RFC: Fir Extensions to the Elm Language

This document proposes all of the changes that might be included in the "ElmPlus" = Fir programming language that aren't in the Elm programming language. They can be grouped as follows:

1. Changes only affecting parsing.
2. Changes that are implemented by changes to the standard core package.
3. Changes that require compiler changes.
4. And so on, where some changes may require changes in several areas...

Changes That Only Affect Parsing

Using the `_` Character as a Visual Separator in Numbers

This is a feature that all programming languages should have as it is easy to implement while making numbers (both the mantissa's of `Float`'s and `Int`'s) much easier to read as follows:

```
x = 1_000_000_000 -- instead of 1000000000
```

Changes to the Standard Packages

Add a Console Module to the Core Package

When using Fir to write a console application, this Module provides functions to read environment variables, to use command line arguments, and to read and write strings to the terminal as well as read characters from the same (when the target back-end supports some or any of these, otherwise some sort of error value...). This new Console Module would only be able to be imported and used when the application is a command line application as designatied in the "fir.json" file for the project.

Add a FileIO Module to the Files Package

This FileIO Module provides the ability to find the home and current directories/folders, to manipulate directory/folder structure including adding and removing directories/folders, to get file status such as file size or last modified time, to remove a files, to read and write plain text files, and to read and write binary files (when the target back-end supports some or any of these, otherwise some sort of error value...). This new Console Module would only be able to be imported and used when the application is not a web application as identified from the "fir.json" file for the project.

Add Symbolic Binary Operators for Bitwise Functions

In a manner similar to the bitwise operators used by F#, implement `infix` operators `(&&&)`, `(|||)`, `(^^^)`, `(<<<)`, and `(>>>)` with the same associativities and precedence as for these operators in F# for bitwise and, or, exclusive or, shift left, and shift right, respectively; in particular, use of these symbolic operators fixes a potential name clash with the `Bitwise.xor` and the Boolean `Basics.xor` that is included in the "Prelude" automatically imported functions. These operators would be defined in the `Basics` Module to be automatically imported and can be applied to all integer/`Countable` types where the arguments and the result must be the same integer/`Countable` Type.

Add the `Lazy` Type and Package Back to the Language

This also allows having a standard `LazyList` module in the core package with appropriate functions as per non `Lazy List`'s. Although using the Lazy type and lazy lists are slower than using more iterative techniques such as recursive functions, they can be used to write elegant code when performance is not

critical such as in outer "loops" or for handling block structures such as producing a lazy list of array. The documentation for these should warn users and show appropriate use.

Add a `Change` Type to the Language

This is just a different name for the `ST` type in GHC Haskell and is required to do modification without side effects by running a "Real World" value through the calculation chain for functions to produce a new "frozen" value. With Linear Array's above supporting mutation in place when they are last used, this type is not necessary for arrays and would only be used for the Fir equivalent to the `STRef` type in Haskell. There would be a matching `runChange` function available to run the monadic chain of mutations/"side effects" and unwrap the wrapped `Changeable` type to produce its final contents.

Add a `Changing` Type to the Language - NO LONGER PROPOSED

This was proposed to just be a wrapper type for any type that has been "thawed" so that it can be modified by an `AndThenable` chain of functions. In this way, it is conceivable to modify any, even primitive, Type's to do low level system programming types of work. This is considered now to be unnecessary since linear arrays (and other "persistent data structures") can be modified in place when known to be "last-used" (also including the "Mod Cons" optimization), and it is likely that any lower level use would be relegated to the Foreign Function Interface (FFI - which may include the ability to emit C code when this is necessary...).

Add a `ChangingIndexable` Type to the Language - NO LONGER PROPOSED

This is a wrapper type for any indexable type such as a `LinearArray` that can be "thawed" to be changeable or "frozen" back to its original form. "Thawing" and "Freezing" will have both safe versions that make a copy of the original or unsafe forms that allow modification in-place. As per the immediate above, with many/most/all data structures modifyable in-place when they are "last-used", there would be no need for this feature.

Fix the Process Module to Support Multi-Processing

The Elm standard core package contains some partial implementations of this, but they are unusable since although a concurrent Process can be started, it cannot return any results cross-Process other than for Process's that have access to core facilities, not accessible to user code. This proposes implementing a full multi-Process facility much as the current documentation for "Process" proposes in order to support multi-Processing with user programmable communication through Process mailboxes; this could also support some sort of "async/await" interface for easy use of the feature.

Changes that Require Compiler Changes

Fix that the List Type may be Hidden

Currently, the List type is a "compiler magic" type that isn't exported by any Standard Modules including the List one; it should be defined in the List module just as other "compiler magic" types such as Int's, Float's, String's, and Char's are so that one can access this type by qualification as `List.List` when it is hidden by new Modules code that wants to define a different `List` Type with the same name for its own purpose.

Make the List Type able to be Specified by `[<Type>]`

Along with exposing the named List Type through the List module above to keep back compatability with Elm, in order to be consistent with other "compiler magic" collections as in Tuple's where the type is `(...)` and Record's where the type is `{...}`, the type of List's should be able to be specified as `[...]` to be the same as for Haskell where all three are implemented this way (the F# designation of these makes much less sense, although understandable given the need to support DotNet types).

Add all of the Bit Size/Signed and Unsigned Integer Options

This will include `Int8`, `Int16`, `Int32`, `Int64`, `UInt8`, `UInt16`, `UInt32`, and `UInt64`, with the current `Int` equal to `Int32` and a new `UInt` equal to `UInt32`. All of these will have a context or "type class"/`capability` of `Countable` with a set of standard operators and functions that can be called on any of them when all arguments and results are the same type.

Add `Float32` to the Float Types

This will include `Float32` and `Float64` with the current `Float` equal to `Float64`. All of these will have a context or "type class"/`capability` of `Uncountable` with a set of standard operators and functions that can be called on any of them when all arguments and results are the same type.

Add `BigInt` Type for Multi-Precision Integers

This type will also automatically subscribe to the `Countable` `capability`.

Make Contexts as in `comparable`/`appendable` More Consistent

First, all types including tuples, records, type aliases, custom types, and functions should be "Equatable" in a default sense but type aliases and custom types can have the default definition for being "Equatable" overridden to be based on certain fields, different ordering of tags, etc.

Next, all types should be automatically comparable if their fields are comparable, which should be true by default for all types except for functions. Again, the default definition of being "Comparable" can be overridden as for "Equatable"

The "number" context will be expanded to include all of the new integer and float types, with sub contexts for all the types of integers and for all the types of floats, rationals, complex numbers, etc.

Some of these changes are related to using the following "capabilities" to override default definitions and apply custom features to type aliases and custom types.

Change that "Triple-Quote String's Apply String Escapes

Currently in Elm, triple quoted strings automatically include carriage return, linefeed, and quotation symbols within the triple-quoted string as if they had been typed as escape characters, but there is still the difficulty that enclosing strings that contain escaped characters won't be correctly interpreted by string processing in an Elm program. For instance, if the string represents Elm source code and contains a `\ x = 42` lambda definition, the Elm compiler will complain at the `\` symbol as trying to interpret it as an escape withou the required following character so one has to change these into double `\\` sequences; also, combinations that are intended to be escape sequences will also have to have the single back slashes converted to double back slashes. Even for Unicode code point escape sequences, one could just copy and

paste in the actual code characters when a character is desired that can't be typed on the current language keyboard, so is unnessary within these "special" strings.

This is potentially a back breaking compatibility change so one might keep the current behaviour for source files with the ".elm" file extension and reserve this new behaviour for source files with the new ".fir" file extension.

Add Contexts/`capability`'s/`Ability`'s to the Language

These are similar to GHC Haskell's Type Classes with a different name and (probably) instantiated by "duck typing" where if a type implements all of the required functions for a `capability` inside a `where` clause, it has that capability. The justification for `capability`'s instead of just Elm's context's is that the current contexts get more and more complicated as the number of primitive types and the variety of `capabilities` increase with all their combinations. For instance, already Elm had to define a `compappend` context for those types that have both `comparable` and `appendable` contexts and even with only the proposed `equatable` context (in order for lambdas/functions to be tested for equality/sameness), this adds even more combinations and confusion. Also, it would be good to have almost all types, including Tuple's, Record's, and Custom Type's, be `comparable` and `equatable` if their sub types are these, which would get very complex. As well, with the current Elm context system, one can't "mix and match" type restrictions freely where with `capability`'s one can. The syntax does not have to be complex, where all of the basic `capability`'s are defined with a default implementation in the `capability` definition `where` block, and types can override the default implementations by redefining them in their own `where` block at the time the Type (and maybe Type Alias) is defined, where this may only rarely need to be done for new Type's (and maybe Type Alias's) or for the more complex `capability`'s.

Proposed `capability`'s will be at least the following:

1. Equatable, which should automatically apply to all primitives and structured Types other than functions and types containing functions because all of the building blocks have this `capability`. Implementing instances of this `capability` for types containing functions would be the work-around so that these can also be compared for equality.
2. Sortable, automatically defined for all types except for functions and complex types containing functions.
3. FromStringable, which means that a Type can be parsed from a string input, which should be defined for all types except for functions and complex types containing functions.
4. Stringable, which means that a Type can be converted to a string, which should be defined for all types except for functions and complex types containing functions.
5. Boundable, which should be defined for all Types that have definite lower and upper bound limits such as all integer types, maybe float types and char types, but not `String` types which may theorectially be extended indefinitely, and likely do not include complex types.
6. Enumerable, which applies to Types that can be incremented so applies to a limited set of Types such as all integers, maybe `Char`'s, and not complex Type's in general but to custom Type's that have multiple variants where none of the variants contain any data.
7. Countable, which applies to any integer Type, where all Countable Type's can be converted to Uncountable Type's by use of the `fromCountable` function that is a part of this `capability` interface, possibly with loss of precision where the Uncountable representation isn't addequate to keep the Countable precision. It will also include functions and operators such as `(//)` (integer

division) that applies only to the Uncountable `capability (and also the modulo and remainder operators)`.
8. Uncountable, which applies to any `float` Type or types derived from `Float` type's/containing `Float` type's; this interface includes `truncate`, `ceiling`, `floor`, and `round` functions that convert to a Countable Type as well as functions and operators such as `(/)` (floating point division) that applies only to the Uncountable `capability`.
9. Numerable, which applies to the set of all types that are either Countable or Uncountable and will include the basic functions (and operators) that apply to both. These do not include pure equality/non-equality operators which apply to everything, not only numbers.
10. Appendable, applies to any Type that can be merged together without changing the order of the contents of either argument (just add the second to the first without sorting), such as `List`'s, `String`', Array`'s, and LinearArray`'s.
11. Traversable -> things like list's, seq's, etc. that have a sequence ordering.
12. Foldable -> things that can be reduced from a "many" to a "single" value through application of a folding function.
13. Mappable -> things that can be converted by use of a mapping function, which is also the diefinition of a `Functor` in Haskell and such languages but we don't refer to them that way in Elm/Fir.
14. MultiMappable -> includes map2 (or Applicative in Haskell) and also mapping by 3 to 5 arguments.
15. Indexable -> types whose components can be accessed by an integer index value for both reading and writing (Copy on Write when not "last-used"; mutated in place when "last-used").
16. Chainable -> actually defines the binding(s) for a Haskell monad although we don't call it that in Elm/Fir; and includes ordering of arguments both ways as in `andThen` and `chain` functions.
17. Changeable -> this is used for types that allow mutation via a monadic state combinator type chain of functions; an example of this would be the `Ref` type equivalent to the `STRef` type in Haskell.

Some of these capabilities/abilities, such as `Mappable`, `MultiMappable`, and `Chainable`, which are applied to Type's that are containers for one or more other same Type's, require Higher Kinded Type's (HKT's) in order to work. The pure functional languages of Haskell and PureScript have HKT's as necessary in their use of "Type Classes" which are essentially the same as the capabilities/abilities described here; OCaml, which is not a pure functional language also essentially uses HKT's in their equivalent abilities using "module's" but are limited to HKT's containing only one Type although that Type can be a compound Type containing other types such as a Tuple.

Add Many More `capability`'s

There need be `capability`'s to be similar to many of GHC Haskell's type classes such as `Showable`, `Readable`, `Enumerable`, `Boundable` (these and `Equatable` and `Comparable` may be derived automatically when the base types have the requisite `capability`'s, thus not including functions), as well as `Appendable`, `Traverseable`, `Foldable` (for which a `map` makes it a form of Functor and `map2` makes a form of Applicative without calling them that), `Chainable` (which makes the type a monad without calling it that), and so on as required.

Add the Ability to Specify Type Constraints

A new proposal is to explicitly specify when a Type and or type variable must have a particular `capability`; with this also applying to the definition of `capability`'s: This means that a `capability` can require a "lesser" `capability` be implemented on the Type in order to "add" this new `capability`, and function Type Annotations can also require particular `capability`'s. For example:

```
capability Equatable a => Sortable a where -- defines a capability
  <capability function type signatures with optional default
implementations>
type Whatever a = Constructor a with
  Equatable -- no `where` definitions when defaults function
implementations are suitable
  Sortable where
    <the actual functions to be used if defaults (if any) are unsuitable>
myfunc : Sortable a => a -> List a
```

which specifies the requirements for a `Sortable capability` and implements that `capability` for a new Custom Type (which would be used in the same way for a `type alias` to add `capability`'s to an existing Type).

Add a `LinearArray a` Type to the Language

By itself, this is an immutable content array where it contains elements all of the same Type just as for lists and the current persistent `Array` implementation, but can be used with other features so it can be modified in a structured way using similar techniques to how `STArray` works in GHC Haskell. Alternately, it may be considered to use compiler magic to do in-place mutation when the array to be changed is never referenced again after the modification call. These arrays will also support automatic elision of array index bounds checks when it can be proved that the index is within bounds (lifting the bounds check ouside of inner loops). To easier support the optimizations to make use of these contiguous arrays fast, they should likely be a built-in Type such as `List a` and can be represented in Type signatures as `[| <designated content Type> |]`, and by the same symbols when defining `LinearArray a` literals as for example `[| 1, 2, 3 |]` to define an array of `Number` containing the three `Number` elements. As a built-in Type, there will be operators to access the indexed contents of this `LinearArray` Type. There will be an access operator as `(!!) : LinearArray a -> Int -> Maybe a` which returns the contents of the given `Int` index wrapped in a `Maybe` or `Nothing` if the index is out of bounds; The modification operator would be as `(:=) : LinearArray a -> (Int, a) -> LinearArray a` with the compiler producing a new array if the old one is ever referenced again but if last used will modify the array in place and return the original array and no out of bounds indication returned but just the original array returned unmodified. As a built-in Type that allows mutation in space for specific circumstances, this will require some compiler "magic" to accomplish and therefore requires compiler changes.

Add `UnsafePointer`Primative Type to the Language

This and other "unsafe" types and functions as required will aid in Foreign Function Interface (FFI) with C/C++ although the definition of how this is done hasn't yet been finalized. For instance, there may be primitive allocation and deallocation functions that work with these so that the implementation of byte arrays can be almost directly written in Fir as can reference counting automatic memory management. The current `port` implementation may well be implemented by channelling these through the FFI.

Another unsafe function added to the language might be `unsafeConvert` that will cast any Type to any other type as long as they have the same memory representation (unchecked as to memory representation). Use of these "unsafe" abilities might be limited to only in conjuntion with the FFI module.

Add `do` Notation "Sugar" to the Language

This makes code using `Chainable` types much easier to read in minimizing the requirement for brackets and indentation to define scope of interor functions and will apply when manipulating `Chainable`'s of the same type within the `do` block. In actual fact, the `do` keyword may be optional in that as long as all of the "chains" containing a `<-` symbol are aligned (as they would have to be anyway) and the end of any path through the "do" block ends in a "wrap" or other function returning the `AndThenable` Type also aligned with the "block" (as it has to do anyway), everything is fine without the `do`.

In other words, `do` blocks are made up of assignments by the `<-` symbol to a lower-case Name where the right hand side of the symbol can have any alignment to the right of the Name and all Name's must align with each other, with lower-case Name definitions of values or functions using `=`, or with a function call that results in the same `AndThenable` Type (including the Type's it wraps).

It is now not considered to be a desirable feature of the Fir language and not really needed as, other than the `Chainable` capability that can be applied to container types, the resulting types are not general Monad types because it is not proposed that Higher Kinded Type's (HKT's) be added to the language as discussed below. Rather, it is considered that equivalent "chaining" code be placed in a bloack to apply some special formatting rules for the "fir-format" ability to avoid continual indentation further to the right for evey level of chaining. In that way, the code that is compiled and for which error messages are produced is the same code as was written by the programmer for less complexity in decoding error message and determining the cause of such errors.

Add a TEST Pragma that Tests Code When Run in Test Mode

This is like the Rust `#[test]` pragma, with tests having the possibility of being unified in sets by referring to combinations of module tests from programs defined in the current Elm test directory structure. In this way, function tests are defined in line with the code and should idealy be defined as the code functions are written with the tests used to qualify the code being able to be left in place. As well, code run in test mode will be run in a debug mode by default, with such things as bounds checks and number overflow/underflow errors flagged. Alternatively, this could be a `DebugAssert` mechanism that is only run when in a as-compiled Debug and/or Test mode. For instance, this could be written as a pragma as {-# DEBUG_ASSERT " <code to be run in this mode, with the result expected to be True>" #-}

Add `INLINE`/`NOINLINE` Pragmas

Fir will automatically fairly agressively inline functions as an optimization step, but when more control is desired over when this takes place, these pragmas may be used, as follows:

```
{-# INLINE #-}
add : Int -> Int -> Int
add a b = a + b
```

Note that one doesn't have to include the name of the function in these pragmas as for GHC Haskell but the pragmas must be used immediately preceeding the function definition to which they apply with the same alignment of the function Name.

Add Code Generation Macros Capability

In order to keep the Fir language simple, it is not proposed to have full Abstract Syntax Tree (AST) generation macros, but it is desired that Fir be able to generate repetitive looping code using a macro system similar to that of the Crystal language that can compute some values to be injected into the generated code at compile time. Rather than using some macro keywords as in Crystal, it is proposed that all macro definitions are defined as Haskell type pragmas used to instruct the compiler just as for the INLINE/NOINLINE/DEBUG_ASSERT pragmas.

Add Explict `forall` and RankNTypes - CHANGED TO USE WITH EXPLICT EXISTENTIAL TYPES WITHOUT RANKNTYPES

The following code has an explicit `forall a.` after the `:` symbol because the type variable `a` is not defined on the left side of the `"` symbol:

```
-- a Custom Type container containing a function of functions
type Church : Church (forall a. (a -> a) -> (a -> a))
```

When there is an explict `forall` as in the above, it is not implied that the type variable that has the implicit `forall` can also automatically be higher ranked but still must be a value with rank of 0 or 1 just as in current Elm; however, since such types may be ambiguous when it comes to Type Inference, functions using such Type's must alway use Type Annotions to ensure that their type can be inferred. What this implies is Existential Type's as in creating distinct Type's have the same Capabilities but are distinct from their underlying Type's as per the follwing Type Annotation for a `runChange` function:

```
runChange : forall s. Change s a -> a
```

with this Type Annotation meaning that the Type `s` may be any Type (of rank 0 or 1) but that it is considered to be distinct from any other use of that same type for the whole duration of `s`.

Add Higher Kinded Types (HKT's) - NO LONGER A PLANNED FEATURE

For use in definitions where the one can manipulate just the Type "constructor" without applying its type variables, which then results in a higher kinded type that would be a "real" type only when a sufficient number of type arguments have been applied. This is particularly required in definitions of `Chainable` contrained functions.

This is a requirement for defining some of the more complex `capability`'s such as for chained function mutation of array contents since the type of container must be modified to one that supports the passing of the "RealWorld" state across the chain.

Make the `let` and `in` Keywords Optional

These keywords are redundant as it is known that the code is a value or function definition when the Name starts with a lower-case letter and the Name is followed by an `=` symbol (possibly after some interposing pattern matching arguments including Name's to make it a function definition), with all alignment rules as

to finding the end of the definition defined by the first column of the first Name in the definition and all definitions in the same scope must have these Name's aligned, just as currently. The only difference is that without an `in` terminator, the following expression needs to also be aligned with (or to the left of) the definition Name's instead of any alignment within the current outer definition block as currently. This is what the Elm language formatter program does anyway. This is the same as GHC Haskell's use of `do` blocks in that the definition of `let` definitions then changes so that `in` is forbidden **but** the aligment is restricted to be always aligned for each line within a `do` block.

For use where both keywords are optional, use of the optional `let` keyword is completely redundant but use of the optional `in` keyword allows the following expression to be aligned to the right of the definition aligment where this might be considered useful (not according to standard formatting style guide rules). This change is completely backward compatible with the current use of `let` and `in` by Elm. Use without these as per the following code:

```
test =
--let -- removed the let
    x = 42
    str = "Hey there!"
--in -- removed the in
  str ++ String.fromInt
```

The only real use for the `in` is when using non-standard formatting such as several `let`'s on one line as per the following code:

```
test =
  x = 42 in str = "Hey there!" in str ++ String.fromInt x
```

This making `let` and `in` optional is not an important change and keeping them mandatory may be preferred in the interests of code clarity.

Things Not Proposed to Change - ADDED TO MAKE USING CAPABILITIES MORE GENERAL

Currently, it is not considered to provide the programmer the ability to add new symbolic operators other than as defined in the core package currently and as above in order to not have an excess of symbols, although consideration will be made to allow packages such as parsers to be included as "key"/core packages that may be allowed to define a few of these. This is to keep the language easy to learn and its packages and code easy to read and understand without having to look up definitions for a myriad of symbolic operators.

Also, currently it is not considered that the ability to turn common functions into infix operators with "back-ticks" around the name of the function be added back to the language as with a reasonably rich variety of mathematic and manipulation operators, this should not be necessary. For instance, a frequent use of this in GHC Haskell is the back-ticked `div` operator for integer division instead of floating point division with the `/` operator, but Elm/Fir have the `//` operator for that. It is proposed that the standard packages provide sufficient symbolic operators so that there shouldn't be a need for adding more. The difficult with being

able to use definied functions as operators is that, in order to be generally useful, they must also have the ability to have infix precedence and association runs specified for them, which adds complexity to little advantage.

Originally, it consider considered that the ability to define new `capacility`'s would be confined to the core package to remove the temptation of programmers to define a full Haskell type system, although it will be considered to add other `capability`'s to the core package than as listed above if they would be commonly used to simplify code written in Fir. As it is not proposed that Fir have Higer Kinded Type's (HKT's) that would allow generalized category theory Type definitions, it would be unlikely that Fir's limited `capability`'s be abused to excessive extension of the Type system other than as proposed above.

Make the `type` and `type alias` Keywords Optional

I thought that these keywords are redundant as the `Type` definition is known when the Name starts with a capital letter and the Name is followed by an `=` symbol, and the `alias` is known when the right hand side of the definition after the `=` refers to a Type (not a Constructor, which may have the same name as a Type) that is already defined. All alignment rules as to finding the end of the definition is defined by the first column of the first Name in the definition block and all definitions in the same scope must have these Name's aligned, just as currently.

However, there is an ambiguity in that there may be a definition of `TestType = Something` where if "Something" is the same word "TestType", if it is a Custom Type then it passes but if "Something" happens to be an existing Type then a `type alias` will result which may not be the desired outcome. Also, it precludes being able to check for cyclic `type alias` references (although if found then it could default to a Custom Type). Finally, in cases where ambiguities could commonly occur "Explicit is better than Implicit".

Therefore, these keywords will continue to be required in defining Type's and thier Alias's.

Summary

As can be seen above, there aren't all that many changes that affect the syntax to make using Fir much different than using Elm, with the major one being the "type class" feature of using `capability`'s to be able to use general functions without limiting the Module/Type with which they work, which feels something like having function overloading as used in other languages.

Appendix

Examples Using and Not Using `capability`'s

To be provided...