Atypical: A type system for live performances

Gabriel Barbosa Nunes

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
May 2017

Professor Kenneth Perlin

Professor Daniele Panozzo

Acknowledgements

I would like to express my deepest thanks to Professor Ken Perlin, who so graciously allowed me to become one of the core researchers on one of his most important research projects, and has helped guide me through every step of the process, as well as to Professor Daniele Panozzo, my second reader. I would also like to express my appreciation for the the students in the FRL and Chalktalk communities for their very useful feedback and support during the entirety of this research project. Lastly, I would like to thank my parents, Fernando Nunes and Rosana Barbosa, whose lifelong support was crucial to getting me to where I am today.

Abstract

Chalktalk is a visual language based around real-time interaction with virtual objects in a blackboard-style environment. Its aim is to be a presentation and communication tool, using animation and interactivity to allow easy illustration of complex topics or ideas. Among many of the capabilities of these virtual objects is the ability to send data from one object to another via a visual linking system. In this paper, we describe a way of making the link system more robust by adding type information to these links, and compare and contrast the requirements of a presentation-oriented visual language with a more traditional programming language.

Contents

1 Background **1**
- 1.1 Introduction to Chalktalk . . . 1
- 1.2 The Chalktalk interface . . . 3
- 1.3 Creating new sketches . . . 4
- 1.4 Linking sketches . . . 5
- 1.5 Limitations of the existing link system . . . 7

2 Introduction to Atypical **9**
- 2.1 Solving the linking problem . . . 9
- 2.2 Constraints and criteria for Atypical's design . . . 9
- 2.3 Evaluation of existing type systems . . . 11
  - 2.3.1 Static vs. dynamic typing . . . 11
  - 2.3.2 Inheritance, polymorphism, and subtyping . . . 12
  - 2.3.3 Generic types . . . 12
  - 2.3.4 Existing graphical languages . . . 13
- 2.4 Atypical's design . . . 14

3 Atypical's type API **16**
- 3.1 Defining types . . . 16
- 3.2 Generic types . . . 17
- 3.3 Defining explicit conversions . . . 19
- 3.4 Special conversion functions for generic types . . . 19
- 3.5 Intermediary conversions . . . 22
- 3.6 Primitive types . . . 23

4 Integration with Chalktalk **24**
- 4.1 Defining inputs and outputs . . . 24
- 4.2 Backwards compatibility . . . 25
- 4.3 Integration with Chalktalk's user interface . . . 26

5 Evaluation of Atypical **29**
5.1 Comparison against the constraints and criteria 29
5.2 Additional benefits 30
5.3 Limitations of the design 32

6 Future work **33**

7 Conclusion **35**

Background

1.1 Introduction to Chalktalk

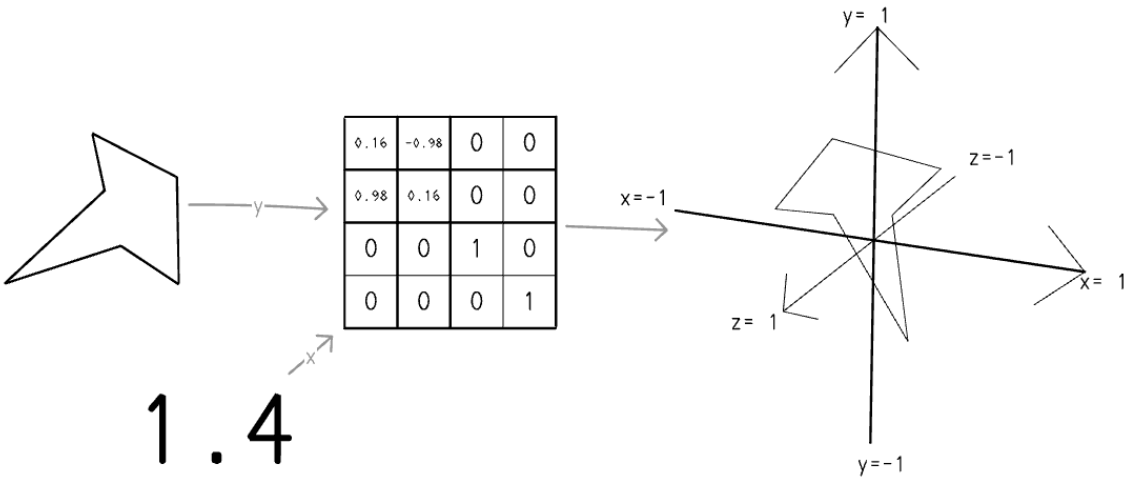

Figure 1.1: A polygonal shape (left) is transformed by a rotation matrix (middle) that is configured to rotate the polygon 1.4 radians around the Z axis, displaying the results on a set of axes (right).

Blackboards or whiteboards are a common fixture in most classrooms around the world. Their purpose is to allow teachers to support their verbal explanations with visual aids and graphical depictions of the subject at hand. This is an effective technique that has been proven to facilitate and enhance student learning [Pashler et al. 2007].

Classrooms have also more recently been implementing modern technological tools, such as computer-connected projectors or electronic smart boards. However, the ability of these tools to display dynamic content has traditionally been underutilized. They are often used

solely to display pre-made content such as slides, images, or video, and only rarely do they allow teachers to adjust their depiction of the content in the middle of a lesson the way a blackboard does. At best, they replicate a blackboard's functionality by allowing the teacher to draw and erase static images on a digital canvas.

Chalktalk [Perlin 2016] takes this blackboard metaphor into the future. At its core, it is a computer-based visual language based around real-time interaction with dynamic animated objects in a blackboard-style environment. Users create these objects by drawing a shape that corresponds to one of the objects in Chalktalk's large library of "sketches". When this drawing is clicked, it is then interpreted as its corresponding sketch and "comes to life" by replacing the freehand drawing with its animated sketch object. These sketches can, among other things:

- Draw text, lines, or shapes, in 2D or 3D,
- Move and animate,
- Respond to input events such as mouse clicks or drags,
- Generate or play audio, and
- Send data from one sketch to another.

The combination of these features allow for the illustration and visualization of complex topics or ideas, such as those displayed in Figure 1.1. Chalktalk is currently in active development and use by Professor Ken Perlin of New York University, as well as several other students and researchers at New York University's Future Reality Lab.

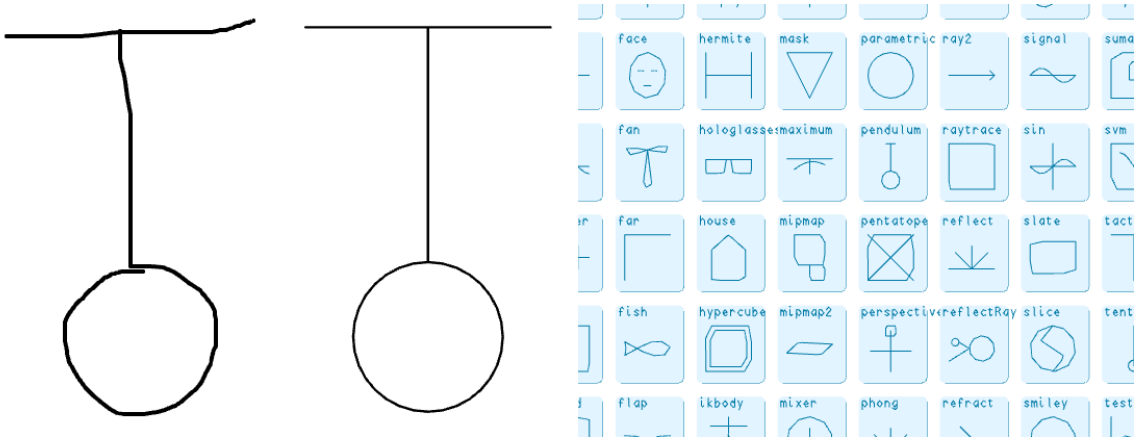

Figure 1.2: Left: a pendulum before and after interpretation. Right: a small segment of Chalktalk's library of sketches, with the pendulum shown near the center.

Beyond just teaching, Chalktalk has wide potential as a general communication tool. As computers become smaller and more integrated into our daily lives, and VR and AR technology becomes more practical for consumer use, computer-aided visualizations may very well end up being a part of our everyday communication. In this kind of enviroment, Chalktalk would not just be a teaching or presentation tool, but would be an aid to anyone wishing to communicate complex concepts to another person.

1.2 The Chalktalk interface

Given its use as a presentation tool, the Chalktalk interface is deliberately minimal to allow the viewers of the presentation to focus on the content of the presentation rather than the interface. When first running the program, the user is presented with an empty canvas. Clicking and dragging the mouse begins a drawing, and by clicking on a drawing, it is interpreted and becomes a live object.

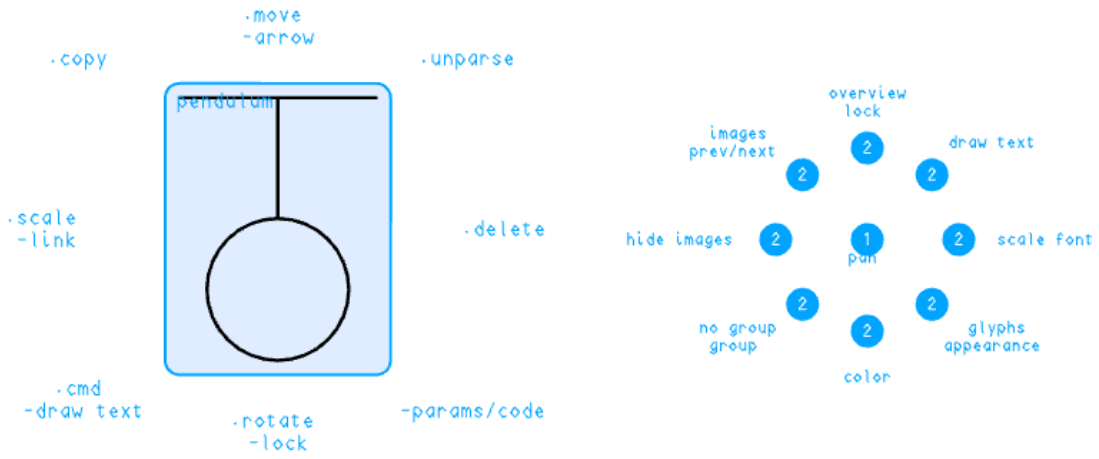

Figure 1.3: Left: the sketch radial menu, shown in "help mode" when the cursor hovers over a sketch. Right: the radial menu shown in "help mode" when the cursor hovers over the background.

Other actions are done through an invisible radial menu. For example, clicking anywhere on the background, then clicking below the first click, reveals a color palette, allowing the user to change the color of the lines being drawn, while clicking the background and then clicking above the first click zooms the view out to show an overview of the blackboard. Sketches can also be manipulated with a similar radial menu, where the clicks go in the opposite direction. For example, clicking to the right of a sketch then on the sketch itself

deletes a sketch, clicking below a sketch then on the sketch rotates it, and by clicking to the left of a sketch followed by dragging from one sketch to another, a link is created between the two sketches so that data can travel from the first sketch to the second.

However, this invisible interface can be difficult to learn for novice users. As such, Chalktalk also contains an auxiliary interface, termed "help mode", activated by pressing the space bar. In help mode, the radial menus are made visible, and auxiliary information about sketches, hotkeys, and other features of the system are shown to the user directly. Due to its more cluttered appearance, help mode is not suitable for a live performance, but it is useful when learning to navigate Chalktalk's interface.

1.3 Creating new sketches

Users also have the ability to create their own sketches. Because Chalktalk is implemented in Javascript and runs in any modern web browser, sketches are implemented by creating a Javascript file. This file contains a single function which initializes the sketch object, and this function should define, at minimum, a `label` string with the name of the sketch, and a `render` function that describes both how the sketch should be displayed and how it is drawn by the user.

While Chalktalk is implemented on top of WebGL, users do not have to deal with low-level WebGL API calls themselves. Instead, a simplified high-level API is provided for sketch authors, similar to artistic programming tools such as Processing [Processing Foundation 2017] or p5.js [McCarthy et al. 2017]. A similar high-level API is provided for user interactions and interaction with other sketches. While the full range of this API is out of scope for the purposes of this paper, we will focus on one detail: the input/output system for sending data between sketches over a link.

Listing 1.1: The code to define a minimal "house" sketch, which is a simple pentagon shape with no animation or user interaction.

```
function() {
    this.label = 'house';
    this.render = function() {
        mCurve([[-1,.8],[-1,-1],[1,-1],[1,.8]]);
        mCurve([[1,.8],[0,1.7],[-1,.8]]);
    }
}
```

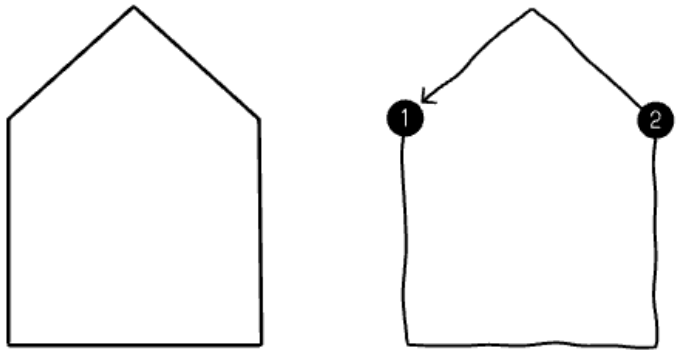

Figure 1.4: The house sketch defined by Listing 1.1. Left shows the sketch after interpretation, while right shows the steps required to draw it.

1.4 Linking sketches

As previously mentioned, sketches can be linked in order to send data from one sketch to another. Each sketch has exactly one output that can be sent to any number of sketches, and each sketch can receive any number of inputs from other sketches. This data is updated and propagated from each sending sketch to each receiving sketch at every frame.

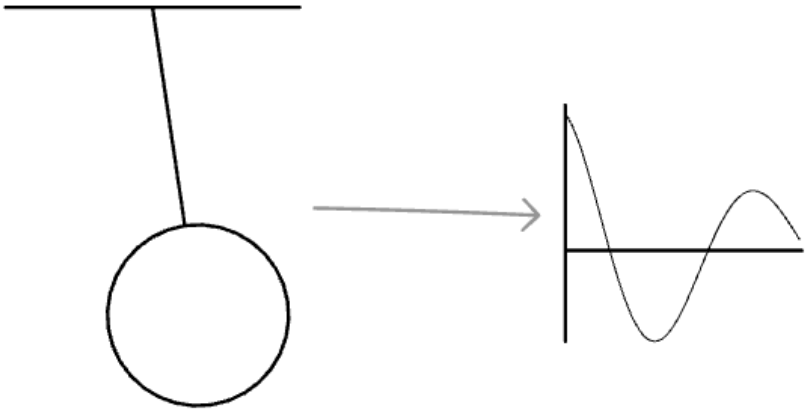

Figure 1.5: A swinging pendulum sketch (left) is linked to a graph sketch (right) that plots the pendulum's angle over time.

These links are represented in Chalktalk's visual interface as arrows pointing from the sending sketch to the receiving sketch. The order in which inputs are connected matter: the first link created is considered the sketch's first input, the second link is the second input, and so on. For sketches with multiple inputs, the first, second, and third input link are each annotated visually with an "x", "y", and "z" respectively in order to differentiate the arrows.

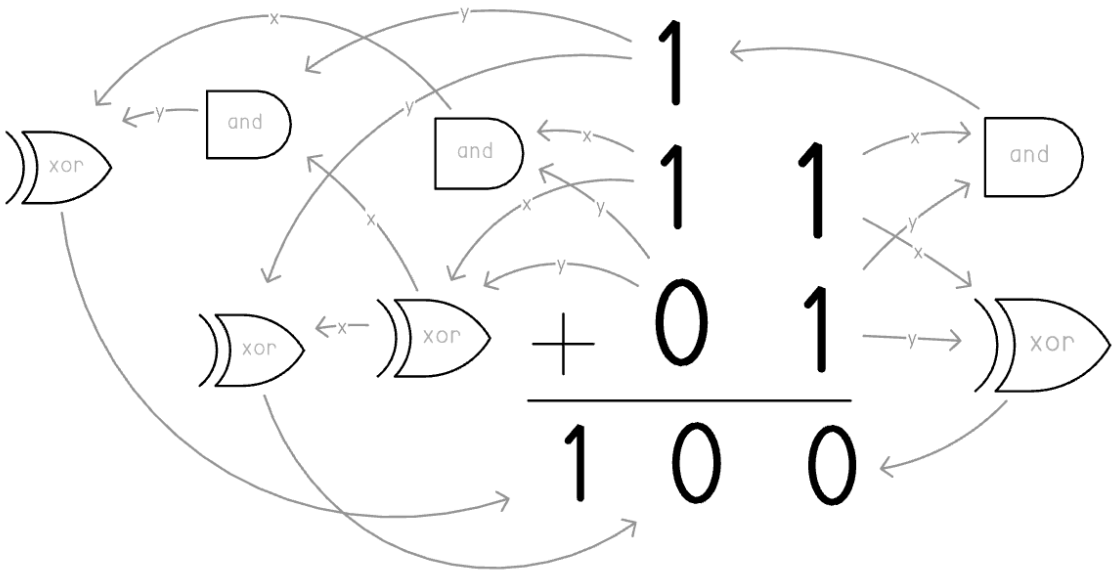

Figure 1.6: Logic gate and numerical sketches are used to implement a two-bit binary adder in Chalktalk. The addition symbol and line here are purely decorative: all addition is being performed through the linked logic gates.

In the existing link system, the data that is sent can be any Javascript data, from simple primitive values such as numbers, strings, and arrays to more complex values like objects and closures. Sketches use this system to send a variety of data, including but not limited to:

- Numerical values or functions to be plotted on a graph (as seen in Figure 1.5) or used as controls for another object (as seen in Figure 1.1),
- Boolean true and false values, that can be used to demonstrate concepts in digital logic,
- Vectors and matrices, represented by arrays of numbers, that are used to perform linear algebra operations,

- Note frequencies to be played through an audio synthesizer,
- Functions that take in a time value and return an audio sample to be played through the computer's speakers,
- A mesh object containing a list of 3D vertices and edges that can be displayed in a set of 3D axes (as seen in Figure 1.1),
- A spline curve, used to control the shape or animation of the receiving sketch,
- Text strings, to be displayed, manipulated, or interpreted by the receiving sketch, and
- Color data that can control the display color of a 2D or 3D object.

This variety of data creates a large space for experimentation, and this helps to take Chalktalk from a simple animation tool to an exploratory medium. Users can ask questions such as, "What happens if I take data from *this* sketch and put it into *that* sketch?" and by performing these links find out the answer, or demonstrate it for an audience. As a result, this linking system is crucial to the usefulness of Chalktalk as a whole.

To allow sketches to respond to links in the existing link system, two things must be done:

1. For a sketch to emit output values, it must define an `output` function. The return value of this function will be the output value of this sketch.
2. For a sketch to respond to input values, it must look at its `inValues` array, where data coming in from other sketches is stored, in the order in which they were connected.

Listing 1.2: The output function for the pendulum object, which outputs its angle in radians.

```
this.output = function() {
    return this.angle;
}
```

1.5 Limitations of the existing link system

As mentioned above, Chalktalk supports a wide variety of data being sent over the link connections. However, there are several flaws to this system:

1. Sketches have no guarantee over what kind of data will be arriving at their inputs. For example, a sketch that outputs a full 3D mesh object can be connected to a sketch that expects a simple number as the input, leading to runtime errors if the receiving sketch attempts to treat the mesh as a number.

2. While sketches can inspect the runtime type of the input data, these runtime types are very broad categories that do not inform the sketch of how its value is to be interpreted. For example, many sketches accept a numerical input as a floating-point number, but Chalktalk has no way of knowing whether this numerical input is intended to be a time value in seconds, an audio sample, a frequency in hertz, an angle in radians, or any other form of numerical input. This can often lead to confusing situations and unusual behavior, such as when a sketch outputting audio samples (whose range is -1 to 1) is connected to a synthesizer sketch expecting a frequency in hertz (in the range of hundreds to thousands), leading to unusual audio effects due to the misinterpretation of the input data.

3. Because all input values are simply packed into a dynamically-sized array, Chalktalk has no knowledge of how many inputs each sketch uses and cannot display this information to the user.

4. Due to all of the above issues, users have no feedback as to which sketches can be connected together. Every sketch is allowed to connect to every other sketch regardless of whether that connection will do anything, and regardless of whether the connection will cause the sketches to act sensibly. Not even help mode displays this information to the user.

The end result of these problems is that the presenter must *know* what sketches are designed to work well together. This is feasible when the presenter was also the author of the sketches they're using, but the end goal of Chalktalk is to provide users with a suitable "standard vocabulary" of sketches that is suitable for general use in many places, while allowing them to augment the library with their own sketches and with sketches shared by other users. As such, the majority of users are *not* going to be creating the majority of the sketches they use, and as such better feedback and more sensible default behavior is required.

Introduction to Atypical

2.1 Solving the linking problem

Many programming languages have long ago solved the problem of determining what kind of data is allowed to be passed from one object, module, or function to another: they use static type systems, which explicitly declare what kind of data is allowed to flow between them. A type system is an extremely attractive solution to the existing problems with the link system, as it would allow Chalktalk to know what kind of data each sketch is emitting and expecting, show adequate feedback when compatible and incompatible sketches are linked, and support the user in their explorations.

However, most programming languages are designed for the creation of large, long-lasting software systems. As such, type systems in these programming languages focus primarily on preventing errors at execution time, on making certain erroneous program states impossible, and on presenting error messages to the programmer as early as possible, often at compilation time [Cardelli 1996].

Chalktalk is not meant for designing computational artifacts and long-lived software systems, but rather for exploration, presentation, and communication in a real-time setting. Because the focus of the language is different, so too must the focus of the type system be different. It is for this reason why its type system will be called **Atypical**.

2.2 Constraints and criteria for Atypical's design

We will use the following criteria to evaluate potential type system designs:

1. Similar to most programming language type systems, Atypical should allow Chalktalk to discern whether two sketches have compatible input and output types, thus determining whether they can be connected.

2. Related to this, Atypical should allow Chalktalk to inspect these types at runtime so it can display visual feedback to the user showing whether or not two sketches are compatible when the user attempts to connect them.

3. At the same time, Atypical should allow the user to connect any sketches that have data that are "close enough". For example, if the pendulum sketch outputs its angle as a Radians type, it should still be allowed to connect to sketches that expect a floating-point number as input. This should allow Atypical to retain the looser "feel" of a dynamic type system, while still assuring sketch creators that their sketches will only receive the types of data that they specify.

4. Because Chalktalk was designed for use in real-time presentations, often in front of a large audience, both Atypical and its integration with Chalktalk should support this form of interaction by avoiding errors that can spoil the show. Although certain classes of errors caused by misinterpretation of input data can cause problems during a presentation, intrusive error messages or crashes during a presentation are far worse.

5. As mentioned in section 1.2, Chalktalk errs on the side of keeping its interface hidden, so as not to overwhelm those watching a presentation with unnecessary interface details. However, Atypical should allow Chalktalk to display the types for novice users in help mode, allowing them to discover what kind of connections are and are not possible.

6. The type system should also avoid placing an undue burden on sketch authors. As with Processing and p5.js, we make the assumption that sketch authors may not be professional programmers, but instead artists, educators, students, scientists, or anyone else who wishes to communicate an idea to people. We should not require them to understand the details of the structure and implementation of the type system itself in order to use it.

In addition, we also have the following practical constraints when implementing this type system:

1. Because Chalktalk is developed in browser-based Javascript, Atypical must also be developed either in Javascript or in a compatible language, that can either compile to or at very least interoperate with Javascript.

2. Chalktalk's current sketch library contains over 100 sketches from a variety of different domains. Because of the effort it would take to migrate all these sketches to a new input and output system, Atypical and its integration with Chalktalk should allow for at least some level of backwards compatibility with Chalktalk's older sketches that do not declare types.

3. However, Atypical should allow, with sufficient effort, for all the current sketches in Chalktalk to be migrated to use Atypical. In other words, there should exist no sketch

in the Chalktalk sketch library that is impossible to implement using the Atypical type system. Some slight variances in the behavioral semantics of some sketches is acceptable, but overall, sketches should survive the transition effectively intact.

2.3 Evaluation of existing type systems

2.3.1 Static vs. dynamic typing

Because Chalktalk and all its sketches are implemented in Javascript, we can effectively say that the existing link system uses the Javascript type system. Javascript has a dynamic type system, which means that types of values are not checked at compile time, but only at run-time. In addition, when performing an operation on an incorrect type, Javascript favors casting of values instead of returning an error message. For example, performing binary addition on two non-numeric, non-string values converts them both to strings and concatenates those strings [ECMA 2016].

However, most dynamic type systems do not allow for type annotations or restrictions. Variables can take on any value of any type and it is up to the programmer to verify that they have received a value of the correct type before continuing execution. Given that this is the exact problem we described in section 1.5, we conclude that a purely dynamic type system is insufficient for our purposes.

Standing in contrast to dynamic type systems are static type systems, where types are explicitly annotated or inferred and are typically known at compile time [Cardelli 1996]. However, a crucial distinction must be made between the static type systems in typical programming languages and the Atypical system. In a typical programming language, once the source code is distributed to users, it remains effectively static and the data flow through the program does not change. In Chalktalk, the user is actively modifying the data flow during a performance, and as such a purely static, compile-time type system is insufficient for our purposes.

What we find instead is that we must take a hybrid approach. Sketches should be able to declare which types they support, but this must be done at run-time. However, the absolute latest point that the sketch's input and output types can be defined and set is when the sketch is added to the blackboard. Once it is added to the board, this type must remain static and can no longer change, otherwise any links that the user drew may no longer be valid.

2.3.2 Inheritance, polymorphism, and subtyping

Most object-oriented languages implement some form of subtyping via inheritance. In these languages, subtyping is valuable because they allow us to extend a large, existing software system while allowing it to treat our extended derived types and data structures as if they were the parent type, which is a form of polymorphism [Cardelli 1991].

Javascript also supports a somewhat different form of inheritance called prototypical inheritance, inspired by the language Self [Ungar and Smith 1987]. When accessing a Javascript object's properties, if the object does not contain the desired property, then the request is forwarded to another object called the object's prototype, followed by its prototype's prototype, and so on.

However, because Chalktalk is not designed for the creation of large, long-lived software systems, the drawbacks of inheritance begin to outweigh the benefits. In particular, inheritance and subtyping, particularly without multiple inheritance, places types on a strict hierarchy where derived types can be treated as their base type but not vice-versa. In addition, there is usually no interoperability between sibling types without going up the hierarchy. For example, if `Radians` and `Degrees` are both subtypes of the `Float` type, usually an explicit conversion must be defined in order to allow systems that accept `Radians` to also accept `Degrees`, and vice-versa. For Chalktalk, we do not want types to be confined to a strict hierarchy, but instead to be interoperable with a "neighbourhood" of similar types, and as such inheritance-based type systems are not suitable for our purposes.

Other languages such as Haskell provide polymorphism via *type classes* [Hall et al. 1996]. Roughly speaking, if a type implements a set of operations related to a particular type class, then that type is seen as a member of that type class, and is interoperable with all functions that operate on that type class. However, implementing a system like this greatly increases the complexity of the type system, and would make it unsuitable for non-professional programmers and sketch authors.

2.3.3 Generic types

Generic types, sometimes called parametrized types or second-order types [Cardelli 1996], are, in a manner of speaking, functions that take types as arguments and output new types in return. For example, a generic type called `Array` might take in a type such as `Float`, and return the constructed type `Array(Float)`, an array of floating-point numbers. Meanwhile, a generic type called `Pair` may take in two types, such as `Float` and `String`, and return `Pair(Float, String)`, a product type consisting of a single floating-point number and a single string.

Generic types are absolutely essential for Atypical. Chalktalk currently has many sketches where values such as arrays and functions are used as both inputs and outputs. Without generic types, you would not be able to define what these arrays contain, or what types these functions expect as arguments and return as outputs. These sketches would either lose that type information, or specific types for each kind of array and function would have to be defined (e.g. `FloatArray`, `StringArray`, etc.).

Most statically-typed languages have support for generic types for precisely this reason. While dynamically-typed languages can mix and match values of different types in an object such as an array, statically-typed languages cannot do this without losing type assurances. Even modern statically-typed languages that do not implement generic types, such as Go, still have a few special types that allow for this type of functionality, such as their array types [Go Team 2016].

2.3.4 Existing graphical languages

Aside from textual programming languages, graphical languages have also been well studied and are presently in active commercial use. One of the earliest experiments in this area was by Robert Sutherland [1966], who described a graphical system for creating software programs using line drawings and displays. His work only briefly mentions the possibility of type systems and does not go into detail as to the implementation details.

In the commercial space, Max/MSP is a popular graphical programming language focused on music production. Max/MSP also uses a data flow system as the foundation of its graphical language, but the built-in types offered are few and have little opportunity for extension. In addition, types are often domain-specific and thus lack the kind of interoperability that Chalktalk needs [Freed et al. 2011].

Xiong, et al. [2000] describe a very well-crafted static type system for Ptolemy II, a graphical language meant for use in embedded systems. This type system puts types on a lattice that uses type inequalities to determine where data conversions can be done losslessly. However, this approach carries with it two flaws when looked at through the lens of Chalktalk. First, it replicates the same problem we have with inheritance and subtyping: types are placed on a hierarchy which inhibits type conversion in any direction but up. Second, like many other programming languages, but *unlike* Chalktalk, Ptolemy II is meant for the creation of long-running, safety-critical software, where catching and presenting the programmer with an error, either at compile time or runtime, is preferable to allowing the program to enter an undesirable state. Thus, their focus is primarily on ensuring that all data conversions are lossless, whereas Chalktalk has some tolerance for results that are only *approximately* correct.

2.4 Atypical's design

Having identified our constraints and criteria for evaluating the design of the language, as well as having reviewed many aspects of existing programming language type systems, we will now relate the specific design decisions that were made when designing Atypical.

Types will be coded directly in Javascript, with values of a particular type implemented as Javascript objects.

While it may have been feasible to define a separate specification language for the types themselves, ultimately values of a particular type have to be implemented as values in Javascript in order to run in the browser. In addition, type conversion functions and manipulations of the type objects within the sketches must be done in Javascript, so implementing the type itself in Javascript avoids requiring sketch authors to learn two representations of the type, in our specification language and in Javascript.

All types will derive from a single prototype class, `Type`. No other form of inheritance or subtyping will be used.

The only purpose of `Type` is to provide a few convenience methods to all objects that are defined within the Atypical system. But due to the problems with inheritance explained in section 2.3.2, as well as the fact that Chalktalk's data transmission is comprised of short-lived data messages instead of long-lived software objects, Atypical does not use inheritance for the purposes of ensuring interoperability between different sketches with similar input types.

Conversion functions between types will be the primary method of ensuring interoperability. These conversion functions should be as simple as possible to implement.

Instead of inheritance, interoperability is achieved by allowing the author of a type to define explicit conversion functions between their type to other types. Under a naive approach, this would require type authors to define $O(n^2)$ conversion functions in order to make n types interoperable. Atypical does not take this naive approach: type authors have the ability to define whole swaths of conversion functions at once, under many different configurations, depending on the kind of type and the explicit conversion functions the type author chooses to write.

Among the benefits of this is the fact that type interoperability is not restricted to any one rigid topology such as a strict hierarchy or lattice. If a conversion between two types is required, it can simply be defined regardless of what other conversions exist.

Types are to be created very early at runtime, ideally when first loading the Chalktalk web page.

Types are defined in a typical Javascript file by calling API functions that set up the internal state of the type system in memory. Because these types must be set up before any other part of Chalktalk uses them, these files should be loaded early in the page load process before any user interaction begins. This effectively turns Atypical into a mainly-static type system when seen from the perspective of the sketches.

Generic types will be implemented as Javascript functions that take types as arguments and return types.

When the user defines their sketch's output as an `Array(Float)`, this actually calls a function called `Array` passing the `Float` type constructor as an argument. If the `Float` form of `Array` has already been defined at this point, it its constructor is immediately returned to the caller. Otherwise, the type is dynamically created and defined based on the specification of the generic `Array` type.

While this may seem like an on-the-fly type creation, violating the previous design decision, in practice, all sketches, including their associated types, are loaded and evaluated as part of Chalktalk's loading process. This means that any generic types used in sketches are also created long before the user actually interacts with the system.

Atypical's type API

We now discuss the implementation of the above design.[1]

3.1 Defining types

Each value in Atypical is implemented as a Javascript object whose properties contain the actual data for the object.

To give an example, a `Color` object is represented by a Javascript object containing `r`, `g`, `b` and `a` properties, each containing a primitive Javascript number. These properties represent the red, green, blue, and alpha channels of the color respectively.

Its prototype provides methods and values common to all `Color` objects, such as a method for calculating the color's luminance. Its prototype's prototype, in turn, is `Type` as described in section 2.4, which provides methods common to all typed objects.

In most types, the properties on each object are defined as immutable by using a special flag passed into `Object.defineProperty`. The `_set` method is provided as part of `Type` as a convenience for defining these immutable properties.

Types are defined by calling `defineType(implementation)`, where `implementation` is an object containing:

1. An `init` method for initialization, which is also used as the type's constructor function,

[1] In the code listings in this section, you may see "`AT.`" used as a prefix for most of the functions related to the type system. Atypical is implemented using the Javascript module pattern to provide encapsulation [Cherry 2010], and `AT` is the short name for the Atypical Javascript module. This pattern is somewhat comparable to namespaces in other programming languages, and for the purposes of this paper the "`AT.`" prefix can be ignored.

2. A `typename` string, which must be a valid Javascript identifier with no dollar sign characters,
3. Optionally, the special `toPrimitive` method, used to convert this type into a primitive Javascript object (described in more detail in section 3.6), and
4. Any other methods or constants that objects of this type should have (such as the aforementioned `luminance` method).

Listing 3.1: An abbreviated version of the `Color` object's type definition, with a `Color` value being created below. Extra code that allows this object to be initialized with an array or another `Color` object was omitted for brevity.

```
AT.defineType({
    typename: "Color",
    init: function(r, g, b, a) {
        // ...Extra convenience code omitted...
        // Define a helper function to help with validation
        function validate(channel, defaultValue) {
            if (channel === undefined) { channel = defaultValue; }
            channel = ensureNumeric(channel);
            return Math.min(1, Math.max(0, channel));
        }
        this._set("r", validate(r, 0));
        this._set("g", validate(g, 0));
        this._set("b", validate(b, 0));
        this._set("a", validate(a, 1));
    },
    luminance: function() {
        return 0.2126*this.r + 0.7152*this.g + 0.0722*this.b;
    }
    // ...A few other methods omitted...
});

let pink = new AT.Color(1, 0.64, 0.94);
console.log(pink.g); // Logs "0.64"
```

3.2 Generic types

Generic types can be defined in a similar manner to standard types, by calling the `defineGenericType(implementation)` method with a similar implementation object. As

described in section 2.4, `defineGenericType` creates a function that automatically defines and returns a type constructor (called the "constructed type") when given other type constructors as arguments.

Much like when defining standard types, the `implementation` object should contain:

1. An `init` method for initialization, which is also used as the constructor function for all constructed types of this generic type,
2. A `typename` string for the generic type, which must be a valid Javascript identifier with no dollar sign characters,
3. Optionally, the special `toPrimitive` method, described in section 3.6,
4. Optionally, the special `changeTypeParamters`, `canChangeTypeParameters`, `convertToTypeParameter`, `convertFromTypeParameter`, `canConvertToTypeParameter`, and `canConvertFromTypeParameter` methods, discussed in section 3.4, and
5. Any other methods or constants that objects of this generic type should have.

Listing 3.2: The full definition for the `Pair` type used in Chalktalk, as well as an instance of one its constructed types below.

```
AT.defineGenericType({
    typename: "Pair",
    init: function(first, second) {
        this._set("first", AT.wrapOrConvertValue(
            this.typeParameters[0], first));
        this._set("second", AT.wrapOrConvertValue(
            this.typeParameters[1], second));
    },
    changeTypeParameters: function(newTypes) {
        return new (this.genericType.apply(
            null, newTypes))(this.first, this.second);
    },
    canConvertToTypeParameter: function(typeIndex) {
        return typeIndex === 0 || typeIndex === 1;
    },
    convertToTypeParameter: function(typeIndex) {
        if (typeIndex === 0) {
            return this.first;
        }
        else {
            return this.second;
        }
```

```
    }
});

// Unfortunately, Javascript syntax limitations require the
// extra set of parentheses around AT.Pair(AT.Float, AT.String).
let floatStringPair
    = new (AT.Pair(AT.Float, AT.String))(5.0, "five");

console.log(floatStringPair.first.value); // Logs "5"
```

3.3 Defining explicit conversions

Conversion functions can be defined between any two types (including constructed types) by using the `defineConversion` function, and providing it the two types to be converted, along with a function that takes in an object of the first type and returns an object of the second.

Listing 3.3: A sample conversion between the `Hertz` type that represents frequency and the standard `String` type.

```
AT.defineConversion(AT.Hertz, AT.String, function(hz) {
    return new AT.String(hz.value.toFixed(0) + " Hz");
});
```

Importantly, defining any conversion allows sketches that output the first type to connect to sketches that take in the second type. Chalktalk automatically performs the conversion when propagating the output values to each sketch.

3.4 Special conversion functions for generic types

Although explicit conversions, if defined, always take priority, generic types have the opportunity to define a large series of conversion functions at once. These are created by defining one or more of the following methods on the implementation object during its definition:

`convertToTypeParameter`, `convertFromTypeParameter`

These are optional functions that allow types constructed from this generic type to be automatically made convertible with their type parameters. For example, if this function is defined on our `Pair` example from section 3.2, then these could be defined to automatically allow `Pair(Float, Int)` to be convertible to and/or from `Float` and `Int`.

If defined, `convertToTypeParameter` must be a function of one argument taking in the index of the type parameter this object should be converted to, and returning the converted object. `convertFromTypeParameter`, on the other hand, must be defined as a function of two arguments taking in both the index of the type parameter this object should be converted from and the value it should be converted from.

When either of these two functions are defined, Atypical automatically generates conversion functions in the appropriate direction between the type parameters and the constructed type. This is done at the moment the constructed type is first created.

`canConvertToTypeParameter`, `canConvertFromTypeParameter`

These are optional functions that allow the type author to restrict which type parameters this generic type can be converted to or from. These functions are called by Atypical with a single argument containing the index of the type parameter, and returning `true` from these functions allows the conversion, while returning `false` does not. This is useful in several cases. For example, a generic `Function` type should be convertible from (but not to) the type parameter representing its return value, because a simple value is effectively equivalent to a function that always returns a constant value while ignoring its arguments. However, it would be nonsensical for a `Function` object to be convertible to or from the types of its arguments.

If `convertToTypeParameter` and/or `convertFromTypeParameter` is defined on a generic type and its corresponding `canConvert` function is not, Atypical assumes that this generic type can be converted to and/or from all of its type parameters and will automatically define a default implementation of the corresponding `canConvert` function to reflect this assumption.

`changeTypeParameters`

If defined, this function is used to create the conversion function between different constructed types that share the same generic type, such as, for example, `Array(Float)` and `Array(String)`. By defining this one function, the type author can define a large number of

conversions between all constructed types of a particular generic type, such as all `Function`s or all `Pair`s.

This must be defined as a function of a single argument that takes in an array of the new type parameters.

Listing 3.4: The `changeTypeParameters` function for the generic `Array` type. This function returns a new array with the values converted from their existing types to the new types.

```
changeTypeParameters: function(newTypeParams) {
    return new (this.genericType(newTypeParams[0]))(
        this.values.map(function(value) {
            return value.convert(newTypeParams[0]);
        }
    ));
}
```

`canChangeTypeParameters`

By default, constructed types can only be converted via `changeTypeParameters` when the number of type parameters in the initial type (the "source type") and the type being converted to (the "destination type") are the same, and when each type in the source type's type parameters is convertible to its corresponding type in the destination type's type parameters.

However, by defining `canChangeTypeParameters`, the type author can customize the conditions under which this kind of type conversion is allowed to take place. By returning `true`, the conversion is allowed to take place, and by returning `false`, it is prevented. Like `changeTypeParameters`, this function must be defined as a function of a single argument that takes in an array of the new type parameters.

For an example where this is useful, consider the generic `Function` type, which is implemented as a wrapper around an ordinary Javascript function. If we consider a `Function(A, B, C)`—that is, a function that takes in two arguments of type `A` and `B` and returns a value of type `C`—and we wish to convert it to a `Function(X, Y, Z)` object, we can do so by wrapping our original `Function(A, B, C)` inside a `Function(X, Y, Z)`, and convert both the argument types and the return type internally. For this kind of conversion to be possible, we need `C` to be convertible to `Z` so that we can convert the return value from the internal `Function(A, B, C)` to a `Z`. However, the argument types need to be converted in the opposite direction, from `X` to `A` and from `Y` to `B`. We can say that the conversion requirements for a generic `Function` type are covariant in the output type but contravariant in the input types.

In addition, functions that take in fewer arguments can easily be converted to functions that take in more arguments by simply ignoring the additional arguments. By customizing `canChangeTypeParameters` and comparing the type parameters currently used to the type parameters given, we can account for all of these conditions, making a large variety of different `Function` types interoperable and convertible by default.

3.5 Intermediary conversions

Certain types, such as `Radians`, can be seen as specializations of more general types such as `Float`. We can easily define a conversion from `Radians` to `Float` in order to make sketches that output `Radians`, such as the pendulum, interoperable with sketches that expect a `Float` value as input. However, this approach has a drawback. Sketches that output `Float` can be connected to a wide variety of sketches that take in types such as `Bool`, `Int`, `Function(Float)` and `Array(Float)`, because all these types are convertible from `Float`. However, although `Radians` is convertible to `Float`, it is not immediately convertible to any of these other types.

Defining explicit conversions for all of these types would be extremely time-consuming, and as such Atypical allows for a special kind of conversion to be defined: an intermediary conversion. In short, defining an intermediary conversion allows for conversion from a source type `S` to a destination type `D` by first converting `S` to an intermediary type `I`, and then converting the resulting value from `I` to `D`. For example, we can define a conversion from `Radians` to `Function(Float)` by first converting the `Radians` to a `Float` value, and then converting the `Float` to a `Function(Float)`.

There are three forms of intermediary conversions:

1. Explicit intermediary conversions, defined by calling `defineConversionsViaIntermediary(S, I, D)` with all three arguments defined. This creates a single intermediary conversion from `S` to `D` using `I` as an intermediary type, and is treated identically to a regular explicit conversion.
2. Broad intermediary conversions from any source, defined by calling `defineConversionsViaIntermediary(null, I, D)`. This allows *any* type that is explicitly convertible to `I` to be convertible to `D` by first converting to `I` and then converting to `D`.
3. Broad intermediary conversions from any destination, defined by calling `defineConversionsViaIntermediary(S, I, null)`. This allows `S` to be converted to *any* type that is explicitly convertible from `I`, by first converting `S` to `I` and then converting to the destination type.

Above, we mention that types must be "explicitly convertible" to allow broad conversions to take place. This means they must be convertible through either an explicit conversion or a special generic conversion. Atypical does not allow broad intermediary conversions to be "chained", to avoid creating spurious and surprising conversions between types that were not meant to be interoperable. Broad intermediary conversions are also a lower priority than both explicit conversions and generic type conversions, and will not be used if either of those types of conversion functions are defined for the types being converted.

3.6 Primitive types

While every value in Atypical is implemented as a Javascript object, for certain types such as `Float`, `Array`, or `String`, it can be burdensome to work with the values as wrapped objects. In a sketch, for example, sketch authors must remember to always unwrap the values after they receive them and wrap them again before sending them as output.

To alleviate this problem, any type can be defined as a "primitive type" by defining the `toPrimitive` method in its implementation. The return value of `toPrimitive` is assumed to be the primitive Javascript value that corresponds with the Atypical value, such as the Javascript floating-point number in an Atypical `Float` object. Primitive types are also expected to accept these primitive Javascript values in their `init` functions to allow them to be re-wrapped.

When using a primitive type as an input to a sketch, it is automatically unwrapped by default, and when using a primitive value as an output, it is automatically re-wrapped. This allows sketch authors to work with native Javascript values within their sketches in cases where a more complex object is not needed.

Integration with Chalktalk

4.1 Defining inputs and outputs

In addition to the Atypical type system, Chalktalk's system of sketch inputs and outputs was completely reworked to be more rigorous. Instead of simply defining an output function, sketches must now define an output function and a type that the function outputs. Sketches must also define the types of inputs that they intend to receive.

The functions for defining input and output types are as follows:

- `defineOutput(outputType, outputFunction)`: Defines the sketch's output type and the function that is called every frame to generate the output value.
- `defineInput(inputType, preprocessingFunction)`: Defines an input to the sketch. This function may be called multiple times to define multiple inputs.

  The preprocessing function is optional: if defined as a function of one argument, it will be called every time this input receives a value, with the value as the argument. When accessing the input value in the sketch, the return value of this function will be received instead. For primitive types, this is set to the type's `toPrimitive` method by default.
- `defineAllRemainingInputs(inputType, preprocessingFunction)`: This function allows sketches to accept a variable number of inputs of the given type. The number of inputs is dynamically adjusted to match the number of input links currently connected to the sketch. As with `defineInput`, the preprocessing function is optional.
- `defineAlternateInputType(inputType, preprocessingFunction)`: This function, called after an input has been defined, adds another supported type to the most recently defined input (including variable-size inputs). This allows sketches to accept

multiple different types over the same input port. Each supported type has its own independent preprocessing function.

This does require the sketch author to do some runtime checking of the input value within the sketch, or to normalize the input values in the preprocessing functions, bringing us a bit closer to dynamic type checking instead of static type checking. However, this ability is extremely important for allowing a second layer of interoperability between sketches. If a particular type is either not convertible to the sketch's input type, or if the type must be accepted without the potential data degradation that comes from a lossy conversion, then it can simply be added as an alternate input type for the same port.

Listing 4.1: The input and output definitions for a simple sketch that displays boolean values. This sketch outputs its current state, and can either be toggled by clicking on it, or can be controlled by a boolean input value. Note that `Bool` is a primitive type, allowing the sketch author to work with normal Javascript boolean values within the sketch.

```
this.defineInput(AT.Bool);

this.defineOutput(AT.Bool, function() {
    if (this.inputs.hasValue(0)) {
        return this.inputs.value(0);
    }
    return this.state;
});
```

4.2 Backwards compatibility

As mentioned in section 2.2, Atypical and the new input/output system must allow for some level of backwards compatibility with older sketches. This was achieved by creating a type called `Unknown`. `Unknown` is simply a wrapper around any Javascript value and carries with it no additional type information. In addition, almost all Atypical types are convertible *to* `Unknown` but none are convertible *from* it.

Because `Unknown` is not much more than a simple wrapper, converting a value of another Atypical type to `Unknown` and then unwrapping the `Unknown` is effectively equivalent to converting the Atypical type to a normal Javascript value. With these conversions, we can define the inputs and outputs of these legacy sketches to be `Unknown`. This allows the output of most of the new-style sketches to be connected to the inputs of the old-style sketches, and allows new-style sketches to *opt in* to receiving values from old-style sketches by defining `Unknown` as one of their accepted input types.

4.3 Integration with Chalktalk's user interface

When performing presentations, users now have direct feedback as to whether the connection they're attempting to make is valid or not. Chalktalk no longer creates links from sketches that have no outputs, no longer allows links to sketches that have no inputs, and only allows links between sketches whose types can be properly converted.

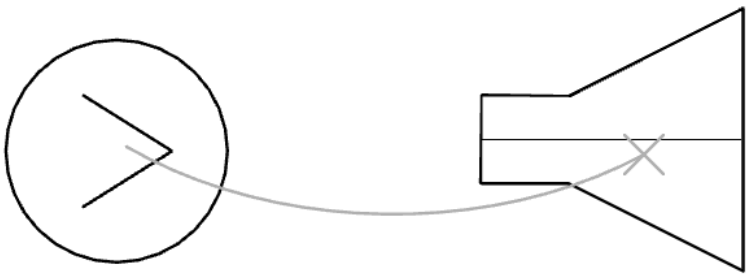

Figure 4.1: The user is attempting to connect a sound file sketch (left)—whose output is a `Function(Seconds, AudioSamples)` *—to a synthesizer sketch whose first input is* `Hertz`*. These types are incompatible, and so Chalktalk prevents the link from being created and shows feedback in the form of an X-shaped arrowhead.*

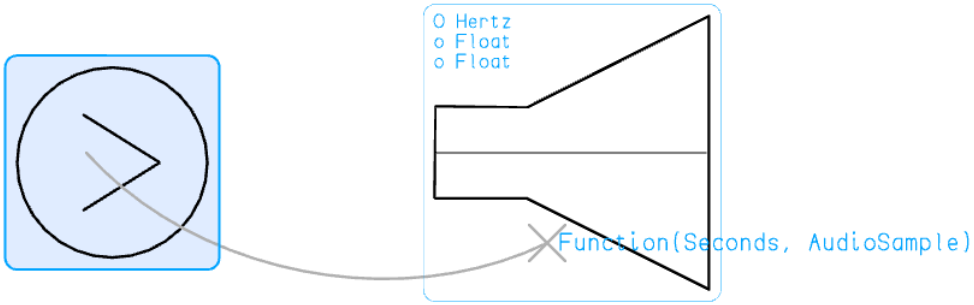

Figure 4.2: The user is attempting the same connection from Figure 4.1, but with help mode enabled. The output type is displayed at the end of the arrow, and the input types are displayed within each sketch, making the reason for the incompatibility immediately apparent.

The more dramatic changes, however, are in help mode. When the user begins a link while in help mode, several visual changes occur:

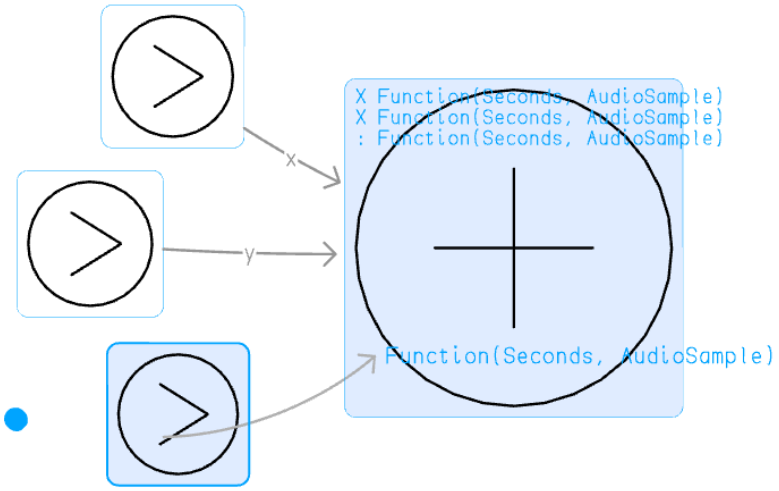

Figure 4.3: Multiple sound files are connected to a mixer sketch that combines their sound into one audio stream. The mixer contains a variable-sized input, which is displayed with dots (:) to its left. Filled inputs are displayed with an `X`*.*

1. The name of the output type is displayed at the end of the arrow the user is drawing.
2. The name of the sketch's input types are shown at the top left of each sketch, in the following format:
    - The names of the accepted types of each input are shown starting at the top left corner of the sketch, with each subsequent input below the previous one.
    - If an input has multiple accepted types, the names of all the accepted types are shown separated by the word "or".
    - The first unlinked input is shown with a large circle to its left (O).
    - Later unlinked ports are shown with a small circle to their left (o).
    - Filled, linked ports are shown with an X to their left.
    - Variable-size ports are shown with dots to their left (:).
3. Sketches that are compatible with the output type of the link being drawn are highlighted in blue.

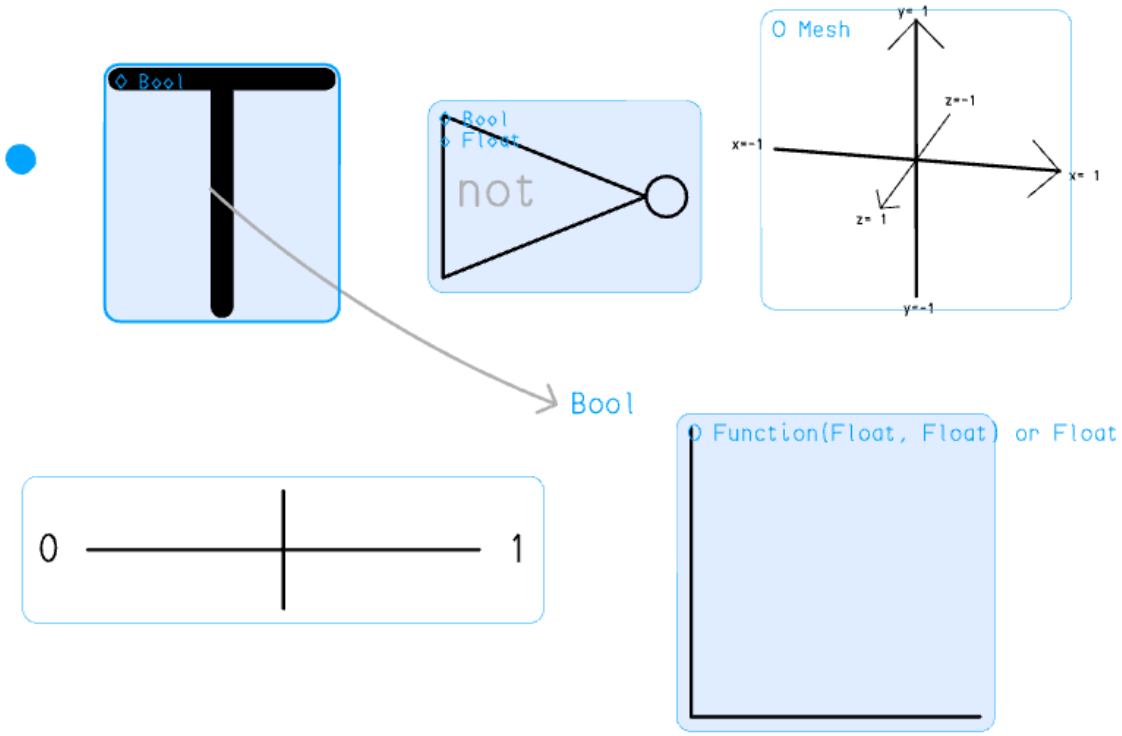

Figure 4.4: A link is being dragged from a sketch that outputs a `Bool` *value. Sketches that are compatible with this link are highlighted in blue. Compatible sketches are the "not" gate, which takes in a* `Bool`*, and the graph, which takes in a* `Float`*, which* `Bool` *can be converted to. Incompatible sketches include the slider, which has no inputs, and the axes, which are intended to display 3D* `Mesh` *objects.*

Evaluation of Atypical

5.1 Evaluation against the constraints and criteria

We can now look back at our constraints and criteria from section 2.2, and evaluate the current incarnation of Atypical against them:

1. *Atypical should allow Chalktalk to discern whether two sketches have compatible input and output types.*

   Atypical succeeds at this by requiring sketch authors to specify the desired input and output types of each sketch.

2. *Atypical should allow Chalktalk to inspect these types at runtime so it can display visual feedback to the user.*

   This has been implemented, and is shown by the changing arrow heads as the user draws a link.

3. *Atypical should allow the user to connect any sketches that have data that are "close enough" to be interpreted.*

   The system for defining conversions, and the "type graph" it creates, strikes an acceptable middle ground between overly-strict typing that harms interoperability and overly-loose typing that harms type safety. In addition, it allows for tweaking the amount of conversion in particular instances for particular types.

4. *Atypical and its integration with Chalktalk should support live performances by avoiding intrusive errors that can spoil the show.*

   Atypical achieves its goals of improving on feedback for the user without displaying intrusive error messages or crashing.

5. *Atypical should allow Chalktalk to display the types for novice users in help mode, allowing them to discover what kind of connections are and are not possible.*

   Help mode now contains a significant amount of type information displayed when the user begins a link, in a way that is useful for novice users.

6. *The type system should avoid placing an undue burden on sketch authors.*

   While it increases the burden to a certain degree, by requiring sketch authors to specify and manage the types of data, this is actually an improvement over the previous status quo.

   The key point to be made here is that these issues with compatibility had always existed, but were previously implicit. There were no guarantees as to the type of data sketches were to receive, and so sketch authors were required to implement runtime type checks in order to avoid errors. Atypical requires sketch authors to specify the types for their sketches, but it transitions the problem from being implicit to being explicit, and helps guarantee that fewer runtime errors will occur once the types are specified.

As for our practical constraints:

1. *Atypical must be compatible with Chalktalk's Javascript-based infrastructure.*

   Being implemented in Javascript was the easiest way to achieve this goal.

2. *Atypical and its integration with Chalktalk should allow for some level of backwards compatibility with Chalktalk's older sketches that do not declare types.*

   The `Unknown` type, and its interoperability with the older input and output system, implements this to an acceptable degree.

3. *Atypical should allow, with sufficient effort, for all the current sketches in Chalktalk to be migrated to use Atypical.*

   Of the nine examples of different data sent over sketch links given in section 1.4, eight of them have already been migrated to Atypical and the last can be relatively easily ported. A review of Chalktalk's sketch library showed that no sketches implemented pathological behavior that would be impossible to define in Atypical.

5.2 Additional benefits

Atypical has also aided interoperability between some sketches. Rather than implementing compatibility with a wide variety of data types within each sketch, these concerns can now be brought up to the type system level by implementing type conversions. Figure

5.1, for example, shows a scenario that would have been much more difficult, or even impossible, to implement using Chalktalk's existing linking system, involving a type-system level conversion between two very disparate types, and an output link that allows multiple pieces of data to sketches that accept either.

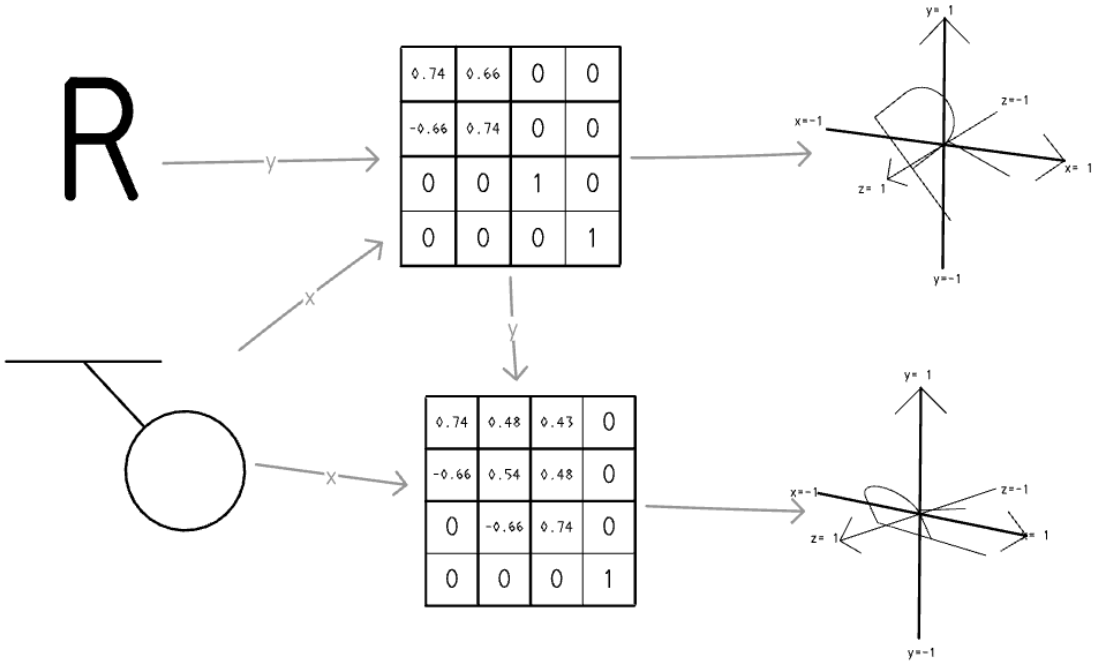

Figure 5.1: A swinging pendulum (outputting `Radians`*) is acting as a control input to two matrices implementing a rotation operation. The "R" is outputting a* `String`*, which is being converted at the type system level into a 3D* `Mesh` *in the shape of an "R". The outputs of the matrix sketches are a* `Pair(Matrix, Mesh)` *containing both the values of the matrix itself and the "R" mesh after being transformed by the matrix. The first set of axes shows the "R" after the first rotation (about the Z axis), receiving only the* `Mesh` *part of the pair through a generic type conversion. The second matrix receives the whole* `Pair`*, multiplies the first matrix by its own internal state (which contains a rotation about the X axis), and outputs both the multiplied* `Matrix` *value and the twice-rotated* `Mesh` *to the final set of axes, where the* `Mesh` *is again plotted showing the combined rotation.*

5.3 Limitations of the design

This approach did have a few drawbacks, however. First and foremost, the performance of this new system is somewhat worse than the old one, largely due to the conversions that this system has to do at every frame in order to implement the type conversions. The asymptotic complexity is typically linear in the number of links, but when called with a densely connected board 60 times per second, even a linear slowdown can be noticed on slower computers. In most cases, the performance degradation is not enough to impede functionality, but it becomes noticeable in certain cases, such as the `AudioSample` sketches, or when help mode is enabled with a large number of sketches and links on the screen.

In addition, while this type system does not create a burden on sketch designers, and even frees them from having to consider certain classes of errors, it adds the necessity of maintaining the type system itself, of adding new types and new conversions where needed. These can sometimes be complex to manage, especially when it comes to broad intermediary conversions that can greatly impact the type conversion graph.

Future work

There is still much room for exploration and future research with this system.

To begin with, both the linking system and Chalktalk as a whole would benefit from more formal user testing. While I did gather informal feedback throughout the project from both novice and experienced Chalktalk users and sketch authors—the earlier ones of which drove the decision to take on the specific project of reforming the linking system—a more formal set of user tests would give far greater clarity as to which parts of the interface are difficult to use or confusing, both in the linking system and in Chalktalk as a whole.

At the moment, the only way to define conversions between two different generic types, such as `Pair` and `Array`, is to define explicit conversions for each of their constructed types. This is limiting, and Atypical should allow large classes of conversions to be defined at once, such as between any `Pair` to any `Array`, or, for example from any `Array(T)` to `String` wherever `T` is convertible to `String`.

Although I specified in section 3.5 that broad intermediary conversions were not "chainable", to avoid unplanned spurious conversions from appearing in distant types, it may be worth exploring the benefits of allowing this. This would involve broadening the intermediary conversion functionality to essentially becoming a graph search algorithm, and would allow things such as the scenario depicted in Listing 6.1:

Listing 6.1: If intermediary conversions were chainable, the two types `A` and `B` would for all intents and purposes become equivalent, where `A` can convert to everything that `B` can convert to and vice-versa, including other broad intermediary conversions.

```
defineConversionsViaIntermediary(null, A, B);
defineConversionsViaIntermediary(B, A, null);
defineConversionsViaIntermediary(A, B, null);
defineConversionsViaIntermediary(null, B, A);
```

This would allow far a greater number of conversions to be defined by defining broad intermediary conversions, but likely at an additional performance penalty and a cognitive cost on the users and developers in understanding how two types came to be convertible to each other.

Finally, Chalktalk's links are all visually depicted as arrows, which works very well for static sketches but is less well-suited to sketches that have significant amounts of animation. Not only that, but each link must be explicitly created by the performer, and each link is one-to-one by design. It may be worth exploring different possibilities for how sketches can interact, such as broadcast messaging or sending data based on proximity. This would enable a much wider variety of interactions between different sketches.

Conclusion

This paper investigated the question of what a type system would look like when applied not to a programming language, but to Chalktalk, a visual communication language meant for real-time presentation and communication. In investigating the requirements of such a type system, we identified where they differed from the typical requirements of a type system meant for programming languages, such as an increased need for flexibility and interoperability, and a decreased need for strict static type safety assurances and intrusive error messages. The system presented, termed Atypical, is a type system built on top of Javascript to meet those requirements.

While Atypical does make some tradeoffs in terms of performance and development burden, it was shown to be more than capable of acting in place of the previous dynamic type system, with significantly increased type safety and much more robust user feedback. In the end, Atypical allows Chalktalk to more safely perform all the same functions it always has, while enabling it to perform new functions it never could.

Bibliography

Luca Cardelli. 1991. *Typeful programming.* Springer-Verlag.

Luca Cardelli. 1996. Type systems. *Comput. Surveys* 28, 1 (1996), 263–264.

Ben Cherry. 2010. Javascript module pattern: In-depth. (March 2010). `http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html`

European Computer Manufacturers Association, ECMA. 2016. Ecmascript language specification. (June 2016). `https://www.ecma-international.org/ecma-262/7.0/index.html`

Adrian Freed, John MacCallum, and Andrew Schmeder. 2011. Dynamic, Instance-based, object-oriented programming in Max/MSP using open sound control message delegation.. In *ICMC*.

Go Team. 2016. *The Go programming language specification.* Technical Report. Google Inc. `https://golang.org/ref/spec`

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.* 18, 2 (March 1996), 109–138. `https://doi.org/10.1145/227699.227700`

Lauren McCarthy, Processing Foundation, and NYU ITP. 2017. p5.js. (2017). `https://p5js.org/`

Harold Pashler, Patrice M. Bain, Brian A. Bottge, Arthur Graesser, Kenneth Koedinger, Mark McDaniel, and Janet Metcalfe. 2007. Organizing Instruction and Study to Improve Student Learning. IES Practice Guide. NCER 2007-2004. *National Center for Education Research* (2007).

Ken Perlin. 2016. Future Reality: How Emerging Technologies Will Change Language Itself. *IEEE Computer Graphics and Applications* 36, 3 (May 2016), 84–89. `https://doi.org/10.1109/MCG.2016.56`

Processing Foundation. 2017. Processing.org. (2017). https://processing.org/

William Robert Sutherland. 1966. *The On-line Graphical Specification of Computer Procedures*. Ph.D. Dissertation. Massachusets Institute of Technology.

David Ungar and Randall B Smith. 1987. *Self: The power of simplicity*. Vol. 22. ACM.

Yuhong Xiong and Edward A. Lee. 2000. *An Extensible Type System for Component-Based Design*. Springer Berlin Heidelberg, Berlin, Heidelberg, 20–37. https://doi.org/10.1007/3-540-46419-0_2