felix articles

Release 2016.07.12-rc1

Aug 01, 2020

Contents

1	Using Uniqueness Types	3
2	Subtyping in Felix	15
3	Open Recursion	21
4	Cofunctions	27
5	Coroutine Basics	31
6	What’s Wrong With C++	33
7	Indices and tables	41

Contents:

CHAPTER 1

Using Uniqueness Types

1.1 Ownership

Uniqueness types provide a way to help enforce a contract of exclusive ownership. Lets look at an example to see how they work.

1.1.1 Raw Operations

The first thing we're going to do is provide a Felix binding to some basic C string manipulation routines.

Raw Memory Manipulation

```
class Memory
{
  gen malloc: !ints -> address = 'checked_malloc($1)'
    requires Cxx_headers::cstdlib, checked_malloc
  ;

  proc free: address =
    "::std::free($1);"
    requires Cxx_headers::cstdlib
  ;

  gen realloc: address * !ints -> address =
    "::std::realloc($1,$2)"
    requires Cxx_headers::cstdlib
  ;
}
```

Raw C String Manipulation

```
class CString
{
  //$ C strcpy.
  proc strcpy: +char * +char =
    "(void)::std::strcpy($1,$2);"
    requires Cxx_headers::cstring
  ;

  //$ C strncpy.
  proc strncpy: +char * +char * !ints =
    "(void)::std::strncpy($1,$2,$3);"
    requires Cxx_headers::cstring
  ;

  //$ C strlen: NTBS length.
  fun strlen: +char -> size = "::std::strlen($1)"
    requires Cxx_headers::cstring
  ;

  fun len (s:+char) => strlen s;

  //$ Traditional NTBS strdup.
  gen strdup: +char -> +char =
    "::flx::rtl::strutil::flx_strdup($1)"
    requires package "flx_strutil"
  ;
}
```

1.1.2 Problems with Raw Operations

The raw operations shown above are difficult to use safely, whether you're writing code in C, C++, or Felix.

What can go wrong?

Forgetting to Free

After you *malloc* a string, you have to eventually *free* it or you get a memory leak. It is easy to forget to do this.

Double Free

An even worse problem is freeing the memory you allocated twice. This may cause your program to abort, particularly if you have a modern checking allocator. Historically, however, the double free simply corrupted memory, and all sorts of weird things could happen. If you were lucky, you would soon get a core dump, if you were unlucky, your program ran but produced the wrong results.

Dangling Pointers

Another problem which occurs is that you free the pointer, but then go on and use it anyhow. This is known as a dangling pointer, since it points off into the wild blue yonder. Again, if you're lucky, you might get a program abort quickly, but it is easy to be unlucky.

Invalid Pointers

Because allocators often maintain free lists, with the most recently freed memory at the top, if you free your memory, thinking it is gone, but retain a pointer to it somewhere, then allocate a new block of memory, it is quite likely to be to the same location as the last memory you freed.

Now, your old pointer is not actually dangling, it is pointing, unexpectedly, into the new string you allocated. If you then write through that pointer, the memory is changed at mysterious modifications appear in the string, even though you have not made any changes through the new pointer you allocated.

1.1.3 C++ Strings: Encapsulation

The very first class every C++ programmer wrote was a string class.

The technique used was basic object orientation. The idea is to hide the pointer in a C++ class, by making it a private member, and provide public methods that safely manipulate the string, without revealing it to the programmer.

This technique is using *abstraction* for the purpose of *hiding representation details*. It is a good method, but C++ string classes had their problems.

Lack of facilities

A key problem with any string class is that since you cannot access the underlying pointer directly, you may want to do something to the string which either cannot be done, or can only be done inefficiently.

Most string classes programmers wrote started off simple, but the programmer had to come back time and again, to add new methods to the class so things could be both efficient and safe.

For this reason most string classes acquired a mix of two flavours to solve this problem: the first was to provide a rich, kitchen sink of methods that covered as much as experience showed was required.

The second method was to provide a cheat method that did actually expose the underlying pointer.

Copying

A second serious problem with string classes was that in order to ensure the user could modify the string, without the modifications turning up unexpectedly in someone elses string, the string had to be copied quite often.

C++ did this copying using a copy constructor, so that, for example, when you pass a string to a function, the function is free to modify it.

The cost of copying is reduced in C++ by using const references, however this method is not safe either. The problem is, the same string can be passed as both a const and non-const reference, and the function receiving them can modify the non-const version, and the const version mysteriously changes.

This is an example of a general class of problems known as aliasing problems, characterised by the existence of a single object with multiple names, or, more precisely, multiple access paths.

Move Constructors

In C++11 a major advance was made due to the introduction of rvalue references. An rvalue reference can only bind to an rvalue, and rvalues are always unique. So an rvalue passed to a function with an rvalue reference parameter can safely modify the underlying memory, because the type system ensures it is the exclusive owner.

C++ uses this feature primarily by allowing a so-called move constructor, which, instead of copying the underlying memory, simply moves the pointer from the argument object to the parameter object, leaving the argument evacuted.

It helps a lot, providing reasonable safety and improved performance, but we can do better!

1.1.4 What is the Real Problem?

We need to fully understand the actual problem here. The difficulty arises because the pointer and the memory it refers to are *decoupled*. They're different things that have to be kept in sync. You can copy the memory, and copy the pointer, separately.

The C++ class enforces coupling, and it enforces ownership by copying. The more recent use of move constructors leverages the type system to gain performance by using the knowledge that rvalues are unique.

So the problem is about coupling, and we can state this another way: its about ownership. If there are many copies of the pointer for one memory block, ownership is shared. If there is only one, it is exclusive.

If we can enforce exclusive ownership, mutations are always safe, in particular, exclusive ownership implies a one to one correspondence between the representative of a value and the value itself. It means, if the string is deleted, the pointer must be inacessable.

1.2 Uniqueness Types

Felix provides some machinery to further aid in establishing and maintaining knowledge of, and the ability to, reason about ownership: uniqueness types.

The facility is used to enforce a contract, but it does not provide a global safety guarrantee. Our system provides a type constructor *uniq* which can be applied to any type.

We also provide three operators which can be applied to expressions. The *box* operator takes a value of some type T, and returns a value of type *uniq T*. The *unbox* operator takes a value of type *uniq T*, for some type T, and returns a value of type T.

These operators are type coercions which have no run time impact. The uniq typing is erased by the compiler in the back end after type checking, ensuring there is no run time penalty for unique typing.

In addition there is an unsafe cheat operator *peek* which can be applied to a read-only pointer to store of a uniq type, which returns the stored value.

Finally there is an unsafe procedure *kill* which consumes a uniq value without doing anything. It can be used when the value has been consumed in a way that has escaped the notice of the type system, such as by use of a pointer, not notify the type system that the value is dead. It has no utility unless applied to a variable.

```
open class Unique
{
  // box up a value as a unique thing
  fun box[T] : T -> _uniq T = "($t)";

  // unsafely unpack the unique box
  fun unbox[T] : _uniq T -> T = "($t)";

  // kill a live unique value
  proc kill[T] : uniq T = ";";

  // functor for typing
  typedef fun uniq (T:TYPE):TYPE => _uniq T;

  // peek inside the box without changing livenes state
  fun peek[T] : &<(uniq T) -> T = "*($t)";
}
```

In this code the type operator *_uniq* is the compiler intrinsic, the functor *uniq* is used to provide the public version.

The system also provides conversions to strings which delegate to the conversions for the underlying type, not shown here.

1.2.1 Example: UniqueCStrings

This example presents a cut down version of the Felix standard library component *UniqueCStrings* which illustrates a real use of uniqueness typing.

Setup

We are going to use a class to encapsulate our methods so we start like this:

```
open class UniqueCStrings
{
  open CString;
  open Memory;
```

We have included the unsafe raw operations inside the class privately for our use.

Abstraction

We're going to steal the OO idea and make out representation type abstract.

```
// abstract representation
private type _ucstr = new +char;
```

In Felix, the *type* binder introduces an abstract type. The RHS of the construction may be either the *new* operator followed by a type expression, or it may be a C++ type wrapped in a string. In the above the type +*char* means an incrementable non-null pointer to an array of *char*.

As well as being abstract, we're also preventing the name *_ucstr* from being visible outside the enclosing class *UniqueCStrings*.

Uniqueness

We're going to make the publically visible version a unique type.

```
// make it uniq
typedef ucstr = box _ucstr;
```

The type constructor *uniq* specifies a uniquely typed version of the type it qualifies.

Internal Access

Felix enforces abstraction fully. It does this by providing two methods for an abstract type, both of which are private so they can only be used inside the class defining the abstract type.

These methods are *_repr_* which casts the abstract type to its implementation, and *_make_ucstr* which casts the representation to the abstraction.

Since these are a bit messy to write, we will provide private wrappers functions:

```
// privatise access to representation
private fun pack (p: +char) => p._make__ucstr.uniq;
private fun unpack (var p: ucstr) : +char => p.unbox._repr_;
```

You should think of *pack* as a way to take a raw char pointer and wrap it up in a package you can move about. You can pass this box from one variable to another. If a function has a local variable, we can say the function owns that variable. The variable is like a cupboard, into which you can put things.

You can take the box out of one cupboard, and put it it another, but you cannot easily copy the box, because you cannot see inside it.

When you have a new box put in your cupboard, you can take the box out of the cupboard and *unpack* it to find out what is inside. You can then play with its contents with relative safety, knowing that it is exclusively yours to play with. Its important to note, there isn't space to unpack your box inside the cpuboard. Its like a mail box, you have to take your letter out, emptying the mail box, before you can open the letter.

Constructors

Now we need a constructor. We're going to use a C++ string which is a Felix string and copy its value into an a C array using the method *_unsafe_cstr*:

```
// Constructors
ctor ucstr (var s:string) = {
   // malloc'd copy of string contents
   var p =  s._unsafe_cstr;
   return pack p;
}
```

What *_unsafe_cstr* does is malloc a new array and copy the contents of the C++ internal array into it, it used the C++ method *c_str()* to gain access to the internal C array. This is the C++ *string* class cheat method. Our *_unsafe_cstr* is not safe because it is returning a raw pointer which we might forget to free, but we're trying to fix that by coercing it to a unique type. By wrapping into a box.

Another constructor just copies an existing C string and packs it up into a unique type:

```
ctor ucstr (s:+char) => s.strdup.pack;
```

You can see here our code is doing unsafe things, with raw C strings, but we are then using *pack* to at least notify our client. Because the constructor returns a unique type, the client of the constructor believes they have exclusive ownership of the returned value.

And that indeed is the intent and purpose of our constructor code, and we can easily verify we have met our part of the bargain.

Destructor

We need to provide a way to free our string:

```
// deletes the store
proc delete (var p:ucstr) {
  var q = unpack p;
  free q;
}
```

How do we know it is safe to free the underlying pointer here? The answer is, as the client of the unique value, we are entitled to believe that we are the exclusive owner of it. It has been moved to the variable *p* which is exclusively our variable, and it has a type which indicates it holds an exclusively owned value.

The type system cannot enforce the exclusive ownership, but it does enforce a useful contract. It ensures that if the caller *claims* to be the exclusive owner by typing the argument value *uniq* then that claim will be recognised by the callee routine because the type system will *ensure* that the parameter is also typed *uniq*.

In other words, this is a coupling contract. The argument passing to the parameter is a contract of transfer of ownership *witnessed* by the type system. All bets are off, before the client signs the contract by wrapping the value with a *uniq* operator. All bets are off, after the service routine signs the contract by accepting the packed value, and then unpacks it.

What the contract enforces is an agreement that the value is *moved* from the client routine to the service routine, instead of being copied.

Display

We have to be able to see our strings. We already have facilities to see Felix strings which are bindings of C++ strings. So we will leverage our generic classes *Str* and *Repr* which allow us to specify a default way to convert any particular type to a standard string:

```
instance Str[_ucstr] { fun str(p:_ucstr)=>p._repr_.str; }
instance Repr[_ucstr] { fun repr(p:_ucstr)=>p._repr_.repr; }

inherit Str[_ucstr];
inherit Repr[_ucstr];
```

The instances there implement the functions *str* and *repr* which are for displaying as human readable text and machine readble programming language level literals, respectively.

Then, we yank bindings to these methods into our class with the *inherit* directives.

What is important to observe about these routines is that they do not operate on the unique type. The values that they receive are abstracted representations of the underlying C pointer which we observe using the private *_repr_* method. What this means, is that you cannot apply the *str* operator to a packed up box, it won't work on a *uniq* value.

The box has to be unpacked first. Here, we're using abstraction to ensure that we can provide this operation to the user without needing to expose the underlying pointer, but the user must already own the value and have taken responsibility for its safe management by unpacking a unique value.

Length

How long is a piece of string?

```
// length
fun len(var s:&<ucstr) : size => s.peek._repr_.strlen;
```

This method accepts a read-only pointer to a *ucstr* because it only wants to inspect it. Such a pointer is unsafe in general! We need to examine the code carefully to see that whilst we have unsafely acquired a reference to the *ucstr*, we have used it and then promptly forgotten it.

The pointer we acquired is an *alias* so retaining it would threaten ownership. But we used it and forgot it immediately, so all is well. The *peek* operator you see above is used to look inside a unique type, which is not safe in general, it is only safe if you only take a peek.

The machinery of taking the address of a *uniq* value and then passing it to a client is known as *lending* the value. It is not safe in general to lend something to a service you do not trust. We can trust this function, because we can see its implementation. In particular we can see that whilst it does *peek* inside the packed up box, in order to calculate the length of the string, it does not pass on the secret knowledge to anyone else, it returns only the length, not the pointer to the C array it peeked at. Similarly, the *len* function itself has trusted that the C *strlen* function has only used the supplied pointer transiently.

Modifying One Character

Now we're going to modify one character. We use an unsafe function, *Carray::set* to so this. It operates on a pointer to a C array, or *+char* type. Here we make no assurance that the location being set is inside the string. This operation, therefore is unsafe, in that the index could be out of bounds. Our concern here is not with the validity of the bound, but that, assuming the bound is indeed valid, we can safely modify the string in place and return it, without anyone observing we did so.

```
// modify one char
fun set (var s:ucstr, i:int, c:char) : ucstr =  {
  var cs = unpack s;
  Carray::set (cs, i, c);
  return cs.pack;
}
```

Now, we reason that, because *s* has a *uniq* type, we are the sole owner, so no one else can observe any change we make. So instead of making a copy of the string and modifying it, we can safely modify the original.

That is the key to uniqueness types. Using the type system to reason that a mutation cannot be a side effect, because, whilst there is actually an effect in the form of a mutation, it can only be observed in the output of the function. It cannot be observed *on the side* because no one else knows about it.

To say this another way, the caller cannot know if the string was modified, or a copy created and modified. What we gain is this: if the caller believes they have exclusive ownership, the caller can sign the transfer of ownership contract by making the type unique. In doing so they enable a significant optimisation whereby the need to copy the string to avoid a side effect is removed, a significant saving in both time and memory.

The saving is actually considerably greater because the system is relieved of the cost of calculating whether the string is reachable from any live part of the program, a calculation which is normally done by the garbage collector.

Appending two strings

Now we get to a more difficult routine.

```
// append: consumes y
fun append (var x:ucstr, var y:ucstr): ucstr = {
  var cx = unpack x;
  var cy = unpack y;
  var lx = cx.len;
  var ly = cy.len;
  var r = realloc (cx, lx+ly+1);
  strncpy (r+lx,cy,ly+1);
  free cy;
  return pack r;
}
```

This routine takes two unique arguments, and returns their concatenation. It destroys *both* the arguments in the processes. We unpack the two arguments to their underlying pointers, and use C to calculate their lengths. Then we reallocate the first argument, which can either add more store to its end, or make a completely new copy with extra store at the end, and free the old string.

Then we add the characters of the second argument at the end of the first, in the space we just allocated, and free the second argument.

You may ask, how do we know we can destroy these arguments safely?

The answer is that we own them exclusively. No one else knows about them, we can do what we like with them. Because they have a *uniq* type.

In fact that is not quite correct. There is one thing we cannot do with a uniquely typed value: forget it. We're responsible for it, we cannot forget it. We have to either return it, handing back ownership to our caller, or free it. We freed the second argument in this routine and returned the first, with modifications, and possibly at a new location. Realloc took care of what to do if we needed a new store: it freed the old store and allocated the new store for us. That's what realloc is specified to do.

Nondestructive Append

Here's how to avoid destroying the second argument:

```
// append: doesnt consume y
noinline fun append (var x:ucstr, var py:&ucstr): ucstr = {
  var cx = unpack x;
  var cy = py.peek._repr_;
  var lx = cx.len;
  var ly = cy.len;
  var r = realloc (cx, lx+ly+1);
  strncpy (r+lx,cy,ly+1);
  return pack r;
}
```

This is similar to our destructive version, however because a pointer was passed as the second argument instead of a value, we know we're only allowed to peek at it.

The type system will not stop you, if, instead of just peeking, you dereference the pointer, unpack the resulting value, and then free it.

Felix uniq types do *not* ensure correct usage. What they actually do, is make the contract explicit. There is enforcement, but it is not complete.

Convenience wrappers

Here are some convenience wrappers:

```
// nicer appends
fun + (var x:ucstr, var y:ucstr) => append (x,y);
fun + (var x:ucstr, var py:&ucstr) => append (x,py);

proc += (var lhs: &ucstr, var rhs: ucstr) =>
  lhs <- append (*lhs,rhs)
;
proc += (var lhs: &ucstr, var rhs: &ucstr) =>
  lhs <- append (*lhs,rhs)
;
```

These wrappers allow you to use the infix + functional operator and += procedural instruction. Note carefully the procedural implementations! By derefercning the *lhs* pointer we have created a uniq value. We were only given a loan, so is this safe?

In general, it isn't safe. But we can see here that, although we have abused the loan by modifying the value, we are then storing the modified value back into the original location. We have taken the box, opened it, changed the contents, and put them back. The original owner remains the owner, although the value they own has changed.

Of course, that is not only what the owner expected, it is what the owner demanded! There's no point putting a broken phone in for repair, if the repair shop doesn't actually fix it!

1.2.2 Enforcement: An example of usage

Finally here is a simple example of usage.

```
proc test() {
  var s = ucstr "hello";
  println$ &s;
  s = set (s, 0, char "e");
  var s2 = s;
  println$ &s2;
  delete s2;
}
test();
```

This snippets hides something I haven't explained yet.

See how I copied the value in variable *s* to *s2*? *No you don't!*

I didn't copy it. I *moved it*. Felix enforces a special rule for uniq types. They cannot be copied, only moved.

Programming languages have no natural syntax for movement, only copying. So we need some help, when we do an assignment and we really mean to move, and not copy the value. Below I explain how we do that.

A variable of a uniq type has to be used *exactly once*. If you pass the value in a variable to a function, the variable goes out of scope and cannot be accessed. You owned the value, but you gave away ownership.

If you do an assignment like the one from *s* to *s2* the same thing happens. *s* loses the value and *s2* gains it. Felix won't let you use *s* now.

But *s* was already used! Yes, it was passed to *set*, but set then reinitialised it. Every variable of a unique type is either initialised or dead. Assignments from one variable to another kill the first variable and liven the second one. The first one has to be alive to start, and the second one has to be dead. After the assignment, the first one is dead and the second one is alive. Life has been moved from one variable to another. We actually say that we have transfered *ownership*, as if you have a dog in a kennel and then give the dog as a gift to a friend (who puts it in another kennel, or lets it sleep on the sofa).

Taking the address of a variable does not kill it, which means if you do take the address you must only use it whilst the variable remains alive. Felix does *not* enforce that.

So here you see the contract. Felix enforces correct use of whole variables, the programmer must enforce the correct use of pointers.

Errors

So what happens if you make a mistake? Let me show you:

```
proc test () {
  var x = box 1;
  println$ unbox x;
  println$ unbox x;
```

(continues on next page)

(continued from previous page)

```
}
test;
```

Here we broke the rules. We used x twice. And here is what Felix has to say about it:

```
~/felix>flx tmp
Once error: Using uninitialised or already used once variable
(50313:->x)
Variable x defined at
/Users/skaller/felix/tmp.flx: line 3, cols 3 to 18
2: proc test () {
3:   var x = box 1;
     ****************
4:   println$ unbox x;
```

What Felix does is a control flow analysis. The requirement is that on every *statically* possible control path, a uniquely typed variable alternates between two states: *live* and *dead*. A variable is dead until it is initialised or assigned to. Parameters of a function are considered live, since we assume they were initialised by the caller with an argument.

If the variable is passed to a function, it must be live, but at the point it is passed it is now killed and considered dead. Some people say the value has been *consumed*.

A dead variable can be relivened by assigning it a new uniq value.

At the end of a function, all uniqely typed variables must be dead.

Felix does *NOT* recognise taking the address of a variable as significant, *except* in the special case the address is immediately used as the first argument of the store at operator <-. Tracking pointer aliases is not impossible but it is hard to do properly, and it can be very expensive. Felix is a lazy cat: he helps you get things right but won't force you.

The idea here is simple: a live variable contains a wrapped box, a dead variable does not. When you move a value out of a variable, it is no longer in the cupboard so the variable is marked empty or dead. When you put something back in, the cupboard is full and the variable live again.

CHAPTER 2

Subtyping in Felix

Felix supports certain implicit conversions in certain contexts which are considered subtyping coercions. The context currently supported is the coercion of an argument to the type of a parameter in a function application or procedure call.

2.1 Context of Implicit Coercions

Felix does *not* support implicit coercions in simple assignments or local variable initialisations, even though in some sense initialisation at least is somehow equivalent to binding a parameter to an argument.

The reason is that whilst for a local variable initialisation, there is space to write the coercion explicitly in a neutral manner, the argument and parameter of a function call are lexically separated.

The rule for selection of a function from an overloaded set in the presence of implicit coercions is a generalisation of the usual subsumption rule for overloads of polymorphic functions, namely, the selection of the most specialised function from the set which matches the argument.

A parameter type A is more specialised than another B if A is a subtype of B.

This imposes a strict coherence constraint on subtyping coercions. In particular if A is a subtype of B, and B a subtype of C, then A must be a subtype of C as well, and the composition of the subtyping coercions from A to B and then B to C must be semantically equivalent to a subtyping coercion from A to C.

Furthermore, any two subtyping coercions from A to B must be semantically equivalent.

The transitivity rules has two vital consequences. The first is that the compiler must be able to calculate a composite subtyping coercion from A to C via B, if there is a coercion from A to B and from B to C. The second is that the programmer should take care that if in such circumstances a coercion is also given from A to C, it is a semantically equivalent to the composite.

Generally, the compiler must be free to pick any composition as the implementation of a coercion, and we can view the picking of an efficient composition, such as the single user defined coercion from A to C as an optimisation.

2.2 Standard Subtyping Coercions

2.2.1 Record Coercions

Record support two a stage coercion rule. The first rule is called a *width* coercion and allows fields of a record to be thrown away.

The second stage, called a *depth* coercion, permits the field values to be individually coerced covariantly. In other words a coercion from a subtype to a super type consists of discarding some fields, and then applying subtyping coercions to the values of the remaining fields.

The justification of the width coercion rule is this: if a function requires a record with a certain set of fields, the supplying a record with more fields is acceptable, because the function ignores them anyhow.

2.2.2 Tuple Coercions

Felix supports covariant depth coercions of tuples.

We do not support width coercions, however. The reason is that the programmer would be surprised if components of a tuple magically disappeared at random just to match a function signature.

In particular, since in Felix we identify a tuple of length one with that element, allowing width coercions would be tantamount to allowing a tuple to be supplied if *any* of its components could be coerced to a function parameter.

2.2.3 Array Coercions

Felix also supports covariant depth coercion of arrays with a constraint that the same coercion must be applied to each element of the array.

We do not support implicit with coercions because the programmer might be surprised if an array was magically truncated.

2.2.4 Polymorphic Variant Coercions

Polymorphic variant coercions also support two stage coercions, in reversed order: for the first stage we can covariantly coerce the constructor arguments, and in the second stage add additional constructors.

The justification of the width coercion rule is this: if a function requires a polymorphic variant from a certain set of cases which it analyses, then the analysis will completely handle less cases.

2.2.5 Function Value Coercions

Function values are coerced contravariantly on their domain and covariantly on their codomain.

The justification is as follows. Suppose we have a function that accepts another function as an argument. When we apply that function to a value, it must handle all the argument values that the function can throw at it. Therefore the domain of the function supplied must be a supertype of the domain of the parameter.

Conversely, it is fine if the supplied function returns a more restricted set of values than is required in the context in which it is supplied, thus, the codomain of the argument can be a subtype of the codomain of the parameter.

2.2.6 Machine Pointer Coercions

Felix has three core pointer kinds: read-only pointers, write-only pointers, and read/write pointers. Read/write pointers are considered subtypes of read-only pointers and write-only pointers with an invariant target type.

In theory, read-only pointers should be covariant and write only pointers should be contravariant, so that read-write pointers are invariant.

2.2.7 Top and Bottom

In a subtyping lattice, it is usual in the theory to have a type *top* which all types are subtypes of, and, a type *bottom* which is a subtype of all other types.

Felix has both these types. The type *any* is the top type and is defined by the equation:

```
typedef any = any;
```

It is, clearly, a recursive type, since it refers to itself. Felix uses this type for functions which never return, such as *exit*. In principle, *any* should unify with any type, and every type should be a subtype of *any* but Felix currently does not implement this.

Similarly, Felix has a type *void* which is the bottom type defined by

```
typedef void = 0;
```

the sum of no units. There are no values of type *void*. In Felix, a function returning void is a procedure, which returns control but no value.

In principle, void is a subtype of all other types, however Felix does not do this. Instead, void unifies only with void, and otherwise unification fails. Were the theoretical subtyping rule applied, a function with a void parameter would accept an argument of any type. It would do this by simply throwing out the argument. However we do not currently support that.

Functions of void certainly exist, in the category of sets there is indeed a unique function from void to every other type: void is simply the empty set, and a function from the empty set to any other set is modelled by a set of pairs which happens to be empty.

Felix does in fact use some internal tricks where a constant constructor, that is, one with no arguments, is modelled as an injection from void. One can argument that, in fact, all literal values, are in fact precisely function from void to a singleton type containing the value only, as a subtype of the type of the literal. But we don't do that, except as an internal trick.

We may relax these rules later and explicitly support any and void as top and bottom elements with corresponding coercions. Unfortunately the theory of types is based on functional programming model and fails to properly account for effects. Because of this, it is dangerous to provide the full theory, because we would be out of types for procedures and exits, and we would allow dangerous compositions which had side effects functions are not permitted to have.

2.2.8 User Defined Coercions

Felix currently supports a very limited set of coercions which can be defined by the user. The user defines a function named *supertype* which is a coercion from its domain, the subtype, to its codomain, the super type. For example:

```
supertype (x:int) => x.long;
```

says that *int* is a subtype of *long*. This means a function with a long parameter can be called with an int argument. The domain and codomain must be monomorphic nominal types. This requirement may be relaxed in future versions.

The compiler does *not* find composite coercions so technically to retain coherence the user is required to define all composites.

2.3 Discussion

Felix has certain rules which could be represented by coercions but, instead, are represented as identities. In addition, it has some rules which appear to the user as if they were identities but which are, in fact, coercions!

In Felix, a record of all anonymous fields is a tuple, a tuple of all components of the same type is an array, and an array of one element is that element. These are identities of the language, not coercions. Although they appear as a kind of subtyping rule: an element is a special case of an array which is a special case of a tuple which is a special case of a record, in fact, these special cases are only notional.

On the other hand, Felix allows a function to be used when a function value is required, and that is real implicit coercion. Indeed, unlike some other languages there are contexts in which projections and injections can also be used as function values.

This case is a real coercion. Not only does the compiler use quite distinct terms internally, but the generated C++ code is also quite distinct. For example, a function in Felix in general form is represented by a C++ class, whereas a function value is a pointer to a heap allocated object of that class type, completely different kinds of entity.

Nevertheless coherence concerns exist, especially mixing these morphisms with subtyping conversions. It may surprise a user that this is a match:

```
fun f(p: int * long) => ...
.. f (field="Hello", 1,2) ..
```

assuming that we have a coercion from int to long, however the application of f here fails:

```
fun f(p: int * long) => ...
.. f (1,2) ..
```

even though we just dropped the field of string type, which was thrown out by the record coercion anyhow, because now, the argument is an array of two ints, and the same coercion must be applied to all elements, and no coercion exists to convert the array of two ints to a non-array tuple. With the string field in place, distinct coercions were allowed.

Such surpises arise in most languages. The most common is more annoying than surprising: one wants a value of the type of some entity which, in the language, is only a second class citizen. For example modules in Ocaml (until recently!) or type classes in Haskell.

By comparison, in many dynamically typed languages a lot more entities are first class, of necessity. This is because the languages are traditionally interpreters, and the first class values must exist for the interpreter to work at all. This is an often overlooked reason why programmers like dynamic languages: it is not, as many claim, that they dislike static typing as such, but because static type systems are extremely weak by comparison. The extensibility of a large set of Python programs by dynamically loading user extensions to a framework are simply impossible without run time type checks.

Another overlooked features is that consistent and well documented run time type checks actually facilitate dynamic extension. By comparison whilst the same effect can always be obtained in a statically typed language, the programmer of such a system has to reinvent the wheel to obtain dynamics. Python, for example, has a well specified layout for module lookup tables and for type objects which greatly simplify the task of dynamic extension whilst also constraining the kinds of extensions that can be provided to those that are readily supported by the existing framework.

It is indeed quite suprising to find that completely open nature of how dynamics can be implemented in static languages is a severe impediment to reasoning about such systems, not an advantage as often claimed. It is not uncommon, for example, for programmers of strongly typed static languages to resort to parsing strings to implement dynamics.

It is disappointing, for example, that in Felix whilst the type laws

```
3 = 1 + 1 + 1
int * int = int ^ 2
```

hold, the law

```
int + int = 2 * int
```

does not. In fact, the standard representation of sum types and unions does in fact use a pair consisting of a an integer tag and a pointer to the constructor argument, there are also special cases for unions which use more compact and efficient representations, which thereby break the law at the representation level. For example the representation of an list uses a single pointer, not a pair, with the NULL value representing the Empty case and a non-NULL value representing a non-empty tail. Similarly, a standard C pointer which could be NULL, is in fact the representation of the type:

```
union Cptr[T] = nullptr | &T;
```

which allows Felix to use possibly NULL pointers from C directly in the language without any binding glue. Similarly the representation of *int + int* is optimised to a single pointer with the discriminant tag in the low bit of the pointer. Its a nice trick for performance but the C code is not the same as the representation of *2 * int* even though it is isomorphic.

It may seem tempting to introduce many identities and representations as subtyping coercions but the unfortunate fact is that such apparent simplification actually ends up breaking the coherence rule for subtyping and thus is inadmissable. No matter what representations you choose, some coercions will always be value conversions rather than simply type casts.

Dynamic languages, on the other hand, rarely have this problem because all the conversions are run time value conversions: in some sense, dynamic systems are, in fact, more coherent than static ones.

CHAPTER 3

Open Recursion

3.1 Open/Closed Prinicple

One of the most fundamental principles of programming languages is that the language should support some kind of *module* which is simultanously open for extension, yet also closed so it may be used without fear changes will disrupt usage.

This principle was clearly stated by Bertrand Meyer in his seminal work "Object Oriented Software Construction". It was a key motivator for the idea of a class which provided a closed, or *encapsulated* resource with a closed set of well specified methods to manipulate it, whilst at the same time being available for extension via inheritance.

As it turns out this idea fails because identifying a module with a single type is the wrong answer. Never the less the core concept is very important.

3.1.1 The Hard Working Programmer

An unenlightened programmer is asked to provide a term representing an expression which can perform addition on expressions. This is the type:

```
typedef addable = (
 | `Val of int
 | `Add of addable * addable
 )
;
```

and here is the evaluator:

```
fun eval (term: addable) =>
  match term with
  | `Val j => j
  | `Add (t1, t2) => eval t1 + eval t2
;
```

This solve the problem quite nicely. Unfortunately the client asks for an extension to include subtraction. The programmer used *copy and paste polymorphism* to get this type:

```
typedef subable = (
 | `Val of int
 | `Add of subable * subable
 | `Sub of subable * subable
 )
;
```

and here is the new evaluator:

```
fun eval2 (term: subable ) =>
  match term with
  | `Val j => j
  | `Add (t1, t2) => eval2 t1 + eval2 t2
  | `Sub (t1, t2) => eval2 t1 - eval2 t2
;
```

This seems reasonable, we still have the old addable type, but the modifying the original code in your text editors is a pretty lame way to go: what happens if there is a bug in the original routine? Now you have to remember to fix both routines.

Would it surprise you if the client now wants to multiply as well?

3.1.2 The Lazy programmer

The smart programmer writes the same addable routine as the stupid programmar. But the smart programmers is not surprised when the client wants and extension. The smart programmer knows the client will want another one after that too.

So the smart programmer writes this:

```
typedef addable'[T] = (
 | `Val of int
 | `Add of T * T
 )
;

fun eval'[T] (eval: T-> int) (term: addable'[T]) : int =>
  match term with
  | `Val j => j
  | `Add (t1, t2) => eval t1 + eval t2
;

typedef addable = addable'[addable];
fun eval (term:addable) : int => eval' eval term;
```

Now to see why this is a really cool solution:

```
typedef subable'[T] = (
 | addable'[T]
 | `Sub of T * T
 )
;

fun eval2'[T] (eval2: T-> int) (term: subable'[T]) : int =>
```

(continues on next page)

(continued from previous page)

```
  match term with
  | `Sub (t1, t2) => eval2 t1 - eval2 t2
  | (addable'[T] :>> y) => eval' eval2 y
;

typedef subable = subable'[subable];
fun eval2 (term:subable) : int => eval2' eval2 term;
```

What you see here is that there is no code duplication. The new subable' type extends the old addable' type. The new eval2' routine calls the old eval' routine.

This is the extension required by the open/closed principle. On the other hand, by making these parametric entities refer to themselves we fixate them to obtain a recursive closure.

3.2 Open Recursion

The method shown above is called *open recursion*. In its simplest form above it requires polymorphic variant types and higher order function.

With this technique, we make flat, linearly extensible data types by using a type variable parameter in the type where would normally want recursion. Similarly in the flat function, we use a function passed in as a parameter to evaluate the values of the type of the type variable.

The flat forms are extensible, so these type are open.

But when self-applied, the types become closed and directly usable.

So the technique provides a method to define a type with a discrete number of cases, and an an evaluator for it, and to extend the type to one with more cases, without impacting uses of the original type, and critically, without repeating any code.

3.3 Subtyping and Variance

Its important to understand why the technique above works, but an object oriented solution does not.

What you may not have realised is that this works:

```
fun f(x:addable) => eval2 x;
```

What? Yes, addable is a subtype of subable. First, it is a *width subtype*, because addable has less cases. But that is not enough. As well, the arguments of the constructors are subtypes as well. Because they, too, have less cases. This is called *depth subtyping*. It applies recursively, and the subtyping is said to be *covariant*.

Object orientation cannot do this, because method arguments in derived classes must be *contravariant* whereas we want them to be *covariant*. You would like to do this:

```
class Abstract {
  public: virtual Abstract binop (Abstract const &)const=0;
};

class Derived : public virtual Abstract {
  public: Derived binop (Derived const &)const;
};
```

where you see because the argument of the binop method has varied along with the derivation direction, it is said to be covariant. The problem is, the argument of a method must be either *invariant* meaning the same type as in the base, or *contravariant* meaning a base of the base! The return type is covariant, and that is allowed but covariant method arguments are unsound and cannot be allowed.

You can do this:

```
class Derived : public virtual Abstract {
  public: Derived binop (Abstract const &other)const {
    Derived *d = dynamic_cast<Derived*>(&other);
    if (d) { ... }
    else { .. }
  }
};
```

But how do you know you covered all possible derived classes in the downcast? You don't. If someone adds another one, you have to write code for it, and this breaks encapsulation.

The simple fact is OO cannot support methods with covariant arguments which restricts the utility of OO to simple types where the methods have invariant arguments. OO is very good for character device drivers, because the write method accepts a char in both the abstraction and all the derived classes: it is an invariant argument.

3.4 Mixins

It is clear from the presentation that any number of extensions can be added using open recursion in a chain. This means you can form a whole tree of extensions with subtyping relations from the leaves up to the root. Lets make another extension:

```
typedef mulable'[T] = (
 | addable'[T]
 | `Mul of T * T
 )
;

fun eval3'[T] (eval3: T-> int) (term: subable'[T]) : int =>
  match term with
  | `Sub (t1, t2) => eval3 t1 - eval3 t2
  | (addable'[T] :>> y) => eval' eval3 y
;

typedef mulable = mulable'[mulable];
fun eval3 (term:mulable) : int => eval3' eval3 term;
```

Its the same pattern as subable of course. The question is, can we combine this with subable, so we can do addition, subtraction, and multiplication?

```
typedef msable'[T] = (
 | subable'[T]
 | mulable'[T]
 )
;

fun eval4'[T] (eval4: T-> int) (term: msable'[T]) : int =>
  match term with
  | (subable'[T] :>> y) => eval2' eval4 y
  | (mulable'[T] :>> a) => eval3' eval4 z
```

(continues on next page)

(continued from previous page)

```
;

typedef msable = msable'[mslable];
fun eval4 (term:msable) : int  => eval4' eval4 term;
```

The problem here is that both subable' and mulable' contain the case for Add and Val. You will get a warning but in this case it is harmless (because it is the same case).

Here's some test code:

```
val x = `Sub (`Add (`Val 42, `Add (`Val 66, `Val 99)), `Val 23);
val y = `Mul (`Add (`Val 42, `Mul (`Val 66, `Val 99)), `Val 23);
val z = `Sub (`Add (`Val 42, `Mul (`Val 66, `Val 99)), `Val 23);

println$ eval2 x; // subable
println$ eval3 y; // mulable

println$ eval4 x; // subable
println$ eval4 y; // mulable
println$ eval4 z; // msable
```

Note that eval4 works fine on x and y as well as z!

CHAPTER 4

Cofunctions

4.1 Yield

In many programming languages today, there is a special instruction usually called *yield* which allows a procedure to return a value, suspending its execution state in such a way it can resume where it left off.

```
gen producer () : int = {
 var i = 0;
again:>
 yield i;
 ++i;
 goto again;
}
```

This is a simple example of what is called a *yielding generator* in Felix. A *generator* is a function like construction which may return a different value on each invocation, depending on some mutable state. The prototypical generator is *rand* which returns a random number.

The *yield* instruction seen above is similar to *return*, however the producer above does not loose its current location or local variables when it provides a value, instead it suspends so that it may be resumed and continue on where it left off.

You can use the generator above like this:

```
var next = producer;
var current = next();
while current < 10 do
  println$ current;
  current = next();
done
```

The key here is that the variable *next* is used to store the suspended state of the producer as a closure, which is resumed by each call.

4.2 Iterators

This particular kind of construction is also known as an *iterator* in Python. In C++ it is called an *input iterator* although the use is slightly different, and the definition is via a class object.

In general, iterators can be constructed in any object oriented language using an object with mutable state and a *get* method which simultaneously returns a fresh value and also updates the state so the next value can be calculated.

However, OO based iterators are weak compared to yielding generators because the *yield* instruction automatically saves the current location in the generator.

4.3 Cofunctions

I want you to see that an iterator is roughly a function *turned inside out* and therefore deserves the technical name *cofunction*. The natural output of an iterator is a *stream*, but there is more: it a temporal stream.

Functional programming models have a very serious weakness which is that they attempt to be *atemporal*. Advocates laud the fact that an FPL is primarily declarative. Data structure are indeed spatial, but there is more to programming that space, and more to programming than data and functions.

Cofunctions provide a space time transform. You can take a list and produce a stream. A purely spatial, linear data structure has been converted actively into a temporaly linear sequence of codata.

Where data lives at addresses, some of which may be adjacent, and others linked indirectly by pointers, codata has temporal coordinates, marked by a clock.

The world of space and time provide the coordinate system of a program, and the flow of control explains how an algorithm looks at one location at one time, but progresses the construction of new data by simultaneous spatial and temporal sequencing. You move down the list, physically, and you do so in time.

A cofunction, therefore, is an iterator over an abstract data structure, which produces a stream of values. The stream has no natural end and this is a fundamental property which is entirely misunderstood.

Many people think streams are infinite lists but this is in fact completely and utterly wrong. It is hard to comprehend how such fundamentals are so badly misunderstood.

Contrary to popular belief, inductive data types like lists are infinite, whereas streams are finite! It is not hard to understand when one realises that computing is like science not mathematics, it requires a concept of *observation*.

Suppose I give you a list and ask you how long it is:

```
 fun len (x: list[int]) => match x with
  | Empty => 0
  | Cons (head, tail) => 1 + len x
;
```

You would use that algorithm and say that

```
[B] -> [C] -> [D] -> *
```

was length 3. But, you would be wrong! You see, I didn't show you the whole list:

```
[A] -> [B] -> [C] -> [D] -> *
```

I only showed you the tail starting at B. Now you realise your answer should have been *at least 3*. That is the correct answer because the algorithm for the length counts in time by following pointers to the end, but it is a singly linked list so you cannot go backwards!

Let me say that again another way: irrespective of what exists in space, or not, the only thing that matters is what you can observe by an *effective procedure* which is also called an *algorithm*.

So we can observe only a *lower bound* of the size of a list because we only ever see the *tail* of the list. You can never tell, or, *measure* if the first element you see is the head of the list.

So we must emphasise again the relativistic nature of computing: it is all about what you can observe, not about what is. So inductive data types, like lists, all have the same structural property that observations are finite, but are necessarily only lower bounds.

Because the list *could be* longer than any calculated bound we have to assume it is, because no observation can contradict that assumption, in other words, lists are infinite!

Now it is vital to understand that a functional observation of some property, is intrinsically bounded. Suppose you write a function and make a mistake and write an infinite loop. The function never returns, so it is not, in fact able to be used to make an observation: it is not an algorithm.

So I am now going to blow your minds, by claiming that due to duality, streams are finite, and, in fact, when you make an observation on a stream, you are producing an *upper bound*.

Suppose you have a an iterator producing a stream of all the integers. You might think, this is an infinite stream but you could not be more wrong! If it were infinite, an program using the stream to perform a calculation would never terminate!

Let us see how to measure the length of a stream:

```
gen observer () : int = {
  var next = producer;
again:>
  var current = next();
  println$ current; // observation
  yield current;
  goto again;
}
```

Now here is the critical thing: to actually use a stream and calculate some value, we have to *impose* a bound on our observations:

```
gen sum () : int = {
  var it = observer;
  for i in 0 ..< 10 perform
     x += it();
  return x;
}
```

Now if we call sum, how many prints do you see? Did you say 10? So you think, the stream is at least 10 long but, you have it arse about.

Suppose you only saw two:

```
gen producer () : int = {
 var i = 0;
 yield i;
 yield i + 1;
 yield i + 2;
 again: goto again;
}
```

If you see this producer it produces 3 values, then it goes into an infinite loop. So you can write code that reads the first 4 values from it, and that code will never return. It does not make any observation. If you reduce the number to

3,2,1, or 0, you get an observation.

Now think about the producer code itself, and ask, how many values does it produce? Well, if it is called 10 times it produces 3 values. If it is called 9 times it produces 3 values. if it is called twice it produces 2 values. And it if it is never called, it produces NO values.

So the number of values produced by our iterator above is what? You got it! It is *at most 3*.

Streams, by their nature, are finite, not infinite! They are characterised by an upper bound.

All streams are finite, in the sense of a program being a terminating algorithm, and functions, necessarily, most complete and return a value or they're not functions.

CHAPTER 5

Coroutine Basics

Coroutines are not a new concept, however they have been ignored for far too long. They solve many programming problems in a natural way and any decent language today should provide a mix of coroutines and procedural and functional subroutines, as well as explicit continuation passing.

Alas, since no such system exists to my knowledge I have had to create one to experiment with: Felix will be used in this document simply because there isn't anything else!

A *coroutine* is basically a procedure which can be *spawned* to begin a *fibre} of control which can be {em suspended} and {em resumed* under program control at specific points. Coroutines communicate with each other using *synchronous channels* to read and write data from and to other coroutines. Read and write operations are synchronisation points, which are points where a fibre may be suspended or resumed.

Although fibres look like threads, there is a vital distinction: multiple fibres make up a single thread, and within that only one fibre is ever executing. Fibration is a technique used to structure sequential programs, there is no concurrency involved.

In the abstract theoretical sense, the fundamental property possessed by coroutines can be stated like this: within any thread, there exists some total ordering of all events. The ordering may not be determinate, but of any two events which occur, one definitely occurs before the other.

In addition, events associated with one fibre which occur between two synchronisation points, are never interleaved by events from another fibre of the same thread. All interleaving must occur interior to the synchronisation point, that is, after it commences and before it completes. In other words, given a sequence of events from one fibre prior to a synchronisation point, and a sequence of event from another after a synchronisation point, all the events of each sequence occur before or after all the events of the other.

Premptive threads, on the other hand, allow arbitrary interleaving of each threads sequence of events, up to and after any shared synchronisation. Mutual exclusion locks provide serialisation, which is the default behaviour of coroutines.

Therefore, fibre based programming can proceed where general code may assume exclusive access to memory and other resources over all local time periods not bisected by a volutary synchronisation event; threads, on the other hand, can only assume exclusive access in the scope of a held mutex.

The most significant picture of the advantages of coroutines is thus: in a subroutine based language there is a single machine stack. By machine stack, I mean that there is an important *implicit* coupling of control flow and local variables. In the abstract, a subroutine call passes a continuation of the caller to the callee which is saved along with local variables the callee allocates, so that the local variables can be discarded when the final result is calculated, and then passed to the continuation. This technique may be called *structured programming*. With coroutines, the picture is simple: each fibre of control has its own stack. Communication via channels exchanges data and control between stacks.

Coroutines therefore leverage control and data coupling in a much more powerful and flexible manner than mere functions, reducing the need for state to be preserved on the heap, thereby making it easier to construct and reason about programs.

For complex applications, the heap is always required.

5.1 A Simple Example

The best way to understand coroutines and fibration is to have a look at a simple example.

5.1.1 The Producer

First, we make a coroutine procedure which writes the integers from 0 up to but excluding 10 down a channel.

```
proc producer (out: %>int) () {
  for i in 0..<10
    perform write (out, i);
}
```

Notice that as well as passing the output channel argument *out* there is an extra unit argument *()*. This procedure terminates after it has written 10 integers. The type of variable *out* is denoted *%>int* which is actually short hand for *oschannel[int]* which is an output channel on which values of type verb%int% may be written.

5.1.2 The Transducer

Next, we make a device which repeatedly reads an integer, squares it, and writes the result. It is an infinite loop, this coroutine never terminates of its own volition. This is typical of coroutines.

Here, the type of variable *inp* is denoted *int* which is actually short hand for verb%ischannel[int]%, which is an input channel from which values of type verb%int% may be read.

5.1.3 The Consumer

Now we need a coroutine to print the results:

```
proc consumer (inp: %<int) () {
  while true do
    var y = read inp;
    println y;
  done
}
```

Each of these components is a coroutine because it is a procedure which may perform, directly or indirectly, I/O on one or more synchronous channels.

CHAPTER 6

What's Wrong With C++

This article is intended to clearly specify *fundamental* problems in C++. There are many problems, in any language, but the concern here is with serious core issues.

6.1 Syntax

C and C++ have really bad syntax. C started out weak, got worse, C++ inherited the problems, and then made them worse again.

To understand how bad it is, we look at a brief history. Originally K&R C was designed so that top level constructions, at least after pre-processing, could be rapidly parsed in a single pass and in isolation. Type checking was, to the extend it allowed it, also possible in a single pass over the whole translation unit.

The fundamental reason this was possible was that the set of types were fixed and represented syntactically by keywords such as *int* or *double*. User defined types in the form of *struct* definitions were allowed, however the types had to be refered to using the *struct* keyword and a tag. Because of this, an incomplete type could be used in a type defintion, and correctly parsed.

In particular, consider these fragments:

```
(X)(y)          /* function application */
(struct X)(y)   /* cast */
```

There is no need to see the surrounding context. In other words, the language was context free.

Unfortunately the ANSI committee came along and destroyed this property by introducing *typedef*. Consequently the first case above is ambiguous, and can only be correctly parsed if the whole of the previous code is seen, in case X is type introduced by a typedef.

The loss of context freedom was a serious mistake for C, but for modern C++ it is an unmitigated disaster. C++ programmers today use a lot of templates and many libraries are *header file only* and every compilation of every translation unit has to parse all the header files sequentially, every time.

However the situation was even worse than that! With templates, it is not possible to tell if X is a type or not, just be parsing all the header files. The following example shows why:

```
template<class T>
void f() { int x = (T::X)(g); }
```

If T is instantiated by a class which contains X as a typedef, then the RHS of the assignment is a cast, if X is a function, then it is a function application. Without any further information, the template cannot be parsed at all.

The ISO C++ committee introduced a new keyword to fix this, *typename*:

```
template<class T>
void f() { int x = (typename T::X)(g); }
// its a cast!
```

If you don't use *typename* then its a function application.

The real situation is worse again because you can also pass templates as arguments!

There are other cases in C++ where parsing is ambiguous. The most famous is that it is impossible to tell the difference in C++ between an declaration and an initialisation:

```
T f(X);
```

This could be declaring the function f, returning type T and accepting type X, or, it could be an initialisation of the variable f, of type T, to the value X.

The ISO C++ committee introduced disambiguating rules, but again, the choice depends on context. Luckily C++ has other, non-ambiguous ways to achieve the required result, but still, this yet another serious design fault which makes parsing difficult.

Of course the famous problem with >> in templates is well known, which stems from another serious mistake in ARM C++ using < and > as brackets, as well as comparison operators.

The need for context to parse C and C++ is not merely a problem for the compiler, it is a problem for the reader as well. And it is an even worse problem for the programmer when trying to refactor code.

In addition, the C++ committee had a desire when adding new features to avoid introducing new keywords, so many constructions are introduced by syntactic forms which are hard to decipher and in many cases the design is actually flawed because it fails to allow syntact distinctions to be made which have semantic impact.

The worst example of this is the template specialisation syntax. Contrary to popular belief, function templates cannot be specialised, only overloaded, however class member function can be specialised, but not overloaded! For example:

```
template<class T>
void f(T);

template<class U>
void f<vector<U>>(vector<U>);
```

This looks like it is declaring a specialisation of the function template f, but it isn't. It is actually introducing a completely new function which happens to be defined by a specialistion of the original f.

The new function overloads with the original one, and since it is more specialised will be selected by overload resolution. However a real specialisation has no impact on lookup at all, only on instantiation. This is the case for classes:

```
template<class T>
class X { void f(T); };

template<class U>
class X<vector<U>>;
```

This introduces a specialisation, and by default the member f is also specialised .. there is no overloading here. Even if a replacement is defined for the f, this has no impact on overloading.

The problem is that the committee didn't understand the difference between these two cases and provided a syntax in which it is impossible to distinguish them. Hence, function template specialisations are overloads not specialisations, because some choice had to be made given the faulty syntax.

6.2 No type checking in templates

This is a very serious design fault. Templates should introduce polymorphic types and functions, but they do not, because they cannot be type checked. Therefore, templates are just syntax macros, and the result is a disaster.

Recently there was an attempt to solve this problem the way Haskell does with type classes: the feature known as *concepts*. Unfortunately the design was rejected and replaced by a much weaker version known as *concepts-lite*.

If templates could be type checked, this would mean instantiations would not require type checking: all instantiations would be guaranteed to be correct. That also means the instantiation would be entirely independent of context, and in particular two instantiations with the same template arguments in different places would necessarily be the same type.

6.3 Lvalues and references

In C, a variable name has two distinct meanings depending on context. If it is used on the LHS of an assignment, or as the argument of the addressof operator, then it represents a storage location. The assignment puts a value into that location, and the addressof operator finds a pointer to that location.

In C, the context where a variable name is treated as refering to a storage location is called an l-context, other contexts are called r-contexts. The *l* and *r* refer to which side of an simple assignment it might be.

A variable name is an *lvalue* which means it refers to a storage location in an lcontext, but the value stored at that location in an rcontext.

Similar rules apply to, for example, pointer dereferences. Certain syntactically recognisable expressions in C are said to be lvalues, others are rvalues. Lvalues can be used in both lcontexts and rcontexts, in an rcontext the lvalue degrades to an rvalue. An rvalue cannot be used in an lcontext.

In summary in C, the semantics of certain expressions depends on a context which is locally syntactically determinate.

The ambiguity is bad, and causes a lot of confusion, but the disambiguation is possible by simply examining the expression in isolation and following the rules layed down in the C Standard.

Unfortunately C++ introduced a notion of references and reference types and all hell broke loose! Because a reference is universally an lvalue, but is also a type, it is not longer possible to determine the meaning or correctness of an expression from local syntactic examination. For example

```
f(x) = g(y);
```

would never be allowed in C (after pre-processing), because the LHS does not have the syntactic form of an lvalue. In C++, you need to examine the function *f* to see if it returns a non-const reference to determine if the above code is correct: and that also means determining the type of *x* because the function *f* could be overloaded. If we replace *x* with an expression:

```
f(h(x)) = g(y);
```

we now have to type the expression *h(x)* which recursively involves overload resolution for *h*.

This may seem complicated but the situation is much worse. For a start, the ARM was very confused about overloading function with reference type arguments:

```
void f(int);
void f(int&);
void f(int const&);
int x=1;
f(x); // which f?
```

What is the type of x? It is an lvalue, but it has type *int*, but lvalues are replaced by references, so the type should actually be *int&*. But consider now:

```
int &x = y;
```

and clearly *x* now refers to the same store as *y*, so the type is *int&* but the definition has quite distinct semantics from an int definition: an int definition creates a new store to put an int in, the int& definition causes x to refer to existing store. The types in an expression are the same however, and that means *f(x)* must call the same overload in both cases.

The ARM got this wrong. The ISO committee debated this issue at length and resolved it, but they chose the wrong solution. The correct solution was to throw out the whole idea of reference types: instead a perverted form of reference types was introduced in which they were just renamed as lvalue types.

It is legitimate to allow function arguments to be passed by reference, and this is certainly part of the type information of the function, but references have no place as types in themselves because they are not proper type constructors.

A polymorphic type constructor must be combinatorial for parametric polymorphism to work. For example for any type T, the type **T* makes sense, it is the type of a pointer to T. The pointer type constructor is properly parametric because it can be applied to any type, including another pointer type.

References are not combinatorial, it is nonsense to take a reference to a reference. No one would do this in practice in monomorphic code so it might be excused but for templates.

If a reference is a type, then a template type parameter could be set to one, and then all hell breaks loose because it changes, utterly, the semantic of the template.

```
template<class T>
void f() { T x = T(); }
```

In this template, all is fine provided T has a default constructor. But what can we say if T is a reference:

```
f<int&>();
```

Since references don't have default constructors, we get error. But consider this one:

```
template<class T>
void f(T x, T y) { x = y; }
```

For a value type T, f does nothing, except perhaps exhibit the behaviour of an overloaded assignment operator. But if T is a reference this code has an effect, it assigns the value of y to the location to which x refers.

In theory, there is no need for references at all. Pointers are perfectly good enough and pointer calculations are purely functional. They are first class types and the pointer constructor is parametric.

Introducing references was a serious design fault. It has lead to introduction of even worse design faults including *decltype* to handle the problems.

6.4 Const

Const is another thing inherited from C and messed up in C++ very badly.

In C, the type syntax makes it seem like you can have a const type. This is not the case. The syntax is misleading, there are no const types in C. In C there are pointers to const, and that is all.

It may seem otherwise examining this code:

```
int const x = 1;
int const *px = &x;
```

In C, x has the type int, not const int. Rather, C introduces a new form of lvalue, a const lvalue. If you take the address of a const lvalue you get a pointer to const. But as an rvalue, x has type int.

Of course it works the other way too:

```
*px = 1; // error, const lvalue!
```

Because px is a pointer to const, a derefernce produces a const lvalue which can be addressed but not assigned to.

Const lvalues in C cause a problem though because now, the kind of lvalue is context dependent. In C++ this is true as well.

There is another problem with const: that which is pointed at by a const pointer need not be immutable because of aliasing. C introduced the *restrict* keyword to enhance optimisation opportunities since overlapping array arguments were never allowed in Fortran, and Fortran remained the premier numerical programming language for decades (and still is). Restrict disallows aliasing and so a restricted const pointer, whilst still does not pointer to immutable store, can be assumed to point at store which doesn't change during the lifetime of the function.

In C++ all hope is lost when we consider templates. Because both const and reference are effectively types, the semantics of a template are utterly indeterminate until it is instantiated. Weird effects can occur, and be type correct, when instantiating a template with a const and/or reference type.

6.5 Offsets

In C, address arithmetic can be done with casts, and by use of the *offsetof* macro. The result isn't type safe, but all useful calculations can be done.

In C++, a type safe version of the *offsetof* macro was introduced, namely a type *pointer to member*. Unfortunately, the ISO committee again made a mess of things by insisting on pointers to members working with virtual functions and classes. As a result, the full calculus is incomplete.

In principle, if you have a struct nested in another struct, you should be able to calculate the offset of a member of the inner struct by adding the offset of the member of the outer struct with respect to the outer struct, to the offset of the inner member, with respect to the inner struct, obtaining the offset of the inner member with respect to the outer struct. Unfortunately, there is no syntax to do this addition, in part because the calulation cannot be done in the presence of bases. The problem is, you need to know the layout of the classes to do the operation: given a pointer to the outer struct, you can add the outer pointer to member, then the inner one, to obtain a pointer to the inner member, but the operation isn't associative, so you cannot add the pointers to members together first.

Of course it can be done with a closure, that is, with lazy evaluation, but this requires a pointer to member to be an arbitrarily complex data type with indeterminate storage requirements.

Pointers to member of ordinary non-OO style structs are vitally important because they are first class projections, and they should be composable.

6.6 Object Orientation

Adding OO to C gave us the slogan for early C++ as *C with classes*. It seemed like a good idea at the time, but object orientation is fraught with peril and it has been established for a long time that it does not provide a general mechanism for providing abstract data types. This is due to what is called the *covariance problem*, which requires the argument of a method to be contra-variant. Unfortunately to implement a binary operator we require covariance, and so OO cannot represent even binary operators. It is still useful when a method has no arguments, that is, for properties, or, when the arguments are invariant, for example for character device drivers.

No one takes OO seriously in modern C++: most programming uses templates which is roughly functional programming.

Classes introduced a whole host of bugs. Arrays of value of a derive type can be implicitly converted to arrays of the base type via the degradation of array types to pointers to the first member, which then results in increments and random accesses to the array using the size of the base type as an offset instead of the derived type.

This is not the only unsound feature introduces. Another well known example is the ability of a class constructor to export a non-const pointer to itself, even if the value is specified as const. This is because in the body of a constructor the this pointer is non-const, which is required for storing values in the object.

Another related unsoundness in the type system is that in a constructor body, the whole of the currently in scope class is visible, including bases. Unfortunately, with muliple inheritance there is no assurance that all the bases have been constructed yet.

Worse, the constructor body can invoke a virtual function which would, after the whole object is built, dispatch to a method of the complete type. In single inheritance, the virtual table can be built for the base first, so a dispatch in a derived class constructor will work correctly for that point in time provided it doesn't dispatch to a method in the current class and depend on members which have no yet been set.

Unfortunately, with multiple inheritance and virtual bases, it is not possible to assure the correct virtual table is installed, because a virtual base can dispatch to an as yet unconstructed derived class which is not even visible to the current class. This is known as sibling dispatch:

```
struct V { virtual void f()=0; };
struct D : virtual V { void f(); };
struct E : virtual V { E() { f(); } };
struct X : E, D {};
```

The problem here is that E is constructed first and it calls f() which dispatchs via its base V to D's override of f. The problem is D hasn't been constructed yet and so the virtual table in V points f off into thin air, typically to a run time diagnostic followed by a program abort which usually says that a pure virtual has been called. Of course once X is completed the dispatch would work just fine.

6.7 How Felix fixes the problems

Felix fixes all the above problems.

6.7.1 Parsing

First, the language is designed so that, with one exception, all top level constructs, including any file, can be parsed independently of context.

All files can be parsed independently of all others, however the rule for constructions has an exception: if a scope, including a file scope, opens a syntax module (called a DSSL in Felix), then the grammar parsed changes from the current grammar at the point of the open directive.

Opening new grammar is unusual in most code, and when used in a file it is usually done at the top of the file so it applies to the whole file. The grammar extensions are scoped so they cannot be exported from the file, and, if used inside some scope such as a class, they cannot be exported from the class.

6.7.2 Syntactic complexity

Felix fixes the arcane complexity of C++ by the simple expedient of throwing out the whole grammar and starting afresh.

Indeed, the grammar in Felix is part of the library, in user space, so considerable complexity can and is introduced, but there is an opportunity to design the syntax in a sane manner.

One specific feature that should be noted is that in Felix there are no keywords. Felix uses a GLR+ extensible parser and recognises identifiers as keywords only in a context sensitive manner. There are a lot of such context sensitive keywors but they can be designed into the grammar with impunity because most of the time they are effective only in the context for which they're introduced, and where there is an conflict, Felix provides a special lexical form to force recognition of an identifier. For example:

```
var var = 1;
n"var" = var;
println$ var;
```

Here, the first *var* is in a context where it is treated as a keyword whilst the second is not, so it is treated as an identifier. The third use would be treated as a keyword so we use the special lexeme which forces interpretation of an identifier. The fourth and fifth contexts treat *var* and an identifier.

6.7.3 Lvalues and references

Felix has no concept of a reference, it just uses pointers. There is only one allowed kind of lvalue, namely a variable name, and only one operation on it, namely to take its address.

In principle even this is not the case: in Felix a variable definition of a name *x* is actually a pointer which has to be dereferenced to get a value, but the language does this automatically and the use of the addressing operator merely inhibits this behaviour.

Felix uses the following to store values:

```
var x = 1;
var px = &x;
px <- 2;
storeat (px, 2);
&x <- 1;
x = 1;
```

The <- symbol is an infix operator which invokes the *storeat* procedure, which is the *only* way to store a value supported by the language. The last line is an assignment, but that is actually syntactic sugar for a call to the storeat procedure to the address of the LHS, and it cannot be used for components of a product type, only simple variables.

To store values into the component of a product type, Felix uses first class projections which apply to pointers. For example:

```
struct X { x:int; };
var a = X (1);
&a.x <- a.x + 1;
```

In the last line, the RHS symbol *x* is actually a value projection of type:

```
X -> int
```

Projections are first class functions. There is no "member access syntax" either. Instead, operator dot (.) is unniversally just reverse application. Therefore you can write this as well:

```
x &a <- x a + 1;
```

Here you can see *x* is overloaded so that as well as the type of the value projection, there is also an overload for pointers:

```
&X -> &int
```

This overload calculates the address of the member and so now you can store in the member.

6.7.4 Const

Felix, like C++, has a const pointer type, however unlike C++ there are no references (they're not needed, see above). And there is no confusion about const types, there is no such thing in either language but in C++ the syntax suggests there is. Instead in Felix we have read only and write only pointers:

```
x &>a <- *(x &<a) + 1;
```

The LHS address of operator returns a write only pointer, whilst the RHS operator a read only pointer. Read/write pointers use the plain & operator and the type is a subtype of both read only and write only pointers.

6.7.5 Object Orientation

Felix fixes this problem by annihilation. There is no OO in Felix.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search