A type system for dynamic instances

Delft University of Technology

Albert ten Napel

September 5, 2019

Outline

Effects

- Used everywhere in programming
- Examples: mutable state, input/output, reading/writing to files/channels, randomness, ...
- Makes reasoning, testing and debugging difficult

Effects

Example

```
guesses = 0

// guess : () -> Int
def guess():
  global guesses
  n = input("give a number: ")
  guesses += 1
  if n == "42":
    print("you guessed correctly!")
  else:
    print("wrong number")
  return guesses
```

Algebraic effects and handlers

- Approach to programming with effects
- Composable: easily combine different effects
- Type safe

Algebraic effect interfaces

Example

```
effect State {
  get : () -> Int
  put : Int -> ()
}

effect IO {
  input : String -> String
  print : String -> ()
}
```

Using algebraic effects

Example

```
guess : () -> Int!{State, IO}
guess () =
  n <- #input("give a number: ");
  x <- #get();
  #put(x + 1);
  if n == "42" then
    #print("you guessed correctly!")
  else:
    #print("wrong number");
  guesses <- #get();
  return guesses
```

Handling algebraic effects

Example

```
handleGuessIO : Int!{State}
handleGuessIO =
  handle( guess() ) {
    input msg k -> k "42"
    print msg k -> k ()
    return x -> return x
  }
```

Shortcoming

- How to express multiple mutable references?
- How to express opening and reading/writing to files/channels?
- How to express local effects?

Dynamic effect instances

Example

```
r1 <- new State;
r2 <- new State;
handle#r1 (
  x <- r1#get();
  r2#put (x + 1)
) { ... }
```

Escaping instances

Example

```
escape ref =
  return \() -> ref#get ()

escaped =
  ref <- new State;
  fn <- handle#ref (escape ref) { ... };
  return fn
```

The problem

- Dynamic instances allows us to express more (local) effects
- Dynamic instances are not type-safe
- How to ensure that all operations will be handled?

Miro

- Language with algebraic effects and handlers
- With a restricted form of dynamic instances
- We can define mutable references and local effects
- Type safe: ensures that all operations will be handled

Miro - Creating instances

Example

```
effect Config {
  get : () -> Int
}

makeConfig : forall s. Int -> (Inst s Config)!{s}
makeConfig [s] v =
  new Config@s {
    get () k -> k v
    return x -> return x
    finally x -> x
  } as x in return x
```

Miro - Using and handling instances

Example

```
useConfig : Int
useConfig =
  runscope(myscope ->
    -- c : Inst myscope Config
    c <- makeConfig [myscope] 42;
    x <- c#get();
    return x)
```

Miro - Runscope typing rule

T-RUNSCOPE

$$\frac{\Delta, s_{var}; \Gamma \vdash c : \tau \,!\, r \qquad s_{var} \notin \tau}{\Delta; \Gamma \vdash \mathsf{runscope}(s_{var} \rightarrow c) : \tau \,!\, (r \backslash \{s_{var}\})}$$

How does Miro make instances safe?

- Effect scopes: instances belong to a specific effect scope
- Instances always have a handler associated with them
- All instances of an effect scope are handled all at once
- Check that an effect scope does not escape its runscope

Miro - Type safety

Theorem 4 (Type safety)**.**

$$\text{if } (\cdot;\cdot \vdash c : \tau \,!\, \varnothing) \text{ and } (c \underset{\cdot}{\rightsquigarrow}^* c') \text{ then value}(c') \text{ or } (\exists c''.\ c' \underset{\cdot}{\rightsquigarrow} c'')$$

Lemma 6 (Preservation)**.**

$$\text{if } (\Delta;\Gamma \vdash c : \underline{\tau}) \text{ and } (c \underset{\cdot}{\rightsquigarrow} c') \text{ then } (\Delta;\Gamma \vdash c' : \underline{\tau})$$

Miro - Type safety issue

$$\mathsf{runscope}(s \to \mathsf{new\ State@}s\ \{h; \mathsf{finally}\ f \to f\ 0\}\ \mathsf{as}\ r\ \mathsf{in\ return}\ (\lambda\_.r' \leftarrow \mathsf{return}\ r;\ \mathsf{return}\ 42))$$

$$\rightsquigarrow \qquad \text{(S-RunscopeNew)}$$

$$\mathsf{runscope}^s(\mathsf{new\ State@}s\ \{h; \mathsf{finally}\ f \to f\ 0\}\ \mathsf{as}\ r\ \mathsf{in\ return}\ (\lambda\_.r' \leftarrow \mathsf{return}\ r;\ \mathsf{return}\ 42))$$

$$\rightsquigarrow \qquad \text{(S-RunscopeNew)}$$

$$\mathsf{runscope}^s(f \leftarrow \mathsf{runinst}^l_{s,\mathsf{State}}\{h\}(\mathsf{return}\ (\lambda\_.r' \leftarrow \mathsf{return\ inst}(l);\ \mathsf{return}\ 42));\ f\ 0)$$

$$\rightsquigarrow \qquad \text{(S-RunscopeCong, S-Seq, S-RuninstReturn)}$$

$$\mathsf{runscope}^s(f \leftarrow \mathsf{return}\ (\lambda st.\mathsf{return}\ (\lambda\_.r' \leftarrow \mathsf{return\ inst}(l);\ \mathsf{return}\ 42));\ f\ 0)$$

Conclusion

- Algebraic effects are great, but can be improved upon
- Miro - safely combine algebraic effects and dynamic instances
- Proving type safety is difficult

More in the thesis

- More examples
- Formalization of algebraic effects and handlers with static instances
- Formalization of Miro, type system and a small-step operational semantics
- Mechanization of algebraic effects and handlers with static instances in Coq (with type safety proofs)
- Prototype implementation of Miro in Haskell

Questions

Any questions?