Functional Programming

Expression Evaluation

- conceptually, you can consider Haskell runtime as executing a loop which
  - searches for a function call in the current expression
  - searches for a matching equation for the function
  - sets values of variables in matching pattern to corresponding arguments
  - replaces LHS of equation with RHS
- loop terminates when current expression contains no function calls
- what order should be chosen for rewriting?
  - Church-Rosser theorem: order doesn't matter for final value
  - does matter for efficiency

Church-Rosser Theorem

- for rewriting system of lambda calculus, regardless of the order in which the original term's subterms are rewritten, final result is always the same
- Haskell is based on variant of lambda calculus, so the theorem holds
- **not** applicable to imperative languages

Referential transparency

- **referential transparency:** expression can be replaced with its value
  - requires expression has no side effects and is pure: must return same results on the same input
- **impure functional language:** e.g. Lisp, permits side effects like assignment so programs are not referentially transparent

Single Assignment

- imperative/OO languages: variable has current value, which is mutable
- functional languages: variables are **single assignment**
  - no assignment statements
  - immutable: can define variable's value, but cannot redefine it

Haskell type system

- type system is **strong, safe, static**
- strength refers to how permissive a type system is, with a stronger type system accepting fewer expressions as valid than a weaker one
- **strong**: type system guarantees a program cannot errors from trying to write expressions that don't make sense
  - no loopholes: cannot make an integer a pointer
    - C: `(char *)42`
- **safe:** running program will never crash due to a type error
- **static:** types are checked when program is compiled
  - c.f. dynamic: types are checked when program is run
  - safe follows partially from static
- types can be automatically **inferred**

Type classes

- a type in `Ord` must also be in `Eq`

Disjunction and conjunction

```haskell
data Suit = Club | Diamond | Heart | Spade
data Card = Card Suit Rank
```

- **enumerated type:** value of type `Suit` is **either** `Club` or `Diamond` …
  - disjunction of values
- **structure type:** value of type `Card` contains a value of type `Suit` **and** a value of type `Rank`
  - conjunction of values
- most imperative languages permit types as disjunction or conjunction, but not both at once
- Haskell doesn't have this limitation

Discriminated Union Types

- **discriminated union types:** can include both disjunction and conjunction

  - in C, you could create a similar union, but wouldn't be able to determine which field was applicable
  - in Haksell, data constructor tells you, hence **discriminated**

- **algebraic type system:** permits combination of disjunction + conjunction
  - **algebraic types:** types produced under algebraic type system

```haskell
data JokerColor = Red | Black
data JCard = NormalCard Suit Rank | JokerCard JokerColor
```

- value of `JCard` constructed
  - either using `NormalCard` constructor, containing a value of type `Suit` and a value of type `Rank`
  - or using `JokerCard` constructor, containing a value of type `JokerColor`

Representing Expressions in Haskell

```haskell
data Expr
    = Number Int
    | Variable String
    | Binop Binopr Expr Expr
    | Unop Unopr Expr

data Binopr = Plus | Minus | Times | Divide
data Unopr  = Negate
```

- very direct, much shorter than C/Java implementation, no comments required

Errors

The C implementation is error prone:

- able to access fields that aren't meaningful
  - caught by Haskell, Java compiler
- can forget to initialise fields
  - caught by Haskell compiler
  - not caught by Java
- can forget to process some alternatives

  - caught by Java
  - can be caught by Haskell (with particular flags)

Memory

- C: requires 8 words per expression
- Java/Haskell: maximum of 4
  - can be **more** efficient than a C program

Maintenance

- adding a new expression:
  - Java: add new class
    * implement all methods
  - C: add new alternative to enum
    * add needed members to the type
    * add code for it to all functions handling that type
  - Haskell: add new alternative to the type
    * add code to all functions handling that type
- adding a new operation for expressions
  - Java: add new method to abstract `Expr` class
    * implement it for all classes
  - C: write one new function
  - Haskell: write one new function

Non-Exhaustive Patterns

- Haskell: Detect with `-fwarn-incomplete-patterns`
  - if not handled, will throw an exception
- C: without default case program may continue and silently compute incorrect result
  - requires more implementation of default cases
- Java: forgetting to write a method for subclass will probably inherit the wrong behaviour of the superclass