttt{clos}, \langle \texttt{fn}, j, i, x_1, \ldots, x_k\rangle \succ \\
\texttt{c\_mkbigint:}\langle n, x_1, \ldots, x_n\rangle &= \prec \texttt{bigint}, \langle n, x_1, \ldots, x_n\rangle \succ \\
\texttt{c\_utype\_tag:}\prec \texttt{utype}, \langle x, y\rangle \succ &= x \\
\texttt{c\_utype\_val:}\prec \texttt{utype}, \langle x, y\rangle \succ &= y \\
\texttt{c\_cons\_car:}\prec \texttt{cons}, \langle x, y\rangle \succ &= x \\
\texttt{c\_cons\_cdr:}\prec \texttt{cons}, \langle x, y\rangle \succ &= y \\
\texttt{c\_clos\_fn:}\prec \texttt{clos}, \langle \texttt{fn}, j, i, x_1, \ldots, x_k\rangle \succ &= \texttt{fn} \\
\texttt{c\_clos\_num:}\prec \texttt{clos}, \langle \texttt{fn}, j, i, x_1, \ldots, x_k\rangle \succ &= j \\
\texttt{c\_clos\_togo:}\prec \texttt{clos}, \langle \texttt{fn}, j, i, x_1, \ldots, x_k\rangle \succ &= i \\
\texttt{c\_clos\_sel:}\langle h, \prec \texttt{clos}, \langle \texttt{fn}, j, i, x_1, \ldots, x_k\rangle \succ\rangle &= x_h \\
\texttt{c\_bigint\_len:}\prec \texttt{bigint}, \langle n, x_1, \ldots, x_n\rangle \succ &= n \\
\texttt{c\_bigint\_sel:}\langle h, \prec \texttt{bigint}, \langle n, x_1, \ldots, x_n\rangle \succ\rangle &= x_h \\
\texttt{len:}\langle x_1, \ldots, x_n\rangle &= n \\
\texttt{sel:}\langle h, \langle x_1, \ldots, x_n\rangle\rangle &= x_h \\
\texttt{s:i:}\langle x_1, \ldots, x_n\rangle &= x_i \\
\texttt{al:}\langle x_0, \langle x_1, \ldots, x_n\rangle\rangle &= \langle x_0, x_1, \ldots, x_n\rangle \\
\texttt{ar:}\langle\langle x_1, \ldots, x_n\rangle, x_0\rangle &= \langle x_1, \ldots, x_n, x_0\rangle \\
\texttt{c\_char2int:}c &= \texttt{code}(c) \\
\texttt{c\_int2char:code}(c) &= c \\
\texttt{c\_seq2vect:}\prec \texttt{cons}, \langle x_1, \ldots \prec \texttt{cons}, \langle x_n, \texttt{NULL}\rangle \succ \ldots\rangle \succ &= \langle x_1, \ldots, x_n\rangle \\
\texttt{c\_vect2seq:}\langle x_1, \ldots, x_n\rangle \succ &= \prec \texttt{cons}, \langle x_1, \ldots \prec \texttt{cons}, \langle x_n, \texttt{NULL}\rangle \succ \ldots\rangle \succ \\
\texttt{id:}x &= x \\
\texttt{K:}x\texttt{:}y &= x \\
\texttt{apply:}\langle f, x\rangle &= f\texttt{:}x
\end{aligned}
$$

Figure 5: Semantics of implementation primitives (part 2)

equation, $\perp_s$ is the empty store:

$$\mathtt{c\_func}\langle\langle i_1,\ldots,i_n\rangle,i_r,x_1,\ldots,x_m\rangle{:}\langle v_1,\ldots,v_n\rangle = s(i_r)\ \mathtt{where}\ s = \mathtt{c\_prog}{:}\langle x_1,\ldots,x_n\rangle{:}\perp_s [i_j \leftarrow v_j]$$

The definitions of the other control constructs:

$$\begin{aligned}
\mathtt{c_cond}{:}\langle i,q,r\rangle{:}s &= \begin{cases} q{:}s' & \text{if } s(i)=1 \\ r{:}s' & \text{if } s(i)=0 \end{cases} \\
\mathtt{c_for}{:}\langle \mathtt{ind},\mathtt{lim},\mathtt{ith},\mathtt{res},\mathtt{body}\rangle{:}s_0 &= s'\ \mathtt{where} \\
s_i &= \mathtt{body}{:}s_{i-1}[\mathtt{ind}\leftarrow i]\ \mathtt{for}\ 1\le i\le s_0(\mathtt{lim}) \\
s' &= s_{s_0(\mathtt{lim})}[\mathtt{res}\leftarrow\langle s_1(\mathtt{ith}),\ldots,s_{s_0(\mathtt{lim})}(\mathtt{ith})\rangle] \\
\mathtt{c_reduce}{:}\langle \mathtt{ind},\mathtt{lim},\mathtt{acc},\mathtt{init},\mathtt{ith},\mathtt{res},\mathtt{body}\rangle{:}s &= s'\ \mathtt{where} \\
s_0 &= s[\mathtt{acc}\leftarrow s(\mathtt{init})] \\
s_i &= \mathtt{body}{:}s_{i-1}[\mathtt{acc}\leftarrow \mathtt{body}{:}s_{i-1}(\mathtt{ith})]\ \mathtt{for}\ 1\le i\le s_0(\mathtt{lim}) \\
s' &= s_{s_0(\mathtt{lim})}[\mathtt{res}\leftarrow s_{s_0(\mathtt{lim})}(\mathtt{ith})] \\
\mathtt{c_signal}{:}s &= \prec \mathtt{exc},s\succ \\
\mathtt{c_catch}{:}\langle x,y\rangle{:}s &= \begin{cases} y{:}s' & \text{if } x{:}s = \prec \mathtt{exc},s'\succ \\ x{:}s & \text{otherwise} \end{cases} \\
\mathtt{c_push}{:}\langle x,i_1,\ldots,i_j\rangle{:}s &= \mathtt{c_push}{:}\langle x,i_2,\ldots,i_j\rangle{:}s[x\leftarrow\langle i_1,s(x)\rangle] \\
\mathtt{c_push}{:}\langle x_i\rangle{:}s &= s \\
\mathtt{c_pop}{:}\langle x,i_1,\ldots,i_j\rangle{:}s &= \mathtt{c_pop}{:}\langle x,i_2,\ldots,i_j\rangle{:}s[x\leftarrow s(y);i_1\leftarrow z]\text{if } s(x)=\langle z,y\rangle \\
\mathtt{c_pop}{:}\langle x\rangle{:}s &= s \\
\mathtt{fn}{:}\langle i_1,\ldots,i_n\rangle{:}s &= \mathtt{fn}{:}\langle s(i_1),\ldots,s(i_n)\rangle
\end{aligned}$$

For the target language, we make the following restrictions. In an assignment, the function is always the name of an implementation primitive. A control statement is always one of: **c_func, c_prog, c_for, c_reduce, c_cond, c_catch, c_signal, c_pop, c_push.** Furthermore, **c_func** may appear only on the immediate right-hand side of a function definition. This scheme models a simple abstract machine with global memory. The resulting code is very close to C.

Here is a simple example using this scheme. Function f adds two numbers and multiplies the result by the value produced by another function g. Every function takes the stack pointer as an extra argument. By convention, this is the first argument. Before any function call, local variables are pushed onto the stack, and after the function call local variables are restored from the stack.

```
def f ≡ c_func:⟨⟨0,8,9⟩,
    1,
    ⟨10,c_addi2,9,8⟩,
```

```
    c_push: ⟨0, 8, 9, 10⟩
    ⟨11, g, 10⟩
    c_pop: ⟨0, 10, 9, 8⟩
    ⟨1, c_muli2, 10, 11⟩
⟩
```

The algorithm that maps FL to this target language is the subject of Section 8.

6.1 Notes

Note that the function **c_reduce** does a copy of its last **ith** to the result location **res** . This inefficiency arises because **c_reduce** is introduced by just substituting for existing **reduceb** constructs introduced by the prog mechanism. This inefficiency should be eliminated by any decent C compiler. If we ever go to native code, we can eliminate it ourselves by doing some renaming.

We are as yet undecided on whether to implement bigints. We will implement exception checking for integer arithmetic, which will be slow. If we were generating native code, we could just check the condition register after the operation (2 or 3 instructions). We are punting on floating point arithmetic exceptions.

7 Implementing PF

We need an environment of functions that defines the functions in PF in terms of the implementation primitives. These primitives do not have to be written at the level of the target language (i.e., **c_prog**); we can map them to the target language just like everything else. For now, we just show one example, implementing **mul** (without bigints).

```
def intbits ≡ ~0

def floatbits ≡ ~2

def consbits ≡ ~4

def isint ≡ c_eq2i ∘ [intbits, c_gettag]

def isfloat ≡ c_eq2i ∘ [floatbits, c_gettag]

def iscons ≡ c_eq2i ∘ [consbits, c_gettag]

def pfmul ≡ iscons → pfmuli ∘ [id, ~1, id]; pfmulerr

def pfmuli ≡
    c_isnull ∘ s1 → s2;
    isint ∘ c_car ∘ s1 → pfmuli ∘ [c_cdr ∘ s1, c_mul2i ∘ [c_car ∘ s1, s2], s3];
    isfloat ∘ c_car ∘ s1 → pfmulr ∘ [s1, c_float ∘ s2, s3];
    pfmulerr ∘ s3

def pfmulr ≡
    c_isnull ∘ s1 → s2; isint ∘ c_car ∘ s1 → pfmulr ∘ [c_cdr ∘ s1, c_mul2r ∘ [c_float ∘ c_car ∘ s1, s2], s3];
    isfloat ∘ c_car ∘ s1 → pfmulr ∘ [c_cdr ∘ s1, c_mul2r ∘ [c_car ∘ s1, s2], s3];
    pfmulerr ∘ s3

def pfmulerr ≡ c_mkutype ∘ [~0, c_mkcons ∘ [~R(''mul''), c_mkcons ∘ [~R(''arg1''), c_mkcons ∘ [id, ~NULL]]]]
```

7.1 Notes

PF primitives should be written at this level so that the rewriter can work with them. Once all of the code is to this stage, the rewriter can inline and do optimizations, so long as it maintains the invariants that no tagged objects (or functions that produce tagged objects) are introduced.

The PF functions should be stored as a library. Depending on the level of optimization desired, they can be either inlined or not.

8 Mapping to the Target Language

After representations are made explicit, we (possibly) add the PF definitions (Section 7) and prog everything. This gives a program where the constituent functions map general FL values to values. We then substitute **c_for** for **forb**, **c_reduce** for **reduceb**, etc., add the save-return code for applications of user-defined functions, and the result is a program that is easily translated to C.

The program feature ("prog") is documented elsewhere. For our purposes, all we need is a translation from the output of the prog phase into the c_prog constructs.

For brevity, we use x instead of a location that holds x in cprogs. For example

$$\texttt{c\_prog:}\langle\langle \texttt{c\_vect\_sel}, \tilde{}\texttt{1}, \texttt{3}\rangle\rangle$$

is shorthand for

$$\texttt{c\_prog:}\langle\langle \texttt{1}, \tilde{}\texttt{1}\rangle, \langle \texttt{c\_vect\_sel}, \texttt{1}, \texttt{3}\rangle\rangle$$

where "1" is a location not used elsewhere.

The first translation is for the top-level prog of a function definition. In the new program, the function **f** uses the standard interface of taking a single argument. The function **fastf** is used by those calls that can be untupled statically. Note that the runtime stack is included as an explicit argument.

$$\texttt{H(def f} \equiv \texttt{prog:}\langle\langle \texttt{i}_{\texttt{f}_1}, \ldots, \texttt{i}_{\texttt{f}_n}\rangle, \sigma\texttt{:r}_\texttt{f}, \langle \ldots, \langle \texttt{k}_\texttt{j}, \texttt{e}_\texttt{j}\rangle, \ldots\rangle\rangle\texttt{)}$$

$$\Rightarrow$$

$$\begin{aligned}
&\texttt{def f} \equiv \texttt{c_func:}\langle\langle \texttt{stack}, \texttt{arg}\rangle,\\
&\qquad \texttt{r}_\texttt{f},\\
&\qquad \langle \texttt{i}_{\texttt{f}_1}, \texttt{c_vect_sel}, \tilde{}\texttt{1}, \texttt{arg}\rangle,\\
&\qquad \ldots,\\
&\qquad \langle \texttt{i}_{\texttt{f}_n}, \texttt{c_vect_sel}, \tilde{}\texttt{n}, \texttt{arg}\rangle,\\
&\qquad \langle \texttt{r}_\texttt{f}, \texttt{fastf}, \texttt{stack}, \texttt{i}_{\texttt{f}_1}, \ldots, \texttt{i}_{\texttt{f}_n}\rangle\rangle
\end{aligned}$$

$$\texttt{def fastf} \equiv \texttt{c\_func:}\langle\langle \texttt{stack}, \texttt{i}_{\texttt{f}_1}, \ldots, \texttt{i}_{\texttt{f}_n}\rangle, \texttt{r}_\texttt{f}, \ldots, \texttt{H(k}_\texttt{j}, \texttt{e}_\texttt{j}\texttt{)}, \ldots\rangle$$

The function H takes a location and a prog expression and translates it into a **c_prog** expression. The location is the store address where the result of the prog expression should be stored. The next several translations are straightforward.

$$\begin{aligned}
\texttt{H(k,}\sigma\texttt{:i} \to \texttt{L;R)} &\Rightarrow \texttt{i} \to \texttt{H(k,L);H(k,R)}\\
\texttt{H(k, prog:}\langle -\texttt{i},\sigma\texttt{:j},\langle \ldots,\langle \texttt{k}_\texttt{h}, \texttt{e}_\texttt{h}\rangle, \ldots\rangle\rangle\texttt{)} &\Rightarrow \texttt{c_prog:}\langle \ldots, \texttt{H(}\langle \texttt{k}_\texttt{h}, \texttt{e}_\texttt{h}\rangle\texttt{)}, \ldots, \langle \texttt{k, id, j}\rangle\rangle\\
\texttt{H(k, forb:}\langle \texttt{ind},\sigma\texttt{:l},\sigma\texttt{:ith},\langle \ldots,\langle \texttt{k}_\texttt{h}, \texttt{e}_\texttt{h}\rangle, \ldots\rangle\rangle &\Rightarrow \texttt{c_for:}\langle \texttt{ind, l, ith, k,}\langle \ldots, \texttt{H(k}_\texttt{h}\texttt{, e}_\texttt{h}\texttt{)}, \ldots\rangle\rangle\\
\texttt{H(k, vforb:}\langle \texttt{ind},\sigma\texttt{:l},\sigma\texttt{:ith},\langle \ldots,\langle \texttt{k}_\texttt{h}, \texttt{e}_\texttt{h}\rangle, \ldots\rangle\rangle &\Rightarrow \texttt{c_vfor:}\langle \texttt{ind, l, ith, k,}\langle \ldots, \texttt{H(k}_\texttt{h}\texttt{, e}_\texttt{h}\texttt{)}, \ldots\rangle\rangle\\
\texttt{H(k, reduceb:}\langle \texttt{ind},\sigma\texttt{:l, acc},\sigma\texttt{:init},\sigma\texttt{:ith},\langle \ldots,\langle \texttt{k}_\texttt{h}, \texttt{e}_\texttt{h}\rangle, \ldots\rangle\rangle &\Rightarrow \texttt{c_reduceb:}\langle \texttt{ind, l, acc, init, ith, k,}\langle \ldots, \texttt{H(k}_\texttt{h}\texttt{,}
\end{aligned}$$

Note that a translation is given for **prog** only in the case where the input variable is dead (a negative number). This is necessary for the translation to succeed; otherwise, the **prog** is an unnamed function that can be called with some unknown argument.

The final case is function application. The translation for an implementation primitive call (except apply) is easy.

$$\mathtt{H}(\mathtt{k}, \mathtt{app}{:}\langle \mathtt{pfn}, \sigma{:}\mathtt{i}_1, \ldots, \sigma{:}\mathtt{i}_n\rangle) \Rightarrow \langle \mathtt{k}, \mathtt{pfn}, \mathtt{i}_1, \ldots, \mathtt{i}_n\rangle$$

Defined function calls are the most complex case. Let $\mathtt{local}_1, \ldots, \mathtt{local}_k$ be the set of locations used in the current defined function. (Ideally, this should be the set of locations that might be used after the function call; however, initially it is easier to just save every local variable.) The function call must push these variables on the stack. The function is invoked, and upon return the stack is popped and the temporaries restored.

$\mathtt{H}(\mathtt{k}, \mathtt{app}{:}\langle \mathtt{f}, \sigma{:}\mathtt{i}_1, \ldots, \sigma{:}\mathtt{i}_n\rangle$

$\Rightarrow$

$\mathtt{c\_push}{:}\langle \mathtt{stack}, \mathtt{local}_1, \ldots, \mathtt{local}_h\rangle,$
$\langle \mathtt{k}, \mathtt{f}, \mathtt{stack}, \mathtt{i}_1, \ldots, \mathtt{i}_n\rangle,$
$\mathtt{c\_pop}{:}\langle \mathtt{stack}, \mathtt{local}_h, \ldots, \mathtt{local}_1\rangle$

8.1 Notes

We have to decide how to handle immediate values (i.e., is $\langle \mathtt{2}, \mathtt{c\_addi2}, \mathtt{3}, \mathtt{\tilde{}1}\rangle$ OK?).

This scheme does nothing about tail recursion; we should optimize mutually recursive functions with all static calling sites later. This could be expressed at the FL level by adding a goto construct and labels to **c_prog**.

We need to make sure that in the output of prog the argument variables are all dead except for the outermost prog of a defined function. This can be done easily by adding new function definitions to replace such progs before performing the translation to c_prog.

9 Code Generation

This is the most C specific part of the design. In fact, except for our general function call mechanism (**c_func**) up to this point we could just as well be targeting native code.

Each FL user-defined function is compiled to a C function. The translation is not difficult, but hasn't been spelled out.

10 Separate Compilation

We need a mechanism for separate compilation because we want to use precompiled libraries. To handle separate compilation, we need:

1. A mechanism for compiling environments of functions.
2. A name resolution mechanism for functions from different libraries (suggestion: flat namespace of libraries, function name within a library qualified by name of library.)
3. Some "linker" code to correctly map names in an FL program to names in the library.

10.1 Notes

We need to think carefully about the details of this scheme. For instance, does the C linker do what we expect for separately compiled libraries?

11 Garbage Collection

We use a standard stop-and-copy garbage collection algorithm. The initial set of items to be collected is the set of global variables in the C program. The GC algorithm does a search of the pointer structure, moving items from old space to new space. When an item is moved, a bit is set in another array "moved" to indicate that that address has been moved and a forwarding pointer is left at that address. The following pseudo-code conveys the idea.

```
function GC ()
    ∀i moved [i] ← false
    for each variable v in the program do
        collect (v)

function collect (x)
    if x is a small integer or special then
        copy x to next address a in newspace
        return (a)
    else
        (* x is a pointer *)
        for each component x_i of x do
            a_i ← collect (x_i)
        copy ⟨x_1, ..., x_n⟩ to next address a in newspace
        return (a)
```

There are a couple of inaccuracies in the pseudo-code. First, when a pointer is collected, the tuple (or floating point number) that it points to must be allocated at an address ending in "100" for the tagging scheme to work. Second, the value returned by *collect* should not be the actual pointer, but rather the pointer with the tag bits appropriately encoded.

11.1 Notes

We have to decide when the check is done to see if a GC is required. We should be able to limit the checks to the top of defined functions, fors, and reduces. The check will see if there is enough space left to allocate everything that might be allocated before the next defined function call, for, or reduce.

The need for tag bits is driven entirely by the need of the garbage collector to identify the type of structure it is examining. We should keep thinking about alternatives (see Section 13).

12 I/O

Very few ideas at this point. We obviously must use UNIX I/O facilities. Beyond that, it would be nice if someone spent some time thinking about what I/O should look like in FL. We will need an FL printer and pretty-printer; these could be written in FL.

13 Extensions

This implementation should serve as a "base system." Hopefully, it will be relatively easy to make modifications and try out other, more radical, designs. Some possibilities are:

- different sequence representations
- exotic garbage collection schemes
- generating native code (requires a register allocator)