Alternative Programming Interfaces for Alternative Programmers

Toby Schachman
New York University, Interactive Telecommunications Program
tqs@alum.mit.edu

Name2 Name3
Affiliation2/3
Email2/3

Abstract

This is the text of the abstract.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Abstract

2. Introduction

- Somehow set the tone and context
- My motivation and background, ITP
- Lay out structure of the paper

Rethinking causality, focusing on program transformation rather than construction, de-emphasizing numbers as a base representation

I intend to take a very broad definition for the words "programming" and "interface" in this paper. To illustrate scope, in the following sections I define what I mean by "alternative programming interfaces" and "alternative programmers".

3. What are Alternative Programming Interfaces?

I will divide programming interfaces into three overlapping aspects: physical, conceptual, and social.

3.1 Physical Interfaces

Physical interfaces concern the human body and the computer's "body"–its hardware inputs and outputs–and the affordances allowed by each of these bodies.

The dominance of text tends to dictate the form of physical interfaces. The human communicates to the computer through typing on the keyboard. The computer communicates back through whichever outputs the program addresses: the screen, speakers, or other peripherals. But even if a program is intended to create audio or graphical output, throughout much of the programming process the computer communicates to the programmer through text on the screen (through the console).

Communicating text back and forth through these interfaces has proven to be effective because this medium can be made largely unambiguous (through formal languages), it emulates how we communicate intellectually with other humans (talking, writing), and we have evolved conventions (syntax and semantics) for densely packing abstract information into this form.

Alternative physical interface possibilities for programming include:

1. Visual interfaces. These exploit the full graphical capabilities of the screen rather than just text, and often encourage human input through a (singular) spatial pointing device such as a mouse, trackpad, or stylus, in addition to or replacing keyboard input.
2. Touch screen interfaces. Building on visual interfaces, but with the human touching the screen directly. Two potential advantages over mouse-based visual interfaces are a more direct feeling of manipulation of the screen's output, and the expressive possibilities of multiple points of contact (multitouch).
3. Arbitrary interfaces. These include the human communicating to the computer using physical gestures in space, sounds, or the manipulation of peripheral sensors such as knobs and accelerometers. The computer communicates back visually, aurally, or haptically.

None of these alternative physical interfaces have produced widely adopted general purpose programming environments. However, they have had significant success in limited domains. Patching environments such as PD, Max/MSP, vvvv, Quartz Composer, and Isadora use visual interfaces with dataflow semantics to program interactive audio and visual works. Rebecca Fiebrink's Wekinator uses arbitrary inputs (such as cameras and accelerometers) and arbitrary outputs (usually sound) to program novel musical instruments using a supervised learning workflow both on the part of the computer (recognizing human gestures) and the human (learning to "play" the instrument). [http://wekinator.cs.princeton.edu/]

I see three reasons to continue pursuing these alternative physical interfaces, in growing order of importance:

1. We have the technology. The programming interfaces today are largely a result of the technological evolution of the computer going all the way back to mainframes and teletypes. This historical bias suggests that alternative physical interfaces may have dormant potentials.
2. Alternative physical interfaces allow new workflows for programming. Communicating text back and forth is a largely turn-based experience. Each of the alternative interface examples I gave allow a *continuous* feedback loop between human and computer. For example, each of the patch-based visual environments allow the user to adjust parameters, usually with sliders, and see the results in realtime. This "live coding" workflow is possible with text–syntax highlighting is a form of realtime feedback–but the medium does not naturally support it. This is why textual interfaces supporting live coding are often augmented with visual inputs like sliders, as in Bret Victor's "Inventing on Principle" [http://vimeo.com/36579366] and OpenEnded Group's Field [http://openendedgroup.com/field].
3. Alternative physical interfaces engage different parts of the human brain. Textual interfaces engage the "language center" of our brain. We have difficulty expressing concepts to the computer which we cannot translate through this part of our brain. Yet many of our most profound ideas we think of *visually* or *kinesthetically*. Alan Kay relates an anecdote about the mathematician Jacques Hadamard, who polled the great mathematicians and scientists of his day about how they "do their thing." Most replied that they did not think using mathematical symbols (the language center of mathematics) but rather imagined figures or even experienced sensations. Einstein replied, "I have sensations of a kinesthetic or muscular type." [Doing with Images Makes Symbols]

3.2 Conceptual Interfaces

Conceptual interfaces concern the the metaphors we use to think about our programs. Examples include objects, actors, structures, and streams. Conceptual interfaces are largely equivalent with the semantics of a programming language. They are the key mental structures that must exist solidly and isomorphically in the mind of the programmer and the mind of the computer (that is, in its implementation), in order for programmer and computer to collaborate effectively.

There is no shortage of alternative conceptual interfaces, especially at Onward!, so I will cut this section short.

XXX maybe more here

3.3 Social Interfaces

Social interfaces concern programming's relationship to society and how program creation interacts with social systems. It addresses the questions:

1. What is programming and how does it relate to the rest of the world?
2. How should we program together?
3. Who should program?

A traditional, now humorously out-dated view, is that programming is calculating. Computer operators feed problems into a (physically huge) computer which spits out an answer. Calculation (usually prefaced with "cold") is the antithesis of humane, creative activity. We have made inroads towards a new perspective, where programming is seen as a creative, collaborative process between human and machine. This is largely due to pioneering work such as Ivan Sutherland's Sketchpad and Douglas Engelbart's NLS. Yet the traditional view still maintains a hold on the collective (un)consciousness. Many people are intimidated by computers, or intimidated by programming. They see the computer as *the Other*, with whom they cannot engage.

The next question concerns how we relate to our human collaborators in programming. People have always worked together in teams when appropriate, but the internet and platforms such as Github have made the world of code more like an ecosystem than ever before.

The semantics of a language often reflect and reinforce the organizational structures that collaborate using the language. Social interactions are subtle–and I want to avoid making sweeping generalizations–but for the purposes of illustration I will provide a stereotyped example: Java's semantics reinforce an insulated hierarchical organization of programmers where one programmer cannot "step on the toes" of another. Contrast this with Ruby, whose semantics encourage substantial monkeying with the language internals. Ruby's semantics thus require more cross-communicative teams, necessarily smaller, or alternatively the top-down institution of conventions like Rails. I'm not implying that any one way of collaborating is better or worse, just that there is a relationship between programming interface design and the way we work with each other. I see substantial opportunities to research the sociological implications of human collaboration in programming, but this paper will henceforth leave these implications unaddressed.

Finally, the question of who should program I will address in the next section.

4. Who are Alternative Programmers?

Many profound advances in programming were the result of people reconsidering the question, *who are the programmers*? Engelbart's NLS expanded the view of programmers from business analysts and artificial intelligence researchers to any information worker. Smalltalk originally focused on children as programmers. Hypercard was developed and distributed at Bill Atkinson's insistence that "end users" need programming capabilities. [http://www.savetz.com/ku/ku/quick_genius_behind_hypercard_bill_atkinson_the_november_1987.html] Even web programming, at least initially, promoted a culture where anybody could contribute their content or software to the web. [There seems to be a pattern where an environment is developed for alternative programmers, then as a consequence of success is overtaken by "real" programmers. Adobe Flash, originally designed for animators who wanted to add interaction, also follows this pattern.]

I believe a new generation of programmers is emerging. These "alternative" programmers are anybody who, if you ask them what they do, would not reply "I am a programmer", yet who regularly program computers in order to achieve their goals. Alternative programmers can include for example musicians, performers, writers, visual artists, designers, scientists, architects, and activists.

Evidence of this emergence includes:

1. The growth of a DIY hacker culture, with hacker spaces, hackathons, workshops, and meetups. These serve as social support structures for alternative programmers.
2. Platforms and communities built around beginner-friendly, dive-right-in programming, such as Arduino and Processing.
3. The growing use of computers as a means of creative expression, ranging from editing video for YouTube to using Max/MSP for live performances.

The use of computers in general for creative expression prompts the question: Where do we draw the line between *programming* and *authoring*–the use of specialized computer tools to produce specialized results? I don't have a good answer but I encourage the reader to take a broad view of programming. For the purposes of this paper, I will take programming to mean any instance of *designing a system*. Bret Victor's distinction between static and dynamic pictures may also be helpful. [Dynamic Pictures Motivation]

Like Smalltalk or Hypercard, I intend to blur the line between *programmer* and *user*, between programming and authoring. Consequently, throughout this paper the reader is encouraged to play with substituting the words "user" and "programmer".

5. Case Study: Recursive Drawing

To explore alternative programming interfaces, I implemented Recursive Drawing, a port of the textual programming language Context Free[1] to a graphical, directly manipulable interface.

Context Free is similar in thrust to Logo's Turtle Graphics. It is a small, elegant programming language for creating graphics. But the two languages diverge fundamentally in their semantics and the programming experience they induce. Logo has imperative semantics and induces in the programmer a "body syntonic"[2] feeling. The programmer directs a drawing robot's movements through space. Context Free has pure (side-effect free) semantics and induces in the programmer a more abstract, disembodied feeling. The programmer declaratively nests spatial transformations. Each declaration is independent of any global context, hence the name Context Free.

In Context Free, the programmer specifies *rules*. A rule is simply a list of references to other rules, each with a spatial transformation (e.g. translation, rotation, scale) to apply to that rule. There are two primitive rules, circle and square, which simply draw the shape. Rules can reference themselves. See [Figure] for an example. [3]

XXX need CF figure

Through self-reference, Context Free encourages the exploration of self-similar shapes: fractals. These shapes are co-recursively generated infinite structures, the graphical equivalent to Lisp's streams or Haskell's infinite data structures. Execution of a Context Free program can be seen as referentially transparent rule substitution. Context Free features a form of lazy evaluation, in that when drawing to the screen, recursion halts when the shapes are too small to be seen (i.e., a suitably small, sub-pixel size).

Context Free presents a paradox. On one hand, it features semantics which are considered advanced, even esoteric, by the mainstream programming community: referential transparency and co-recursive structures. On the other hand, it is visually intuitive and has been enthusiastically adopted by artists.[4]

My port, Recursive Drawing, was inspired by this paradox, along with Bret Victor's call for directly manipulable programming interfaces[5]. Directly manipulable interfaces not only feature continuous feedback, but also allow the programmer to manipulate the program using a representation that resembles the final output of the program, in this case graphics.

Context Free presented a comparatively easy target for a directly manipulable interface. Its output representation is graphical and its semantics are based around spatial transformations. Thus user interface conventions from graphical authoring tools such as Photoshop could be employed by the programming interface. Additionally, lazy evaluation is a powerful abstraction which allows very small programs, so I could mitigate the information density constraints that often plague graphical programming interfaces.

[1] http://www.contextfreeart.org/phpbb/viewtopic.php?f=2&t=455

[2]

[3] Context Free has many more features, but these basic ones will be sufficient for the purposes of this paper.

[4] cfa community

[5] Dynamic Pictures Motivation

Dynamic interaction is central to the point of Recursive Drawing, so I encourage the reader to watch a short video demonstration of Recursive Drawing or to try it out (in the browser). These resources are available at `http://totem.cc/onward2012`. Several figures showing basic operations are also included in [appendix].

In the following sections I will be contrasting Context Free with Recursive Drawing in order to illustrate alternative programming interfaces. I am not claiming that Recursive Drawing is a better way to program than Context Free, or even that any of these alternative interfaces are better than traditional interfaces. I only wish to illustrate that alternative possibilities are available.

6. Rethinking Causality

Programming is traditionally a forward-progressive activity. We think in procedures: one thing leads to another. We may have a goal in mind, but in order to reach our goal we start at the foundation and build our software step by step. This section explores relaxing this notion of forward progress.

6.1 Cause and Effect

Changing the source code of a program so as to effect a specifically desired change in its output is a very common activity in programming. Indeed this activity can be seen as equivalent to debugging.

Traditionally, if a programmer wants to change the output of a program in a specific way, she must solve two problems:

1. Find the line(s) of code which resulted in the part of the output she is concerned with.
2. Understand the relationship between this code and the output, in order to adjust the code accordingly.

To deal with the first problem, the programmer must trace backwards in the causal chain ending at the output. That is, she must trace back to the line of source code which initially started the chain. Some tools keep track of this chain of causality, and attach this history to the output in some form. For example:

1. Stack traces show the chain of functions which were called to get to a given breakpoint in the execution of a program.
2. Console logging, in the form of logging “got here”, lets the programmer manually do a binary search through the source code, testing whether different parts of code are or are not part of the causal chain of concern.
3. In the canvas drawing demonstration of Bret Victor’s “Inventing on Principle”, the programmer can point at an element on the output picture and see which line of code was responsible for drawing it.
4. A DOM inspector in a browser will allow the programmer to point at an element on the screen and see what node of the DOM tree was responsible for drawing it. However the chain stops there. The programmer cannot see further back in the causal chain to see, for example, what line of JavaScript was responsible for creating that DOM node.
5. Patch-based languages graphically show the flow of data through the system. Because there are no side-effects, the flow of data is equivalent to the flow of causality.

To deal with the second problem, understanding the relationship between code and output, a programmer usually uses some form of test-and-repeat. This may be a purely mental procedure, simulating the computer’s operation step-by-step in her head. It may involve turn-based feedback with the computer: changing the code and recompiling, or interacting with a REPL. It may involve continuous feedback with the computer: live coding.

A program is a collection of causal relationships, and to program effectively the programmer must understand these causal relationships. The tools and processes mentioned above help build this understanding, and help solve the practical need of effecting a desired output with a program. But I believe we can go further by rethinking the nature of our programming interfaces.

6.2 Constraints Generalize Procedures[6]

We currently think of a programming interface as a one-way procedure. An alternative is to think of a programming interface as a two-way constraint solver.

Specifically, we think of compilers as functions. They take input (source code) and return output (a running process). REPLs and live coding environments take this a step further. These we can see as stream processors, taking a stream of input and incrementally modifying a running process in either a turn-based or continuous fashion.

The alternative is to think of the input (code) and output (running process) as related by constraints. The programming interface is a constraint solver. So in the traditional case, when the programmer modifies the source code, these modifications propagate via the constraints to the running process, as normal. However the programmer can also manipulate the running process, and these modifications will back-propagate to the source code.

Whereas Context Free uses the one-way procedural programming model, Recursive Drawing uses the two-way constraint model.

Specifically, when working with a self-referential rule in Context Free, the programmer can modify the transformation under which the rule calls itself. Because the rule is self-referential, this initial transformation gets called on itself iteratively so as to produce different recursive effects. The only way to adjust the recursive effects is to adjust the initial transformation. However in Recursive Drawing, the programmer can modify a shape *at any depth in the recur-*

[6] Gerald Sussman

sion. This modification then back-propagates to the rule definition which specifies the initial transformation. This feature is implemented as a constraint solver (in this case, with a numerical algorithm).

It is important to note that users intuitively think of the common drag-and-drop convention as a constraint-based operation. When the user presses down the mouse button in preparation for dragging, she expects that the mouse pointer and the point she pressed on will remain constrained together. So of course when she moves the mouse, the object she is dragging moves with it. Recursive Drawing's constraint model is a generalization of this convention.

The major design challenge of programming interfaces as constraint solvers is providing the user with the power to specify what is constrained in the current context. With procedural programming interfaces, the programmer modifies a line of source code and expects every other line of source code to stay the same.[7] But if the programmer modifies the output, there may be multiple ways to change the source code in order to produce the output. The programming interface must either infer further, "natural" constraints, or the programmer must be able to manually specify further constraints so as to remain in control of her program.

Recursive Drawing currently solves this problem by only allowing the direct editing of the rule which is currently shown in the workspace. Thus it assumes that every other rule's definition is constrained to its current state. However I don't believe this is always the most "natural" constraint to impose. A further line of research would be to port Recursive Drawing to a touch-screen interface. With this interface, the programmer could drag, or more accurately constrain, with multiple fingers, allowing more expressive modifications to the program.

7. Programming: Construction or Transformation?

Much of the design process in programming is concerned with how information is represented: the model. The programmer carefully chooses the data structures with which the program is built. The programming language designer carefully chooses the primitive constructs with which programmers will build their programs. Both of these workflows reflect a reductionist mindset. Elements are defined by the smaller elements they are made up of. This forms an "abstraction pyramid", with primitives on the bottom and the finished piece of software on top.

The abstraction pyramid has served us well in the past. Reductionism allows us to reason at the relevant level of the pyramid. It can enable us to build quite complex software (tall pyramids) because we can build on previously laid foundations. But a reductionist mindset can also reduce our flexibility and produce cognitive dissonance when a piece of software is approached from a different perspective.

For example, users approach software differently than the creators of the software. Discrepancies inevitably arise between the model underlying the program and the model that forms in the user's mind. Many recognize these discrepancies as the root cause of usability issues. [Design of Everyday Things] Often it is argued that the model needs to be simplified–made more elegant and powerful–so that the user can more fully grasp it. This is often true but it misses a subtle issue. A creator of software is concerned with its reductionist nature–the pyramid of pieces it's made out of. But the user of software is concerned with what she can *do* with the software. That is, the user is only concerned with the aspects of the software which are *operationally relevant* in the context of a larger system. [The Inmates are Running the Asylum]

The same applies to programmers approaching existing code. When we choose our representation for the program, we limit the ways in which we can easily modify the program. By "easily modify" I mean transforming the program without choosing new primitives–what programmers appropriately call "refactoring". This is why experienced developers think long and hard about the primitives they will use before they touch the keyboard.

Is there a way make refactoring cheaper? Is there an approach to program design that will not conceptually lock us in to the primitives we initially choose, so that we can open our minds to the various contexts in which our software might be used?

This is of course a deep challenge, but I believe we can better attack it with a shift in mindset.

I suggest we shift our focus from program construction to program transformation. Instead of concentrating on the primitives and what we can build from them, we concentrate on the transformations we might want to apply to our program in various contexts. In other words, I'm suggesting we think of programs *operationally* rather than *reductively*. We define a program by its relationships to other (potential) programs, not by the atoms which constitute it.

This is analogous to the shift in mindset from set theory to category theory. In set theory, we define a property of a set in terms of its elements. In category theory, we define a property of a set in terms of its relationships to other sets.

I'll provide two examples from Recursive Drawing. In each case, I will show how my initial design reflected a reductionist mindset, then how I rethought the issue from a transformation-centric mindset.

7.1 Relativity

The first example relates to how we traditionally use coordinate systems. In Context Free Art, the underlying representation consists of a hierarchy of coordinate systems. The nest-able coordinate system is a key primitive on which Context Free Art programs are built.

[7] This can be tedious for certain types of modifications, which is why modern IDEs relax this constraint by supporting advanced search-and-replace functionality

Now, recursively nested spatial transformations are intrinsic to the concept of Context Free Art, but their representation as coordinate systems is an implementation detail which is forced on the programmer. Indeed early versions of Recursive Drawing did the same thing. Every coordinate system was explicitly shown in the graphical interface.

[pic from 2]

When we force an underlying representation on the programmer, *program transformations can only be performed with respect to that underlying representation*. Indeed this is the only way to tweak a Context Free Art program. The programmer must tweak a value which makes a change with respect to the coordinate system that the value lives in.

But as Recursive Drawing's interface evolved, I found that it was more intuitive to tweak a shape by transforming it *with respect to the other shapes*. That is, it didn't matter where a shape was or how it was oriented with respect to the underlying coordinate system, it only mattered how it related to the other shapes. Thus Recursive Drawing's canvas–the model it presents to the programmer–has no center, orientation, or scale. Shapes are positioned, oriented, and sized with respect to each other.

7.2 Ontology

The second example relates to ontology: how we divide a program into separate objects, or equivalently, how we define identity. A reductionist mindset implies a fixed ontology. But in life, we can shift between contexts. Each context provides a different way to divide the world. Analogously, we would like our ontology to change depending on the context in which we're working with a program.

A heuristic we can use to produce an ontology in a given context is based on the transformations that the context supports. Given two things, A and B, if in our context every transformation we apply to A also uniformly applies the same transformation to B and vice versa, then A and B can be considered identical in that context.

To illustrate, say we have a rigid body like a coffee cup. It is unclear that this should be a single object if we're looking at it in the context of atoms or quantum clouds. However, in the context of everyday interactions, we can identify the coffee cup as a single entity. We determine this based on the transformations available in our everyday interaction context. If I transform the handle of the cup by lifting it two feet upwards, then the rest of the cup is also lifted two feet upwards. Rigidity by definition implies that a transformation on any given point of the object must apply uniformly to every other point on the object. In this way, an operational context–a set of allowable transformations–implies an ontology.

This principle was violated in early versions of Recursive Drawing. In an initial design, a primitive shape (circle or square) was always individually highlighted when the programmer hovered her mouse over it. This was intended to show the relevant parts of the abstraction pyramid, to help the programmer comprehend the reductionist model. But in user testing, I realized that the feature was confusing. It made users think they could only manipulate primitive shapes individually, not understanding that they could define compound shapes out of the primitive shapes. An improvement was when highlighting was applied to *all* shapes which would transform uniformly if the user started dragging the hovered shape. If it moves the same, it is the same. This more closely mapped a user's intuition about what constituted a singular object.

Focusing only on uniform transformations creates sharp object boundaries which may not always be appropriate. This principle can be extended to other types of transformations. In the real world analogy, a water balloon does not transform uniformly when I move part of it, but this transformation still preserves certain properties of the balloon. Thus I am suggesting that programming environments support an ontology sensitive to the program transformations the programmer is interested in; this ontology will naturally fall out of the structure-preserving properties of those transformations.

In each of these examples, when I started with the reductionist model, the primitives and their combinations determined the transformations that were available to the user. This could be seen as a pernicious form of representation exposure. In the alternate version, the transformations were considered first, then an appropriate model was derived from the transformations.

8. Numbers are Overloaded

Numbers are heavily emphasized in elementary mathematics education and in traditional programming practices. So much so that the general populace thinks that mathematics is *about* numbers and computers are *about* manipulating numbers. Of course the mathematics enthusiast knows that numbers are just one instance of many mathematical objects. Likewise, computers are capable of manipulating any mathematical object. Further, computers afford interface possibilities in working with these objects that paper does not.

Machine learning algorithms, feature space

9. Conclusion

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

[1] P. Q. Smith, and X. Y. Jones. ...reference text...