

Felix Language Elements

John Skaller

November 27, 2015

Contents

1	Functions	1
1.1	Purity	1
1.2	Totality	1
1.3	Strictness	2
1.4	Lifting from C	2
1.5	Evaluation Strategies	2
1.5.1	Eager Evaluation	3
1.5.2	Lazy Evaluation	3
1.5.3	Elimination	3
1.5.4	Indeterminate Evaluation	4
1.5.5	Semantics and Evaluation	4
1.6	Contracts	6
1.6.1	Preconditions	6
1.6.2	Postconditons	6
1.6.3	Contracts	7
2	Procedures	7
3	Generators	8
3.1	Generator lifting	8
3.2	Yielding Generators	9
4	Fibres	10
5	Restrictions on Service Calls	11
6	Low level Control exchange	12

1 Functions

In Felix functions are a group of constructions used to perform calculations. The term *function* is a historical abuse, taken from C. There is a relation to mathematical functions which we explore here.

An *fun* binder is used to introduce a calculation which returns a value but has no side effects.

1.1 Purity

A function which depends only on its parameters is said to be a *pure* function. Here is a simple example:

```
pure fun f (x:uint) => x + x;
```

The adjective *pure* may be used to assert a function is pure.

Here is a function which is not pure:

```
var x = 1;  
impure fun g() => x;
```

The adjective *impure* is optional and may be used to assert a function is not pure.

The value returned by this function like construction depends on a variable. If the variable is changed, the function will return a different value. This function is *impure* and is sometimes called an *observer* or *accessor*.

1.2 Totality

A function which returns a proper value for each element of the domain type, provided it terminates, is said to be *total*. Here is a total function:

```
total fun f (x:uint) => x + x;
```

The adjective *total* may be used to assert a function is total.

In Felix, not all functions are total. A function which is not total is said to be partial.

For example:

```
partial fun h(x:int) => x + x;
```

This is only a *partial* function, because if the argument is more than half the maximum *int* or less than half the minimum, the result is unspecified.

The adjective *partial* may be used to assert a function is not total. A partial function may be total on a suitable subdomain. In this case the behaviour of the function outside this domain is unspecified.

In the case of our partial function above, an exception may be thrown, an incorrect answer returned, or the system may delete your hard disk.

1.3 Strictness

A function is said to be *strict* if and only if the application of the function to an argument which fails to properly evaluate also fails to properly evaluate. For example:

```
strict fun h(var x:uint) => x + x;
```

Here, the *var* forces the argument to be evaluated before the body of the function, therefore if the argument evaluation fails, so does the function application. On the other hand since we know the function is total, if the argument does evaluate correctly, the function cannot fail.

1.4 Lifting from C

As well as writing functions in Felix, you can also lift functions from C. For example this encoding:

```
fun add1 : int -> int = "$1+1";
```

specified the Felix function `add1` is defined in C++. The notation `$1` means the first argument. In the example:

```
fun sum : int * int -> int = "$1+$2";
```

the first and second components of the single tuple argument are used. Felix is a C++ code generator and calculates this function by emitting the defining C++ code with the `$1` and `$2` strings replaced by the code evaluating the first and second components of the argument tuple.

Adjectives for lifted functions are particularly important because the compiler is unable to derive any properties by analysis.

1.5 Evaluation Strategies

Felix uses several different strategies for evaluating function applications.

1.5.1 Eager Evaluation

In this strategy, the argument of a function is evaluated before the function is applied to it: the result of the evaluation is assigned to the function parameter and then the body of the function executed. Therefore, if the argument evaluation fails, so too does the function application. Therefore, if this strategy is used, then if the function is total, it must also be strict.

Eager evaluation is usually used for big functions and for closures. It may also be used if the compiler determines the parameter is used many times in the body of the function.

Using a *var* parameter forces eager evaluation, provided the parameter is actually used.

Another related technique is to use the *noinline* adjective. This ensures the function is not inlined and the resulting code will be a separately generated C function or C++ applicative object, which therefore always use eager evaluation. For example:

```
noinline fun h(x:uint) => x + x;
```

prevents the function being inlined, and therefore it will be eagerly evaluated.

1.5.2 Lazy Evaluation

In this strategy, the argument of the function replaces the parameter and is evaluated on demand inside the function. If control does not flow through an expression requiring the parameter, the argument is not evaluated.

Lazy evaluation is typically effected by inlining the function and replacing the parameter with the argument expression. However a specialised version of the function might be created in which the substitution is done, but the function itself is not inlined.

Lazy evaluation cannot be forced, however it can be simulated by passing a closure. It can be encouraged by specifying the adjective *inline*. This forces inlining of the function, except in two cases: the function is recursive, or, a closure of the function is formed.

```
inline fun h(x:1 -> double) =>
    if x() > 1.0 then 1.0 / x() else 0.0 endif
;
```

1.5.3 Elimination

Elimination is an evaluation strategy that deletes computations that are not required. Felix uses elimination in two contexts.

First, Felix does constant folding, which is compile time evaluation. In particular conditionals with constant conditions can result in elimination of branches which are never taken, that is, conditional compilation. This happens very early and can eliminate expressions which would not pass type checking or bind correctly.

```
if PLAT_WIN32 do
    LoadLibrary();
else
    dlopen();
done
```

Felix also elides unused variables. Except for parameters this is guaranteed. For example:

```
gen f() {
    var f = open_in ("filename");
    return 42;
}
```

will not actually open the file because the variable `f` is never used.

1.5.4 Indeterminate Evaluation

Felix actually uses *indeterminate evaluation strategy* which means that the semantics do not dictate by default whether eager or lazy evaluation is used, however the *elimination* rules are determinate.

Whilst we provide ways to enforce a particular strategy, the core idea is that the compiler chooses the most performant strategy. Typically direct function calls are inlined, and therefore lazy, because this is most efficient. In particular, calls to C function bindings are lazily evaluated so that a Felix expression reduces to a C/C++ expression, allowing the target C/C++ compiler its own optimisation opportunities. Enforcing eager evaluation would require unravelling all expressions into a sequence of assignments to variables and applications to variables, just to fix the evaluation order. It's not clear the target C/C++ compilers could unravel these assignments and eliminate the explicit temporaries.

On the other hand, it is quite hard to do lazy evaluation if the function is represented as a C function, or C++ class object. Since closures are modelled by C++ class objects, arguments are arguments to C++ methods and these are eagerly evaluated by C/C++ rules. Although we could pass arguments in closure wrappers as Haskell might, this is horribly expensive.

1.5.5 Semantics and Evaluation

So now, we must ask, does the evaluation strategy matter? Most languages enforce a particular strategy. Almost all programming languages use lazy evaluation for almost everything. In particular procedural languages like C are necessarily lazy, because lazy evaluation is just another name for control flow. Function calls and operators are a special case in C, which specifies eager evaluation except for a short cut operators and macros which are lazy.

On the other hand strangely, Haskell is eager, contrary to popular belief! This is because functional programming in general does not make control flow or order of evaluation determinate. When the order is based on dependencies, and evaluated by need, the needs must be propagated by an eager seed. For example a variant in Haskell is not a proper sum type because the case tag is evaluated eagerly, in order to avoid eagerly evaluation the wrong branch. Additionally, Haskell must do strictness analysis to evaluation basic functions and their derivatives, otherwise nothing would happen at all.

This leads to the observation that in a lazy language, it is not only safe to evaluate strict function applications eagerly, but in fact it is necessary (otherwise the whole program just returns a big fat closure without actually doing anything!) Eager evaluation is what triggers execution in lazy languages.

So, if we have a strict total terminating function it is safe to evaluate it by either eager or lazy method. Because it terminates, lazy evaluation can be promoted to eager evaluation due to strictness, and all lazily evaluated terminating functions are total. If it is eagerly evaluated, then since all eager evaluation is strict, termination ensures that it can be dropped to lazy evaluation.

So we come to the observation that if a function is applied to an argument in its actual domain of correct operation, it is logically total, so if the argument itself also evaluates correctly, the application is as if the function were also strict.

In other words, in practice, indeterminate evaluation is the best strategy because for most functions the evaluation strategy is irrelevant, and the performance is best.

So we now come to the semantic effect of the adjectival descriptors. For C bindings, these descriptors are taken at face value by the compiler since it is unable to do any analysis, C code is generally regarded as opaque.

If conflicting adjectives are used the compiler may reject the program with a diagnostic message but is not required to.

The compiler may analyse the code of any function and determine its properties, but is not required to. If these properties conflict with a specified adjective, it may reject the program with a diagnostic message, but is not required to. It may issue a warning and continue, in which case it is required to discard the incorrect description.

The compiler may also add descriptive attributes based on analysis.

It is vital to understand how the compiler is allowed to use the descriptors for optimisation. If a function is pure total and strict, it is referentially transparent, and can be evaluated at any point for which the arguments can be transparently computed.

On the other hand, most of the adjectives can be ignored by the compiler, due to the indeterminate evaluation strategy. It may not be safe to eagerly evaluate the argument of a nonstrict function, but the compiler is entitled to do so anyhow.

Except in special circumstances, the descriptors may be used to aid optimisation, in which case a fault is the programmers if the descriptor is incorrect, but they do not enforce a particular evaluation technique.

The special circumstances are currently: if a function is marked inline, a direct call to the function will be inlined if it is not recursive. A closure, however, might not be inlined.

If a function is marked `noinline`, it will never be inlined.

If a function has a `var` parameter, it will act as if eagerly evaluated, unless it is statically determined the result of the evaluation is unused, in which case it may be eliminated.

If the argument type is a function, it will act as if the argument closure is evaluated only on demand, that is, lazily evaluated.

For a `fun` binder, side-effects are not allowed, and the compiler may reject a program in which a fun-bound function has side-effects. However there is a special caveat, debugging side-effects are allowed. Since it is hard to determine exactly what a debugging side effect actually is, and whether in fact a function has side-effects or not, the compiler is generally conservative and accepts functions with side effects without protest. However it may assume for optimisation purpose that there are no side-effects, reordering or eliminating debugging side-effects. Indeed, this is part of the reason for using such debugging, to determine what the compiler actually did to evaluate the function. Unfortunately such side effects may be detectable and change the evaluation strategy. Writing to standard error is a designated debugging side-effect. Felix also contains some built-in and library supplied monitoring.

1.6 Contracts

1.6.1 Preconditions

Some partial function can be made total by restricting the domain to a suitable subset. However often there is no suitable type available and we may use a *precondition* instead. For example:

```
fun h (x:int where x< MAXINT / 2 and x > MININT /2) => x + x;
```

Preconditions may serve as documentation or be tested dynamically either at the function call site or in the body of the function.

A precondition may be thought of as a run time typing constraint. For example the actual type of the domain of the function above is not `int`, but a subrange of `int` with bounds halved. A type error therefore consists of both static and dynamic checking.

Static typing can be tricked by use of casts. Dynamic type checks may be elided by the compiler because it is lazy about implementing them, because the compiler can prove that they will never fail, or because the programmer told the compiler to elide them.

1.6.2 Postconditions

Sometimes semantic constraints may be specified like this:

```
fun f (x:double) : double expect result <= 1.0 => sin x;
```

The `expect` expression introduces a *postcondition* which serves as a constraint on the implementation. The postcondition may be checked before returning the result or serve as documentation.

1.6.3 Contracts

When both a pre and post condition are given, together they constitute a *contract*. For example

```
fun h (  
  x:int  
  where x< MAXINT / 2 and x > MININT /2)  
: int  
  expect if x < 0 then result < 0 else result >= 0 endif  
=>  
  x + x  
;
```

The contract says if you provide a suitably small argument then the result will be the same sign as the argument. A contract is a constraint on the implementation of the function, but not usually a complete specification.

A contract has two interpretations: first, as a checkable precondition and postcondition, this is the interpretation used by Felix.

But second, it may be viewed as merely saying that if the precondition is met then the postcondition will be, and in particular not implying that an argument not satisfying the precondition is a wrong value to give to the function. That is, the contract can be checked but if a value outside the specified precondition is supplied to the function, the contract is satisfied. Felix allows you to say this by writing the contract as an implication in the postcondition:

```
fun h (
  x:int : int
  expect
    x < MAXINT / 2 and x > MININT / 2)
  implies
    if x < 0 then result < 0 else result >= 0 endif
=>
  x + x
;
```

2 Procedures

If a function return type is specified as **void** or the **proc** binder is used, that designates a special kind of function which normally returns control but no value called a *procedure*. Unlike other kinds of functions, procedures are not only allowed to have side-effects but should have them, since they have no impact on a program if they do not. Useless procedures, however, are allowed because they may result from commenting out the body of a procedure during development.

Calls to procedures are statements, and are executed deterministically when control flows through the procedure call.

Procedures with limited side effects may be used in functions, provided the side effects do not escape the function, for example if they only modify a local variable.

We will say a bit more about procedures later, however we will note that the **inline** and **noinline** adjectives apply to procedures. Evaluation strategy issues also apply to procedures.

3 Generators

There is a kind of function which is allowed to have side-effects called a *generator*. The prototypical generator is the **rand()** function. See below for more information on generators. Generators may be defined using the **gen** binder. A simple example of a generator:

```
var counter = 1;
gen fresh() : int = {
  ++counter;
```



```

    return counter;
}

```

Each call to this function will return a different value since the function modifies the variable it depends on. A generator may still depend only on its arguments:

```

gen fresh(counter: &int) : int= {
    ++*counter;
    return *counter;
}

```

Although this generator has side effects, and depends on an external variable, the address of the variable is passed to the generator.

3.1 Generator lifting

When Felix sees a direct call to a generator in an expression, the call is lifted out and replaced by a fresh variable. The variable is declared and assigned to the application in a statement preceeding the statement containing the original application. Just how far back this is depends on the context and it may result in access to uninitialised variables.

If the lifted expression itself contains a direct generator application, the process is repeated.

If a call has an argument which is an explicit tuple expression, and both subexpressions contain a generator application, the first written application will be evaluated before the second one.

These rules are therefore equivalent to a depth first left to right tree visitation. The resulting order of evaluation is therefore deterministic module re-ordering in the parser and front end desugaring, with the following caveat: if a generator result is not used, it will be elided and the side-effects lost.

Note carefully these rules only apply to direct applications. Generator closures have the same type as a similar function, so closure evaluations cannot be reordered.

3.2 Yielding Generators

A special kind of generator is available in Felix called a *yielding* generator. Such a generator does not use any external storage explicitly for its state.

```

gen fresh() : int= {
    var counter = 10;
    while counter > 0 do
        --counter;
        yield counter;
    done
    return counter;
}

```

```
var fresher = fresh;
println$ fresher(), fresher();
```

The **yield** statement causes a value to be returned but preserves the current program location so a subsequent invocation resumes execution where it previously left off. In the case of a **return** statement, execution resumes by re-evaluating the return statement. The above generator, therefore, will count down from 9 through to 1 and then return an infinite stream of 0.

This kind of generator does not really hold its state internally. Instead, it must be explicitly assigned to a variable which holds a closure of the generator function. Calls are then made through the variable which contains the local variables of the generators stack frame as subobjects.

Generators differ from functions operationally in that calls through variables do not cause the clone of the value to be spawned, and therefore are deliberately not reentrant. Functions on the other hand clone the closure on every invocation to ensure they commence with clean local variables. Although normally this does not matter, the program counter for a function closure is stored inside the function closure and if the function is called again, it would restart like a generator, where it last left off: at the end. This is especially vital for recursive functions which have to be re-entrant.

With yielding generators, the programmer is responsible for creating the state object, so real recursion can be implemented by suitably managing the state variables.

The run time behaviour of a generator or procedure is determined by the original definition, that is, when a closure is passed to a higher order function, if it was a function closure the closure will be cloned before execution, if it was a generator closure, it will not be. The higher order function does not know or need to know. However a higher order function which is expected to be purely functional may not be if it is passed a generator closure.

4 Fibres

A *fibre* or *fthread* is an object participating in a synchronous interleaved sharing of a physical thread of control. The context switching is constrained but not determinate.

Fibres use *synchronous channels* or *schannels* to mediate exchange of control.

```
begin
  var ins, outs = mk_ioschannel_pair();
  proc reader () {
    var k = read ins;
    while true do
      println$ k;
      k = read ins;
    done
  }
```

```

proc writer () {
  for i in 1..20 do
    write (outs, i);
  done
}
spawn_fthread reader;
spawn_fthread writer;
end

```

Here, we create two bindings of the same schannel, one restricted to reading and one to writing. These act as the two ends of the data pipeline.

The reader will read a value from the channel and print it, the writer writes a finite number of values and gives up.

The use of **begin** and **end** pair to create a block here is critical. When the block has finished spawning the two fibres it will exit. However the program will not terminate yet. The program itself is just an fthread, and although that fthread is now finished, the spawned fthreads have not.

When the writer is finished writing it will terminate, leaving the reader hanging on the schannel for input which will never come. However, the reader is only reachable through the schannel, and the schannel is only reachable through the block, and the block is not reachable at all, so the garbage collector will remove the block and the schannel and the program will terminate.

More precisely, programs terminate when the current fibre has complete and there are no other fibres waiting to be scheduled. When a read or write on a schannel is performed, the executing fibre is moved to the schannel's wait queue. Then, if the request was for a read and there is a waiting writer, or a write and there is a waiting reader, both the requestor and requestee are moved off the schannel's wait queue back onto the master scheduler queue. Because of this an unsatisfied read or write allows any fibre scheduled on the master queue to become active and it may perform the required read or write if the schannel is reachable.

It is technically not determined which fibre runs first at a control exchange point. However for pragmatic reasons the reader is always allowed to proceed first in the current implementation so it may fetch a value from the writers stack frame as it is at the time of the synchronisation without the writer having a chance to modify its state.

Similarly, when an fthread is spawned it is technically not determined whether the spawnnee or spawner will proceed first, but for pragmatic reasons the spawnnee proceeds first. This makes a spawn operation identical to a subroutine call and allows the spawnnee to fetch data from the spawner before it has a chance to modify it.

This enables more controlled shared memory data access.

Note carefully fthreads are coroutines which do not have a master/slave or push/pull relationship: they are peers. The main routine is in fact just another such fibre with no special privileges.

Note again that fibres interleave control within a single pthread, and in

particular a single CPU. There is no pre-emption here, and no concept of concurrency or parallelism. Control exchanges are entirely synchronous.

5 Restrictions on Service Calls

Although Felix procedures may be nested in functions, certain operations called *service calls* may not be. Spawning an fthread and reading and writing on schannels is performed by a service call. A service call is basically an entry into the abstract operating system.

Unfortunately, Felix performs service calls by the simple expedient of setting a flag in the procedure making the call and returning to the scheduler. In order for a return to actually return to the scheduler, the machine stack must have the scheduler's return address at the top. That is, when the scheduler calls the procedure, the procedure must pop the machine stack back to where it was on entry before returning.

On the other hand, in order to resume after a service call, the scheduler just calls the same method again. So the procedure is responsible for saving the code address of point after the service call, known as the *current continuation* and jumping back to it when re-entered.

Felix organises this by placing local variables in C++ class objects instead of on the machine stack, and avoiding constructions such as for loops and blocks which may use the machine stack. In this way, it is possible to jump directly to the current continuation, which is done either by a switch or computed goto.

In other words, Felix procedures may not use the machine stack except transiently. Instead, procedure stack frames are allocated on the heap and linked together with pointers. In particular a procedure return address is saved by the calling procedure, not the callee. This means that procedures are not reentrant. This is not an issue because the frame contains mutable variables anyhow, and the current state is associated with the current locus of control.

Unfortunately, whilst functions also may use heap allocated stack frames, the return address for a function is stored in the usual way on the machine stack. This is done to ensure C compatibility, in particular C expressions built out of Felix function closures will work fine in C. Conversely, C functions wrapped as Felix functions will work fine in Felix.

However the impact is that a function may not, directly or indirectly, perform a service call. Procedures called by the function will work even if they call and return, because the function calling the procedure provides its own mini-scheduler which supports calls and returns. But the mini-scheduler does not support service calls.

6 Low level Control exchange

Felix also provides a low level control exchange instruction which allows two procedures to swap control. The instruction is a branch-and-link instruction

which jumps to a location stored in a variable and at the same time storing the current location in another variable.

The fthread mechanism is higher level and is implemented in C++ rather than using this instruction. The effect is the same however: a peer-to-peer control exchange. However the direction of control transfer with branch-and-link is fully determinate.

The branch-and-link instruction involves a service calls and so the same restrictions apply to it.