Combining effects with dependent types

Maya Mückenschnabel

supervisor: Tomáš Petříček

Abstract

Dependent type systems provide a novel way of reasoning about program correctness, by embedding behavior of the program into the more expressive type system. Correctness is achieved by not allowing incorrect states to be representable.
Languages like Idris show that dependent type systems are practically useful, not only for formal proofs, but also for creating fewer bugs in production. But the purity of computation poses a problem for composability of stateful computations and of side effects.
Effect handlers provide one possible solution for this problem. In this thesis we propose an effect extension of dependent type systems. The resulting system not only makes it possible to provide guarantees about correctness of a program, but also make it easy to compose such guarantees using effects. We formalize the type system and present a prototype implementation.

Generators in Kelp

With algebraic effects, we can define our own generators, similar to the ones seen in Python or C#. With similar technique we can construct async/await but algebraic effects are more flexible, there is no function coloring problem, and asynchronous functions and synchronous functions can be intermixed without any modification.

```
:Yield (Effect [Integer] Boolean)

:go
(lambda [n       :: Integer
         produce :: Boolean]
  :-> []
  ;; each time go is invoked we can both Yield
  ;; and print to stdout
  :! [Yield core:io:Stdout-write]
  (if (not produce)
      (core:io:print-line "done")
      (begin
        (let [next-n (+ n 1)
              next-produce (raise Yield next-n)]
          (go next-n next-produce)))))

:main
(lambda []
  ;; with handler catches Yield
  ;; so only the printing gets forwarded upwards
  :! [core:io:Stdout-write]
  ;; yield true while n < 3
  (with [Yield (lambda [resume n]
                 (core:io:print-line n)
                 ;; resume returns to the place
                 ;; where raise was called with
                 ;; a value
                 (resume (< n 3)))]
        ;; run go for the first time
        (go 0 true)))
;; STDOUT
0
1
2
done
```

Type and effect polymorphic map

We can create a map that can take any function and any list and produce a new list by applying the function on each element. This function is a dependent function and combines both the dependent type system and the algebraic effect system.
The thing to note here is that the type of `[1 2 3]` is not actually `(List Integer)` initially. Rather the constraint `(Collection Integer 3)` is constructed and coerced by the type-checking algorithm into the proper type.

```
:map
(lambda [A   :: Type
         B   :: Type
         E   :: (List Any-Effect)
         f   :: (-> [A] B)
         lst :: (List A)]
  :-> (List B)
  :! E
  (if (empty? lst)
      lst
      [(f (first lst)) . (map f (rest lst))]))

:main
(lambda []
  :! [core:io:Stdout-write]
  (map Integer Unit core:io:Stdout-write
       core:io:print-line
       (map String Integer []
            integer->string [1 2 3])))
;; STDOUT
1
2
3
```

Algebraic effects

Algebraic effects provide a way to specify side-effects as a part of our program APIs. Not only that, effects provide tools for writing libraries that are in other languages only available as in-language constructs, like:

- Async/Await
- Generators (`yield`)
- Fibers, Green Threads and Coroutines
- Resumable Exceptions
- ...

Novel features of this type system

The type system is the key innovation of Kelp and is formalized in the thesis. This type system:

- Combines dependent types with effect handlers
- Supports unified tuple and list literals via constraints
- Powerful bidirectional type inference for lists
- Allows control of effects in type-level computations

Type system and effects

The function application behaves similarly to the regular bidirectional type-checking. The difference is that the application is only allowed if the effects of the function are in the effect context. That is either are handled directly or passed to the caller.

$$\frac{\Gamma, E \vdash t_1 \Rightarrow \rho_1 \to \rho_2 \uparrow^! \varepsilon \qquad \Gamma, E \vdash t_2 \Leftarrow \rho_1 \qquad \varepsilon \subseteq E}{\Gamma, E \vdash t_1 t_2 \Rightarrow \rho_2}$$

Creating a lambda with an effect encapsulates it, so we can create functions that have side-effects without having to have those side-effects allowed ourselves.

$$\frac{\Gamma \boxplus \{x : \rho_1\}, \varepsilon \vdash t \Rightarrow \rho_2}{\Gamma, \emptyset \vdash \lambda x.t \Rightarrow \rho_1 \to \rho_2 \uparrow^! \varepsilon}$$

Raising an effect behaves from the callers perspective like calling a regular function. We can check it the same way as application.

$$\frac{\Gamma, E \vdash t_1 \Rightarrow e \qquad \Gamma, E \vdash t_2 \Leftarrow \rho_1 \qquad e : \rho_1 \to^! \rho_2 \in E}{\Gamma, E \vdash \textbf{raise}\ t_1\ t_2 \Rightarrow \rho_2}$$

Head-Tail/Tail-Head synthesis

We introduce constraints and namely collections to extend the bidirectional type-checking algorithm by adding non-instantiable pseudo-types that can only appear as a result of writing language constructs that are not syntax driven. In Kelp, for example, `[1 2 3]` can be a tuple, compile-time known length list or a regular list.
We extend the bidirectional type-checking by adding a special $\vdash_c$ judgment that allows terms to be synthesized and checked to be of constraint pseudo-types. This extension also allows for stronger type-checking of collections, as we can both infer type from the first element and coerce the others or from any element further down the list.

$$\frac{\Gamma, E \vdash t_1 \Rightarrow \rho_1 \qquad \Gamma, E \vdash_c t_2 \Rightarrow (\textbf{Collection}\ \rho_1\ n)}{\Gamma \vdash_c (\textbf{cons}\ t_1\ t_2) \Rightarrow (\textbf{Collection}\ \rho_1\ n+1)}$$

$$\frac{\Gamma, E \vdash_c t_1 \Leftarrow \rho_1 \qquad \Gamma, E \vdash_c t_2 \Rightarrow (\textbf{Collection}\ \rho_1\ n)}{\Gamma \vdash_c (\textbf{cons}\ t_1\ t_2) \Rightarrow (\textbf{Collection}\ \rho_1\ n+1)}$$

Future work

In future the work can be extended to formalize the computational semantics based on the works of Plotkin and Pretnar combined with dependent types. The typing rules for effects are fairly restrictive a capability-based tracking of allowed effects in the spirit of Effekt could prove to be more useful.