Tangible Functional Programming

Conal M. Elliott
LambdaPix
conal@conal.net

Abstract

We present a user-friendly approach to unifying program creation and execution, based on a notion of "tangible values" (*TVs*), which are visual and interactive manifestations of pure values, including functions. Programming happens by gestural composition of TVs. Our goal is to give end-users the ability to create parameterized, composable content without imposing the usual abstract and linguistic working style of programmers. We hope that such a system will put the essence of programming into the hands of many more people, and in particular people with artistic/visual creative style.

In realizing this vision, we develop algebras for visual presentation and for "deep" function application, where function and argument may both be nested within a structure of tuples, functions, etc. Composition gestures are translated into chains of combinators that act simultaneously on statically typed values and their visualizations.

Categories and Subject Descriptors D.2.6 [*SOFTWARE ENGINEERING*]: Programming Environments—graphical environments, interactive environments

General Terms Human Factors, Languages, Theory

Keywords interactive programming, end-user programming, gestural composition, combinator libraries, arrows, interactive visualization

1. Introduction

The activities of *creating* and *executing* interactive programs typically differ greatly in form and thought process. Program creation (including modification) happens in an abstract and linguistic setting, while program execution is a (relatively) concrete and visual experience.

Suppose *users* of interactive programs could also create such programs with a simple extension of their current style of interaction. First, such a development would enable many more people to create and share computational content. Second, it would allow this content to be created without imposing the abstract, linguistic mode of creativity. This freedom may give birth to new kinds of programs whose creation is nurtured by a concrete and visual environment.

This paper presents an approach to unifying program creation and execution, based on a notion of "tangible values" (*TVs*), which serve two roles. First, they allow interactive inspection of values (including functions). Second, they enable composition, to create new TVs. In this way, end-users become programmers, without the usual division between (a) the run-time world of visual interaction, and (b) the compile-time world of syntax. Users run *and create* (functional) programs by interacting with the same visual representation. We have implemented this idea of TVs in a system called Eros.

The work described in this paper makes the following contributions:

- An algebra of *interactive visualizers* for presenting typed values. In contrast to Haskell's *show* function, which produces strings and is not generally useful for functions, our approach produces GUIs and is *especially* useful for functions. TVs are formulated by simply pairing values and visualizers, combined for convenience and separable for composability.
- A new approach to end-user functional programming. Users create and compose functional programs *gesturally* (purely without syntax), by working directly with the same concrete visualizations used for interactive inspection. Specifically, a user selects compatibly-typed input and output widgets, typically in different TVs. The result is a *fusion* of the two source TVs, containing all of the original inputs and outputs *except* for the connected input and output. Due to the structured nature of visualizers and TVs, higher-order programming is fully supported.
- A new functional programming style, in which a buried function can be applied to any compatibly typed buried value. This programming style, which we call "deep application", subsumes function application and function composition. If the function's domain has a structured type (say nested pairs), it may be applied to a portion of a domain value, yielding a residual function. Deep application is based on a set of combinators, used in a simple way, to describe three paths in the type structure leading to (a) the buried function, (b) the relevant portion of that function's domain value, and (c) to the argument value.
- Generalization of these techniques to apply beyond functions, via a new *Arrow* subclass. We formulate gestural composition in this general setting, and specialize to tangible values.

The next two sections demonstrate the user's experience of visualizing and composing TVs. Section 4 then presents the deep application combinators and their use, as applied to standard functions. To prepare for the gestural setting, the tools are then generalized in Section 5 with a new "*DeepArrow*" type class. Section 6 briefly addresses the mechanics of translating a user's gestures into (invisible) application of the deep application combinators. Section 7 describes support for persistence and (run-time) compilation, via a data type and corresponding arrow instance that fits it into our framework. We conclude with a discussion of related and future work.

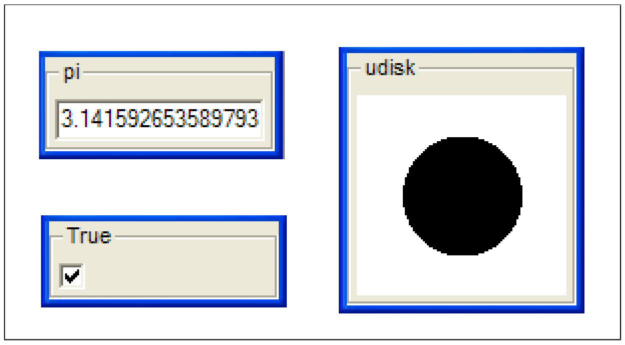


Figure 1. Simple, non-interactive visualizations

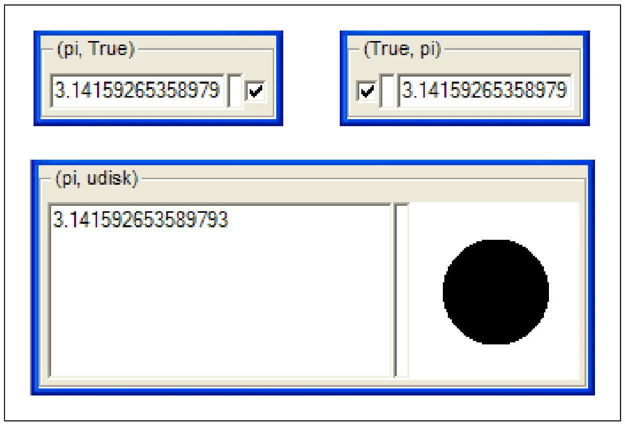


Figure 2. Composite visualizations

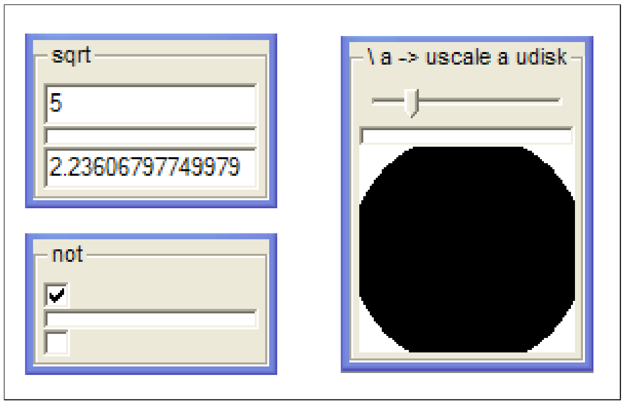


Figure 3. Functions (input above output)

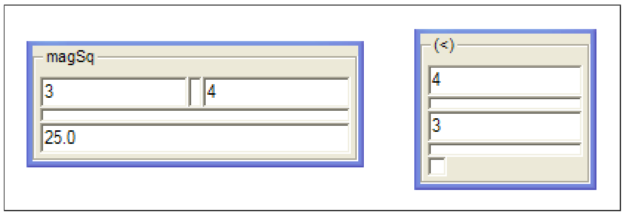


Figure 4. Multiple arguments via pairing or currying

2. Interactive Visualization of Values

The standard Haskell library comes with a *Show* type class for rendering values into strings. Class instances exist for many types, including numbers and booleans, and for type constructors, including tuples and lists. Additional instances may be defined or, for algebraic data types, derived automatically. Notably missing, however, is a means of rendering *functions*. Moreover, some types are rendered more naturally *visually*, e.g., graphs, structured 2D or 3D graphics, some functions, and formatted text.

2.1 GUIs as visualizations

Functions and visual presentation can both be supported by rendering values into (possibly interactive) GUIs. In their simplest form, these GUIs may have a single widget, as in Figure 1, which shows simple non-interactive visualizations of pi, $True$, and the unit disk.

Visualizations may also be constructed out of simpler ones, as in Figure 2. In those examples, pairs are rendered as horizontal juxtaposed visualizations separated by a thin vertical space. (The purpose of the space is explained below.)

To visualize functions in general, a simple technique suffices: create an interactive presentation that allows the user to sample the function dynamically, i.e., vary input values interactively, while watching the corresponding output changes. By visualizing a function, we mean "function" in a semantic sense, as a mapping (usually infinite), rather than any sort of syntax ("code") for the function. Figure 3 shows some function visualizations. Each take the form of an input for the function's domain stacked above a visualization for the function's range. Between input and output is a thin horizontal space. A user varies the input values and immediately sees the result of applying the function to the new value.

Functions of more than one argument may be handled in the usual two ways, via tupled arguments or currying. Both variants are shown in Figure 4, where[1]

$$magSq :: R^2 \rightarrow R$$
$$magSq\ (x, y) = x^2 + y^2$$

For $magSq$, the pair-valued input is composed out of two inputs. For $(<) :: R \rightarrow R \rightarrow Bool$, the visual composition is a single real-valued input above a single function-valued output, which consists of a real-valued input above a $Bool$-valued output.

The structure of a visualization directly reflects the structure of the *type* of the visualized value. In other words, the GUI's *structure* is a *type* visualization, i.e., "visual syntax" for a type. In Figure 1, the types are all atomic ($double$, $Bool$, and $Region$), so each visualization is atomic (i.e., contains a single widget). In Figure 2, the types are pairs of atomic types, so the visualizations contain two widgets horizontally abutted and connected by a vertical space, which refers to the pair itself. That vertical space is the visual counterpart for the comma in a pair type's textual syntax. Similarly, in Figure 3, the types are functions from one atomic type to another, so the visualizations again contain two widgets, *vertically* abutted and connected by a horizontal space, corresponding to the arrow in textual syntax. Just as arrow is right-associative, vertical stacking is "bottom-associative", as in the example on the right in Figure 4.

In addition to inspecting values, a user may hover over any piece of visual structure to see the corresponding type's textual syntax. Figure 5 shows atomic types corresponding to atomic sub-widgets, while Figure 6 shows composite types corresponding to composite widgets. (Note the down-associativity.) The value presented in Figures 5 and 6 is $\lambda a\ b \rightarrow magSq\ b < a^2$.

[1] For brevity, we use the following type synonyms throughout this paper:

$$\textbf{type}\ R \ = Double$$
$$\textbf{type}\ R^2 = (R, R)$$

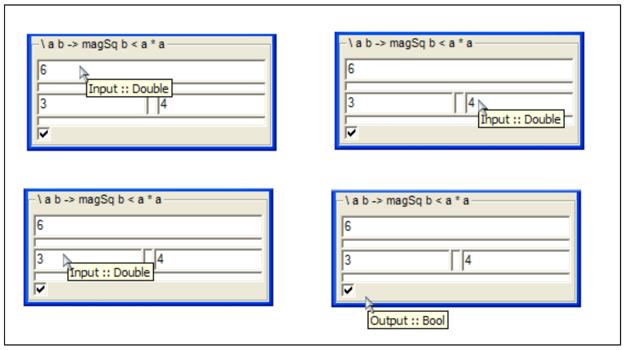


Figure 5. Atomic inputs and outputs

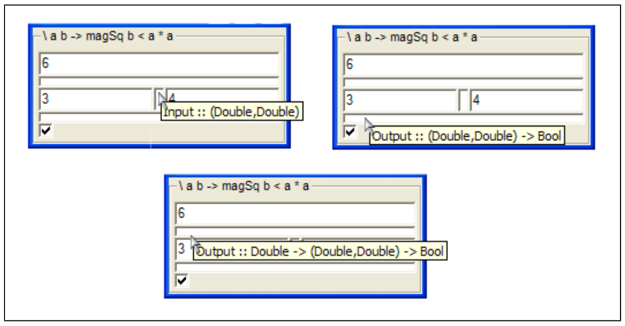

\ a b -> magSq b < a * a
6
3
4
Input :: (Double,Double)
Output :: (Double,Double) -> Bool
Output :: Double -> (Double,Double) -> Bool

Figure 6. Composite inputs and outputs

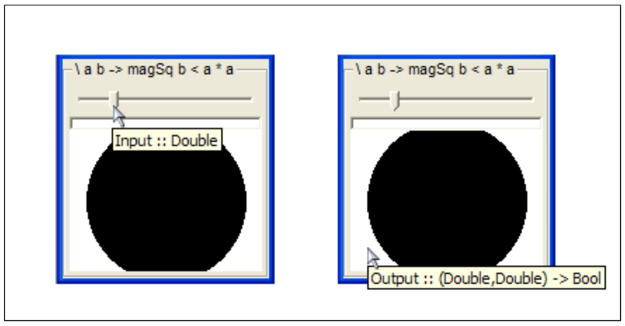


Figure 7. Example with slider and graphic

The value shown in Figures 5 and 6 may also be visualized quite differently, as in Figure 7. This version is based on modeling spatial regions as $R^2 \to \mathit{Bool}$, as in (Hudak and Jones 1994; Elliott 2003). This visualization reveals that the function describes a disk, with the first argument determining the radius. Points inside the region (mapped to True by the function) are painted black.

2.2 Inputs and Outputs

To give the terms "visualization", "input", and "output" precise meaning, we define two type constructors, Output and Input. Values of type $\mathit{Output}\ \alpha$, which we also call "α visualizers", describe how to present values of type α to a user. Values of type $\mathit{Input}\ \alpha$ describe how to get values of type α from a user.[2]

[2] The reason we package interactive visualizers via type constructors instead of type classes (as *Show*) is to allow more than one way to input or output a single type. For convenience, Eros also provides classes to assign *default* inputs and outputs to types, such as as those used in Figure 4.

$$
\begin{array}{llll}
\mathit{iGet} & :: \mathit{WGet}\ a & & \to \mathit{Input}\ \ a \\
\mathit{iPair} & :: \mathit{Input}\ \ a \to \mathit{Input}\ b & & \to \mathit{Input}\ \ (a\ ,\ b) \\
\mathit{oPut} & :: \mathit{WPut}\ a & & \to \mathit{Output}\ \ a \\
\mathit{oPair} & :: \mathit{Output}\ a \to \mathit{Output}\ b & & \to \mathit{Output}\ (a\ ,\ b) \\
\mathit{oLambda} & :: \mathit{Input}\ \ a \to \mathit{Output}\ b & & \to \mathit{Output}\ (a \to b)
\end{array}
$$

Figure 8. The visualizer algebra

The visualizations shown so far are all generated by outputs composed via the visualizer algebra shown in Figure 8. The types $\mathit{WPut}\ \ a$ and $\mathit{WGet}\ \ a$ are functions that take a container subwindow and add some number of widgets to present values to or get values from a user.

For example, the visualization in Figures 5 and 6 is described by o_1, where

$$
\begin{array}{l}
o_1 :: \mathit{Output}\ (R \to R^2 \to \mathit{Bool}) \\
o_1 = \mathit{oLambda}\ \mathit{iRead} \\
\qquad\quad (\mathit{oLambda}\ (\mathit{iPair}\ \mathit{iRead}\ \mathit{iRead})\ \mathit{oCheck})
\end{array}
$$

$$
\begin{array}{l}
\mathit{iRead}\ \ \ = \mathit{iGet}\ \mathit{readWGet} \\
\mathit{oCheck} = \mathit{oPut}\ \mathit{checkBoxWPut}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{readWGet} & :: \mathit{Read}\ a \Rightarrow \mathit{WGet}\ a \\
\mathit{checkBoxWPut} & :: \mathit{WPut}\ \mathit{Bool}
\end{array}
$$

Note that in the definition of o_1 there are four inputs (three iRead and one iPair) and three outputs (two $\mathit{oLambda}$ and one oCheck), as reflected in Figures 5 and 6.

A single value can be visualized in different ways. For instance, the value displayed with o_1 in Figures 5 and 6 may also be displayed as in Figure 7, using o_2, where

$$
\mathbf{type}\ \mathit{Region} = R^2 \to \mathit{Bool}
$$

$$
\begin{array}{l}
o_2 :: \mathit{Output}\ (R \to \mathit{Region}) \\
o_2 = \mathit{oLambda}\ \mathit{iSlider}\ \mathit{oRegion}
\end{array}
$$

$$
\begin{array}{l}
\mathit{iSlider}\ \ = \mathit{iGet}\ \mathit{sliderWGet} \\
\mathit{oRegion} = \mathit{oPut}\ \mathit{regionWPut}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{sliderWGet} & :: \mathit{WGet}\ R \\
\mathit{regionWPut} & :: \mathit{WPut}\ \mathit{Region}
\end{array}
$$

2.3 Tangible Values

Each of the visual examples above combines a value with a means of visualizing that value. This combination, which we call a *tangible value* (or "*TV*"), is what the user inspects and, as described later, creates.

$$
\mathbf{data}\ \mathit{TV}\ a = \mathit{TV}\ (\mathit{Id}\ a)\ (\mathit{Output}\ a)
$$

The Id type constructor is just a wrapper around values:

$$
\mathbf{newtype}\ \mathit{Id}\ a = \mathit{Id}\ a
$$

This wrapper will make some later definitions a bit more uniform. More importantly, Id serves as a placeholder for an alternative in Section 7 that enables persistence and compilation. The Haskell expressions appearing at the top of TVs rely on that alternative. They appear in this paper for clarity but are probably not desirable for our target audience.

-- Pre-defined (in `Parts` menu)

$$\begin{array}{lll}
udisk, checker & :: & Region \\
uscale, rotate & :: & R \to Region \to Region \\
intersect & :: & Region \to Region \to Region
\end{array}$$

-- Defined below

$$\begin{array}{lll}
scaleDisk & :: & R \to Region \\
scaleChecker & :: & R \to Region \\
rotScaleChecker & :: & R \to R \to Region \\
intersectDisk & :: & R \to Region \to Region \\
diskChecker & :: & R \to R \to R \to Region
\end{array}$$

$$\begin{array}{lll}
scaleDisk\ ds & = & uscale\ ds\ udisk \\
scaleChecker\ cs & = & uscale\ cs\ checker \\
rotScaleChecker\ cs\ ang & = & rotate\ ang\ (scaleChecker\ cs) \\
intersectDisk\ ds\ reg & = & intersect\ (scaleDisk\ ds)\ reg \\
diskChecker\ cs\ ang\ ds & = & intersectDisk\ ds \\
 & & \quad (rotScaleChecker\ cs\ ang)
\end{array}$$

Figure 9. Region examples in conventional syntactic form

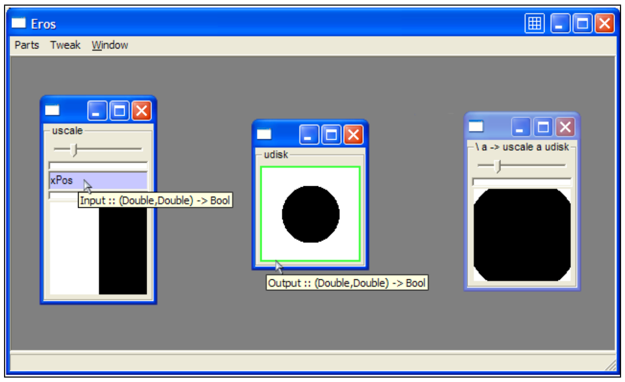

Eros
Parts Tweak Window
uscale
xPos
Input :: (Double,Double) -> Bool
udisk
Output :: (Double,Double) -> Bool
\ a -> uscale a udisk

Figure 10. Composing a scaling disk

3. Gestural Composition

TVs are not just a way to view values, but also to create them. As such, they serve as a "tangible" (concrete and visual) means of programming. "Gestural composition", illustrated in this section, causes an output of one TV to be fed directly into an input of another TV. The result is a *fusion* of the two source TVs, i.e., a new TV containing all of the original inputs and outputs *except* for the connected input and output, which vanish.

Figures 10 through 14 show the stages of development of a parameterized visual design (or an "interactive" visual design, depending on perspective). Figure 9 shows conventional syntactic definitions corresponding to the gestural development.

The user selects $uscale$ and $udisk$ from `Parts`, resulting in the left and middle GUIs shown in Figure 10. The $uscale$ GUI has a slider for the first argument and an evaluator for the second argument. This evaluator passes the input string to GHC (for parsing, type-checking, and run-time compilation), via hs-plugins (Pang et al. 2004). In this case, the input image is $xPos$, which is the half-space to the right of the Y axis.

Now we come to gestural composition. To replace the second input of $uscale$ with the output of $udisk$, the user right-clicks first the region input and then the region ouput. The result is the

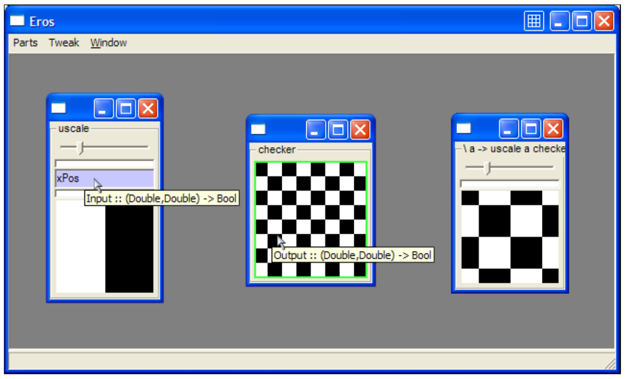

Eros
Parts Tweak Window
uscale
xPos
Input :: (Double,Double) -> Bool
checker
Output :: (Double,Double) -> Bool
\ a -> uscale a checke

Figure 11. Scaling checker

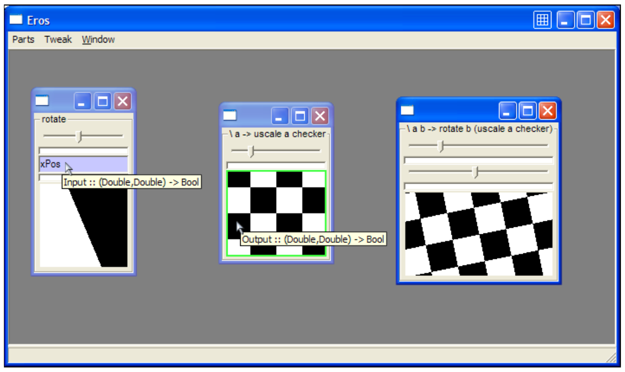

Eros
Parts Tweak Window
rotate
xPos
Input :: (Double,Double) -> Bool
\ a -> uscale a checker
Output :: (Double,Double) -> Bool
\ a b -> rotate b (uscale a checker)

Figure 12. Rotating, scaling checker

appearance of the right-most GUI. Note that the matched region input and output have vanished, leaving just the slider input and the region output from $uscale$.[3]

Eros gives visual feedback to aid composition, highlighting inputs in blue when the cursor passes over them. After an input is chosen, type-compatible outputs are highlighted in green, and only those outputs are candidates for connection.

Figure 11 continues the interactive development. The user adds $uscale$ (left) and $checker$ (middle), and then links $uscale$'s region input (as with $udisk$ above) with $checker$ to get a scalable checker (right). In Figure 12, the user next adds $rotate$ (left) and links its $Region$ input to the scaled checker's $Region$ output. The result is a checker that scales and rotates interactively. Again, with each of the input/output linkings, the resulting TV is a hybrid of the two given TVs, except for the matching input and output, which vanish.

Next the user wants to intersect the scaling disk with the scaling, rotating checker. Figure 13 shows the first composition step, connecting the first region input of $intersect$ (of type $Region \to Region \to Region$) with the region output of the scaling disk of Figure 10, to get a TV of type $R \to Region \to Region$. Figure 14 then shows the final composition step, filling in the remaining region input (left over from $intersect$) with the region output of the scaling, rotating checker.

The previous examples are only nominally higher-order, in that regions, though functions, are shown with an atomic visualizer.

[3] All of the examples in this section follow the form of this example: TV with selected input on the left, TV with selected output in the middle, and resulting composition on the right. For clarity, we show simultaneously input and output selections with cursors and type annotations, though in reality, first the input is selected and then the output.

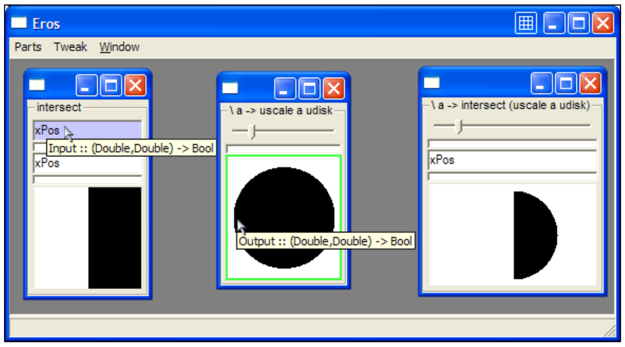

Eros
Parts Tweak Window
intersect
xPos
Input :: (Double,Double) -> Bool
xPos
\ a -> uscale a udisk
Output :: (Double,Double) -> Bool
\ a -> intersect (uscale a udisk)
xPos

Figure 13. Intersection with disk

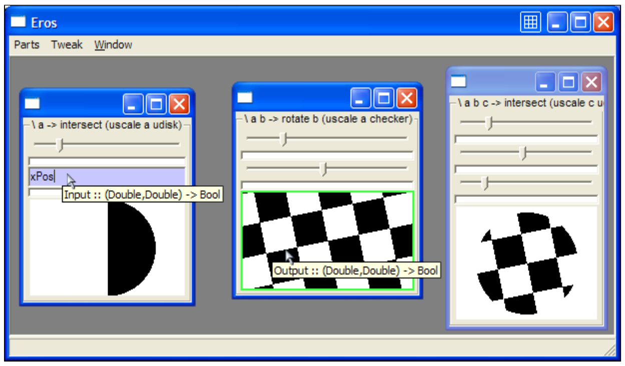

Eros
Parts Tweak Window
\ a -> intersect (uscale a udisk)
xPos
Input :: (Double,Double) -> Bool
\ a b -> rotate b (uscale a checker)
Output :: (Double,Double) -> Bool
\ a b c -> intersect (uscale c u

Figure 14. Intersection of scaling disk with scaling, rotating checker

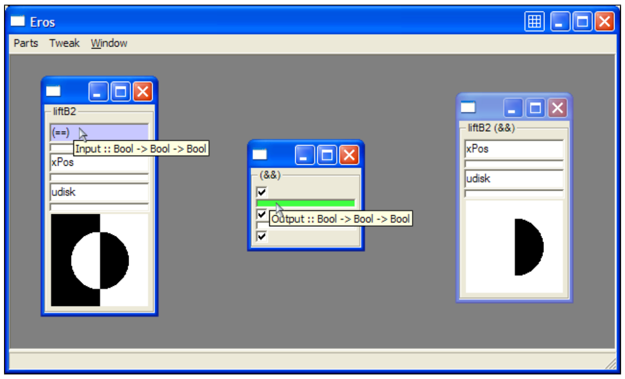


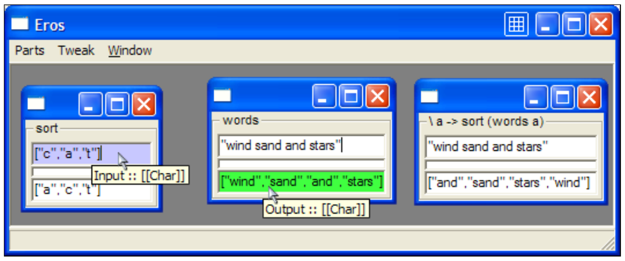
Figure 15. Higher-order composition: region intersection

Figure 15 demonstrates Eros's support for *higher-order* gestural programming, to construct the *intersect* function, using the following functions.

$$\begin{aligned} liftB_2 \;&::\; (Bool \;\;\to Bool \;\;\to Bool\;\;)\\ &\;\;\;\;\to (Region \to Region \to Region)\\ (\&\&) \;&::\;\;\; Bool \;\;\to Bool \;\;\to Bool \end{aligned}$$

In the $liftB_2$ TV on the left, the user has temporarily chosen equality as the input Boolean operation and $xPos$ and $udisk$ as the input regions. For composition, the trick is to use one of the composite, function-valued output handles in the middle TV (logical conjunction).

Eros is not limited to visual designs but can be used with any type for which visualizers can be constructed. As a final example, Figure 16 shows an example using standard Haskell functions. A sentence is broken up into words and then the words are sorted. Figure 17 then puts the sorted words back together into a new sentence.



["c","a","t"]

["a","c","t"]

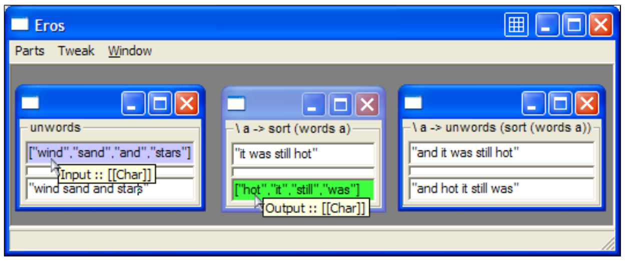
"wind sand and stars"

["wind","sand","and","stars"]

"wind sand and stars"

["and","sand","stars","wind"]

Figure 16. Extract words and sort



["wind","sand","and","stars"]

"wind sand and stars"

"it was still hot"

["hot","it","still","was"]

"and it was still hot"

"and hot it still was"

Figure 17. Sort a phrase

The `Tweak` menu allows the user to restructure a GUI by applying duplication, swapping, currying and uncurrying, argument reordering, pair member extraction, and re-association of pairs. In particular, duplication allows an output to be used more than once.

4. Deep Application

Eros implements gestural composition by translating gestures into sequences of the combinators developed in this section.

As illustrated in Section 3, there are three gestures to translate: (a) selection of an input, (b) selection of an output to be matched up with an input to create a new hybrid TV, and (c) application of one of the reshaping tools in the `Tweak` menu to create a new TV.

Semantically, these three gestures collapse into two actions. The first is extraction of a function in response to gesture (a). The second action is application of a function to a "buried" argument, which is performed in response to gestures (b) and (c). For example, consider Figure 12. To create the TV on the right, the user selects the *Region* input from the left TV and then the *Region* output (the checker board) from the middle TV. The left TV presents

$$rotate :: R \to Region \to Region$$

By selecting the *Region* input of $rotate$, the user indicates not $rotate$ itself to be applied, but rather a related function:

$$\begin{aligned} &rotate' :: Region \to R \to Region\\ &rotate'\; reg\; r = rotate\; r\; reg \end{aligned}$$

In other words, $rotate' = flip\; rotate$. Then in selecting the region output of the middle TV, the user indicates that the previously chosen function ($rotate'$) is to be applied not to the whole (function-valued) TV, but just to the result of the function. In other words, $rotate'$ is to be *composed* with the checker-scaling function. Thus, the result is

$$rotate' \circ (\lambda a \to uscale\; a\; checker)$$

which is equivalent to the expression shown in the right TV:

$$\lambda a\ b \rightarrow rotate\ b\ (uscale\ a\ checker)$$

The rest of this section presents the combinators used in translating user gestures, breaking them into three groups. The first group enables application of functions "deeply", i.e., to buried arguments. The second group extracts buried functions as well, so that the result may be applied (perhaps deeply). The final group extracts individual parts of a function argument, corresponding to the *inputs* of Section 2.2. Typically, all three of these groups are combined to create a new value via gestural composition.

Although our combinators may be viewed as an editing algebra, they work on *values*, rather than *syntax*.

4.1 Transformation Embedders

When composing (whether gesturally or syntactically), we often want to apply a function to *part* of a value. If we're composing syntactically, we simply insert the function application somewhere inside of an expression rather than at the top. Working with values, rather than syntax, requires corresponding semantic tools.[4] Suppose, for example, that we have a pair expression "(e_1, e_2)" and we want to apply a function expression "f" just in the first half. If working syntactically, we would change the pair expression to "$(f\ e_1, e_2)$". Working with values, we can instead "embed" a function f into a function that works on pairs, and then apply that pair-transforming function. Similarly, we can embed a function g to work on pairs by transforming the second element. Thus, we have two embedders:

$$\begin{aligned} &\mathit{first}\ \ :: (a \rightarrow a') \rightarrow ((a, b) \rightarrow (a', b)) \\ &\mathit{second} :: (b \rightarrow b') \rightarrow ((a, b) \rightarrow (a, b')) \\ &\mathit{first}\ \ f = \lambda(a, b) \rightarrow (f\ a, b) \\ &\mathit{second}\ g = \lambda(a, b) \rightarrow (a, g\ b) \end{aligned}$$

Note that, as needed, *first* and *second* apply their first arguments to part of a pair value, carrying along the other half of the pair.

When working syntactically, we often apply functions under lambdas. The corresponding value-level, embedding combinator is just function composition. We use the name "*result*" as a synonym for "$\circ$", for consistency with "*first*" and "*second*" and, more importantly, for generalizing in Section 5.

$$\begin{aligned} &\mathit{result} :: (b \rightarrow b') \rightarrow ((a \rightarrow b) \rightarrow (a \rightarrow b')) \\ &\mathit{result}\ g = \lambda\ f \rightarrow g \circ f \end{aligned}$$

What about deeper embeddings, say doubly layered? Given a value of type $(a, (b, c))$, we might want to transform just the b part and leave the rest intact. In other words, we'd like an embedder

$$\mathit{secondFirst} :: (b \rightarrow b') \rightarrow ((a, (b, c)) \rightarrow (a, (b', c)))$$

Fortunately, this deeper embedder can be defined easily from *second* and *first*:

$$\mathit{secondFirst} = \mathit{second} \circ \mathit{first}$$

since, given $f :: b \rightarrow b'$, we have $\mathit{first}\ f :: (b, c) \rightarrow (b', c)$, and $\mathit{second}\ (\mathit{first}\ f) :: (a, (b, c)) \rightarrow (a, (b', c))$.

We can make longer composition chains to transform more deeply buried arguments. For example, given a value of type $(a \rightarrow (f, b \rightarrow (c, g)), e)$, we might want to apply a function to just the c part and leave the rest intact.

$$\begin{aligned} &emb_1 :: (c \rightarrow c') \rightarrow (a \rightarrow (f, b \rightarrow (c, g)), e) \\ &\qquad\qquad \rightarrow (a \rightarrow (f, b \rightarrow (c', g)), e) \\ &emb_1 = \mathit{first} \circ \mathit{result} \circ \mathit{second} \circ \mathit{result} \circ \mathit{first} \end{aligned}$$

Note from this definition that a deep embedder is formed simply by listing the path from the whole (root) value leading to a buried part. For instance, to define emb_1, we wrote down the steps in reaching c within the type of the second argument (*first* of the pair, *result* of that function, *second* of that pair, etc.)

Let's return to our examples from Section 3 for examples of forming and using transformation embedders. In Figure 13, the *intersect* function is applied to the *result* of *scaleDisk* function. Thus

$$\mathit{intersectDisk} = \mathit{result}\ \mathit{intersect}\ \mathit{scaleDisk}$$

Likewise, in Figure 14, an argument-flipped version of *intersectDisk* is applied to the (region-valued) *result* of the *result* of the curried function *rotScaleChecker*. Thus

$$\begin{aligned} &\mathit{diskChecker} = \\ &\quad (\mathit{result} \circ \mathit{result})\ (\mathit{flip}\ \mathit{intersectDisk})\ \mathit{rotScaleChecker} \end{aligned}$$

4.2 Function Extractors

Instead of a buried *argument*, we might instead have a buried *function*, e.g., $\lambda x \rightarrow x * x$ in $(\lambda x \rightarrow x * x, \texttt{"square"})$. To apply a buried function in a syntactic setting, one can float the lambda to the top of the containing expression (following an η-expansion if there is no explicit λ, and α-converting where needed to avoid variable capture). For instance, $(\lambda x \rightarrow x * x, \texttt{"square"})$ becomes $\lambda x \rightarrow (x * x, \texttt{"square"})$.

Again, working with values rather than syntax requires a different trick. The following three function-extracting combinators are handy:

$$\begin{aligned} &\mathit{funF} :: (c \rightarrow a, b) \rightarrow (c \rightarrow (a, b)) \\ &\mathit{funS} :: (a, c \rightarrow b) \rightarrow (c \rightarrow (a, b)) \\ &\mathit{funR} :: (a \rightarrow c \rightarrow b) \rightarrow (c \rightarrow (a \rightarrow b)) \\ &\mathit{funF}\ (f, b) = \lambda c \rightarrow (f\ c, b) \\ &\mathit{funS}\ (a, f) = \lambda c \rightarrow (a, f\ c) \\ &\mathit{funR}\ g = \lambda c \rightarrow \lambda a \rightarrow g\ a\ c \end{aligned}$$

Returning to our example, if $h = (\lambda x \rightarrow x * x, \texttt{"square"})$, then $\mathit{funF}\ h$ is a function ready to apply. For instance, $\mathit{funF}\ h\ 3 \equiv (9, \texttt{"square"})$.

Next consider more deeply buried functions. Recall that the combinators *first*, *second*, and *result* from Section 4.1 were directly composed for arbitrarily deep application. Our new combinators *funF*, *funS*, and *funR*, however, do not have types suitable for composition with each other. To fix this problem, define three slightly more complex combinators. Each one maps an extractor into another one that reaches more deeply.

$$\begin{aligned} &\mathit{funFirst}\ \ :: (d \rightarrow (c \rightarrow a)) \rightarrow ((d, b) \rightarrow (c \rightarrow (a, b))) \\ &\mathit{funSecond} :: (d \rightarrow (c \rightarrow b)) \rightarrow ((a, d) \rightarrow (c \rightarrow (a, b))) \\ &\mathit{funResult}\ \ :: (d \rightarrow (c \rightarrow b)) \rightarrow ((a \rightarrow d) \rightarrow (c \rightarrow (a \rightarrow b))) \end{aligned}$$

Given a way to extract a function from the first element of a pair, *funFirst* produces a way to extract a function from the pair itself. Similarly for *funSecond* and *funResult*.

As an example function extractor,

$$\begin{aligned} &\mathit{fxt}_1 :: (d \rightarrow (c \rightarrow b)) \rightarrow (e \rightarrow (a, d), f) \\ &\qquad\qquad \rightarrow (c \rightarrow (e \rightarrow (a, b), f)) \\ &\mathit{fxt}_1 = \mathit{funFirst} \circ \mathit{funResult} \circ \mathit{funSecond} \end{aligned}$$

To perform an extraction, apply the extractor to the identity function:

[4] There is a difference in flexibility between working with syntax and with semantics (values). For example, given the expression "3 + 4", a syntactic programmer can insert a function application around "4". When working with values, we simply have 7, so the change would have to be made at an earlier point in development.

$$\begin{aligned} xt_1 &:: \quad (e \to (a,(c \to b)),f) \\ &\to (c \to (e \to (a, \quad b),f)) \\ xt_1 &= fxt_1\ id \end{aligned}$$

In Section 4.3, we will see alternatives to id.

The three composable function extractors are defined easily in terms of the simpler, non-composable ones. For instance, to extract a function buried somewhere in the first element of a pair, extract the function from the first element and then extract one last level to the top.

$$\begin{aligned} &funFirst\ \ \ h = funF \circ first\ \ \ h \\ &funSecond\ h = funS \circ second\ h \\ &funResult\ \ h = funR \circ result\ \ h \end{aligned}$$

The examples of Section 3 use only very simple function extractors, namely id and $funResult$. For instance, in Figure 12, the user selected the (region-valued) input of the *result* of the function $rotate :: R \to Region \to Region$. The extracted function is therefore

$$\begin{aligned} &rotate' :: Region \to R \to Region \\ &rotate' = funResult\ id\ rotate \end{aligned}$$

Of course, in this case, a simpler definition would be $rotate' = funR\ rotate$. As mentioned above, we use $funResult$ in general, because the simpler $funR$ (a synonym for the standard *flip* function) lacks the composability property required to extract deeply buried functions.

4.3 Input Extractors

Recall from Section 3 that our goal is to allow a user to connect compatibly typed inputs and outputs. The function extractors in Section 4.2 are helpful but not sufficient for this goal, since they do not reach into structured inputs.[5]

For example, consider $f :: ((a,b),c) \to d$. We could consider f to have five “inputs”, of types a, b, c, (a,b), and $((a,b),c)$. Any of these five inputs could be eliminated by filling it in with a correspondingly typed value, possibly leaving a residual input. Eliminating the input of type b would yield a result of type $(a,c) \to d$ (which has three inputs, of types a, c, and (a,c)). The usual notion of function application is a special case, in that eliminating the input of type $((a,b),c)$ results in a value of type d.

To support elimination of part of a function argument, we define some tools for “input extraction”. The idea is to transform the given function into a (possibly curried) function of just the desired input, which may then be applied directly. For example, to eliminate the b input from f, we would first transform f into a function of type $b \to (a,c) \to d$, and then apply the transformed function.

The simplest case is that in which the input is the whole function argument, so there is nothing to be done to extract the input.

If the chosen input is directly within a pair type, extraction is also easy, using one of the following two functions:

$$\begin{aligned} &inpF :: ((a,b) \to c) \to (a \to (b \to c)) \\ &inpS :: ((a,b) \to c) \to (b \to (a \to c)) \\ \\ &inpF = curry \\ &inpS = flip \circ curry \end{aligned}$$

A trickier case is where the chosen input is more deeply buried. For example, consider how to eliminate the b input in a function of type $((a,b),c) \to d$ or $((a,(b,e)),c) \to d$. We will want to make “input extractors” with the following types:

$$\begin{aligned} &ixt_1 :: (((a,\ b\ \ ),c) \to d) \to (b \to (\ a\ \ \ ,c) \to d) \\ &ixt_2 :: (((a,(b,e)),c) \to d) \to (b \to ((a,e),c) \to d) \end{aligned}$$

We cannot use the simple combinators $inpF$ and $inpS$, because they do not have composable types (as with the simple combinators $funF$, $funS$, and $funR$ from Section 4.2). To get composability, define the following two combinators, which promote input extractors from simpler to more complex types:

$$\begin{aligned} inpFirst &:: ((\ a \quad \to c) \to (d \to (\ a' \quad \to c))) \\ &\to (((a,b) \to c) \to (d \to ((a',b) \ \to c))) \\ inpSecond &:: ((\quad b\ \to c) \to (d \to (\quad b' \ \to c))) \\ &\to (((a,b) \to c) \to (d \to ((a\ ,b') \to c))) \end{aligned}$$

Given a way to extract a d input from an a input, leaving an a' residual input, $inpFirst$ yields a way to extract a d input from an (a,b) input, leaving an (a',b) residual input. Similarly for $inpSecond$. Definitions can be inferred from the types:

$$\begin{aligned} &inpFirst\ \ \ h\ f\ d\ (a',b) = h\ (\lambda a \to f\ (a,b))\ d\ a' \\ &inpSecond\ h\ f\ d\ (a,b') = h\ (\lambda b \to f\ (a,b))\ d\ b' \end{aligned}$$

Input extraction works by using compositions of $inpFirst$ and $inpSecond$ to target an application of $inpS$ or $inpF$. For example, ixt_1 and ixt_2, whose types are given above, are easily defined:

$$\begin{aligned} &ixt_1 = inpFirst \qquad\qquad\qquad\ \ inpS \\ &ixt_2 = (inpFirst \circ inpSecond)\ inpF \end{aligned}$$

Typically, we will have to combine *function* and *input* extractors. For instance, using fxt_1 from Section 4.2 with ixt_2,

$$\begin{aligned} xt_2 &:: \quad (e \to (g,(((a,(b,e)),c) \to d)),f) \\ &\to (b \to (e \to (g,(((a, \quad e),c) \to d)),f)) \end{aligned}$$

or, spelled out,

$$\begin{aligned} xt_2 = (\ &funFirst \circ funResult \circ funSecond \\ &\circ inpFirst \circ inpSecond \\ )\ &inpF \end{aligned}$$

If xt_2 is applied to a suitably-typed value, the resulting function can be directed, via a function embedder, at an argument of type b buried deeply inside some other value.

The examples from Section 3 do not need input extractors (other than the identity), because they do not have pair-valued inputs.

4.4 Deep Application—Definitions and Examples

We now have the tools to fully implement the constructions given in Section 3. See the conventional syntactic definitions in Figure 9 for comparison.

Using our new tools, we have the following definitions. To clearly relate our combinators with gestural composition, define the following higher-order function for “deep application”:

```
deep ::
      (d → (a → b))            -- function extractor
   → d                         -- function container
   → ((a → b) → (a' → b'))     -- transformation embedder
   → a'                        -- argument container
   → b'                        -- overall result
deep xtr fc emb ac = emb (xtr fc) ac
```

The definitions in Figure 18 exactly parallel the gestural compositions given in Section 3. In each gestural composition, the user selects (a) an input, determining the (combined function and input) extractor xtr and function-containing value fc, and (b) an output, determining the embedder emb and the argument-containing value ac. In these examples, the name “idA” is synonymous with “id” (the identity function), though it will be generalized in the next section.

[5] For now, we use the word “input” to mean part of a function argument, rather than as defined in Section 2.2 for TVs. In Section 5, we will generalize these techniques to a setting that applies to TVs.

```
scaleDisk =        deep (funResult idA)
                        uscale
                        id
                        udisk
intersectDisk = deep (id idA)
                        intersect
                        result
                        scaleDisk
scaleChecker = deep (funResult idA)
                        uscale
                        id
                        checker
rotScaleChecker = deep (funResult idA)
                            rotate
                            result
                            scaleChecker
diskChecker =   deep (funResult idA)
                        intersectDisk
                        (result ∘ result)
                        rotScaleChecker
```

Figure 18. Deep Application Examples

5. Generalizing Deep Application

The combinators from Section 4 enable the gestural composition of pure values. We want, however, to compose not just the values, but visualizers for those values. In fact, we want to compose values and visualizers in tandem, i.e., *tangible values* (TVs), as described in Section 2. In this section, we generalize the combinators so that they apply more broadly, including to visualizers and TVs.

5.1 An Arrow for Visualization

The *first* and *second* combinators used in Section 4.1 are already defined quite broadly, for *arrow* types (Hughes 2000).

```
class Arrow (⇝) where
  first  :: (a ⇝ a') → ((a, b) ⇝ (a', b ))
  second :: (b ⇝ b') → ((a, b) ⇝ (a , b'))

  arr    :: (a → b) → (a ⇝ b)
  (⋙)   :: (a ⇝ b) → (b ⇝ c) → (a ⇝ c)
```

The definitions of *first* and *second* in Section 4.1 belong to the *function Arrow* (i.e., the "→" instance of *Arrow*), where also *arr* is the identity and (⋙) is reverse function composition.

To handle visualization, we'll need another *Arrow* instance, to represent transformation of the "outputs" introduced in Section 4. The following simple definition suffices:

```
newtype OFun a b = OFun (Output a → Output b)
```

We define the *OFun* instance of *Arrow* in close analogy with the "→" instance. Recall the definitions of *first* and *second* for functions (Section 4.1):

```
first  f = λ(a, b) → (f a, b  )
second g = λ(a, b) → (a  , g b)
```

The definitions for *OFun* are almost identical, using "output pairs" rather than value pairs. In order to pattern-match on structure, we represent outputs and inputs as generalized algebraic data types (Peyton Jones et al. 2006) that directly mirror the visualizer algebra in Figure 8.[6]

```
instance Arrow OFun where
  first   (OFun f) =
    OFun (λ(OPair a b) → OPair (f a)   b)
  second (OFun g) =
    OFun (λ(OPair a b) → OPair    a (g b))
```

Arrow composition is defined via function composition.

```
OFun f ⋙ OFun g = OFun (f ⋙ g)
```

Given only a pure function, *arr* has no useful way of rendering values. One could signal an error or yield an output that displays a warning message. Unavailability of *arr* comes up in other contexts (Alimarine et al. 2005), so it may be worth introducing a *Arrow* superclass without *arr*.

5.2 An Arrow for Tangible Values

Recall from Section 2.3 that a tangible value is simply a value and an output. Providing an arrow instance for TVs is a simple matter of combining the function arrow for transforming values and the *OFun* arrow for transforming outputs.

```
data TvFun a b = TvFun (a → b) (OFun a b)
```

The instance definition then operates on values and their visualizers in tandem.

```
instance Arrow TvFun where
  first   (TvFun f ox) = TvFun (first   f) (first   ox)
  second (TvFun f ox) = TvFun (second f) (second ox)
  TvFun f ox ⋙ TvFun f' ox' =
    TvFun (f ⋙ f') (ox ⋙ ox')
  arr f = TvFun (arr f) (arr f)
```

5.3 Deep Arrows

Section 4 introduced additional combinators besides *first* and *second*. To generalize these combinators beyond values, we define a new type class *DeepArrow*, as shown in Figure 19.

Besides the methods introduced in Section 4, the *DeepArrow* class contains several more that are useful for manipulating outputs and TVs. Most of the *DeepArrow* methods are generalizations of familiar functions.

As shown in Figure 20, the function instance of *DeepArrow* defines *result* as function composition (as in Section 4.1) and otherwise uses *arr* to give very simple definitions. Any arrow with a useful *arr* can use these same *arr*-based definitions.

The *OFun* instance of *DeepArrow* is defined by mimicking the function instance.

```
instance DeepArrow OFun where
  result (OFun ox)
             = OFun ( λ (OLambda a b)
                          → (OLambda a (ox b)))
  funF     = OFun ( λ (OPair (OLambda c a) b)
                          → (OLambda c (OPair a b)))
  lAssocA = OFun ( λ (OPair a (OPair b c))
                          → (OPair (OPair a b) c))
  ...
```

[6] Note that the functions in the right-hand sides of the definitions may fail, because pair-valued outputs may be constructed by *OPut* rather than *OPair*. Our gestural setting, however, ensures that *first* and *second* get applied only to outputs constructed by *OPair*. In a less constrained setting, a simple alternative to partiality of *first* and *second* would be mapping a non-*OPair* to an *OPut*-based output that displays a warning message in a GUI component.

```
class Arrow (⇝) ⇒ DeepArrow (⇝) where
  result    :: (b ⇝ b′) → ((a → b) ⇝ (a → b′))

  funF      :: (c → a , b) ⇝ (c → (a , b))
  funS      :: (a , c → b) ⇝ (c → (a , b))
  funR      :: (a → c → b) ⇝ (c →  a → b )

  curryA    :: ((a ,  b) → c ) ⇝ ( a →  b  → c )
  uncurryA  :: ( a  → b  → c ) ⇝ ((a ,  b) → c )
  lAssocA   :: ( a ,( b  , c)) ⇝ ((a ,  b) , c )
  rAssocA   :: ((a ,  b) , c ) ⇝ ( a , (b  , c))

  idA       ::   a    ⇝   a
  dupA      ::   a    ⇝ (a, a)
  fstA      :: (a, b) ⇝   a
  sndA      :: (a, b) ⇝   b
  swapA     :: (a, b) ⇝ (b, a)

flipA :: DeepArrow (⇝) ⇒ (a → c → b) ⇝ (c → a → b)
flipA = funR

inpF :: DeepArrow (⇝) ⇒ ((a, b) → c) ⇝ (a → (b → c))
inpF = curryA

inpS :: DeepArrow (⇝) ⇒ ((a, b) → c) ⇝ (b → (a → c))
inpS = curryA ⋙ flipA
```

Figure 19. *DeepArrow* class

Given the function and *OFun* instance of *DeepArrow*, the *TvFun* instance (for transforming TVs) is defined to work on values and their visualizers in tandem (as with the *Arrow* instance defined in Section 5.2).

```
instance DeepArrow TvFun where
  result (TvFun f ox) = TvFun (result f) (result ox)

  idA     = TvFun idA     idA
  dupA    = TvFun dupA    dupA
  fstA    = TvFun fstA    fstA
  ...
  rAssocA = TvFun rAssocA rAssocA
```

Defining generalized versions of the input extractors *inpFirst* and *inpSecond* is considerably trickier. The function-based definitions from Section 4.3 use λ abstraction and hence do not translate directly to an arrow setting. We use an indirect approach. First re-express the λ-based versions in combinator (λ-free) form, and then replace the combinators with generalized versions. The combinator versions are as follows, with a breakdown of these definitions given Figure 21. These definitions use "lexically scoped type variables" (Peyton Jones and Shields 2002), to relate the types of the locally defined q_i to the type of the top-level definitions.

```
inpFirst  h = result uncurry ∘ result flip ∘ flip
                     ∘ result h ∘ flip ∘ curry
inpSecond h = result uncurry ∘ flip ∘ result h ∘ curry
```

The generalized definitions then follow simply by replacing each function-based combinator with its generalized version from *Arrow* or *DeepArrow* (replacing "$g \circ f$" with "$f \ggg g$"). Figure 22 shows the generalized types and definitions of the function and input extractors of Section 4.

```
instance DeepArrow (→) where
  result   = (∘)
  funF     = arr (λ(f, b) → λc → (f c, b))
  funS     = arr (λ(a, f) → λc → (a, f c))
  funR     = arr flip
  curryA   = arr curry
  uncurryA = arr uncurry
  lAssocA  = arr (λ(a, (b, c)) → ((a, b), c))
  rAssocA  = arr (λ((a, b), c) → (a, (b, c)))
  idA      = arr id
  dupA     = arr (λx → (x, x))
  fstA     = arr fst
  sndA     = arr snd
  swapA    = arr (λ(a, b) → (b, a))
```

Figure 20. *DeepArrow* instance for functions

```
inpFirst :: ∀a a′ b c d.
              (( a      → c) → (d → ( a′      → c)))
            → ((((a, b) → c) → (d → ((a′, b) → c)))

inpFirst h f = q6
  where
    q1 = curry f              :: a → (b → c)
    q2 = flip q1              :: b → (a → c)
    q3 = result h q2          :: b → (d → (a′ → c))
    q4 = flip q3              :: d → (b → (a′ → c))
    q5 = result flip q4       :: d → (a′ → b → c)
    q6 = result uncurry q5    :: d → ((a′, b) → c)

inpSecond :: ∀a b b′ c d.
               ((     b  → c) → (d → (     b′  → c)))
             → ((((a, b) → c) → (d → ((a, b′) → c)))

inpSecond h f = q4
  where
    q1 = curry f              :: a → (b → c)
    q2 = result h q1          :: a → (d → (b′ → c))
    q3 = flip q2              :: d → (a → (b′ → c))
    q4 = result uncurry q3    :: d → ((a, b′) → c)
```

Figure 21. Derivations of *inpFirst* and *inpSecond*

5.4 Transforming Values

In order to use our *DeepArrow* values to transform other values, we introduce a class that relates an arrow (e.g., "→", *OFun*, or *TvFun*) with a wrapper type (e.g., *Id*, *Output*, or *TV*). The class method *toArr* turns wrapped functions into arrow values, while $$ turns arrow values into functions on wrapped values.[7]

[7] This (multi-parameter) class definition uses "functional dependencies" (Jones 2000), expressing that each of the type parameters uniquely determines the other. This detail helps the compiler resolve some ambiguities in type inference and is otherwise unimportant.

```
funFirst   :: DeepArrow (⇝) ⇒
  (d ⇝ (c → a)) → ((d , b) ⇝ (c → (a , b)))
funSecond  :: DeepArrow (⇝) ⇒
  (d ⇝ (c → b)) → ((a , d) ⇝ (c → (a , b)))
funResult  :: DeepArrow (⇝) ⇒
  (d ⇝ (c → b)) → ((a → d) ⇝ (c → (a → b)))

funFirst  h = first  h ⋙ funF
funSecond h = second h ⋙ funS
funResult h = result h ⋙ funR

inpFirst   :: DeepArrow (⇝) ⇒
                (( a      → c) ⇝ (d → ( a'      → c)))
             → (((a, b) → c) ⇝ (d → ((a', b) → c)))
inpSecond  :: DeepArrow (⇝) ⇒
                ((    b   → c) ⇝ (d → (    b'   → c)))
             → (((a, b) → c) ⇝ (d → ((a, b') → c)))

inpFirst h =  curryA ⋙ flipA ⋙ result h ⋙ flipA ⋙
              result flipA ⋙ result uncurryA
inpSecond h = curryA ⋙ result h ⋙ flipA ⋙
              result uncurryA
```

Figure 22. Generalized function and input extractors

```
class FunArr (⇝) w | (⇝) → w, w → (⇝) where
  toArr :: w (a → b) → (a ⇝ b)
  ($$)  :: (a ⇝ b)   → w a → w b
```

The simplest *FunArr* instance relates "→" and the identity wrapper:

```
instance FunArr (→) Id where
  toArr (Id f) = f
  f $$ Id a    = Id (f a)
```

An *OFun* can transform an *Output*, with *toArr* discarding the *Input*, and "$$" applying the output function.

```
instance FunArr OFun Output where
  toArr (OLambda _ res) = OFun (const res)
  OFun ox $$ oa         = ox oa
```

Finally, a *TvFun* can transform a *TV*, working in tandem (as usual) on values and visualizers:

```
instance FunArr TvFun TV where
  toArr (TV f fo)          = TvFun (toArr f) (toArr fo)
  TvFun f ox $$ TV ida o = TV (f $$ ida) (ox $$ o)
```

5.5 Deep Application

The "deep application" higher-order function, *deep* from Section 4.4, generalizes simply:

```
deep :: FunArr (⇝) w
     ⇒ (d ⇝ (a → b))                -- function extractor
     → w d                            -- function container
     → ((a ⇝ b) → (a' ⇝ b'))         -- transformation embedder
     → w a'                           -- argument container
     → w b'                           -- overall result
deep xtr fc emb ac = emb (toArr (xtr $$ fc)) $$ ac
```

With this new definition, the examples in Figure 18 all work for tangible values, exactly as written (assuming TV versions of the pre-defined *udisk*, *checker* etc from Figure 9). That is, the definition simultaneously construct values *and* visualizers for those values. Only the definition types differ. For instance (cf. Figure 9),

```
scaleDisk    :: TV (R → Region)
diskChecker :: TV (R → R → R → Region)
```

6. Translating Gestures to Deep Application

Where are we now?

Recall that our goal is to enable an end-user to construct pure values—including higher-order values—with a concrete presentation in the form of GUIs. This combination, which we call "tangible values" (TVs), allows interactive inspection, as shown in Section 2. Section 3 illustrated "gestural composition", in which a user matches up compatibly-typed inputs and outputs of existing TVs to create new TVs. The newly created TV is a fusion of the two TVs, with the connected input and output removed. Sections 4 and 5 provided the all of the tools (combinators) we need, and in a general enough setting, to support gestural composition.

To put all of these pieces together, Eros uses the GUI structure and user gestures to synthesize combinator chains, which are then used to create new TVs. Each GUI is a tree of input and output sub-GUIs (individual widgets and composites). Every input node has an associated a function extractor (Sections 4.2 and 4.3) that can extract that input all the way to the root, resulting in a function whose domain is the type of the extracted input. Similarly, every output node has an associated transformation embedder (Section 4.1) that can deeply apply a user-chosen TV transformation (*TvFun* from Section 5.2) at the given output node.

7. Persistence and Compilation

While the first definition of *TV* (Section 2.3) suffices for interaction and composition, it does not support persistence or compilation. These latter goals can both be achieved by introducing a term representation to be used in place of the *Id* constructor in the definition of *TV*, and a corresponding *DeepArrow* instance in place of "→" in the definition of *TvFun*.

```
data TV a = TV (Term a) (Output a)

data TvFun a b = TvFun (TFun a b) (OFun a b)
```

For simplicity, we use higher-order abstract syntax (as apparent in the *Lam* constructor below), which frees us from dealing with variable capture and α-conversion (Pfenning and Elliott 1988). Because we wanted static typing wherever possible, the term representation is a generalized algebraic data type (GADT) (Peyton Jones et al. 2006).[8]

```
type TX a b = Term a → Term b

data Term :: ∗ → ∗ where
  Lit  :: String → a → Term a
  App  :: Term (a → b) → Term a → Term b
  Lam  :: PatShape a → TX a b → Term (a → b)
  Fst  :: Term ((a, b) → a)
  Snd  :: Term ((a, b) → b)
  Pair :: Term (a → b → (a, b))

newtype TFun a b = TFun (TX a b)
```

[8] In the GADT definition for *Term*, the first line says that *Term* maps types to types. The indented lines that follow list the *Term* constructors and their types.

```
instance Arrow TFun where
  TFun f >>> TFun g = TFun (f >>> g)
  first (TFun g)    =
    toArr (Lam (SPair SVar SVar) (λab →
               Pair @^ f (Fst @^ ab) @^ Snd @^ ab))
  ...
instance DeepArrow TFun where
  result (TFun tx) = TFun (
    inLets (λab@(Lam sa _) →
      Lam sa (tx ∘ (ab@^))))
  idA              = TFun id
  ...
instance FunArr TFun Term where
  toArr tf       = TFun (toArrT tf)
  TFun tx $$ ta = inLets tx ta

toArrT :: Term (a → b) → TX a b
toArrT f = inLets (f@^)

  -- Apply under β-redices
inLets :: TX a b → TX a b
inLets tx (App (Lam s f) t) = Lam s (inLets tx ∘ f) @^ t
inLets tx term              = tx term
```

Figure 23. Term transformation instances

The last three *Term* constructors are necessary only for simplifications made during term construction, together with limitations in type checking for GADTs. Ongoing improvements to GADT type checking may address this issue, allowing us to simplify the term representation and improve simplification.

For pretty-printing, lambda terms have "pattern shapes":

```
data PatShape :: * → * where
  SVar  :: PatShape a
  SPair :: PatShape a → PatShape b → PatShape (a, b)
```

7.1 Evaluation and Compilation

Evaluation is very simple. The only novel case is *Lam*. The trick there is to apply the contained function to a new literal.

```
eval :: Term a → a

eval (Lit _ a)       = a
eval (App fun arg) = (eval fun) (eval arg)
eval (Lam _ tfun)  = λx → eval (tfun (Lit "_" x))
eval Fst             = fst
eval Snd             = snd
eval Pair            = (,)
```

Term evaluation is simple but involves repeated interpretation. For faster execution, Eros transforms terms into strings of Haskell code and invokes GHC, by means of hs-plugins (Pang et al. 2004). There is a noticeable but quite tolerable pause during compilation (typically well under a second, after GHC is first loaded). Haskell code generation is also used to generate the syntactic descriptions seen near the top of TVs in the figures throughout this paper.

7.2 Term Transformation

Terms are transformed via *TFun*, defined above. Figure 23 shows how *TFun* serves as a deep arrow and how it is used to transform terms. The function *inLets* is very handy for term simplification. It applies applies a term transformation under a sequence of outer "let"-bindings (represented as β-redices). The infix @^ operator is a simplifying "smart constructor" for function application. The *toArr* method (turning a function-valued term into a *TFun*) just makes a (smart) application inside of outer lets.

8. Related Work

8.1 Visual Programming Languages

Eros has similar goals to those of "visual" (or "graphical") programming languages (VPLs), namely making programming more accessible through a concrete and visual style of construction. The user experience of Eros, however, is meant to be profoundly more direct that of VPLs. Eros users construct and interact *directly with values* (semantics) rather than *indirectly through code* (syntax). (Our use of a term representation is not essential and is intended to be hidden from the user.)

In addition to the philosophical and psychological shift, working with values rather than code has a significant effect on scalability. The size and complexity of a TV depends only on its type, not its construction. For this reason, users can construct TVs with considerable internal complexity while maintaining visual simplicity. The key to retaining simplicity is that each deep application step *eliminates* the connected input and output. The newly created TV is thus *strictly less complex* than the combined complexity of the two source TVs. In contrast, the composition step in VPLs (as also with textual languages) creates a visual representation that is strictly larger than the combined sizes of the source components, so VPLs quickly clutter visual space. On the other hand, by preserving and displaying syntactic structure, VPLs allow editing of that structure, unlike Eros. (An easy partial solution for Eros would be to give each TV a button that regenerates its source TVs.)

Most VPLs make a strict stylistic separation between "functions" and "values" (typically rendered as boxes and arrows). VPLs thus tend not to support higher-order programming, though there are exceptions (Poswig et al. 1994; Kelso 1994; Najork and Golin 1990). Eros allows functions to be visualized differently, and yet is fully higher-order. Not only are a function's inputs and outputs accessible and connectable, but so is the function itself.

8.2 Spreadsheets

The most popular end-user programming environment is the spreadsheet, which has an execution model much like Eros's. Once relationships are established, input cells can be altered by the user, and related output cells update automatically. Moreover, the relationships are often established gesturally, using a mouse to select input cells and operation. Wakeling (2007) demonstrated use of (purely syntactic) functional code in the cells of a standard spreadsheet.

On the other hand, Eros remedies some of the significant limitations of spreadsheets. Eros supports a rich, static type system. In particular, it fully supports higher-order functional programming. Spreadsheets typically do not even support definability of functions, though see Peyton Jones et al. (2003). The Forms/3 system (Burnett et al. 2001), while still first order, relaxes the usual spreadsheet style with free-form layout and graphical types.

Another spreadsheet system that explores graphical types is "Spreadsheets for Images" (SI) (Levoy 1994). The author remarks that SI is more scalable because it "spends its screen space on operands rather than operators, which are usually more interesting to the user". On the other hand, SI shows all intermediate values as with spreadsheets in general and unlike Eros. SI has a mixed functional/imperative style, exposing "registers" and commands that explicitly write into registers. It also lacks the early feedback made possible by static typing.

8.3 Vital

Vital (Hanna 2002) is a document-oriented system that presents both (syntactic) source code and the results of evaluating the code, with rich facilities for value visualization. Syntactic editing results in automatic value update, and direct-manipulation editing of values results in update of corresponding expressions. In contrast to Eros, Vital's direct manipulation is used to edit first-order values, rather than to explore (possibly higher-order) values, and functions are created syntactically.

9. Limitations and Future Work

Eros could be improved in a number of significant ways.

- Polymorphism. How can one *concretely* visualize polymorphic values? The very tangibility of our enterprise would seem to preclude parametric polymorphism, at a considerable loss of expressiveness. A possible solution is inspired by the idea of type abstraction in explicitly polymorphic λ-calculi (Reynolds 1974). In place of a type abstraction, a polymorphic TV could have a sort of "viz-abstraction", a different kind of input (e.g., a menu) with which a user can choose a visualizer and, in doing so, a type. Initially, these special inputs would have default values (as regular inputs do). When the user selects a different visualizer, uses of the previously chosen visualizer would get dynamically replaced.
- Naturalness. Gestural composition could be made much more natural and intuitive. For instance, to flip the arguments of a function, the user must select *flip* from the `Tweak` menu. At the very least, a visual representation of *flip* would be friendlier. Better yet would be for the user to grab and move the inputs directly. Similarly for duplication, currying, pair reassociation and the other methods of the *DeepArrow* class (Figure 19).
- Fluidity and optimization. Especially with improvements to gestural composition, the user experience would be enhanced by even faster generation of new TVs. For instance, one could imagine dragging an output over various inputs and having the tentative composition results shown immediately. If the user moves on without completing the connection (the "drop" of drag & drop), the composition vanishes. The main obstacle to this level of fluidity is the time taken by dynamic compilation.
- Haskell integration. It is fairly easy to import definitions from Haskell libraries and make them available in Eros. However, it could be more convenient still. For instance, there could be a pre-processor that reads Haskell source files, interpreting certain comments as declaring what visualizers to use (including lower and upper bounds for slider inputs). A form of integration that is completely lacking now is the ability to save TVs to a Haskell module. Given the persistent representation from Section 7, we know of no obstacle to doing so.

10. Acknowledgments

The inspiration for Eros came out of conversations with Holly Croydon. Sean Seefried, Simon Peyton Jones, Kragen Javier Sitaker, Paul Potts and anonymous reviewers made several helpful suggestions on improving presentation. Sean suggested the name "tangible values".

References

Artem Alimarine, Sjaak Smetsers, Arjen van Weelden, Marko van Eekelen, and Rinus Plasmeijer. There and back again: arrows for invertible programming. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 86–97, New York, NY, USA, 2005. ACM Press.

Margaret Burnett, John Atwood, Rebecca Walpole Djang, James Reichwein, Herkimer Gottfried, and Sherry Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11 (2):155–206, 2001.

Conal Elliott. Functional Images. In *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, March 2003.

Keith Hanna. Interactive visual functional programming. In S Peyton Jones, editor, *ICFP '02: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 100–112. ACM, October 2002.

Paul Hudak and Mark P. Jones. Haskell vs. Ada vs. C++ vs Awk vs . . . an experiment in software prototyping productivity. Technical report, Yale, 1994.

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

Mark P. Jones. Type classes with functional dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer-Verlag.

Joel Kelso. A visual representation for functional programs. Technical Report CS/95/01, Western Australia, Australia, 1994.

Marc Levoy. Spreadsheets for images. In *SIGGRAPH '94: Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, pages 139–146, New York, NY, USA, 1994. ACM Press.

M. A. Najork and E. Golin. Enhancing show-and-tell with a polymorphic type system and higher-order functions. In *Proceedings of the IEEE Workshop 1990 on Visual Languages*, pages 215–220, Skokie, IL, 1990.

André Pang, Don Stewart, Sean Seefried, and Manuel M. T. Chakravarty. Plugging Haskell in. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 10–21. ACM Press, 2004.

Simon Peyton Jones and Mark Shields. Lexically scoped type variables. March 2002.

Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. A user-centred approach to functions in Excel. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 165–176, New York, NY, USA, 2003. ACM Press.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, Portland, Oregon, September 2006. ACM SIGPLAN.

Frank Pfenning and Conal Elliott. Higher-Order Abstract Syntax. In *Programming Language Design and Implementation*, 1988.

Jörg Poswig, Guido Vrankar, and Claudio Moraga. VisaVis: a higher-order functional visual programming language. *Journal of Visual Languages and Computing*, 5(1):83–111, 1994.

John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, London, UK, 1974. Springer-Verlag.

David Wakeling. Spreadsheet functional programming. *Journal of Functional Programming*, 17(1):131–143, 2007.