Felix Type System

John Skaller

April 11, 2020

Contents

Chapter 1

Synopsis

1.1 Nominal Types

There are three primary nominal types.

1.1.1 Primitives

Primitives, including polymorphic types, can be lifted from C++:

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

type vector[T] = "std::vector<?1>"
  requires header "#include <vector>"
;
```

1.1.2 Products

Nominally typed products can be lifted from C++:

```
cstruct point {
  x : int;
  y : int;
} requires header """
  struct point {
    int x;
    int y;
  };
""";
```

or defined in Felix:

```
struct point {
  x : int;
  y : int;
}
```

1.1.3 Variants

Usually defined in Felix:

```
variant list[T] =
  | Empty
  | Cons of T * list[T]
;
```

1.2 Structural Products

1.2.1 Unit

A unit is a product of no components:

```
var u : unit = ();
var u2 : 1 = ();
```

The unit projection:

```
uprj: 1 -> 0
```

exists but cannot be applied [NOTE NOT IMPLEMENTED, conflicts with a procedure accepting unit]

1.2.2 Identity Function

A product of one component has the identity function as a projection:

```
ident of T
```

[FIXME: special form in grammar "ident of texpr", can't use overloading.]

1.2.3 Array

An array is an an indexable collection of at least two values of a common type:

```
var a : int ^ 3 = 1,2,3;
```

The index type must be compact linear. Arrays have projections taking a value of the index type as an argument:

```
aproj: N -> T^N -> T
```

[FIXME: special form in grammar "aproj expr of texpr", can't use overloading.]

1.2.4 Tuple

A tuple is a heterogenous collection:

```
var a : int * string * double = 1, "Hello", 42.9;
```

If all the components have the same type, it is an array. Tuples have constant projections:

```
proj: N -> T0 * T1 * .. Tn -> Tk
```

where the argument selects the component. [FIXME: special form in grammar "aproj expr of sexpr" requires an integer constant, and won't accept a sum type N where N is the number of components]

1.2.5 Generalised Projections

There is a tuple projection accepting a non-constant index called a generalised projection, it returns a value of the dual sum type.

```
gproj: N -> T0 * T1 * .. Tn -> T0 + T1 + ... Tn
```

[FIXME, not implemented]

1.2.6 Record

A record is a heterogenous collection of labelled components:

```
var r : (x:int, y:int) = (x=1,y=2);
```

An empty record is the unit value.

The labels in a record can be repeated:

```
var s : (x:int, y:int, x:int) =
  (x=42, x=1,y=2)
;
```

Records are identified by a stable sort on the field labels.

Record labels can be omitted, in which case they're considered as an empty string with the lowest sort order. If all the labels are omitted, the record is a tuple.

```
var d : (x:int, string, double, int) =
  (=21,n""="hello",x:42,99)
;
```

Note the first component of a record type must include an equals sign even if the label is empty for parsing.

Records of one element are distinct from component value if and only if the label is not omitted.

The label of a labelled record component can be used as a projection. It returns the first component if there is more than one.

Other components can be accessed by first using the `getall` construction on the selected field, which returns a tuple, then applying a constant projection to that. [FIXME: there should be a single rproj construction]

1.2.7 PolyRecord

A polyrecord is a record with one special component which must be a type variable or polyrecord; the type variable must be instantiated to a polyrecord. Polyrecords were designed to support row polymorphism:

```
fun f[T] reflect45(r:(x:int,y:int| extra:T)) =>
  (x=r.y, y=r.x | extra=r.extra)
;
```

On monomorphisation the extra component must reduce to a record, the fields of which are appended to the head, resulting in a record.

Polyrecords support the same projections as records, including the extra component.

1.2.8 Pointer Projections

All projections of product type T are overloaded to accept read, write, and read-write pointers to T, and return a corresponding pointer to the selected component.

1.3 Structural Coproducts

1.3.1 Void

The type `void` or `0` is the sum of no components, it is uninhabited. There are no values of this type. It does not have an injection.

1.3.2 Sums

Indexed sums of two more more components have injections from a compact linear type to a sum type:

```
case: N -> T0 + T1 + .. Tn
```

The argument can be compact linear. Sums are decoded in pattern matches by:

```
match x with
| case 0 of v => ..
| case 1 of v => ..
...
endmatch
```

[FIXME: Only unitsum indices are supported]

Note the identity function is an injection for a single valued sum.

1.3.3 Coarrays

If all the components of a sum have the same type, it is called a coarray or repeated sum. A injection:

```
ainj : N -> T -> N *+ T
```

can be used to construct it. Note that the operator `*+` is isomorphic to the product `*` but unfortunately not identical. This is because Felix optimised representations.

1.3.4 Polymorphic Variants

Polymorphic variants are the dual of records with unique labels, however they do not currently supported repeated labels in the type. They are constructed with a label and argument and require a subtyping coercion to including in a type with more cases. The coercion is applied implicitly in some circumstances.

```
typedef pv = (`A | `B of int);
fun f() : pv => `B 42; // implicit coercion to pv
```

Polymorphic variants support open recursion.

1.3.5 Pointers

Felix has read only, write only, and read-write pointers to any type:

```
var x = 1;
var rx = &<x; //read only
var wx = &>x; // write only
var px = &x; // read-write
```

Note that the pointer formation can only be applied to `var` variables.

1.3.6 Functions

Felix has three kinds of functions:

```
D -> C      // standard function
D ->. C     // linear function
D --> C     // C function
```

Currenty generators use the same type as a standard function. Note that the types given above are the types of function values, that is, exponentials.

1.3.7 Subtyping

Felix supports a systematic set of subtyping rules for structural types. Monomorphic nominal types can be subtyped as well by providing explicit coerions. Nominal subtyping must be transitive and acyclic.

1.4 Type Classes

1.5 Kinds

Chapter 2

Routines

Felix has several different kinds of routines.

2.1 Displays

A display is simply a list of pointers to the last activation record of the calling parent, grand parent, and all ancestors, up to and including the top level thread frame object.

An activation record containing a display is called a closure. Note that closures capture activation record addresses not the values in them, so that if an activation record contains a mutable variable which is changed, the closure will see the changed value too.

2.2 Functions

Felix supports three function types:

```
D -> C      // standard function
D ->. C     // linear function
D --> C     // C function
```

2.2.1 Standard Function

The standard and linear functions are represented by an object containing a display, and containing a non-static member named `apply` which accepts as an argument a value of the function domain type, and returns a value of the function codomain type.

The caller address for a function is stored on the machine stack.

The representation of a Felix function is a C++ class derived from an abstract class which represents the function type. The apply method is virtual. Closures of functions are just pointers to an instance of the function. The C++ class constructor of a function accepts the display values and saves them in the function object, this forming an instance object.

The run time function pointer can be safely upcast to the type of the function, thus allowing a higher order function to accept and return function pointers. Note that function objects are heap allocated by default although the optimiser can stack allocate function objects if it is safe.

2.2.2 Function example

Given the following Felix function

```
fun f(x:int)=>new (x + 1);
var k = f;
println$ *(k 42);
```

the C++ header generated includes this code:

```
//TYPE 66321: int -> RWptr(int,[])
struct _ft66321 {
  typedef int* rettype;
  typedef int argtype;
  virtual int* apply(int const &)=0;
  virtual _ft66321 *clone()=0;
  virtual ~_ft66321(){};
};
```

which defines the function type, int a read-write pointer to int. The actual function is derived from this type:

```
struct f: _ft66321 {
  thread_frame_t *ptf;      // display: thread frame only

  int x;                    // parameter copy
  f(FLX_FPAR_DECL_ONLY);    // constructor
  f* clone();               // clone method
  int* apply(int const &);  // apply method
};
```

The contructor and clone method are given by:

```
//FUNCTION <64762>: f: Constructor
f::f(_ptf):ptf(_ptf){}

//FUNCTION <64762>: f: Clone method
f* f::clone(){
  return new(*PTF gcp,f_ptr_map,true) f(*this);
}
```

The macros in the constructor just pass the thread frame pointer.

The implemetation of the apply method is:

```
//FUNCTION <64762>: f: Apply method
int* f::apply(int const &_arg ){
  x = _arg;
  return (int*)new(*ptf->gcp, int_ptr_map, true) int (x + 1 );
}
```

The calling sequence is evident in the init routine:

```
//C PROC <64766>: _init_
void _init_(thread_frame_t *ptf){
  ptf->k  = (FLX_NEWP(f)(ptf)); //init
  {
    _a17556t_66323 _tmp66332 =
      ::flx::rtl::strutil::str<int>
      (
        *(PTF k)->clone()->apply(42)
      ) + ::std::string("\n")
    ;
    ::flx::rtl::ioutil::write(stdout,((_tmp66332)));
  }
  fflush(stdout);
}
```

2.2.3 Optimisation

The standard model for a function can be replaced by a more efficient representation. Felix can use an ordinary C function sometimes, or even eliminate the function by inlining. The display can also be reduced to contain only pointers that are required. In particular special binders can force the elimination of the display so that the resulting function, if it compiles, is not only C callable, but also has no direct access to Felix specific resources such as the thread frame or garbage collector.

2.2.4 C function type

For the C function type, a trick is used. Since C has no concept of a tuple, if the domain has a tuple type, the components are passed separately.

A C function can be used wherever a Felix function is required, this is done by generating a Felix function wrapper around the C function. The wrapper generation is automatic.

2.2.5 Subtyping rule

C functions and linear functions are subtypes of standard functions. Linear functions promise to use their argument exactly once.

Felix functions are not permitted to have side effects. This is not enforced, however the compiler performs optimisations assuming the rule is followed.

2.2.6 Generators

A special kind of function called a generator, which has the same type as a standard Felix function, is permitted to modify internal state.

The difference between a Felix function and generator is as follows: when a Felix function closure is stored in a variable, an invocation of the function calls a `clone` method to copy the object before calling the `apply` method on that copy. This is to ensure recursion works by ensuring the function get a separate data frame from any other invocation.

If the function was a generator, the clone method is still called but simply returns the this pointer of the object, ensuring that all invocations of the `apply` method use the same data frame. In particular state data is retained between applications, since the object is stored in a variable, and even more particularly generators can save program locations, allowing them to return a value and control in such a way that calling the `apply` method again resumes execution where it last left off.

2.3 Procedures

A procedure is Felix is radically different to a function. Procedure are given the type

```
D -> 0
```

where the 0 suggests that procedures do not return a value. The representation of a procedure consists of three methods.

1. constructor, creates closure

2. call method, binds arguments
3. resume method, steps procedure a bit

The procedure constructor, like a function, accepts the display to form a closure object. The call method, applied to the closure, binds the procedure argument into the data frame. The result is a new kind of object called a continuation object.

2.3.1 Continuation Object

Continuation objects all have the same type, `con_t` which is a base class for the procedure type abstraction, which in turn is a base for the actual procedure. The call method is a member of the type object, the resume method is a member of the continuation object.

```
namespace flx {namespace rtl {
struct con_t                    // abstract base for mutable continuations
{
  FLX_PC_DECL                   // interior program counter
  union svc_req_t *p_svc;       // pointer to service request

  con_t();                      // initialise pc, p_svc to 0
  virtual con_t *resume()=0;    // method to perform a computational step
  virtual ~con_t();
  con_t * _caller;              // callers continuation (return address)
};
}} //namespaces
```

Continuation objects all contain a variable called the program counter which tells where execution is up to within the procedure. Initially, it is set to the start of the procedure. When the resume method is called, some work is done, and then the procedure returns a pointer to a continuation object. It may be the same object, however if the procedure is returning, it could be the callers continuation, and if the procedure is calling another, it could be the called procedure's continuation object. A NULL is returned when the original procedure is finished.

Continuation objects are suspensions, the resume method is a function that in effect accepts a suspension and returns another suspension. The role of suspended computations will soon become evident!

2.3.2 Routines

For completeness, a routine is a procedure for which the current continuation is not passed implicitly. Felix currently uses procedure representation for routines and passes the caller address anyhow.

2.3.3 Coroutines

A coroutine is a procedure which, directly or indirectly, does channel I/O. It has the same representation as a procedure.

Technically, all procedures in Felix are coroutines. This includes the mainline!

2.4 Procedure operation

Procedures are run by repeatedly calling the resume method of a continuation object until NULL is returned. In Felix, the loop is also sometimes enclosed in a try/catch block to implement special semantics. The basic run loop is extremely simple and lightning fast:

```
while (p) p = p -> resume();
```

2.4.1 Resume Method structure

The resume method of a procedure for sequential flow is very simple:

```
con_t *resume() {
   switch (PC) {
   case 0: ... PC = 1; return this;
   case 1: ... PC = 2; return this;
      ...
   case 42: ... return caller;
   }
 }
```

In each case, at the end, the current continuation is simply the next case, except at the end, when the current continuation of the caller is returned.

2.4.2 Procedure Call

A procedure call is implemented by constructing a new procedure object, binding its arguments, and setting its caller to `this`, then returning that new procedure pointer. The call operation binding the parameters also sets the PC to 0 to ensure it starts in the right place.

2.4.3 Procedure Return

This is implemented by simply returning the stored caller's this pointer. When the driver loop calls resume() that procedure will continue where it left off.

2.4.4 Other operations

Procedures can also do various tricky things. For example the driver above can be augmented:

```
void run(::flx::rtl::con_t *p)
{
  while(p)
  {
    try { p=p->resume(); }
    catch (::flx::rtl::con_t *x) { p = x; }
  }
}
```

This allows a procedure to throw a continuation object which then replaces the existing continuation. This routine is in the standard library.

2.4.5 Spaghetti Stack

Continuations are linked in two ways to form a spaghetti stack: there is a list of callers, terminated by NULL for the top level procedure call, and there is notionally a list of ancestors, particularly the most recent activation records of the ascestors, maintained through the parent pointer. Felix actually uses a display, so that every procedure contains a pointer to every ancestor, including the base ancestor, the thread frame. This reduces access time at the cost of additional storage in nested frames, however unused pointers can be optimised away.

2.4.6 Utility

The technology described above has very little utility in itself. There is no good reason for ordinary procedural code to return control to the driver, only to be called again immediately. The utility will become evident in the next section!

2.5 Procedure example

This example is a bit more complex that for functions;

```
proc f (x:int) { println$ x; }
proc g (x:int, h: int -> 0) {
  h x;
  h (x+1);
}
```

```
var k = g;
k (42, f);
```

The procedure types are:

```
//TYPE 66319: int -> void
struct _pt66319: ::flx::rtl::con_t {
  typedef void rettype;
  typedef int argtype;
  virtual ::flx::rtl::con_t *
    call(::flx::rtl::con_t *, int const &)=0;
  virtual _pt66319 *clone()=0;
  virtual ::flx::rtl::con_t *resume()=0;
};
```

and

```
//TYPE 66321: int * (int -> void) -> void
struct _pt66321: ::flx::rtl::con_t {
  typedef void rettype;
  typedef _tt66320 argtype;
  virtual ::flx::rtl::con_t *
    call(::flx::rtl::con_t *, _tt66320 const &)=0;
  virtual _pt66321 *clone()=0;
  virtual ::flx::rtl::con_t *resume()=0;
};
```

the functions are declared as:

```
//------------------------------
//PROCEDURE <64762>: f int -> void
//    parent = None
struct f: _pt66319 {
  FLX_FMEM_DECL

  int x;
  f(FLX_FPAR_DECL_ONLY);
  f* clone();
  ::flx::rtl::con_t *
    call(::flx::rtl::con_t*,int const &);
  ::flx::rtl::con_t *resume();
};
```

```
//------------------------------
//PROCEDURE <64766>: g int * (int -> void) -> void
//    parent = None
struct g: _pt66321 {
  FLX_FMEM_DECL

  _pt66319* h;
  int _vI64768_x;
  g(FLX_FPAR_DECL_ONLY);
  g* clone();
  ::flx::rtl::con_t *call(::flx::rtl::con_t*,_tt66320 const &);
  ::flx::rtl::con_t *resume();
};
```

The procedure f is defined with methods:

```
//PROCEDURE <64762:> f: Call method
::flx::rtl::con_t *
f::call(::flx::rtl::con_t *_ptr_caller, int const &_arg){
  _caller = _ptr_caller;
  x = _arg;
  INIT_PC
  return this;
}

//PROCEDURE <64762:> f: Resume method
::flx::rtl::con_t *f::resume(){
  {
    _a17556t_66323 _tmp66332 =
      ::flx::rtl::strutil::str<int>(x) + ::std::string("\n") ;
    ::flx::rtl::ioutil::write(stdout,((_tmp66332)));
  }
  fflush(stdout);
  FLX_RETURN // procedure return
}
```

and the procedure g with methods:

```
//PROCEDURE <64766:> g: Call method
::flx::rtl::con_t *
g::call(::flx::rtl::con_t *_ptr_caller, _tt66320 const &_arg){
  _caller = _ptr_caller;
  _vI64768_x = _arg.mem_0;
  h = _arg.mem_1;
```

```
    INIT_PC
    return this;
  }

  //PROCEDURE <64766:> g: Resume method
  ::flx::rtl::con_t *g::resume(){
    FLX_START_SWITCH
        FLX_SET_PC(66331)
        return (h)->clone()->call(this, _vI64768_x);
      FLX_CASE_LABEL(66331)
        {
          ::flx::rtl::con_t *tmp = _caller;
          _caller=0;
          return (h)->clone()
        ->call(tmp, _vI64768_x + 1 );//tail call (BEXE_jump)
        }
        FLX_KILLPC
        FLX_RETURN // procedure return
        FLX_KILLPC
      FLX_RETURN
    FLX_END_SWITCH
  }
```

This illustrates the derivation heirarchy of procedures, and also shows how a procedure is called. The first call sets the program counter to the next operation after the call, then invokes the procedure call method, passing the current object this pointer as the caller, along with the argument. The resulting continuation pointer is returned to the driver loop. The called procedure is resumed which causes it to print its argument, then it returns the callers pointer.

The second call has an optimisation. Instead of passing this as the caller continuation, it passes g's caller instead. When the f procedure returns it returns that instead of this. This is possible because the second call is the last thing done, the optimisation is known as tail call optimisation. In fact the compiler is generating four extra instructions after that which will never be executed.

This example was contrived to prevent the compiler optimising the procedures to ordinary C ones. We had to force the generation of closures by storing g in a variable k, and passing f to it as a parameter.

2.6 Fibration

Fibration is a method of interleaving control between fibres.

2.6.1 fibre

A fibre, or fthread (Felix thread) is an object containing a continuation pointer. in Felix, fibre objects also contain a pointer to another fibre, possibly NULL, which is used as decribed below.

```
struct RTL_EXTERN fthread_t // fthread abstraction
{
  con_t *cc;          // current continuation
  fthread_t *next;    // link to next fthread

  fthread_t();        // dead thread, suitable for assignment
  fthread_t(con_t*); // make thread from a continuation

  svc_req_t *run();  // run until dead or service request
  void kill();        // kill by detaching the continuation
  svc_req_t *get_svc()const;   // get current service request
private: // uncopyable
  fthread_t(fthread_t const&) = delete;
  void operator=(fthread_t const&) = delete;
};
```

2.6.2 synchronous channel

A synchronous channel, or schannel, is a set of fibres, possibly empty, all of which are either waiting to read or waiting to write. It is represented in Felix by linking fibres together through the special link, and storing the head pointer in the schannel object, setting the low order bit of the pointer to 1 if the fibres are waiting to write.

```
struct RTL_EXTERN schannel_t
{
  fthread_t *top; // has to be public for offsetof macro

  void push_reader(fthread_t *); // add a reader
  fthread_t *pop_reader();       // pop a reader, NULL if none
  void push_writer(fthread_t *); // add a writer
  fthread_t *pop_writer();       // pop a writer, NULL if none
  schannel_t();

private: // uncopyable
  schannel_t(schannel_t const&) = delete;
  void operator= (schannel_t const&) = delete;
};
```

2.6.3 scheduler

A scheduler is an object with two variables, one which contains the running fibre, and the other of which represents a set of fibres called active fibres. These Felix representation uses the special link in the fibre to represent the set of active fibres, and may be NULL.

The scheduler is a C++ procedure, which returns control when both there are no active fibres left and the running fibre is exhausted.

2.6.4 Channel I/O

There are two operations for channel I/O, read an write. They are identical except that data is transfered from the writer to the reader. In Felix, the data is always a single machine word. Read and write are said to be matching operations.

Read operates as follows. If you read a channel which is empty, or contains only readers, then the fibre is added to the channel reader list, and is removed as the running fibre.

If the channel contains a writer, one is selected and removed from the list of waiting writers, data is transfered from the writer to the reader, the reader is removed as the running fibre, and both reader and writer are made active.

Write operation is dual, except the direction of data transfer is still from the write to reader.

2.6.5 Scheduler operation

The scheduler runs the current fibre using the driver loop. However in addition, it also checks a special code in the continuations called a service request. This is a request to do a channel read, channel write, suicide or spawn operation.

The read an write have already been described. Suicide simply removes the currently running fibre. It is usually implemented by setting the fibre's continuation pointer to NULL rather than using a service request.

Spawn takes a newly created fibre and makes it active, also making the currently running fibre active but no longer running.

The scheduler operation is to start with some fibre as running, and then, after a service request, the previously running fibre is no longer running, so the scheduler picks another one to run, removing it from the active set. If the active set is empty, it returns control.

Although the formal semantics allow the scheduler to randomly pick any active fibre to run, in Felix the operation is deterministic. When a reader and write match up on an I/O request the reader is always made active first, and the writer ends up on the top of the active list so it will run next if not displaced.

Spawn always pushes the running fibre onto the active list and runs the newly spawned fibre. This means that spawning and procedure calling are identical if the spawned fibre makes no service requests.

2.7 Performance

It's important to understand why the fibration model works as it does.

A fibre is just a pointer to the top level continuation of a spaghetti stack, so it is a single machine pointer. The scheduler has two pointers, one to the running fibre, and one to the active list.

Channels are a single machine pointer.

Pushing to a list requires only one read and two writes, one to update the top pointer, and one to save the next pointer in the continuation being pushed. Similarly popping requires only one write, updating the top pointer.

I/O operations involve only pushing and popping, and changes to the scheduler state involve only changing the active list and currently running fibre, therefore context switches are extremely fast.

With pthreads, context switches are extremely expensive, as they involve machine stack swapping. With Felix fibres, the stacks are heap allocated linked lists identified by a single pointer, so stack swapping is lighting fast.

Felix thus obtains extremely high performance, faster than any other context switching machinery, whilst at the same time preserving C/C++ compatibility. Functions still use the machine stack, procedures are coroutines and use spaghetti stacks.

Current work in progress is modifying the scheduler to support concurrency. A single active list is maintained, but multiple running fibres are allowed. In effect these fibres are elevated to concurrent processes. Because the core operations only involve extremely fast pointer reads and writes, they can be protected by spinlocks using atomic operations, so that the base machinery can provide wait free concurrency (one CPU is always making progress). The performance gain depends on the level of contention. With extremely high contention, performance is slower than a non-concurrent configuration. However with fibres doing more work and less I/O, contention drops and performance gain is effectively linear in the number of CPUs.

The biggest hit is memory allocation. Felix uses `malloc()`, which is thread safe, but may need to use a mutex and or swap to the OS to obtain storage. Descheduling a fibre this way is not observed by the user space scheduler, and so effectively hangs one of the CPUs. It's also possible, though unlikely, that a fibre holding a spinlock will be pre-empted, which would block all other fibres contending for that spinlock.

List of Listings