Polymorphic Blocks: Formalism-Inspired UI for Structured Connectors

Sorin Lerner **Stephen R. Foster** **William G. Griswold**
University of California, San Diego
{lerner, srfoster, wgg}@cs.ucsd.edu

ABSTRACT

We present a novel block-based UI called *Polymorphic Blocks*, in which a connector's shape visually represents the structure of the data being passed through the connector. We use Polymorphic Blocks to add visual type information to block-based programming environments like Blockly or Scratch. We also use Polymorphic Blocks to represent logical proofs. In this context, if we erase all symbols, our UI becomes a puzzle game, where solving the puzzle amounts to building a proof. We show through a user study that our Logical Puzzle Game is faster, more fun, and more engaging than an equivalent pen-and-paper interface.

Author Keywords

Proofs; Games; Block-based Programming Environments.

ACM Classification Keywords

H.5.2. Information Interfaces and Presentation (e.g. HCI).

INTRODUCTION

Blocks that snap into place are commonly used in UIs for novice visual programming environments such as Scratch [5], Blockly [2], App Inventor [1], and TinkerBlocks [6]. A key component of these block interfaces is the UI mechanism for connectors, and the most common such mechanism consists of shaped ports, making blocks akin to jigsaw puzzle pieces. For example, Blockly uses a groove reminiscent of jigsaw puzzles for ports where values can be transmitted, and a semi-circular groove for ports where statements can be connected. Scratch uses circular ports for integers, angled ports for booleans, and a small flat trapezoid for statements.

Although these shaped ports provide some form of visual feedback about the kind of information being passed through the port, typically environments only use a handful of shapes, and as a result, these shaped ports do not visually expose the full structure of the data being passed through them. For example, in most block-based visual programming environments, the shape for an integer, a boolean, a list of integers, and a list of booleans would look exactly the same (in particular, it will be the "value" shaped port).

CHI 2015, April 18–23, 2015, Seoul, Republic of Korea.

ACM 978-1-4503-3145-6/15/04 ...$15.00.
http://dx.doi.org/10.1145/2702123.2702302

In this paper, we present a new block-based UI called *Polymorphic Blocks* that is better suited for visualizing the structure of information passed through connectors. There are two key insights behind Polymorphic Blocks that lead to its flexibility and expressive power.

First, the shape of connectors in Polymorphic Blocks are produced through a compositional translation process that can map complex recursive structures into unique shapes. This allows Polymorphic Blocks to visually represent complex structures such as types of the form $List[int]$ and $Pair[int, bool]$, as well as complex logical formulas.

Second, we allow the shape of connectors in Polymorphic Blocks to include *Colored Polymorphic Ports*, which work as follows: when a shape is connected to a Polymorphic Port of a certain color, the port in question takes on the shape it is connected to, and furthermore, all Polymorphic Ports of the same color also change to that shape. As a simple example, suppose we want a Polymorphic Block that takes two parameters which must be of the same type, for example (int, int) or (bool, bool) but not (int, bool). The Polymorphic Block to encode this operation would have two input Polymorphic Ports of the same color, and when one is connected to a shape, say the shape of bool, the other will also change to that shape, indicating that henceforth only the shape of bool can be passed to the other parameter.

In the context of block-based visual programming interfaces, Polymorphic Blocks can visually represent complex types such as $List[int]$, $List[bool]$, $Pair[int, bool]$, as well as parametric types such as $List[T]$, $Pair(T_1, T_2)$, and higher-order types, such as function types. In this setting, Polymorphic Blocks have a theoretical grounding, in that they correspond precisely to the notion of *parametric polymorphism* (alternatively called *generics*) from the programming languages community, a kind of type polymorphism found in languages like Java, C#, Standard ML, OCaml, F#, Haskell and Scala.

Even more interestingly, it turns out that Polymorphic Blocks are applicable beyond block-based visual interfaces. In particular, we will show how Polymorphic Blocks can be used as a novel visual interface for building logical proofs, where each block represents an inference rule, and connectors represent formulas. If the logical formulas are hidden, and only the visual connectors are left, the resulting interface turns into a visual Puzzle Game where solving the puzzle amounts to completing the logical proof. This application, which we find intriguing and exciting, has possible implications for the future crowdsourcing of interactive proofs in theorem provers like Coq.

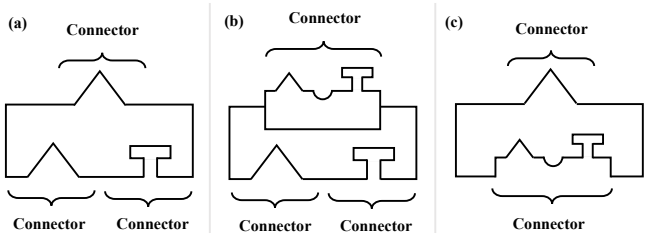


Figure 1. Examples of Polymorphic Blocks: (a) shows a block with some simple connectors; (b) shows a block with more complex connectors.

Finally, the design of Polymorphic Blocks was heavily guided by the formalism of type theory and logical proofs. We discuss this notion of *formalism-inspired UI design* further throughout the paper, showing both its benefits and pitfalls.

In summary, our contributions are as follows:

- We present the idea of Polymorphic Blocks, a novel UI mechanism for visualizing the structure of data passed through connectors.
- We implement Polymorphic Blocks in two settings: (1) a typed version of Blockly and (2) a visual interface for building logical proofs.
- By means of a user study, we evaluate Polymorphic Blocks as a visual interface for building logical proofs in a Puzzle Game. Our results show that participants are able to understand the mechanics of Polymorphic Blocks through self-guided experimentation, and are able to quickly build logical proofs. Furthermore, our results also show that, while Polymorphic Blocks do not teach all aspects of logical proofs in a class-room setting, they lead to faster completion times, and are more fun and engaging than a traditional pen-and-paper interface.
- Finally, we discuss our results and techniques within the context of several possible applications and in relation to existing HCI theories, pointing out lessons learned from our formalism-inspired design.

OVERVIEW

We start with an overview of how Polymorphic Blocks work through a set of increasingly more complex examples.

Basics of Blocks and Connectors. Figure 1(a) shows a very simple example of a Polymorphic Block. Each polymorphic block has a set of *connectors*, with each connector representing one of the block's inputs or outputs. For example, the block in Figure 1(a) has three such connectors, two on the bottom, and one at the top. Each connector has a shape, which intuitively represents the structure of the data being passed through that connector. For example, the block in Figure 1(a) could be used to represent a computation that takes two inputs from below, and produces one output above. In this case, the shape of the connectors would represent the types of the values being passed through the connectors, for example the triangle could be *int* and the "T" could be *bool*. The block in Figure 1(a) would then represent a computation that takes an *int* and a *bool*, and produces an *int*. To start, let us assume that two blocks can be connected through a connector only if the

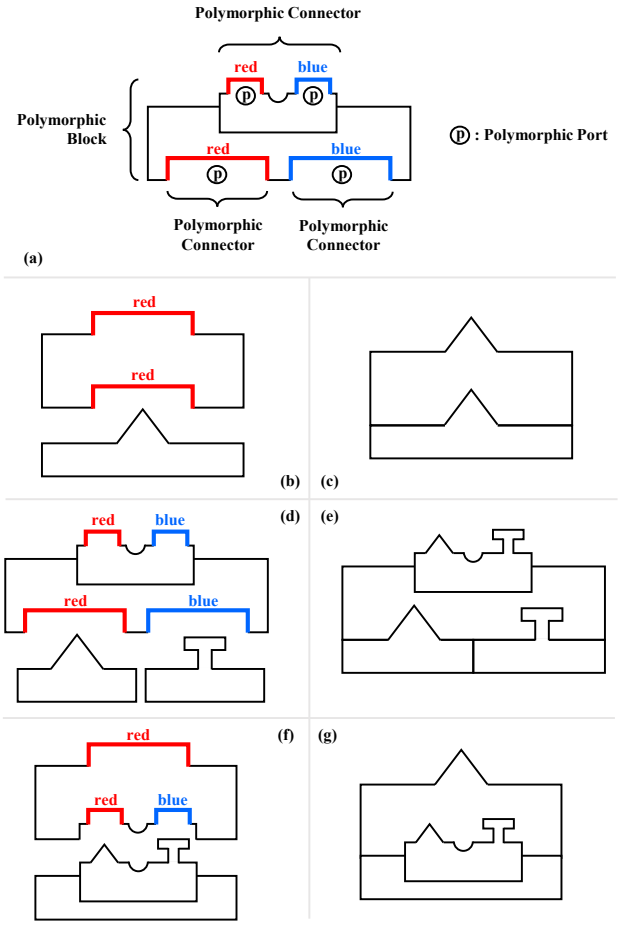


Figure 2. Examples of Polymorphic Ports: (a) shows an example containing Polymorphic Ports with each constituent part labeled; (b) shows two blocks before they are connected and (c) shows what happens after these blocks are connected; similarly for (d)-(e) and (f)-(g).

shape of the connector matches exactly. For example, a block producing a triangle on top could be connected to the bottom-left connector of the block in Figure 1(a). In this way, shaped-connectors provide a jigsaw-puzzle metaphor which enforces that correctly typed values are passed along each connector.

More Complex Shapes for Connectors. One of the key insights in Polymorphic Blocks is that the shape of connectors are produced through a compositional translation process which can map complex recursive structures into unique shapes. Figure 1(b) shows a block with a connector on top that is produced by this translation process. The shape of the top connector is produced by the combination of two simpler shapes (the triangle and the "T") together with a semi-circle. If we continue with the domain where blocks are computations and shapes are types, the semi-circle could represent the notion of a "pair". In this case, the block in Figure 1(b) represents a computation which takes an *int*, represented as a triangle, and a *bool*, represented as a "T", and produces a value of type *Pair[int, bool]*, i.e. a pair of an *int* and a *bool* (an example: the computation which takes an *int* and a *bool* and creates a pair out of them). Figure 1(c) shows another example, which could represent a computation that takes a

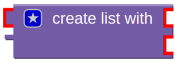
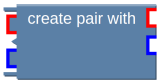
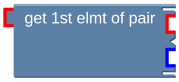
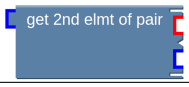
Basic exprs with types	In Polymorphic Blockly
int 0	
bool true	
T cond$(bool, T, T)$	
$List[T]$ list(T, T, T)	
$Pair[T_1, T_2]$ pair(T_1, T_2)	
T_1 first$(Pair[T_1, T_2])$	
T_2 second$(Pair[T_1, T_2])$	

Figure 3. Basic expressions and their corresponding Polymorphic Blocks

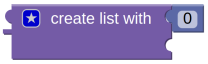
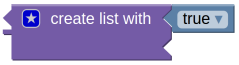
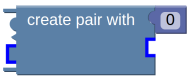
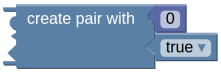
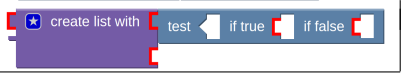
Expression	In Polymorphic Blockly
cond$(\ldots, 0, \ldots)$	
list$(0, \ldots)$	
list$(\text{true}, \ldots)$	
pair$(0, \ldots)$	
pair$(0, true)$	
first(pair$(0, \text{true})$)	
list(cond$(\ldots), \ldots)$	

Figure 4. Complex expressions and corresponding Polymorphic Blocks

Pair[int, bool] and produces an *int* (e.g.: the computation which returns the first element of an *int*, *bool* pair).

Note that the semi-circle here acts as what we call a *shape constructor*: it takes two shapes on either side and glues two together into a larger shape. We support arbitrarily many shape constructors, and later in the paper we will see another example, namely a "V" shaped dent. We can also easily support shape constructors with arities other than 2.

It's important to note that when a shape constructor like the semi-circle combines two shapes into a larger one, the two constituent shapes themselves can be complex shapes built out of shape constructors. As a result, the shapes of connectors are produced from a recursive visual language capable of expressing arbitrarily complex shapes. Through this lens, the shapes of connectors represent an underlying structure corresponding to a language of recursive mathematical terms. These mathematical terms are made up of: function symbols (which correspond to shape constructors) and constants (which correspond to basic shapes). In our instantiations of Polymorphic Blocks, this language of terms will be: (1) the language of type expressions for typed Blockly, and (2) the language of formulas for our visual proof environment.

Polymorphic Ports. Another key insight of Polymorphic Blocks is that the shape of connectors are allowed to include *Colored Polymorphic Ports*. Figure 2(a) shows an example containing Polymorphic Ports, with each constituent part labeled, and where the color of the port is also labeled (for readability on black-and-white printouts). Polymorphic Ports work as follows: when a shape is connected to a Polymorphic Port of a certain color, the port in question takes on the shape it is connected to, and furthermore, all Polymorphic Ports of the same color also change to that shape. For example, Figure 2(b) shows two blocks before they are connected, and Figure 2(c) shows what happens after these blocks are connected – similarly for (d)-(e) and (f)-(g).

INSTANTIATION TO THE BLOCKLY DOMAIN

Now that we have presented an overview of the ideas behind Polymorphic Blocks, we show how these ideas can be used in practice. We start by showing how Polymorphic Blocks can be used to enhance block-based programming environments with a visualization of program types.

We use the Blockly visual programming editor [2] as the vehicle for our demonstration, but the ideas can be applied to any block-based editor. Blockly is a block-based editor similar in many ways to Scratch, where blocks representing computations can be snapped together to create programs. Below are sample blocks from the traditional Blockly UI:

The shape of all connectors is a groove reminiscent of jigsaw puzzles. Note that values flow right-to-left.

We extend Blockly with Polymorphic Blocks to produce *Polymorphic Blockly*, where the shape of connectors represent the types of values flowing through those connectors. Figure 3 shows several basic expressions with their types, along with the Polymorphic Block for the expression – "basic expression" here means that the representation in Polymorphic Blockly is a single block. We use the Java and C style of writing types for functions, where the return type is written first, followed by the name of the function, followed by the types of parameters. We use the naming convention T (with possible subscripts) for type variables, akin to the generics found in Java. Because values in Blockly flow right-to-left, the connectors of Polymorphic Blocks are on vertical edges, as opposed to horizontal edges. Note that all the Polymorphic Blocks in Figure 3 use red and blue for port colors, and this is because Figure 3 shows how each block appears when it is created in isolation (i.e.: without other blocks on the screen). More generally, our implementation keeps tracks of what port colors are in use, and generates unique colors for ports when a new Polymorphic Block is created, with red and blue just being the first two colors. As a result, if one were to create multiple blocks on the same screen, we would in fact see many more port colors than in Figure 3. The importance of generating port colors in this way will be explained shortly.

Integer values are represented using a semi-circle, boolean values using a triangle. Note that semi-circle here is not used as a shape-constructor, as is done elsewhere in the paper – here semi-circle is just a basic shape. The cond function is a conditional similar to the "?" operator in C, and its Polymorphic Block encodes the fact that the types of the two parts of the conditional should be the same. The *List*$[T]$ type is represented by adding a single small side-ways "l" next to the shape of the T type. The *Pair*$[T_1, T_2]$ is represented by putting a small triangular wedge between the shapes of T_1 and T_2, and then surrounding the entire shape using two slim barriers. Note that the "create pair" block has a single output connector which is visually large because it represents a pair, and similarly, the "first" and "second" blocks have a single large input connector each.

Figure 4 shows some more complex examples involving multiple Polymorphic Blocks. The occurrence of ". . ." in an expression indicates that the given part of the expression has not been filled in yet (or in the Blockly perspective, has not yet been connected to another block). In the first two examples, once an integer is connected to the red Polymorphic Port, all red ports take on the semi-circular shape representing integers. The same happens when a triangular boolean value is connected to the red port, as shown in the third example. The next few examples in Figure 4 show pairs.

The last example shows what happens when two colored Polymorphic Ports are connected. In this example, we created a cond block, which Polymorphic Blockly built with a red port, and then a list block, which Polymorphic Blockly built with blue port (since the red color was already in use in the cond block). When the red output of cond is connected to the blue input of list, the red shape connected to the blue input of list is propagated to all blue ports of list, producing the final result shown in Figure 4. Note how the red Polymorphic Port is now *global*, in that it spans multiple blocks. This underscores the importance of generating unused colors when creating blocks: the interpretation of colors is not block-local, and we must therefore avoid coloring conflicts.

Parametric Polymorphism. From a programming language perspective, Polymorphic Blockly implements a typing discipline called *parametric polymorphism* (alternatively called *generics*). Parametric polymorphism manifests itself in the type system as unconstrained type variables, for example in the declaration *List*$[T]$ reverse(*List*$[T]$), where T is an unconstrained type variable that can be instantiated to any type. Generics in Java and C# were initially introduced to enable parametric polymorphism, although generics have since increased in expressiveness to encode more complex kinds of polymorphism. Parametric polymorphism is also at the core of programming languages like Standard ML, OCaml, F#, Haskell and Scala. Polymorphic Blocks encode exactly the notion of parametric polymorphism, with Colored Polymorphic Ports acting as unconstrained type variables. Furthermore, the *Hindley-Milner* type inference algorithm [18], which is at the core of type systems for Standard ML and OCaml, falls out "for free" from the way that Polymorphic Ports get matched against shapes when blocks are connected.

INSTANTIATION TO THE PROOF DOMAIN

It turns out that Polymorphic Blocks are applicable well beyond block-based programming environments. In this section, we show how Polymorphic Blocks can be used to build a novel visual environment for manipulating logical proofs. The shape of connectors will represent formulas, and Polymorphic Blocks will represent inference rules, the proof steps that manipulate formulas. This user interface to logical proofs is entirely visual, completing eschewing symbols. As a result, it lends itself very well to becoming a *Puzzle Game*, which we call the Proof Puzzle Game

We focus on a particular kind of proof system called *natural deduction*, which was introduced by Gentzen in 1934 to closely mimic the "natural way" in which humans think. For simplicity, we consider here the subset of natural deduction involving implication and conjunction. Nonetheless, since what we show here is how to encode the syntactic manipulations of inference rules using Polymorphic Blocks, the ides can easily be extended to more complex forms of natural deduction, and more broadly to other inference systems.

Short Tutorial on Natural Deduction

We start with a quick overview of natural deduction. In the subset we are considering, a formula ϕ is a boolean predicate defined according to the following grammar:

$$\begin{array}{rcll} \phi & ::= & v & \text{boolean variables, e.g.: } x, y, z \\ & ::= & \phi \Rightarrow \phi & \text{implication} \\ & ::= & \phi \wedge \phi & \text{conjuction} \end{array}$$

Note that the variables above are *boolean* variables, meaning that their interpretation is either *true* or *false*.

Examples: (1) x (2) $x \wedge y$ (3) $x \Rightarrow y$ (4) $x \Rightarrow (y \wedge z)$

A *judgment* J has the form $\Gamma \vdash \phi$, where Γ is a comma-separated list of formulas. It essentially states: "using logical reasoning, it is provable that if all assumptions in Γ are true, then ϕ is also true".

Examples: (1) $x, y \vdash x \wedge y$ (2) $x \wedge y \vdash y \wedge x$

To establish that a judgment holds, proof systems like natural deduction make use of *inference rules*. Inference rules correspond to logical steps in a proof, allowing certain judgments to be concluded from other judgments. An inference rule has the form:

$$\frac{J_1 \quad \cdots \quad J_n}{J}\ \textit{Name}$$

The interpretation is that if judgments J_1 through J_n (called premises) hold then judgment J holds. The simplest example of a judgment is the *Assume* rule, which has no premises:

$$\frac{}{\ldots,\ A,\ \ldots \vdash A}\ \textit{Assume}$$

The A above is a so called *meta-variable*, which can stand in for any formula (A is a *meta-variable* to distinguish it from boolean *variables* like x, y, z). The "..." is just an informal way of saying that there can be other formulas in the list. For example, the *Assume* rule can be used to establish the judgment $x, y, z \vdash y$, where A is instantiated to y.

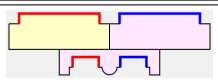
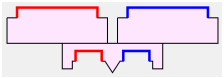
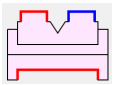
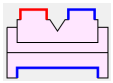
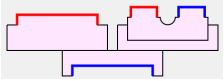
Inference Rule	In Proof Game
$\dfrac{A \vdash B}{\vdash A \Rightarrow B}$ *ImpI*	
$\dfrac{\vdash A \quad \vdash B}{\vdash A \wedge B}$ *AndI*	
$\dfrac{\vdash A \wedge B}{\vdash A}$ $AndE_1$	
$\dfrac{\vdash A \wedge B}{\vdash B}$ $AndE_2$	
$\dfrac{\vdash A \quad \vdash A \Rightarrow B}{\vdash B}$ *ImpE*	

Figure 5. Inference rules with corresponding Polymorphic Blocks

The left column of Figure 5 shows some additional natural deduction inference rules – the right column shows the corresponding Polymorphic Block, and can be ignored for now. Because we will apply inference rules bottom-up, it's easiest to read rules starting at the bottom. For example, the first rule says: "to prove $A \Rightarrow B$, assume A and prove B". This rule is traditionally called *ImpI*, or "Implication Introduction", because when reading the rule top to bottom, we are "introducing" an implication. The second rule says: "to prove $A \wedge B$, we need to prove A and we need to prove B". The "E" in the names of the other rules stands for "elimination".

The inference rules shown here are not the most general versions – instead we show the simplest form of each rule, which corresponds directly to the Polymorphic Blocks shown in the right column of Figure 5. The most general version of each rule has an additional Γ "context" at the left of the $\vdash$, to allow the rule to be applicable in settings where the set of assumptions is non-empty.

A formal proof is a tree of connected inference rules, which is rooted at the bottom with the judgment to prove, and grows upward as inference rules are connected. The proof is complete when all branches are topped with an *Assume* rule. For example, here are two complete proofs in natural deduction:

$$\dfrac{\dfrac{}{x \vdash x}\ \textit{Assume}}{\vdash x \Rightarrow x}\ \textit{ImpI} \qquad \dfrac{\dfrac{\dfrac{}{y \vdash y}\ \textit{Assume} \quad \dfrac{}{y \vdash y}\ \textit{Assume}}{y \vdash y \wedge y}\ \textit{AndI}}{\vdash y \Rightarrow (y \wedge y)}\ \textit{ImpI}$$

Natural Deduction in Polymorphic Blocks

We now describe how to encode natural deduction using Polymorphic Blocks. Each inference rule is represented as a block; each formula is represented as a connector, where the shape of the connector encodes the formula; each meta-variable is represented as a Polymorphic Port.

Let's first start with a single judgment. Figure 6 shows several examples of judgments that we may want to prove, and the corresponding encoding using Polymorphic Blocks. From the first row of Figure 6, we can see how the different variables are encoded: x as a triangle, y as a "T" and z as a

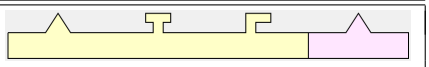
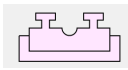
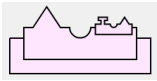
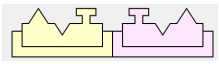
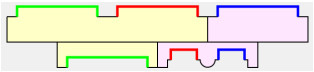
Judgment	In Proof Game
$x, y, z \vdash x$	
$\vdash y \Rightarrow y$	
$\vdash x \Rightarrow (y \Rightarrow x)$	
$x \wedge y \vdash y \wedge x$	

Figure 6. Judgments with corresponding Polymorphic Blocks

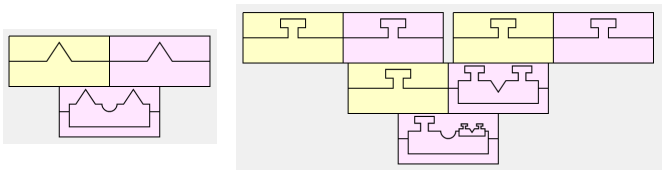
hook-like shape. The list of formulas on the left of the $\vdash$ is shown in yellow, whereas the formula on the right of the $\vdash$ is shown in pink. Although this difference in color is not strictly necessary, it helps to visualize how the blocks line-up. The remaining rows show how $\Rightarrow$ and $\wedge$ are encoded. In particular, $\phi_1 \Rightarrow \phi_2$ is represented by adding a semi-circle between the shapes of ϕ_1 and ϕ_2, and $\phi_1 \wedge \phi_2$ is represented by adding a "V" between the shapes of ϕ_1 and ϕ_2.

Finally, we can return to Figure 5, and look at the blocks in right column. In these blocks, formulas are encoded as connectors, and meta-variables are encoded as Polymorphic Ports of different colors. The structure of each block mimics faithfully the structure of the corresponding inference rule. As was mentioned previously, Figure 5 shows the most basic version of the inference rules, without any Γ "context" to the left of the $\vdash$. We encode additional context to the left of the $\vdash$ by extending the block with additional connectors, each one being a Polymorphic Port. For example, here is a version of the *ImpI* rule which has been extended *once*:



This rule now applies when there is a single formula to the left of the $\vdash$ on the bottom. We provide the player of the game with explicit buttons to add and remove these kind of additional "context" connectors.

Recall the complete proofs we showed previously in natural deduction for $\vdash x \Rightarrow x$ and $\vdash y \Rightarrow (y \wedge y)$. Using Polymorphic Blocks, these two proofs would look as follows:



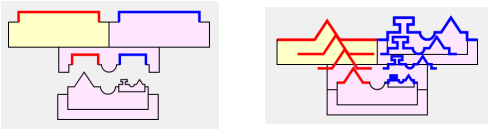
Additional UI for Game

The Proof Puzzle Game contains a few additional UI mechanism to make Polymorphic Blocks easier to work with, and the game more playable, which we describe here.

Assume rule. Although the *Assume* rule can be encoded purely using Polymorphic Blocks, this encoding requires the user to create a block with the right number of connectors and

in the right order, using a variety of block manipulation primitives for adding/removing "context" connectors. Preliminary testing showed that while players were able to create the right *Assume* blocks, it was more tedious than the intuition behind the *Assume* rule. Intuitively, the *Assume* rule is applied by finding the formula on the right of the $\vdash$ somewhere in the list on the left of the $\vdash$. We designed a UI for *Assume* that mimics this intuition: users simply double click on the two connectors that the *Assume* rule is meant to discharge, and the system automatically creates the *Assume* block in one step.

Port Animation. Having users understand the semantics of Polymorphic Ports is crucial: when a port of a given color is connected to a shape, all ports of that color take on that shape. We found through preliminary trials that players had trouble keeping track of the changes if they happened all at once. Inspired by the animated transitions of Heer and Robertson [14], we created an animation where the new shape of a colored port "flies" from the place where the connection was made to all the other occurrences of that colored port. For example, below on the left we show two blocks before they are connected, and on the right we show the animation that occurs when the blocks are connected as a "trail":



This animation makes it much easier to understand the semantics of Polymorphic Ports through visual observation.

Curry Howard Isomorphism

Although our two applications of Polymorphic Blocks seem disparate, they are in fact connected through a concept called the *Curry-Howard Isomorphism* [18]. The Curry-Howard isomorphism establishes a correspondence between types and programs on the one hand and formulas and proofs on the other. For example, the type $Pair[T_1, T_2]$ corresponds to the conjunction $T_1 \wedge T_2$, and the pair function in Figure 3 corresponds to the *AndI* rule from Figure 5. In our setting, the Curry-Howard isomorphism is reified as follows: types and blocks in Polymorphic Blockly have corresponding formulas and blocks in the Proof Game, and vice-versa. For example, the Polymorphic Block for the pair function in Figure 3 is isomorphic to the Polymorphic Block for *AndI* in Figure 5 (where the left-side of a block in Figure 3 is the bottom-side in Figure 5). Similarly, the block for first is isomorphic to the block for $AndE_1$, and the block for second is isomorphic to the block for $AndE_2$. This correspondence through the Curry-Howard isomorphism provides a better understanding of how our two seemingly disparate applications of Polymorphic Blocks are in fact related.

EVALUATION

Our goal in evaluating Polymorphic Blocks is two fold. First, we want to understand how easy Polymorphic Blocks are to use. Second, we want to understand how Polymorphic Blocks compare to a more traditional encoding. To evaluate these research questions, we must pick a domain in which to study Polymorphic Blocks. We pick the Proof Domain, embodied in the Proof Puzzle Game, for the following reasons.

Focus. The Proof Puzzle Game allows us to more narrowly focus on Polymorphic Connectors: since formulas are erased in the game, the only mechanics still in play are those of Polymorphic Connectors. For example, there is no notion of code or code understanding that interferes with our study of how subjects understand and interact with Polymorphic Connectors. Furthermore, Polymorphic Connectors in the Puzzle Game are *essential* – there is no way to remove them, since they have become the primary mode of interaction.

Complexity. The Proof Puzzle Game is a more complex domain as far as the Polymorphic Connectors are concerned, so it will push the limits of the UI further.

Novelty. The Proof Puzzle Game is a more novel domain. As the section on Related Work will explain in further details, while there has been previous work attempting to encode type polymorphism in block-based UIs (although none has achieved the expressiveness that we achieve), there has far less work on designing and evaluating our kind of encoding of logical formulas and logical proofs.

Now that we have focused on a domain, our evaluation goals can be reified as two hypotheses:

1. **Hypothesis 1: Ease of understanding.** The mechanics of Polymorphic Blocks in our Puzzle Game can quickly be understood through self-guided experimentation.
2. **Hypothesis 2: Comparison to traditional encoding.** Our encoding of natural deduction as a Puzzle Game using Polymorphic Blocks leads to (a) smaller completion times and (b) better engagement than a traditional encoding of natural deduction using symbolic inference rules.

Experimental Methodology

More specifically, we gathered 30 high-school students who agreed to be part of our study, and randomly split them into two groups of 15 each. We invited the first group to a session where participants solved a set of proof exercises encoded in the Puzzle Game using Polymorphic Blocks. We call this the Game group. We invited the second group to a session where participants solved the *exact same set* of proof exercises using symbols and inference rules on paper. We call this the Symbols group. Groups were not told a priori what was going to be covered in the session. Participants of each group also completed a common survey at the end of their session. Because of last minute cancellations from study participants, 14 participants attended the session for the Symbols group, and 10 participants attended the session for Game group.

We use the Game group to evaluate Hypothesis 1, and a comparison of the Game vs Symbols group for Hypothesis 2.

Teaching Style. Study participants had no previous knowledge of natural deduction proofs or the Puzzle Game. As a result, we made use of *tutorials exercises* to teach participants how to build proofs, both in the Game group and the Symbols group. A tutorial exercise is an exercise in which participants are guided to the solution step by step. Such exercises are

used to teach new concepts, for example a new puzzle piece or inference rule. In the Game group, tutorial exercises are implemented using an in-game tutorial that guides participants through the solution. In the Symbols group, tutorial exercises are implemented using a lecture-style approach where an instructor shows the solution step by step on a tablet computer connected to a projector. We intersperse tutorial exercises and non-tutorial exercises to create a pedagogical progression of proof exercises.

Furthermore, the Symbols group also received a 30 minute lecture at the beginning of the session to introduce the concepts of formulas, inference rules, and proofs. This lecture presentation, along with the lecture-style tutorials for the Symbols group, were designed with the level of polish comparable to a well prepared lecture for an undergraduate class.

Interface Differences. Participants in the two groups used very different interfaces, offering different tradeoffs. Game participants, through the computer interface, get more *immediate* feedback, whereas the Symbols participants get more *customized* feedback (since it's help from a human instructor). To help reduce these differences, we provided the Symbols group with as much immediate feedback as we could, both through peer instruction (participants worked in pairs) and through checking the correctness of each exercise on the spot (which is not the way traditional exercises are done, and in some sense brings a gamification feel of "level completion" to the pen-and-paper interface). As a result, the Symbols group ended up receiving a learning experience using the best pedagogical practices, including interactive example-based material, peer learning, and quick feedback.

Session Structure and Recorded Data. Each session, which ran 3 hours, started with a main part consisting of required exercises, followed by an open-ended part, in which participants were given a set of "optional" exercises (to test engagement). We recorded completion times, number of exercises solved, amount of help required, and an exit-survey at the end with questions about their experience.

Hypothesis 1: Ease of Understanding

Based on both informal observations, and various measurements that we describe in more detail below, we found that Game participants were able to easily understand the dynamics of the Game through the help of tutorials and experimentation. In fact, all participants finished all puzzles, including the ones in the open-ended session.

The in-game tutorial appropriately guided users through the solution of puzzles involving new pieces. The average time spent on a tutorial level was 22 seconds, far less than the 48 seconds spent on average on non-tutorial puzzles.

Connecting Puzzle pieces. To evaluate the extent to which study participants understood the mechanics of connecting puzzle pieces, we collected the total number of times users tried to connect pieces and out of this total, the number of times they tried to connect pieces that did not match. We only consider the counts for non-tutorial levels, since tutorial levels explicitly ask players to connect pieces that don't match for pedagogical purposes. We recorded a total of 541 connection attempts, out of which 4 failed, for a success rate of 99%. Each of the 4 failed attempts occurred in a puzzle that immediately followed a tutorial level, hinting at the fact that the introduction of new pieces causes cognitive load which in turn leads to failed connection attempts. At the same time, we can also see that participants learn quickly from the failed connection attempts, since beyond the one puzzle immediately following a tutorial, they stop making connection mistakes. This leads us to conclude that participants can quickly understand the connection semantics of Polymorphic Blocks from tutorial levels and from their own mistakes.

UI for Assume. Recall that double clicking on matching connectors is a technique that we use for applying the puzzle equivalent of the *Assume* rule. To evaluate the effectiveness of the double-clicking interface, we collected the total number of double-clicking attempts, and of these, the number of failed attempts (a failed attempt occurs because the two connectors did not match). There were a total of 260 double-clicking attempts in non-tutorial levels, of which 53 failed, leading to 80% success rate. Here again, many of the failed attempts occurred right after a tutorial, and the main source of errors stems from confusion as to whether a sub-part of the connector can match. Still, despite these confusions, participants were able to play the game effectively, in essence because the confusion was limited in scope (to puzzles right after tutorial levels and to a handful of additional ones).

Help provided. Participants needed relatively little help to solve the puzzles, and the help was at a higher level than understanding the mechanics of Polymorphic Blocks. Instead it had to do with strategic decisions about what piece to use (keep in mind that the search space is unbounded). We provided two kinds of feedback when participants asked for help. First, if the proof became so big that it was obviously wrong, we told participants to restart the puzzle from scratch. We did this 5 times. Second, if only a small portion of the proof was wrong, we either narrowed down the pieces that participants should look at, or told them to undo the last step, and think carefully about the next step. We did this a total of 7 times. One key observation is that these two kinds of feedback are *extremely uniform*, and can easily be automated.

Hypothesis 2: Comparison of Symbols vs Game Group

Solve time. When pairs in the Symbols group did exercises on paper, they were placed in front of a laptop computer, which not only displayed the exercises to perform but also measured the amount of time taken on each exercise. To make sure exercises were solved correctly, each pair would raise their hand when they were done with an exercise, and we would check their solution. If the solution was correct, we would advance them to the next exercise using a secret key combination on the keyboard; if the solution was not correct, we would give them feedback (the nature of the feedback will be described shortly). Figure 7 shows the average completion time in seconds for each of 29 non-tutorial puzzles, for the Game group and the Symbols group. The Game group has significantly faster completion times overall.

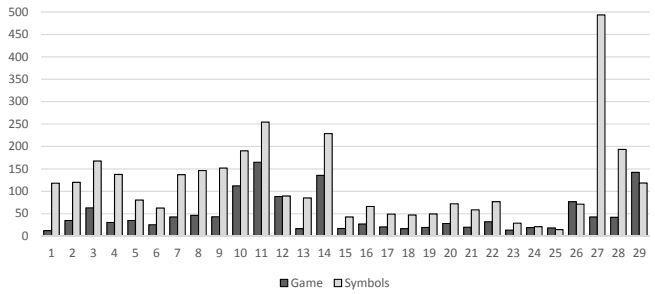


Figure 7. Exercise completion times in seconds for the Game and Symbols group; x-axis: the sequential # of the exercise; y-axis: average time in seconds. The Game group was faster than the Symbols group.

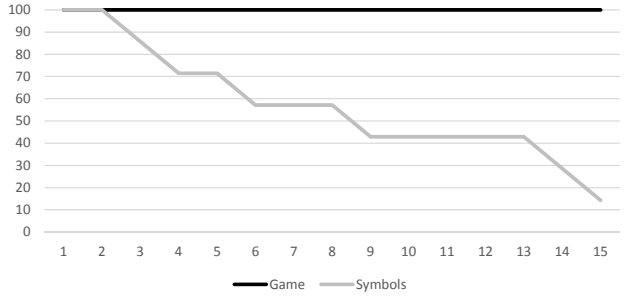


Figure 8. Percentage of pairs who completed each one of the 15 exercises in the open-ended session, for the Game and Symbols group; x-axis: sequential # of the exercise; y-axis: percentage of pairs who completed the exercise. The Game group was more engaged than the Symbols group.

Help provided. We helped participants when they thought they finished an exercise, in which case we helped them understand what they did wrong. We also helped participants when they raised their hand, and stated that they were stuck. The kind of feedback we provided varied, but mostly included: clarifications about which inference rules to apply; clarifications about how to apply a given inference rule; clarification about how substitution works.

Recall that we helped Game participants a total of 12 times (for 5 pairs). In the Logic group, we helped participants a total of 16 times (for 7 pairs). Although the number of times we provided help per pair is similar, the help for the Game group was much easier to administer and would be far easier to automate. This is because the Game group received stylized uniform help, whereas the Symbols group received more customized help. Furthermore, Symbols participants were given a 30 minute lecture at the beginning, including what amounts to instructions on "strategies" for doing proofs. As there is no equivalent in the Game group, this lecture can be considered as additional help that the Symbols group received.

Engagement. To measure engagement, we want to understand the extent to which participants are willing to continue doing exercises if left to their own devices. To this end, following the regular session we presented participants with a new set of 15 exercises, and told them to work for at least 5 minutes, after which they could choose to continue working, or do something else. Figure 8 plots the percentage of pairs who completed each of the 15 exercises in this open-ended session, for the Game group and for the Symbols group. In the Symbols group, there is steady drop-off in the number of solved exercises. In addition to this quantitative data, we also noticed qualitatively that many groups started to browse the web, look at their phones, and play games online. On the other hand, in the Game group, all pairs went past the five minutes, and all groups solved all the exercises. In fact, we noticed that one pair started re-playing some puzzles *after* the were done with the study, without any prompting from us.

Both the quantitative and qualitative data suggest that the Puzzle Game is indeed more engaging. To better understand why this is the case, we can turn to the survey results. In particular, one of the questions on the survey asked participants to rate how fun the experience was on a 5-point Likert scale. The average Symbols response was 3.6, whereas the Game group was 4.3. This difference in the "fun-ness" rating is likely the root explanation for the better engagement. Furthermore, the survey results can also help us understand what led the Game group to have fun. In particular, the survey included a question "What does this experience remind you of?" In response to this question, Game participants consistently used gaming metaphors like "jigsaw puzzle", "Tetris", "oneline puzzle game", whereas Symbols participants consistently used school metaphors like "Math", "Class", "Proofs", and "lecture". This shows that our gamification strategy was successful, in that the Proof Puzzle Game puts participants in a gaming frame of mind. In turn, this is likely an important contributor to the Game participants having more fun.

In summary, our results point to the following conclusion: the Game group had a more Game-like experience, which in turn was more fun, and as a result led to more engagement.

DISCUSSION OF APPLICATIONS

We now discuss our techniques and results in the context of several possible applications.

Learning Logic. Our visualization and gamification of proofs has potential applications for math education. However, it is important to understand some subtle nuances in our results. Although the Puzzle Game is more fun and engaging, it does *not* teach all aspects of symbolic proofs. For one, it does not teach the symbols; but more importantly, it does not teach the full extent of *unification* and *substitution application* because the Game does these automatically (*unification* matches two expressions to determine which meta-variables map to which formulas; *substitution application* replaces a meta-variable with a given formula). Instead, the Puzzle Game focuses on teaching high-level strategies, for example how to pick the right puzzle piece (proof rule), a non-trivial aspect of proof building given the infinite search space.

As a result, we would not expect learning in the Puzzle Game to transfer to the symbolic domain without some additional teaching. The same also holds in the other direction: we don't know if learning in the symbolic domain with pen-and-paper would help learn the Puzzle Game. These questions of knowledge transfer, although interesting and worth investigating, are not the focus of this paper. Instead, our goal in the study was to use a common set of metrics to compare and understand two different encodings, a novel encoding, and a traditional one. In this respect, our experiments support the

following characterization of the trade-offs: while Polymorphic Blocks do not teach all aspects of logical proofs, they lead to faster completion times, and are more fun and engaging than a traditional pen-and-paper interface.

Our results then point to a hybrid approach for further investigation: the Puzzle game can provide a fun and engaging first look, whereas a pen-and-paper example-driven lecture can complete the knowledge about *unification* and *substitution application*. Future work can further explore the viability of this hybrid approach through educational studies.

Types in Visual Programming Environments. Our visualization of types in Blockly can bring the traditional benefits of static types to block-based languages: 1) it would steer programmers away from mal-formed programs 2) it would help programmers quickly see which operations can (or cannot) be connected together. More specifically, once students are faced with many types, they really need to understand the input/output types for each block. Our techniques would provide a constant visual reminder of these types, as opposed to a textual description only visible on the side or in a pop-up.

Although our participants were only exposed to the proof domain, our study also focused narrowly on the mechanics of Polymorphic Blocks, since that's the only mechanics visible to the player (who is not exposed to formulas or proofs). As such, we have reason to believe that our results would generalize to the typing domain, in that the mechanics of Polymorphic Blocks can also be understood in that domain. Future work can investigate this further with additional user studies.

Crowdsourcing Proofs. Our gamification of proofs could lay the foundation for crowdsourcing proofs via gamification, a direction inspired by the Verification Games project [10] on crowdsourcing software verification. Crowdsourcing formal proofs is particularly interesting in light of *Foundational Verification*, a technique that has shown increasing promise in the past decade. In this technique, the programmer writes programs in a proof assistant like Coq, and then interactively proves these programs correct, in full formal detail. Experimental studies have shown that software built this way is far more reliable than software written in a traditional way. The main challenge is that building the proofs is still a painstakingly manual and difficult-to-automate process. The hope is that our work on gamification of proofs can pave the way for crowdsourcing of Foundational Verification proofs.

DESIGN PRINCIPLES AND RECOMMENDATIONS

The Polymorphic Blocks Puzzle Game instantiates five visual design principles: (a) colors and shapes, which are common in the design of virtual manipulatives for math; (b) the Animated Transitions of Heer and Robertson [14]; (c) Direct Manipulation Interfaces [15], most notably giving the impression of manipulating actual puzzle pieces; (d) exploiting cultural conventions, in that we leverage the popular visual cues of jigsaw puzzles and Tetris; (e) Interface Metaphor [8], in which the UI is made similar to a domain already familiar to the user, thus providing intuition on how to interact with the UI.

In our experience, colors and shapes (a) were central in helping users see what shapes match. However, as the complexity of formulas/proofs grow, colors and shapes suffer from scalability issues, which could be addressed in a variety of ways. To avoid overly long connectors, we could add a mechanism to selectively collapse/expand nested shapes. To reduce color conflicts, we note that colors need only be unique within a set of connected blocks, allowing some color re-use. Furthermore, we could continually re-color based on the current viewport so that visible ports use highly contrasting colors.

As stated earlier, we also found that the Animated Transitions of Heer and Robertson (b) were critical in helping users understand the semantics of Polymorphic Ports. Without these animations, too many visual changes would occur at once, and the user would not be able to follow what happened.

Finally, we found that our Interface Metaphor (puzzles for logic) was effective in part because it caused a change in cognitive mindset. In particular, Interface Metaphors vary in how accurately they map elements of the UI to the target domain, with our interface being very close to a precise one-to-one mapping. Larkin and Simon [16] suggest that such a precise mapping may lead to an ineffective diagram representation, as effectiveness is attributed precisely to *differences* in the structure and efficiency of operations. Nonetheless, we found that our puzzle metaphor was effective in part because it leads to a different cognitive state, in two different ways: (1) through the use of cultural conventions, our puzzle pieces put users in a "gaming" (rather than "classroom") state of mind, leading to a predisposed state of enjoyment; (2) the game focuses the user's attention on higher-level strategic thinking.

In addition to leveraging existing principles, our work points to the following more unique principles/recommendations.

Formalism-inspired Design. Our work illustrates how a mathematical formalism (proof theory) can guide an HCI design, yielding several benefits. A well-chosen formalism can be simple, concise, uniform and general, which has the potential to confer similar properties to the UI. The UI's semantics is clean and unambiguous as it is grounded in a mathematical formalism. Finally, one can potentially exploit the connection between the formalism and the cognitive mindset of the user: familiarity with the formalism could help understand the UI, or understanding the UI could help learn the formalism. At the same time, our work also hints at the dangers of following the formalism blindly. Recall that a direct encoding of the Assume rule led to a tedious UI, which we addressed by adapting the UI to match the logical intuition. Our strategy can then be summarized as *formalism-inspired* design: we focus on bringing the structure/intuition of the formalism into the UI, without exposing the symbolic formalism to the user.

Formulas as Shapes. Our work shows that shapes with colored ports can represent formulas with variables. Shape shifting of ports becomes variable substitution. This new visual metaphor can be used for any kind of mathematical expression, and could be applicable beyond polymorphic blocks.

RELATED WORK

Block-based Visual Programming Environments. There are a variety of block-based visual programming environments [5, 2, 1, 6], and our encoding of types using Polymorphic Blocks

could be applied to any of them. We note here two projects that attempt to add types to block-based visual programming environments, although we could not find peer-reviewed papers. First, there is undergraduate honors thesis on *TypeBlocks* [19]. Although *TypeBlocks* uses shapes to represent types, it does not use colors to represent type variables: all type variables look the same and thus *TypeBlocks* doest not implement full parametric polymorphism. Second, there is a Blockly-based editor for the Bootstrap educational program [3]. Instead of using shapes to represent types, their system uses colors to represent types, while keeping connector shapes as in Blockly. Unfortunately, colors are not compositional through the use of constructors the way shapes are, in the sense that it's not clear what color could be used to represent the "list" of another color, or the "pair" of two other colors. For this reason, it's not clear how type constructors are handled in a general way. With respect to these two lines of work, Polymorphic Blocks distinguish themselves by using *both* shapes and colors: shapes to distinguish types, and colors to distinguish type variables. Furthermore, we show how Polymorphic Blocks are applicable *beyond* block-based programming environments, namely for encoding proofs.

Interfaces for proofs. There have been a variety of tools built for interactively building proofs, for example Pandora [7], Panda [13], Theorema [21] and DeduceIt [12]. However, all these tools expose symbolic formulas, something which our gamification approach does not. There has been *one* attempt at gamifying logic *without* exposing formulas: Benkmann built a visualization of natural deduction as a dominos-style game [4] (we could not find a peer reviewed paper). Each formula is mapped to a colorful pattern on a domino tile, and placing domino tiles next to each other is equivalent to applying inference rules. This approach, which is very different from ours, is not evaluated through a user study, and does not generalize to block-based programming environments.

Games with a purpose, for example ESP [20], FoldIt [9], and Verification Games [10], use gamification to leverage human-based computation through crowdsourcing. *Educational Games*, for example Refraction [17] and Codespells [11], use gamification to teach material like fractions or coding. The techniques and domain of our Proof Game are different from these previous games, and our interface is also applicable beyond the gaming interface (i.e.: Polymorphic Blockly).

Acknowledgments. We would like to thank Nadir Weibel and Scott Klemmer for their feedback. This work was supported in part by NSF grant CCF 1423517.

REFERENCES

1. App inventor. `http://appinventor.mit.edu/`.
2. Blockly. `https://code.google.com/p/blockly/`.
3. Bootstrap block editor. `http://bootstrap-block-editor.appspot.com/`.
4. Natural deduction visualized as a game of dominoes. `http://www.winterdrache.de/freeware/domino/`.
5. Scratch. `http://scratch.mit.edu/`.
6. Tinkerblocks. `http://www.tinkerblocks.org/`.
7. Broda, K., Ma, J., Sinnadurai, G., and Summers, A. Pandora: A reasoning toolbox using natural deduction style. *Logic Journal of the IGLP 15*, 4 (2007), 293–304.
8. Carroll, J. M., Mack, R. L., and Kellogg, W. A. Interface metaphors and user interface design. In *Handbook of Human-Computer Interaction*, M. Helander, Ed. Elsevier Science, 1988, 67–85.
9. Cooper, S., Khatib, F., Treuille, A., Barbero, J., Lee, J., Beenen, M., Leaver-Fay, A., Baker, D., and Popović, Z. Predicting protein structures with a multiplayer online game. *Nature 466*, 7307 (Aug. 2010), 756–760.
10. Dietl, W., Dietzel, S., Ernst, M. D., Mote, N., Walker, B., Cooper, S., Pavlik, T., and Popović, Z. Verification games: Making verification fun. In *FTfJP 2012: 14th Workshop on Formal Techniques for Java-like Programs* (Beijing, China, June 12, 2012), 42–49.
11. Esper, S., Foster, S. R., and Griswold, W. G. Codespells: Embodying the metaphor of wizardry for programming. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13 (2013).
12. Fast, E., Lee, C., Aiken, A., Bernstein, M. S., Koller, D., and Smith, E. Crowd-scale interactive formal reasoning and analytics. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13 (2013).
13. Gasquet, O., Schwarzentruber, F., and Strecker, M. Panda: A proof assistant in natural deduction for all. A gentzen style proof assistant for undergraduate students. In *Tools for Teaching Logic - Third International Congress, TICTTL* (2011).
14. Heer, J., and Robertson, G. Animated transitions in statistical data graphics. In *IEEE Information Visualization (InfoVis)* (2007).
15. Hutchins, E. L., Hollan, J. D., and Norman, D. A. Direct manipulation interfaces. *Hum.-Comput. Interact. 1*, 4 (Dec. 1985), 311–338.
16. Larkin, J. H., and Simon, H. A. Why a diagram is (sometimes) worth ten thousand words. *Cognitive science 11*, 1 (1987), 65–100.
17. Martin, T., Smith, C. P., Andersen, E., Liu, Y.-E., and Popović, Z. Refraction time: Making split decisions in an online fraction game. In *American Educational Research Association Annual Meeting (AERA)* (2012).
18. Pierce, B. C. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.
19. Vasek, M. *Representing Expressive Types in Blocks Programming Languages*. Undergraduate honors thesis, Wellesley College, 2012.
20. von Ahn, L., and Dabbish, L. Labeling images with a computer game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, ACM (New York, NY, USA, 2004), 319–326.
21. Windsteiger, W. Theorema 2.0: A graphical user interface for a mathematical assistant system. In *CEUR Workshop Proceedings* (2012), 73–81.