1 Introduction

Functional languages are gaining support throughout the academic world due to their strong mathematical basis and amenity to proof. Many different languages are available — for example, Miranda[1], Hope, Ponder, Pebble, ML, SML, and Haskell — all based on the lambda calculus. Given their common basis, an intermediate language can be designed to meet the following objectives:

- Span a large portion of the spectrum between a users program and the machine it is running on thus relieving a large part of the compilation burden for new (experimental) languages.
- Provide a standard intermediate representation for portability and to facilitate programming environments which handle several languages.
- Provide an intermediate compiler which may be proven correct with the view to future "total systems correctness. (The focus of current research in the U of C's VLSI group is towards proving the correctness of microchip designs. In conjunction with a proven compiler, this would be a major step towards a system in which both hardware and software are verified).
- Provide a simple yet expressive language with which to do research into functional language issues. It should provide mechanisms to assist research without the complexities of a high level language.

In order to define this intermediate functional language, the complete semantics must be worked out so that the meaning of each statement is well defined. As well, it is possible to find semantics for the translation of the intermediate language, which will tie in well with current research in complete system correctness.

2 Background

Following are some justifications for using functional languages:

- Functional languages are all based on the lambda calculus. By definition there exists an intermediate language for them — the lambda calculus. Further, the expressive power of the lambda calculus is well known.
- The semantics of the lambda calculus have been completely defined and are relatively easy. Thus the semantics of an intermediate language (which will be based on lambda) will benefit.
- High level functional languages and the lambda calculus are amenable to proof. In particular, the pattern matching explained below matches the form of proofs done by *structural induction*. As well, since no expression in a functional language can cause a side effect, these types of complications are removed from proofs.
- Large amounts of recent research has looked at functional languages and machines for them. The direct correlation between functional languages and functional architectures is mathematically mean in that the propoerties of the lambda calculus span the boundary. Current architectures for functional languages tend to be slow, but faster machines are on the horizon. The Church Rosser property holds for the lambda calculus, and thus parallel implementations are also possible.

2.1 Intermediate Functional Languages

Current intermediate functional languages (for. example [6],[16]) can be characterized as.

- They exist at the level of the typed lambda calculus with lots of builtin functions and very few "higher level structures. This means.

[1] Miranda is a trademark of Research Software Ltd.

  - The languages cannot themselves be typechecked but they rely on well-typed expressions. That is, the only expressions which have meanings in these languages are those which have been typechecked by some external agent.
  - They are difficult source languages to program in. For a researcher who needs access to low level structures but does not want to be burdened by "assembly–like" programming these languages are insufficient.

- They ignore the fact that high level functional languages are interactive. This means that environment issues must be addressed elsewhere.

But, these languages form an important part of a system in that:

- They provide the lowest level representation from which a designer would like to generate machine code. This is because the "higher" level structures they contain allow optimizations for machine code.
- They contain a complete set of primitive functions for implementing high level functional languages.
- They allow many compilation functions to be performed as source to source transformations. Performing dependency analysis, strictness analysis, lazy lambda lifting and other functions can be performed directly on the intermediate language. (These issues are discussed in [15]).

This paper deals with a design for an intermediate language at a higher level than those discussed above. This language is designed to:

- Provide a wide array of structures and primitives so that high level languages are easily translated. The composition of these structures and primitives will support high–level structures with a minimum of effort, but will not limit the generality of the high–level structures.
- Provide a type system which will be sufficient for most applications and which can be sidestepped for those applications which require a different system.
- Handle the interactive features of high level languages so that binding in the environment is well modelled.
- Use existing intermediate languages (or a slight variation of them) as a target language. In this way, the primitives from the existing languages will help to implement the structures in the new intermediate language.

The most important consideration in the design of this language is: "Which structures can be contained in the intermediate language without a loss of generality to the structures in high level languages?" The next section works through an example of a high level expression in order to motivate an answer to this question.

2.2 Introduction to Functional Expressions

Consider the following expression (in a generic high level functional language syntax):

```
datatype list *      := nil | cons * (list *)
length nil           == 0
length (cons a l)    == 1 + (length l)
```

This is the declaration of the type list * and the function length on lists. A list is either nil or it is the cons of an item of type * onto a list of type list *. The * is a type variable, meaning that the type list is polymorphic. We may have items of type list int, list (list *), list char,....

length is defined using *pattern matching* – each constructor in the type of list is given a case in the definition of the function (this corresponds well with proofs done by *structural induction* for the correctness of programs). Thus, if the argument to length is nil, the first expression will be evaluated. Otherwise, the argument must be a cons, so the second expression will be evaluated.

If the argument is not nil it must be cons since, in general, high level functional languages are statically typechecked – the type of an expression is computed at compile-time. The type of length is a function from list * to int, usually written length:list * $\rightarrow$ int. Thus, if the first defining equation for length has a parameter of type list *, the following equations must have a type at least as specific as list * (a more specific type, such as list int is also acceptable).

Programming occurs in an *interactive* environment. After each function is typed in, it is compiled or interpreted. It is important to note that several interactive commands may be given before a function is defined (as in the case of length).

In order to compile the length example, we must have some representation of the type list *. It is sufficient to have a representation for both nil and cons since together they totally define list:

```
nil               ==> (list,0)
cons * (list *)   ==> (list,1,*,list *)
```

Thus, cons 3 nil would be represented as: (list,1,3,(list,0)). In general, types can be *packed* in this manner, and then specific parts of the type may be *selected* at will.

Compiling the length function will require a function to select an expression based on which parameter matches the argument. Given the representation of list this would look like:

```
length n == if n = (list,0) then 1 else
            if n = (list,1,a,l) then 1 + (length l)
            else error
```

In this case error will never be encountered (since length was typechecked), but incomplete pattern matches are possible. This definition contains no pattern matching – the matching has been replaced by equality checks on the representation of lists. This process is known as *pattern matching compilation.*

The next step in compiling length is to compile it into the lambda calculus, from which the back end is generated. These steps are completely covered in [15].

2.3 Characteristics of Functional Languages

A review of the ideas touched on above and an introduction to some other characteristics of functional languages is given below:

- *Data types and pattern matching* as introduced above.
- *Strongly typed* expressions are the only ones accepted. Types are *polymorphic* and may be *explicit* (supplied by the user) or *implicit* (inferred by the compiler). Explicit systems allow more general types than implicit ones, but lack the user friendliness that inferring types provides. Most widely used languages (ie. Miranda and SML) have implicit type systems although considerable recent research has been into explicit systems (for example, [4]).
- *List comprehensions* are akin to the declaration of mathematical sets. For example,

  ```
  [n * n | n < 100]
  ```

  is the list containing the squares of all the integers less than 100. These structures allow for very succinct clear programming. Consider this quicksort from [15]:

  ```
  qs nil == nil
  qs (cons a l) == cons (qs [b | mem b l; b <= a]) ( cons a (qs [b | mem b l; b > a]))
  ```

  where mem is a function which returns true if its first argument is included in the list which is its second argument.

- *Modules* are user friendly and useful when programming large systems. SML [9] and Pebble [1] are two functional languages which take radically different approaches to modules. Pebble treats modules as values and thus types become first class objects. SML maintains the ability to infer types by raising a strict distinction between functions and modules. Both these systems can be viewed as experimental since neither has been widely accepted yet. Other functional languages, such as Miranda and Haskell do not contain modules.

- *Arrays* are desirable in a functional language if sharing analysis is done [11], since they provide constant access time. With no sharing analysis each update of an array would require making a copy of the entire array.

- *Exceptions* allow errors to be flagged so that functions can recover. For example:

  ```
  exception div0;
  div x 0 == raise div0
  div x y == x / y;
  ```

  The exception div0 can then be *handled* by functions which wish to provide a default value or print out an error message.

- *Lazy evaluation* ensures that expressions which can be evaluated are. Strictness analysis allows some functions to be called by value, and thus allows parallelism. It also increases the speed of non-parallel machines [22].

3 An Intermediate Functional Language (IFL)

My goal is to define an intermediate functional language, IFL, which incorporates as many of the above structures as possible without being too specialized.

The following priorities for designing and implementing a language (given in order of importance) reflect my views on the importance of having a well defined language before it is implemented:

- *Design:* The language must be designed to meet specific applications. A useful subset of programming languages must be defined. In this case I have limited myself to lazy functional languages and have outlined the major features which are necessary in the design. This subset is mathematically clean and the implementation is not overly complex. An abstract syntax must be given for the language, listing the components of the language and their relation to each other. It does not include syntactic information or precedence information. The abstract syntax is sufficient for defining the semantics of the language.

- *Semantics:* Before any implementation is done the semantics of the language must be totally defined. If this is done, all implementations should give identical results for any program. Semantics also give important feedback on the design of a language – if the semantics are complicated this is an indication that the language may be too complex. At this stage it is also worth designing a concrete syntax which specifies syntactic and precedence constraints on programs. If this is done, the language will be totally defined before it is implemented.

- *Implementation:* Once the language is totally defined, a correct implementation, one which maintains the semantics of expressions, can be devised. A proof of the correctness would be advantageous.

- *Optimization:* Any optimization which improves performance without changing the semantics can be added once the implementation is known to be correct.

The following design decisions were incorporated into IFL:

- Data types and pattern matching are the most pervasive feature of recent functional languages. A general type definition and pattern matching system, akin to that of Miranda, ML, SML, Hope, and others has been included.

- Typechecking will be the most general implicit system based on the one developed by Milner [14] and outlined in [15]. Explicit types will be allowed through annotation so that the user has control of types, if he so desires. This also means that more powerful explicit types can be given to IFL expressions, sidestepping the inferencing algorithm.

- The interactive environment will be explicit in IFL. There is a distinction between commands which cause bindings and those which evaluate to some value.

- Modules are not yet included due to the lack of agreement as to their definition. A semantic description of modules is being worked on, and primitives for implementing them may be added later. If, in the meantime, some agreement as to module definition is arrived at, this will be incorporated. (Semantically, a module is simply a set of bindings to be added or saved from the environment. The problem is in enforcing types between modules in a general manner. So far the two solutions to this problem are exemplified by SML and Pebble as discussed earlier).

- List comprehensions can be transformed into iteration over lists. With the appropriate functions defined in the original environment for IFL, transformations for list comprehensions will be easy. This will allow a very general class of list comprehensions, whereas defining a list comprehension structure directly in IFL may lead to some limitations.

- Exceptions are included in IFL by using the builtin operator fail and lazy evaluation. (The failures which are simulated are not as general as those in SML, but SML is not purely functional and uses some imperative features in the implementation of its exceptions. The method of translating exceptions into a functional language is given in [21]).

- Arrays will not be included in the first version of IFL although it is expected that they will be included in the final version. (This is because sharing analysis is a difficult topic which would simply complicate the original version).

- Lazy evaluation with strictness analysis will be the evaluation mechanisms for IFL.

3.1 Abstract Syntax

The BNF for IFL has been organized in order to highlight the difference between commands which evaluate, exp, and those which cause bindings in the environment, bind. In the definition in Figure 1, x^+ means one or more occurrences of x, x^* means zero or more occurrences of x.

3.2 Informal Description of IFL

An ifl term can be an expression, exp, which evaluates to a value in a given environment, or a binding, bind, which enriches the environment for subsequent evaluations, or a series of these two, ifl*.

A pattern is a match for a builtin type. Examples of patterns will be shown as we progress. Constants include integer operations, real operations (prefixed by r), boolean operations, and fail.

A binding is one of the following:

- A *let* binding causes each variable in the pattern on the left-hand side to be bound to the corresponding value on the right-hand side. Multiple bindings are done in parallel. For example:

  let x = 3, y = 4

  binds 3 to x and 4 to y. But,

  let x = 3, y = (+ x 1)

  binds 3 to x and produces an error (if x was not previously defined) for the y binding. If an error occurs in any part of a binding, none of the variables become bound.

ifl	::=	bind | exp | ifl$^+$
bind	::=	letbind
	|	*lettype* tyname = type,...,tyname = type
	|	*abstype* tyname = type,...,tyname = type *in* letbind
letbind	::=	*let* pattern = exp,...,pattern = exp
	|	*letrec* pattern = exp,...,pattern = exp
exp	::=	var | const | exp exp | annot exp | number | string
	|	λ pattern$^+$.exp
	|	*let* pattern = exp,...,pattern = exp *in* exp
	|	*letrec* pattern = exp,...,pattern = exp *in* exp
	|	*case* var *of* pattern => exp,...,pattern => exp
	|	□ exp exp
	|	(exp)
pattern	::=	const pattern* | var | constructor pattern*
tyname	::=	var | constructor tyname*
type	::=	tyname | knowntype | var | constructor type*
constructor	::=	*sum* | *prod* | *fun* | var
knowntype	::=	*int* | *bool* | *char* | *string* | *real* | tyname
annot	::=	var | var => exp ... var => exp | type
const	::=	+ | − | ∗ | / | *r+* | *r−* | *r∗* | *r/*
	|	*if* | *or* | *and* | *xor* | *true* | *false* | *fail*
	|	num | string
var	::=	*alphabetic alphanumeric**
num	::=	digit$^+$ | -num | num.num | num e num
digit	::=	0..9
string	::=	"*alphanumeric**"

Figure 1: Abstract Syntax for IFL

- A *letrec* binding works in the same manner as the *let* except that definitions may be recursive or mutually recursive. For example:

  letrec f = λx. *if* (= x 0) 1 (+ x (f (- x 1)))

- A *lettype* binding declares a new type. An enumerated type is simply the sum of the possible values.

  lettype colors = *sum* (*sum* (green,blue), red)

  A data type can be more complicated and may contain type variables.

  lettype list a = *sum* (nil, cons a (list a))
  lettype pair = *prod* (a,b)

  When more than one type definition is given they are assumed to be dependent on each other.

- A *abstype* binding binds the operators for an abstract type. For example, a stack could be defined as:

  abstype stack a = *sum* (nils, element a (stack a)) *in*
      let push = λ a s. element a s, pop = λ (element a s). a

An exp can be a variable, a constant, an application, a number, a string or one of the following:

- A lambda expression is a function from one or more patterns to the result of the body.

  let head = λ (cons x y). x
  let tail = λ (cons x y). y

  Here are the first examples of patterns in IFL. The cons x y matches our builtin type list. Thus, head could be applied to any list which is not nil.

- A *let* expression executes the body of the *let* in the environment given by the *let* binding. The difference between the binding and the expression is that the binding is permanent. This also holds for the *letrec* expression.

- The *case* expression allows pattern matching on sum types.

  let head = λx. *case* x *of* (cons a b) => a, nil => *fail*
  let tail = λx. *case* x *of* (cons a b) => b, nil => *fail*

  These would be more accurate definitions of head and tail.

- The □ expression executes the first expression. If this evaluates, the second expression is ignored. If the first expression fails, the result of the second is returned.

  let tail2 = λx. □ (tail x) nil

- Annotated expressions allow extra information to be given to the compiler. At the present time annotations may contain strictness and typing information. In the future, sharing, concurrency, and other annotations may also be added.

3.3 Types

The number of different type systems which have been developed is immense (see [18] or [3] for a review of those of interest here). The Milner type system is one of the most widely used since there is a well known algorithm for inferring the types of expressions. The type system I develop here is an extension of the Milner system to IFL with many ideas from the papers cited above.

The usual method for specifying type inference is through a series of inference rules. The *premisses* consist of a *type assignment* which maps (possibly free) identifiers onto types. The following variables are replaced by expressions or types as appropriate to create an instance of a rule.

Variable	**Meaning**
m	type assignment
$v\ v_1...v_n$	variables
$c\ c_1...c_n$	constructors
$k\ k_1...k_n$	constants
$e\ e_1...e_n$	expressions
$p\ p_1...p_n$	patterns
n	integers ($n \geq 0$)
$t\ t_1...t_n$	types
$tv_1...tv_n$	type variables

A *type scheme* is a type containing type variables. A *principle typing* of an expression e is a type scheme which can be instantiated to any valid type for e.

A substitution is given by $\delta, \delta_1, ...\delta_n$. $\delta(t)$ is the result of applying the substitution δ to the type t. $\delta(m)$ is the result of applying the substitution δ to all types assigned by m. Unification, ϑ, finds the most general substitution for two types. That is, $\vartheta(\delta, t_1, t_2) = \delta_1$ where $\delta_1(t_1) = \delta_1(t_2)$. If unification fails, the typing will fail. In Figure 2 the inference rules are given for typing IFL expressions. δm is used as a shorthand for $\delta(m)$.

Figure 3 gives two examples of how the typechecking rules are used to infer the types of expressions.

VAR $\overline{m \vdash v : t}$ when m assigns t to v

CST $\overline{\vdash k : t}$ when t is the type of k

APP

$$\frac{\begin{array}{l} \delta m \vdash e_1 : \delta(t \rightarrow t_1) \\ \delta m \vdash e_2 : \delta t' \end{array}}{\delta_1 m \vdash e_1\, e_2 : \delta_1 t_1} \text{ where } \delta_1 = \vartheta(\delta, t, t')$$

FUN

$$\frac{m, p : t, v_1 : t_1, ..., v_n : t_n \vdash e : t_{n+1}}{m \vdash \lambda p.e : t \rightarrow t_{n+1}}$$

when $v_1 ... v_n$ are the variables in the pattern p

LET

$$\frac{\begin{array}{l} m \vdash e_1 : t \\ m, p : t \vdash e_2 : t_1 \end{array}}{m \vdash let\, p = e_1\, in\, e_2 : t_1}$$

CSE

$$\frac{\begin{array}{l} \delta m, v : \delta t, p_1 : \delta t \vdash e_1 : \delta t_1 \\ ... \\ \delta m, v : \delta t, p_n : \delta t \vdash e_n : \delta t_n \end{array}}{\delta_1 m \vdash case\ v\ of\ p_1\ \Rightarrow e_1 ... p_n\ \Rightarrow e_n : \delta_1 t_1} \text{ where } \delta_1 = \vartheta(...\vartheta(\delta, t_1, t_2), ..., t_n)$$

FIX

$$\frac{\begin{array}{l} \delta m, p_1 : tv_1, ..., p_n : tv_n \vdash e_1 : \delta t_1 \\ ... \\ \delta m, p_1 : tv_1, ..., p_n : tv_n \vdash e_n : \delta t_n \\ \text{where } tv_1, ..., tv_n \text{ are new type variables not occuring in } \delta m \end{array}}{\begin{array}{l} \delta_1 m, p_1 : \delta_1 tv_1, ..., p_n : \delta_1 tv_n \vdash e_1 : \delta_1 t_1 \\ ... \\ \delta_1 m, p_1 : \delta_1 tv_1, ..., p_n : \delta_1 tv_n \vdash e_n : \delta_1 t_n \\ \text{where } \delta_1 = \vartheta(...(\vartheta(\delta, t_1, tv_1), ..., t_n, tv_n) \end{array}}$$

REC

$$\frac{m, p_1 : t_1, e_1 : t_1, ..., p_n : t_n, e_n : t_n \vdash e : t}{m \vdash letrec\, p_1 = e_1, ..., p_n = e_n\, in\, e : t}$$

FTB

$$\frac{\begin{array}{l} \delta m \vdash e_1 : \delta t_1 \\ \delta m \vdash e_2 : \delta t_2 \end{array}}{\delta_1 m \vdash \Box e_1\, e_2 : \delta_1 t_1} \text{ where } \delta_1 = \vartheta(\delta, t_1, t_2)$$

Figure 2: Type Inference Rules

Let the environment m contain:

$0 : int, 1 : int, 2 : int, ...$
$=: int \rightarrow int \rightarrow bool$
$- : int \rightarrow int \rightarrow int$
$if : bool \rightarrow tv_1 \rightarrow tv_1 \rightarrow tv_1$

Let the substitution function δ be empty. Let $e_1 = \lambda n.if\ (= n\ 0)\ 1\ (fac(-n\ 1))$. Let $FAC = letrec\ fac = e_1\ in\ (fac\ 3)$. Assume that each right-hand side becomes part of the premisses for rules following it. The typing for FAC proceeds as follows:

VAR	$\delta m, n : tv_2$	$\vdash n : tv_2$
APP	$\delta m, n : tv_2$	$\vdash (-n) : int \rightarrow int$ and $\delta_1 = (\delta\ and\ tv_2 = int)$
APP	$\delta_1 m, n : int$	$\vdash (-n\ 1) : int$
APP	$\delta_1 m, n : int$	$\vdash (= n) : int \rightarrow bool$
APP	$\delta_1 m, n : int$	$\vdash (= n\ 0) : bool$
APP	$\delta_1 m, n : int$	$\vdash (if(= n\ 0)) : tv_1 \rightarrow tv_1 \rightarrow tv_1$
APP	$\delta_1 m, n : int$	$\vdash (if(= n\ 0)\ 1) : int \rightarrow int$ and $\delta_2 = (\delta_1\ and\ tv_1 = int)$
VAR	$\delta_2 m, n : int, fac : tv_3$	$\vdash fac : tv_3$
APP	$\delta_2 m, n : int, fac : tv_3$	$\vdash (fac(-n\ 1)) : int$ and $\delta_3 = (\delta_2\ and\ tv_3 = int \rightarrow tv_4)$
APP	$\delta_3 m, n : int, fac : tv_4 \rightarrow int$	$\vdash (if\ (= n\ 0) 1 (fac(-n\ 1))) : int$
FUN	$\delta_3 m, n : int, fac : tv_4 \rightarrow int$	$\vdash e_1 : int \rightarrow int$
FIX	$\delta_3 m, fac : tv_4 \rightarrow int$	$\vdash e_1 : int \rightarrow int$ and $\delta_4 = (\delta_3\ and\ tv_4 = int)$
APP	$\delta_4 m, fac : int \rightarrow int$	$\vdash (fac\ 3) : int$
REC	$\delta_4 m$	$\vdash FAC : int$

Assume m also contains $cons : tv_1 \rightarrow (list\ tv_1) \rightarrow (list\ tv_1)$ and $nil : (list\ tv_1)$.
The typing for $letrec\ length = \lambda l.case\ l\ of\ nil => 0, (cons\ a\ x) => +1\ (length\ x)$:

APP	δm	$\vdash (+1) : int \rightarrow int$
VAR	$\delta m, a : tv_2$	$\vdash a : tv_2$
VAR	$\delta m, a : tv_2, x : tv_3$	$\vdash x : tv_3$
APP	$\delta m, a : tv_2, x : tv_3$	$\vdash (cons\ a) : (list\ tv_2) \rightarrow (list\ tv_2)$ and $\delta_1 = (\delta\ and\ tv_1 = tv_2)$
APP	$\delta_1 m, a : tv_2, x : tv_3$	$\vdash (cons\ a\ x) : list\ tv_3$ and $\delta_2 = (\delta_1\ and\ tv_2 = tv_3)$
VAR	$\delta_2 m, a : tv_3, x : tv_3, l : tv_4$	$\vdash l : tv_4$
VAR	$\delta_2 m, a : tv_3, x : tv_3,$ $l : tv_4, length : tv_5$	$\vdash length : tv_5$
APP	$\delta_2 m, a : tv_3, x : tv_3,$ $l : tv_4, length : tv_5$	$\vdash (length\ x) : tv_6$ and $\delta_3 = (\delta_2\ and\ tv_5 = tv_3 \rightarrow tv_6)$
APP	$\delta_3 m, a : tv_3, x : tv_3,$ $l : tv_4, length : tv_3 \rightarrow tv_6$	$\vdash (+1\ (length\ x) : int$ and $\delta_4 = (\delta_3\ and\ tv_6 = int)$
CSE	$\delta_4 m, l : tv_4,$ $length : tv_3 \rightarrow int$	$\vdash (case...) : int$ and $\delta_5 = (\delta_4\ and\ tv_3 = list\ tv_7)$
LAM	$\delta_5 m, l : tv_4,$ $length : list\ tv_7 \rightarrow int$	$\vdash (\lambda l.case...) : list\ tv_7 \rightarrow int$ and $\delta_6 = (\delta_5\ and\ tv_4 = list\ tv_7)$
FIX	$\delta_6 m, length : list\ tv_7 \rightarrow int$	$\vdash (\lambda l.case...) : list\ tv_7 \rightarrow int$

In this case, the last step tells us what the type of the binding of *length* is.

Figure 3: Examples of Typing

4 Examples of Compiling into IFL

It is worthwhile taking a quick look at how some programs in high level languages look in IFL. High level functional language syntaxes do not vary greatly (which is a good indicator for an intermediate language), and the reader is pointed to the references to find out the workings of a given example.

The first example is from Miranda, and is given in [20]. Figure 4 shows both the Miranda and IFL code for this example.

The second example is in HOPE, from [2]. Figure 5 shows the definition of a tree, and some functions on it. In the reference, these are bundled into a module. The module structure has been ignored in this example (since, as mentioned above), a general module structure has not yet been worked out. In HOPE types must be given explicitly, but the functions shown are easily typechecked in IFL.

```
tree * ::= niltree | node * (tree *) (tree *)

sort = flatten.build

foldr op z = g
          where
             g [] = z
             g (a:x) = op a (g x)

build = foldr insert niltree
          where
             insert b niltree = node b niltree niltree
             insert b (node a s t) = node a (insert b s) t, b <= a
                                   = node a s (insert b t), b > a

flatten niltree = []
flatten (node a s t) = flatten s ++ [a] ++ flatten t
```

MIRANDA tree sorting program

```
lettype tree a = sum (niltree, node a (tree a) (tree a))

let sort = comp flatten build

let foldr = λop z.
          letrec g = λ l. case l of nil => z, (cons a x) => op a (g x)
          in g

let build = letrec insert = λ b t.
                  case t of niltree => node b niltree niltree,
                        (node a s t) => if b <= a then node a (insert b s) t else
                                        if b > a then node a s (insert b t) else fail
                  in foldr insert niltree

letrec flatten = λ t.case t of niltree => nil,
                        (node a s t) => cons (flatten s, cons (a, flatten t))
```

IFL tree sorting program

Assume for the translation into IFL that the type list * is defined with the constructors nil and cons. As well, a composition operator, comp and the append operator **append** are defined as:

```
let comp = λ f g x. f ( g x)
let append = λ l1 l2. case l1 of nil => l2, (cons a x) => append x (cons a l2)
```

Figure 4: Example of Miranda to IFL Translation

```
data otree == empty ++ tip(num) ++ node(otree#num#otree)

dec insert: num#otree → otree
dec flatten: otree → list num

—insert (n,empty) <= tip(n)
—insert (n,tip(m)) <= n < m then node(tip(n),m,empty)
                          else node (empty,m,tip(n))
—insert (n,node(t1,m,t2)) <= n < m then node(insert (n,t1),m,t2)
                                  else node(t1,m,insert(n,t2))

—flatten (empty) <= nil
—flatten (tip(n)) <= [n]
—flatten (node(t1,n,t2)) <= flatten(t1) <> (n::flatten(t2))
```

HOPE tree functions

```
lettype otree = sum (sum (empty,tip num), node otree num otree)

letrec insert = λ(prod(n,tree)).
        case tree of empty => tip n
                (tip m) => if n < m then node (tip n) m empty
                                else node empty m (tip n)
            (node t1 m t2) => if n < m then node (insert (prod(n,t1))) m t2
                        else node t1 m (insert (prod (n,t2)))

letrec flatten = λ tree.
        case tree of empty => nil
                (tip n) => cons n nil
            (node t1 n t2) => append (flatten t1) (cons n (flatten t2))
```

IFL tree functions

Assume for the translation into IFL that the type list * is defined with the constructors nil and cons. As well, append which concatenates two lists can be defined in IFL as:

```
let append = λ l1 l2. case l1 of nil => l2, (cons a x) => append x (cons a l2)
```

Figure 5: Example of HOPE to IFL Translation

lam	::=	bind_b | exp_b| lam^+
bind_b	::=	*let* var = exp_b... var = exp_b
	|	*letrec* var = exp_b... var = exp_b
exp_b	::=	var | const_b | exp_bexp_b| annot exp_b | (exp_b) | number | string
	|	$\lambda\text{var}^+.\text{exp}_b$
	|	*let* var = exp_b... var = exp_b *in* exp_b
	|	*letrec* var = exp_b... var = exp_b *in* exp_b
const_b	::=	const | *pack* | *unpack* | ...
var	::=	*alphabetic alphanumeric**
number	::=	digit^+ | -number | number . number | number e number
string	::=	"*alphanumeric**"

Figure 6: Abstract Syntax for lam

5 An Intermediate Functional Language Compiler

In order to be of optimal use, IFL will need a compiler to transform it into the lowest reasonable representation. As shown in Peyton-Jones, many back end optimizations rely on exploiting let and letrec structures. Thus, the back end of the IFL compiler has been defined to include these structures inside supercombinator bodies (supercombinators are a restricted class of lambda expressions where lazy evaluation is preserved). The abstract syntax for the back end is given in Figure 6. Note that the interactive commands still exist at this level, although all type declarations and pattern matching have been eliminated. This language is nearly identical with all intermediate functional languages (including those discussed earlier).

The compilation proceeds as follows:

IFL

↓

Typechecking using a Milner style type discilpline.

↓

Pattern Matching compilation and removal of type declarations. This leads to the *pack*, *unpack*,... builtin functions in lam.

↓

lam

Dependency analysis, floating out *let* and *letrec* expressions, redundancy analysis, common-subexpression elimination.

↓

lam

↓

Fully lazy lambda lifting without eliminating *let* and *letrec* expressions.

↓

lam, limited to supercombinators.

↓

Strictness analysis (future analysis such as sharing).

↓

lam, limited to supercombinators.

Thus, the output will be fully annotated supercombinator definitions ready for compilation into machine code.

6 Semantic Issues

In order for IFL to be used, its semantics must be fully worked out, including the compiler semantics. Using standard denotational and algebraic semantics it is possible to show that IFL has the same semantics as the lambda calculus (with constants). This is achieved in the following manner:

```
IFL  ----------------------------> Domain of values for IFL
 |            M                              ^
 |                                           |
 | Compile                                   | rep
 |                                           |
 V            M'                             |
lam  ----------------------------> Domain of values for lam
```

Goguen el al. [8] have shown that a context-free grammar forms an initial algebra and that semantic functions from the context-free grammar are unique homomorphisms. To show that "meaning" is maintained by the Compile operation we have to show that:

rep o M' o Compile = M

which, since M and M' are unique homomorphisms, means that you must show that Compile and rep are also homomorphisms.

In order to accomplish the above, the semantics of IFL, compile, and lam must be totally defined. The meaning of an IFL term is well defined only when it is correctly typed, implying that the well-typedness of IFL terms must also be specified. This leads to a large amount of semantics.

7 Conclusion

To date, the following has been accomplished:

- IFL and lam have been defined as shown above. The idea of a high level intermediate language with a type system and a model for the environment have been developed.
- The semantics for IFL have been defined (see appendix). An intuitive and easy semantics for expressions involving abstract data types is forwarded.
- Parts of the semantics for typechecking have been defined.
- The compilation issues mentioned above have been researched, and many of them have been implemented for lam.

Further work includes:

- Completing the semantics of types and typechecking.
- Defining a concrete syntax for IFL.
- Completing an implementation of the IFL to lam translation. Many of the individual functions have been written and only need to be drawn together.
- Showing the translation from a high level language into IFL. (Hopefully the high level language will be Haskell – at this time its syntax is still being worked out, although it is hoped that the abstract syntax will be available soon [17].

- Showing the translation from lam into machine code. The TIM machine [5][17] is one of the newest machines for functional languages and would be a good target machine. As well, the SECD machine [12][10]is being used for other research at the U of C which would make it an attractive target machine.

In the future I hope to extend this work in several ways:

- Total systems correctness is a recent idea. A complete correctness proof IFL along with some new ideas on how to simplify such a task would be very useful.
- Abstract interpretation is a growing discipline which will have a profound effect on compiler optimizations and other semantic issues.
- Given the inherent parallelism of functional languages, control structures could be added to IFL as annotations and the semantics of parallelism could be worked out. Lots of work has been done in this area but it is not yet totally conquered.
- Sharing analysis, reference counting, path semantics, and other interesting areas using functional languages could be studied using IFL as a base.

8 Appendix: Semantics

The semantics for the lambda calculus are given so that my style of denotational semantics is clear. Then the semantics of lam are defined, since they parallel those of the lambda calculus. IFL is analyzed and a new approach for the semantics of abstract types is discussed. An outline of what is necessary for a type semantics is given, although this work is not yet complete.

8.1 The Lambda Calculus

The syntax of the lambda calculus with constants is given as:

```
expression ::=   variable
             |   constant
             |   expression expression
             |   λ variable. expression
             |   (expression)
```

The domain of meanings must contain all possible values, V, for expressions. Thus, we will need a set of constants, say C, and we will need functions from values to values, V → V. From Scott's work [19] we know that we can define the domain as:

$$\mathsf{V} \cong \mathsf{C} + (\mathsf{V} \rightarrow \mathsf{V})$$

with the injection function $\hookleftarrow_V$ and the projection function $\hookrightarrow_{V \rightarrow V}$. Using this, the semantics are:

$$\begin{array}{lcl}
\mathcal{L}\text{:}\{\mathsf{E} \mid \mathsf{E} \in \mathsf{expression}\} \rightarrow \mathsf{V} & & \\
\mathcal{L}\ [\![\ \textit{constant}\ ]\!]_\sigma & = & \mathsf{K}\ [\![\ \textit{constant}\ ]\!] \\
\mathcal{L}\ [\![\ \textit{variable}\ ]\!]_\sigma & = & \sigma\ [\![\ \textit{variable}\ ]\!] \\
\mathcal{L}\ [\![\ \textit{exp}_1\ \textit{exp}_2\ ]\!]_\sigma & = & (\mathcal{L}\ [\![\ \textit{exp}_1\ ]\!]_\sigma \hookrightarrow_{V\rightarrow V})\ (\mathcal{L}\ [\![\ \textit{exp}_2\ ]\!]_\sigma) \\
\mathcal{L}\ [\![\ \lambda\ \textit{var}.\ \textit{exp}\ ]\!]_\sigma & = & \lambda\ \mathsf{val}.\ \mathcal{L}\ [\![\ \textit{exp}\ ]\!]_{(\sigma\,[val/var])} \hookleftarrow_V
\end{array}$$

where exp_1 and exp_2 are lambda expressions, var is any variable, and σ [val/var] means environment σ with val associated with the identifier var. K is a function which maps syntactic constants into C.

8.2 LAM

The following syntactic domains will be used for the semantics of lam:

$$\begin{array}{lcl}
\mathsf{Lam} & = & \{\ \mathsf{L} \mid \mathsf{L} \in \mathsf{lam}\} \\
\mathsf{Exp}_b & = & \{\ \mathsf{E} \mid \mathsf{E} \in \mathsf{exp}_b\} \\
\mathsf{Bnd}_b & = & \{\ \mathsf{B} \mid \mathsf{B} \in \mathsf{bind}_b\} \\
\mathsf{Var} & = & \{\ \mathsf{V} \mid \mathsf{V} \in \mathsf{var}\}
\end{array}$$

The semantic domain is the same for both lam and IFL:

$\mathsf{V} \cong \mathsf{B}_0 + \mathsf{B}_1 + \ldots + \mathsf{F} + \mathsf{P} + \mathsf{S} + \mathsf{E}$	All Values
$\mathsf{B}_0 = \{\perp, \mathsf{true}, \mathsf{false}\}$	Booleans
$\mathsf{B}_1 = \{\perp, 0, 1, 2, \ldots\}$	Numbers
... *other flat domains of basic values*	
$\mathsf{F} = \mathsf{V} \rightarrow \mathsf{V}$	Functions
$\mathsf{P} = \mathsf{V} * \mathsf{V}$	Products
$\mathsf{S} = \mathsf{V} + \mathsf{V}$	Disjoint Unions
$\mathsf{E} = \{\perp, \mathsf{error}, \mathsf{untyped}\}$	Errors

The sub-domain E holds error information: error implies a compile time error, such as an unbound variable; untyped implies that an expression could not be typed. Each of B_i is a flat domain containing constant information.

The semantics for lam closely parallel those of the lambda calculus (I will prove here -later – that they are identical). In addition, the semantic function $\mathcal{R}_{\mathcal{B}}$ is defined to model the interactive environment. It produces a finite product of values (V * (V * (...))) for each command which evaluates. For binding commands, the environment is updated, but no value appears in the product result. If a binding fails, the old environment is still used.

Define the environment as Env: Var $\rightarrow$ V and represent the environment as σ.

$$
\begin{aligned}
&\mathcal{R}_{\mathcal{B}}\text{: Lam} \rightarrow \text{Env} \rightarrow \text{V} && \\
&\mathcal{R}_{\mathcal{B}} \llbracket \text{super;ine}_b \rrbracket \sigma && = (\mathcal{S}_{\mathcal{B}} \llbracket \text{super} \rrbracket \sigma, \mathcal{R}_{\mathcal{B}} \llbracket \text{ine}_b \rrbracket \sigma) \\
&\mathcal{R}_{\mathcal{B}} \llbracket \text{binder;ine}_b \rrbracket \sigma && = \text{if } \sigma_1 = \{\text{error}\} \text{ then } \mathcal{R}_{\mathcal{B}} \llbracket \text{ine} \rrbracket \sigma \text{ else } \mathcal{R}_{\mathcal{B}} \llbracket \text{ine} \rrbracket \sigma_1 \\
& && \qquad \text{where } \sigma_1 = \mathcal{B}_{\mathcal{B}} \llbracket \text{binder} \rrbracket \sigma
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{B}_{\mathcal{B}}\text{:Bnd}_b \rightarrow \text{Env} \rightarrow \text{Env} && \\
&\mathcal{B}_{\mathcal{B}} \llbracket \textit{let}\ \text{v}_1 = \text{e}_1 \ldots \text{v}_n = \text{e}_n \rrbracket \sigma && = \sigma\, [\text{v}_1 \ldots \text{v}_n / \mathcal{S}_{\mathcal{B}} \llbracket \text{s}_1 \rrbracket \sigma, \ldots\ \mathcal{S}_{\mathcal{B}} \llbracket \text{s}_1 \rrbracket \sigma] \\
&\mathcal{B}_{\mathcal{B}} \llbracket \textit{letrec}\ \text{v}_1 = \text{e}_1 \ldots \text{v}_n = \text{e}_n \rrbracket \sigma && = \textit{fix}\ (\lambda \sigma_1.\ \sigma\, [\text{v}_1 \ldots \text{v}_n / \mathcal{S}_{\mathcal{B}} \llbracket \text{s}_1 \rrbracket \sigma_1, \ldots\ \mathcal{S}_{\mathcal{B}} \llbracket \text{s}_1 \rrbracket \sigma_1])
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{S}_{\mathcal{B}}\text{:Lam} \rightarrow \text{Env} \rightarrow \text{V} && \\
&\mathcal{S}_{\mathcal{B}} \llbracket \text{var} \rrbracket \sigma && = \sigma\ \text{var} \\
&\mathcal{S}_{\mathcal{B}} \llbracket \text{const} \rrbracket \sigma && = \mathcal{K} \llbracket \text{const} \rrbracket \sigma \\
&\mathcal{S}_{\mathcal{B}} \llbracket \text{number} \rrbracket \sigma && = \mathcal{N} \llbracket \text{number} \rrbracket \\
&\mathcal{S}_{\mathcal{B}} \llbracket \text{string} \rrbracket \sigma && = \mathcal{S} \llbracket \text{string} \rrbracket \\
&\mathcal{S}_{\mathcal{B}} \llbracket \text{s}_1\ \text{s}_2 \rrbracket \sigma && = (\mathcal{S}_{\mathcal{B}} \llbracket \text{s}_1 \rrbracket \sigma \hookrightarrow_{V \rightarrow V}) (\mathcal{S}_{\mathcal{B}} \llbracket \text{s}_2 \rrbracket \sigma) \\
&\mathcal{S}_{\mathcal{B}} \llbracket (\text{s}) \rrbracket \sigma && = \mathcal{S}_{\mathcal{B}} \llbracket \text{s} \rrbracket \sigma \\
&\mathcal{S}_{\mathcal{B}} \llbracket [\text{annot}]\ \text{s} \rrbracket \sigma && = \mathcal{A} \llbracket [\text{annot}]\ \text{s} \rrbracket \sigma \\
&\mathcal{S}_{\mathcal{B}} \llbracket \lambda \text{v}_1 \ldots \text{v}_n.\text{s} \rrbracket \sigma && = \lambda\ \text{s}_1 \ldots \text{s}_n.\ \mathcal{S}_{\mathcal{B}} \llbracket \text{s} \rrbracket (\sigma\, [\text{v}_1 \ldots \text{v}_n / \mathcal{S}_{\mathcal{B}} \llbracket \text{s}_1 \rrbracket \sigma \ldots \mathcal{S}_{\mathcal{B}} \llbracket \text{s}_n \rrbracket \sigma]) \\
&\mathcal{S}_{\mathcal{B}} \llbracket \textit{let}\ \text{v}_1 = \text{e}_1 \ldots \text{v}_n = \text{e}_n\ \textit{in}\ \text{s} \rrbracket \sigma && = \mathcal{S}_{\mathcal{B}} \llbracket \text{s} \rrbracket (\mathcal{B}_{\mathcal{B}} \llbracket \textit{let}\ \text{v}_1 = \text{e}_1 \ldots \text{v}_n = \text{e}_n \rrbracket \sigma) \\
&\mathcal{S}_{\mathcal{B}} \llbracket \textit{letrec}\ \text{v}_1 = \text{e}_1 \ldots \text{v}_n = \text{e}_n\ \textit{in}\ \text{s} \rrbracket \sigma && = \mathcal{S}_{\mathcal{B}} \llbracket \text{s} \rrbracket (\mathcal{B}_{\mathcal{B}} \llbracket \textit{letrec}\ \text{v}_1 = \text{e}_1 \ldots \text{v}_n = \text{e}_n \rrbracket \sigma)
\end{aligned}
$$

$\mathcal{K}$ and $\mathcal{N}$ and $\mathcal{S}$ have their obvious meanings. $\mathcal{A}$ is dependent on the annotation, some of which are discussed later.

8.3 IFL

The semantics of IFL are fairly straightforward except for pattern matching and abstract types. Pattern matching will be explained as it is developed below, but abstract types warrant some discussion before we proceed.

The meaning of an expression is dependent on the meanings of its subexpressions. Since some expressions will contain constructors from type definitions, in order to understand the expression we must know the meaning of the constructor. In the following discussion a constructor is a simply a function. The fact that it is a function whose purpose is to make a new type is not important at this stage since we are not discussing the semantics of types, but of expressions. Later, expressions will be limited to those which are well-typed, and the meaning of a type will then be a addressed.

Consider this simplistic abstract type in IFL:

```
abstype list * = sum (nil, cons prod(*, list *)) in
        let hd = λ(cons h t). h,
            tl = λ(cons h t). t;
command;
```

The idea behind an abstract type is that it is only defined for the operations given on it (in this case list * is only defined for hd and tl) and not for any expressions given subsequent to the definition (ie. command). For this to be the case, the constructors (nil and cons) must be in the environment whenever hd or tl is called, but at no other time. As well, both hd and tl must access exactly the same constructor in the environment or they will be working on different types. (ie. we cannot take the easy way out and simply add the constructors to the environment whenever hd or tl is used since this would result in new constructors for each instance of hd and tl).

An elegant solution to this problem is to give an indirection to variable access in the environment. This can be done by defining the environment σ to be of type ($Var \rightarrow Int, Int \rightarrow V$) where Int forms the indirection pointer. Define σ to be (σ_1, σ_2) to represent these two functions, and σvar to be $\sigma_2(\sigma_1\ var)$ to represent the value in the environment corresponding to var. In all cases except abstract types, this environment works exactly like the ordinary environment mapping $var \rightarrow V$. (This indirection is often used in denotational semantics to incorporate branches, side effects, jumps, and procedure calls [7]. For abstract types it is slightly more complicated since we also require the use of annotations, as described below).

For an abstract data type we need a mapping in the environment for each constructor. For example, $nil \rightarrow n$ and $cons \rightarrow m$ are the mappings to Int (given that m and n are unique integers) - call these σ_1^a. $n \rightarrow [\![$ nil $]\!]$ and $m \rightarrow [\![$ cons $]\!]$ are the mapping from Int to the meanings of the constructors - call these σ_2^a.

The condition that hd and tl access the same constructors is equivalent to making sure that both access the same unique indirection keys (the integers m and n). This can be accomplished by adding σ_2^a to the environment.

The condition that only hd and tl can access the constructors means that σ_1^a must be in the environment when hd or tl are used, but at no other time. Notice that the mapping from $Var \rightarrow Int$ is syntactic in that it can be represented by pairs of values. Since this is the case, σ_1^a can be passed to hd and tl using an annotation.

This idea is developed by the semantics given below.

The syntatic domains are given below:

Ifl	=	{ I | I ϵ ifl}
Exp	=	{ E | E ϵ exp}
Bnd	=	{ B | B ϵ bind}
Pat	=	{ P | P ϵ pattern}
Tyn	=	{ T | T ϵ tyname}
Typ	=	{ T | T ϵ type}
Con	=	{ C | C ϵ constructor}
Var	=	{ V | V ϵ var}

The semantic domain is the same as the one developed for lam.

8.3.1 Expressions

Define a Binding in the domain V as:

Binding: (Var $\rightarrow$ (Int + {error}), (({error} $\rightarrow$ {error}) + (Int $\rightarrow$ V))

A binding maps a variable to a value in the domain using an indirection key (an integer) as discussed above. If the variable is not bound to a key or a key is not bound to a value, the result of the binding is an error.

$\mathcal{A}$ is used to add bindings to the environment. If a new binding is added then any access to the environment will first check the new bindings and then the old ones. $\mathcal{A}_\mathcal{O}$ and $\mathcal{A}_\mathcal{T}$ are variations of $\mathcal{A}$ which add Var $\rightarrow$ Int or Int $\rightarrow$ V mappings only, not both. $\mathcal{N}$ gives an empty binding where any access will result in an error, since nothing is bound.

$\mathcal{A}$: Binding $\rightarrow$ Binding $\rightarrow$ Binding
$\mathcal{A}\ (\sigma_1,\sigma_2)\ (\sigma_3,\sigma_4) = \lambda$var. (if ($\sigma_1$ var) = {error} then (σ_3 var) else (σ_1 var),
if (σ_2 (σ_1 var)) = {error} then (σ_3 (σ_4 var)) else (σ_2 (σ_1 var)))

$\mathcal{A}_{\mathcal{O}}$: Binding $\rightarrow$ (Var $\rightarrow$ Int) $\rightarrow$ Binding
$\mathcal{A}_{\mathcal{O}}\ (\sigma_1,\sigma_2)\ \sigma_3 = \lambda$var.(if ($\sigma_1$ var) = {error} then (σ_3 var) else (σ_1 var), σ_2)

$\mathcal{A}_{\mathcal{T}}$: Binding $\rightarrow$ (Int $\rightarrow$ V) $\rightarrow$ Binding
$\mathcal{A}_{\mathcal{T}}\ (\sigma_1,\sigma_2)\ \sigma_4 = \lambda$var.($\sigma_1$,if ($\sigma_2$ var) = {error} then (σ_4 var) else (σ_2 var))

$\mathcal{N}$: Var $\rightarrow$ V
$\mathcal{N} = \lambda$var.{error}

$\mathcal{R}$ models the interactive run-time environment. It takes in a countably infinite sequence of IFL terms and outputs a countably infinite product term of values. That is, if a binding is evaluated, no value is returned. The binding either succeeds and the new environment is used from then on, or it fails and the old environment is still used. A ";" is used to separate individual commands inside ifl^+.

$\mathcal{R}$: Ifl $\rightarrow$ V
$\mathcal{R}$ [[exp;ifl]] σ = ($\mathcal{E}$ [[exp]] σ, $\mathcal{R}$ [[ifl]] σ)
$\mathcal{R}$ [[bind;ifl]] σ = if σ_1 = {error} then $\mathcal{R}$ [[ifl]] σ else $\mathcal{R}$ [[ifl]] σ_1
where $\sigma_1 = \mathcal{B}$ [[bind]] σ

$\mathcal{M}$ is used for pattern matching. Lazy pattern matching is used on single patterns. If the pattern matches, a new environment is returned with each variable in the pattern bound. If the match fails, {error} is returned. An auxilary function must be defined first:

SEL s i (s $a_1,\ldots,a_i,\ldots a_n$)	= a_i
SEL s i (s' $a_1,\ldots,a_n$)	= {error}
SEL s i x	= {error}
SEL s i $\perp$	= $\perp$

$\mathcal{M}$: Pat $\rightarrow$ Ifl $\rightarrow$ Binding $\rightarrow$ Binding
$\mathcal{M}$ [[p]] x {error} = {error}
$\mathcal{M}$ [[var]] x (σ_1,σ_2) = (σ_1[x/n],σ_2[n/var])
where n is a unique integer key
$\mathcal{M}$ [[const]] x σ = if const = $\mathcal{E}$ [[x]] σ then σ else {error}
$\mathcal{M}$ [[con $p_1 \ldots p_n$]] e σ = $\mathcal{M}$ [[p_1]] (SEL con 1 e) ($\mathcal{M}$ [[p_2]] (SEL con 2 e)
(... $\mathcal{M}$ [[p_n]] (SEL con n e) σ)...)

$\mathcal{T}$ makes bindings for new type constructors.

$\mathcal{T}$: Typ $\rightarrow$ Binding $\rightarrow$ Binding
$\mathcal{T}$ [[sum $t_1 \ldots t_n$]] σ = $\mathcal{T}$ [[t_1]] ($\mathcal{T}$ [[t_2]] (... $\mathcal{T}$ [[t_n]] σ...))
$\mathcal{T}$ [[con $t_1 \ldots t_n$]] (σ_1,σ_2) = (σ_1[con/n],σ_2[n/$\lambda t_1 \ldots t_n.(con, t_1, \ldots, t_n)$])
where n is a unique integer key

$\mathcal{B}$ models binding in the environment. The binding for an abstract data type was discussed above and is shown below. As mentioned above, the function from $Ifl \rightarrow Int$ could be represented in an annotation in the form [con_1 ==> n_1 ... con_m ==> n_m] where each con_i is a constructor and each n_i is unique integer key. For simplicity I have simply included the mapping inside the annotation.

$\mathcal{B}$ Bnd → Binding → Binding
$\mathcal{B}$ ⟦ *let* p_1 e_1 p_n e_n ⟧ σ
  $\mathcal{M}$ ⟦ p_n ⟧ e_n ($\mathcal{M}$ ⟦ p_1 ⟧ e_1 σ)
$\mathcal{B}$ ⟦ *letrec* p_1 e_1 p_n e_n ⟧ σ
  fix ($\lambda\sigma_1$ $\mathcal{A}$ ($\mathcal{M}$ ⟦ p_1 ⟧ e_1 σ_1,(,$\mathcal{A}$ ($\mathcal{M}$ ⟦ p_1 ⟧ e_1 σ_1,σ)))
$\mathcal{B}$ ⟦ *lettype* tn_1 t_1 tn_n t_n ⟧ σ
  fix ($\lambda\sigma_1$,$\mathcal{A}$ ($\mathcal{T}$ ⟦ t_1 ⟧ σ_1, ($\mathcal{A}$ ($\mathcal{T}$ ⟦ t_n ⟧ σ_1,σ))))
$\mathcal{B}$ ⟦ *abstype* tn_1 t_1 tn_n t_n *in* letbind ⟧ σ
  $\mathcal{B}_{ADT}$ ⟦ letbind ⟧ σ ($\mathcal{B}$ ⟦ *lettype* tn_1 t_1 tn_n t_n ⟧ $\mathcal{N}$)

$\mathcal{B}_{ADT}$ Bnd → Binding → Binding → Binding
$\mathcal{B}_{ADT}$ ⟦ *let*(*rec*) p_1 e_1 p_n e_n ⟧ σ (σ_1,σ_2)
  $\mathcal{B}$ ⟦ *let*(*rec*) p_1 [σ_1] e_1 p_n [σ_1] e_n ⟧ ($\mathcal{A}_I$ σ σ_2)

$\mathcal{C}$ is used for case statments Successive matches are tried until one is found which does not cause an error

$\mathcal{C}$: V → V → V
$\mathcal{C}$ {error} f   f
$\mathcal{C}$ ifl x   ifl

$\mathcal{K}$ evaluates constants These are pretty standard and not all of them are given below

$\mathcal{K}$ Cnst → V
$\mathcal{K}$ ⟦ *if* ⟧ σ λb t e if $\mathcal{E}$ ⟦ b ⟧ σ true then $\mathcal{E}$ ⟦ t ⟧ σ else $\mathcal{E}$ ⟦ e ⟧ σ
$\mathcal{K}$ ⟦ + ⟧ σ *lambda* a b $\mathcal{E}$ ⟦ a ⟧ σ + $\mathcal{E}$ ⟦ a ⟧ σ
$\mathcal{K}$ ⟦ int ⟧ σ Integer
$\mathcal{K}$ ⟦ *fail* ⟧ σ {error}

$\mathcal{ANN}$ is used for annotations At this time the only annotation described below is for abstract data types It simply takes the *Var* → *Int* mapping out of the annotation and adds it to the environment

$\mathcal{ANN}$ Ifl → Binding → V
$\mathcal{ANN}$ ⟦ [σ_1] exp ⟧ σ $\mathcal{E}$ ⟦ exp ⟧ ($\mathcal{A}_O$ σ σ_1)

$\mathcal{E}$ models expressions which evaluate to values.

$$
\begin{array}{ll}
\mathcal{E}\colon \mathsf{Exp} \to \mathsf{Binding} \to \mathsf{V} & \\
\mathcal{E}\ [\![\,\mathsf{x}\,]\!]\ \{\mathsf{error}\} & = \{\mathsf{error}\} \\
\mathcal{E}\ [\![\,\mathsf{var}\,]\!]\ \sigma & = \sigma\ [\![\,\mathsf{var}\,]\!] \\
\mathcal{E}\ [\![\,\mathsf{const}\,]\!]\ \sigma & = \mathsf{K}\ [\![\,\mathsf{constant}\,]\!] \\
\mathcal{E}\ [\![\,\mathsf{number}\,]\!]\ \sigma & = \mathsf{N}\ [\![\,\mathsf{number}\,]\!] \\
\mathcal{E}\ [\![\,\mathsf{string}\,]\!]\ \sigma & = \mathsf{S}\ [\![\,\mathsf{string}\,]\!] \\
\mathcal{E}\ [\![\,\mathsf{exp}_1\ \mathsf{exp}_2\,]\!]\ \sigma & = (\mathcal{E}\ [\![\,\mathsf{exp}_1\,]\!]\ \sigma \hookrightarrow_{V\to V})\ (\mathcal{E}\ [\![\,\mathsf{exp}_2\,]\!]\ \sigma) \\
\mathcal{E}\ [\![\,[\mathsf{annot}]\ \mathsf{exp}\,]\!]\ \sigma & = \mathcal{ANN}\ [\![\,\mathsf{annot\ exp}\,]\!]\ \sigma \\
\mathcal{E}\ [\![\,(\mathsf{exp})\,]\!]\ \sigma & = \mathcal{E}\ [\![\,\mathsf{exp}\,]\!]\ \sigma \\
\mathcal{E}\ [\![\,\lambda\ \mathsf{p}_1 ... \mathsf{p}_n.\mathsf{exp}\,]\!]\ \sigma & = \lambda\ \mathsf{e}_1 ... \mathsf{e}_n.\ \mathcal{E}\ [\![\,\mathsf{exp}\,]\!] \\
 & \quad (\mathcal{B}\ [\![\,\mathit{let}\ \mathsf{p}_1 = \mathsf{e}_1\ ...\ \mathsf{p}_n = \mathsf{e}_n\,]\!]\ \sigma) \hookrightarrow_V \\
\mathcal{E}\ [\![\,\mathit{let}\ \mathsf{p}_1 = \mathsf{e}_1\ ...\ \mathsf{p}_n = \mathsf{e}_n\ \mathit{in}\ \mathsf{exp}\,]\!]\ \sigma & = \mathcal{E}\ [\![\,\mathsf{exp}\,]\!]\ (\mathcal{B}\ [\![\,\mathit{let}\ \mathsf{p}_1 = \mathsf{e}_1\ ...\ \mathsf{p}_n = \mathsf{e}_n\,]\!]\ \sigma) \\
\mathcal{E}\ [\![\,\mathit{letrec}\ \mathsf{p}_1 = \mathsf{e}_1\ ...\ \mathsf{p}_n = \mathsf{e}_n\ \mathit{in}\ \mathsf{exp}\,]\!]\ \sigma & = \mathcal{E}\ [\![\,\mathsf{exp}\,]\!]\ (\mathcal{B}\ [\![\,\mathit{letrec}\ \mathsf{p}_1 = \mathsf{e}_1\ ...\ \mathsf{p}_n = \mathsf{e}_n\,]\!]\ \sigma) \\
\mathcal{E}\ [\![\,\mathit{lettype}\ \mathsf{tn}_1 = \mathsf{t}_1 ... \mathsf{tn}_n = \mathsf{t}_n\ \mathit{in}\ \mathsf{exp}\,]\!]\ \sigma & = \mathcal{E}\ [\![\,\mathsf{exp}\,]\!]\ (\mathcal{B}\ [\![\,\mathit{lettype}\ \mathsf{tn}_1 = \mathsf{t}_1\ ...\ \mathsf{tn}_n = \mathsf{t}_n\,]\!]\ \sigma) \\
\mathcal{E}\ [\![\,\mathit{case}\ \mathsf{v}\ \mathit{of}\ \mathsf{p}_1 \Rightarrow \mathsf{e}_1 ... \mathsf{p}_n \Rightarrow \mathsf{e}_n\,]\!]\ \sigma & = \mathcal{C}\ (\mathcal{E}\ [\![\,\mathsf{e}_1\,]\!]\ (\mathcal{M}\ [\![\,\mathsf{p}_1\,]\!]\ (\mathcal{E}\ [\![\,\mathsf{v}\,]\!]\ \sigma)\ \sigma),\ ..., \\
 & \quad \mathcal{C}\ (\mathcal{E}\ [\![\,\mathsf{e}_{n-1}\,]\!]\ (\mathcal{M}\ [\![\,\mathsf{p}_{n-1}\,]\!]\ (\mathcal{E}\ [\![\,\mathsf{v}\,]\!]\ \sigma)\ \sigma), \\
 & \qquad (\mathcal{E}\ [\![\,\mathsf{e}_n\,]\!]\ (\mathcal{M}\ [\![\,\mathsf{p}_n\,]\!]\ (\mathcal{E}\ [\![\,\mathsf{v}\,]\!]\ \sigma)\ \sigma))...) \\
\mathcal{E}\ [\![\,\Box\ \mathsf{e}_1\ \mathsf{e}_2\,]\!]\ \sigma & = \mathcal{C}\ (\mathcal{E}\ [\![\,\mathsf{e}_1\,]\!]\ \sigma,\ \mathcal{E}\ [\![\,\mathsf{e}_2\,]\!]\ \sigma)
\end{array}
$$

where S maps strings into the appropriate B_i, N maps numbers into B_1, and K maps other constants into the other B_i domains.

8.3.2 Typechecking

Work on the semantics of the types for IFL is still underway. The type discipline is a simple Milner style system at this point.

Typechecking semantics simply limit the domain V to those meanings which come from well-typed IFL expressions.

The abstract syntax of a type is given by:

texp ::= tvar | tcon texp ... texp
tvar ::= "*" integer
tcon ::= *sum* | *prod* | *fun* | *alphabetic alphanumeric**

A builtin type, such as int, is a tcon of zero arity.

The syntactic domains for the semantics of typechecking will include Ifl, Exp, Bnd, Pat, Tyn and Var for IFL terms as well as the following domains for type expressions:

$\mathsf{Texp} = \{\mathsf{T} \mid \mathsf{T} \in \mathsf{texp}\}$
$\mathsf{Tvar} = \{\mathsf{V} \mid \mathsf{V} \in \mathsf{tvar}\}$
$\mathsf{Tcon} = \{\mathsf{C} \mid \mathsf{C} \in \mathsf{tcon}\}$

In order to show that a well-typed expression does will not cause any run-time type errors, it will be necessary to show that an syntactic object of a certain type always maps to an semantic object with that same type. That is, an integer should always map into the domain B_1. The easiest way to do this is to define the domain of type expressions to be the same as the domain for expressions, namely V. In this case, we could map the type "int" to the domain of integers, B_1.

MacQueen's work with *ideals* [13] allows us to do exactly this. To summarize his work, define a subset I of V to be an *ideal* if and only if:

1) $I \neq \{\}$
2) $\forall y \in I,\ x \in V.\ x \sqsubseteq y \ \to x \in I$
3) $\forall < x >.(\forall n. x_n \in I) \ \to \sqcup < x > \in I$

$\perp$ is assumed to have every type, thus an ideal cannot be empty. An ideal contains chains beginning with $\perp$ and containing a least fixed point.

Denote the subset of V which contains only ideals as $\mathcal{I}$(V). Given that I and J are ideals, we can define binary functions on them:

- I $\oplus$ J is the union of the injection of I into V + V and the injection of J into V + V.
- I $\otimes$ J is the cross product of I and J, I $\times$ J.
- I $\oplus\!\!\!\!\rightarrow$ J is the function space $\{f \epsilon\ V \rightarrow V \mid f(I) \subseteq J\}$

Each of these preserve ideality.

Given a mapping from type variables to $\mathcal{I}$(V), we can define

$\mathcal{D}$: $\mathsf{t} \,\epsilon\, \mathsf{texp} \rightarrow (\mathsf{v} \,\epsilon\, \mathsf{tvar} \rightarrow \mathcal{I}(\mathsf{V})) \rightarrow \mathcal{I}(\mathsf{V})$
$\mathcal{D}\ [\![\ \mathsf{c}\ ]\!]\ v = \mathcal{D}_{\mathcal{C}}\ [\![\ \mathsf{c}\ ]\!]$
$\mathcal{D}\ [\![\ \mathsf{t}\ ]\!]\ v = v\ \mathsf{t}$
$\mathcal{D}\ [\![\ \mathit{prod}\ \mathsf{texp}_1\ \mathsf{texp}_2\ ]\!]\ v = \mathcal{D}\ [\![\ \mathsf{texp}_1\ ]\!]\ v \otimes \mathcal{D}\ [\![\ \mathsf{texp}_2\ ]\!]\ v$
$\mathcal{D}\ [\![\ \mathit{sum}\ \mathsf{texp}_1\ \mathsf{texp}_2\ ]\!]\ v = \mathcal{D}\ [\![\ \mathsf{texp}_1\ ]\!]\ v \oplus \mathcal{D}\ [\![\ \mathsf{texp}_2\ ]\!]\ v$
$\mathcal{D}\ [\![\ \mathit{fun}\ \mathsf{texp}_1\ \mathsf{texp}_2\ ]\!]\ v = \mathcal{D}\ [\![\ \mathsf{texp}_1\ ]\!]\ v\ \oplus\!\!\!\!\rightarrow\ \mathcal{D}\ [\![\ \mathsf{texp}_2\ ]\!]\ v$

$\mathcal{D}_{\mathcal{C}}\ [\![\ \mathsf{bool}\ ]\!] = \mathsf{B}_0$
$\mathcal{D}_{\mathcal{C}}\ [\![\ \mathsf{int}\ ]\!] = \mathsf{B}_1$
...

This ideal model was actually defined in order to handle recursive types, thus, if recursive types are added to the discipline, I may still work with the same model.

The semantics for types are only partially complete, and are thus not included here.

References

[1] R. Burstall and B. Lampson. A kernel language for abstract types and modules. In G. Khan, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types, LNCS 173*, pages 1–50, 1984.

[2] R.M. Burstall, D.B. MacQueen, and D.T. Sannella. Hope: An experimental applicative language. Internal Report CSR-6280, University of Edinburgh, May 1980.

[3] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–520, December 1985.

[4] Jon Fairbairn. *Design and Implementation of a Simple Typed Language Based on the Lambda Calculus.* PhD thesis, Cambridge University, 1985.

[5] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In *Functional Languages and Computer Architecture. LNCS 274*, pages 34–45, 1987.

[6] Anthony J. Field and Peter G. Harrison. *Funtional Programming.* Addison-Wesley, 1988.

[7] Micheal Gordon. *The Denotational Description of Programming Languages.* Springer–Verlag, 1979.

[8] J. A. Guguen, J. W. Thatcher, E. G. Wagner, and J.B. Wright. Initial algebra semantics and continuous algebra. In *Association for Computing Machinery ??*, pages 68–95, 1977.

[9] Robert Harper, David MacQueen, and Robin Milner. *Standard ML.* University of Edinburg, 1987.

[10] P. Henderson. *Functional Programming – Application and Implementation.* Prentice/Hall International, 1980.

[11] Paul Hudak. *Compiling Functional Languages.* Tutorial for Functional Programming and Computer Architectures Conference, Portland Oregon, 1987.

[12] P. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[13] D. MacQueen. An ideal model for recursive polymorphic types. In *ACM ??*, pages 165–174, 1983.

[14] R. Milner. A theory of type polymorphism in programming. Internal Report 9, University of Edinburgh, September 1977.

[15] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International Series in Computer Science, 1987.

[16] Simon Peyton Jones. FLIC — a Functional Language Intermediate Code. *ACM SIGPLAN Notices*, 23(8):30–48, August 1988. Also: Internal Note 2048, Department of Computer Science, University College London, Gower St., London WC1E 6BT.

[17] Simon Peyton Jones. Personal communication. March 1988.

[18] John C. Reynolds. Three approaches to type structure. In *Springer Verlag LNCS 185*, pages 97 - 138, 1985.

[19] Dana Scott. Data types as lattices. *Siam J. Comput.*, 5(3):522–587, September 1976.

[20] D. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture, Nancy France.* Springer–Verlag, 1985.

[21] Phillip Wadler. How to replace failure by a list of successes. In *Functional Programming Languages and Computer Architecture, Nancy France.* Springer–Verlag, 1985.

[22] Stuart Wray. *Implementation and Programming Techniques for Functional Languages.* PhD thesis, Cambridge University, 1986.