Why some people use functional languages

Philip Wadler
University of Edinburgh / IOHK
Loft, São Paulo
27 November 2019
Haskell
Use Case 1:  
Haxl @ Facebook
blog :: Fetch Html
blog = renderPage <$> leftPane <*> mainPane

leftPane :: Fetch Html
leftPane = renderSidePane <$> popularPosts <*> topics

mainPane :: Fetch Html
mainPane = do
  posts <- getAllPostsInfo
  let ordered =
      take 5 $ 
      sortBy (flip (comparing postDate)) posts
  content <- mapM (getPostContent . postId) ordered
  return $ renderPosts (zip ordered content)

popularPosts :: Fetch Html
popularPosts = do
  pids <- getPostIds
  views <- mapM getPostViews pids
  let ordered =
      take 5 $ map fst $ 
      sortBy (flip (comparing snd))
      (zip pids views)
  content <- mapM getPostDetails ordered
  return $ renderPostList content
Figure 1. Data fetching in the blog example
Use Case 2:
SeL4
Figure 1. Specification layers in the L4.verified project.
schedule ≡ do
    threads ← all_active_tcbs;
    thread ← select threads;
    switch_to_thread thread
    od OR switch_to_idle_thread

Figure 3: Isabelle/HOL code for scheduler at abstract level.
schedule = do
  action <- getSchedulerAction
  case action of
    ChooseNewThread -> do
      chooseThread
      setSchedulerAction ResumeCurrentThread
      ...
    chooseThread = do
      r <- findM chooseThread' (reverse [minBound .. maxBound])
      when (r == Nothing) $ switchToIdleThread
    chooseThread' prio = do
      q <- getQueue prio
      liftM isJust $ findM chooseThread'' q
    chooseThread'' thread = do
      runnable <- isRunnable thread
      if not runnable then do
        tcbSchedDequeue thread
        return False
      else do
        switchToThread thread
        return True

Figure 4: Haskell code for schedule.
void setPriority(tcb_t *tptr, prio_t prio) {
    prio_t oldprio;
    if(thread_state_get_tcbQueued(tptr->tcbState)) {
        oldprio = tptr->tcbPriority;
        ksReadyQueues[oldprio] =
            tcbSchedDequeue(tptr, ksReadyQueues[oldprio]);
        if(isRunnable(tptr)) {
            ksReadyQueues[prio] =
                tcbSchedEnqueue(tptr, ksReadyQueues[prio]);
        }
    else {
        thread_state_ptr_set_tcbQueued(&tptr->tcbState,
            false);
    }
    }
    tptr->tcbPriority = prio;
}

Figure 5: C code for part of the scheduler.
Use Case 3: Finance
module Incremental : sig
    type 'a t

    val return : 'a -> 'a t

    val map : 'a t -> f:('a -> 'b) -> 'b t
    val map2 : 'a t -> 'b t -> f:('a -> 'b -> 'c) -> 'c t
    val bind : 'a t -> ('a -> 'b t) -> 'b t

    val stabilize : unit -> unit

    val value : 'a t -> 'a
    val on_update : 'a t -> ('a -> unit) -> unit
end

module Variable : sig
    type 'a t

    val create : 'a -> 'a t
    val set : 'a t -> 'a -> unit
    val read : 'a t -> 'a Incremental.t
Concurrency
Transistor count still rising
Clock speed flattening sharply
Source: Intel
Agda for Fun and Profit: System F
# Plutus Core

## Kinds

<table>
<thead>
<tr>
<th>J, K ::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
</tr>
<tr>
<td>J \to K</td>
</tr>
</tbody>
</table>

## Types

<table>
<thead>
<tr>
<th>A,B ::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
</tr>
<tr>
<td>A \to B</td>
</tr>
<tr>
<td>\forall X.B</td>
</tr>
<tr>
<td>\mu X.B</td>
</tr>
<tr>
<td>\rho</td>
</tr>
</tbody>
</table>

## Terms

<table>
<thead>
<tr>
<th>L,M,N ::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>\lambda x:A.N</td>
</tr>
<tr>
<td>L M</td>
</tr>
<tr>
<td>\Lambda X:K.N</td>
</tr>
<tr>
<td>L A</td>
</tr>
<tr>
<td>wrap M</td>
</tr>
<tr>
<td>unwrap M</td>
</tr>
<tr>
<td>\rho</td>
</tr>
</tbody>
</table>
Plutus Core in Agda

```
data Kind : Set where
  *    : Kind
  _⇒_ : Kind → Kind → Kind

data _⊢_* : Ctx* → Kind → Set where
  `  : ⦃ J ⦄ ⊢ ⦃ J ⦄
       → ⦃ J ⦄

  λ  : ⦃ ⦄ , ⦃ K ⦄ ⊢ ⦃ J ⦄
       → ⦃ K ⦄ ⇒ ⦃ J ⦄

  _·_ : ⦃ K ⦄ ⊢ ⦃ J ⦄
       → ⦃ K ⦄ ⇒ ⦃ J ⦄

  Λ : ⦃ ⦄ , ⦃ K ⦄ ⊢ ⦃ B ⦄
       → ⦃ B ⦄

  _·_ : ⦃ ⦄ ⊢ ⦃ ⦄
       → ⦃ A ⦄ ≡ B

  conv : ⦃ A ⦄ ≡ B
       → ⦃ A ⦄
       → ⦃ B ⦄
```

```
Roman Kireev 3 months ago

I haven't talked with James except for a couple of messages, but I read what he wrote in Agda and I'm very surprised that you can formalize System F in a non-disgusting way. Or at least I do not see those huge clunky theorems which I see everywhere including my own attempts.
A Profound Pun
yoghurt \lor \neg\text{yoghurt}
Gilbert & Sullivan's
The Gondoliers
or, The King of Barataria

LIGHT OPERA THEATRE OF SACRAMENTO
A & B
A → B
A ∨ B
$A \& B$

$A \lor B$

$A \rightarrow B$
Lambda Calculus
Alonzo Church, 1932-40

Natural Deduction
Gerhard Gentzen, 1935
Principle Type Scheme

Roger Hindley, 1969

Type Polymorphism

Robin Milner, 1978
System F

Polymorphic Lambda Calculus

Jean-Yves Girard, 1972

John Reynolds, 1974
Object-Oriented Programming
Alternative

- F-bounded polymorphism
  \[ \text{max} : \forall t <: \text{greater} : t \rightarrow \text{bool}. t \Rightarrow t \rightarrow t \]

- Higher-order bounded polymorphism
  \[ \text{max} : \forall F <: \lambda t. \text{greater} : t \rightarrow \text{bool}. \mu F \Rightarrow \mu F \rightarrow \mu F \]

- Similar in spirit but technical differences
  - Transitive relation
  - “Standard” bounded quantification
### Translation of Types:

\[ |C| = C \quad |C.E| = C.E.Fix\]
\[ |X| = X \quad |X.E| = X.E \quad |E| = E\]

**Definition of `nestedclasses`**

\[ nestedclasses(Object) = \cdot \]

\[ \text{class } C \langle D \{ \ldots \vec{N} \} \rangle \]
\[ nestedclasses(D) = \vec{E'} \]
\[ nestedclasses(C) = \vec{E}', \{ E \mid E \notin \vec{E'}, \text{class } E \{ \ldots \} \in \vec{N} \} \]

**Translation of Type Arguments:**

\[ nestedclasses(C) = \vec{E} \]
\[ |F|_C = |F.E|, |F| \]

**Translation of Expressions:**

\[ |x|_{\Delta, \Gamma, A} = x \]
\[ |e_0.f_i|_{\Delta, \Gamma, A} = |e_0|_{\Delta, \Gamma, A}.f_i \]
\[ \Delta; \Gamma; A \vdash e_0 : T_0 \]
\[ mtype_{P_3}(m, bound_{\Delta}(T_0 \otimes A)) \]
\[ = |\langle x \upharpoonright C \rangle \cup \mathcal{U} \upharpoonright U_0| \]
\[ |e_0.m(\vec{E})|_{\Delta, \Gamma, A} \]
\[ = |e_0|_{\Delta, \Gamma, A}.|\vec{E}|_{\Delta, \Gamma, A} \]
\[ \text{new } A_0(\vec{E})|_{\Delta, \Gamma, A} = \text{new } A_0(|\vec{E}|_{\Delta, \Gamma, A}) \]

**Definition of Ceiling:**

\[ |C.E| = \begin{cases} 
    C.E & \text{if class } E\{ \ldots \} \in \vec{N} \\
    |D.E| & \text{otherwise}
\end{cases} \]

where class \( C \langle D \{ \ldots . \vec{N} \} \rangle \)

### Translation of Methods:

\[ nestedclasses(C) = \vec{E} \]
\[ |x<C| = |x.E_1| \triangleleft [C.E_1] \triangleleft |x.E|, \ldots \\
\[ |x.E_n| \triangleleft [C.E_n] \triangleleft |x.E|, x<C \]
\[ \Gamma = \vec{x} : \vec{T}, \text{this: } \text{thisType}(\Delta) \quad \Delta = \vec{x}<\vec{C} \]
\[ |\langle x \upharpoonright C \rangle |_{T_0} m(\vec{T} \vec{x}) \{ \uparrow e_0; \} |_{\Delta} \]
\[ = |\langle x \upharpoonright C \rangle |_{T_0} m(\vec{T} \vec{x}) \{ \uparrow \bullet |_{\Delta, \Gamma, A}; \} \]

**Translation of Classes:**

\[ nestedclasses(C) = \vec{E} \]
\[ |C| = |C.E| \triangleleft [C.E] \triangleleft |\vec{E}|, \ldots \\
\[ |E_n| \triangleleft [C.E_n] \triangleleft |\vec{E}| \]

\[ \text{class } C \langle D \{ \ldots \} \rangle \quad \text{nestedclasses}(C) = \vec{E} \]
\[ \text{nestedclasses}(D) = \vec{E'} \]
\[ \text{class } i \ C.E_i \langle \langle C \rangle \rangle \langle \langle D.E_i \rangle \rangle \langle \langle \vec{E} \rangle \rangle \{ \\
\[ |T| \upharpoonright f; \vec{M}_{C.E_i} \} \]

\[ \text{class } C \langle D \{ \ldots \} \rangle \quad \text{nestedclasses}(C) = \vec{E} \]
\[ \text{nestedclasses}(D) = \vec{E'} \quad \vec{E} \notin \vec{E'} \]
\[ \text{class } i \ C.E_i \langle \langle C \rangle \rangle \langle \langle D.E_i \rangle \rangle \langle \langle \vec{E} \rangle \rangle \{ \\
\[ |T| \upharpoonright f; \vec{M}_{C.E_i} \}
\]

**Definition of Ceiling:**

\[ \text{fix}(C, E) = \text{class } |C.E| \langle |C.E| \rangle \langle |C.E| \rangle \{ \\
\[ |C.E| \langle \langle C.E \rangle \rangle \langle \langle C.E \rangle \rangle \{ \\
\[ \text{class } C \langle D \{ |T| \upharpoonright f; \vec{M}_{C.E_i} \}, \vec{N} \} \}
\]
\[ \text{fix}(C, \text{nestedclasses}(C)) \]
### Class Types

\[ C \triangleq \{ c \mid c \text{ is declared in } CT \} \]

\[ \tau^c(t_{\text{obj}}) \triangleq \prod_{c' \in F^c} t_{\text{obj}} \]

### Method Types

\[ M \triangleq \{ m \mid m \text{ is declared in } CT \} \]

\[ \cup \{ \text{get}[f] \mid f \in F \} \]

\[ \cup \{ \text{cast}[c] \mid c \in C \} \]

\[ \tau_m(t_{\text{obj}}) \triangleq \tau'_m(t_{\text{obj}}) \text{ cmd} \]

where

\[ \tau'_m(t_{\text{obj}}) \triangleq \tau^\text{arg}(t_{\text{obj}}) \rightarrow (t_{\text{obj}} \text{ cmd}) \]

\[ \tau'_{\text{get}[f]}(t_{\text{obj}}) \triangleq t_{\text{obj}} \]

\[ \tau'_{\text{cast}[c']}(t_{\text{obj}}) \triangleq t_{\text{obj}} \]

### Figure 1: Compiling to a typed intermediate language

\[ \tau^c \triangleq \forall t_{\text{obj}} \cdot \tau^{cv}(t_{\text{obj}}) \rightarrow \tau^{mv}(t_{\text{obj}}) \rightarrow \tau^c(t_{\text{obj}}) \rightarrow \tau_m(t_{\text{obj}}) \]
Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch
That leads us to our second principle of object-oriented design:

*Favor object composition over class inheritance.*

Ideally, you shouldn’t have to create new components to achieve reuse. You should be able to get all the functionality you need just by assembling existing components through object composition. But this is rarely the case, because the set of available components is never quite rich enough in practice. Reuse by inheritance makes it easier to make new components that can be composed with old ones. Inheritance and object composition thus work together.

Nevertheless, our experience is that designers overuse inheritance as a reuse technique, and designs are often made more reusable (and simpler) by depending more on object composition. You’ll see object composition applied again and again in the design patterns.
User community
LAMBDA WORLD
CÁDIZ
by 47 Degrees

OCTOBER 17TH
CADIZ 18TH
2019 SPAIN
lambda DALS
13-14 FEBRUARY 2020
KRAKÓW | POLAND
Conclusions
Propositions as Types

By Philip Wadler
Communications of the ACM, December 2015, Vol. 58 No. 12, Pages 75-84
10.1145/2699407
Comments (1)

Powerful insights arise from linking two fields of study previously thought separate. Examples include Descartes's coordinates, which links geometry to algebra, Planck's Quantum Theory, which links particles to waves, and Shannon's Information Theory, which links thermodynamics to communication. Such a synthesis is offered by the principle of Propositions as Types, which links logic to computation. At first sight it appears to be a simple coincidence—almost a pun—but it turns out to be remarkably robust, inspiring the design of automated proof assistants and programming languages, and continuing to influence the forefronts of computing.
Propositions as Types
Philip Wadler
University of Edinburgh
Strange Loop
St Louis, 25 August 2015

"Propositions as Types" by Philip Wadler
61,321 views
I just proved commutativity of multiplication in Agda and got way too much serotonin out of it. 😊

Programming Language Foundations in Agda is AMAZING. Check it out at plfa.github.io.

Thank you, Phil Wadler and @wenkokke.

(PS: If you have a better proof, let me know!)

```
{-#.comm : (m n : N) -> m * n ≡ n * m
{-#.comm zero n
   rewrite #≡-absorption n = refl
{-#.comm m zero
   rewrite #≡-absorption m = refl
{-#.comm (suc m') (suc n')
   rewrite #≡-comm m' (suc n')
   | sym (+-assoc n' m' (n' * m'))
   | #≡-comm n' m'
   | +-comm n' m'
   | #≡-comm n' (suc m')
   | #≡-assoc m' n' (m' * n')
   = refl
```

10:35 AM - 16 Oct 2018

14 Likes
http://plfa.inf.ed.ac.uk
https://github.com/plfa

Or search for “Kokke Wadler”

Please send your comments and pull requests!