Felix Tutorial

Release 2016.07.12-rc1

Mar 23, 2021

Contents

An introduction to the Felix programming language.

Contents:

CHAPTER 1

Hello

This is the Felix tutorial

1.1 Basics: Hello

1.1.1 Running a Program

Let's start with a simple script:

```
println$ "Hello World!";
```

To run it you just say:

```
flx hello.flx
```

It's pretty simple. Felix runs programs like Python does, you run the source code directly. Behind the scenes, Felix translates the program into C++, compiles the program, and runs it. All the generated files are cached in the .felix/cache subdirectory of your $HOME directory on Unix like systems, and $USERPROFILE on Windows.

This means the script can be run in a read-only directory.

CHAPTER 2

Felix 101: The Basics

The first module gives a brief overview of some basics.

2.1 Comments

Felix has two kinds of comments.

2.1.1 C++ comments

C++ style comments begin with // and end at the end of the line.

```
println$ "Hi"; // say Hi!
```

2.1.2 Nested C comments

C style comments begin with /* and end with a matching */.

```
/* This is an introductory program,
   which says Hello!
*/
println$ "Hello"; /* Say it! */
```

Unlike C comments, in Felix C style comments nest. Take care of and leadin or leadout marks hidden in string literals!

```
// a comment in C++ style
/* a C style comment
   /* nested comment */
   still commented
*/
```

Nested comments are often used to temporarily remove code:

```
/*
/* This is an introductory program,
   which says Hello!
*/
println$ "Hello"; /* Say it! */
*/
println$ "Bye";
```

2.2 Identifiers

Felix has a rich lexicology for identifiers.

2.2.1 C like identifiers

C like identifiers start with a letter, and are optionally followed by a sequence of letters, underscores, single quotes, hyphens, or ASCII digits.

The first character may also be an underscore but that is reserved for the system.

A letter may be any Unicode code point accepted by ISO C++ Standard as a letter, it must be encoded as UTF-8.

```
Hello
X123
a'
julie-marx
x
```

Note again please, only ASCII digits are permitted.

2.2.2 Tex Identifiers

A leading slosh followed by a nonempty sequence of ASCII letters is recognised as an identifer. With suitable output machinery, the corresponding LaTeX or AmSTeX symbol should display if there is one.

```
\alpha
```

displays as

```
\(\alpha\)
```

No space is required after a TeX identifier. Therefore

```
\alpha2
```

encodes the symbol alpha followed by the integer 2.

2.3 Variables

Felix provides three simple forms to define and initialise variables. The *var* binder is used to define a variable, it binds a name to a storage location.

2.3.1 Variables with type and initialiser

A variable can be defined with a type annotation and an initialiser.

```
var b : bool = true;
var i : int = 1;
var s : string = "Hello";
var d : double = 4.2;
var f : float = 4.2f;
```

The specified type must agree with the type of the initialiser.

2.3.2 Variables without type annotation

A variable can also be defined without a type annotation provided it has an initialiser.

```
var b = true;
var i = 1;
var s = "Hello";
var d = 4.2;
var f = 4.2f;
```

In these cases the type of the variable is the type of the initialiser.

2.3.3 Variables without initialiser

Variables can be defined without an initialiser.

```
var b : bool;
var i : int;
var s : string;
var d : double;
var f : float;
```

In this case the variable will be initialised by the underlying C++ default initialiser. It is an error to specify a variable this way if the underlying C++ type does not have a default initialiser.

If the underlying C++ default initialiser is trivial, so that the store is not modified, then the Felix variable is uninitialised.

2.3.4 Simple Assignment

An assignment can be used to assign the first value stored in the location of a variable, to modify the value which an explicit initialiser previously provided, or to modify the value which the underlying C++ default initialiser provided.

```
var b : bool;
var i = 1;
b = true;
i = 2;
```

Assignments are executed when control flows through the assignment.

2.3.5 Variable Hoisting

Var binders are equivalent to declaration of an uninitialised variable and an assignment. The location of the declaration within the current scope is not relevant. The position of an initialising assignment is. For example:

```
a = 1;
var b = a;
var a : int;
```

is equivalent to

```
var a = 1;
var b = a;
```

2.4 Logic Type Bool

Felix provides a type for simple logic which tradiationally is called *bool* after mathematician George Bool.

2.4.1 Type

In Felix, *bool* is a special case of a more general mechanism we will meet later. It is an *alias* for the type *2*, which is the type that handles two alternatives:

```
typedef bool = 2;
```

The *typedef* binder binds a name to an existing type, that is, it creates an alias.

2.4.2 Constants

There are two predefined constants of type bool, *true* and *false*.

2.4.3 Operations

The prefix operator *not* provides negation, infix *and* conjunction, and infix *or* disjunction, with weak precedences, of decreasing strength.

```
not a and b or c
```

is parsed as

```
((not a) and b) or c
```

These operators are all weaker than the comparisons they often take as arguments, so that

```
a < b and b < c
```

is parsed as

```
(a < b) and (b < c)
```

2.4.4 Summary: Logical operations

Operator	Type	Syntax	Semantics
or	bool * bool -> bool	Infix	Disjunction
and	bool * bool -> bool	Infix	Conjunction
not	bool -> bool	Prefix	Negation

2.5 Integers

In Felix we have at type named *int* which has values which are small integers close to zero.

2.5.1 Type

This type is defined by lifting it from C++ in the library by:

```
type int = "int";
```

The exact range of integers represented is therefore determined by the underlying C++ implementation.

2.5.2 Literals

Non-negative values of type *int* can be written as a sequence of the decimal digits like this:

```
0
1
23
42
```

You can also put an underscore or single quote between any two digits:

```
123_456
123'456
```

2.5.3 Negation

There are no negative integer literals. However you can find the negative of an integer using a prefix negation operator, the dash character -, or the function *neg*:

```
-1
neg 1
```

2.5.4 Infix Operators

Integers support simple formulas:

```
(12 + 4 - 7) * 3 / 6 % 2
```

Here, + is addition, infix - is subtraction, * is multiplication, / is division, and % is the remainder after a division.

2.5.5 Sign of quotient and remainder

Division and remainder require a little explanation when negative numbers are involved. The quotient of a division in C, and thus Felix, always rounds towards zero, so:

```
-3/2 == -1
```

The quotient is non-negative if the two operands are both negative or both non-negative, otherwise it is non-positive.

The remainder, then, must satifsy the formula:

```
dividend == quotient * divisor + remainder
```

so that we have

```
remainder == dividend - quotient * divisor
```

Therefore the remainder is non-negative if, and only if, the dividend is non-negative, otherwise it is non-positive.

2.5.6 Comparisons

We provide the usual comparisons from C: == is equality, *!*= is inequality, < is less than, > is greater than, <= is less than or equal to, and >= is greater than or equal to.

The result of a comparison is value of *bool* type.

2.5.7 Constant Folding

If you write a formula involving only literals of type *int*, the Felix compiler will perform the calculation according to mathematical rules, using a very much bigger integer representation. At the end, the result will be converted back to the smaller *int* representation.

If the result of the calculations exceeds the size of the compiler internal representation, or, the final result is too large for an *int*, the result is indeterminate.

2.5.8 Division by Zero

If a division or remainder operation has a divisor of zero, the compiler may abort the compilation, or it may defer the problem until run time. If the problem is deferred and the code is executed, an exception will be thrown and the program aborted. However the code may not be executed.

2.5.9 Out of bounds values

If the result of a calculation performed at run time is out of bounds, the result is indeterminate.

2.5.10 Summary: Integer Comparisons

Operator	Type	Syntax	Semantics
==	int * int -> bool	Infix	Equality
!=	int * int -> bool	Infix	Not Equal
<=	int * int -> bool	Infix	Less or Equal
<	int * int -> bool	Infix	Less
>=	int * int -> bool	Infix	Greater or Equal
>	int * int -> bool	Infix	Greater

2.5.11 Summary: Integer Operations

Operator	Type	Syntax	Semantics
+	int * int -> int	Infix	Addition
-	int * int -> int	Infix	Subtraction
*	int * int -> int	Infix	Multiplication
/	int * int -> int	Infix	Division
%	int * int -> int	Infix	Remainder
-	int -> int	Prefix	Negation
neg	int -> int	Prefix	Negation
abs	int -> int	Prefix	Absolute Value

2.5.12 More Integers

Felix has many more integer types. See the reference manual:

https://felix.readthedocs.io/en/latest/integers.html

for details.

2.6 Slices

A slice is a range of integers.

2.6.1 Type

The type is defined in the library as

```
slice[int]
```

2.6.2 Inclusive Slice

From first to last, inclusive:

```
first..last
```

2.6.3 Exclusive Slice

From first to last, exclusing last:

```
first..<last
```

2.6.4 Counted Slice

From first for a certain number of values:

```
first.+count
```

2.6.5 Infinite Slices

Felix provides three apparently infinite slices, which are actually bounded by the limits of the integer type:

```
first..      // first..maxval[int]
..last       // minval[int]..last
..<last      // minval[int]..<last
```

There is also a slice over the whole range of a type:

```
Slice_all[int]
..[int]
```

2.6.6 Empty Slice

There is an empty slice too:

```
Slice_none[int]
```

2.6.7 More Slices

More detail on slices in the reference manual:

https://felix.readthedocs.io/en/latest/slices.html

2.7 Floating Point Numbers

Floating point literals are local approximations to reals. Local means close to zero. Floats are dense near zero and lose precision far from it.

2.7.1 Type

We lift the main floating point type, *double* from C.

```
type double = "double";
```

It is a double precision floating point representation usually conformat to IEEE specifications.

2.7.2 Literals

Floating literals have two parts, a decimal number known as the mantissa, and a power of 10 known as the exponent.

```
12.34
12.34E-4
```

The mantissa must contain a decimal point with a digit on either side. The exponent is optional, and consists of the letter *E* or *e* followed by a small decimal integer literal, or a + sign or minus sign, and a small decimal integer literal.

If the exponent is present, the mantissa is multiplied by 10 raised to the power of the signed integer part exponent.

2.7.3 Operations

Floating numbers support negation with prefix -, addition with infix +, subtraction with infix -, multiplication with infix * and division with infix / as well as many other operations given by functions in the library.

It is also possible to perform comparisons, equality ==, inequality *!*=, less than <, less than or equal to <=, greater than > and greater than or equal to >=. However these comparisons reflect floating point arithmentic which only approximates real arithmetic. Do not be suprised if the formula

```
1.0 / 3.0 * 3.0 == 1.0
```

is false. To remedy this properly requires a deep knowledge of numerical analysis. Felix helps by providing the function *abs* which can be used like this:

```
abs ( 1.0 / 3.0 * 3.0 - 1.0) < 1.0e-3
```

to check the result is with about 3 decimal places of 1.0.

2.7.4 Summary: Double Comparisons

Operator	Type	Syntax	Semantics
==	double * double -> bool	Infix	Equality
!=	double * double -> bool	Infix	Not Equal
<=	double * double -> bool	Infix	Less or Equal
<	double * double -> bool	Infix	Less
>=	double * double -> bool	Infix	Greater or Equal
>	double * double -> bool	Infix	Greater

2.7.5 Summary: Double Operations

Operator	Type	Syntax	Semantics
+	double * double -> double	Infix	Addition
-	double * double -> double	Infix	Subtraction
*	double * double -> double	Infix	Multiplication
/	double * double -> double	Infix	Division
-	double -> double	Prefix	Negation
neg	double -> double	Prefix	Negation
abs	double -> double	Prefix	Absolute Value

2.7.6 More Floats

Felix has more floating types. See the reference manual:

https://felix.readthedocs.io/en/latest/floats.html

for details.

2.8 Strings

A string is basically a sequence of characters with a definite length. The type is the traditional C++ string type, and supports Unicode only by UTF-8 encoding.

Strings are 8 bit clean, meaning all 256 characters, including nul, may occur in them.

However string literals may not directly contain a nul. String literals are actually C nul terminated char strings which are lifted to C++ automatically.

2.8.1 Type

The string type in Felix is the based on C++:

```
type string = "::std::basic_string<char>"
  requires Cxx_headers::string
;
```

2.8.2 Literals

There are two simple forms of string literals:

```
var x = "I am a string";
var y = 'I am a string too';
```

using double quote delimiters and single quote delimiters. These literals cannot exceed a single line. However you can concatenate them by juxtaposition:

```
var verse =
  "I am the first line,\n"
  'and I am the second.'
;
```

Notice the special encoding \n which inserts an end of line character into the string rather than a \ followed by an *n*. This is called an escape.

You can prevent escapes being translated with raw strings like this:

```
r"This \n is retained as two chars"
```

Only double quoted strings can be raw.

Felix also has triple quoted strings, which span line boundaries, and include end of lines in the literal:

which contains two end of line characters in the string, whilst this one:

```
'''
Here is another
long string
'''
```

has three end of lines (one after the first triple quote).

2.8.3 Length

Use the *len* function:

```
var x = "hello";
var y = x.len.int;
```

2.8.4 Concatenation

Strings can be concatenated with the infix + operator or just written next to each other, juxtaposition has a higher precedence than infix +.

```
var x = "middle";
var y = "Start" x + "end";
```

In this case, the first concatenation of x is done first, then the second one which appends "end". The result is independent of the ordering because concatenation is associative, the run time performance, however, is not, because concatenation requires copying.

2.8.5 Substring Extraction

A substring of a string can be extracted using a slice with the notation shown:

```
var x = "Hello World";
var y = x.[3..7]; // 'lo Wo'
```

2.8.6 Indexing

Select the n'th character:

```
var x = "Hello World";
var y = x.[1]; // e
```

2.8.7 Comparisons

Strings are totally ordered using standard lexicogrphical ordering and support the usual comparison operators.

2.8.8 Summary: String Comparisons

Operator	Type	Syntax	Semantics
==	string * string -> bool	Infix	Equality
!=	string * string -> bool	Infix	Not Equal
<=	string * string -> bool	Infix	Less or Equal
<	string * string -> bool	Infix	Less
>=	string * string -> bool	Infix	Greater or Equal
>	string * string -> bool	Infix	Greater

2.8.9 Summary: Double Operations

Operator	Type	Syntax	Semantics
len	string -> size	Prefix	Length
+	string* string -> string	Infix	Concatenation
.[_]	string * slice -> string	Postfix	Substring
.[_]	string * int -> char	Postfix	Indexing

2.9 Characters

The atoms of a string are ASCII characters.

2.9.1 Type

Lifted from C++:

```
type char = "char";
```

2.9.2 Extraction from String

The first character of a string can be extracted:

```
var ch : char = char "Hello";
```

This sets ch to the letter H. If the string is empty, the char becomes NUL.

2.9.3 Ordinal Value

The code point of the character, from 0 to 255, as an *int*:

```
var ch = char "Hello"; // H
var j = ch.ord; // 72
```

2.9.4 Construction from Ordinal Value

Finds the n'th character in the ASCII character set.

```
var ch = char 72; // H
```

2.10 Simple Control Flow

2.10.1 Sequential Flow

Normally, control flows from one statement to the next.

```
first;
second;
third;
```

2.10.2 No operation

Felix provides several statements which do nothing. A lone semi-colon *;* is a statement which does nothing.

```
first;
; // does nothing
second;
```

2.10.3 Labels and Gotos

The default control flow can be modified by labelling a position in the code, and using a goto or conditional goto targetting the label.

```
  var x = 1;
next:>
  println$ x;
  if x > 10 goto finished;
  x = x + 1;
  goto next;
finished:>
  println$ "Done";
```

An identifier followed by *:>* is used to label a position in the program.

The unconditional *goto* transfers control to the statement following the label.

The conditional goto transfers control to the statement following the label if the condition is met, that is, if it is true.

2.10.4 Chained Conditionals

A chained conditional is syntactic sugar for a sequence of conditional gotos. It looks like this:

```
if c1 do
  stmt1a;
  stmt1b;
elif c2 do
  stmt2a;
  stmt2b;
else
  stmt3a;
  stmt3b;
done
```

At least one statement in each group is required, the no-operation *;* can be used if there is nothing to do. A semicolon is not required after the *done*.

2.11 Loops

Loops statements are compound statements that make control go around in circles for a while, then exit at the end of the loop.

2.11.1 While loop

The simplest loop, repeatedly executes its body whilst its condition is true. If the condition is initially false, the body is not executed. On exit, the statement following the loop is executed.

```
var x = 10;
while x > 0 do
  println$ x;
  x = x - 1;
done
println$ "Done";
```

A semicolon is not required after the *done*. Make sure when writing while loops that the condition eventually becomes false, unless, of course, you intend an infinite loop.

2.11.2 For loop

For loops feature a control variable which is usually modified each iteration, until a terminal condition is met. The simplest for loop uses a slice:

```
for i in 0..<3 do
  println$ i;
done
```

Here, we print the variable *i*, which is initially 0, and takes on the values 1,2 as well before the loop terminates. The slice used indicates it is exclusive of the last value. An inclusive slice is illustrated here:

```
for i in 0..3 do
  println$ i;
done
```

and the loop iterations include the value 3. The values of a the slice start and slice end delimiters can be arbitrary expressions of type *int*. Slices can be empty if the end is lower than the start, in this case the loop body is not executed.

The control variable, *i* above, is automatically defined and goes out of scope at the end of the loop. It should not be modified during the iteration.

2.12 Arrays

In Felix, arrays are first class values.

2.12.1 Type

An array is given a type consisting of the base type and length.

```
int^4
```

is a the type of an array of 4 ints. Note, the *4* there is not an integer but a unitsum type.

2.12.2 Value

An array is given by a list of comma separated expressions:

```
var a :int^4 = 1,2,3,4;
```

2.12.3 Operations

Projection

The most fundamental operation is the application of a projection to extract the n'th component of an array. Components are numbered from 0 up.

```
var a :int^4 = 1,2,3,4;
for i in 0..<4 do
  println$ a.i;
done
```

The projection here is indicated by the *int* i. An expression can be used provided it is in bounds.

Length

The length of an array may be obtained with the *len* function. The value returned is of type *size* which can be converted to *int* as shown:

```
var x = 1,2,3,4;
var lx = x.len.int;
println$ lx; // 4
```

Value Iteration

A for loop may take an array argument. The control variable takes on all the values in the array starting with the first.

```
var x = 1,2,3,4;
var sum = 0;
for v in x do
  sum = sum + v;
done
println$ sum;
```

2.13 Tuples

A tuple is a heterogenous array. It is a sequence of values of any type.

2.13.1 Type

The type of a tuple is written as a product of the types of the components:

```
int * string * double
```

2.13.2 Values

A tuple value is written as a comma separated sequence:

```
var x : int * string * double = 1, "hello", 4.2;
```

If all the components have the same type, you get an array instead.

2.13.3 Projections

A tuple projection is like an array projection except that only literal integer index is allowed. This is so that the type is known. The indices are zero origin, as for arrays.

```
var x : int * string * double = 1, "hello", 4.2;
println$ x.1; // string
```

Unit Tuple

There is a special tuple with no components. It is given the type *1* or *unit*. The value is written *()*.

2.14 Procedures

A sequence of statements can be wrapped into a named entity called a *procedure*. In addition, a procedure may accept an argument. The accepting variable is called a parameter.

2.14.1 Type

The type of a procedure with parameter type T is written

```
T -> 0
```

A procedure is a subroutine, it returns control, but it does not return a value. To be useful, a procedure must change the state of the program or its environment. This is called an effect.

Procedures in Felix are first class and can be used as values.

2.14.2 Definition

A procedure is defined like this:

```
proc doit (x:int) {
   println$ x;
   x = x + 1;
   println$ x;
}
```

A procedure may explicitly return control when it is finished.

```
proc maybedoit (x:int) {
  if x > 0 do
    println$ x;
    return;
  done
  x = -x;
  println$ x;
}
```

If the procedure does not have a return statement at the end, one is implicitly inserted.

A procedure can have a unit argument:

```
proc hello () {
  println$ "Hello";
}
```

2.14.3 Invocation

A procedure is called with a call statement. The identifier *call* may be omitted. If the argument is unit, it also may be omitted.

```
proc hello () {
  println$ "Hello";
}
call hello ();
hello ();
hello;
```

2.15 Functions

Functions encapsulate calculations.

2.15.1 Type

The type of a function with a parameter type D returning a value type C is written

```
D -> C
```

2.15.2 Definition by Expression

A function can be defined by an expression:

```
fun square (x:int) : int => x * x;
```

or without the return type:

```
fun square (x:int) => x * x;
```

in which case it is deduced.

2.15.3 Definition by Statements

More complex functions can be defined by statements.

```
fun addUp(xs:int^4) : int = {
  var sum = 0;
  for x in xs do
    sum = sum + x;
  done
  return sum;
}
```

The return type can be elided:

```
fun addUp(xs:int^4) = {
  var sum = 0;
  for x in xs do
    sum = sum + x;
  done
  return sum;
}
```

2.15.4 No side effects

The effect of a function must be entirely captured in its returned value; that is, it may not have any side effects. This asumption is currently not checked, so you could write code like this:

```
var mutMe = 0;

fun addUp(xs:int^4) : int = {
  mutMe = 1;  // bad!
  var sum = 0;
  for x in xs do
    sum = sum + x;
  done
  return sum;
}
```

However, this kind of usage may be useful from time to time, for example for debugging.

The lack of side effects in a function are used in optimizations, and the optimizations may have an effect on program behavior. For example, the following toy program takes the second projection (`. 1`) on a tuple involving three function calls. Since functions are assumed to have no side effects, the other function calls (`f` and `h`) are erased as their return values are never used.

```
fun f(x:int) = {
  println "hi from f!";
  return 2*x;
}
fun g(x:int) = {
  println "hi from g!";
  return 3*x;
}
fun h(x:int) = {
  println "hi from h!";
  return 4*x;
}

val res =  (f 5, g 5, h 5) . 1;
println res;
```

The output of the program is just:

```
hi from g!
15
```

2.15.5 Purity

Functions can further be annotated to be `pure` or `impure`, but at the moment, the semantics of these are not defined and are not checked:

```
pure fun addUp(xs:int^4) : int = {
  // ...
}

// or

impure fun addUp(xs:int^4) : int = {
  // ...
}
```

2.16 Function Application

In an expression, a function *f* can be applied to an argument *x* like this:

```
f x
```

This is known as forward, or prefix, application, although it can also be considered, somewhat whimsically, that the gap between the *f* and the *x* is an infix operator known as *operator whitespace*.

Although this is the mathematical syntax for application, many programmers, may prefer to swap the order of the function and argument like this:

```
x.f
```

This is known as reverse application. Operator dot binds tighter than operator whitespace so this expression:

```
g x.f
```

is parsed as

```
g (x.f)
```

Both operator dot and operator whitespace are left associative.

Another application operator is stolen from Haskell:

```
h $ g $ h $ x;
```

Operator *$* binds more weakly then either dot or whitespace, and is right associative so the above parses as:

```
h (g (h x));
```

Finally, there is a shortcut for applying a function to the unit tuple ():

```
#f
```

which means the same as:

```
f ()
```

Operator hash *#* is a prefix operator that binds more tightly than the other application operators.

2.17 Pythagoras

The ancient Greek mathematician Pythagoras is famous for the invariant of right angle triangles:

$$h^2 = w^2 + a^2$$

where *h* is the hypotenuse, *w* is the width of the base, and *a* is the altitute.

We can calculate the hypotenuse in Felix like this:

```
fun hypot (w:double, a:double) : double =>
  sqrt (w^2 + a^2)
;

println$ hypot (3.0, 4.0);
```

The type *double* is a standard double precision floating point real number.

The *sqrt* function is in the library, and calculates the square root of a double precision number.

The operator ^ denotes exponentiation, in this case we are squaring, or muliplying the argument by itself twice, the literal *2* is a value of type *int*, a type of small integers.

Of course, the operator + is addition.

The *fun* binder is used here to define a function. Then we give the function name we want to use, in this case *hypot*.

Then, in paranthesis we give a comma separated list of parameter specifications. Each specification is the name of the parameter, followed by its type.

It is good practice, but not required, to follow the parameters with *:* and the return type of the function.

Then the => symbol is used to begin the formula defining the function in terms of the parameters.

The function can be used by applying it to an argument of the correct type, in this case a pair, or tuple, of two numbers of type *double*.

The *println* is then called on the application using the application operator *$*.

CHAPTER 3

Felix 102: More Detail

3.1 Record

A record is like a tuple, except the components are named:

```
var x = (a=1, b="hello", c=42.0);
println$ x.b;
```

Actually, you can use a blank name, or leave a name out:

```
var x = (a=1,42.0,n""="What?");
```

Note the use of the special identifier form *n"..."* in which the text of the identifier is zero length.

3.1.1 Duplicate Fields

Fields names in a record can be duplicated:

```
var x = (a=1,a=2,32,77);
```

In this case, when the field name is used to access a component it refers to the left most instance of the field. While this may seem like an unusual feature in isolation, it is needed to support polyrecords (row polymorphism).

There is a special case: if all the field names are blank, the the record is a tuple. So in fact tuples are just a special case of records.

3.1.2 Function Application

Earlier we saw examples of function application, but function application is implicitly performed on tuples and records:

```
fun f(x:int,y:double)
// accepts either of the following
f (1,2.1)
f (x=1,y=2.1)
```

The order doesn't matter if you use names, except for duplicates.

3.2 Classes

In Felix a class is used to provide a space for defining functions and procedures.

```
class X {

  // quadratic ax^2 + bx + c, solution:
  fun qplus (a:double, b:double, c:double) =>
    (-b + sqrt (sqr b + 4.0 * a * c)) / (2.0 * a)
  ;

  // square it
  private fun sqr(x:double) => x * x;
}
```

Notice the *qplus* function can call the *sqr* function even though it is defined later. Felix uses random access, or setwise, lookup, not linear lookup.

In order to use a function in a class, we can use explicit qualification:

```
println$ X::qplus(1.0, 2.0, 3.0);
```

Alternatively, we can open the class:

```
open X;
println$ qplus(1.0, 2.0, 3.0);
```

However we cannot access the function *sqr*, because it is private to the class.

A class definition must be contained in a single file, it cannot be extended.

3.3 Generic Functions

Generic functions and procedures provide a simple definition.

```
// generic function
fun add3 (x,y,x) => x + y + z;

// used with different argument types
println$ add3 (1,2,3); // 6
println$ add3 (1.0,2.0,3.0); // 6.0
println$ add3 ('Hello',' ','World'); // Hello World
```

For each uses of a generic function, Felix makes a copy and adds the argument types. So the three calls above actually call these automatically generated functions:

```
fun add3 (x:int, y:int, z:int) => x + y + z;
fun add3 (x:double, y:double, z:double) => x + y + z;
fun add3 (x:string, y:string, z:string) => x + y + z;
```

Note that the rewritten functions are generated in the same scope as the generic function so any names used in the generic function refer to the names in the generic function's original scope.

3.4 Lists

Lists are a fundamental, polymophic data type. Felix list is a singly linked, purely function data type. All the values in a list have the same type.

3.4.1 Creating a list.

A list can be created from an array:

```
var x = list (1,2,3,4);
```

A more compact notation is provided as well:

```
var x = ([1,2,3,4]);
```

3.4.2 Empty lists

For an empty list there are two notations:

```
var x = list[int] (); // empty list of int
var y = Empty[int];   // empty list of int
```

3.4.3 Displaying a list

A list can be converted to a human readable form with the *str* function, provided the values in the list can be converted as well:

```
println$ "A list is " + ([1,2,3,4]).str;
```

3.4.4 Concatenation

Lists can be concatenated with the + operator:

```
println$ list (1,2,3) + ([4,5,6]);
```

3.4.5 Length

The length of a list is found with the *len* function, the result is type *size*:

```
println$ ([1,2,3,4]).len; // 4
```

3.4.6 Prepending an element

A new element can be pushed on the front of a list with the Cons function or using the infix *!* operator, or even with +:

```
var a = ([2,3,4]);
var b = Cons (1, x);
var c = 1 ! a;
var d = 1 + a;
```

The lists *b*, *c* and *d* all share the same tail, the list *a*. This means the prepend operation is O(1). It is safe because lists are immutable.

The use of + is not recommended because it is rather too heavily overloaded. In particular note:

```
1 + 2 + ([3,4]) // ([3,3,4])
1 + (2 + ([3,4]) // ([1,2,3,4])
```

because addition is left associative.

3.4.7 Pattern matching lists

Lists are typically decoded by a recursive function that does pattern matching:

```
proc show(x:list[int]) =>
  match x with
  | Empty => println$ "end";
  | head ! tail =>
    println$ "elt= " + head.str;
    show tail;
  endmatch
;
```

The text between the | and => is called a pattern. To analyse a list, there are two cases: the list is empty, or, the list has a head element and a following tail. The procedure prints "end" if the list is empty, or the head element followed by the tail otherwise.

3.5 Option Type

Another useful type that requires pattern matching is the polymorphic option type *opt*. It can be used to capture a value, or specify there is none:

```
fun divide (x:int, y:int) =>
  if y == 0 then None[int]
  else Some (x/y)
  endif
;
```

3.5.1 Pattern matching optional values

This is done like:

```
proc printopt (x: opt[int]) {
  match x with
  | Some v => println$ "Result is " + v.str;
  | None => println$ "No result";
  endmatch;
}
```

3.6 Varray

A varray is a variable length array with a construction time bounded maximum length. Unlike ordinary arrays, varrays are mutable and passed by reference. Underneath a varray is just a pointer.

3.6.1 Empty Varray

An empty varray can be constructed by giving the bound, the bound must be of type size:

```
var x = varray[int] 42.size;
```

The type of the varray must be specified in this form.

3.6.2 Construction from container

A varray can be constructed from an ordinary array, list, or string, and from an another varray with or without a bound specified:

```
var v4 = varray (1,2,3,4);              // length 4, maxlen 4
var v8 = varray (v4, 8.size);           // length 4, maxlen 8
var y8 = varray (v8);                   // length 4, maxlen 4
var y12 = varray (y8, 12.size);         // length 4, maxlen 12
var z4 = varray ([1,2,3,4]);            // length 4, maxlen 4

var s12 = varray "Hello World";         // trailing NUL included!
```

3.6.3 Construction from default value

A varray can also be built to given size and filled with a default value:

```
var v100 = varray (100.size, char 0); // buffer
```

3.6.4 Length

The length of a varray is given by the *len* function, the bound is given by the *maxlen* function:

```
var x = varray 42.size;
println$ "len=" + x.len.str + ", maxlen=" + x.maxlen.str;
```

3.6.5 Extend and Contract

A new element can be pushed at the end of a varray with the push_back procedure, provided the resulting array doesn't exceed its maxlen bound. Similarly an element can be removed from the end, provided the array isn't empty:

```
var x = varray[int] 42.size;
x.push_back 16; // length now 1
x.pop_back;     // remove last element
```

3.6.6 Insert at position

An element can be inserted at a given position, provided the position does not exceed the current length, and is greater than or equal to the maxlen:

```
var x = varray[int] 42.size;
insert (x, 0, 42);
insert (x, 0, 41);
insert (x, 2, 42);
```

3.6.7 Erase elements

Elements can be erased by giving the position to erase, or, an inclusive range:

```
var x = varray (1,2,3,4,5,6);
erase (x, 2);
erase (x, 2, 20);
```

This procedure cannot fail. Attempts to erase off the ends of the array are simply ignored.

3.6.8 Get an element

An element can be fetched with the get function, provided the index is in range:

```
var x = varray (1,2,3,4,5,6);
println$ get (x, 3.size); // 4
println$ x.3;             // 4
```

The last form allows any integer type to index a varray.

3.6.9 Set an element

An element can be modified with the set procedure:

```
var x = varray (1,2,3,4,5,6);
set (x, 3.size, 99); // 4 changed to 99
```

3.7 Objects, Values and Pointers

Felix supports both functional and imperative programming styles. The key bridge between these models is the pointer.

An array in Felix is an immutable value, it cannot be modified as a value. However an array can be stored in a variable, which is the name of an object. An object has two significant meanings: it names a storage location, and also refers to the value located in that store.

In Felix, the name of a variable denotes the stored value, whilst the so-called address-of operator applied to the variable name denotes the location:

```
var x = 1,2,3,4;  //1: x is an array value, and ..
var px = &x;      //2: it also denotes addressable store
var y = *px;      //3: y is a copy of x
px <- 5,6,7,8;    //4: x now stores a new array
```

This code illustrates how to get the address of a variable on line 2, to fetch the value at the address in line 3, and to modify the value at the address, in line 4.

The prefix symbol *&* is sometimes called the address-of operator, however it is not an operator! Rather, it is just a way to specify that we want the address of the store a variable denotes, rather than the value stored there, which is denoted by the plain variable name.

The address is a Felix data type called a pointer type. If a variable stored a value of type T, the pointer is denoted by type &T.

In line 3 we use the so-called dereference operator, prefix *, to denote the value held in the store to which a pointer points. Dereference is a real operator.

In line 4, we use the infix left arrow operators, which is called *store-at*, it is used to store right hand argument value in the location denoted by the left hand pointer value.

3.8 The new operator

Felix also provide the prefix *new* operator which copies a value onto the heap and returns pointer to it.

```
var px = new 42;
var x = *px;  // x is 42
px <- 43;     // px now points to 43
```

This is another way to get a pointer to an object, which allows the value stored to be replaced or modified.

3.9 Pointer projections

All product types including arrays, tuples, records, and structs provide value projections for fetching parts of the value, the parts are called components:

```
var ax = 1,2,3,4;                  // array
var ax1 = ax.1;                    // apply projection 1 to get value 2

var tx = 1, "hello", 42.0;         // tuple
var tx1 = tx.1;                    // apply projection 1 to get value "hello"

var rx = (a=1, b="hello", c=42.0); // record
var rx1 = rx.b;                    // apply projection b to get value "hello"

struct X {
  a:int;
```

(continues on next page)

(continued from previous page)

```
  b:string;
  c:double;
}
var sx = X (1, "hello", 42.0);       // struct
var sx1 = sx.b;                      // apply projection b to get value "hello"
```

Arrays and tuples have numbered components, and thus are accessed by numbered projections, records and structs have named components and thus values are accessed by named projections.

Although the indicators here are numbers and names, value projections are first class functions. The functions and their types, respectively, are:

```
proj 1: int^4 -> int
proj 1: int * string * double -> string
b: (a:int, b:string, c:double) -> string
b: X -> string
```

These are value projections. To store a value in a component of a product type, we must first obtain a pointer to the store in which it is located, and then we can apply a *pointer projection* to it, to obtain a pointer to the component's store. Then we can use the store-at procedure to set just that component, leaving the rest of the product alone:

```
&ax . 1 <- 42;          // array
&tx . 1 <- "world";     // tuple
&rx . b <- "world";     // record
&sx . b <- "world";     // struct
```

In each case we use the same projection index, a number or a name, as for a value projection, but the projections are overloaded so they work on pointers too. These pointer projections are first class functions, here are their types, respectively:

```
proj 1: &(int^4) -> &int
proj 1: &(int * string * double) -> &string
b: &(a:int, b:string, c:double) -> &string
b: &X -> &string
```

What is critical to observe is that pointers are values, and the pointer projections are first class, purely functional functions. Unlike C and C++ there is no concept of lvalues or references. The store-at operator is a procedure, and so it is used in imperative code, but the calculations to decide where to store are purely functional.

The programmer should note that C address arithmetic is also purely functional, however, C does not have any well typed way to calculate components of products other than arrays: you do the calculations only by using the *offsetof* macro and casts.

C++ has pointers to members, but the calculus is incomplete, they cannot be added together!

In Felix, projections are functions so adding component offsets in products is, trivially, just function composition!

CHAPTER 4

Felix 102a: Regular Expressions

4.1 Regular Expressions

TBD

CHAPTER 5

Felix 102b: File I/O

5.1 Filenames

Filename function reside in class Filename. This class is not open by default, it is recommended the functions below are called explicitly qualified by *Filename::*.

5.1.1 Path separator

A function returning the system path separator, / on Unix, and '' on Windows:

```
fun sep: 1 -> string;
```

5.1.2 Path kind

A function detecting if a filename is absolute or relative

```
fun is_absolute_filename : string -> bool;
```

5.1.3 Root

A function returning the filesystem root directory name, / on Unix and 'C:' on Windows:

```
virtual fun root_subdir : string -> string;
```

5.1.4 Binary codefile Extensions

Functions returning the extensions (including the dot)

```
virtual fun executable_extension : 1 -> string;
virtual fun static_object_extension: 1 -> string;
virtual fun dynamic_object_extension: 1 -> string;
virtual fun static_library_extension: 1 -> string;
virtual fun dynamic_library_extension: 1 -> string;
```

5.1.5 Path Splitting

split1 returns a pair consisting of a directory name and basename with the separator between them lost except in the special case "/x" where the "/" is kept as the directory name. If there is no separator, the path is the basename and the directory name is the empty string (NOT . !!!)

```
fun split1(s:string): string * string;
```

split a filename into a list of components.

```
fun split(s:string)=> split (s, List::Empty[string]);
```

Get the basename of a path (last component).

```
fun basename(s:string) :string;
```

Get the directory name of a path (all but the last component).

```
fun dirname(s:string) : string;
```

Return a list of all the directory names in a path. For example a/b/c gives "a", "a/b"

```
fun directories (s:string) : list[string];
```

Split off extension. Includes the dot. Invariant: input = basename + extension. Works backwards until it hits a dot, path separator, or end of data. If a dot, strip it and the tail of the string, otherwise return the original string.

```
fun split_extension (s:string): string * string;
```

Remove an extension from a filename if there is one.

```
fun strip_extension (s:string) => s.split_extension.0;
```

Get extension if there is one. Includes the dot.

```
fun get_extension (s:string) => s.split_extension.1;
```

5.1.6 Path joining

Join two pathnames into a single pathname. split and join are logical inverses, however join is not not associative: join("x", join("","y")) = "x/y" whereas join(join("x",""),"y") = "x//y" since split pulls components off from the RHS we have to fold them back from the left

```
fun join(p:string, b:string) : string;
```

Note it is common to write in your code:

```
fun /(p:string, b:string) => Filename::join (p,b);
```

Join all the strings in a list into a pathname.

```
fun join(ps: List::list[string]): string;
```

5.2 FileStat

Information about files can be found from class FileStat. It is not open by default.

Get information about a file into a status buffer. Sets error code at argument 3 pointer.

```
proc stat: string * &stat_t * &int;
```

set access and modification time of a file. Sets error code at argument 4 pointer. Times are in seconds, nominally from Epoch (Jan 1 1970).

```
proc utime: string * double * double * &int;
```

Change read,write permissions for group, owner etc. Return 0 on success. On Windows this function may silently fail to obey unsupported operations.

```
gen chmod: string * mode_t -> int;
```

set mask for subsequent permissions. On Windows this function may silently fail to obey unsupported operations.

```
gen umask: mode_t -> mode_t;
```

Abstracted platform independent file type taxonomy.

```
variant file_type_t =
  | PIPE
  | STREAM
  | DIRECTORY
  | BLOCK
  | REGULAR
  | SYMLINK
  | SOCKET
  | INDETERMINATE
  | NONEXISTANT
  | NOPERMISSION
;
```

Get the file type from a file stat buffer.

```
virtual fun file_type: &stat_t -> file_type_t;
```

Fill a stat buffer with information about a file.

```
gen stat(file: string, statbuf:&stat_t);
```

Get a file last modification time from a stat buffer. Time is in seconds.

```
fun mtime: &stat_t -> double = "(double)($1->st_mtime)";
```

Get a file creation time from a stat buffer. Note: not available on Unix. Time is in seconds.

```
fun ctime: &stat_t -> double = "(double)($1->st_ctime)";
```

Get modification time of a file by name. Time is in seconds.

```
fun filetime(f:string):double;
```

Set the last access and modification time of a file by name.

```
gen utime(f:string, a:double, m:double): bool;
```

Set the last access and modification time of a file by name, where the two times are given by a single argument.

```
gen utime(f:string, t:double);
```

Check if a file exists.

```
fun fileexists(f:string):bool=> filetime f != 0.0;
```

Find the type of a file.

```
fun filetype(f:string):file_type_t;
```

File time conversions.

```
fun past_time () => -1.0;
fun future_time () => double(ulong(-1)); // a hacky way to get a big number

fun strfiletime0 (x:double) :string;
fun strfiletime (x:double) : string;
fun dfiletime(var f:string, dflt:double) : double;
```

CHAPTER 6

Felix 103: Polymorphism

An introduction to parametric polymorphism.

6.1 Polymorphic Functions

Felix allows function to be polymorphic. This means you can write a function that works, in part, for any type.

```
fun swap[A,B] (x:A, y:B) : B * A => y,x;
```

This is called parametric polymorphism. The names *A* and *B* are called type variables. The above function will work for any actual types:

```
println$ swap[int,string] (42, "Hello");
```

Here, the specific types used we given explicitly. This is not required if the types can be deduced from the arguments of the application:

```
println$ swap(42, "Hello");
```

Here, *A* must be *int* because parameter x has type *A*, and the argument 42 has type *int*. Similarly, *B* must be *string* because "hello" has type string.

6.2 Higher Order Functions

In Felix, functions are first class which means a function can be used as a value. A function which accepts another function as a parameter, or returns a function as a result, is called a *higher order function*, abbreviated to *HOF*.

Here's an example:

```
fun twice (x:int):int => x + x;
fun thrice (x:int):int => x + x + x;

fun paired (f:int->int, a:int, b:int) =>
  f a, f b
;

println$ paired (twice,2,3);
println$ paired (thrice,2,3);
```

Here, the function *twice* is passed as an argument to *paired*, binding to the parameter *f*, which is then applies to the arguments *a* and *b* to obtain the final result.

Then, we do it again, this time passing *thrice* instead.

6.3 Polymorphic Classes

Polymorphic classes can be used to systematically introduce notation systems representing algebras.

For example let us start with a sketch of the library class *Eq* which introduces equivalence relations:

```
class Eq[t] {
  virtual fun == : t * t -> bool;
  virtual fun != (x:t,y:t):bool => not (x == y);

  axiom reflex(x:t): x == x;
  axiom sym(x:t, y:t): (x == y) == (y == x);
  axiom trans(x:t, y:t, z:t): x == y and y == z implies x == z;

  fun eq(x:t, y:t)=> x == y;
  fun ne(x:t, y:t)=> x != y;
}
```

The class has a single type parameter *t*. Two *virtual* functions are introduced first, == for equivalence, and *!=* for inequivalence.

The type of the == function is given but it is not defined. We only have the interface.

The inequivalence is defined as the negation of equivalence.

Next we have three axioms which specify the required semantics.

The first axiom, *reflexivity*, says a value is equivalent to itself.

The second axiom, *symmetry*, says a if x is equivalent to y, then y is equivalent to x.

The third axiom, *transitivity* says if x is equivalent to y, and y is equivalent to z, then z is equivalent to z too.

In this case these three laws are a complete specification, and, the laws are independent, in that one cannot be deduced from another. These two properties mean that our rules are in fact mathematical axioms.

Felix does not require *axioms* actually be axioms. However, simple rules which can be derived from the stated axioms can be specified by *lemma*. The idea is that lemmas can be so eaily proven from the axioms that an automatic theorem prover can do it, without any further assistance.

You can also a *theorem* which is a rule that can be proven from the axioms, but where the proof requires a human, or a human with a proof assistant, to establish its correctness.

Finally in our class, we have two named function defined in terms of the virtual functions. Notice these functions are not virtual.

6.4 Class Instances

Our *Eq* class defines a simple algebra which specifies an interface for equivalence relations together with their semantics. We have to use an *instance* to populate the algebra.

```
instance Eq[int] {
  fun == : int * int -> bool = "$1==$2";
}
```

Here we have defined an instance of the Eq class for the argument type *int*. Our code *implements* the Eq class interface, by defining all the non-default virtual methods.

In this case the definition is done by delegating the implementation to C++, using a binding.

We would like equality for *double* as well:

```
instance Eq[double] {
  fun == double * double -> bool = "$1==$2";
  fun != double * double -> bool = "$1!=$2";
}
```

Here we choose to also define the *!=* function as well, overriding the default provided in the class.

When you provide instances, it is usual to ensure there is a definition for all the class methods. Care must be taken to avoid circular definitions, because that will lead to inifinite recursions at run time.

6.5 Using Classes

Once we defined a class and provided some instances, we can use the class for those instances. There are several ways to do this.

6.5.1 Qualified names

You can use a fully qualified name to access a class method:

```
println$ Eq[int]::==(1, 2);
println$ Eq[int]::eq(1, 2);
println$ Eq[double]::eq(1.1, 2.2);
```

You can leave out the type when it can be deduced by overload resolution:

```
println$ Eq::==(1, 2);
println$ Eq::eq(1, 2);
```

6.5.2 Missing Instances

If you try to use a method which has not been defined by an instance for the type you used, you will get an *instantiation error*. This is not a type error, it just means you forgot to provide an instance for that type:

```
// println$ Eq::==("Hello", "World");
// WOOPS! we didn't defined it for strings
```

6.5.3 Opening a Class

The easiest way to use a class is to open it:

```
open Eq;
println$ eq(1,2);
println$ 1 == 2;
println$ 1.1 == 2.2;
```

This is the most common method for script. In fact many classes in the standard library have already been opened for you.

6.5.4 Opening a Class Specialisation

It is also possible to open a specialisation of a class:

```
open Eq[int];
println$ eq(x,y);
println$ x == y;
// println$ 1.1 == 2.2;
// WOOPS, can't find == for double * double
```

This is not an instantiation error, and instance for double exists. The problem is that we didn't open Eq for double, only for int.

6.5.5 Passing a class to a function

You can also pass a class or specialisation to a function. Here is a monomorphic example:

```
fun eq3[with Eq[int]] (x:int, y:int, z:int) =>
  x == y and y == z
;
```

This is exactly the same as opening the class or specialisation inside the function. It is useful because it isolates the access to the class by functional abstraction.

Here is a polymorphic example:

```
fun eq3[T with Eq[T]] (x:T, y:T, z:T) =>
  x == y and y == z
;
```

6.6 Inheritance

When defining classes, you can inherit methods from other classes or specialisations thereof. For example here is a *total order*:

```
class Tord[t]{
  inherit Eq[t];

  virtual fun < : t * t -> bool;
  fun lt (x:t,y:t): bool=> x < y;
```

(continues on next page)

(continued from previous page)

```
  axiom trans(x:t, y:t, z:t): x < y and y < z implies x < z;
  axiom antisym(x:t, y:t): x < y or y < x or x == y;
  axiom reflex(x:t, y:t): x < y and y <= x implies x == y;
  axiom totality(x:t, y:t): x <= y or y <= x;

  fun >(x:t,y:t):bool => y < x;
  fun gt(x:t,y:t):bool => y < x;

  fun <= (x:t,y:t):bool => not (y < x);
  fun le (x:t,y:t):bool => not (y < x);

  fun >= (x:t,y:t):bool => not (x < y);
  fun ge (x:t,y:t):bool => not (x < y);

  fun max(x:t,y:t):t=> if x < y then y else x endif;
  fun \vee(x:t,y:t) => max (x,y);

  fun min(x:t,y:t):t => if x < y then x else y endif;
  fun \wedge(x:t,y:t):t => min (x,y);
}
```

The *inherit* statement pulls in the methods of Eq so you can write:

```
println$ Tord[int]::eq(1,2);
```

and expect it to work. However when instantiating a total order you *cannot* provide a definition for inherited methods, you must provide the instance for the original class:

```
instance Eq[int] {
  fun == : int * int -> bool = "$1==$2";
}
instance Tord[int] {
  fun < : int * int -> bool = "$1<$2";
}
```

Although in this case, we inherited Eq[t], for all t, we could have inherited Eq[int], for example. Instances can only be provided for a class, not a specialisation, because the instances are themselves defining specialisations.

CHAPTER 7

Felix 104: C bindings

7.1 C Bindings

In Felix you can use C bindings to lift types from C and C++. The resulting nominal types are abstract. Unless otherwise specified, all types lifted from C or C++ must be first class; that is, semi-regular. This means that values of those types can be default constructed, copy constructed, copy assigned, move constructed, move assigned, and destroyed; in other words, behave "like an integer".

To lift immobile objects, lift a pointer instead.

7.1.1 Lifting Primitives

Here's how you lift primitive types:

```
type int = "int";
type long = "long";
type ulong = "unsigned long";
type address = "void*";
```

The quoted name must be usable in the forms *T x*, *T *x*. This means you cannot lift a C function pointer type such as *int (*)(double, long)* because the variable name has to go just before the *.

7.1.2 Lifting Class types

Class types require a bit more work:

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

Here, the *?1* annotation means the first type variable so that for example *vector[ulong]* lifts *::std::vector<unsigned long>*.

When lifting C++ types make sure to always specify the absolute pathname of the type. Starting with *::* is strongly recommended. This avoids any possible ambiguity in generated code.

The *requires* clause with *header* option tells Felix to emit the quoted text in the header file of generated code, before the type is used. If the type is not used, the *#include* will not be emitted.

7.1.3 C function types

C function types can be defined directly in Felix:

```
typedef int2int = int --> int;
// typedef int (nt2int*)(int)
```

The type *int2int* is an alias, and the C function type is a structural type, not nominal type.

7.1.4 Lifting C structs and unions

C structs can be lifted in a way that exposes their fields, provided the fields are not *const*:

```
header mypair_h =
"""
  struct mypair { int x; int y; };
""";

cstruct mypair {
  x: int;
  y: int;
} requires mypair_h;
```

Here, to make the code complete, the C definition is given with floating header code so it can be required by the binding. Usually, it will be given in a header file.

A *cstruct* works exactly the same as a struct except that no definition is emitted.

There are caveats! Felix generates the usual elementwise constructor but it will lead to corruption unless the *cstruct* model of the type in Felix exactly matches the definition in C. However this is not required for access to the field components. In particular the *cstruct* construction in Felix can be used to model a C union as well.

It's not possible to specify a namespace or class qualification for the C type.

7.1.5 Lifting functions and procedures

Since primitive types lifted from C are abstract, we have to be able to define operations on them with C bindings too.

```
proc push_back[V]: &vector[V] * V = $1->push_back($2);";
fun front[V]: vector[V] -> V = "$1.front()";

var v : vector[int];
push_back (&v, 42);
println$ v.front; // 42
```

We use *$1* abd *$2* fir the first and second arguments, respectively.

7.1.6 Lifting Constants and Expressions

You can lift a C constant, variable, or even expressions using the *const* binder:

```
const pi : double = "M_PI" requires C99_headers::math_h;
```

7.1.7 Lifting enums

A special shorthand is available for lifting C enums, intended for sequences:

```
cenum color = red,blue,green;
```

This is (roughly) equivalent to:

```
type color = "color";
const red: color = "red";
const blue: color = "blue";
const green: color = "green";
fun == : color * color -> bool = "$1==$2";
```

Note in particular equality is automatically defined. This is required for using the enumeration values in pattern matches.

7.1.8 Lifting Flags

A special shorthand is available for lifting C enums, intended for flags:

```
cflags color = red,blue,green;
```

This defines equality as for cenums, but also makes all the standard bitwise operations available.

7.2 Floating Insertions

A floating insertion is a string containing C/C++ code which is literally inserted into the C++ that Felix generates.

7.2.1 Insertion Locations

There are two locations for the insertion:

1. *header* insertions go near the top of the generated header file
2. *body* insertions go near the top of the generated body file *after #include* directives, but before any definitions generated by Felix

```
header '#include "interface"`;
body 'void implementation() {}';
```

7.2.2 Polymorphic Insertions

There are two kinds of insertions:

1. plain insertions
2. polymorphic insertions

A plain insertion is inserted as written. A polymorphic insertion is inserted as written except that it may contain type parameters written using the *?9* kind of notation. Polymorphic insertions must be named and dummy parameters specified, as with a C type binding:

```
//$ In place list reversal: unsafe!
// second arg is a dummy to make overload work
proc rev[T,PLT=&list[T]] : &list[T] = "_rev($1,(?1*)0);" requires _iprev_[T,PLT];

body _iprev_[T,PLT]=
  """
  static void _rev(?2 plt, ?1*) // second arg is a dummy
  { // in place reversal
    struct node_t { void *tail; ?1 elt; };
    void *nutail = 0;
    void *cur = *plt;
    while(cur)
    {
      void *oldtail = ((node_t*)FLX_VNP(cur))->tail;   // save old tail in temp
      ((node_t*)FLX_VNP(cur))->tail = nutail;          // overwrite current node tail
      nutail = cur;                                    // set new tail to current
      cur = oldtail;                                   // set current to saved old tail
    }
    *plt = nutail;                                     // overwrite
  }
  """
;
```

In this example we defined a function *_rev* in C++, used to reverse a list in place. The insertion containing it, *_iprev_[T,PLT]* is polymorphic, the actuual types are specified in the C bindings for *rev*.

7.2.3 Requires Clause

C bindings may specify a *requires* clause with a list of requirements. For floating insertions there are three options:

1. require literally given header
2. require literally given body
3. require by insertion tag name

Insertions themselves may have requirements.

7.2.4 Tag names

Insertion tag names are scoped by classes and may require explicit qualification. However the tag namespace is independent of ordinary definitions.

Any number of insertions may have the same tag name.

7.2.5 Closure Formation

When a named requirement is triggered, all the insertions with that tag name, and any insertions they themselves require, recursively, will be inserted. Requiring a non-existant tag is an error.

Circular references are permitted.

7.2.6 Order of Insertion

Insertion of the same kind defined in the same Felix file retain their order in the generated output, in otherwords, they're emitted in the order of writing, not the order of dependence.

If a text has been emitted, it will not be emitted again: duplication is eliminated. Note this rule applies to polymorphic insertions *after* substitution of types.

7.2.7 Global Insertions

An unnamed insertion specified by a *requires* statement is taken as a requirement of all C bindings and floating insertions in the class containing it, including any nested classes.

7.2.8 Triggering Insertions

Insertions are only triggered if the C binding or insertion they're attached to is actually used in the final program. This reduces the output to what is actually needed.

7.3 Fixed Insertions

Felix also allows you to insert executable code literally in fixed places inside Felix code. There are two cases.

7.3.1 Lifting Plain Statements

Statements can be lifted with the plain *cstmt* statement:

```
proc hello() {
  cstmt """
    std::cout<< "Hello World" << ::std::endl;
    std::cout<< "C++ Embedded in Felix" << ::std::endl;
  """;
}
```

7.3.2 Lifting NonReturning Executable Statements

If the C statement does not return, use the *noreturn* option:

```
proc leave() {
  noreturn cstmt "::std::exit(0);";
}
```

7.3.3 Lifts with arguments

Lifted code can accept arguments:

```
proc error(x:int) {
  noreturn cstmt "::std::exit($1);" x;
}
```

The argument can be a tuple, the components are inserted by the usual rules for C bindings using *$9* style notation.

7.3.4 Lifting Expressions

An expression can be lifted too, however the type must be given:

```
var x = cexpr[int]"42" endcexpr;
```

If the expression is not atomic, it is wise to enclose it in parentheses.

7.3.5 Lifting Variables

There is a short hand for lifting variables:

```
cvar[int] M_PI;
```

which is equivalent to:

```
cexpr[int]"M_PI" endcexpr;
```

7.4 Binding Callbacks

C code often uses callbacks. Callbacks always consist of two type of functions:

1. A registration function
2. The callback function

The registration function is called with the callback function as an argument, and it stores the callback inside a server.

Then when some specified kind of event occurs, the callback is invoked by the server.

Callback functions almost always have a special parameter which is typically called *client_data* and is a *void** to some arbitrary data. When the callback is registered, a particular client data pointer is accepted as well. When the callback is invoked, the server passes the registered client data pointer to it.

Felix provides a way to write C callbacks in Felix. To do this, a fixed function is generated, of the type required for the callback, which casts the client data pointer to a closure of the Felix function implementing the callback, and then invokes it.

Both procedural and function callbacks are supported. The Felix code has full access to the garbage collector, but must not actually trigger a collection. Service calls cannot be done because the C stack is in the way.

The callback is defined like:

```
callback fun cb1 : int * cb1 * double -> int;

callback proc cb2 : int * cb2 * double;
```

Here the special parameter which is the name of the callback type given, *cb1* and *cb2* in the example, indicates the position of the client data pointer.

These functions will be generated in the C code:

```
int cb1(int, void *client_data, double);

void cb2(int, void *client_data, double);
```

To actually define the callbacks in Felix:

```
fun cback1 (x:int, y:double) => (x.double + y).int;

proc cback2 (x:int, y:double) {
  println$ "x=" + x.str + ",y=" + y.str;
}
```

Now, when you register your callback pass *cb1* or *cb2* as the callback, and pass *C_hack::cast[address]cback1* or *C_hack::cast[address]cback2* as the client data pointer.

7.4.1 Complete Example

A function callback with an *int* argument.

Procedure callbacks with an *int* and *double* arguments.

CHAPTER 8

Felix 105: Nominal Types

Structs, Variants, and C bindings

8.1 Structs

8.1.1 Definition

The *struct* construction introduces a nominally typed, or named product.

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

instance Str[Point] {
  fun str (p:Point) =>
    "Point(" + p.x.str + ", " + p.y.str + ")"
  ;
}

var p = Point (2,3);
println$ p.str;
```

Here, we note a Point consists of two integers, x and y, and we can make a Point by applying the type name Point to a pair of integers.

Also we provide an instance of the *Str* class with method *str* or type *Point->string* for converting a point to a human readable form. *Str* is a standard library class which is universally open.

8.1.2 Methods

Struct can have methods of two kinds: *accessors* and *mutators*. Here is an expanded version of Point:

```
struct Point {
  x: int;
  y: int;
  fun norm => max (self.x, self.y);
  proc reset { self.x <- 0; self.y <- 0; }
  proc set (a:int, b:int) { self.x <- a; self.y <- b; }
}
var p = Point (1,2);
var y = p.norm;
p.set(p.x + 1, p.y + 1);
```

Accessors, with *fun* binder, implicitly get an extra parameter *self* of type *Point*. Mutators, with *proc* binder, implicitly get an extra parameter *self* of type *&Point*, a pointer to a Point.

8.1.3 Object Closures

In fact, these methods are ordinary functions and procedures. The nesting is just syntactic sugar for:

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

fun pythag (self: Point) => max(self.x, self.y);
proc reset (self: &Point) { self.x <- 0; self.y <- 0; }
proc set (self: &Point) (a:int, b:int) { self.x <- a; self.y <- b; }
```

Because of this, the methods are higher order functions and we can form closures over objects of type Point:

```
var p = Point (1,2);
var setp = &p.set; // object closure
var resetp = { &p.reset; }; // object closure
setp (23,42);
println$ p.str;
resetp;
println$ p.str;
```

Note that *&p.reset* would call reset so we need to wrap the call in the general closure constructor *{_}*.

8.2 Struct Constructors

Sometimes, you want to define extra constructors for a struct, you can do this with a *ctor* function.

```
struct Point {
  x: int;
  y: int;
}
ctor Point (a:int) => Point (a,a);
var p = Point 42;
```

In fact, this works for any type, not just a struct, provided the type has a single word name. If it doesn't, you can always introduce a *typedef*:

```
typedef pair = int * int;
ctor pair (x:int) => x,x;
```

Using a *typedef* is a way to introduce an extra constructor for a type with the same signature; that is, provide *named constructors*. For example:

```
struct complex {
  x: double;
  y: double;
}

typedef polar = complex;
ctor polar (modulus: double, argument: double) =>
  complex (modulus * cos argument, modulus * sin argument)
;
```

8.3 Struct Subtyping

Felix structs do not provide any inheritance mechanism. However, you can define a subtyping coercion:

```
struct Point {
  x: int;
  y: int;
  fun norm => max (self.x,self.y);
  proc reset { self.x <- 0; self.y <- 0;}
  proc set (a:int, b:int) { self.x <- a; self.y <- b; }
}

struct ColouredPoint {
  p: Point;
  colour : int;
}

var colp = ColouredPoint (p, 5);

supertype Point (cp: ColouredPoint) => Point ( cp.p.x, cp.p.y);

println$ colp.norm.str;
```

The coercion has the parameter of subtype ColouredPoint and returns a value of the supertype Point. Now the *norm* method which takes a point will work with a ColouredPoint too, because it is a subtype.

Note that the mutating methods will *not* work because subtyping does *not* extend from values to read-write pointers. In fact, whilst read-pointers are covariant, write pointers are actually contra-variant.

Finally note, explicit subtyping coercions given by the *supertype* construction are transitive.

8.4 Variants

Variants are dual to structs. Whereas structs represent “this and that and the other”, variants represent “this or that or the other”, in other words, alternatives.

8.4.1 Definition

We have already met a common variant, the option type:

```
variant opt[t] =
| None
| Some of t
;
```

The names *None* and *Some* are sometimes called *type constructors*. When they're used to make a value of a variant type, they're functions, known as *injection functions* because they inject their argument into the variant type.

8.4.2 Construction

Here's how we can make values of the option type:

```
var n = None[int];
var s = Some 42;
```

Notice that for the *None* case we have to provide the instance type of *t*, in this case *int*. For the *Some* case, the instance *int* of *t* is deduced by overload resolution and so can be elided.

8.4.3 Extraction

Injections are conditionally invertible, that is, the inverses are partial functions which I call *extractors*. You use these with pattern matches:

```
fun show (x:opt[int]) =>
  match x with
  | None => "None"
  | Some i => "Some(" + i.str + ")"
;
```

Felix "knows" which injection was used to construct a variant value, and selects the matching branch from the *match* expression. This ensures the extractor is well defined. For the *Some* branch, the extract sets *i* to the argument of the injection function used to construct the value.

CHAPTER 9

Felix 106: Uniqueness Typing

Uniqueness Types

9.1 Uniqueness Types

Felix has a special type combinator *uniq* which is used to specify the value of its argument type is exclusively owned. It is usually applied to pointers.

First lets write a procedure to convert a varray of chars to upper case:

```
proc toupper(x: varray[char]) {
  for i in 0 ..< x.len.int
    perform set(x,i,toupper(get(x,i)));
 }
```

Varray's are passed by reference, so this modifies the varray in place. So how would we use this in a function?

```
fun upcase(x: varray[char]) {
  var y = varray x; // copy
  toupper y;
  return y;
 }
```

We can't just call toupper on the parameter x and return it because in Felix functions are not allowed to have side effects.

But consider this case:

```
var upped = upcase$ varray("Hello World");
println$ upped.str;
```

This is inefficient because in *upcase* we copy the array, modify the copy and return it, but the argument *x* is then no longer reachable, so it will be garbage collected.

Why not just modify the argument directly in this case:

```
fun upcase(x: uniq varray[char]) {
  toupper x,peek;
  return x;
 }

 var upped = upcase$ box (varray("Hello World"));
 println$ upped.str;
```

CHAPTER 10

Felix 107: Polymorphic Variants

An advanced and flexible variant based on Ocaml's polymorphic variant types.

10.1 Polymorphic Variants

Felix polymorphic variants are based on those in Ocaml. Polymorphic variant types can be formed from an arbitrary collection of constructors. They satisfy both depth and width subtyping rules.

10.1.1 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.

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?

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 =>
  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.

10.1.2 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.

10.1.3 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;
};
```

(continues on next page)

(continued from previous page)

```
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.

10.1.4 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]
)
```

(continues on next page)

(continued from previous page)

```
;

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
;

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 11

Felix 108: Row Polymorphism and Subtyping

Subtyping and an alternative to subtyping for records with row variables.

11.1 Subtyping Rules

Subtyping rules specify when Felix can perform implicit coercions.

11.1.1 Records

Records support both width and covariant depth subtyping.

Width subtyping

A parameter accepting a certain set of fields can be passed a record value with additional fields. Felix inserts a coercion which builds a new record with only the required fields:

```
fun f(p : (a:int, b: int)) => p.a + p.b;
f (a=1,b=2,c=3); // OK, c field is discarded
```

Depth subtyping

A parameter accepting a record with a field of type P will accept a record with the field of type B provided B is a subtype of P:

```
fun f(p : (a=int, b=(c:int, d:int))) => p.a + p.b.c + p.b.d;
f (a=1,b=(c=2,c=3,d=4)); // OK, b value is coerced
```

In this example, the type of field *b* of the argument is coerced to the type of field *b* of the parameter, as it happens by width subtyping.

Depth and width subtyping can be used simulaneously and apply recursively.

11.1.2 Tuple and array subtyping

Tuples, and thus arrays, support covariant depth subtyping. For example:

```
f(p: (a:int,b:int), d:int) => p.a + p.b +p.c;
println$ f ((a=1,b=2,c=3,42); // OK, coerce first component
```

Although well principled, width subtyping is not supported for tuples and arrays, even though this is a special case of record subtyping. This is because it is likely to be confusing in the presence of overloading.

11.1.3 Polymorphic Variant Subtyping

Polymorphic variants support width and covariant depth subtyping, however, the width subtyping rule is the reverse of that for records, the argument must have less fields:

```
typedef psub = (`A | `B | `C);
typedef psup = (`A | `B | `C | `D);
var v = `A :>> psub;
fun f(p: psup) => match p with
  | `A => "A"
  | `B => "B"
  | `C => "C"
  | `RD=> "D"
  endmatch
;
println$ f v;
```

The function *f* can handle 4 cases, so passing an argument which could only be one of three of them is safe.

11.1.4 Anonymous Sum Type Subtyping

Like tuples anonymous sums support covariant depth subtyping but not width subtyping, for the same reason. It would be confusing if a function with a bool parameter accepted a unit argument, even though in principle 1 is a subtype of 2.

11.1.5 Function subtyping

Functions support subtyping, the domain is contravariant and the codomain is covariant.

11.1.6 Abstract Pointer Subtyping

Abstract pointers are defined by:

```
typedef fun rptr (T:TYPE) : TYPE =>  (get: 1 -> T);
typedef fun wptr (T:TYPE) : TYPE =>  (set : T -> 0);
typedef fun rwptr (T:TYPE) : TYPE => (get: 1 -> T, set : T -> 0);
```

and therefore as record types follow the subtyping rules for records. In particular, the read-write pointer type *rwptr* is a subtype of both the read-only pointer *rptr* and write-only pointer *wptr* by record with subtyping rules.

rptr is covariant by a combination of depth subtyping rules for records and the covariance of function codomains.

wptr is contravariant by a combination of depth subtyping rules for records and contravariance of function domains.

rwptr is invariant since it must be simultaneously covariant and contravariant.

11.1.7 Machine Pointer Subtyping

Felix has three primary machine pointer types, a read-only pointer, a write-only pointer, and a read-write pointer, which is a subtype of the other two types. The type pointed at is invariant.

Although in principle, machine pointers should follow the model for abstract pointers, depth subtyping is not supported because it cannot be concretely implemented in general.

There is however one special exception. In principle, variance can be implemented if the coercion is phantom, that is, it impacts the type system but does not change the underlying machine address. Felix considers a write-only machine pointer to a *uniq T* to be a subtype of a write-only pointer to a *T*. This is necessary so that the procedure:

```
proc storeat[T] ( p: &>T, v: T) = { _storeat (p,v); }
```

works with pointer to a *uniq T*. Without this rule, assignments to uniquely type variable would not be possible, such assignments actually model moves.

11.1.8 Unique Subtypes

The type *uniq T* is a subtype of *T*. This means a parameter of type T can be passed an argument of type *uniq T*, discarding the uniqueness. This works because the compiler back end discards the uniqueness constructor anyhow, so the binding degenerates to a non-unique operation after unique typing rules are validated.

11.1.9 Subtyping with Nominal Types

Felix allows monomorphic nominal types to form subtyping relations by allowing the user to define coercions.

Signed Integers

For example Felix provides:

```
supertype vlong: long = "(long long)$1";
supertype long: int = "(long)$1";
supertype int : short= "(int)$1";
supertype short : tiny = "(short)$1";
```

Such subtyping coercions are transitive, for example, *int* is a subtype of *long*. Felix will generate a composite coercion automatically. If there is more than one composition, the composites must have the same effect because Felix will chose one arbitrarily.

Consider the following functions:

```
fun add(x:int, y:int):int => x + y;
fun add(x:long, y:long):int => x + y;
fun add(x:vlong, y:vlong):int => x + y;
```

together with the subtyping rules above, the functions will behave the same way as addition in C; that is, as if C integral promotion rules applied.

Exact Signed Integers

Felix provides subtyping for normal C signed integer types and exact C signed integer types.

```
supertype int64: int32 = "(int64_t)$1";
supertype int32 : int16 = "(nt32_t)$1";
supertype int16 : int8  "(int16_t)$1";
```

however there are no conversions between signed and unsigned types, between normal and exact types, or between unsigned types.

Real to Complex

There are no conversions between floating point reals. C defined promotions from float to double and double to long double but in fact these are wrong: numerical analysis suggests the safe conversions actually go in the other direction.

However float reals can be embedded in complex numbers of the same precision:

```
supertype fcomplex: float = "::std::complex<float>($1,0.0f)";
supertype dcomplex: double = "::std::complex<double>($1)";
supertype lcomplex: ldouble = "::std::complex<long double>($1)";
```

Note this means any complex parameter will accept a float real argument.

Annoyance

Its very annoying we cannot embed integers into the floats implicitly. However for this to work, the would have to go to *double* alone. To get the embedding to all floating types would require subtyping the floating types to avoid ambiguity.

11.2 Row Polymorphism

11.2.1 Polyrecord Parameters

Row polymorphism provides an alternative to record subtyping where instead of the subtype coercion forgetting extraneous fields, they're preserved as group, even though the client function does not individually know them. Here's a simple example:

```
typedef point = (x:int, y:int);
typedef coloured_point = (x:int, y:int, colour: int);
typedef elevated_point = (x:int, y:int, z:int);

fun step_right[T] (a : (x:int, y:int | T)) => (a with x=a.x+1);

var cp : coloured_point = (x=0, y=0, colour=42);
var ep : elevated_point = (x=0, y=0, z=100);

println$ cp.step_right._strr;
println$ ep.step_right._strr;
```

Here, the type T will be a record type containing all the fields of the argument type other than *x* and *y*. This can vary from call to call. Unlike subtyping, the actual argument is updated with a functional update which preserves all the original fields, even ones the function doesn't know about.

The type of the parameters here is called a polyrecord type, it consists of the usual record field list and a vertical bar followed by another type, which can be a type variable, as in the example.

11.2.2 Polyrecord values

Polyrecord types are ordinary types, you can define values for them:

```
var a = (x=1, y=2 | (e=42));
```

The type of the value on the RHS of the vertical bar must resolve to a record, tuple, array, or unit after monomorphisation, including a polyrecord which so resolves, recursively.

11.2.3 Use with Objects

Because object types are just record types row polymorphism can be used with objects. For example:

```
object coloured_point (var x:int, var y:int, var colour: int) {
  method fun getx () => x;
  method proc setx (newx: int) { x = newx; }
  method proc show { println$ "x=" + x.str + ", y="+y.str + ",colour=" + colour.str; }
};

proc step_right[T] (a : (getx: 1-> int, setx: int->0 | T)) {
  a.setx (a.getx() + 1);
}

var cp = coloured_point (x=0,y=0,colour=42);
cp.step_right;
cp.show();
```

CHAPTER 12

Felix 109: Objects and Plugins

Java Like Dynamic Objects and Plugins

12.1 Objects

12.1.1 Basic Construction

Felix provides a dynamic object construction mechanim.

```
object ob (x:string) = {
    var age = 0;
    method fun name () => x;
    method fun underage () => age < 21;
    method proc setage (y:int) { age = y; }
};
var joe = ob "joe";
joe.setage 42;
println$ "name " + joe.name() + " is " +
  if joe.underage()
  then "underage"
  else "of age"
  endif
;
```

Here, *ob* is an ordinary function with one special property: the return is done automatically, and the returned value is a record of closures of the functions and procedures specified as *method*. They are each closed over the local data frame of the *constructor* function *ob*.

Applying the object constructor *ob* to an argument that matches its parameter *x* creates the actual object, which is an ordinary record value.

12.1.2 Object type

The type of the object above is given by:

```
typedef ob_t =
(
  name: 1->string,
  underage: 1->bool,
  setage: int -> 0
);
```

An alternative syntax is:

```
interface ob_t  {
  name: 1->string;
  underage: 1->bool;
  setage: int -> 0;
};
```

You can use this to specify the return type of the constructor:

```
object ob (x:string) implements ob_t = {
    var age = 0;
    method fun name () => x;
    method fun underage () => age < 21;
    method proc setage (y:int) { age = y; }
 };
```

12.1.3 Dynamics

Felix objects are more flexible than Java or C++ because they're not defined by classes, they're just values contructed by a function. You can, in fact, constuct an object with an ordinary function: the only magic with an *object* bound function is the automatic return of the record of *method* closures.

Because an instance object is just a record, you can apply any record operations to it!

However, the local variables of the constructor to which the method closures are bound are protected by functional abstraction, and so can be considered private. This privacy cannot be bypassed.

But you can do this:

```
var joe = ob "joe";
joe = (joe with name = (fun () => "Joe")); // change name method
println$  "name " + joe.name(); // prints Joe not joe

var xjoe = extend joe with (surname = "Blogs") end;
println$ xjoe.surname;
```

Note, the type of *xjoe* is no longer the same, it has an extra field called *surname*. This field is just a string, not a method.

12.2 Plugins

One of the primary uses of objects is to implement *plugins*. A plugin is a *dynamic load library* or DLL, also called a shared library on Unix platforms.

Felix can load DLLs at run time. However plugins are special DLLs with a specific structure.

12.2.1 Loading Plugins

To load a plugin named "ob_implementation" we do this:

```
var joe =
   Dynlink::load-plugin-func1
     [ob_t,string]
     ( dll-name="ob_implementation", setup-str="")
;
```

The *load-plugin-func1* loads a plugin with DLL basename "ob_implementation", initialising global memory by calling a function *setup_ob_implementation* and passing it the empty string "", then it calls the entry-point function *ob_implementation* which accepts a value of type *string* and returns a value of type *ob_t*.

The DLL will be searched for on the standard OS search path, given by the environment variables *LD_LIBRARY_PATH* on Linux, *DYLD_LIBRRY_PATH* on MacOSX, and *PATH* on Windows. The standard extension for each platform is appended, that will be *.so* on Linux, *.dylib* on MacOSX, and *.dll* on Windows.

The load operation uses a wrapper around the OS load function: *dlopen* on Linux and MacOSX, and *LoadLibrary* on Windows.

The system then uses the symbol finding function to find *setup*. It uses *dlsym* on Linux and MacOSX and *LoadAddress* on Windows. It must be a C function, not a C++ function, and it must accept a C++ string as an argument and return an int.

Then the system find the entry point, which must be a C function which accepts a value of type string, and returns a value of type *ob_t*. The return type and parameter type of the function are given in square brackets in the call to *load-plugin-func1*.

The important thing here is that the type *ob_t* as to be known to both the plugin code as well as the client code. Therefore it is usual to put the interface specification in a separate file and include it in both places, in just the same manner as header files in C and C++. There is no type checking done on loading, so it's important if the interface changes to rebuild both the plugin and the client.

Because plugins typically provide a lot of functions, not just one, we typically provide just one function, and object factory, which will return a set of function packed into a record when called.

12.3 Building a Plugin

Here are specific instructions to build and use a plugin.

12.3.1 Interface File

First we have a file *ob_interface.flx* in the current directory:

```
// ob_interface.flx
interface ob_t  {
  name: 1->string;
  underage: 1->bool;
  setage: int -> 0;
};
```

12.3.2 Implementation File

Now we implement the plugin in *ob_implementation.flx* in the current directory:

```
// ob_implementation.flx
include "./ob_interface";

object ob (x:string) implements ob_t {
  var age = 0;
  method fun name () => x;
  method fun underage () => age < 21;
  method proc setage (y:int) { age = y; }
};

fun setup(x:string) {
  println$ "Setup string `" + x + "`";
  return 0;
}

export fun setup of string as "ob_implementation_setup";
export fun ob of string as "ob_implementation";
```

Here the *include* directive is loading the interface relative to the including file. Put both in the current directory for convenience!

The *export fun* directive generates an *extern "C"* wrapper for a Felix function, giving it the quoted linker name. Because of overloading you have to specify the function domain type to pick the right function (even if there's only one).

12.3.3 Client File

Our plugin client is the file *ob_client.flx* in the current directory:

```
// ob_client.flx
include "./ob_interface";

var ob =
  Dynlink::load-plugin-func1
    [ob_t,string]
    ( dll-name="ob_implementation", setup-str="", entry-point="ob")
;
var joe = ob "joe";
println$ "name " + joe.name();
```

This will load the plugin, initialise it using the setup function, and then run the primary entry point.

12.3.4 Building the plugin

The plugin is built on all platforms using the command line:

```
flx -c -od . ob_implementation.flx
```

The *-c* tells Felix to compile and link the file but not run it. The *-od .* tells Felix to put the linker output in the current directory. The file will have the basename *ob_implementation* on all platforms, and will have an extension appropriate to a DLL on your platform.

12.3.5 Running the client

To run the client on OSX I just did this:

```
flx ob_client.flx
```

That's it! You may need to set the environment variable that controls the DLL search path. So on OSX:

```
DYLD_LIBRARY_PATH=. flx ob_client.flx
```

On Linux:

```
LD_LIBRARY_PATH=. flx ob_client.flx
```

and on Windows:

```
PATH=. flx ob_client.flx
```

12.3.6 Result

The result should be this output on your console:

```
Setup string ``
name joe
```

12.3.7 Debugging

If you set the environment variable *FLX_SHELL_ECHO=1* then all calls the system shell will be echoed to the console. In addition, calls to the system dynamic loader to load a plugin will be reported.

12.4 Static Linked Plugins

12.4.1 Building an Executable

Instead of just running *ob_client* as script, we can build an executable. The command works on all platforms:

```
flx --static -c -od . ob_client.flx
```

This will put an executable *ob_client* in the current directory on Linux or OSX, or *ob_client.exe* on Windows.

You can run the executable:

```
./ob_client
```

and it will work as before, loading the plugin dynamically. Hopefully. But the executable is not self-contained so cannot be easily shipped, the plugin binary has to be shipped too, and put on the dynamic linker search path.

12.4.2 Static Prelinking Plugins

There is a solution! When Felix tries to load a plugin by its text name, it first looks in a special data structure called the *prelink respository* to see if it is pre-loaded. If not, it tries an actual OS level library load. If we want to avoid this, we can statically link the plugin with the program, and then load the appropriate information into the pre-link repository.

To do this, we have to first create an object file for the plugin, instead of a DLL.

```
flx --static -c -od . --nolink ob_implementation.flx
```

This puts the object file into the current directory. It will be called *ob_implementation_static.o* on MacOSX or Linux, or *ob_implementation_static.obj* on Windows.

Now we have to compile our program as an object file too:

```
flx --static -c -od . --nolink ob_client.flx
```

Now we need a special file called a *static loader thunk*:

```
// ob.flx
open Dynlink;

static-link-plugin ob_implementation;

static-link-symbol ob_client_create_thread_frame in plugin ob_client;
static-link-symbol ob_client_flx_start in plugin ob_client;

val linstance =  Dynlink::prepare_lib("ob_client");
C_hack::ignore(linstance);
```

This is actually our mainline now! We need to open the class *Dynlink* to find the functions used for storing stuff in the registry.

Next, we specify the name of our plugin.

We also need to specify the names of two symbols used to run our program. *ob_client_create_thread_frame* allocates a global storage object. *ob_client_flx_start* runs the initialisation procedure for that object: this is actually what you previously, and incorrectly, thought of as your program.

What?? Yes, that's right. Felix doesn't do programs, only libraries. What you thought was your program is actually the side-effects of the initialisation procedure for a library.

Finally, we create an instance of the library *ob_client* with Dylink's function *prepare_lib*. This creates the thread frame object and initialises it (yep, that's your "program" running).

Since that's all we want to do we just ignore the library handle and we're finished.

When the *ob_client* code runs, it tries to load the plugin *ob_implementation*. But it finds it in the registry, along with the stanard symbols a plugin has. The *static-link-plugin* statement generates code that updates the repository.

Here's how you link the program:

```
flx --static -c -od . \
  ob_implementation_static.o \
  ob_client_static.o \
  ob.flx
```

Notice that unfortunately you have to give the platform dependent name of the object files. Notice also Felix adds the suffix *_static* to object files compiled for static linkage. Object files compiled for dynamic linkage get the suffix *_dynamic* instead. On some platforms these are the same, but not Linux. Dynamic link objects are compiled with

-fPIC for position independent code. Static link files are not. For the x86_64 processor, leaving out -fPIC generates much faster function calls.

The final program can be run like:

```
./ob
```

12.5 Web Server Plugins

The Felix webserver consists of a program *dflx_web* which is loads several plugins. There is a version *flx_web* which consists of exactly the same program, but with the plugins pre-linked. Here is the prelink code:

```
class WebserverPluginSymbols
{

  // We have to do this dummy requirements because static
  // linking removes
  requires package "re2";
  requires package "faio";
  requires package "flx_arun";

  open Dynlink;

  // Now add all the symbols.
  proc addsymbols ()
  {
    static-link-plugin
      fdoc2html,
      flx2html,
      fpc2html,
      py2html,
      ocaml2html,
      cpp2html,
      fdoc_scanner,
      fdoc_slideshow,
      fdoc_heading,
      fdoc_fileseq,
      fdoc_paragraph,
      fdoc_button,
      fdoc_frame,
      fdoc_edit,
      toc_menu
    ;
    // webserver
    static-link-symbol dflx_web_create_thread_frame in plugin dflx_web;
    static-link-symbol dflx_web_flx_start in plugin dflx_web;

  }
}

// Add the symbols
WebserverPluginSymbols::addsymbols;

// Now invoke the webserver!
println$ "Running webserver";
val linstance =  Dynlink::prepare_lib("dflx_web");
```

(continues on next page)

(continued from previous page)

```
println$ "Webserver prepared";
var init: cont = Dynlink::get_init linstance;

Fibres::chain init;
```

In this case, the program runs continuously after initialisation, so we get the mainline continuation *init* from the plugin instance *linstance* and chain to it.

12.6 Flx Plugins

The Felix command line tool consists of a program *dflx* which is loads several plugins, primarily the compiler drivers.

There is a version *flx* which consists of exactly the same program, but with the plugins pre-linked. Here is the prelink code:

```
class FlxPluginSymbols
{

  // We have to do this dummy requirements because static
  // linking removes
  requires package "re2";
  requires package "faio";
  requires package "flx_arun";

  open Dynlink;

  // Now add all the symbols.
  proc addsymbols ()
  {
    static-link-plugin
      toolchain_clang_macosx,
      toolchain_iphoneos,
      toolchain_iphonesimulator,
      toolchain_clang_linux,
      toolchain_gcc_macosx,
      toolchain_gcc_linux,
      toolchain_msvc_win
    ;
    // flx
    static-link-symbol dflx_create_thread_frame in plugin dflx;
    static-link-symbol dflx_flx_start in plugin dflx;

  }
}

// Add the symbols
FlxPluginSymbols::addsymbols;

// Now invoke the program!
val linstance =  Dynlink::prepare_lib("dflx");
var init: cont = Dynlink::get_init linstance;

Fibres::chain init;
```

(continues on next page)

(continued from previous page)

```
In this case, the program runs after initialisation,
```

so we get the mainline continuation *init* from the plugin instance *linstance* and chain to it.

CHAPTER 13

Felix 110: Coroutines

Coroutines and Chips

13.1 Coroutines

Coroutines are the fundamental building block of Felix. Your mainline program is actually a coroutine!

```
// schannels for communication
var inp, out = mk_ioschannel_pair[int]();

// writer fibre
spawn_fthread {
  for i in 1..10 do
    println$ "Sending " + i.str;
    write (out,i);
  done
};

// reader fibre
spawn_fthread {
  while true do
    var x = read inp;
    println$ "Read " + x.str;
  done
};
```

In this simple example, the *mk_ioschannel_pair* function is used to construct a *synchronous channel* object, and returns two references to it, *inp* which is typed to allow reading, and *out* which is typed to allow writing.

Then we use the *spawn_fthread* to construct fibres from argument coroutines. A coroutine in Felix is just a procedure. No wait! Actually .. a procedure is just a coroutine!

A single pre-emptive *thread* of control can be woven from many strands called *fibres*. In Felix the kind of fibres we make are called *fthreads*, whereas the pre-emptive kind are called *pthreads*.

Our writer coroutine is a loop which writes 10 integers down the channel. Our reader coroutine is an infinite loop, which reads integers until there are none to read. We use debugging prints to witness the activity.

Fibres do not run concurrently or asynchronously! Fibration is a non-deterministic method for interleaving control and is entirely synchronous. In particular *schannels* do not provide any buffering, their primary function is to mediate control interleaving.

Suppose our writer starts first. When it reaches the write statement it is suspended. Then our reader starts up and runs until it reaches the read operation. Then it also suspends.

Now, the scheduler copies the integer from the reader to the writer, places both on the list of active fibres, and then picks a fibre to run, which it does by unfreezing or *resuming* the suspension.

13.2 Hgher Order Coroutines

Our previous example would not be consider good practice because the coroutines are not reusable components. Programming with coroutines is all about modularity. Lets see a better way:

```
// writer fibre
proc writer (out: %>int) ()
{
  for i in 1..10 do
    println$ "Sending " + i.str;
    write (out,i);
  done
}

// reader fibre
proc reader (inp: %<int) ()
{
  while true do
    var x = read inp;
    println$ "Read " + x.str;
  done
}

proc start_network () {
  // schannels for communication
  var inp, out = mk_ioschannel_pair[int]();
  var co_writer = writer inp;
  var co_reader = reader  out;
  spawn_fibre co_writer;
  spawn_fibre co_reader;
}

start_network;
```

This program does the same thing as our first example, however we have modularised the reader and writer by making them named procedures, and, explicitly passing in the channels they should communicate with.

The type *%>int* is a shorthand for *oschannel[int]*, an output channel for ints, and *%<int* is a shorthande for *ischannel[int]*, an input channel for ints. Since *spawn_fibre* requires a procedure that takes the argument *()* we have to add that as well.

Another refactoring here is that we have modularised the network constructing code into a procedure as well. It creates the channels needed, binds the reader and writer procedures to the channels they need, then spawns the bound procedures to create our fibres.

Then it returns. This is important! When it returns, it *forgets* the names temporarily given to the channels. This means the mainline, following the call to *start_network*, cannot reach the channels. Although the channels are named in the constructor, they're forgotten afterwards, except by the coroutines that use them. We also forget the names temporarily given to the bound coroutines.

Forgetting what you do not need to know is an absolute imperative when using coroutines because reachability is what is used to drive termination. I hope you noticed, that even though the reader has an infinite loop, the program still terminates!

13.3 Coroutine Termination

13.3.1 Run procedure

In order to better understand coroutines, we will examine the semantics of a fundamental procedure *run*.

```
run start_network;
```

The *run* procedure is an ordinary procedure that returns when it is finished. It creates a new *coroutine scheduler* object, and then spawns its argument on that scheduler. It then runs all the fibres on the scheduler until there are none left, and returns.

13.3.2 Fibre States

Every fibre is in one of four states. It is either *running*, *active*, *waiting*, or *dead*.

At any time, for any scheduler, there is only ever one running fibre. If there are no running fibres, the scheduler terminates. Every pthread that is spawned automatically runs a fibre scheduler, including the mainline thread!

A fibre is *active* if it is suspended but ready to run. The scheduler keeps a list of active fibres. When the currently running fibre suspends, the scheduler just picks another fibre off the active list and runs it. If there are no fibres on the active list, the scheduler procedure returns.

A fibre which is waiting will be either *waiting to read* or *waiting to write*. The waits always occur on a specific schannel. Every schannel is therefore either *empty*, contains a list of fibres waiting to read, or contains a list of fibres waiting to write.

If a write operation is performed on a channel which is empty, the writing fibre is added to the channel wait list. If the channel already contains fibres waiting to write, the writing fibre is also added to the channel wait list.

But if the channel contains a list of fibres waiting to read, then one of the fibres is taken off the channel wait list instead, the object being written is transfered from the writer to the reader, and both the fibres are added to the schedulers active list. Since the writer was running, and is now merely active, the scheduler picks another fibre to run.

Read operations work exactly the same way, swapping the meaning or read and write about.

13.3.3 Reachability

If a fibre is waiting to read, but no other fibre ever writes to the channel it is waiting on, the fibres is said to be *starved*. If a fibre is waiting to write, but no other fibre ever reads from the channel it is waiting on, the fibre is said to be *blocked*.

With pthreads, this lockup is fatal and invariably a design fault. Not so with fibres! Recall our reader:

```
// reader fibre
proc reader (inp: %<int) ()
{
  while true do
    var x = read inp;
    println$ "Read " + x.str;
  done
}
```

This is an infinite loop but our write only wrote 10 integers. So after reading 10 integers our fibre starves!

Now here is the trick! Only the reader and writer fibres can reach the channel that connects them. No one else could write on that channel, because no one else knows its name. What is more, the writer has returned so is now dead, so it cannot reach the channel either, because it doesn't exist!

So since the reader is not running or active, even the scheduler doesn't know about it. Only the channel knows about it, and only the reader knows about the channel.

So the scheduler now has no running or active procedures and it returns. The starving reader and the channel are unreachable and forgotten. So the garbage collector simply deletes them.

Unlike pthreads, locking up is potentially a *correct* way to terminate. The motto is that fibres cannot dead-lock because if they do they're dead and if they're dead the don't exist so they cannot be dead-locked.

However, coroutines can *livelock*. A livelock occurs when there is a fibre which can do write on a channel which would relieve starvation of another fibre, but choses not to, or, a fibre which can read from a channel which would relieve a blockage, but chooses not too. In other works the channel is *statically reachable* but is *dynamically ignored*.

The key to correct design with coroutines is simple: if a routine is not going to perform I/O on a channel it must forget it. In other words the channel must go out of scope.

Our *start_network* procedure obeys this rule. It never reads or writes from the channel it constructs but after binding the channel to the reader and write procedures and spawning the bindings as fibres, it returns, thereby its knowledge of the channel is forgotten, in particular because it doesn't exist after returning!

13.4 Pipelines

We are now going to change our way of modelling coroutines again, by using records are arguments instead of tuples.

```
// writer fibre
proc writer (io: (out: %>int)) ()
{
  for i in 1..10 do
    println$ "Sending " + i.str;
    write (io.out,i);
  done
}

// reader fibre
proc reader (io: (inp: %<int)) ()
{
  while true do
    var x = read inp;
    println$ "Read " + x.str;
  done
}
```

(continues on next page)

(continued from previous page)

```
proc doubler (io: (inp: %<int, out: %>int)) ()
{
  while true do
    var x = read io.inp;
    write (io.out, 2 * x);
  done
}

proc start_network () {
  // schannels for communication
  var inp1, out1 = mk_ioschannel_pair[int]();
  var inp2, out2 = mk_ioschannel_pair[int]();
  spawn_fibre$ writer (out=out1);
  spawn_fibre$ doubler (inp=inp1,out=out2);
  spawn_fibre reader (inp=inp2);
}

start_network;
```

Here the coroutines use a single record as a parameter, each field of the record is a channel. This has the advantage that it's easier to pass the arguments correctly, since they're named, rather than having to be put in a specific order.

Using this protocol, we can also rewrite *start_network*:

```
proc start_network () {
   run$ writer |-> doubler |-> reader;
}
```

A writer is also called a *source*. A reader is also called a *sink*. A loop which reads, does something with the value, then writes, is called a *transducer*.

If you have a source connected to a series of transducers and ending with a sink, that is called a *closed pipeline*. The left associative infix pipe operator |-> can be used to connect coroutines into a pipeline. The result of the composition is itself a coroutine.

Not all coroutine networks are pipelines, however they are very common both as whole networks, and also as sub-parts of more complex networks.

Pipelines are dual to monads in functional programming .. but they're much easier to use and a lot more efficient.

13.5 Chips

Well now we're going to change notations again! Here we go:

```
// writer fibre
chip writer
  connector io
    pin out: %>int
{
  for i in 1..10 do
    println$ "Sending " + i.str;
    write (io.out,i);
  done
}
```

(continues on next page)

(continued from previous page)

```
// reader fibre
chip reader
  connector io
    pin inp: %<int
{
  while true do
    var x = read inp;
    println$ "Read " + x.str;
  done
}

chip doubler
  connector io
    pin inp: %<int
    pin out: %>int
{
  while true do
    var x = read io.inp;
    write (io.out, 2 * x);
  done
}

run$ writer |-> doubler |-> reader;
```

The chips are identical to our previous reader, writer and doubler procedures, we just have some syntactic sugar to make them look like integrated circuit specifications.

A chip can have more than one connector, the connector name is just the parameter name. Each connector can have several pins, the pin names are just the record field names. This syntax is a bit more verbose but it is easier to read and makes the purpose and intended use of the procedures clear.

13.6 Standard Chips

The library contains a lot of standard chips.

13.6.1 Write block

Blocks a writer, deliberately. When a writer writes to a channel connected to this device, nothing happens, so the write blocks.

```
chip writeblock[T]
  connector io
    pin inp : %<T
{
}
```

13.6.2 Read Block

Starves a reader, deliberately. When a reader tries to read from a channel connected to this device, nothing happens, so the reader starves.

```
chip readblock[T]
  connector io
    pin out: %>T
{
}
```

13.6.3 Universal sink

Reads input forever, but ignores it.

```
chip sink[T]
  connector io
    pin inp : %<T
{
  while true do
    var x = read (io.inp);
    C_hack::ignore (x);
  done
}
```

13.6.4 Constant Source

Writes the fixed value *a* forever.

```
chip source[T] (a:T)
  connector io
    pin out: %>T
{
  while true do
    write (io.out, a);
  done
}
```

13.6.5 One shot source

Writes the fixed value *a* once then exits.

```
chip value[T] (a:T)
  connector io
    pin out: %>T
{
  write (io.out, a);
}
```

13.6.6 Source from generator

Calls a generator to obtain a value, then writes it, repeatedly.

```
chip generator[T] (g: 1->T)
  connector io
```

(continues on next page)

(continued from previous page)

```
    pin out: %>T
  {
    repeat perform write (io.out, g());
  }
```

13.6.7 Source from iterator

This chip reads values from an iterator and streams them to output until the iterator returns None. It is a hand optimised version of the less efficient *for v in x perform write(io.out,v);*.

```
chip iterate[T] (g: 1->opt[T])
  connector io
    pin out: %>T
  {
    again:>
      var x = g();
      match x with
      | Some v =>
        write (io.out, v);
        goto again;
      | None => ;
      endmatch;
  }
```

13.6.8 Source from list

A list iterator specialised to lists. It returns when all the values in the list have been written out.

```
chip source_from_list[T] (a:list[T])
  connector io
    pin out: %>T
{
  for y in a perform write (io.out,y);
}
```

13.6.9 Bound Source from list

This routine generates an option stream from the list *a*. Each value is written as *Some v* until the list exhausted, then an infinite stream of *None[T]* is written.

This is a bound stream because the *logical* content of the stream is terminated by None; that is, the option type is used as a carrier type.

```
chip bound_source_from_list[T] (a:list[T])
  connector io
    pin out: %>opt[T]
{
  for y in a perform write (io.out,Some y);
  while true perform write (io.out,None[T]);
}
```

13.6.10 Function adaptor

One of the most useful chips, the function adaptor reads a stream of values, applying the function *f* to each value and writing them out as it goes. It is the dual to functional programming *map* over lists.

```
chip function[D,C] (f:D->C)
  connector io
    pin inp: %<D
    pin out: %>C
{
  while true do
    var x = read io.inp;
    var y = f x;
    write (io.out, y);
  done
}
```

13.6.11 Procedure adaptor

Converts a procedure to a sink. Reads values and calls the procedure *p* on each one.

```
chip procedure[D] (p:D->0)
  connector io
    pin inp: %<D
{
  while true do
    var x = read io.inp;
    p x;
  done
}
```

13.6.12 Filter

Reads values and writes out that that satisfy the predicate *f*. Dual to functional programming filter over lists.

```
chip filter[D,C] (f:D->opt[C])
  connector io
    pin inp: %<D
    pin out: %>C
{
  while true do
    var x = read io.inp;
    match f x with
    | Some y => write (io.out, y);
    | None => ;
    endmatch;
  done
}
```

13.6.13 Filter and map

Applies the predicate *c*, and writes the application of *f* to values satisfying it out.

```
chip filter[D,C] (c:D->bool) (f:D->C)
  connector io
    pin inp: %<D
    pin out: %>C
{
  while true do
    var x = read io.inp;
    if c x do
       write (io.out, f x);
    done
  done
}
```

13.6.14 Sink to List

This chip is a sink that reads values and pushes them onto an extenal list identified by a pointer. The list must be initialised before the coroutine is spawned.

```
chip sink_to_list[T] (p: &list[T])
  connector io
    pin inp : %<T
{
  while true do
    var x = read (io.inp);
    p <- Cons (x,*p);
  done
}
```

13.6.15 Sink to unique list

Same as *sink_to_list* except the value is only pushed onto the list if it is not already present.

```
chip sink_to_unique_list[T with Eq[T]] (p: &list[T])
  connector io
    pin inp : %<T
{
  while true do
    var x = read (io.inp);
    if not (x in *p) perform
      p <- Cons (x,*p)
    ;
  done
}
```

13.6.16 Buffer

Perhaps the most important and useful chip, it simply copies its input to its output.

```
chip buffer [T]
  connector io
    pin inp: %<T
    pin out: %>T
```

(continues on next page)

(continued from previous page)

```
{
  while true do
    var x = read io.inp;
    write (io.out, x);
  done
}
```

CHAPTER 14

Felix 111: Generalised Algebraic Data types

Variants with generalised type indices.

14.1 Generalised Algebraic Datatypes

A Generalised Algebraic Datatype, or GADT, is an extension of the basic variant concept:

```
variant pair[T] =
| PUnit of unit => pair[unit]
| PInt[T] of int * pair[T] => pair[int * pair[T]]
| PFloat[T] of float * pair[T] => pair[float * pair[T]]
| PString[T] of string * pair[T] => pair[string * pair[T]]
;
```

This looks like an ordinary variant except there is an extra term on the RHS which is always the variant with some subscript.

With an ordinary variant of one type variable *T* the RHS constructor is always the variant type, in this case *pair* with its universal quantifiers, in this case type variable *T*.

With a GADT the subscript can be an arbitrary type function of the type variables instead of just *T*.

Given the above GADT here are some values:

```
var x1 : pair[unit] = #PUnit[unit];
var x2 : pair[int * pair[unit]] = PInt (1,x1);
var x3 = PFloat (42.33f, x2);
```

To use a GADT you need to write a generic function:

```
fun show [W:GENERIC] (x:W):string=
{
 match x with
 | PUnit => return "PUnit";
```

(continues on next page)

(continued from previous page)

```
 | PInt (head, tail) => return "PInt(" + head.str+", " + tail.show+ ")";
 | PString (head, tail) => return "PString(" + head+", " + tail.show+ ")";
 | PFloat (head, tail) => return "PFloat(" + head.str+", " + tail.show+ ")";
 endmatch;
}

println$ "x3=" + x3.show;
```

The reason for the generic function is that it provide static polymorphic recursion.

14.1.1 GADTs with existentials

A GAD constructor can introduce an extra type variable called an existential variable:

```
variant pair[T] =
| PUnit of unit => pair[unit]
| Pair[T,U] of U * pair[T] => pair[U * pair[T]]
;

var x1 = #PUnit[unit];
var x2 = Pair (22,x1);
var x3 = Pair (99.76,x2);

fun f[T:GENERIC] (x:T) = {
  match x with
  | Pair (a,b) => return a.str + ","+b.f;
  | PUnit => return "UNIT";
  endmatch;
}

println$ f x3;
```

The advantage of this *pair* over the previous one is that it works for *any* type U, not just int, string or float. This GADT is actually defining a tuple recursively.

The function which analyses the GADT must be generic since the decoder requires polymorphic recursion. Note in particular the type of *b* on which *f* is called is not the same as *x*.

14.1.2 Another Example

This example is from Wikipedia:

```
variant Expr[T] =
  | EBool of bool => Expr[bool]
  | EInt of int => Expr[int]
  | EEqual of Expr[int] * Expr[int] => Expr[bool]
;

fun eval(e) => match e with
  | EBool a => a
  | EInt a => a
  | EEqual (a,b) => eval a == eval b
  endmatch
;
```

(continues on next page)

(continued from previous page)

```
var expr1 = EEqual (EInt 2, EInt 3);
println$ eval expr1;
```

In this example we have boolean and integer values and an equality operator. The important thing is that equality only works on integers and returns a bool: without GADTs there is no type safe way to express this constraint.

CHAPTER 15

Felix 112: Compact Linear Types

15.1 Compact Linear Types

A compact linear type either the unit type 1, the void type 0, or any small sum or product of compact linear types. Small here means the representation as described below fits into a 64 bit machine word.

Compact linear types have two primary uses: for sub word level value manipulation and to support polyadic array operations.

15.1.1 Unitsums

The following types are *unitsum* types:

```
typedef void = 0;
typedef unit = 1;
typedef bool = 2; // 1 + 1
4 // 1 + 1 + 1 + 1
5 // 1 + 1 + 1 + 1 + 1
...
```

These are sums of 0, 1, 2, 3, 4, 5 .. units and represent 0, 1, 2, 3, 4, 5 alternatives. Values of unitsum types have two notations:

```
case 3 of 5
`3:5
```

The first syntax is general, the second is peculiar to unitsums. Note that there are no values of type void, because there are no alternatives. The value of type unit can also be written:

```
()
```

and represents exactly one alternative. Values of type bool can also be written

```
false // `0:2
true // `1:2
```

Note that somewhat unfortunately, the value index is zero origin so *case 0 of 5* is the first of 5 cases and *case 4 of 5* is the last. It reads badly but zero origin was chosen for symmetry with C array indexing conventions.

All unitsums from 1 up are represented by 64 bit unsigned integers. [This may be relaxed and/or extended in future versions of Felix]

15.1.2 Products

Products of compact linear types are compact linear types. For example:

```
var x : 2 ^ 4 = true, false, false, true;
println$ x.1; // false
```

This is an array of 4 bits, but it has a magic property: it is compact linear so it is represented by a single machine word. Arrays of up to 64 bits are represented by single machine words. Here is another example:

```
var x : 2 * (3 * 4) = true, (`1:3, `2:4);
println$ x.1.1._strr; // case 1 of 3
```

Again, the value is a single machine word. Compact linear types are similar to C bit fields, however a C bitfield must consist of n bits and so represents 2^n values. Felix compact linear types have no such constraint.

Compact linear types are represented by standard variable radix number system. The easiest explanation is the following example:

```
var time = (`2:24, `30:60, `1:60); // 1 second past 2:30 am
var secs = time :>> int;
println$ "Seconds into day = " + secs.str;
assert secs == 60 * 60 * 2 + 60 * 30 + 1;
```

Compact linear types are named that because if there are n possible value they're represented by a range of integers from 0 upto $n-1$. This range is compact, meaning there are no holes in it, and linear, because it is integral.

CHAPTER 16

Felix 113: Serialisation

16.1 Serialisation

TBD

CHAPTER 17

Felix 114: Kinds

Felix has a basic explicit kinding system used primarily to express constraints.

17.1 The Kinding System

TBD

CHAPTER 18

Felix GUI

18.1 Getting Started With the Felix GUI

Felix comes with a library to create platform independent graphical user interfaces. It uses the Simple Direct Media Layer, version 2, SDL2 system with add-ons to do this.

SDL2 runs on Linux, OSX, Windows, iOS and Android and is designed for implementation of portable games.

18.1.1 Installation

For the Felix GUI to work, *development versions* of the following components must be installed:

```
SDL2
SDL2_image
SDL2_ttf
SDL2_gfx
```

On Linux using apt package manager do this:

```
sudo apt-get install libsdl2-dev
sudo apt-get install libsdl2_image-dev
sudo apt-get install libsdl2_ttf-dev
sudo apt-get install libsdl2_gfx-dev
```

Felix already contains database entries for these packages, but at the time of writing this tutorial, the libraries are expected to be in */usr/local* which is where you would put them if you built them yourself.

However Debian filesystem layout standards used by Linux OS such as Ubuntu that use *apt* package manager put components in */usr/* instead. So unfortunately you will have to modify the database by hand by editing these files

```
build/release/host/config/sdl2.fpc
build/release/host/config/sdl2_image.fpc
build/release/host/config/sdl2_ttf.fpc
```

replacing */usr/local* with just */usr*. To preserved these modifications across upgrades, you should also copy the files:

```
cp build/release/host/config/sdl2.fpc $HOME/.felix/config
cp build/release/host/config/sdl2_image.fpc $HOME/.felix/config
cp build/release/host/config/sdl2_ttf.fpc $HOME/.felix/config
```

I hope this problem can be reduced in future versions, but it is likely to be an issue for some time because most developers will have a libraries installed in both places, depending on whether they're using a package manager to install them, or building from source.

To check your installation works, do this:

```
make tutopt-check
```

and a series of tests will run which use the Felix GUI and SDL2.

18.2 GUI Basics

We will now show how to do basic GUI programming. The first thing we want to do is open a window!

```
include "gui/__init__";
open FlxGui;

println$ "Basic Window Test";
FlxGui::init();

var w = create_resizable_window("Felix:gui_01_window_01",100,100,400,400);
w.add$ mk_drawable FlxGuiSurface::clear lightgrey;
w.update();
w.show();

sleep(15.0);
```

The Felix gui is not included by default, so we have to first include the library with

```
include "gui/__init__";
```

Although this makes the library available, we would have to prefix the names of all functions in the library with `FlxGui::` to use them. Since programmers hate typing stuff, we will open the library, so the functions are all in the current scope and can be used without the prefix.

```
open FlxGui;
```

Next, we will print a diganostic to standard output so we know which program is running, and then initialise library. Initialisation is a requirement imposed by SDL, which has a lot of work to do on some platforms to connect to the GUI devices such as the screen, touch (haptic) inputs, joysticks, mice, keyboards, audio, and other multi-media hardware.

```
println$ "Basic Window Test";
FlxGui::init();
```

Now it it time for the fun! We will create a resizable window:

```
var w = create_resizable_window("Felix:gui_01_window_01",100,100,400,400);
```

The first parameter is the title of the window, which should appear on the titlebar (it doesn't on OSX! Because there is no pause to accept input).

The next four parameters describe the window geometry. The first two are the x and y coordinates. The SDL coordinate system puts the origin in the top left corner, and x increases down the screen.

The unit of measurement is approximately the pixel. I say approximately because on a Mac with a Retina display, each pixel is often four display elements on the screen. To confuse the issue, the Mac can do hardware scaling. You'll just have to experiment!

The second two values are the width and height of the window's client area, this does not include the title bar. However the x and y coordinates are the top left corner of the whole window including the title bar!

What we have created is a data structure representing the window. The next thing we want to do is put some coloured pixels in it.

```
w.add$ mk_drawable FlxGuiSurface::clear lightgrey;
```

This is an example of a fundamental operation, to add to a windows display surface, the commands for drawing something.

The `w.add` method adds a *drawable* to a list of drawables kept for window `w`.

The `mk_drawable` method is a function which constructs a drawable object. Its first argument is the actual drawing command, `FlxGuiSurface::clear` which clears a whole surface. The second argument is an argument to that command, in this case `lightgrey`, which tells the clearing command what colour to write on the surface it is clearing.

We have not actually drawn on the window at this point. What we have done is packaged up the drawing instructions, and *scheduled* them for drawing later.

To actually draw, we do this:

```
w.update();
```

Now we have drawn the objects we scheduled to be drawn on the systems internal representation of the window's surface but still, nothing appears on the screen!

This is because the window has not been shown yet. We've been drawing on it whilst it was invisible. So we now make it visible:

```
w.show();
```

FInally, we want the window to hang around for 15 seconds so you can admire your fine art work.

```
sleep(15.0);
```

This causes the program to sleep for 15 seconds. The argument is a double precision floating point number representing a delay in seconds. The decimal point is mandatory and trailing zero is mandatory!

18.3 Putting Stuff in the Window

Now we have created a window, we want to put stuff on it!

Here's how:

```
include "gui/__init__";
open FlxGui;

println$ "Basic Drawing Test";
FlxGui::init();
```

(continues on next page)

(continued from previous page)

```
var w = create_resizable_window("Felix:gui_03_draw_01",100,100,400,600);
w.add$ mk_drawable FlxGui::clear lightgrey;

var font_name = dflt_sans_serif_font();
var font : font_t = get_font(font_name, 12);
var lineskip = get_lineskip font;
w.add$ mk_drawable FlxGui::write (10,10,font,black,"Basic Drawing Test");

fun / (x:string, y:string) => Filename::join (x,y);
var imgfile = #Config::std_config.FLX_SHARE_DIR / "src" / "web" / "images" /
↪"FelixWork.jpg";

var ppic : surface_t = surface (IMG_Load imgfile.cstr);

w.add$ mk_drawable blit (20,20, ppic.get_sdl_surface ());

w.add$ mk_drawable draw_line (RGB(0,0,255), 100,110,200,110);
w.add$ mk_drawable draw_line (RGB(0,0,255), 100,210,200,210);
w.add$ mk_drawable draw_line (RGB(0,0,255), 100,110,100,210);
w.add$ mk_drawable draw_line (RGB(0,0,255), 200,110,200,210);

w.update();
w.show();

Faio::sleep(15.0);
```

Here, the program is as before, except we now do three basic ways to draw on a window.

18.3.1 Text

First, we want to be able to put ordinary text on the window. To do that, we have to first create a font to write the text with:

```
var font_name = dflt_sans_serif_font();
var font : font_t = get_font(font_name, 12);
var lineskip = get_lineskip font;
```

The first line gets the name of a default sans serif font file which is packaged with Felix so you don't have to figure out where all the fonts on your system are.

The second line actually creates the font from the file at a particular size, in this case 12 point. The size is a conventional printers measure. You'll just have to get used to what it means!

The third line helps tell how big the font it. We retrieve from the font the distance in pixels we should allow between lines, for readable text. Anything less and the characters would bump into each other.

Now we have a font, we schedule drawing some text on the window:

```
w.add$ mk_drawable FlxGui::write (10,10,font,black,"Basic Drawing Test");
```

This is our usual machinery for adding a drawable object to the windows list of drawables, to be drawn when we say to *update*. The drawing function is `FlxGui::write`. Notice we used the fully qualified name of the function, to avoid confusiuon with other functions named *write*.

The argument to write is the x and y coordinates of the initial base point, the font to use, the colour to write in, and the actual text to write.

Text is always written with respect to a base point. The base point is origin of the first character which is approximately the left edge of the character, and the point at which an invisible underline would occur: in other words, under the main body of the character, but on top of any descender that letter like *g* may have.

The exact location is font dependent. Font rendering is an arcane art, not an exact science so you will have to practice to get text to appear where you it has the correct visual significance.

18.3.2 Picture

Now we are going to put a picture in the window. The image is a JPEG image, and is supplied for testing purposed in Felix at a known location.

First we define a little helper function:

```
fun / (x:string, y:string) => Filename::join (x,y);
```

What this says is that when we try to divide one string by another string, we actually mean to join the strings together using `Filename::join` which is a standard function which sticks a `/` character between strings on unix platforms, and a slosh on Windows.

The file is here:

```
var imgfile = #Config::std_config.FLX_SHARE_DIR / "src" / "web" / "images" /
↪"FelixWork.jpg";
```

The prefix of this code finds the share subdirectory of the Felix installation, which contains the picture we went in the images subdirectory of the web subdirectory of the src subdirectory.

Now to schedule the drawing we do this:

```
var ppic : surface_t = surface (IMG_Load imgfile.cstr);
w.add$ mk_drawable blit (20,20, ppic.get_sdl_surface ());
```

The first line loads the image file into memoy using a low level primitive from SDL2_image. That primitve requires a C char pointer, not a C++ string, which is what Felix uses, so we use `cstr` to convert. Then the *surface* function translates the loaded file into an Felix surface object.

In the second line we add the drawable to the window based on the `blit` function. This copies one surface to another. We copy the image surface to the window surface at position 20,20 in the window, and use the `get_sdl_surface()` method to translate the Felix surface object into a lower level SDL surface.

Its all a bit mysterious, so you just have to so some things by copying the patterns that work.

18.3.3 Lines

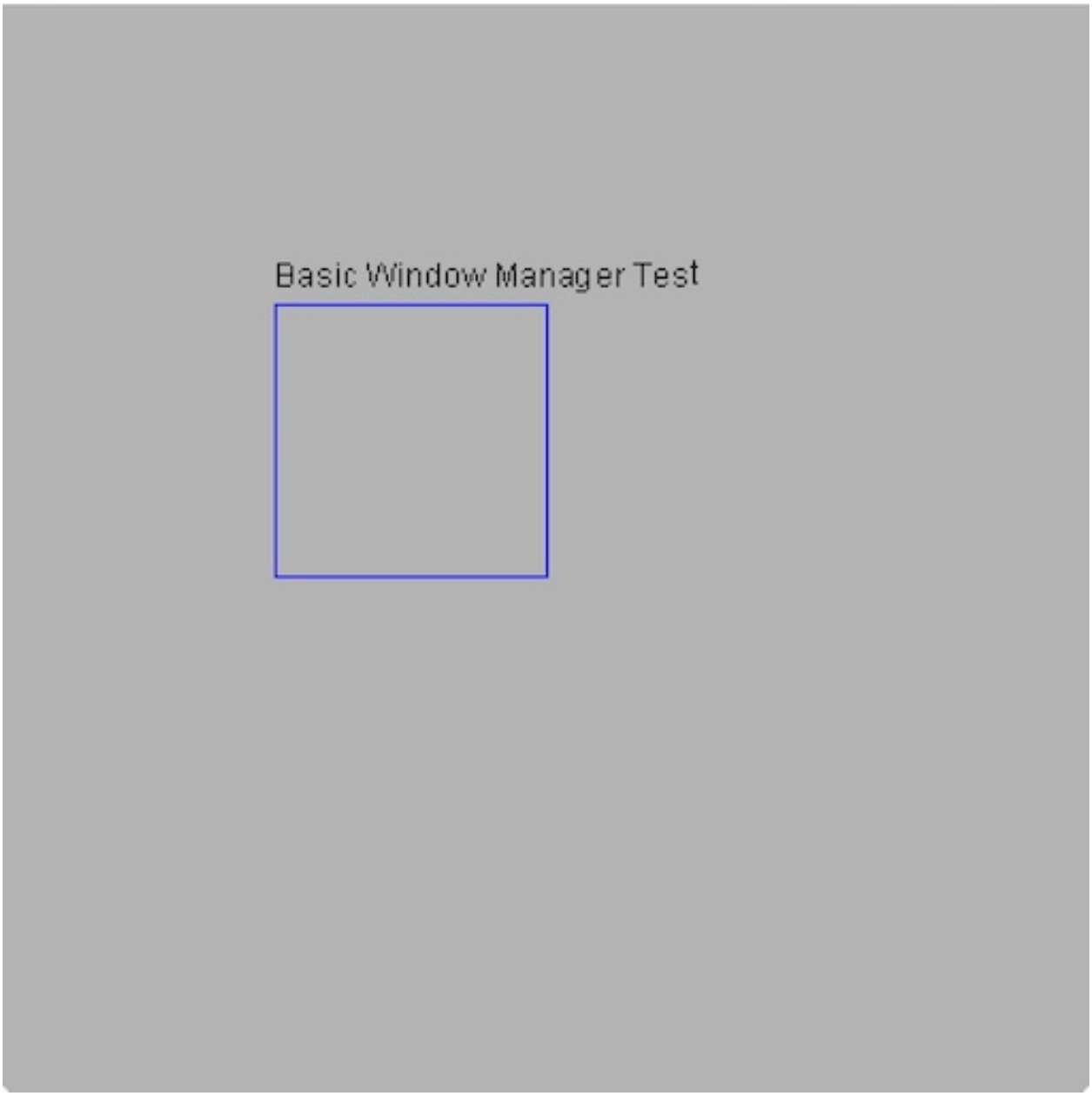
Finally, we draw a blue rectangle on top of the picture. I'm sure you can figure out how that works!

18.4 Using An Event Handler

So far we have just done some drawing. But now, we want to respond interactively to user input. To do this, we need to use an *event handler*.

18.4.1 The initial window

Lets start by making a window that looks like this:



which we do with this code as usual:

```
include "gui/__init__";
open FlxGui;
FlxGui::init();

var w = create_resizable_window("Felix:gui_04_wm_01",100,100,400,400);
w.add$ mk_drawable FlxGui::clear lightgrey;
```

(continues on next page)

(continued from previous page)

```
var font_name = dflt_sans_serif_font();
var font : font_t = get_font(font_name, 12);
var lineskip = get_lineskip font;

w.add$ mk_drawable FlxGui::write (100,100,font,black,"Basic Event Handler Test");
w.add$ mk_drawable draw_line (RGB(0,0,255), 100,110,200,110);
w.add$ mk_drawable draw_line (RGB(0,0,255), 100,210,200,210);
w.add$ mk_drawable draw_line (RGB(0,0,255), 100,110,100,210);
w.add$ mk_drawable draw_line (RGB(0,0,255), 200,110,200,210);

w.update();
w.show();
```

18.4.2 The event handler

Now, the next thing is we are going to make a *chip* which can handle events:

```
chip event_displayer (w:window_t)
  connector pins
    pin inevent : %<event_t
    pin quit: %>int
{
  while true do
   var e = read pins.inevent;
   var s =
      match e with
      | WINDOWEVENT we =>
        we.type.SDL_EventType.str + ": " +
           we.event.SDL_WindowEventID.str +
           " wid=" + we.windowID.str
      | MOUSEMOTION me =>
        me.type.SDL_EventType.str
      | _ => ""
    ;
    var linepos = 100 + 4 * lineskip;
    if s != "" do
      println$ s;
      var r = rect_t (100,linepos - 2*lineskip,300,4*lineskip);
      w.remove "evt";
      w.add$ mk_drawable "evt" 100u32 fill (r, green);
      w.add$ mk_drawable "evt" 100u32 FlxGui::write (100,linepos,font,black,"EVENT:
↪"+ s);
      w.update();
    done
  done
}
```

We are using a powerful new idiom: fibrated programming. What you see is a special kind of routine called a *coroutine*. Lets see what it does.

First, the interface tells us that it displays events on window *w*. Now our chip has a connector named *pins*, and on that connector, we have two pins named *inevent* and *quit*.

The pin *inevent* is an input pin for data of type *event_t* whilst the pin *quit* is an output pin for an int. I can tell the direction of the pin from the channel type: %< is for input, and %> is for output. The type of data the pin handles

comes next.

Now lets look at the code. We can see immediately this chip runs in an infinite loop. It starts of by reading an event from the *inevent* pin.

Next, we analyse the event, to see what it is, using a pattern match. There are two kinds of event we're interested in: a *WINDOWEVENT* and a *MOUSEMOTION*.

For now, the weird code for these events just converts some of the event information into a string *s* we can display on the window, lets not worry about exactly what it means (you'll see, when you try it!).

Now the next bit calculates the position inside the box we drew to display the string, then, if the event description *s* is not the null string, we print the string to standard output.

Now we calculate a bounding rectangle for the string. Its not very accurate!

Now comes the fun bit! The next thing we do is *remove* all the drawables from the window tagged with the string `"evt"`. Then we add two drawables, the first one fills our bounding rectangle with green, and the second writes some text. Then we update the window.

Now what is that magical *100u32* you ask? The answer is, this is the *z* coordinate of the drawing operation, which is a 32 bit unsigned integer. When Felix is drawing on a surface, it draws at the smallest *z* coordinate first. Then it draws at the next smallest, and so on. At any particular *z* coordinate, it draws in the order you addes the drawable to the list of drawables.

By default, drawing occurs at *z=0u32*. So why are we specifying a z coorinate? The answer is: the background of the window was drawn at *z=0*. It was not given a tag, so it has the default tag `""`. Importantly, we did not remove drawables with that tag, so the background drawable is still in the drawable list.

The thing is, we want to draw *on top* of the background, so we have to ensure we draw at a higher *z* coordinate.

18.4.3 The Mainline

Now, as promised, it is time to install our event handler:

```
begin
  var qin,qout = mk_ioschannel_pair[int]();
  device windisp = event_displayer w;
  circuit
    connect windisp.inevent, event_source.src
    wire qout to windisp.quit
  endcircuit

  C_hack::ignore(read qin);
  println$ "QUIT EVENT";
end
```

The *begin* and *end* here are important for reasons that will be explained later, for the moment it suffices to know you need to do this to ensure the schannels we create become inaccessible when you click to quit, so that the program actually terminates.

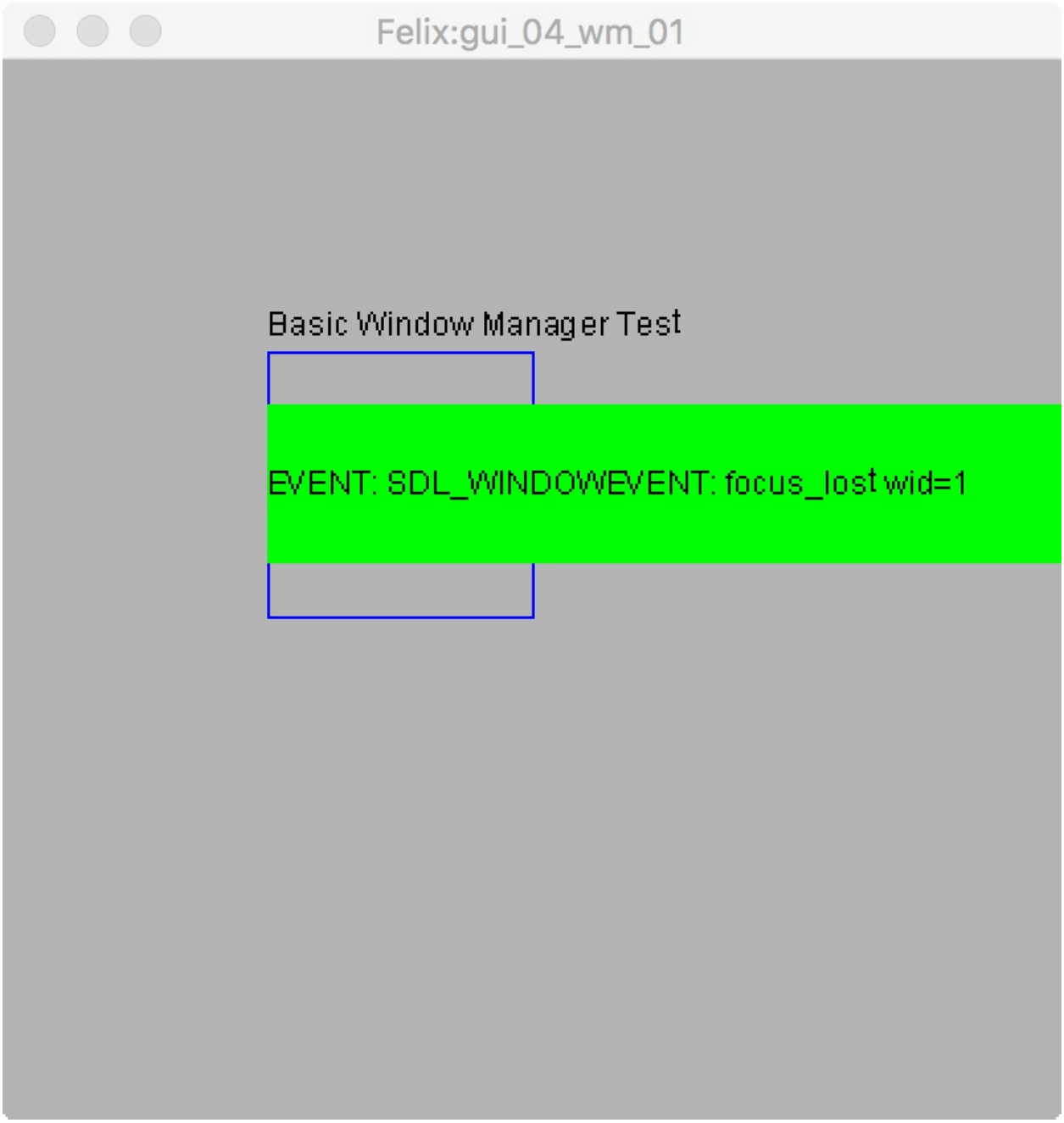
The first thing we do is make a single synchronous channel of type *int* which has two end points: *qin* and *qout*. The first one, *quin* is a read channel, and the second one, *qout* is a write channel.

Next, we make a *device* called *windisp* by applying our event handler function to the window it is to operate on.

Then we build a circuit by connecting the event handler to an event source, and wiring up the output end of the quit channel to the event handler as well. Our circuit begins running immediately.

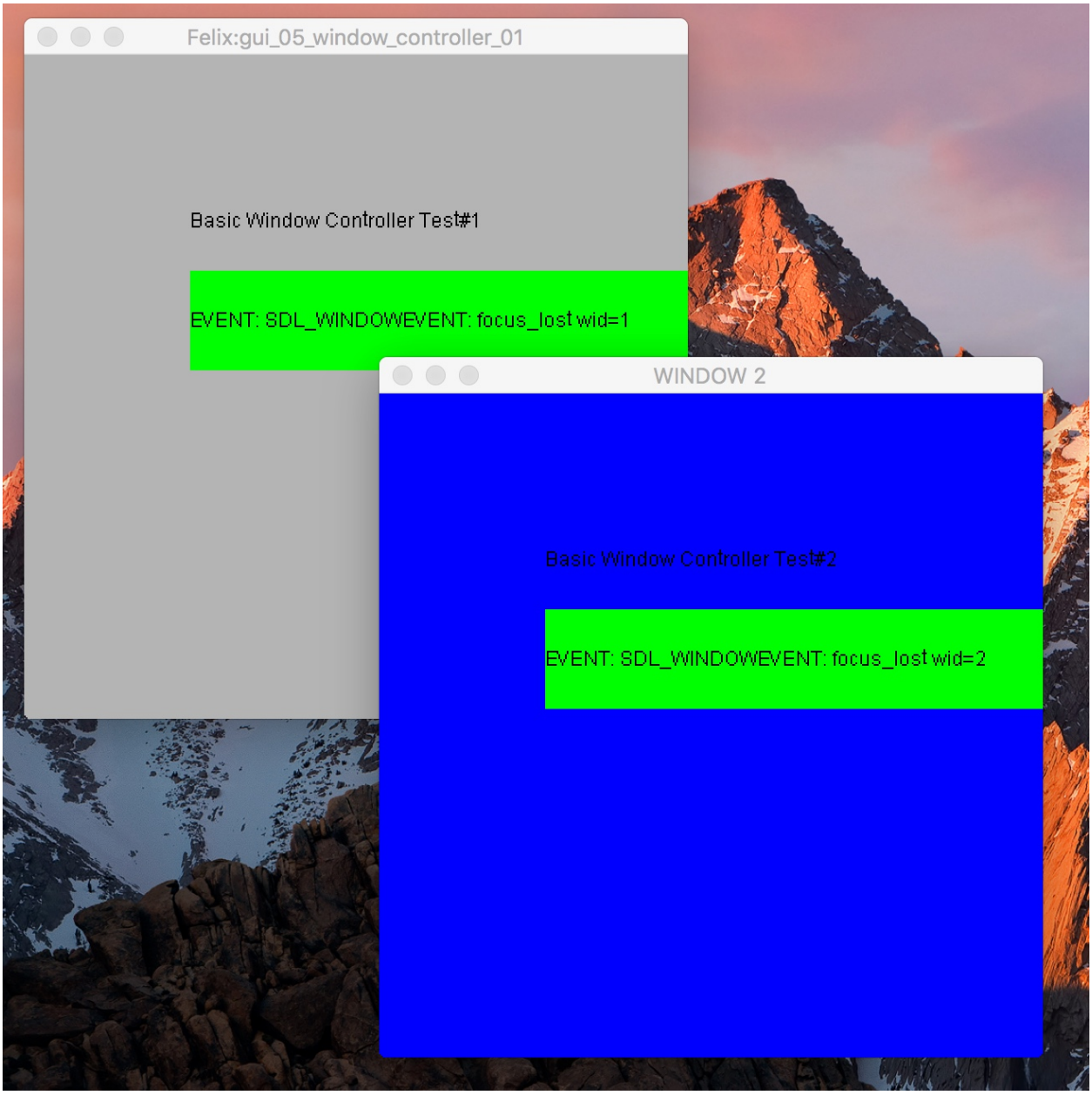
Now we wait until the user clicks to close the window, or presses the quit key. On a Mac, Apple-Q is the quit key. We use `C_hack::ignore` because we don't care what the quit reason is.

You should see something like this:



18.5 Using a Window Manager

A *window manager* is a component that automates distribution of message such as mouse clicks and key presses to one of several event handlers.



18.5.1 The initial windows

First our usual setup:

```
include "gui/__init__";
open FlxGui;

println$ "Basic Window Controller Test";
FlxGui::init();

var font_name = dflt_sans_serif_font();
var font : font_t = get_font(font_name, 12);
var lineskip = get_lineskip font;
```

Now, we make two similar windows, at different locations but different titles.

```
var w1 = create_resizable_window("Felix:gui_05_window_controller_01",100,100,400,400);
w1.add$ mk_drawable FlxGui::clear lightgrey;
w1.add$ mk_drawable FlxGui::write (100,100,font,black,"Basic Window Controller Test#1
↪");
w1.show();
w1.update();

var w2 = create_resizable_window("WINDOW 2",400,100,400,400);
w2.add$ mk_drawable FlxGui::clear blue;
w2.add$ mk_drawable FlxGui::write (100,100,font,black,"Basic Window Controller Test#2
↪");
w2.show();
w2.update();
```

18.5.2 The Event handler

The same as before!

```
// make an event handler for our window
chip ehandler
  (var w:window_t)
connector pins
  pin input : %<event_t
{
   // get a first event from the window manager
  var e: event_t = read pins.input;
  // while the event isn't a quit event ..
  while e.window.event.SDL_WindowEventID != SDL_WINDOWEVENT_CLOSE  do
    // print a diagnostic
    var s =
      match e with
      | WINDOWEVENT we =>
        we.type.SDL_EventType.str + ": " + we.event.SDL_WindowEventID.str + " wid=" +␣
↪we.windowID.str
      | _ =>
        e.type.SDL_EventType.str
    ;
    var linepos = 100 + 4 * lineskip;
    if s != "" do
      println$ s;
      var r = rect_t (100,linepos - 2*lineskip,300,4*lineskip);
      w.add$ mk_drawable fill (r, green);
      w.add$ mk_drawable FlxGui::write (100,linepos,font,black,"EVENT: "+ s);
      w.update();
    done
    // get another event
    e= read pins.input;
  done

  // we must have got a quit ..
  println$ "++++++CLOSE EVENT";
}
```

18.5.3 The Window manager

Noe for the fun bit. First, our mainline creates a window manager object:

```
begin
  //create a window manager
  var wm = window_manager();
```

Now, we create two window controllers. There will be clients of the window manager.

```
// create a window controller for our window
var eh1 = ehandler w1;
var wc1 = window_controller (w1, eh1);
var eh2 = ehandler w2;
var wc2 = window_controller (w2, eh2);
```

Note that in this case the same event handler is bound to two distinct windows, and then a window controller is bound to them, as well as the window (again!)

Next, we simply add the window controller clients to the window manager.

```
// attach controller to window manager
var wno1 = wm.add_window wc1;
println$ "Window number " + wno1.str;

var wno2 = wm.add_window wc2;
println$ "Window number " + wno2.str;
```

When we do this, we get back a window number, assigned by the window manager, so we can refer to the windows in a way the window manager understands (although we're not doing that here).

Finally:

```
  wm.run_with_timeout 10.0;
  println$ "gui05 quitting";
end
```

we just run the window manager, in this case with a timeout because its a demo.