Representing Expressive Types in Blocks Programming Languages

Marie Vasek

Submitted in Partial Fulfillment of the
Prerequisite for Honors in Computer Science

April 24, 2012

© 2012 Marie Vasek

Acknowledgments

First and foremost, I would like to thank my advisor, Lyn Turbak, for always being there. Lyn is always excited about *everything* and that excitement has infected me. Even his desire for everything to be beautiful has managed to rub off on me, something that my mother would consider to be a "miracle of miracles." Lyn encourages me to do things that I would not have otherwise thought to do. Lyn scares me, but this is not a bad thing - intelligent, driven, loud, strong-willed, colorful people might always frighten me.

I would also like to thank all of the computer science professors that I have taken classes or done research with (Brian Tjaden, Stella Kakavouli, Sohie Lee, Randy Shull, Jean Herbst, Takis Metaxas, Lyn Turbak, Tyler Moore). Thank you for showing me that there might be hope for the person who wishes to spend her entire life solving puzzles.

I would like to acknowledge Chelsea Hoover who drafted some original designs for these expressive types. Her work has inspired mine.

I am grateful for Emily Erdman and Karishma Chadha who worked alongside me this winter, particularly for discussions of polymorphism, which have influenced my design.

My friends have definitely helped me through this year. In particular, I'd like to thank Shannon for taking me out to coffee to work on this project, Anthea for taking me out to coffee to *not* work on this project, and JJ for helping me see things more clearly.

I am thankful for my coworkers at StopBadware (Max Weinstein, Isaac Meister, Caitlin Condon, Matthew Frost) for listening to my ramblings about this project and creating a safe workspace. Working there through the summer and this school year has been an amazing experience.

I'd like to thank my family. Thanks to my little brother, Reid, for always being supportive and accepting of me and to my older brother, Arthur, for indirectly teaching me most things I came to know before college. I'd also like to thank my parents for encouraging me to come to Wellesley and supporting me throughout my time here. Thanks for supporting me throughout my life, even when I did not want or ask for your support.

Finally, none of this would have come to fruition without the inspiration of my role model in life, Eleanor Roosevelt. Her words have been influential throughout my entire life and without her guiding force in my life, I would not be where I am today.

Contents

List of Figures

Chapter 1

Introduction

1.1 Visual Programming

1.1.1 Types of Visual Programming Languages

Traditional programming with textual languages has a high barrier to learning because of the need to master syntax and memorize APIs. Visual languages use graphical features to address these problems. Simply put, visual programming lowers the barriers to programming, whether for the elementary schooler who wants to explore the world around her through code, the college student who is taking her first computer science class, or a programmer who wants to avoid errors in her code or have a brief introduction to new language. Instead of programming by typing out lines of code, users create programs by manipulating visual code fragments on their screen. These fragments are then connected in some logical manner, often aided by the look and feel of the fragment, to create a program. Visual programming allow users to see the flow of the program, which reduces parenthesization errors as well as basic logical errors. Nontrivial-sized visual programming languages collect visual code fragments into groups which implies which fragments are logically similar to others.

The two most common kinds of visual programming languages are dataflow languages and blocks languages. Dataflow programs represent programs as directed graphs where nodes represent operations and an edge from one node to another represents a path along which values flow between operations. These languages can get confusing as programs grow, since it is hard to see the structure of the program. Edges can get crossed, nodes can get stacked, and it can become unclear from a glance how the program should be interpreted. In Figure 1-1, the structure of the program is clear upon inspection, however in Figure 1-2, the structure of the program is not clear, even upon close inspection. Even though many of these languages, such as LabView, have a clean-up method which makes the blocks appear in a more orderly fashion, this is not as helpful as it seems, since more orderly is not very orderly. Also, this is only the part of the program that fit on one computer screen.

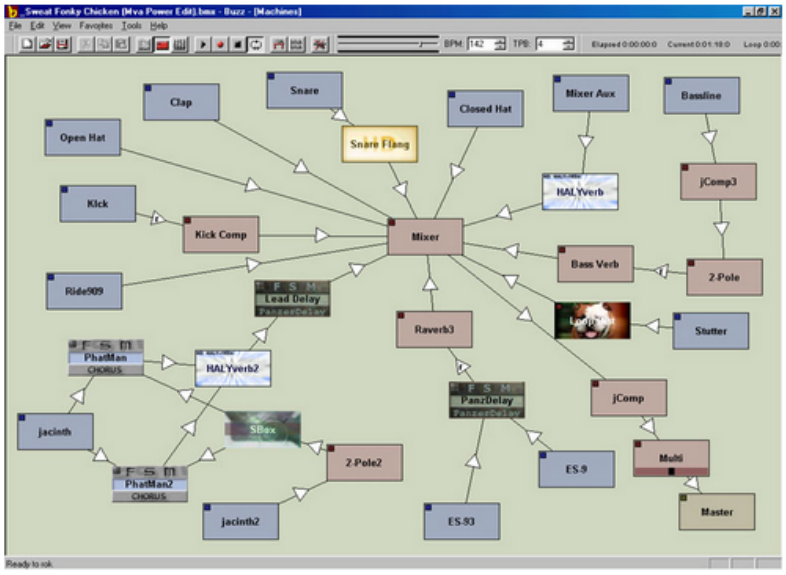


Figure 1-1: An example dataflow program, written in Buzz [21].

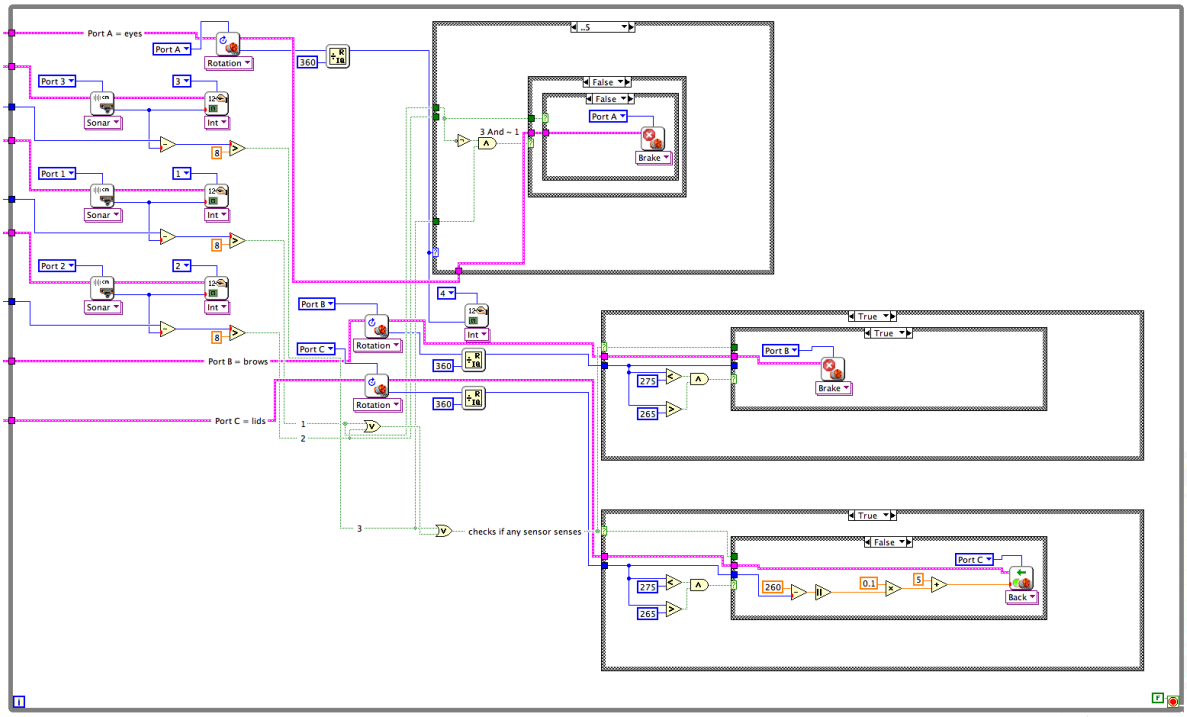


Figure 1-2: Part of a program written in LabView.

Blocks languages represent programs like jigsaw puzzles, where each piece represents a code fragment and two pieces fit together (and look as if they fit together) if the code fragments logically fit together. Blocks languages represent programs like trees which makes the structure of the program clear, at least locally,

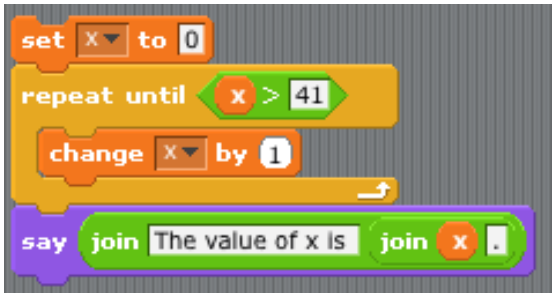


Figure 1-3: A simple program written in SCRATCH.

for arbitrarily large programs, since those blocks are logically connected via juxtaposition on the screen. However, the growth of the programs, where blocks programs expand past the confines of one computer screen, is an issue that blocks do not solve. Blocks languages also more clearly distinguish statements from expressions, which separates blocks explicitly based on functionality.

1.1.2 What is Type

Many introductory computer scientists have come to understand a type system as "the part of a compiler that tells you it doesn't like your program" [1]. However, this is a naïve understanding of what type is. Type can be formally described as "abstract descriptions of values" [24] or "a linguistic mechanism for reasoning about program behavior" [1]. These definitions highlight that type is not just something that a compiler throws upon a program, but rather a language feature that helps programmers reason about values at a higher level.

Static typing is when the types of variables and expressions are determined during analysis, commonly compilation, time. Some example statically typed languages include JAVA, SML, and C. Dynamic typing is when the types of values are inspected at run time. Some example dynamically typed languages include PYTHON, RUBY, and JAVASCRIPT. For the most part, a program written in a static language that is well typed is guaranteed to run without any type errors. However, in dynamic languages, those type errors are reported at run time.

1.2 Problem with Blocks

1.2.1 Flawed Types Systems

The design of many current blocks languages does not deliver on the expectation that shapes are used to indicate type, or what fragments are valid to connect. It would be logical to assume that two blocks that look like they fit together would logically flow together, but this is not necessarily the case.

App Inventor

In a dynamically typed blocks language, we expect a single connector shape to represent any types of value because types are analyzed at run time and not before. App Inventor is such a language. For example, the sample programs shown in Figure 1-4 shows the usage of the same plug shape to represent numbers, strings, text, and lists.

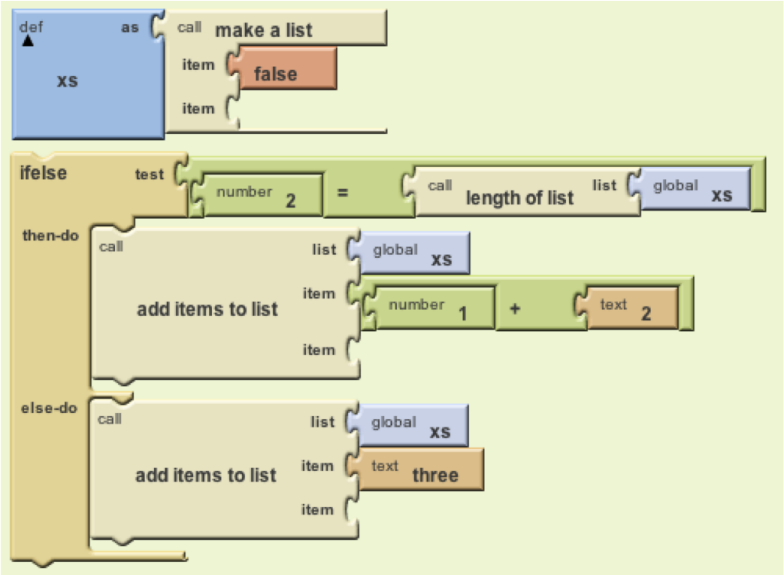


Figure 1-4: An example program in App Inventor.

However, App Inventor attempts to report certain type errors at block connection time in an ad hoc and confusing fashion. For example, Figure 1-5 makes logical sense for every type system to allow. In

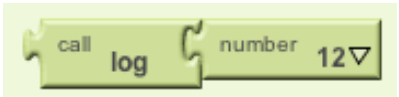


Figure 1-5: A logical statement in App Inventor.

Figure 1-6 App Inventor throws an error since it knows that the `log` function does not accept "hello". However, it does not know whether the `join` function will return a number or a string, so in Figure 1-7, it accepts the top code correctly, where it interprets `log(10 join 24)` as `log ("10" join "24")` to `log "1024"` to `log(1024)`, but it also incorrectly accepts the bottom code, where it throws a dynamic type error, since it does not know how to interpret `log(hello24)`.

Scratch

Scratch is a dynamically typed language (or at least it claims to be) with three different types (boolean,

Figure 1-6: An illogical statement caught by the type system of App Inventor.

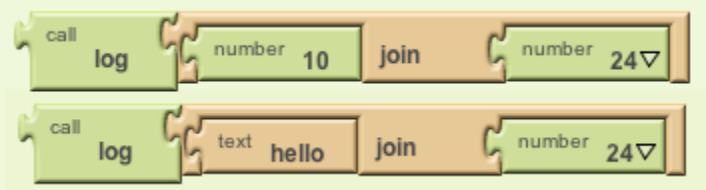


Figure 1-7: Two statements accepted by the type system of App Inventor.

number, string) which are represented by three different shapes (angled, rounded, square) where angled represents boolean, rounded represents number or string, and square represents any of the three types.

Figure 1-8: Returns a boolean

Figure 1-9: Returns a number

Scratch has a type conversion scheme. To make Scratch sufficiently expressive, types of blocks are converted based on context. This allows programmers to do many things that novice programmers find reasonable, such as in Figure 1-11 where `join(1 2)` evaluates to the string "12" and then is converted to the integer 12 and therefore is allowed to be added to 2 to get 14. It also attempts to make all programs

Figure 1-10: Returns a string

run despite potential errors, what the SCRATCH designers call *failsoft*, so when you try to use a string in a similar context, such as in Figure 1-12, the string is converted to the number 0.

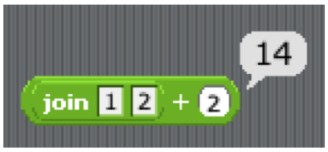


Figure 1-11: Conversion of digit string to number

Figure 1-12: Conversion of non-digit string to the number 0

Figure 1-13: Conversion of the boolean `true` to the number 1

The type conversion of SCRATCH is not as intuitive as the above examples might suggest. For example, booleans are cast to either booleans or strings based on context, as shown in Figure 1-14, where `join` treats the boolean true as the string "true" where as `plus` treats the boolean true as the number 1. Furthermore, when `x` is set to the boolean true, the examples shown in Figure 1-15 evaluate to true. However, the example from Figure 1-16 suprisingly evaluates to false. My hypothesis is that both sides are converted to strings, and the string "true" is not equal to the string "1".

This type system could be potentially confusing to a beginning programmer who is starting to explore what a program means. The ad hoc nature of the typing system makes errors potentially hidden and therefore

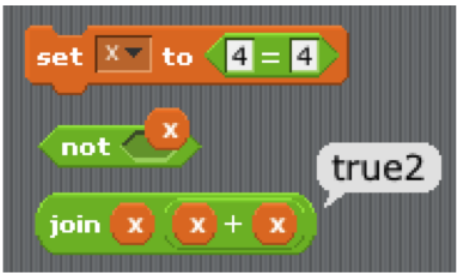


Figure 1-14: Conversion of booleans

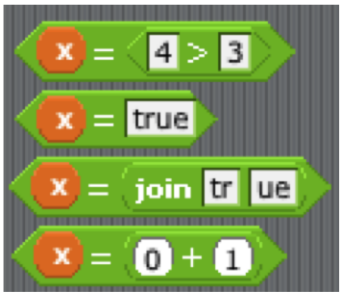


Figure 1-15: Evaluates to true

Figure 1-16: Evaluates to false

confusing to find. The writers of Scratch thought that this was a valid typing system because it relies on a users's natural intuition but I would argue that this reliance is based on an intuition gained from experiences which a programmer might not have [22]. Furthermore, when a program gets sufficiently large, it becomes increasing difficult for even a more experienced programmer to be able to keep track of the varied types of things in her program. Scratch claims to be an dynamically typed language, but the seemingly arbitrary distinction between booleans and everything else and the failsoft nature which tries to convert values into whatever type it expects instead of telling users that they have logical errors, is almost worse, as it gives programmers a false sense of type safeness. Many blocks languages based on Scratch such as BYOB (see Section 2.3) have similar type problems.

StarLogo TNG

StarLogo TNG is a statically typed blocks language that expresses six different types: boolean, string, number, boolean list, string list, number list. (See Figures 1-17 and 1-18.)

StarLogo TNG also has a notion of polymorphism. This is useful, for example to be able to convert

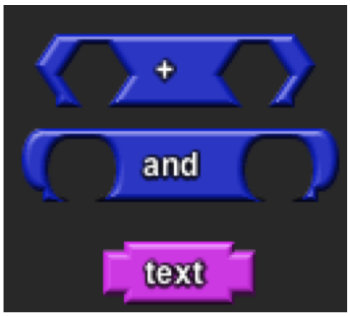


Figure 1-17: Three base types

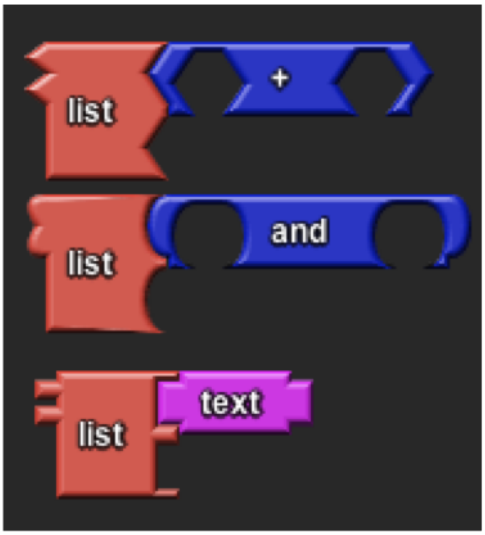


Figure 1-18: Three list types

any input to a string like in Figure 1-19 where the `say` block can accept any expression and convert it into a string for the turtle to say. In Scratch the `=` block accepts any value on both sides, but they could have different types, which allowed comparing any expression with any other expression. In StarLogo TNG, we are restricted to compare any two things of the same type, as shown by Figure 1-20. The `=` block starts out with having both sides of the `=` having type poly, but the moment something is clicked into either side, the other side changes. Here the polymorphic `=` aids in the user's understanding of what equality is.

1.2.2 Finite Number of Types

The sml programming language contains an infinite number of interesting types which would be interesting to be able to represent in a blocks language. For example, the `zip` function, which takes in a list of integers and a list of strings and returns a list whose elements are pairs of integers and strings has type: `(int list) * (string list) -> (int * string) list`. However, current blocks languages only support a limited number of types. The Scratch designers worried that adding more uniquely shaped types over

Figure 1-19: the `say` block

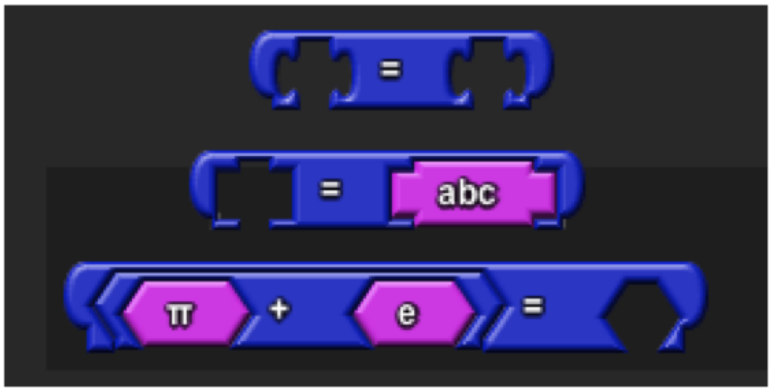


Figure 1-20: the `=` block

their three current uniquely shaped types would "lead to visual clutter and potential confusion" [22] and the OpenBlocks, a blocks framework, designers support fourteen unique connector shapes, where they worried that more types would not be be clearly distinct [23].

In order to be able to express a potentially infinite number of recursive types, some new method of visually representing types is needed. For example, if we tried to make every plug in the same sized box (like all existing blocks languages do), then it would be impossible to draw infinitely many unique shapes that the human eye could distinguish between. Similarly, every type should be algorithmically generated, so that users must be able to distinguish between `int list` and `string list` but still understand that both are lists, and the difference between `int` and `int list` but still understand that the elements of the list are all ints. Now that types are becoming nontrivially complex, the idea of parenthesization will need to be visually represented, since users should be able to clearly tell the difference between `(int * string) list`

and `int * (string list)`.

1.2.3 No Universal Polymorphism

The `zip` function `(int list) * (string list) -> (int * string) list` can only be used with an input that is a pair of a list of integers and a list of strings. A `zip'` function with the type `(bool list) * (int list) -> (bool * int) list` has a different type even though the definition is exactly the same modulo type. This motivates the concept of polymorphism, specifically universal ML-style polymorphism, which expresses computations that look identical modulo certain type information.

1.3 Solution

The goal of my project is to have static, expressive types represented through block connector shapes, where types are generated from a set of base types and a set of type constructors, with an ability to express universal polymorphism.

In my blocks language, TypeBlocks, there are three base types: number, boolean, and string. These base types are represented by three unique connector shapes (Figure 1-21) where angles represent numbers, half-rounds represent booleans, and boxes represent strings. TypeBlocks also has three type constructors,

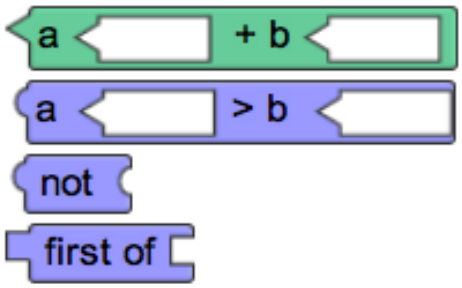


Figure 1-21: number, boolean, and string blocks

list, function, and tuple (Figure 1-22), where the dotted boxes hold the component types.

These then can be combined into type trees in a potentially infinite number of ways. Figure 1-23 shows some sample type shapes.

Blocks fit together if and only if they have the same type. When two blocks that should fit together are moved close to each other, the appropriate socket will highlight, a visual indicator to the programmer that they fit together, as shown in Chapter 3.

TypeBlocks also has a notion of universal polymorphism, albeit not yet perfectly working. The first block of Figure 1-24 shows an identity block which returns something of the same type that it accepts, where that shape represents a polymorphic type. Notice that whenever a block is plugged in, the return type

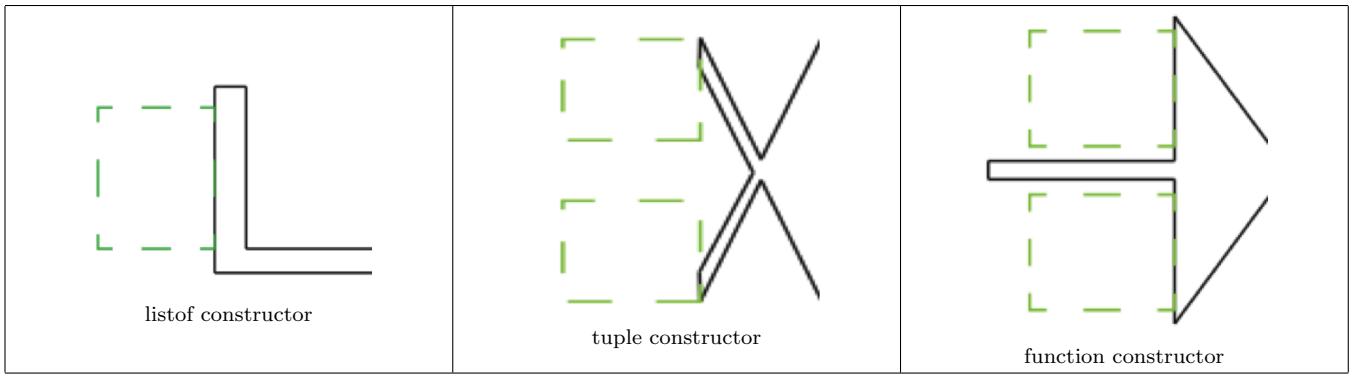


Figure 1-22: Type constructor designs

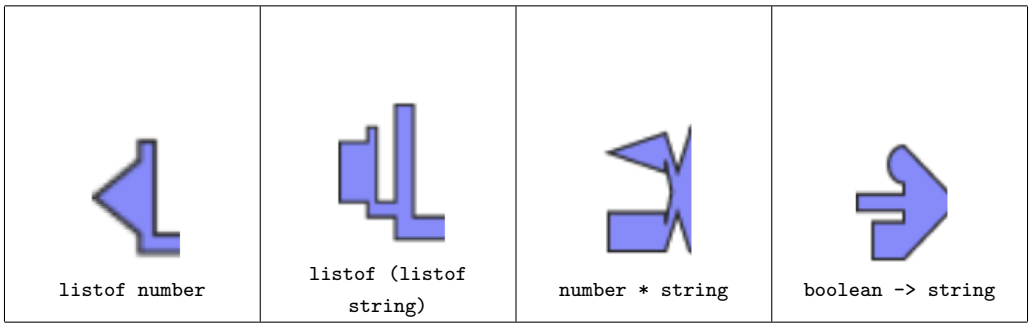


Figure 1-23: Type constructors in use

changes to match that type. Similarly, whenever when this polymorphic block is plugged in as an argument to another block, the argument type changes to match that type (Figure 1-25).

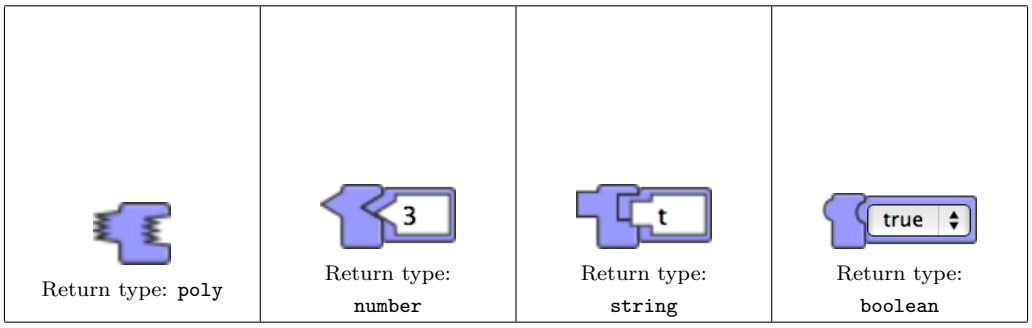


Figure 1-24: Polymorphism

1.4 Road Map

The remainder of this document is organized as follows:

- Chapter 2 gives further details about existing blocks languages, particularly with regards to their type systems. This chapter traces blocks languages back from the first blocks language, LogoBlocks, to one of the most popular blocks language, Scratch, with over a million unique users, through to blocks

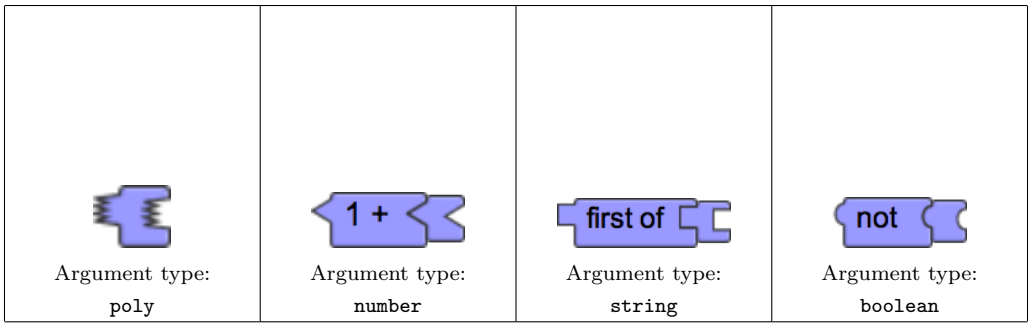


Figure 1-25: More Polymorphism

languages which are still in the early stages of development.

- Chapter 3 talks further about the design of the plug shapes. This chapter outlines the thought process behind designing the plug shapes as well as showing complex types and how the blocks plug together.
- Chapter 4 outlines the implementation details about how the blocks are drawn in code. TypeBlocks is implemented in ScriptBlocks, a blocks framework in JavaScript, and this chapter outlines the changes to that framework that were necessary to make in order to define and draw SML-inspired types.
- Chapter 5 gives a summary of the work done and discusses future related work, such as working towards a blocks language for a SML-inspired statically typed language.

Chapter 2

Background

The first blocks language, LogoBlocks, was implemented in 1995. But it was not until Scratch was developed in 2003-7 that blocks programming (then called "building-block programming") really took off. Many of the current languages are inspired by Scratch, which includes their typing systems. Scratch claims have different shapes for different types where pieces would fit together in only syntactically-correct ways, but this is not the case. This is based on a design decision to eliminate errors, so the blocks instead of complaining, try to "do something sensible"[22]. There have been blocks languages since that have tried to adapt different typing systems (such as App Inventor and StarLogo TNG). However, the type system of App Inventor, while dynamic in spirit, its seemingly arbitrary static error messages seems not too dissimilar from the type system of Scratch. StarLogo TNG seems to be the only blocks language trying to implement a static type system. These blocks languages also only support a small, finite number of different types, which is another issue when working towards a more expressive type system.

2.1 LogoBlocks

Logo is a language developed in the 1960s to teach nontraditional people how to program by creating interesting programs using a LogoTurtle. BrickLogo extended Logo to control a programmable Lego brick which controlled motors and sensors. LogoBlocks was developed as a visual analogue to BrickLogo in 1995 by Andy Begel at the MIT Media Lab and was inspired by the desire to make the language more user friendly.

Figure 2-1 shows what a console in this language looks like. The bar on the left contains all of the possible blocks, and the stage on the right contains the program that you write, a layout that continued with virtually all block-based programming languages to follow. Each connected segment or cluster in the stage logically flows from top to bottom, left to right, similar to how text-based languages flow. However, the placements of the clusters on the stage does not matter.

In this language, there is some distinction of type and some visual clues on how to fit blocks together.

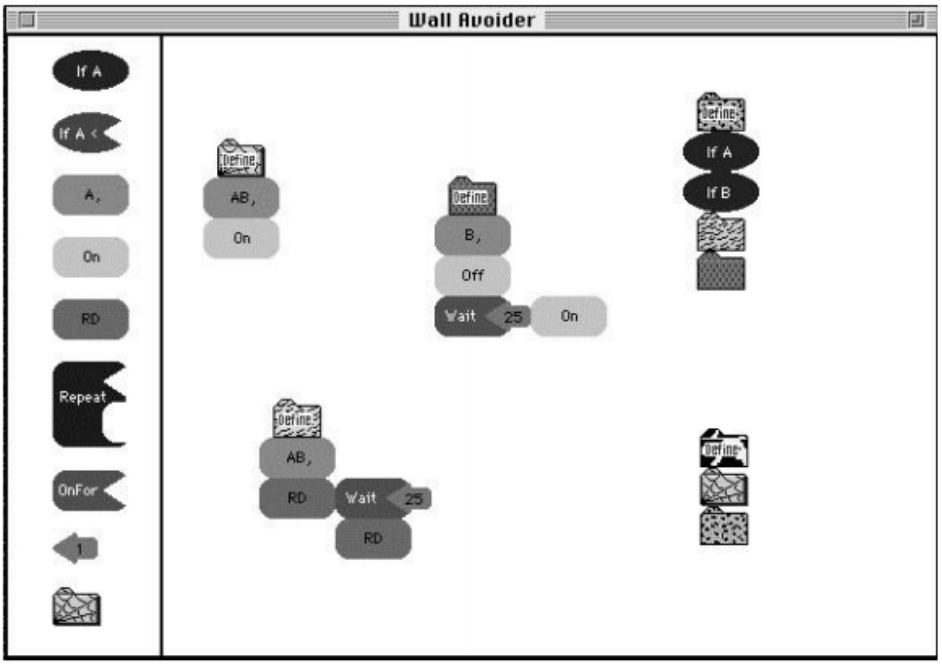


Figure 2-1: A LogoBlocks console [19].

Blocks that have type number have an arrow shape and blocks that take blocks of type number as input have a triangle shaped socket. `if` statements are oval shaped, which distinguishes them from the other statements which are round-edged rectangles. Methods that are "named" are represented as file folders, where instead of being given a name, they are associated with an animal print. Blocks would also snap together if they fit, another clue to how the blocks fit together. However, the design of most of the blocks does not make the flow of the blocks completely intuitive and the size of the block library is limited.

2.2 Scratch

Scratch was initially developed in the Lifelong Kindergarten Group at the Media Lab at MIT [2]. This language was developed with young people in mind to teach them computational thinking while having fun at the same time and was heavily influenced by LogoBlocks. In Scratch, users write programs to manipulate sprites; the default sprite is the Scratch cat. Notice the same toolbar on the left and stage in the middle as LogoBlocks. These blocks all have notches, which control flow through statements.

Each type of value is represented through a different shape. Scratch has three different types (boolean, number, text) which are represented by three different shapes where angled shapes represent boolean types, curved shapes represent both number and text values, and rectangular shapes represent anything as seen in Figures 1-8, 1-9, and 1-10. Each place in a block where another block would fit in, an argument, has an opening shaped according to the type of the argument. Blocks which represent expressions are shaped

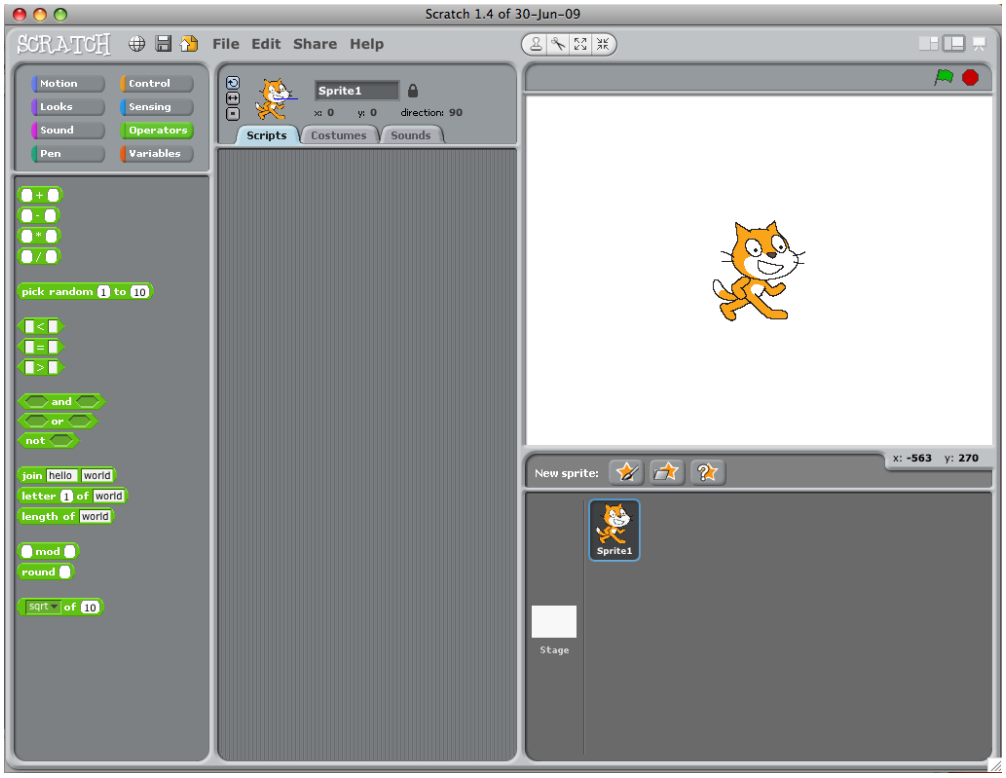


Figure 2-2: A Scratch workspace.

according to which type of value that they return.

Scratch supports global variable creation. Users are then allowed to set the value of `x`, change the value of `x`, and show and hide the variable on the screen. Variables have rounded edges and thus can either represent numbers or text. However, the method to set the variable has a rectangle box which takes anything as input. This makes little sense, since users can store booleans as variables, but are then are forced to either use them as numbers (0,1) or text (`true`, `false`), as seen in Figure 1-14.

Scratch also has a primitive list operators. Users are given a few primitive list operations. However, this distinction between curved and rectangular outputs and inputs is abused in the same manner as it is in variables. For example, users can insert anything into a list, but only remove the inputs as numbers or text. Similarly, as lists are represented as curved boxes, they can be used anywhere numbers and text can be used, where the input is cast into text or numbers in an unintuitive manner. For example, in Figure 2-3, users can add a list to a list. This is represented as a string with spaces, as seen in 2-4. Users are also allowed to do more nonsensical things with this list, such as adding one to the list, such as in Figure 2-5. Using a list as input in a place where a string, number, or boolean is expected just takes the head of the list and applies the operation to that, but this decision is not explicit.

Scratch lists also treat booleans differently than variables do. As seen in Figure 1-16, the variable `x` set to `true` is not equal to the number 1 when treated like a string. However, as seen in Figure 2-6, a

Figure 2-3: Adding a list to a list

Figure 2-4: A view of the list

Figure 2-5: Adding one to the list seen in Figure 2-4

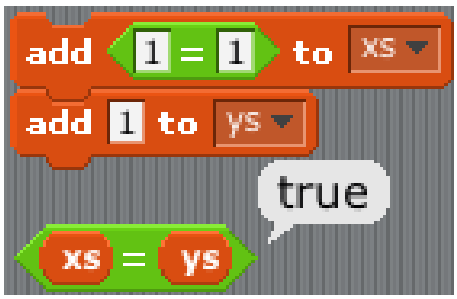


Figure 2-6: Comparing two one-element lists in Scratch

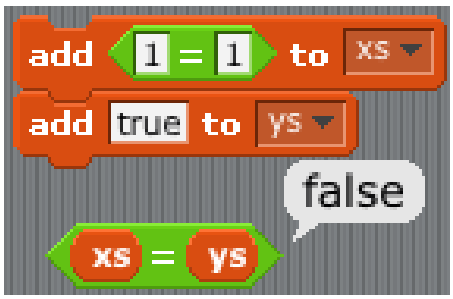


Figure 2-7: Comparing two one-element lists in Scratch

list containing a single element, the boolean `true`, is equal to another list containing a single element, the number `1` in a string context. Similarly, when treated as single element lists, the list `xs` which is the boolean `true` is evaluated as not equal to the list `ys` which is the string `true`, as seen in Figure 2-7. The different treatment of booleans based on list or variable context intuitively does not make any sense. This difference is further shown when again, the variable `x` contains the expression `1=1`, which evaluates to true. If we compare `x` to the first element of a list which contains the expressions `1=1`. this evaluates to false, as seen in Figure 2-8. However, if we store the variable `x` in another list, `ys` and compare the first element of `xs` to the first element of `ys`, such as in Figure 2-9, they are seen as equal. My hypothesis is that variables store booleans as booleans, but lists convert booleans the numbers 0 or 1 when they are added to the list. This is consistent with the representation of booleans on the screen and with this odd behavior.

2.3 BYOB (Build Your Own Blocks) and Snap!

BYOB, or Build Your Own Blocks, targets the Scratch audience and (non-CS major) college students

Figure 2-8: The list element `true` is not equal to the variable containing `true`

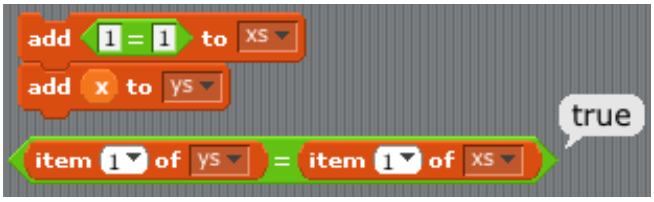


Figure 2-9: unless they are both list elements

and extends Scratch by adding procedures and higher order functions, effectively adding the power of abstraction to Scratch [3]. In order to support these, some additional input shapes were created. BYOB is created by Brian Harvey and Jens Mönig and was recently renamed to Snap!.

BYOB adds functionality to build your own blocks. Users can create any type of block (expression or statement blocks) with arbitrarily many arguments and place this block in any category. The type of the block and the type of the block's arguments defaults to any but users can go to a special menu to define the specific type of the block. However, there are some irregularities in the type system related to building your own blocks.

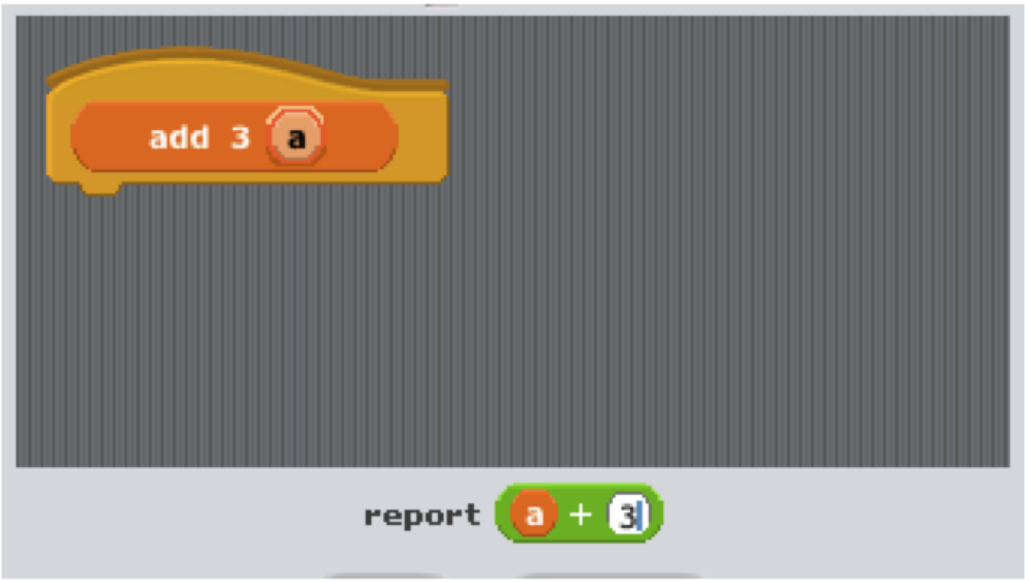


Figure 2-10: Defining an `add3` block

For example, a user can define an `add3` block such as in Figure 2-10, which takes in a parameter and adds three to it. One might expect that this block would logically fit in the same contexts in which that `+` block would, but this is not the case. For instance, in Figure 2-11, the user-defined `add3` block can be used in places where a boolean would fit, such as `not` where the built in `+` block does not fit. This might be because the program knows that `+` always returns a number, but user defined functions might return values of any type. Even though this gives a dynamic type error, this behavior intuitively does not make sense.

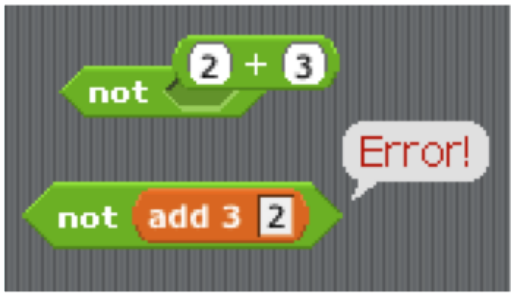


Figure 2-11: Using our `add3` block

Unlike in Scratch, variables can be used in any context, which is confusing, since it allows us to do illegal things which we would not be allowed to do otherwise. For example, unlike in Scratch (Figure 1-

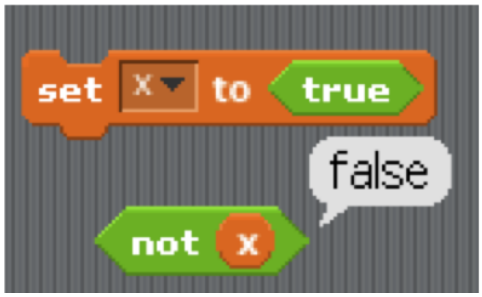


Figure 2-12: variables used logically

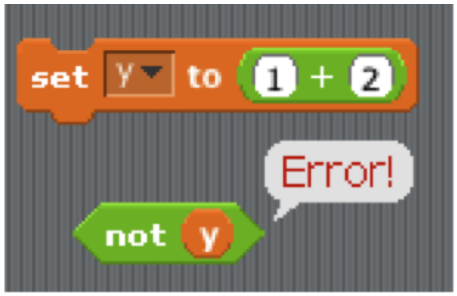


Figure 2-13: variables not used logically

14), BYOB allows users to use variables in a context that expects a boolean, which is shown in Figure 2-12. However, in addition to be able to use variables which are booleans in the correct context, it allows users to use variables of any type in a boolean context, such as in Figure 2-13.

BYOB also supports the use of higher order list functions.

2.4 Other Scratch Derivatives

Panther is aimed at people who have already programmed in Scratch [4]. It adds some cosmetic features (making sprites draggable, get/set methods, pi to 15 significant figures). It also adds file input/output, allowing users to reads and write to .txt files as well as allowing them to start a mesh connection to a local area network. For even more advanced users, Panther allows them to CYOB, code your own block, or write working blocks in Squeak, a language derived from Smalltalk which Scratch is written in.

Other Scratch derivatives of note include Chirp (a precursor to BYOB) [5], DesignBlocks (looks like Scratch but runs in your browser and instead of moving sprites, makes art) [6], and Modkit (looks like Scratch but runs in your browser and instead of moving sprites, produces code in arduino, a language for robotics) [7].

2.5 App Inventor

App Inventor is a blocks programming language for Android mobile applications, initially developed by Google, but has since been transferred to the MIT Center for Mobile Learning [8]. The purpose of App Inventor is for users of mobile phones to be able to create their own applications for their Android phone. The current implementation of App Inventor uses the OpenBlocks blocks framework [9]. App Inventor is dynamically typed. The representation of all values as a single plug shape is consistent with what one might expect of such a language, since in dynamic languages, types are determined at runtime. However, as seen in Figures 1-6 and 1-7, the use of static error messages in App Inventor is potentially confusing to a novice programmers.

2.6 PicoBlocks

PicoBlocks was developed by the Playful Invention Company and was based on research done by the Lifelong Kindergarten Group at the Media Lab at MIT [10]. PicoBlocks was created to write programs for the PicoCricket, a microprocessor-based controller for simple robotics and interactive crafts, and is aimed at elementary school-aged children. PicoBlocks has two different expression block shapes: an oval for booleans and a notched rectangle for everything else, as seen in Figure 2-14. What makes PicoBlocks

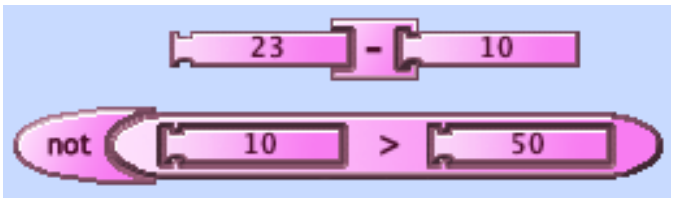


Figure 2-14: Some PicoBlocks blocks

different from the other blocks languages that are mentioned here, is that the block size is fixed and does not grow based on input. This allows us to write code that cannot be parsed easily. For example, the top and bottom expressions in Figure 2-15 say the same thing. However, it is unclear upon inspection what operation is used in the top expression. Now, there exists extender blocks which add no meaning, only readability, for statements but there are no extender blocks for expressions.

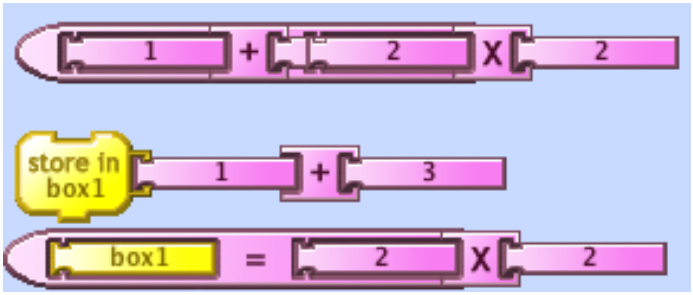


Figure 2-15: Overlapping expressions

2.7 StarLogo: TNG

StarLogo TNG was developed by the MIT Scheller Teacher Education Program using the OpenBlocks blocks framework [11]. This language supports six unique types (boolean, string, number, boolean list, string list, number list) as seen in 1-17 and 1-18. In the program in Figure 2-16, the shape of the different types aids the user to the use of the blocks. It is clear that `x` can be added to `xs` but not to `ys` based on the angle shape of the `x` and `xs` blocks. Similarly, it is suggestive that `xs` and `ys` are both lists, but of different types, base on their shape which is double their respective base type.

StarLogo TNG also supports the notion of polymorphism. This is useful, for example to be able to convert any input to a string like in Figure 1-19 where the `say` block can accept any expression and convert it into a string for the turtle to say. In StarLogo TNG, the polymorphic `=` aids in the user's understanding of what equality is, as shown by Figure 1-20. This block starts out with having both sides of the `=` having type poly, but the moment something is clicked into either side, the other side changes.

StarLogo TNG allows users to define their own procedures, as shown in Figure 2-17. When the program is unable to resolve the type of the output, then the procedure invoker, which is shown below the defined procedure, has no output shape. Then once the procedure has any information about the type of the output, then the type of the procedure becomes defined and this type information defines the output of the procedure invoker. Similarly, the `output` blocks nested inside the `ifelse` block defaults to a polymorphic shape, but then once one branch has a substantiated the `output` type, the other side changes to match.

However, this static typing fails when it comes to defining procedures using list types. This is a result of the decision to let list types be used as input to procedures, but only the polymorphic list type. In the

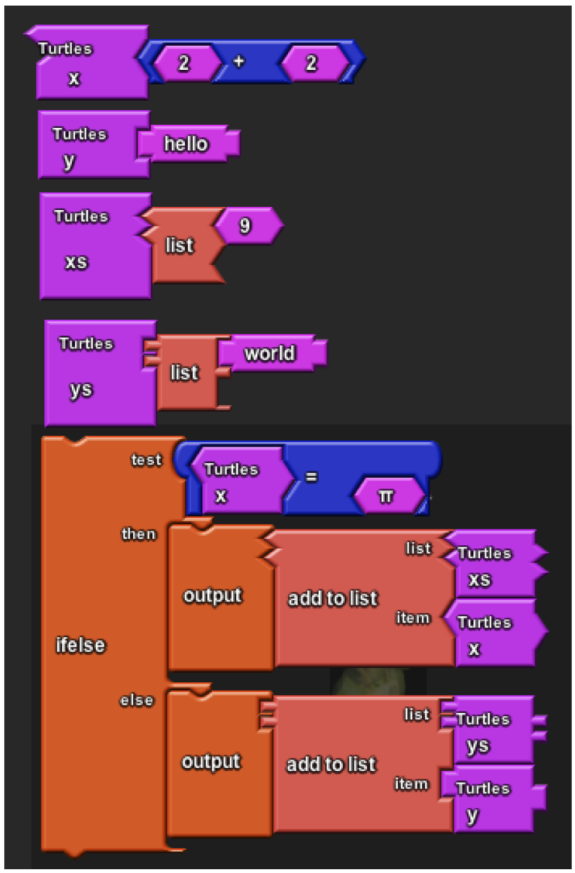


Figure 2-16: An example StarLogo TNG program

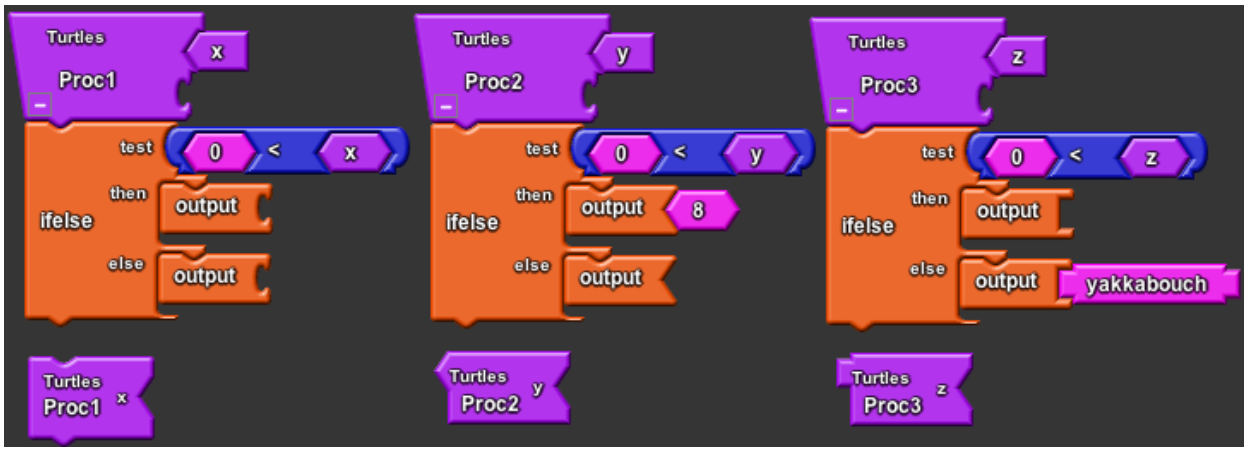


Figure 2-17: Some example StarLogo TNG procedures

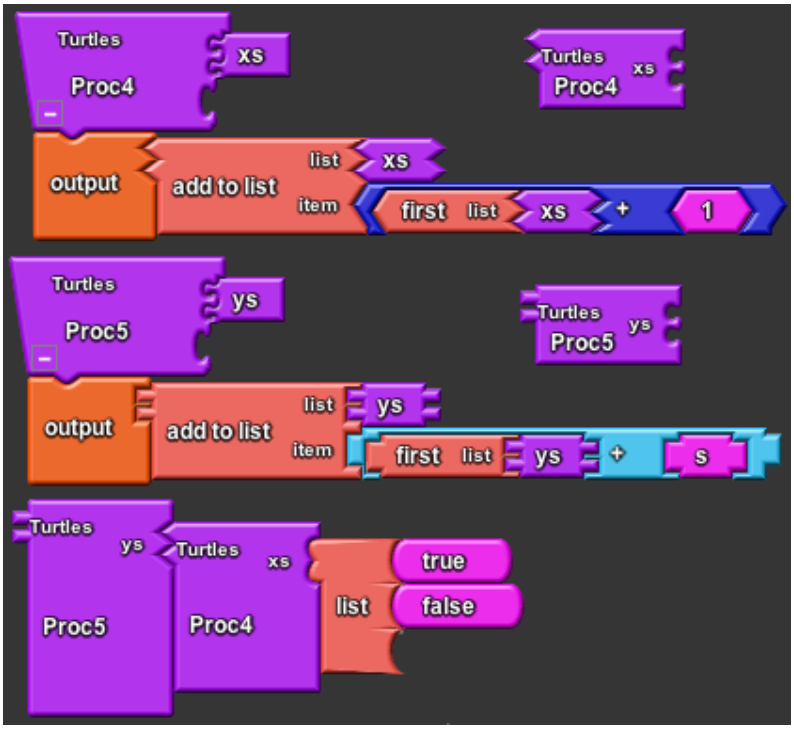


Figure 2-18: A poorly typed StarLogo TNG code fragment

example in Figure 2-18, `Proc4` takes in an argument of type list of poly and returns an output of type list of number. However, since StarLogo TNG only supports homogenous lists, clearly the only well-typed way to use this procedure is with an input of type list of number. Similarly, `Proc5` takes in an argument of type list of poly and returns an output of type list of string, but is only well-typed way to use `Proc5` is with an input of type list of string. Furthermore, we are allowed, without any sort of error, to put an input of type list of boolean into `Proc4` and put the output of that into `Proc5`. This is very clearly not well-typed, but the type system of StarLogo TNG allows us to do put these blocks together.

2.8 Waterbear

Developed by Dethe Elza and inspired by Scratch, Waterbear is a blocks framework makes it easy to define and extend languages [12]. The stated purpose is to eliminate syntax errors, not education, unlike most other block languages. It runs in a browser using HTML5, CSS3, and JavaScript. This language supports two types of blocks (statement and expression) and represents type through color. Even though Waterbear is technically a blocks framework, the primary installation of Waterbear is the JavaScript blocks, which I will refer to as Waterbearfrom now on. Waterbear supports a stage where you put blocks together, a screen showing the code that the blocks correspond to, and another optional screen which shows the code rendering on the screen. Other than those things, it looks very similar to Scratch.

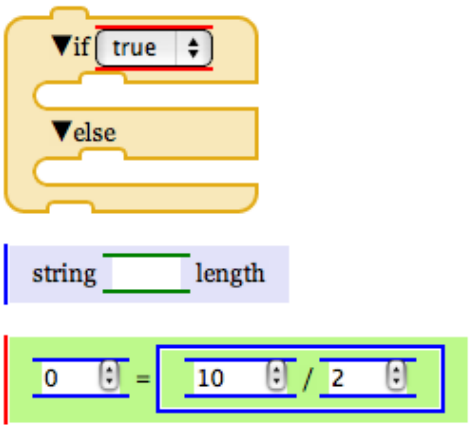


Figure 2-19: Some example WATERBEAR blocks

In the blocks in Figure 2-19, the top block is a statement block and the rest an expression blocks. The color of the actual block refers to where in the block is found in the blocks library. The argument in the top block has a scroll down bar, where you can choose a primitive value, or you can drag an expression into that space. The type of expression that is able to be dragged into that space is defined by the color of the two lines on the top and bottom of the scroll bar. The type of the bottom block and other expression blocks is shown by the color of the vertical lines on the sides of the block. When you drag the expression into the argument slot, they create a red box. This way the number of types that WATERBEAR, in its current state, is limited by the number of colors that a human can uniquely distinguish between.

To write your own language using the WATERBEAR framework, you need to create a JAVASCRIPT file and a CSS file defining the language. The JAVASCRIPT file outlines the categories of blocks (such as control blocks, math blocks, etc) and specific blocks in those categories, where the kind of each block, types of arguments needed, and any text that is to be shown on the block is defined along with what code that the block corresponds to. The CSS file defines the size of the blocks, color of each category, and color of each type.

Currently, there are blocks for JAVASCRIPT and ARDUINO on the Waterbear website [12] as well as followers who have created blocks including blocks for ARDUINO [13], blocks for WPI LibJ robotics [14], blocks for FRC robotics [15], and SCRIPTASTIC blocks for OPENISM[16].

Chapter 3

Shape Types

I set out to represent arbitrarily complex types generated by a set of base types (string, boolean, number) and a set of type constructors (list, tuple, function). I also worked towards implementing universal polymorphism in a blocks environment.

3.1 Preliminary Design

From the beginning, the base type shape designs never changed. Their shapes reflect the shapes from StarLogo TNG. There were also only one or two designs for the constructor shapes. The `list` type shape looks like an "L" for list. The `function` type shape looks like an arrow, since functions are commonly represented as arrows in functional programming languages. The `pair` shape looks like an "X" since pairs are mathematical cross-products. However, the placement of the arguments with relation to the constructor type was an issue that took a few design iterations to settle.

There were many preliminary designs, especially for the function type. Figure 3-1 shows some ideas which did not work, most of which were never implemented.

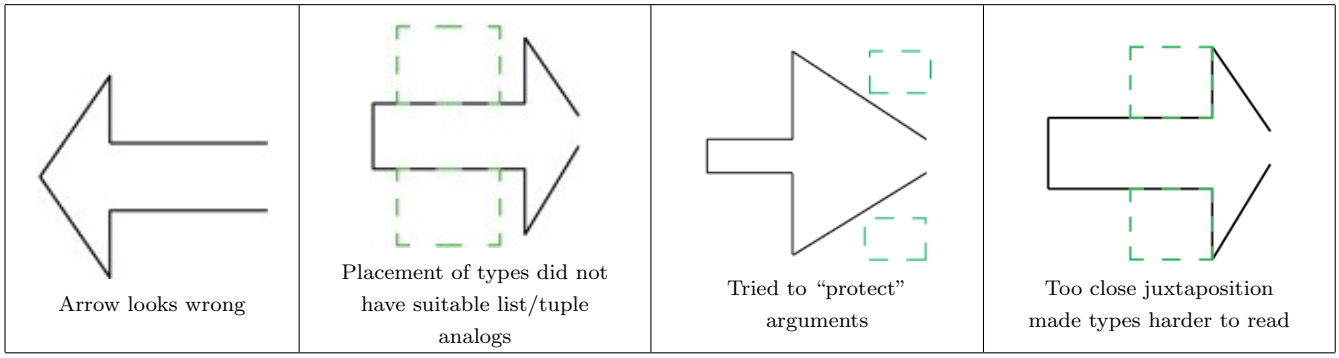


Figure 3-1: Function constructors that did not work

The concept for the first blocks shape iteration was that the base type would be "protected" by the

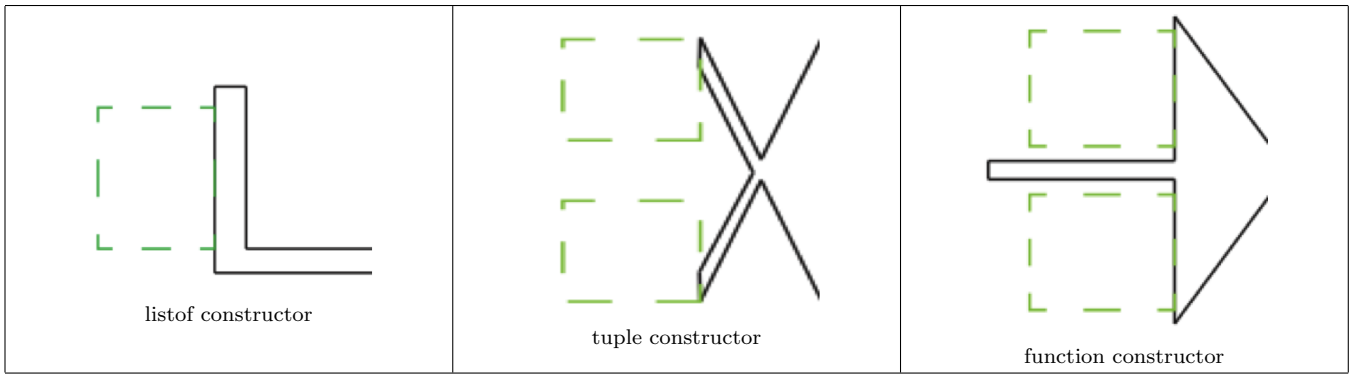


Figure 3-2: Type constructors which worked

constructor shape, so the user would not try to mistakenly try to plug in a block of type `string` into a block of type `listof string`. When implemented in ScriptBlocks, the first iteration of block shapes looked like Figure 3-3. However, this design made types hard to parse. For example, in Figure 3-3, the return type

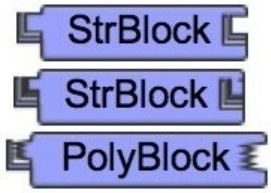


Figure 3-3: The first round of block shapes

of the top block is `listof string` where the return type of the middle block is `listof listof string`, a distinction which is not clear. Also, the difference between the return type of the top block, `listof string`, and the return type of the bottom block, `listof poly` is not as striking as to be intuitive.

To remedy these problems, I made two changes: block sizes depend on the complexity of the type, where each of the base types would be at least as big as the connectors are in ScriptBlocks, and the base types moved to the outside of the type constructors for legibility (Figure 3-4).

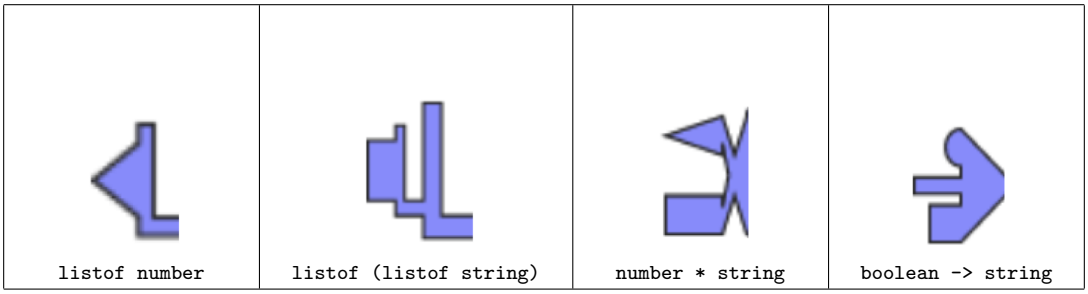


Figure 3-4: Simple examples of type constructors

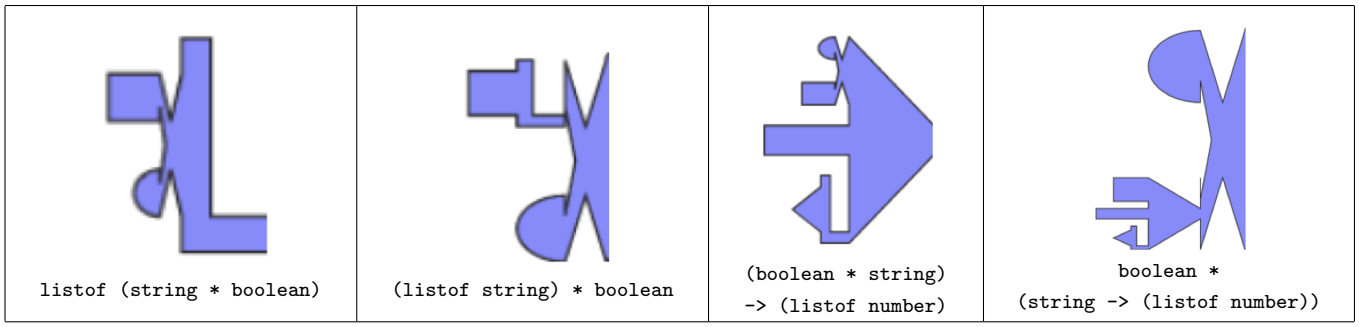
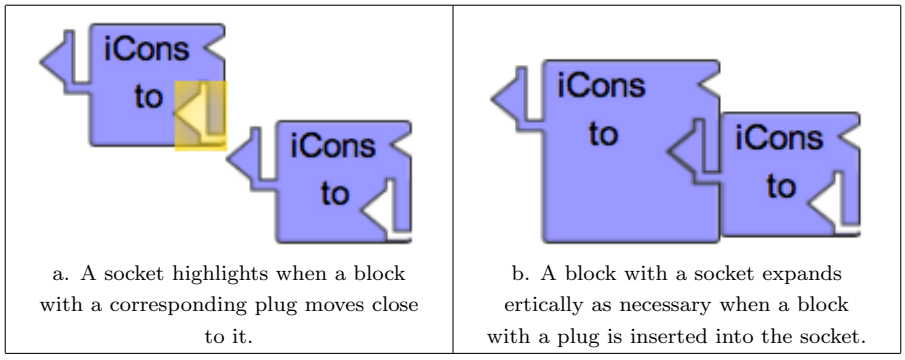


Figure 3-5: More complex type shapes

3.2 Shapes in Action

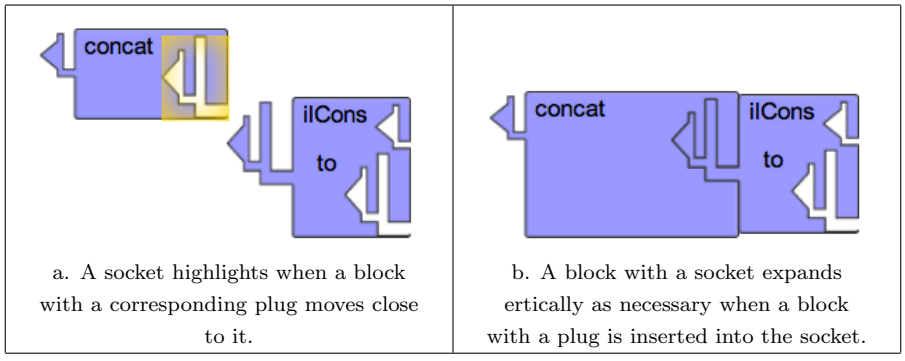
Blocks with constructed type shapes fit together. When block that returns a value with a given type gets close to a socket that accepts that type, the socket highlights. Notice that the highlight size increases as the complexity of the type increases.



a. A socket highlights when a block with a corresponding plug moves close to it.

b. A block with a socket expands ertically as necessary when a block with a plug is inserted into the socket.

Figure 3-6: Two blocks fitting together



a. A socket highlights when a block with a corresponding plug moves close to it.

b. A block with a socket expands ertically as necessary when a block with a plug is inserted into the socket.

Figure 3-7: Two blocks fitting together

However, there are still some unresolved issues with constructor types. As types get more unbalanced, some parts of a type get ridiculously large. For example, Figure 3-8 shows a type which is too large to fit on a screen on a normal resolution and is an astounding 40.5 times the size of a base type.

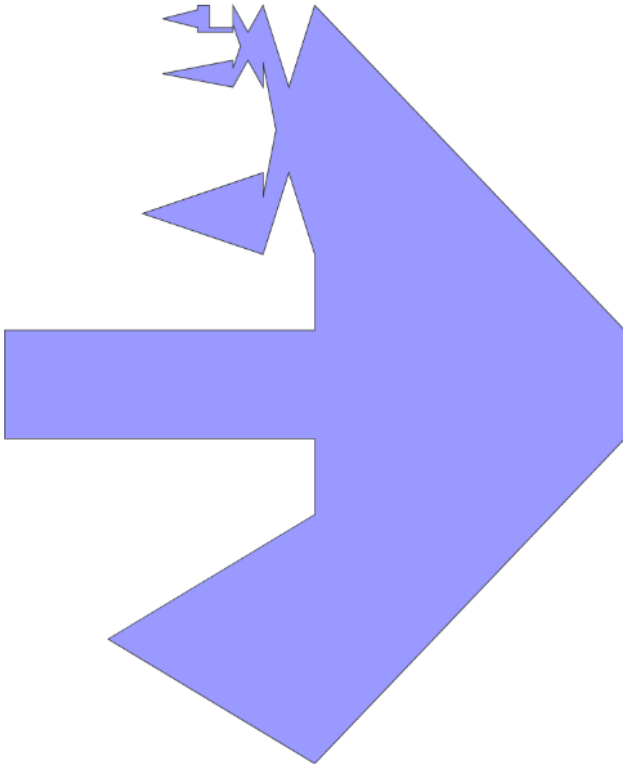


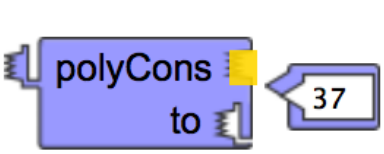
Figure 3-8: `(((listof number) * number) * number) -> number`

3.3 Polymorphism

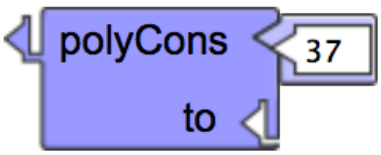
Currently, the polymorphic shape looks like Figure 3-9. This is not the most accurate representation of polymorphism. Intuitively, if something has a polymorphic socket type, it should indicate to the user that something of any type should be able to plug into that socket. This is not the case.

When any type comes near a polymorphic socket, the socket highlights in the same manner that sockets of non-polymorphic types do. However, when the plug clicks in the socket, the type of not only the socket, but all of the corresponding types on the block changes to reflect the resolved polymorphic type (Figure 3-10). Furthermore, this works when any of the polymorphic sockets or plugs are resolved on a block (Figures 3-11 and 3-12). There are some bugs in the current implementation of polymorphism that are discussed in Section 4.6.

Figure 3-9: polymorphic plug shape

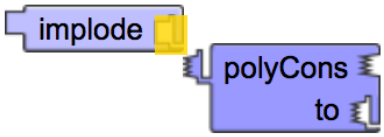


a. A socket highlights when a block with a corresponding plug moves close to it.

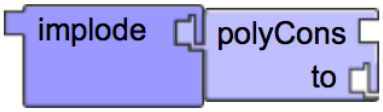


b. A block with a socket expands vertically as necessary when a block with a plug is inserted into the socket.

Figure 3-10: A block with type `number` fits into a polymorphic socket

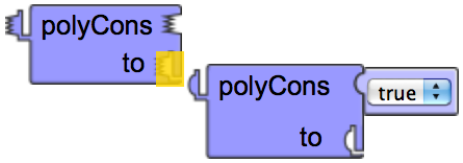


a. A socket highlights when a block with a corresponding plug moves close to it.

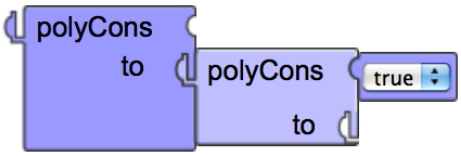


b. A block with a socket expands vertically as necessary when a block with a plug is inserted into the socket.

Figure 3-11: A polymorphic plug fits into a socket of type `listof string`



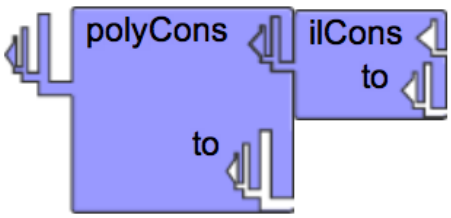
a. A socket highlights when a block with a corresponding plug moves close to it.



b. A block with a socket expands vertically as necessary when a block with a plug is inserted into the socket.

Figure 3-12: A block with type `listof bool` fits into a polymorphic socket

a. A socket highlights when a block with a corresponding plug moves close to it.



b. A block with a socket expands vertically as necessary when a block with a plug is inserted into the socket.

Figure 3-13: A block with type `listof (listof number)` fits into a polymorphic socket

Chapter 4

Implementation

4.1 Overview of ScriptBlocks

TypeBlocks is written in JavaScript and the Google Closure library [20, 17] using the ScriptBlocks blocks framework, which is currently being developed at the MIT Media Lab [18].

In order to run TypeBlocks in your browser, it is currently necessary to run the `plovr` Google Closure compiler. In order to run ScriptBlocks in a browser, it is currently necessary to run a web service that responds to requests for the ScriptBlocks code by dynamically invoking the `plovr` Google Closure compiler on the ScriptBlocks code files to dynamically generate a single JavaScript file implementing ScriptBlocks, combining ScriptBlocks code with the appropriate Google Closure Library code. This web service is launched by executing the `start_plovr_weblogo.sh` command which uses the outline given in the `src/weblogo-config.js` file to know which files and how to compile the code. Further inspection of the `weblogo-config.js` file shows that it defines input files (files which define the highest level API for the language and all use all of the files as dependencies) and paths to all of the files that it might need as dependencies. Notice that the id defined in this file is the same as the id from the `TestWorkspace.html` file in the script tag which loads the JavaScript code. It is important that these match.

ScriptBlocks has three useful base folders: `html`, `style`, and `src`.

The `html` folder contains all of the HTML code used to display the blocks. The only file in here that I use is the `TestWorkspace.html` file where the blocks are defined. For example,

```
var greaterThan = new sb.Block({
    label : "a @arg1 > b @arg2",
    arguments : [ {
        name : "arg1",
        dataType : "number",
        socketType : "internal"
        },{
        name : "arg2",
        dataType : "number",
```

```
        socketType : "internal"} ],
    returnType : "boolean" });
drawer.addBlock(greaterThan);
```

This code defines a block which takes in two numbers and returns a boolean and then adds it to the drawer. The label is what the block text looks like, where @foo signifies that instead of the letters "@foo", that the argument named foo should go there. Each argument has three properties, a `name`, `dataType`, and `socketType`. Each name on a particular block needs to be distinct, but besides that it does not matter. The `dataType` defines the type of the socket, and `socketType` should be where the argument should be placed on the block (internal or edge). This bit of code

```
<script src="http://localhost:9810/compile?id=ScriptBlocks_closure"
type="text/javascript"></script>
```

imports all of the JavaScript files that you need through plovr, which runs on port 9810. This code import the style for the page:

```
<link rel="stylesheet" type="text/css" href="../style/closure/tab.css" />
<link rel="stylesheet" type="text/css" href="../style/closure/tabbar.css" />
<link rel="stylesheet" type="text/css" href="../style/scriptblocks.css" />
```

Make sure that these file paths are correct; so if this HTML file is moved, update them.

The `style` folder contains all of the CSS files needed to format the blocks correctly. The only file in here that I edited was the `scriptblocks.css` file, where CSS for all of the ScriptBlocks elements (drawers, pages, blocks, trash bin, etc.) is defined. The only change I made was to make the page (the area where the blocks are clicked together to create programs) of variable size, so now the `sb.Page` code looks like

```
.sbPage {
    position: absolute;
    height: 100%;
    width: 100%;
    overflow: auto;
    background-color: #FFF
}
```

This says all elements that are in the `sb.Page` class, should be represented on the screen at the size defined, be white, and when you programs grow past the confines of the workspace or blocks are dragged right or down farther than the workspace would statically be, that it should automatically allow it. Most of the CSS for the project is not implemented here, rather it is implemented in the `view` file for the specific element, but this is the most basic, high-level CSS code.

The `src` folder contains all of the JavaScript files necessary to define all of the elements on the page and how they interact.

The `src` folder has three subfolders: `utils`, `model`, and `view`. I completely ignored the `utils` folder. The `model` folder contains files that define the actual screen elements (blocks, drawers, pages). Some important files in this folder include:

- `AllType.js` and `Primitive.js` define types and are discussed in section 4.2
- `BlockSpec.js` defines the properties of a block
- `Block.js` fills in the specification, given by `BlockSpec.js`, for a specific block
- `Argument.js` which defines the logical properties of the sockets on a block

The `view` folder contains files that define how different elements display on the screen. Some important files in this folder include:

- `SocketShapes.js` which draws the connector shapes and is further discussed in section 4.3
- `BlockView.js`
  - defines the CSS elements in a block (`sb.BlockView.prototype.layout` and `sb.BlockView.installStyles`)
  - defines how a block is drawn (`sb.BlockView.prototype.drawShape`)
  - contains other methods that I found necessary to define such as `sb.BlockView.prototype.updateLabelPosition` and `sb.BlockView.hasAnyPolyInType`
- `ViewManager.js` defines how different elements interact, such as:
  - how (and if) blocks click together
  - when different blocks highlight (though where a particular block highlights is defined in `BlockView.js`)
  - how blocks move around the screen.

There were a few major assumptions made in ScriptBlocks which had to be changed to implement TypeBlocks. In the following subsections I explain changes I made and the reasons for doing do.

4.2 Recursive Type Definition

In ScriptBlocks, types are represented by an enumeration of strings in the file `DataType.js` which contains this type definition

```
sb.DataType = {
    BLOCK: "block",
    NUMBER: "number",
    BOOLEAN: "boolean",
    STRING: "string",
    COMMAND: "command"
}
```

In order to extend that to be able to express recursively defined types, I created a file `AllType.js` which defines type using JavaScript objects which contains this type definition

```
/**@typedef {sb.Primitive|{"poly": string} |{"listOf": sb.AllType}|
{"funD": sb.AllType, "funR": sb.AllType}|{"tupX": sb.AllType, "tupY": sb.AllType}} */
sb.AllType;
```

where `sb.Primitive` is defined as

```
/**@typedef {sb.DataType.STRING|sb.DataType.NUMBER|sb.DataType.BOOLEAN} */
sb.Primitive;
```

in the file `Primitive.js`.

These types can then be easily nested. For example, the type `(string * number) -> (listOf boolean)` can be represented by

```
{"funD":{"tupX":"string", "tupY":"number"}, "funR": {"listof":"boolean"}}
```

Representing types as recursively defined objects allows for a greater expressiveness of recursively enumerable types.

4.3 Drawing Recursively Defined Types

Because types are defined recursively, a recursive method to draw types is necessary. ScriptBlocks uses Google Closure paths to draw each block, where the function to draw the connector shape is in the file `SocketShapes.js`. ScriptBlocks draws blocks as a continous Google Closure path starting and ending at the black dot in the direction shown in the arrows in Figure 4-1. Each socket is drawn in the black box, where the only stipulation is that the path for the socket begin and end at the proper corners. However, the plug at the start is drawn from bottom up ("up") and socket at the other side is drawn from the top down ("down"). Originally, the ScriptBlocks `drawSocket` method took as arguments the type of the socket to draw, the path to extend, and whether to draw the socket "up" or "down", and returned the extended path with the socket drawn. The correct socket drawing method then looked up the type in an array of

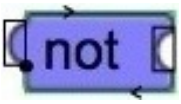


Figure 4-1: Drawing of how the `not` block is drawn

drawing functions and used the proper function to draw the socket. One problem with this method is that each drawing function has to be written twice: once when drawing counter-clockwise and the other when drawing clockwise. This was redundant. Also, since there were finitely many types, there were finitely many

socket shapes. Some of these shapes were not composable: a socket shape was to be drawn in a predefined box, but one shape went outside of the box (Figure 4-2). This function also only allowed a finite number of types to be drawn, since each draw function was "looked up" in an array of draw functions.

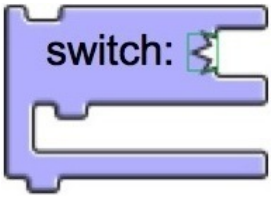


Figure 4-2: Notice how the spikes on the socket jut out further than the space reserved for it

I fixed these problems with drawing sockets by abstracting over orientation of the path and recursively defining socket paths. Instead of thinking about Google Closure paths, which cannot be composed in a way that preserves the background, I thought about composable paths. Then, each recursive type can be drawn by composing composable paths. For example, to draw a function from x to y (as illustrated in FIgure 4-3), first draw the bottom of the arrow (dark blue) then the y argument (green), the middle of the arrow (pink), the x argument (light blue), and finish with the top of the arrow (orange). However, I needed a way of implementing composable paths.

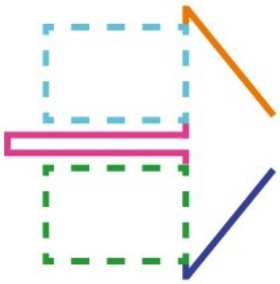


Figure 4-3: Function Constructor

I treated composable paths as sequences of points, where lines are drawn using two points (begin, end) and curves are drawn using three points (begin, corner opposite begin, end) and the corner opposite the beginning point in a curve makes sure the curve "looks right." Here, the curve only uses half of the box, which makes it look more like a half circle (Figure 4-4).

These composable paths are defined by an array of points and an array which indicates whether to treat the points as curves or as lines, stored in a JavaScript object. Each base type and connector shape has a given object of arrays. The program defines the composable path for a particular type by composing objects of arrays according to the types. For example, the object of arrays for a boolean socket looks like

```
{"ptsArr":[x0,y0, x1, y1,x0,y1], "lineOrCurve":[0,0]}
```

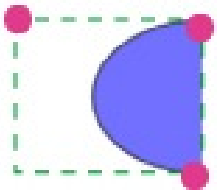


Figure 4-4: Pink dots represent points defined

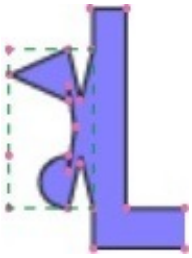


Figure 4-5: Pink dots represent points; green box represents to list argument space

where `(x0,y0)` is the beginning point of the path and `(x1,y1)` is the point opposite it in the defining rectangle. Similarly, the object of arrays for a string socket looks like

```
{"ptsArr":[x0,y0,x1,y0,x1,y1,x0,y1], "lineOrCurve":[1,1,1]}
```

Since each path is defined by a rectangle that it is drawn in, there is no need to do any affine transformations; simply defining the rectangle does any transformations necessary.

Since the path needs to be oriented the correct way so that the rest of the block will be drawn correctly, the points in the array of points and the corresponding curve indicators are reversed if drawn the opposite way. This avoids redundancy in the current implementation of ScriptBlocks. The draw function I designed takes account of the recursive nature of my types.

4.4 Block Size

It became quickly apparent that not all blocks could have uniform size of connector shape, since the base types which were composed to make the recursively-defined types shrank quickly to become indistinguishable from each other. Once the connector shape grew, the block size also had to grow. This violated many built-in assumptions in ScriptBlocks.

Instead of having one default size for every socket, I made the size of the socket depend on the complexity of the block, such that the base types were about the same size as they would be if they were a simple type. In order to do this I defined a function in the `SocketShapes.js` file called `findTypeHeight` where the height was calculated so that the smallest base type would have height "1". This worked by recursively finding the height of the argument(s) and multiplying the argument by a scalar (in the case of 2 arguments, taking the

maximum of the two heights). This scalar is the reciprocal of the amount that the draw function resizes the argument. The draw function for list makes the argument 2/3 the original size, so the height function multiplies the argument height by 3/2. Similarly, the draw function for the list or tuple type makes the argument 1/3 the original size, so the height function multiplies the argument height by 3. Base types have height 1. Then height of the block depends on the complexity height of the return type of the block and the sum of the heights of the argument types (Figure 4-6).

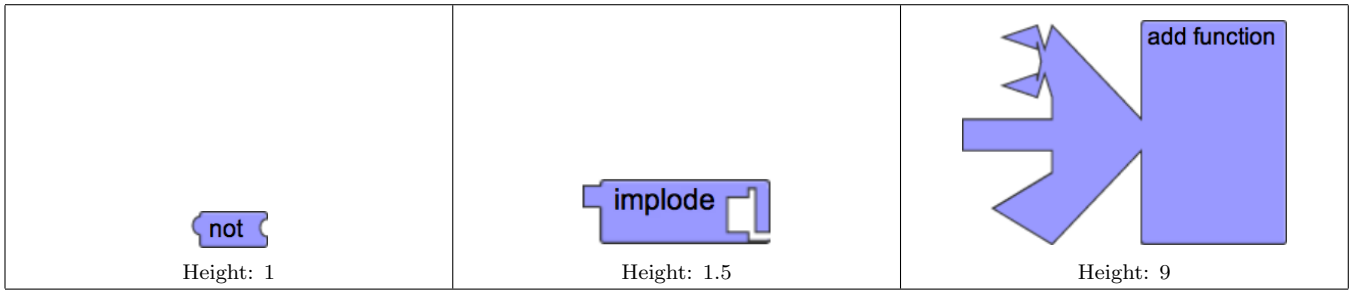


Figure 4-6: Heights of example blocks

4.5 CSS

CSS is the mechanism used to specify styles for blocks. There were many issues dealing with CSS. In ScriptBlocks, each element (page, drawer, block, socket, etc.) is put into a class that has some associated CSS definitions. Then certain properties are defined by CSS, such as how the blocks look like when they fit together. This became problematic, since some many properties were assumed to be the same for every block, such as the size of the connector.

The highlight rectangle for the sockets needed to be changed to reflect the size of the argument type as well as the placement of the label for the name of the block. This was changed by allowing some properties to be universal (like the color of the highlight rectangle) but other properties to be unique to the socket (such as the size of the highlight rectangle). Then the universal properties were not changed, but the individual properties were defined upon creation of the socket, since the size and placement of the socket determines the size and placement of the highlight rectangle.

Also, the socket CSS also had to be changed so that when a block was joined to another, they looked like they fit together. This was changed by looking at the difference of the complexity of the plug and the socket. Then this difference is used as padding for the child when it is connected to the parent. If there was a negative difference, then the return type of the child is more complex than the argument type of the parent, and would have, without the change, appeared with space in between the child and the parent, but with the change be connected visually. If there was a positive difference, then the argument type of the parent is more complex than the return type of the child and would have, without the change, appeared to be overlapping, but with the change be connected visually. If there was no difference, then the original

ScriptBlocks method would have handled it correctly and no change would be made. Associated with this was a change to the original CSS when the blocks disconnected.

There was also an issue of the label of the block. In the `BlockView.js` file, I made a function which updates the label position so that the label did not overlap the drawing of the plug. This change also made it so the width of the block was correctly calculated. This is because each block has a layout element at its base, the size of which approximates the size of the block. Then, each property with an associated layout element (socket, label, arguments) is connected to the layout element. Then the width of the block is the width of the socket + the width of the label + the width of the largest argument.

4.6 Polymorphism

`BlockView.js` contains all the functions used to attempt universal polymorphism. `findAffectedPolyArgs` and `findOtAffectedPolyArgs` find all of the polymorphic arguments that need to be changed. I edited the `addChild` method to change the type on both the child block and the parent block to be the correct type (where the "correct" type is found by the `getMoreComplicatedType` method). The two defined methods which do not work properly are `chainPolysUp` and `chainPolysDown` which attempt to change the appropriate polymorphic types to other types looking up and down the block tree.

However, this implementation fails. Two examples are shown in Figure 4-7. In order to fix these, it will be necessary to fix the `chainPolysUp` and `chainPolysDown` so that they chain. This will be the majority of the work to fix the implementation of polymorphism. Since I am not sure how close my current methods are to chaining, this could take anywhere from a few hours to a few days of constantly working. Once this is figured out, the functions to reset the polymorphic types upon disconnect should logically be similar, so not much additional work would be needed.

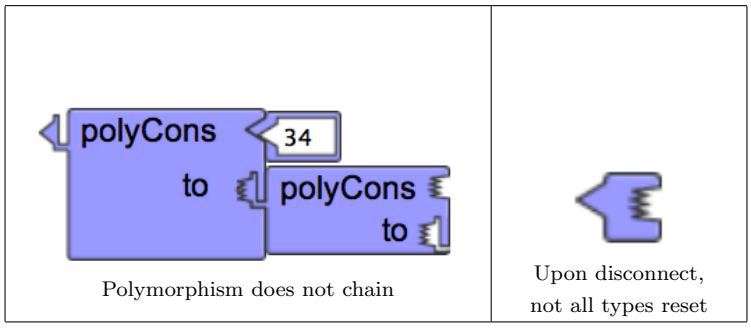


Figure 4-7: Cases where polymorphism does not work properly in the current implementation

Chapter 5

Conclusion and Future Work

5.1 Summary

The design of many current blocks languages does not deliver on the expectation that shapes are used to indicate type, or what fragments are valid to connect. It would be logical to assume that two blocks that looked like they fit together would logically flow together, but this is not necessarily the case. Current blocks languages also only are able to express a limited number of types. The goal of this project is to visualize types for a blocks version of a language similar to SML, a statically typed programming language with expressive types. I developed TypeBlocks, with shapes for `list`, `tuple`, and `function` types for blocks implemented in the blocks framework ScriptBlocks. Here each arbitrarily complex type is translated into a unique connector shape. TypeBlocks also attempts to handle ML- style universal polymorphism.

There are many directions upon which my work can be built upon. These type shapes can be integrated into an SML-inspired blocks language (Section 5.2). My work could also be used as inspiration to look at blocks representations of other interesting type systems (Section 5.3). It has yet to be seen if these shapes are actually understandable and helpful, and user testing could be done before making a system using my types to see if there are any necessary improvements to my types or the ScriptBlocks framework to make such a system more intuitive (Section 5.4). Visualizing types for a blocks language does not necessarily have to be in socket and plugs, so work could be done exploring other paradigms for expressing types in blocks language, such as expressing type as color as in Waterbear (Section 5.5).

5.2 Blocks ML

Expressive types are an important step towards building an SML-inspired statically typed functional blocks language. However, in order to have a full-fledged blocks language based on SML, work still has to be done on concepts such as polymorphism, pattern matching and algebraic datatypes.

TypeBlocks does not currently have a fully working implementation of universal polymorphism. Currently blocks do not change types when a block is unplugged, which could be fixed by essentially reversing the work done when they plug together. More pressingly, polymorphism does not chain; this could be fixed by fixing the logical errors in the code that I initially wrote to fix this. Similarly, blocks only "look right" when a base type is plugged into a polymorphic type, and the block does not register that a block with a more complex type plugged in. This has worked in the past, so it is a matter of finely checking for the assumption that is not being made. If these changes were made, basic concepts like polymorphic `cons` could be appropriately represented in TypeBlocks.

The current visual representation of polymorphism is also flawed. Instead of looking like it could be any type, the polymorphic shape looks like its own special type, so it is not obvious that any shape can plug into it. Perhaps, the polymorphic connector could have an animated or jello-like quality which would visually indicate to the user that the type is not fully determined, an idea from Amon Miller and Erin Davis. Also, there needs to be a visual difference between different polymorphic types (like `'a` and `'b` in sml) so polymorphic higher order list function such as `zip` and `map` could be clearly represented.

There is currently no blocks paradigm for a concept like pattern matching. What would such a block look like? An sml-like blocks language would have to represent functions like the following:

```
fun isSorted [] = true
  | isSorted [x] = true
  | isSorted (x::(xs as (y:: zs)) = (x <= y) andalso (isSorted ys)
```

How would one be able to clearly represent the last line?

In sml, algebraic datatypes are used to represent arbitrary user-defined sum-of-product datatypes, but TypeBlocks currently only defines a simple pair type. What would a type like `tree` look like where type `tree` is defined as `type tree = Empty | Leaf of 'a | Node of tree * tree`?

5.3 Representing other Type Features in Blocks Languages

Blocks languages are currently not able to represent the features for many interesting type systems. I worked on representing ML-inspired types, but what about other languages? OCaml is a functional language with sml-like types and object types. What would those object types look like in blocks? Would they look like arbitrarily sized tuples, or would they be have a completely different feel? Haskell is a purely functional language with interesting types such as monads and type classes. How would those be represented in a blocks context? Java is an object-oriented language and uses object types. What would these object types look like in blocks? Would they look like tuples or something different? Then what would ad hoc polymorphism or overloading look like? Is it best to represent ad hoc polymorphism like StarLogo TNG, or is there a better mechanism? How would concepts like abstract classes be represented?

5.4 Usability

Blocks using my type shapes have not yet been made into a language and have not been tested by any users other than me and my advisor. It has yet to be seen if these shapes are actually understandable and helpful to some subpopulation of programmers. Particularly, it would be interesting to see if these blocks using these shapes help intermediate computer science students understand types better. Also, a more compact representation of these types needs to be explored, since these plugs grow quickly to unwieldy sizes, as seen in Figure 3-8; although, the current lack of compactness grew out of concern that the types be unambiguously readable, so types will need to have enough space to be readable, but not too much.

Currently, to build any language off of ScriptBlocks there needs to be some improvement in the general usability. However, what needs to be done is unclear. Would users understand better what blocks plug into which sockets if, when a block is initially dragged onto the screen, all of the sockets that it fit into are highlighted? It is also unclear to me if the trash can is adequately placed. I find myself dragging blocks off the screen onto the pallete rather than into the trash can based on its awkward placement, so it would be interesting to see if there were a more intuitive manner to get rid of blocks.

5.5 Type as Color

As seen in Chapter 2, plugs and sockets are not the only paradigm for indicating which blocks fit together in blocks languages. Particularly, in Section 2.8, we saw an example of a blocks framework and sample language for Waterbear which represented type as color and an expression fit into an argument by nesting, where when an expression fit, it would make a colored box. But, what would ML-inspired types look like in this language?

I began to think about types here and applied the same concept of representing recursively defined types by composition of base and constructor colors (Figure 5-1). However, instead of representing the constructors by solely a line of color, making these types hard to read, I thought of representing constructors by a uniquely colored pattern, where `list` is represented by what looks like railroad tracks, `function` is represented by repeated arrows, and `tuple` is represented by a repeated asterisk. However, in order for these types to work, there has to be some form of visual parenthesization, a concept which I have not formulated a general representation. Another issue is that type colors in Waterbear are defined in a css file, so in order to implement these types, it will be necessary to figure out another mechanism. It might be helpful to devise a strategy like ScriptBlocks which uses a combination of plain css and css implemented by JavaScript.

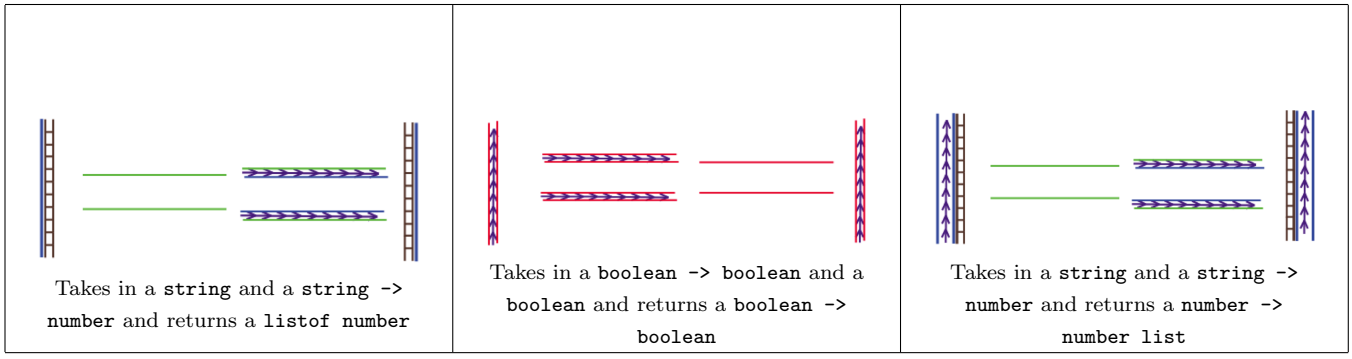


Figure 5-1: ML-inspired types a la Waterbear

Bibliography

[1] Types, Benjamin C. Pierce, In Programming Languages Mentoring Workshop, http://www.cis.upenn.edu/~sweirich/plmw12/Slides/plmw12-Pierce.pdf, accessed April 6, 2012.

[2] Scratch project, MIT Lifelong Kindergarten Group, http://scratch.mit.edu/, accessed Mar. 11, 2012.

[3] BYOB 3.1, Berkeley, http://byob.berkeley.edu/, accessed April 1, 2012.

[4] Panther Programming, http://pantherprogramming.weebly.com/, accessed April 19, 2012.

[5] Chirp, Jens Moenig, http://chirp.scratchr.org/, accessed April 19, 2012.

[6] DesignBlocks, MIT Lifelong Kindergarten Group, http://www.designblocks.net/, accessed April 19, 2012.

[7] ModKit home page, http://www.modk.it, accessed Mar. 22, 2012.

[8] App Inventor home page, MIT Center for Mobile Learning, http://appinventor.mit.edu, accessed April 19, 2012.

[9] OpenBlocks home page, MIT Scheller Teacher Education Program, http://education.mit.edu/openblocks, accessed April 23, 2012.

[10] The Playful Invention Company, PicoCricket Reference Guide, version 1.2a, http://www.picocricket.com/pdfs/Reference_Guide_V1_2a.pdf, accessed April 18, 2012.

[11] StarLogo TNG project, MIT Scheller Teacher Education Program, http://education.mit.edu/projects/starlogo-tng, accessed April 16, 2012.

[12] Waterbear homepage, Dethe Elza, http://waterbearlang.com, accessed April 1, 2012.

[13] Waterbear Arduino, Stretchyboy, http://stretch.deedah.org/waterbear/, accessed April 23, 2012.

[14] WPI Robotics Library, Techwiz Web Design Technologies, http://www.techwizworld.net/waterbear/, accessed April 24, 2012.

[15] EasyJ 4 FRC, Team2648, http://easyj.team2648.com/, accessed April 23, 2012.

[16] ScripTastic, Greenbush Labs, http://scriptastic.greenbush.us/, accessed April 23, 2012.

[17] Closure Library Documentation, Google, http://closure-library.googlecode.com/svn/docs/index.html, accessed April 24, 2012.

[18] ScriptBlocks Library, MIT Media Lab, http://code.google.com/p/scriptblocks/, accessed April 24, 2012.

[19] Andrew Begel. Logoblocks: A graphical programming language for interacting with the world, March 1996. MIT Advanced Undergraduate Project report. Available at http://research.microsoft.com/en-us/um/people/abegel/mit/begel-aup.pdf, accessed Mar. 10, 2012.

[20] Michael Bolin. *Closure: The Definitive Guide*. O'Reilly Media, Inc., 2010.

[21] Michael Gogins. Using the signal flow graph opcodes. *Csound*, 13, 2010. Available at http://www.csounds.com/journal/issue13/signalFlowGraphOpcodes.html, accessed April 23, 2012.

[22] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education*, 10(4), November 2010.

[23] Ricarose Roque. Openblocks: An extendable framework for graphical block programming systems. Master's thesis, MIT, May 2007. Available at http://dspace.mit.edu/bitstream/handle/1721.1/41550/220927290.pdf, accessed Mar. 13, 2012.

[24] Franklyn Turbak, David K. Gifford, and Mark Sheldon. *Design Concepts in Programming Languges*. MIT Press, 2008.