Cervoise: Draft 03/2015

Jacques-Pascal Deplaix

Cervoise is a functional programming language that started as a small toy language to learn. Aside of that, the motivating goal was to do some sort of synthesis between OCaml and Haskell (two languages that I particularly like) and add exceptions checking (like in Java).
The idea came after having read the TAPL (Types And Programming Languages) by Benjamin C. Pierce that, at some point, talks about System Fω. So after, I decided to start making a compiler incrementally, starting by a Simply Typed Lambda Calculus type-checker, then moving to System F, having a backend and so on.
On the backend part I chose LLVM because it's both easy and that it generates optimized and portable code.

I will present the state of the project before the start of the Master project (before March 2015) and the things done from that point until now (18 March 2015)

The project repository is here: https://github.com/jpdeplaix/cervoise

Code examples:

The gallery of working examples can be found here:

before March:
https://github.com/jpdeplaix/cervoise/tree/a3fd02f61d/examples

and Now:
https://github.com/jpdeplaix/cervoise/tree/bfaecc5d3e/examples

What has been done so far:

NOTE: The grammar of Cervoise before March is available at the end of this draft.

Preparation of dynamic exceptions:

I first prepared some of the codes for dynamic exceptions as I plan to have it later.
The handling of those would be like:

- Allowing to catch them at any point of the program (for any term)
- Add the following construct in case these dynamic exceptions haven't been handled above:

```
let main try with
  | CannotAllocate -> ()
end =
  dosomething ()
```

In particular I unified the backend representation of functions so that they now takes always an extra argument for the exception.

Separation of effects and exceptions

Before March, internally what was called effects in Cervoise source code was in fact exceptions, so I had to replace exceptions by effects so that now instead of having the type `Int -[A]-> Int` that described the function that can raise the exception A, we have: `Int -[Exn [A]]-> Int` that describes the function that have the effect of raising an exception A
The two builtin effects are the IO effect (see the next section) and the Exn effect.

The IO effect

As the IO effect has been introduced, I had to use it and enforce it when it was needed. The IO effect makes sense for bindings, so I just had to enforce every bindings to have it's last arrow decorated with the IO effect. The last arrow only is needed because I also enforced the functions binded to not return functions (as it was done before). To do that I wrapped every binded functions into lambdas to allow partial evaluation.
The IO effect also allowed me to check whenever or not there is an effect under a type abstraction so that I can now forbid it to be type safe again.
I also wanted to forbid effects at toplevel so that every side effects are contained into the main function. This main function (that I needed to make a special case of) is the only toplevel value that can have the IO effect. I also enforced it to have the type Unit.

Syntactic sugars and the builtin Unit type

The following syntactic sugars have been added:

- The identifier _ has been introduced
- The type Unit has been added as builtin for every modules (definition below)
- The syntactic sugar `()` has been introduced as a short version of `(_ : Unit)`
- The sequencing operator `x; y` has been added as a short version of `let () = x in y`
- The Haskell-style keyword `\` (backslash) for lambdas has been allowed aside with the old UTF-8 lambda λ

The Unit type:

```
type Unit = Unit : Unit
```

To do all these syntactic sugar cleanly I had to add another intermediate AST just after the parser. Before that the syntactic sugar was done directly in the parser. Now, the parse tree gives the AST with the syntactic sugars and a module called `Sugar` (maybe not really a good name tho) have to transform the parse tree to another AST without syntactic sugar (called `UnsugaredTree`).

Polymorphic effects

I also allowed to have polymorphic effects. The effects variables have kind φ and it can be used like: `forall (X : φ), Int -[X, Exn [A]]-> Int`. All effects will be automatically merged when the type will be applied. For example with the example above: `f [Exn [B], IO]`, the result will have the effect [Exn [A | B], IO]
To avoid mistakes while using effects, I forbid the pure arrows (`->`) for higher-order functions (in

values, types and exceptions parameters). So that the following code is now forbidden:
`let apply : (Int -> Int) -> Int -> Int`

instead we would have to write either:
`let apply : (Int -[]-> Int) -> Int -> Int`

or:
`let apply : forall (X : φ), (Int -[X]-> Int) → Int -[X]-> Int`

Same thing in type aliases as they can be used as parameter. Without that enforcement, it would lead users to possible mistakes in case they making a module that is used anywhere else in the code (in the case of a library for instance).

Typeclasses: the goal

`TODO`

Grammar before March 2015:

The following grammar is almost accurate. The only thing not accurate that has been simplified is the parenthesis for `kind`, `type` and `term` which wasn't allowed for every cases. This was fixed by this commit (after some parser changes): 4045d56.

```
toplevel-expr ::= 'let' lower-name argument* (':' type)? '=' term
                | 'let' 'rec' lower-name argument* ':' type '=' term
                | 'type' 'alias' upper-name '=' type
                | 'type' upper-name-ident (':' kind)? '=' '|'? variants
                | 'exception' upper-name-ident type*
                | 'let' lower-name ':' type '=' 'begin' <llvm-code> 'end'

lower-name ::= ['a' 'z'] (['a' 'z'] | ['A' 'Z'])*
upper-name ::= ['A' 'Z'] (['a' 'z'] | ['A' 'Z'])*
module-name ::= upper-name ('.' upper-name)*
lower-ident ::= (module-name '.')? lower-name
upper-ident ::= (module-name '.')? upper-name

variants ::= upper-name ':' type ('|' variants)?

argument ::= upper-name
           | '(' lower-name ':' type ')'

type-argument ::= upper-name
                | '(' upper-name ':' kind ')'

kind ::= '*'
       | kind '->' kind
       | '(' kind ')'

exns ::= upper-ident ('|' exns)*

type ::= upper-ident
       | type '->' type
       | type '-[' exns ']->' type
       | 'forall' type-argument+ ',' type
       | 'λ' type-argument+ ',' type
       | type type
       | '(' type ')'
```

```
term ::= lower-ident
       | 'λ' argument+ '->' term
       | 'let' lower-name argument* (':' type)? '=' term 'in' term
       | 'let' 'rec' lower-name argument* ':' type '=' term 'in' term
       | term term
       | term '[' type ']'
       | 'fail' '[' type ']' exn-value
       | 'match' term 'with' '|'? pattern-clauses 'end'
       | 'try' term 'with' '|'? try-patterns 'end'
       | '(' term ')'

exn-value ::= upper-name
            | '(' upper-name term+ ')'

pattern-clauses ::= upper-ident pattern-args* '->' term ('|' pattern-clauses)?

pattern-args ::= lower-name
               | upper-ident
               | '(' upper-ident pattern-args+ ')'

try-pattern-clauses ::= upper-ident lower-name* '->' term ('|' try-pattern-clauses)?
```