UNIVERSITY OF CAMBRIDGE

Graded monads in program analysis

Andrej Ivašković

Department of Computer Science and Technology
University of Cambridge

BCTCS 2020, 7 April

What this talk is about

- My research is about tying *program analysis* with writing programs in *functional languages*.
- I will first introduce you to program analysis and the significance of type systems in programming languages.
- I will go on to talk about writing programs using *monads* and *graded monads*. The examples of graded monads will demonstrate the relationship with program analysis.

Program analysis

- It is possible to infer some properties of programs and *reason about its correctness*.
- *Constant propagation* can simplify a sequence of assignments (and thus speed up code execution):

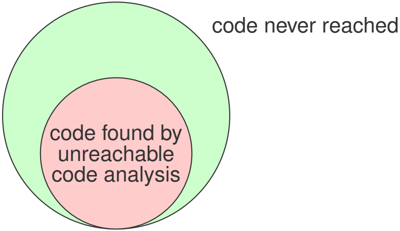
```
x := 1                 x := 2
y := x+5     ⟶        y := 6
x := y-4               z := 10
z := 2*x+y
```

- *Unreachable code analysis* can infer that some parts of the program can never be reached and executed:

```
x := 1                        ⟶   x := 1
if x = 1 then f()                  f()
         else g()
```

Program analysis in practice

- Most program analysis is implemented in compilers or in external static analysis tools (mainly for the purposes of optimisation, but also verification).
- Unfortunately, *Rice's theorem* roughly states that these tools cannot give you an exact answer – analysing semantic properties of programs is *undecidable*.
- Program analysis is necessarily a safe, *conservative overapproximation*.



Types in programming languages

- The purpose of *static type systems* is to constrain programs to catch out some kinds of errors that would otherwise appear.
- The compiler explicitly rejects programs that do not type check.
- For example, in most statically typed languages the following expression does not type check:

  ```
  if b then 42 else "foo"
  ```

  whereas in most functional languages this one does (provided `b` is a boolean):

  ```
  if b then 42 else 17
  ```

More interesting type systems

- A lot of work has been done on more powerful type systems, which tend to provide a lot more information about the data they are manipulating.
    - e.g. *dependent type systems*
    - e.g. programming using GADTs:

```
cons: a -> Vec n a -> Vec (S n) a
```

- **Can you encode a program analysis inside the type system itself?**

More interesting type systems

- A lot of work has been done on more powerful type systems, which tend to provide a lot more information about the data they are manipulating.
  - e.g. *dependent type systems*
  - e.g. programming using GADTs:

```
cons: a -> Vec n a -> Vec (S n) a
```

- **Can you encode a program analysis inside the type system itself?**
  - Yes!
  - Many different approaches, includes *effect systems*.

Monads and side effects

- *Side effects* in functions in programming languages make them *impure*, and running `f(x)` can give different results depending on the global program state.
- In functional programming languages, we don't like side effects – we want our functions to be pure and our language to be *referentially transparent*.
- Thus we use *monads* – these are *type constructors* that represent possibly impure computation.
  - For a monad T, if A is a type, then TA is the type of a computation that 'eventually returns' a value of type A.
  - An impure function of type $A \to B$ is typically represented as a function of type $A \to TB$.

Example: IO monad

- The most common impure effect is dealing with IO – Haskell does this via the `IO` monad. It is deeply magical.
- The main operations are `getLine :: IO String` and `putStrLn :: String -> IO ()`.
- Haskell provides the helpful `do` notation that is very convenient in this setting:

```
do putStrLn "What␣is␣your␣name?"
   name <- getLine
   putStrLn ("Welcome,␣" ++ name ++ "!")
```

- This is just syntactic sugar for the following (>>= composes IO actions):

```
putStrLn "What␣is␣your␣name?" >>= getLine
>>= \name -> putStrLn ("Welcome,␣" ++ name ++ "!")
```

Monad operations and laws

- A monad T is defined by two main operations:
  - `return` : $A \to TA$ for all types A
  - $\gg\!=$: $TA \to (A \to TB) \to TB$ for all types A and B, binary operator pronounced ‘bind‘, associates to the left
- The monad operations have to satisfy the following laws:
  - `do {x <- m; return x}` $\equiv$ `m` (identity 1)
  - `do {y <- return x; f y}` $\equiv$ `f x` (identity 2)
  - `do {y <- do {x <- m; f x}; g y}` $\equiv$ `do {x <- m; do {y <- f x; g y}}` (associativity)

Example: State monad

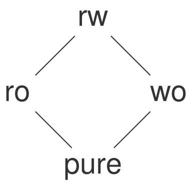
- For a type `s`, there is a monad `State s` representing computations that make use of a mutable variable of type `s`.
- In order to retrieve the result of a stateful computation, you need to use `runState :: State s a -> s -> (a, s)`.
- To read and write from the mutable variable, use `get :: State s s` and `put :: s -> State s ()`
- Example:

```
runState (do x <- get
             put (2 * x + 5)
             y <- get
             return (y - 1))
         3
```

Graded monads: core idea

- The type constructor used in a monad does not provide a lot of information other than that the computation is potentially impure.
- **Key idea:** what if the monad carries an 'annotation'?
- The type constructor are now be of the form T^r, where r is drawn from some *grading algebra*.
- There are still be $\gg=$ and `return` operations, but it is not the case that every T^r is a monad – instead, the new 'graded bind' combines the annotations.

Example: graded State monad with permissions



`return` should carry the pure annotation

$\gg=$ has to consider lub ($\wedge$)

- The elements of the algebra represent the permissions on the mutable variable (‘cannot do anything’, ‘read only’, ‘write only’, ‘read and write’).
- In Haskell, this structure has type `GState s g a`.
- The types of `get` and `put` change: `get :: GState s ro s` and `put :: s -> GState s wo ()`

Graded monad operations

- Given a grading algebra $(E, \cdot, i)$, which is at least a monoid, a graded monad is a family of type constructors $\{T^r | r \in E\}$ along with the following operations:
  - `return` : $A \to T^i A$ for all types A
  - $\gg\!=^{r,s} : T^r A \to (A \to T^s B) \to T^{r \cdot s} B$ for all types A and B and all $r, s \in E$
- Typically, to allow for *subtyping*, we also assume that the monoid is pre-ordered: $((E, \leq), \cdot, i)$.
- Details swept under the rug.

Example: live variable analysis

- We now turn to *live variables* in the `GState s` graded monad. First we consider the case when there is only one mutable variable.

- A variable is *live* at a program point if its ‘current value’ might be used during computation. Otherwise it is *dead*. For example, in

  ```
  do {t <- get; put (t + 1); e}
  ```

  the variable is live at the start.

- We want `GState s f a` to somehow provide information about live variables at ‘the start’ of an expression of type `a`.

Example: live variable analysis (cont'd)

- For every expression there is a *transfer function*: a map from the set of live variables 'just after' the expression to the set of live variables 'just before'. For example, the transfer function for `put` is $\lambda l.\, l \setminus \{x\}$
- The grading algebra is the algebra of transfer functions, with $\cdot$ being function composition and the function $\lambda l.\, l$ as the identity.
- Then the type of `get` is `GState s` $(\lambda l.\, l \cup \{x\})$ `s` and the type of `put` is `s -> GState s` $(\lambda l.\, l \setminus \{x\})$ `()`.
- For an expression of type `GState s f a`, the set of live variables at the starting program point is $\mathtt{f}(\emptyset)$.
- This generalises to multiple variables (easiest approach is with monad transformers).

Conclusions

- These are just some analyses which can be represented as graded monads.
  - **Cut for time:** deadlock-free concurrency is possible.
- The overall point of this exercise is to represent program analyses as type inference or type checking – inside Haskell's type system.
- This opens up potential for the programmer to write their own program analysis without using an external tool or modifying the compiler.
  - **Cut for time:** if the grading algebra is a finite lattice satisfying certain monotonicity properties, the type inference algorithm is simple and guaranteed to terminate in a reasonable amount of time.

Summary

- Monads are a way to write effectful programs in functional languages.
- With the right choice of grading algebra, we can represent program analyses.
- Therefore program analysis can sometimes be represented as type checking or type inference.