The Frank Manual

Conor McBride

May 2, 2012

Shall we be pure or impure? **Frank** is a strongly typed functional programming language answering this question 'Yes.'. Frank is heavily influenced by Paul Blain Levy's call-by-push-value calculus, and follows that work in distinguishing values *being* and computations *doing*.[1] Mais, aussi, c'est la nostalgie de la boue. I designed Frank because I miss programming in ML as much as I enjoy the discipline of Haskell.

Frank bears a considerable conceptual resemblance to Bauer and Pretnar's **eff**: the two were developed independently around the same time in central Scotland, a fine tradition. I first described Frank at the workshop on Effects in Type Theory, Tallinn, December 2007, but have done very little about it until now. Frank, at time of writing, has the more developed type system, tracking effect permissions as well as value compatibility, but eff has a much clearer semantics. I view Frank primarily as a learning experience on my own part, but I hope it proves of some small value, before I eff off back to type theory.

Health warning. The current Frank implementation is but a few days' work. 'Rough and ready' is half right.

I should like to thank Hugo Herbelin for giving me a place to hide while I did this, and taking an interest, and asking me lots of questions to which I need to know the answer, and telling me about lots of things I should read. I apologise in advance for not having read them yet.

Conor McBride
Paris, April 2012.

1 Getting Started

Frank is implemented in Haskell and can be downloaded from Hackage with the incantation

```
cabal install frank
```

Unfortunately, Frank is written making heavy use of the Strathclyde Haskell Enhancement, so you may find that `cabal` infects your computer with it, whether you like it or not.[2] The package delivers an executable, `frank`, which is a typechecker and interpreter. If you issue the command

```
frank myfile
```

the system will seek a file `myfile.fk` and attempt to make sense of then execute its contents, checking for some pathologies, whilst completely ignoring others. Error diagnostics, while not nonexistent, are currently somewhat inscrutable, for which I apologise in advance.

What might you put in your file? I wrote a file called `hello.fk` containing the following...

[1] The interplay between doing and being, like strangers in the night exchanging glances, provoked the name.
[2] SHE provides 'idiom brackets', in many ways a forerunner to Frank. I would not be without them.

```
data List X = nil | X :: (List X)

map {X -> Y} (List X)   []   List Y
map f nil        = nil
map f (x :: xs)  = f x :: map f xs

main [Console] List ()
main = map ouch ('h' :: ('e' :: ('l' :: ('l' :: ('o' :: nil)))))
```

. . . and then this happened:

```
bash-3.2$ frank hello
hello(() :: (() :: (() :: (() :: (() :: nil)))))
```

What might one glean from this?

1. Frank allows the definition of parametrized inductive datatypes, such as `List X`, where such definitions are introduced by the keyword `data`. Type level identifiers — constructors and variables — begin in Uppercase. After the = sign, you give a choice of constructors, separated by `|`s. Constructors are *symbolic*, i.e. anything-but-beginning-Uppercase, and are not necessarily prefix. Too many parentheses are mandatory, as in `X :: (List X)`, but at least it is easy to read the declaration as a *template*, with symbolic *marks*, and value types standing in the *places* where, upon usage, inhabitants of those types will go.
2. You can write polymorphic higher-order functions, like `map`.
3. Top-level functions have type signatures, which contain lots of funny brackets. The [square] brackets the principal connective of the signature. To the left of the brackets goes a *template*, with symbolic marks and value types in places. To the right goes the value type obtained by invoking the function.
4. Curly braces are somehow important. Braces, or only in American 'suspenders', embed *computation* types into value types. Computation types describe things which are *done*, but a 'way of doing something' *is* something, and it is something distinct from either the act of doing it or its result. The `map` needs `f`, a way to compute `Y`s from `X`s, and in the `(x :: xs)` case, it *does* `f` to `x`, as well as passing the value `f` into the recursive call on `xs`.
5. Brackets are not always empty: inside, they list the *effect signatures* required by a function. The `main` function, which takes no arguments, none the less needs access to `Console` effects, in order to output some characters, via the `ouch` command. The latter outputs one character, returning a value `()` of unit type, also `()`. The `map` operation traverses the list of characters, emitting them sequentially. The `frank` runtime also prints the value which results.
6. A good treatment of associative operators remains to be implemented, hence an excess of parentheses.
7. The `map` operation is somehow effect-polymorphic. The `Console` effects required by `main` are somehow made available to the functional argument of `map`, which is why one can traverse with an impure operation like `ouch`. If you delete `Console` from `main`'s bracket, the program is rejected due to ignorance of `ouch`.

But enough impressionistic handwaving! Let us be a little more systematic.

2 File Structure

A Frank source file is a series of top-level blocks. Each line whose first character is not whitespace begins a block which ends at the start of the next line whose first character is not whitespace. Indentation thus signals the continuation of a block, but is otherwise unimportant. Newline-then-space means the same as space. Blocks currently come in five varieties:

1. comments, beginning with keyword `note`
2. datatype declarations, beginning with keyword `data`
3. signature declarations, beginning with keyword `sig`
4. function declarations
5. function definition lines

The order in which blocks occur in a file only matters if they concern the same entity. Duplicate declarations are not currently flagged as problematic (although this is perhaps an oversight) — the first wins. You may scatter the lines of a function definition far and wide, spliced with other blocks (although this is perhaps bad practice), but they are prioritized in order of occurrence, following the conventions of pattern matching.

If the Frank interpreter is given a file with a parameterless `main` function, that function will be executed, performing any `Console` commands, reporting commands from other signatures, and printing any value ultimately returned. That is, the `main` function may enable any signature of effects, but the runtime system can only perform `Console` effects.

3 Types and Signatures

Frank has mutually defined syntaxes of *value* types V, *computation* types C, and *signatures*, Σ.

V	$::=$	$D\ V^*$	datatypes with parameters
	$\mid$	$\{C\}$	suspended computations
	$\mid$	$\{\}$	the empty type
	$\mid$	$()$	the unit type
	$\mid$	$[\Sigma\ ?\ \langle[\Sigma']\rangle^?\ V]$	a request type
	$\mid$	X	value type variable
C	$::=$	$\langle[\Sigma]\rangle^?\ V$	effect and value
	$\mid$	$V \to C$	function type
Σ	$::=$	$\sigma^{,*}$	comma-separated signature actions with parameters
σ	$::=$	$S\ V^*$	a signature extension action with parameters
	$\mid$	$\{\}$	the purity action

Grammar grammar. I use angle brackets $\langle \ldots \rangle$ to delimit compound grammatical entities, not as concrete syntax: a superscript $^?$ indicates an optional component; * indicates zero or more repetitions, sometimes with a given separator; $^+$ indicates one or more, similarly.

Datatype constructors D and signature constructors S are Uppercase identifiers, inhabiting separate namespaces. A datatype is a type of values generated by a fixed choice of *value constructors*. A signature is collection of *commands* by which a given computation may perform some sort of interaction external to itself. A *request* type is a value type which reifies commands, so that we may explain how to respond to them.

The effect signature actions in a computation type explain what the computation is allowed to do. For example, [Console] () is the type of computations which may input and output characters before returning the trivial value in the unit type. Frank is a call-by-value language, but a value of type {[] V} represents not a V, but the ability to compute a V with no external interaction. Laziness is thus possible, but explicit and distinct. The 'purity action' is an artefact of Frank's effect polymorphism, which will need careful explanation later.

Commands, constructors and functions are declared by giving *templates*, interspersing *marks* and *places*. A *mark* is a sequence of non-whitespace characters excluding `()[]{},;!` which does not begin in Uppercase or with a single quote, and is not a keyword. A *place* is indicated by a value type, enclosed in parentheses if it is a datatype with at least one parameter. A typical template is

```
if Bool then {[] X} else {[] X}
```

where `if`, `then` and `else` are the marks, and the places are taken by a Boolean value and two suspended computations of some type `X`. A template may either be a single prefix symbol, followed by any number of places, or a mixfix combination, with no two places consecutive. At present, there is no way to refer to a mixfix template *itself*, without filling its places — this is clearly an omission.

The keywords are currently as follows:

```
note data sig -> ; , = | !  ?
```

3.1 Datatypes

A datatype can have zero or more parameters, being value types abstracted by value type variables (also beginning Uppercase).

$$\texttt{data}\ D\ X^* = \mathit{template}^{|*}$$

The following datatype is built in

```
data Bool = tt | ff
```

The only type variables permitted in constructor types are those bound in the head of the declaration, so existential types like

```
data MonadExp Y = ret Y | (MonadExp X) >>= {X -> MonadExp Y}    (×)
```

are disallowed. It is, however, permitted (i.e., not excluded) to instantiate datatype parameters in recursive positions, as in nested types.

```
data Tm X = var X | app (Tm X) (Tm X) | lam (Tm (Maybe X))
```

It is currently permitted to define non-strictly-positive datatypes, such as

```
data Bad = bad {Bad -> Bad}
```

but don't bet on that staying alive!

An important point to which we shall return is that datatypes are at present in no way effect-polymorphic. If one defines binary trees by functional branching, thus

```
data Tree X = leaf X | node Bool -> Tree X
```

then the function `f` packed in some `node f` is, you may trust and must ensure, pure (up to considerations of totality). Frank is currently not a total language, but it is well suited to become one: effect-tracking is one way to negotiate partiality in a total setting.

3.2 Primitive Value Types

The unit type `()` contains one value, also `()`.

The empty type {} has no values, and an 'ex falso quodlibet' eliminator, which we shall see later.

The type `Char` contains character constants, such as `'h'`,`'\n'`.

3.3 Signatures

A signature is a choice of commands parametrized by some value types, declared as follows:

$$\texttt{sig}\ S\ X^* = \langle \mathit{template} \langle \texttt{[]}\ \langle V \rangle^? \rangle^? \rangle^{|*}$$

A command is specified by a template with places for its parameters, followed by an empty bracket `[]` pronounced 'returns', followed by a value type, indicating the type of values returned by the command. It is permitted to drop the return value type (and even the bracket) if it is the unit type `()`.

There is currently one built-in signature, declared as if

```
sig Console = ouch Char | inch [] Char
```

meaning that `ouch` has one `Char` parameter and returns unit, and that `inch` has no parameters but returns a `Char`.

As with datatypes, the only type variables permitted in command types are those bound by the signature head. In particular, the following desideratum is currently prohibited:

```
sig Abort = abort [] X                                    (×)
```

The workaround is to define a command delivering the empty type

```
sig Abort = aborting [] {}
```

and then to define a polymorphic wrapper

```
abort [Abort] X
abort = aborting ! {}
```

which eliminates the empty type to get you out of whatever trouble you find yourself in. An explanation of this mysterious syntax will follow presently.

3.4 Function Declarations

A function declaration

$$\mathit{template}\ \langle [\Sigma]\ \langle V \rangle^? \rangle^?$$

equips a template with places showing parameter types with a signature of permitted effects and a return type. The return type may be omitted if it is `()`. Less usefully, the signature and return type may both be omitted if the signature is empty and the return type is `()`. Declared functions are polymorphic in their free value type variables, which may be instantiated with any value type, making use of traditional Milner-style specialisation with unification variables at each point of use. Declared functions are also effect-polymorphic, and specialised in a very particular way at each point of use.

4 Typing Expressions

Frank is typechecked bidirectionally. The prospect of any sort of complete type inference is not one I have locally considered, although it is conceivable that some sort of Boolean ring-solving might make for a precarious notion of principal type scheme. I am more likely to move in the direction of dependent typing, where you can expect considerable benefit from constraint-solving but will certainly need a 'manual override' on occasion.

Typing is relative to

1. a global collection of datatypes
2. a global collection of polymorphic declared functions
3. a local set of enabled signatures
4. a local context of monomorphic values.

Terms can be divided into 'checkable' and 'synthesizable' varieties. I shall pretend all templates are prefix to simplify the presentation of the syntax.

$check$	::=	$synth$	if you can synthesize, you can check
	|	$c\ check^*$	value constructor
	|	`'` $char$ `'`	character literal
	|	`()`	the element of the unit type
	|	$\{\langle\langle pat^+ \texttt{ -> } \rangle\ check\rangle^{\mid +}\}$	suspended computation
	|	$check\ \{\}$	ex falso quodlibet
	|	$synth$ `?` $check$	handler installation
$synth$	::=	x	local variable
	|	f	global function
	|	o	command
	|	$synth\ spine$	
$spine$	::=	`!`	empty spine
	|	$check^+$	nonempty spine
pat	::=	x	a pattern variable
	|	$c\ pat^*$	a constructor pattern
	|	`'` $char$ `'`	character literal
	|	`()`	the element of the unit type
	|	`[` pat `]`	return request
	|	`[` $o\ pat^*$ `?` pat `]`	command request with continuation

Contexts Γ assign value types to variables. The judgment forms are as follows:

1. checking that a value type accepts a $check$ term

$$\Gamma|\Sigma \vdash V \ni check$$

2. synthesizing a value type from a $synth$ term

$$\Gamma|\Sigma \vdash synth \in V$$

3. feeding a $spine$ to a computation, yielding a value

$$\Gamma|\Sigma \vdash C\ spine\ V$$

4. checking a suspended computation

$$\Gamma \vdash C \ni pat^* \texttt{ -> } check$$

5. checking a pattern, extracting a context

$$V \ni pat \dashv \Gamma$$

6. checking that a signature action is enabled

$$\Gamma|\Sigma \vdash \Sigma$$

Some judgments carry an *ambient signature set*, written Σ, which will always contain finitely many signatures $S\ \vec{V}$ with each S distinct. We may define the action of each σ on such a Σ: $\{\} \cdot \Sigma$ yields the empty signature set. $S\ \vec{V} \cdot \Sigma$ extends Σ, shadowing any S-signature that Σ previously contained. It is easy to check that signature actions form a monoid acting on signature sets.

So, let us have the rules.

$$\frac{\Gamma|\Sigma \vdash s \in V}{\Gamma|\Sigma \vdash V \ni s} \qquad \frac{\Gamma|\Sigma \vdash U_i[\vec{V}/\vec{X}] \ni t_i}{\Gamma|\Sigma \vdash D\ \vec{V} \ni c\ \vec{t}}\ \texttt{data}\ D\ \vec{X}\ \text{has}\ c\ \vec{U}$$

$$\frac{}{\Gamma|\Sigma \vdash \texttt{Char} \ni \texttt{'}char\texttt{'}} \qquad \frac{}{\Gamma|\Sigma \vdash \texttt{()} \ni \texttt{()}}$$

$$\frac{\Gamma \vdash C \ni b_i}{\Gamma|\Sigma \vdash \{C\} \ni \{b_1|\ldots|b_n\}} \qquad \frac{\Gamma|\Sigma \vdash \{\} \ni t}{\Gamma|\Sigma \vdash V \ni t\ \{\}}$$

$$\frac{\Gamma|\Sigma \vdash h \in \{[\Sigma_0\ \texttt{?}\ [\Sigma_1]\ U]\ \texttt{->}\ [\Sigma_2]\ V\} \quad \Gamma|\Sigma_0 \cdot \Sigma \vdash U \ni e \quad \Gamma|\Sigma_0 \cdot \Sigma \vdash \Sigma_0 \cdot \Sigma_1 \quad \Gamma|\Sigma \vdash \Sigma_2}{\Gamma, \Sigma \vdash V \ni h\ \texttt{?}\ e}$$

Let's begin with that last rule, as it's the least conventional. It explains how to modify the ambient signature set Σ by a given extension Σ_0, handling the extra commands in $\Sigma_0 \cdot \Sigma$. The request type will contain Σ_0 requests, each of which consists of a Σ_0 command and a resumption which might do anything listed in $\Sigma_0 \cdot \Sigma_1$.