Tangible Functional Programming

Conal Elliott

ICFP 2007
ICFP 2007, Freiburg
Conal Elliott
Tangible Functional
Programming

Added TV of type ([Char],[Char]) -> [Char]
Programming favors left-brain creativity.

- Abstract & linguistic
- Usually sequential
- Selects & influences creative processes
- Powerful medium of expression
- with limited access
Can functional programming be artist-friendly?

- Non-sequential
- Still abstract & linguistic
- “Authoring”: concrete & non-composable
- Goal: concrete and composable
- So artists can make their own tools
The insight:

Authoring *is* functional programming.

- In disguise
- Full of interpreted graphs
- Lacks reuse & parameterization
- Scripting bolted on
Programming is a way to express interfaces and denotations.

- Code is a command-line UI.
- Handy & inessential
- Necessarily indirect
Where are we going?

- Eros user experience
- $\lambda$-mechanics
Key idea #1 (of 4):

Use GUIs to visualize typed values.

- GUI *structure* follows type.
- GUI *content* presents value.
- Functions visualize as *interactive* GUIs.
- “Tangible values”
Base type values are widgets.
Pairs lay out horizontally.

“,” in \((\alpha, \beta)\) and \((a,b)\)
Functions lay out vertically.

“→” in \( \alpha \to \beta \) and \( \lambda \ a \to b \)
Functions may be curried or uncurried.
Functions visualize as *interactive* GUIs.
Key idea #2:

Users make new TVs by *fusion*.

- Select compatible input & output,
- which disappear.
- Everything else remains,
- fused into a single new TV.
TV fusion subsumes function \textit{application}.
TV fusion subsumes function *composition*.
Fusion may reach into nested inputs.

$R \rightarrow (R, R) \rightarrow \text{Bool}$  
$R \rightarrow R$  
$R \rightarrow R \rightarrow R \rightarrow \text{Bool}$
Let's take a look.
Where are we?

- Eros user experience
- $\lambda$-mechanics
Key idea #3:

Keep visualization & value combined and separable.

```haskell
type TV a = (Out a, a)
```

- Operate on both parts in tandem
- Combined for convenience
- Separable for composability
Visualizations *assemble*

as types and values do.

type Out a = ...
put    :: Put a -> Out a
opair :: Out a -> Out b -> Out (a, b)
olambda :: In a -> Out b -> Out (a->b)

type In a = ...
get    :: Get a -> In a
ipair :: In a -> In b -> In (a,b)
Key idea #4:
Translate gestural fusion to combinator sequences.

- “Deep application”. Reaches buried
- arguments,
- functions, and
- inputs.

- Define for values & extend to TVs.
We already have the tools to aim functions at buried arguments.

```
first :: (a -> a') -> ((a, b) -> (a', b))
second :: (b -> b') -> ((a, b) -> (a, b'))
result :: (b -> b') -> ((a->b) -> (a->b'))

first     f = \ (a, b) -> (f a, b)
second    g = \ (a, b) -> (a, g b)
result    g = \ f -> g . f
```
Compositions describe type paths to edit *deeply* buried arguments.

\[
\text{sf} :: (b \to b') \to (a, (b, c)) \\
\to (a, (b', c)) \\
sf = \text{second.first}
\]

\[
\text{frsrfrf} :: (c \to c') \to (a \to (f, b \to (c, g)), e) \\
\to (a \to (f, b \to (c', g)), e) \\
\text{frsrfrf} = \text{first.result.second.result.first}
\]
A similar game reaches buried functions.

funFirst ::
(d -> (c->a)) -> ((d,b) -> (c->(a,b)))

- Promotes a function extractor
- Similarly, funSecond, funResult
- Form type paths, as before.
The final combinators reach buried *inputs*.
These tools generalize.

- first and second work on arrows.
- Add DeepArrow subclass & instances for visualizations & pairings,
- types, code, etc.
Functional programming can be artist-friendly.

- Use GUIs to visualize typed values.
- Users make new TVs by fusion.
- Viz & value combined and separable.
- Gestural fusion via combinator sequences.
To explore

- Tangible polymorphism?
- Direct structural tweaks
- Symmetric In/Out (ilambda)
- “GUIs are types” as GUI design guide
- TVs as composable MVC