# Living it up with a Live Programming Language

Sean McDirmid
École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
sean.mcdirmid@epfl.ch

## Abstract

A dynamic language promotes ease of use through flexible typing, a focus on high-level programming, and by streamlining the edit-compile-debug cycle. **Live languages** go beyond dynamic languages with more ease of use features. A live language supports **live programming** that provides programmers with responsive and continuous feedback about how their edits affect program execution. A live language is also based on high-level constructs such as declarative rules so that programmers can write less code. A live language could also provide programmers with responsive semantic feedback to enable time-saving services such as code completion. This paper describes the design of a textual live language that is based on reactive data-flow values known as **signals** and **dynamic inheritance**. Our language, SuperGlue, supports live programming with responsive semantic feedback, which we demonstrate with a working prototype.

***Categories and Subject Descriptors*** D.2.6 [*Programming Environments*]: Interactive Environments—live programming; D.3.2 [*Programming Languages*]: Data-flow, Very High-level, and Object-oriented Languages; D.3.3 [*Language Constructs and Features*]: Inheritance—dynamic.

***General Terms*** Human Factors and Languages

## 1. Introduction

Dynamic programming languages support rapid development by reducing verbosity in both source code and the development process. Although labeled as "dynamic" because they often rely on dynamic typing, dynamic languages are best characterized by their emphasis of flexibility, conciseness, and ease of development over performance and generality. Dynamic typing promotes flexibility by enforcing fewer restrictions on type compatibility and conciseness by eliminating verbose type annotations. Dynamic languages also emphasize high-level programming models, often based on higher-order functions or objects, that reduce the amount of code needed to express a program. Together, flexible typing and high-level programming models support component programming so that programmers can create feature-rich programs with little code by reusing existing components. Finally, dynamic language environments streamline the edit-compile-debug cycle by either eliminating or hiding compilation and by allowing code to be changed while a program is running. For example, many Lisp [25], Smalltalk [12], and Erlang [1] environments support *hot swapping* where a program's code can be updated without restarting.

Can we design languages that are easier to use than existing dynamic languages? For inspiration, look at visual languages such as Fabrik [15], LabVIEW [21], and spreadsheet languages such as Excel and Forms/3 [3], which are designed to be used by casual programmers, novices, and even end users. Visual languages are based on simple, often declarative, programming models that heavily emphasize component reuse. For example, Apple's Quartz Composer [17] supports the assembly of animation components through reactive data-flow connections that encapsulate change-propagation details. The simple programming models of visual languages also enable *live programming*, which provides programmers with immediate and continuous feedback about how their edits affect program execution. For example, editing a connection graph in Quartz Composer immediately changes the composed animation. Compared to live programming, the hot swapping supported by dynamic languages is mushy: the effect of edited code on program behavior is not apparent until the code is re-executed. For example, editing the Smalltalk statement "`w fill: red`" to "`w fill: blue`" will not immediately re-color blue all widgets that have been bound to `w`.

In spite of their benefits, visual languages do not replace textual dynamic languages. For one thing, text can be written more quickly [23]. More significantly, visual languages must deal with a *scaling-up problem* [4]: the ability to build large programs is hindered by concrete visual notation that does not easily support abstraction. Modern visual languages solve the scaling up problem in a variety of ways; e.g.,


*OOPSLA'07,* October 21–25, 2007, Montréal, Québec, Canada.

Forms/3 [3] allow for name-based cell referencing and data abstraction through generic forms. Still, text's natural support for naming makes it more suitable for many abstraction-heavy programming tasks. Also, the simple programming models that enable live programming in visual languages can also be applied to textual languages. A *live language* is then a textual or visual language that supports live programming through a simple programming model.

Support for textual naming in a live language is problematic: how are programmers provided with responsive semantic feedback about what names are valid? Semantic feedback speeds up the programming process by detecting typos early on and enabling services such as code completion, which allows programmers to browse available resources. Programmers are often deprived of responsive semantic feedback in dynamically-typed languages because names are difficult to resolve before run-time. Static typing that typically enables semantic feedback is too verbose and inflexible to be used in a live language. Implicit typing would work better in a live language where inference provides responsive semantic feedback without type annotations.

This paper explores the design of a textual live language that reuses ideas from dynamic and visual languages along with a few new ideas. The language, SuperGlue, is based on a simple reactive data-flow programming model that is suited to building interactive programs out of existing components. Components in SuperGlue are connected together through *signals* into a data-flow graph. Being reactive, the values that flow across signal connections can change over time, which eases the expression of interactive programs. For scaling purposes, signals extend classes through *dynamic inheritance*, where what classes a signal extends can change at run time. Signals and dynamic inheritance seamlessly support live programming by masquerading edits as programmatic changes in the program's mutable execution state. Unlike Self [29] and like Cecil [6], dynamic inheritance in SuperGlue is based on declarative predicates [7], which is very compatible with a reactive data-flow model. Because class extensions are declarative, type inference can provide programmers with responsive semantic feedback. Dependent typing improves type inference so that type annotations are unnecessary.

Previous work [20] introduces SuperGlue's support for object-oriented signals, and shows how this support eases the construction of interactive programs. This paper expands on this work by augmenting SuperGlue with dynamic inheritance, which significantly simplifies the language, and live programming. Figure 1 shows how live programming occurs in a working Eclipse-based prototype. In this paper, we use a PacMan-like game as a running example. A program is edited freely as text in the upper half of the programming environment while the execution is shown in the lower half. As soon as a key press leads to valid code, the effect of the edit is incorporated into the executing program. So that live programming remains responsive, SuperGlue supports error recovery where syntax, semantic, and execution errors are noted in the background while execution continues. The live programming experience that is shown as a frame-expanded movie[1] in Figure 1 is narrated as follows:

(a) The programmer types "`port pacMan ▷ Figure`" to define a `pacMan` signal that extends the game's `Figure` class so `pacMan` is drawn on the game's canvas. Because a shape for `pacMan` has not been specified, it is drawn by default as a rectangle.

(b) The programmer types the extension "`pacMan ▷ Pie`," which causes `pacMan` to extend the `Pie` class and be drawn as a circle. The shape of `pacMan` is updated on screen as soon as the programmer types the "e" in `Pie`.

(c) The programmer types the connection "`pacMan.fill = yellow`," causing this circle to be immediately colored yellow. Through type inference, code completion can complete typing for `.fill` in `pacMan`.

(d) The programmer types "`pacMan.extent = 330`." As the programmer types each character in "330," `pacMan`'s appears respectively as a `357`, `327`, and finally a `30` degree slice. The programmer than finishes this line with "`+ (time % 30)`," which causes `pacMan`'s mouth to close continuously in a `30` degree motion.

The rest of this paper is organized as follows. Section 2 lists live language design problems and how these problems are solved in SuperGlue. Section 3 describes how SuperGlue supports live programming. Section 4 describes preliminary experience on SuperGlue's use in developing a simple game. Section 5 presents related work and Section 6 concludes.

## 2. Language Design

SuperGlue is based on the data-flow programming model that was originally introduced by Karp and Miller [16] to deal with concurrency through a graph program representation. Even without considering concurrency, the data-flow model is appealing to live languages because all computational dependencies are encoded explicitly. The data-flow model by itself does not scale: non-trivial programs involve a large or unbounded number of data-flow connections that are tedious or impossible to express individually. For this reason, SuperGlue augments its data-flow model with object-oriented constructs. As a brief overview, SuperGlue consists of the following constructs:

– **Signals**, which are data-flow values that facilitate inter-component communication. Signals are objects that extend classes, are connected to other signals, and contain signals defined in their extended classes. Signals are defined as class members using the `port` keyword.

[1] Movies are frame expanded in this document. A preferred electronic document with inlined movies is available on the author's website.

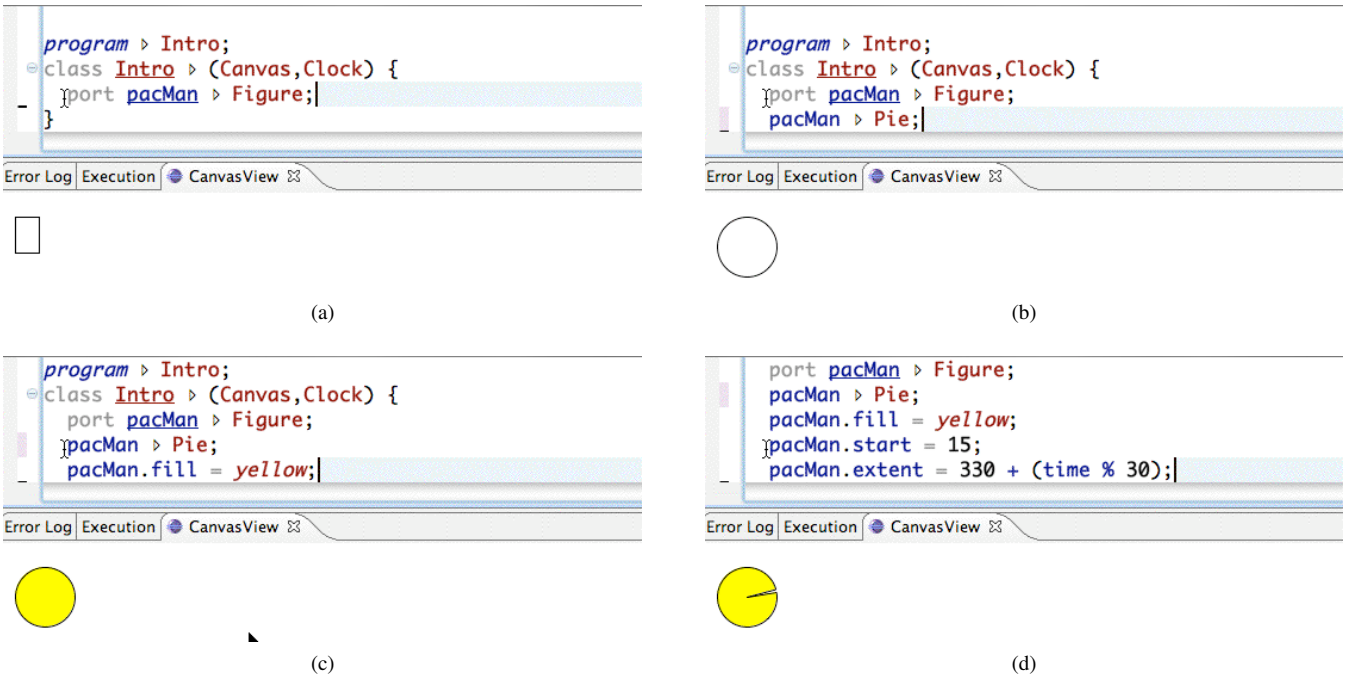


Figure 1: An example of the SuperGlue programming environment being used in a live programming session; the editor is in the top half of each sub-figure while the running program is shown in the lower half.

– **Classes**, which are types that encapsulate behavior. Classes are not values and they can only be extended by signals. Classes are defined at the top-level of a program or as members in other classes using the `class` keyword.
– **Connections**, which relate signals to other signals so that their values are equivalent. Connections are rules that are similar to simple uni-directional constraints. Connections are expressed with the = operator.
– **Extensions**, which cause signals to extend classes. Extensions are rules that can target signals outside of their definitions. Extensions are expressed with the ▷[2] operator.
– **Conditions**, which guard when connection and extension rules can be applied. Conditions query existing connection and extension relationships through the same = and ▷ operators that are used to create these relationships in rules. Conditions are expressed in `if` clauses that surround guarded connection and extension rules.

As an opening example, consider the following SuperGlue code that defines the `Ghost` class, three ghost signals, and connects the fill color of all ghost signals:

```
1 class Ghost { port fill ▷ Color; }
2 port (blinky, clyde, bashful) ▷ Ghost;
3 Ghost.fill = blue;
```

The first line of this code defines a `Ghost` class with a member `fill` signal that extends the `Color` class. The second line defines three signals, `blinky`, `clyde`, and `bashful`, which extend the `Ghost` class. The third line is a rule that connects the `fill` signal of all ghosts, including `blinky`, `clyde`, and `bashful`, to the color blue. When used in expressions, class identifiers are variable that quantify over all signals that extend the identified class; e.g., `Ghost` in `Ghost.fill` is a variable that quantifies over all signals that extend the `Ghost` class. Unlike imperative assignments, connection rules have declarative meaning that is unrelated to statement execution order; e.g., on line three, blue is continuously connected to rather than discretely assigned to the ghost's fill color.

The nature of the values that flow across data-flow connections determines what programs can be concisely expressed in a data-flow language. In SuperGlue as well as many existing textual and visual data-flow languages, such as Lucid [32] and Quartz Composer [17], data flow is **reactive** in that time-varying values flow across connections. Reactive data-flow languages hide change-propagation details from programmers by dealing with the details generically at connection end points. By hiding change-propagation details, continuously changing state is treated in the same simple way that immutable state is. Consider the following code that causes `pacMan`'s mouth to close continuously:

```
1 class Pie { port extent ▷ Number; }
2 port pacMan ▷ Pie;
3 pacMan.extent = 330 + (time % 30);
```

The last line of this code connects `pacMan`'s `extent` signal to an expression that continuously increases from 330 until 360 degrees, causing `pacMan`'s mouth to shut continuously in

[2] The user types `:`, which is then rendered in the editor as ▷.

a 30 degree motion. Such behavior occurs because the `time` signal is a periodically incrementing number; e.g., it is connected to 1042 and, after ten milliseconds, it is re-connected to 1043. At connection end-points, signals in SuperGlue are provided and consumed by components that express event handling details through a general-purpose language such as Java or Scala [22]. In our example, Scala code updates the `time` signal when time advances and redraws `pacMan` when its `extent` signal changes. We describe how this code interfaces with SuperGlue in Section 2.3. Because SuperGlue code is completely oblivious to what happens over signal connections, interactive programs, where continuous change is common, are expressed more easily. As we will see in Section 3, such obliviousness also enables live programming.

In our previous version of SuperGlue [20], class extensions could only be expressed in signal and class definitions. This restriction complicated SuperGlue's class system and prevented it from being very dynamic; i.e., signal types could not change at run-time. As a simplification and to support live programming, class extensions are now rules in the same way that connections are. For example, the statement "`pacMan ▷ Figure`" is an extension rule that can be written down wherever `pacMan` is visible–not just where `pacMan` is defined. As rules, both connections and extensions can be guarded by conditions. Consider the following code that causes signals to extend the `PositionByKeyboard` class when they already extend both the `Position` and `Keyboard` classes:

```
1 class Position; class Keyboard;
2 class PositionByKeyboard;
3 if (Position ▷ Keyboard)
4   Position ▷ PositionByKeyboard;
```

The condition on line three of this code ensures that the `Position` identifier is only bound to signals that extend both the `Position` and `Keyboard` classes. Line four is an extension that then causes these signals to extend the `PositionByKeyboard` class. As we will see in Section 2.2, such extensions are useful in adapting incompatible signals.

Extension rules in SuperGlue target signals and not classes. As syntactic sugar, an extension can be specified in a class definition, but the extension is applied to all signals that extend the defined class and not the class itself. No cyclic inheritance occurs in SuperGlue when the signals of two different classes extend the other class–instead, the classes are semantically equivalent in that a signal that extends one also extends the other. Such mutual extension constructions enable the refinement of signals that have already been created. For example, the definition "`class RichNumber ▷ Number`" and the extension rule "`Number ▷ RichNumber`" does not create a cycle; instead it causes all `Number` signals, including numeric literals, to be `RichNumber` signals and vice versa.

Because conditions query signals whose values change over time, their truth values can also change over time. Because time-varying conditions can guard extension rules, SuperGlue supports *dynamic inheritance* where what classes a signal extends can vary over time. Dynamic inheritance enables the state-based classification of signals without deeply nested `if` statements, and, as described in Section 3, supports live programming without the need for special-case semantics. Unlike Self [29], where dynamic inheritance occurs through imperative slot assignment, dynamic inheritance in SuperGlue is predicated as in Cecil [6, 7]. Consider the following code that causes `pacMan` to extend the `Damaged` class when it overlaps with a ghost:

```
1 class Figure { port (overlaps,hit) ▷ Hit }
2 class Ghost ▷ Figure; class Damaged ▷ Figure;
3 port pacMan ▷ Figure;
4 if (pacMan.overlaps ▷ Ghost.Hit)
5   pacMan ▷ Damaged;
6 Damaged.fill = red;
```

For the duration that `pacMan` overlaps with some figure, `pacMan`'s `overlaps` signal is bound to the `hit` signal of this figure. The condition on line four is then only true for durations where the figure that `pacMan` overlaps with is a ghost. During these durations, `pacMan` extends the `Damaged` class, inheriting all the behavior of this class and being subject to rules that target signals of this class. For example, the connection of red to a damaged figure's `fill` color would apply to `pacMan` while it overlaps with a ghost. As soon as `pacMan` no longer overlaps with a ghost, it unextends the `Damaged` class and the special damaged behavior no longer applies.

### 2.1 Implicit Typing

Dynamic languages avoid static typing because of its verbose type annotations, limited flexibility, and inability to deal with change. However, static typing also provides programmers of responsive semantic feedback that is especially important in a live programming environment: it allows programmers to converge on correct edits more quickly through code completion, where programmers can browse the names of available entities, and validation, which immediately informs programmers of typos. Static types can also convey programmer intent and resolve ambiguity. In SuperGlue, multiple connections can connect the same signal. To resolve ambiguity, connections that target more specific static types are considered before connections that target less specific static types if the guarding conditions of both connections are true. Such by-type prioritization is analogous method overriding in most object-oriented languages.

To provide semantic feedback without type annotations, types are computed in SuperGlue through type inference, which is a form of *implicit typing*. Implicit typing is viable in SuperGlue because extensions are declarative rules that can be accurately inferred by an analysis that is both flow and context-insensitive. The inferred type of a symbol is a set of classes that restricts the symbol from only being bound at run-time to signals that extend all of these classes. Subtyping is then defined as a subset relationship: $t_a$ is a subtype of $t_b$ if all classes in $t_b$ are classes in $t_a$. Type inference builds up a symbol's inferred type by analyzing surrounding extension conditions and known extension rules that target classes already contained in the inferred type.

```
1 class Duplicator
2 { size ▷ Number;
3   class T
4   class get ▷ T { port index ▷ Number; }  }
5 port ghosts ▷ Duplicator;
6 ghosts.T ▷ (Ghost,Figure);
7 ghosts.size = 10;
8 if (ghosts.get.index = 3)
9   ghosts.get.fill = red;
```

Figure 2: SuperGlue code that creates ten ghosts, where the ghost at index three is colored red.

As an example of how type inference occurs, consider the following code that overrides a position's `x` signal when the position extends the `PositionCell` class:

```
1 class Position { port (x,y) ▷ Number; }
2 class PositionCell
3 { port (xc, yc) ▷ NumberCell; }
4 Position.x = 0;
5 if (Position ▷ PositionCell)
6   Position.x = Position.xc.result;
```

Overriding in SuperGlue is defined as follows: if the type of $a$ is $t_a$, the type of $c$ is $t_c$, and $t_c$ is a strict sub-type of $t_a$, then the connection $a = b$ overrides the connection $c = d$. On line four, the inferred type of `Position` includes only the `Position` class because the type of a class identifier always contains itself. On line six, `Position`'s inferred type also contains the `PositionCell` class because of the surrounding extension-querying condition on line five. As a result, the connection on line six, which targets the type `(Position,PositionCell)`, overrides the connection on line four, which targets the type `(Position)`. Because `Position`'s inferred type on line six contains the `PositionCell` class, the member signal `xc` can be accessed through `Position` and is listed by code completion.

SuperGlue supports a form of dependent typing where class types are prefixed by their containing signals. When type inference computes a symbol's type, prefixes are approximated as signal and class symbols. Consider the code in Figure 2 that creates ten ghosts. On line nine, the inferred type of `ghosts.get` contains the following classes: `ghosts.get` by definition, `Duplicator.get` because the `ghosts` prefix extends `Duplicator` on line five, `ghosts.T` because of the extension on line four and by a binding of `Duplicator` to `ghosts` on line five, `Duplicator.T` by line five again, and finally, `Ghost` and `Figure` by the extension on line six. On line nine, the `fill` signal can be accessed and connected because `ghosts.get` signals extend `Figure`.

## 2.2 Dynamic Typing and Adaptation

In SuperGlue, because both connections and extensions can be guarded by state-dependent conditions, when they are applied at run-time is not statically known. As a result, SuperGlue depends on dynamic typing to verify two connection properties: first, that primitive-extending signals are connected to concrete values of the appropriate primitive class; and second, that all signals are connected unambiguously. The first property ensures that component implementations correctly communicate through primitive-extending signals; e.g., a providing component pushes through a number that is consumed as a number by one or more other components. The second property detects multiple active connections to the same signal that do not having overriding relationships. In both cases, dynamic typing ensures that components communicate correct and unambiguous values at run-time.

Independently-developed or differently-purposed components will often not agree on the types of their non-primitive signals. For this reason, SuperGlue permits connections between type-incompatible signals where additional rules can define compatibility. Consider the following code where `pacMan`'s position is connected to a keyboard:

```
1 class Canvas
2 { class Figure { port position ▷ Position; }
3   port keyboard ▷ Keyboard;                  }
4 port game ▷ Canvas, pacMan ▷ game.Figure;
5 pacMan.position = game.keyboard;
```

The signal types involved in the connection on line five are incompatible: the type of `pacMan.position` contains only the `Position` class while the type of `game.keyboard` contains only the `Keyboard` class. The programmer intends the position of `pacMan` to be controlled by the keyboard. Such intent can be realized with code that defines what it means for a signal to extend the otherwise unrelated `Position` and `Keyboard` classes. At run-time, the above connection causes `pacMan`'s `position` signal to extend both the `Position` and `Keyboard` classes. These classes can then be combined to describe the target of an extension rule:

```
1 if (Position ▷ Keyboard)
2   Position ▷ PositionByKeyboard;
```

By this extension, a signal that extends both the `Keyboard` and `Position` classes also extends the `PositionByKeyboard` class, which we refer to as an *adapter mixin*. Normal mixins enable a form of linearized multiple inheritance where a mixin is explicitly applied to an object. An adapter mixin is implicitly applied to a signal that extends a combination of classes; e.g., a signal that extends `Position` and `Keyboard` will implicitly extend `PositionByKeyboard`. With implicit application, adapter mixins can adapt connections to automatically resolve type incompatibility. The `PositionByKeyboard` class resolves the incompatibility between position and keyboard as follows:

```
1 class PositionByKeyboard▷ (PositionCell,Keyboard)
2 { go_up   = key_up  ; go_down  = key_down ;
3   go_left = key_left; go_right = key_right; }
```

The `go_*` and `key_*` signals are respectively defined in the `PositionCell` and `Keyboard` classes whose definitions are not shown. The connections in `PositionByKeyboard` cause imperative position state that is maintained by `PositionCell` to be incremented according to keyboard input and a

time signal that is not shown. This code is applied whenever positions are connected to keyboards. For example, if multiple figure positions are connected to the keyboard, these figures will move in unison by keyboard input.

### 2.3 Implementation and Code Behind

Our prototype implementation of SuperGlue conceptually builds and maintains a dependency graph that tracks how signal values are inter-dependent. So that dependencies are updated reactively, our prototype is based on the UI pattern of *damage and repair*: when a signal changes, its dependencies are "damaged" and scheduled in a work-list for "repair" through re-evaluation. Damage is then propagated through the dependency graph. Maintaining an exact dependency graph is unrealistic because signals can depend on each other in cyclic ways that are difficult to track. Instead, our prototype is conservative: a dependency is recorded when the evaluation of signal $A$ depends on signal $B$, but is never explicitly "unrecorded" when $A$ no longer depends on $B$. Dependent signals are instead flushed after being damaged–if dependent signals still depend on the changing signal, then this dependency is re-recorded during repair. Flushing ensures that signals are notified only once of changes in signals that they no longer depend on.

SuperGlue code cannot directly express computations such as arithmetic or drawing on a canvas. Instead, these computations are expressed as *code behind*[3] connections and extensions that are written in Java (or Scala [22]). A code behind connection expresses the connection's result in Java code according to the following Java `Signal` interface:

```
interface Signal<T> { T eval(Observer o); }
interface Observer { void notify(); }
```

A signal provider implements the `Signal` interface's `eval` method to provide a signal's current value and ensure that an `Observer` is notified when that value changes. Consider the following Java code that implements a clock's `time` signal:

```
time = new Signal<Number> {
  Number eval(Observer o) {
    p = period.eval(o).toLong();
    timer.schedule(new TimerTask() {
      void run() { o.notify(); }}, p);
    return System.currentTimeMillis / p;  }};
```

The `time` signal's `eval` method schedules a timer task to notify the caller that the `time` has changed, which occurs in `period` milliseconds. A signal consumer calls the `Signal` interface's `eval` method to access a signal's value, and can provide an observer so that it is notified of changes in the value. For example, the `time` signal calls the `eval` method on the clock's `period` signal to obtain the clock's period. The `time` signal's implementation passes its called observer directly into the `period` signal. As a result, when a clock's `period` signal change, its `time` clients are notified directly.

[3] "Code behind" is often used to describe imperative code that implements complicated behavior in an otherwise declaratively-specified user interface.

The SuperGlue runtime produces and consumes signals according to the `Signal` Java interfaces based on the the rules of a SuperGlue program. For example, the connection `period = 100` causes the SuperGlue runtime to create a Java `Signal` object that returns `100` when its `eval` method is called. As we discuss in Section 3, edits to code will notify observers of the signals derived from the edited code; e.g., when the `100` token is changed through editing, the SuperGlue runtime will notify all observers that the `period`'s `eval` method was called with. It is through the implementation and routing of signals via declarative rules that SuperGlue adds its value to the program; i.e., SuperGlue acts as a middle man between components that do the real work.

Code behind extensions are evaluated by the SuperGlue runtime when a signal extends or unextends a class. The Java interface for a class extension is as follows:

```
interface Extension
{ void enter(Signal s);   void exit(Signal s); }
```

The `enter` and `exit` methods of a class extension are called when a signal starts or stops extending a class, which occurs when the signal is created, destroyed, or when the guard of an extension changes. The `enter` and `exit` methods of a class extension are responsible for managing the signal's state with respect to the class. Consider the following Java code that implements the `NumberCell` class extension:

```
NumberCell = new Extension() {
  static Map<Signal,State> signals = ...;
  void enter(Signal s)
  { State state = new State(s);
    signals.put(s, state); state.notify(); }
  void exit (Signal s) { signals.remove(s); }}
```

Number cell state is stored in a map and is keyed by the signal's own identity. The state is implemented as follows:

```
class State extends Observer {
  Double value; Signal signal;
  Set<Observer> clients;
  void notify() {
    if (signal.set_enable.eval(this)) {
      value = signal.set_to.eval(this);
      foreach (o : clients) o.notify();
      clients.clear();     }}}
```

A number cell `State` object is an observer that queries the `set_enable` and `set_to` signals when called. If the `set_enable` signal is true, then the value of the number cell is set to the value of the `set_to` signal while the number cell's observers are notified and flushed. The number cell `State` object is used as its own observer in queries to the `set_enable` and `set_to` signals, so it will be called whenever the values of these signals change. The number cell `get` signal is implemented to access number cell state as follows:

```
get = new Signal<Number>() {
  Number eval(Observer o) {
    state = NumberCell.signals.get(container());
    state.clients.add(o);
    return state.value;  }}
```

When evaluated, the `get` signal looks up number cell state according to its containing `NumberCell` signal. It then adds the called observer as a client of the `NumberCell` state object and returns the `NumberCell`'s current value. Beyond managing state, code behind extensions can implement behavior that reacts to a signal extending a class. For example, the code behind extension for a canvas's `Figure` class causes extending signals to be drawn on the canvas.

Signal initialization involves calling class extension `enter` methods to allocate the signal's state and the initialization of the signal's sub-signals. To break cycles, signals that are defined recursively are not initialized, although they can still be used. Currently, signal initialization is a significant drag on performance: even the most trivial signal requires extensive initialization, which is the reason object creation in Section 4 is very expensive. Additionally, with our current scheme, extensions can cause an unbounded number of objects to consume individual resources. Consider the code `Number ▷ canvas.Figure`, which expresses that every number signal in the program's universe should be drawn as a figure in `canvas`. Since the number of `Number` signals in a program is probably quite large, writing such a statement could cause the program to lock up. We plan to fix initialization by changing extension notification semantics so that some classes can act more like concrete sets: code behind could iterate over all of the class's extending signals, as well as be notified of when signals enter or leave the set. To enable iteration, extension of such a class would be restricted to signals that have a fixed location in the connection graph with respect to the class. As an example, if `Figure` in a canvas object is such a restricted class, then it could only be extended by signals that can be addressed from the canvas object; e.g., the code `Number ▷ canvas.Figure` would fail because there is no concrete path of connections from arbitrary `Number` signals to the `canvas` signal. Correspondingly, state management must become more implicit with garbage collection-like semantics. Implementing these features in our prototype is left to future work.

## 3. Live Programming

Live programming in SuperGlue is enabled by its simple reactive data-flow model and a programming environment that takes advantage of this model. By adhering to the declarative data-flow model, the code of a SuperGlue program has a direct relationship with its execution state. As a result, the programming environment can responsively update program execution state according to programmer edits. The object-oriented features that improve SuperGlue's scalability are declarative and do not interfere with this property; e.g., dynamic inheritance occurs via declarative predicates rather than imperative slot assignment. SuperGlue is tightly integrated into a programming environment that includes a source code editor with modern conveniences such as auto-indenting, semantic highlighting, code completion, hover help, and hyperlink navigation. These services enhance live programming by allowing programmers to write, read, and navigate code more quickly; e.g., the programmer can use code completion to conveniently browse what signals and classes are available through a signal. As mentioned before, many of these services, such as code completion, would be difficult to implement reliably and responsively without implicit typing.

The SuperGlue code is processed by the SuperGlue runtime in four phases: scanning, parsing, type inference, and execution. All four phases are incrementally performed after every edit. As described in Section 2.3, the execution phase is additionally performed when program state changes; e.g., when the `time` signal advances. All four phases are based on a layered version of the damage and repair pattern that was described in Section 2.3: a keyboard edit damages the token stream, a change in the token stream damages the parse tree, a change in the parse tree damages inferred type information, and finally, a change in inferred type information can damage the program's execution.

The live programming experience is sensitive to the **error recovery** and **edit latency** characteristics of each code-processing phase. Syntactic, semantic, and run-time errors are inevitable while the programmer is typing. In the presence of these errors, a programmer should still receive useful feedback about the other parts of the program. Forcing programmers to immediately fix errors slows them down by forcing them off their preferred development plan. Edit latency is the amount of time that it takes for an edit to percolate through all four phases. Long edit latency reduces the responsive of the feedback provided to programmers, which leads to a "mushy" live programming experience. Users will notice latency at 100 milliseconds and become distracted by latency that is greater than 500 milliseconds.

Editing in the SuperGlue programming environment is syntax-aware and not syntax-directed [28]; i.e., syntax errors are tolerated. Scanning and parsing in our prototype is fairly fast and supports very robust error recovery: in the presence of syntax errors, the program still executes, other parts of the program can be processed reliably, and editor services will still work correctly. We achieve this through precedence parsing [11], which as a bottom-up parsing method, is easily made incremental and very error tolerant. In fact, precedence parsers are not sensitive to syntax errors at all. The flip-side is that precedence parsing is not very expressive and cannot detect syntax errors as conventional BNF-based parsers can. To deal these problems, we augment precedence parsing with enhanced scanning, a brace matcher, and a type checker that also checks for syntax errors. A detailed discussion of these techniques is beyond the scope of this paper and will be reported on in the future.

### 3.1 Type Inference

Incremental type inference is achieved by storing computed type information and typing context at each node in the

parse tree. The parse node's type information is damaged when the node's parse structure, typing context, or child-tree types have changed. Error recovery involves reporting detected errors in the programming environment and then reverting to a previously computed or default types for use in typing parent parse nodes. Also, symbol tables support change operations. When the symbol table is used by a parse node to lookup some name, a dependency with the parse node is recorded in the table. When a symbol is changed, all parse nodes that depend on the symbol's new and old names are damaged. As with scanning and parsing, type errors, such as the failure to find a symbol bound to a name, are reported without stopping the executing program. At worst, a type error acts to disable the current rule being edited.

All connection and extension rules that are encoded in a SuperGlue program are organized by our prototype according to the inferred types that they target. Inference involves traversing rules that are compatible with a symbol's growing inferred type. When a rule is changed in source code, all symbols whose inferred types are affected by the old and new version of the rule are damaged so that repair will recompute their inferred types using the new rule. Because a rule can be used in computing types for a large number of symbols, changing a rule can incur noticeable latency. For example, changing a rule that targets the `Number` class will not only damage all user-defined symbols whose type contains `Number`, but will also damage all numeric literals. In a 280 line SuperGlue program, which we describe in Section 4, changing a rule that targets the `Number` class incurs a typing delay of around three and a half seconds[4]. However, because most classes are only extended a few times in a program, rule changes often incur no noticeable latency.

Various techniques are used in our prototype to speed up incremental type inference. Because of dependent typing, the number of possible types in a program is exponential with respect to the number of declared classes–although in practice only a few types are significant. When editing changes an extension rule that targets an inferred type, all symbols whose inferred types are subtypes of this type must be damaged. All significant inferred types are hashed in our prototype so that the significant sub-types of a type can be quickly computed and its symbols damaged; otherwise the editing of extension rules would be intractable.

As an example of live type inference, consider the frame-expanded movie in Figure 3 that is narrated as follows:

(a) The programmer types `port clyde ▷ Ghost`, where `Ghost` is underlined with red jaggies because it cannot be resolved to a symbol.

(b) As soon as the programmer defines the `Ghost` class, the reference to `Ghost` in `clyde`'s definition becomes valid. The programmer then tries to access `clyde`'s `position` signal, which is underlined with red jaggies because `position` is not a member of `clyde`.

(c) As soon as the programmer causes the `Ghost` class to extend the `MyFigure` class, the access to `clyde`'s `position` signal becomes valid.

(d) Because the `position` signal in all `Ghost` signals extends the `Position` class, the `x` and `y` position signals show up in code completion on `clyde`'s `position` signal.

### 3.2 Execution

Hot swapping in a dynamic language replaces code so that its re-execution occurs according to the new version of the code as opposed to the old version. Changing class members or inheritance relationships can also cause updates to objects in the heap, as is supported in most Smalltalk [12] environments. By being based on a simple data-flow model, live programming in SuperGlue goes much more farther than this: code edits immediately change the program's data-flow graph and the observable execution state of the program. Because of SuperGlue's support for reactivity and dynamic inheritance, live programming can be supported with very few special-case semantics. Edits that change connection and extension rules semantically masquerade as programmatic change. As we described in Section 2, connection and extension rules are guarded by arbitrary conditions whose truth values can change over time. The editing of a rule can then resemble changing the state of an implicit guarding condition. For example, if the programmer types the code "`pacMan ▷ Pie`," the added extension rule can be realized as a pre-existing extension "`if (P) pacMan ▷ Pie`," where condition `P` changes from false to true. In our prototype, rule changes cause dependent run-time signal values to be damaged, where signal re-evaluation occurs according to an up-to-date set of rules. Through this masquerade, only edits that add or remove signal and class definitions need to be special-cased in our prototype.

Because variables can be bound to an unbounded number of signal values, rule changes can result in higher latency than what can occur during type inference. As mentioned in Section 2.3, edits that add new signals to a program incur latency in the initialization of these signals and all of their member signals. Because edit latencies are linear in the number of created signals, so it is easy to imagine how an edit that increases the size of something can take an arbitrary amount of time to process. We given an example of this problem Section 4.2. Initialization latency is currently one of the weakest parts of our prototype and we hope to improve on it in the future through tuning and multi-threading. Note that signal initialize time is needed in whether a change occurs programmatically or through editing. However, this latency acts to distract the programmer during editing

As with the other phases, all errors that occur during the execution phase are duly reported and then shunted off to the background as program execution continues. Errors that

[4] All times are measured in this paper on a 2 GHz Macbook Pro with 2 GB of RAM, running Java 1.5.0_07 and Eclipse 3.3M5.

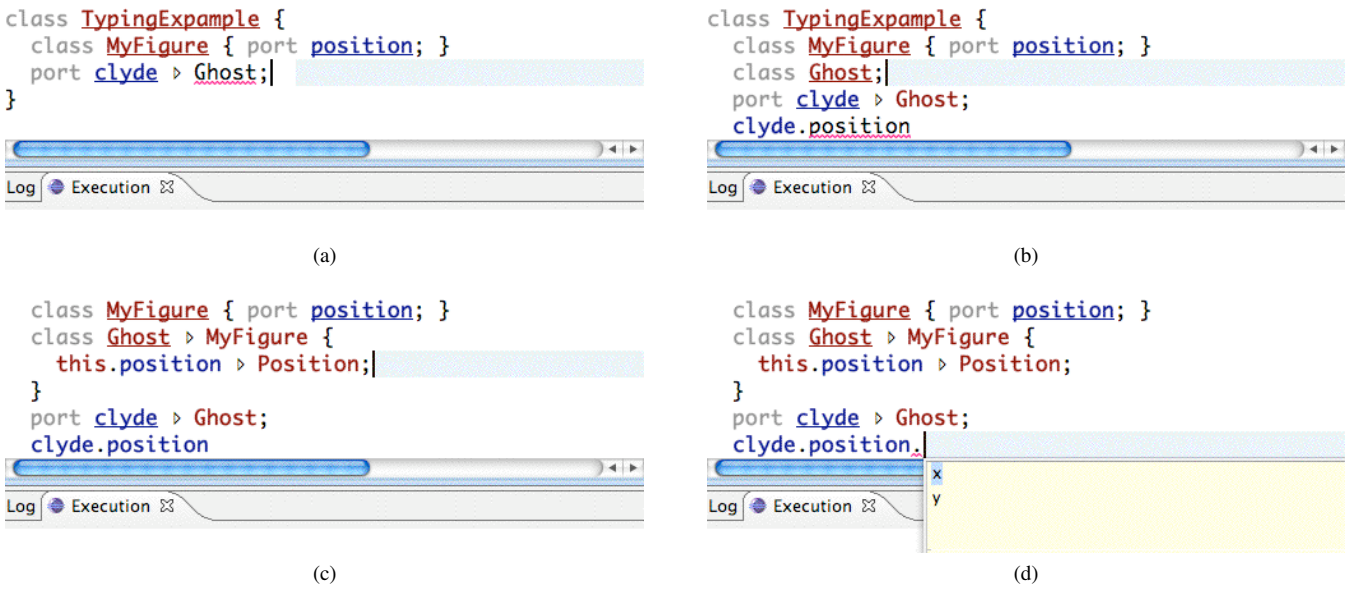


Figure 3: An example of how type inference is used to provide semantic feedback in the SuperGlue programming environment.

can occur during the execution phase include ambiguously connected signals, unconnected primitive signals, and custom errors that are detected in code behind, such as divide by zero. Unlike errors in the previous phases, which can be reported against the program's source code, errors in the execution phase are reported against signal values and are recorded in a separate view. Beyond reporting errors, our prototype does not currently provide facilities for debugging why errors have occurred, which is left to future work. After an error is reported, a default value is used if necessary to allow program execution to continue.

For code behind to play well in SuperGlue's live programming environment, it must report errors on signal values rather than throw exceptions, which would cause the program to stop executing. Code behind should also fall back on default behavior that indicates errors in the program's execution, if possible. For example, the figure draw routine we have coded to draw 2D shapes will report an error and draw a figure as a rectangle if the figure is not a concrete shape. Such error recovery allows the programmer to ignore errors as convenient and focus on other parts of the program.

The frame-expanded movie in Figure 4, which demonstrates live programming execution, is narrated as follows:

(a) `pacMan`'s position is bound to the `keyboard` of the canvas it is displayed in. So that `pacMan`'s orientation corresponds to the direction it is moving in, its `rotation` signal is bound to its `position`. `pacMan` then moves as expected in a video game.

(b) Ghosts for the game are created by defining a `ghosts` signal that extends the `Duplicator` class. Initially two ghosts are created, which are drawn as ghost shapes on the canvas by extending the `Figure` and `Ghost` classes. Because both ghosts occupy the same default position, only one ghost is visible.

(c) The `Ghost` class is defined as an open extension to the `ghosts.get` class. In this class, the `position` and `fill` signals of each ghost extend the `Random` class, causing each ghost to have a random color and motion.

(d) Finally, the number of ghosts is changed from two to eight, causing six more ghosts to appear after around three seconds of edit latency.

## 4. Preliminary Experience

Although our SuperGlue prototype is still immature, it can already be used to write complete programs. Previous work [20] demonstrates how a previous prototype was used to build rich user interfaces with complicated widgets such as tables and trees. This section describes our experience with implementing a simple game in SuperGlue using a library that wraps Java2D components. The game is a simplified version of PacMan with the following features:

– The game consists of a player controlled `pacMan` protagonist and multiple computer controlled `ghost` antagonists on a canvas.

– Normally, if `pacMan` is touched by a ghost, it is transported to the northwest corner of the canvas.

– In the middle of the canvas is a `powerPellet`. If `pacMan` eats/touches the `powerPellet`, the `powerPellet` disappears for some amount of time. During this time, if

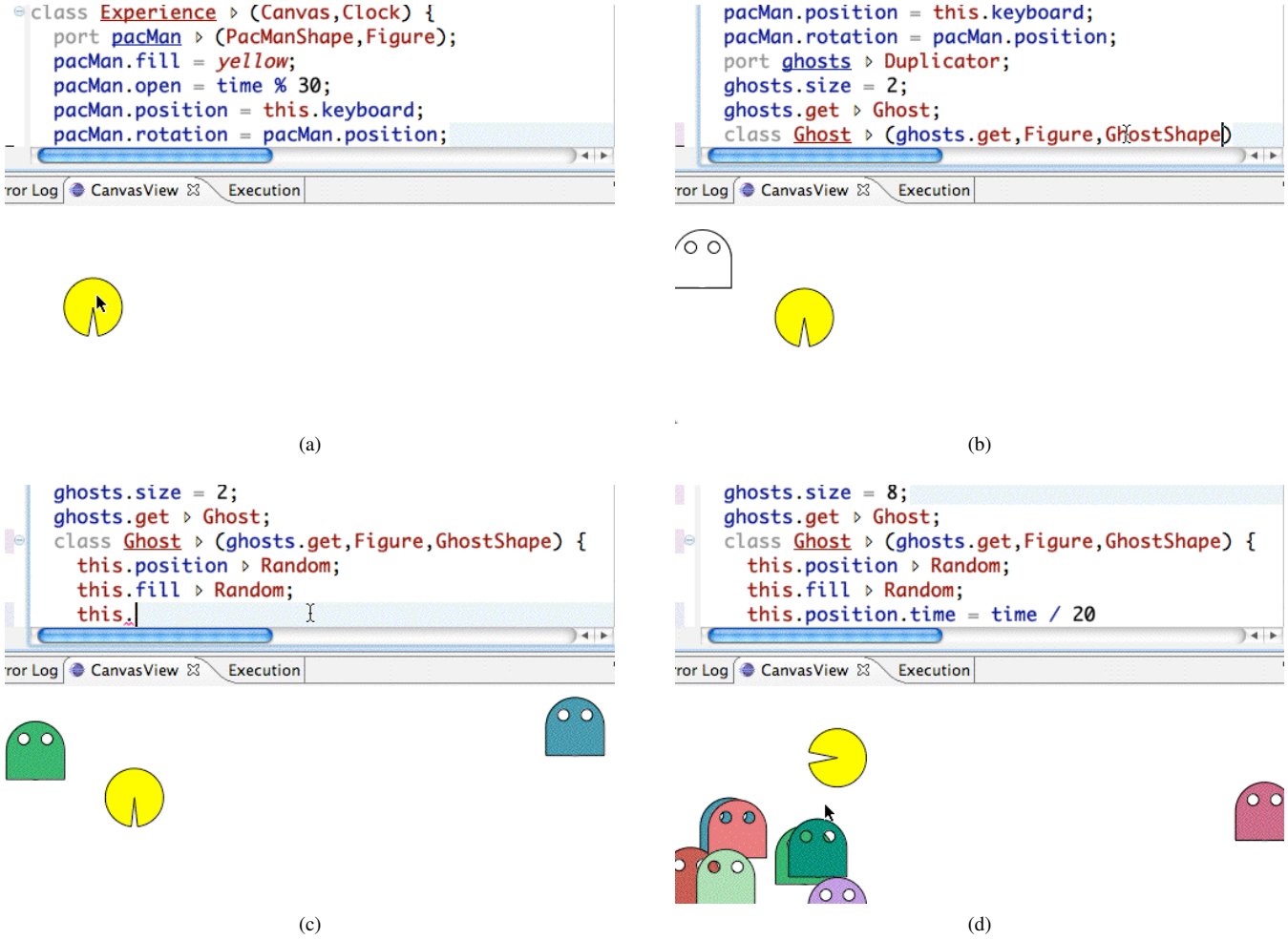


Figure 4: An example of how execution occurs during live editing in the SuperGlue programming environment.

`pacMan` touches a ghost, the ghost is transported to the northwest corner of the canvas.

Aspects of this game have been used in the examples of this paper. We divide our discussion into three parts: code complexity and design (Section 4.1), live programming experience (Section 4.2), and performance (Section 4.3).

### 4.1 Code Complexity and Design

The code for our game can be divided into three categories: 58 lines of game-specific SuperGlue code that is listed in Figure 5, 225 lines of library SuperGlue code, and 5,750 lines of library-specific Scala code. No game-specific Scala code was required, and we do not expect SuperGlue programs to require custom Java or Scala code. To be complete in our discussion, our prototype is implemented in 23,000 lines of Scala code. Our prototype also depends on an IDE framework, under in-house development at the EPFL, that handles most scanning, parsing, and Eclipse-based IDE services, which consists of 28,000 lines of Scala code.

A large portion of the code in our animation library wraps Java2D (2,600 lines of Scala code). This animation library supports the rectangle and ellipse-like arc (Pie) shapes, Affine transforms (scaling, rotation, translation), and shape-area operations (union, intersection, subtraction). On top of this functionality, we can derive more complicated shapes; e.g., the `GhostShape` in Figure 5 is a union of an ellipse and translated article, with two small ellipses cut out to form its eyes. Our Java2D wrapper also supports fill and line colors, keyboard input, and rudimentary collision detection. Many Java2D features have not yet been wrapped and a more complete wrapping will require more code. The other Scala parts of the library are devoted to arithmetic, timers, interpolation, random number generation, and number cells that can be updated. Number cells are especially important because they are currently the only form of state that our library supports. Other library classes are built on top of these classes using SuperGlue code. For example, a `PositionCell` class composes two number cell classes and extends the

```
class GhostShape ▷ Shape {
  port gs ▷ Shape;
  gs = (Pie + Rectangle.scale(height = 0.495).translate(y = 25) -
    Eye.translate(x = 10) - Eye.translate(x = 30));
  this.shape = gs.shape;
  class Eye ▷ Shape;
  Eye.shape = Pie.scale(width = 0.2, height = 0.2).translate(y = 10).shape;
}
class PacManShape ▷ Shape {
  port gs ▷ Shape;
  gs = Pie(start = open / 2, extent = 360 - (open)).rotate(by = rotation.by);
  this.shape = gs.shape;
  port open ▷ Number;
  open = 60;
  port rotation ▷ Rotation;
}
class PowerPellet ▷ (Shape,Timed) {
  port pie ▷ Rectangle(arc.width = 10, arc.height = 10);
  this.shape = pie.scale(width = 0.3, height = 0.3).rotate(by = (time * 4)).shape;
  if (this ▷ Canvas.Figure) {
    this.position.x = 500;
    this.position.y = 100;
    this.fill = cyan;
    this.enable = invisible.on.!.result;
  }
  port invisible ▷ Alarm;
  invisible.duration = 40;
}
class Execution ▷ (Canvas,Clock) {
  this.period = 100;
  port pacMan ▷ (PacManShape,Figure,Entity);
  pacMan.fill = yellow;
  pacMan.open = time % 30;
  pacMan.position = keyboard;
  pacMan.rotation = pacMan.position;
  port ghosts ▷ Duplicator;
  ghosts.size = 2;
  ghosts.get ▷ Ghost;
  class GhostHit;
  class Ghost ▷ (ghosts.get,GhostShape,Figure,Entity) {
    position ▷ Random;
    position.time = time / 20;
    fill ▷ Random;
    hit ▷ GhostHit;
    position.step = 1.9;
  }
  class Pellet ▷ (PowerPellet,Figure) {
    if (overlaps = pacMan.hit)
      invisible.enable = true;
  }
  if (pacMan.overlaps ▷ GhostHit) {
    pacMan.reset_pos = pellet.invisible.on.!.result;
    pacMan.fill ▷ red;
  }
  if (Ghost.overlaps = pacMan.hit)
      Ghost.reset_pos = pellet.invisible.on;
  if (pellet.invisible.on = true) Ghost.fill.color = blue.color;
  class Entity ▷ Figure {
    position ▷ PositionCell;
    port reset_x ▷ position.p_x.set(to = 0,enable = reset_pos);
    port reset_y ▷ position.p_y.set(to = 0,enable = reset_pos);
    port reset_pos ▷ Boolean;
    reset_pos = false;
  }
  port pellet ▷ Pellet;
}
program ▷ Execution;
```

Figure 5: The SuperGlue code for a PacMan-like game; this code has been copied directly from the running version of the game.

`Position` class to provide a mutable position, while the `PositionByKeyboard` class builds on `PositionCell` to update the position according to keyboard input.

Our library leverages rule-based abstractions to create default connections when possible, which reduces the number of explicit connections that must be expressed per program. For example, many library classes depend on timers: number cells use time signals to control how often they are incremented (velocity), random number generators use time signals to control how often they generate new random numbers, and interpolation transforms a time signal into a wave. These classes extend the `Timed` class whose `time` signal is connected by default to the `time` signal of a containing `Clock` signal. By having the `Game` class extend the `Clock` class, any time dependent behavior that is defined inside the `Game` class is by default connected from the game's clock.

We have found mutual extensions (Section 2) and adapter mixins (Section 2.2) to be very useful and repeatedly applicable in the design of our animation library. The `PositionByKeyboard` adapter mixin has already been mentioned in Section 2.2. Additionally, the `PositionByRandom` class adapts the `Position` and `Random` classes to move figures in random patterns, the `ColorByRandom` class adapts the `Color` and `Random` classes to generate random colors, while the `RotationByPositionCell` class adapts the `Rotation` and `PositionCell` classes to compute the rotation of a shape according to the direction it is moving in. Mutual extensions allow us to create rich versions of classes that are only applied in restricted contexts. For example, numbers only support basic arithmetic operations by default. However, when used in the context of a `time` signal, it is often useful that numbers (including literals) also support interpolation and random number generation operators. By defining a `Number` extended `RichNumber` class inside the `Clock` class, programmers can write "`0 <> 360`" to express a value that oscillates between `0` and `360` according to the containing `Clock` signal, or "`0 # 256`" to express a random value between `0` and `256` that also changes according to the containing `time` signal. Type inference computes types that ensures these operators are only accessible on numbers (including literals) that are used in the body of a class that extends the `Clock` class.

Given the design of our animation library, the game-specific code in Figure 5 is very concise and limited to configuration tasks, e.g., how many ghosts, establishing core game relationships, e.g., that `pacMan` is controlled by the keyboard, and core game logic; e.g., reseting `pacMan` or a ghost's position when they collide. All adaptation code between non game-specific classes has been counted as part of the library. Admittedly, programs might require adaptations between library classes that do not already exist, but these adaptations can be published or at least added to the developers own library for reuse in future programs.

### 4.2 Live Programming

The live programming experience is difficult to describe on paper and is best seen in action in the electronic version of this paper. However, edit latency can measurably affect the live programming experience. Many edits used in the construction of our game do not incur noticeable edit latency. However, as discussed in Section 3, there are situations where noticeable and sometimes very long pauses can occur; e.g., by causing many signals to extend a new class or creating many different signals at once. One informal benchmark that we use in our game is based on ghost creation latency: how long does it take to duplicate $N$ ghosts? The good news: the edit latency of ghost creation is linear–creating $2N$ ghosts takes only twice as long as creating $N$ ghosts. The bad news: it take around 500 milliseconds to duplicate one ghost. The ugly news: creating ten ghosts will waste five seconds of the programmer's life. As described in Section 2.3, the problem is in the way our prototype deals with signal initialization, which we plan to fix in the future through a language change. Significant edit latency can also occur when type inference and circuit caches are cold. For example, when a ghost is first created during a game, around two seconds are needed to build the ghost's circuits. Fortunately, all future ghosts obtain these circuits from a cache.

Another part of the live programming experience to consider is error detection and debugging. Our current prototype only supports the reporting of errors, and does not provide any support for debugging beyond providing responsive feedback from the executing program. The value of responsive feedback as a debugging aid cannot be understated: being able to see at one moment on one screen how an edit affects the program's execution allows the programmer to very quickly determine if the edit was good or not. Errors often do not accumulate as they would if the programmer waited until after many edits to test the program. Additionally, because they lack control flow, programs have simpler execution models that do not involve call stacks, instruction stepping, and so on. However, because of SuperGlue's support for object-oriented dispatch of rules, code is substantially more abstract than execution; i.e., execution errors need not be directly related to one piece of code. For example, if a signal is connected ambiguously, the involved rules could be scattered throughout the program.

The immediate feedback of live programming eliminates much of the need to separately debug programs. This elimination should in theory speed up the programming process, but we cannot measure this in our current prototype. Given sufficient error recovery and edit latency characteristics, live programming feedback should always be available to programmers. However, many users often disable feedback in their IDEs or word processors because they are distracted by transient syntax, type, spelling, or grammar errors. In many cases, users are overwhelmed by useless feedback that results from poor error recovery. Part of the distraction is re-

lated to users who prefer little or no feedback; e.g., they use Emacs or notepad instead of Eclipse. Beyond habit and error recovery, good feedback must be tuned so that is continuously available discreetly in the background without distracting the user when its not needed. Ensuring and validating that feedback is engineered in this way is left to future work.

### 4.3 Performance

More than most dynamic languages, SuperGlue eschews performance in favor of being higher-level, more flexible, and supporting live programming. Although edit latency is important to live programming, overall program performance is less so. However, there is the danger that SuperGlue could be too slow for many kinds of applications. One area where our prototype's performance could be improved is in application start-up. Because of our prototype's reliance on incremental work list processing, much work is done and thrown away because many nodes are processed before their dependencies are "ready." For this reason, on start up, our game requires around 32 seconds to start-up: two seconds for parsing, ten seconds for type inference, and 20 seconds to perform initial execution processing. Our focus on edit latency, which work lists address, has led us to poor batch performance, and future work should focus on fixing this.

Another way of measuring performance in our game is through frame rate. For our game, we measure frame rate as related to ghost count: around 100 frames per second is possible for `pacMan` and a couple of ghosts, ten frames per second with ten ghosts, and five frames per second with twenty ghosts. Although such performance might be acceptable for a simple game, our implementation needs better performance so that SuperGlue can be used more widely.

SuperGlue's reactive data-flow model does not support direct compilation. Instead, a SuperGlue program executes as a connection graph that resembles a scene graph in a 3D program. As with scene graphs, the connection graph of a SuperGlue can conceivably undergo high-level dynamic optimizations that speed up how programs execute, especially if domain specific knowledge can be considered. As SuperGlue is only acting as a middle man between Java-implemented components, a compiler could also bind these components directly together by inlining the component Java code with Java code generated from SuperGlue code. Inlining would allow SuperGlue performance to approach Java performance levels. However, to preserve live programming support, such inlining must be retractable. Future work will explore how performance can be improved through dynamic graph optimizations and compilation.

## 5. Related Work

SuperGlue is strongly influenced by Self [29]. Every artifact of a Self program is an object whose every aspect, including what they extend, can be updated at run-time. Self supports no abstractions other than objects: class or method constructions are expressible as objects. SuperGlue is more conventional: behavior is expressed through distinct class constructs. Self supports code hot swapping and responsive live programming in Morphic [18] through polling and a graphical "meta menu" that can directly edit specific objects [30]. However, live programming does not apply to Self source code, and direct changes to objects are not copied back into the code of the program. In SuperGlue, all program behavior is derived from code that can undergo live programming.

Because dynamic inheritance in Self occurs through imperative slot assignment, the type of an object can change in any way at any time without the benefit of static typing. Dynamic object inheritance is supported with static typing in Cecil [6, 7] through declarative predicates, which are also used in SuperGlue. However in SuperGlue, reactive signals make predicated class extensions more responsive: because predicates refer to signals, when a signal extends or unextends a class is known immediately. In Cecil, program behavior cannot responsively react to changes in class extensions because predicates can only be checked on demand.

SuperGlue differs from existing dynamic languages such as Lisp, Smalltalk, or Erlang in its support for live programming and declarative programming model. Of these languages, Erlang [1]'s robust support for fault tolerance and hot swapping comes closest to live programming. Like SuperGlue, Erlang processes communicate explicitly via messages and not implicitly through shared pointers. As a result, a process can be restarted without de-stabilizing the entire program. SuperGlue supports more responsive live programming through signals, which unlike messages encapsulate state change. Also, SuperGlue supports the hot swapping of expressions rather than larger-grained processes.

One of the main contributions of this paper is showing how visual language liveness can be realized in a textual language. Ever since SketchPad [27], visual languages have emphasized simple interactive programming models that accommodate end users or novices. SuperGlue is designed for programmers who typically use dynamic languages, and end-user programming is not a goal. Many modern visual languages solve the scaling up problem in ways that do not degrade live programming; e.g., assemblies in Fabrik [15] are reused as objects, and data types in Forms/3 [3] can be elegantly specified as generic spreadsheet forms. SuperGlue solves the scaling-up problem using conventional class constructs that are more suited to the naming capabilities of textual languages. AgentSheets [24] is a visual spreadsheet language that is based on graphical rewrite rules and implicit component assembly through spatial adjacency. AgentSheets demonstrates how live languages can be based on rewrite rules, and how many useful visual concepts (spatial adjacency)do not transfer easily to text.

Many existing textual languages are based on reactive data-flow programming models. Signal [2], Lucid [32], and

LUSTRE [5] are textual reactive data-flow languages that emphasize the formal verification of concurrency in real-time systems. As a result, they are not designed to support the rapid development capabilities of a dynamic or live language. SuperGlue is very similar to functional-reactive languages such as Fran [10], Yampa [14], and FrTime [8]. Such languages augment the functional paradigm with reactive signals; e.g., the result of a function call $f(x)$ forms a signal that changes whenever the $x$ signal changes. In contrast, SuperGlue is based on rules, which are less flexible but have the advantage that they do not need to be applied explicitly. It remains an open question whether functional-reactive languages can support responsive live programming with recursive applicative programming models.

Subtext [9] bridges the gap between code and execution by allowing programmers to edit code that directly represents program executions that lacks textual names. Because edits operate directly on execution, Subtext supports responsive live programming. On the other hand, Subtext supports scaling through an unproven "copy-flow" rather than data-flow. Code and execution remain separate in SuperGlue and their gap is only bridged through live programming.

Flogo II [13] is another textual live language that focuses on end-user robot programming. Flogo II supports both reactive data-flow and procedural code, where the latter does not support responsive live programming. Given its focus on end users, Flogo II scaling constructs (Erlang-like processes) are more concrete and do not scale as well as SuperGlue's classes. An intriguing idea in Flogo II's is *live text*: the state of an executing program is presented as graphical annotations in the program's textual source code. For example, statements are grayed out when they are guarded by a condition that is currently false. SuperGlue does not currently support similar capabilities.

SuperGlue's static type system relies on type inference and dependent typing to provide programmers with semantic feedback while avoiding verbose type annotations. Scala [22] is a statically typed object-oriented language that utilizes dependent typing and type inference to reduce verbosity, and therefore is a viable alternative to dynamic languages. SuperGlue's type system is influenced by Scala's; however, dynamic typing is still used to validate connections. An alternative way of obtaining semantic feedback in a dynamic language is through the use of heuristics and program analysis; e.g., Olin Shivers describes various analyses for inferring types in Scheme [26]. However, global program analysis is a hard problem and so is often not very accurate or efficient. Type inference in SuperGlue relies on a very simple program analysis that remains accurate while being both flow and context insensitive.

## 6. Conclusions and Future Work

Programmer-centric dynamic languages have been around for awhile: Lisp and Smalltalk are respectively more than 40 and 25 years old. New dynamic languages, such as Ruby [19] and Python [31], still strongly resemble Lisp and Smalltalk in their feature sets. Now is the time to explore languages that support higher-level live programming with responsive semantic feedback. This paper demonstrates the viability of both the design and implementation of such a live language that also supports textual abstraction. We have shown that a live language could be based on a simple declarative reactive data-flow model that is augmented with predicated object-oriented inheritance and dispatch. We believe that a live text languages could be based on other simple computational models such as constraints, first-order logic, and rewrite rules, where live visual languages already exist.

Future work will focus on improving SuperGlue's design and implementation in the following areas:

- As described at the end of Section 2.3, SuperGlue's support for class extension and signal initialization will be changed to enhance performance.
- Our prototype does not currently provide enough debugging support. We plan to allows programmers to inspect, reason about errors in, and edit specific signals.
- We must improve on latency and performance if the widespread use of SuperGlue is to be viable. Building a live programming environment is an undocumented black art that requires a complete re-think of how code is scanned, parsed, typed, and executed. The prototype described in this paper has already undergone some tuning and there are still many improvements that we can make.
- We should explore the incorporation of other visual and interactive features into the SuperGlue's programming environment. Features to be explored include support for direct manipulation, where specific signal values can be changed directly, and live text, where source code is annotated with execution details.

Finally, we plan to support SuperGlue with rich libraries that will make it a good platform for building interactive programs in an interactive way.

## Acknowledgments

We thank Adriaan Moors, Gilles Dubochet, Lex Spoon, and the anonymous reviewers for comments on drafts of this paper. This work was partially supported by a grant from the European Framework 6 PalCom project.

## References

[1] J. L. Armstrong and R. Virding. An experimental telephony switching language. In *Proc. of Interantation Switching Symposium*, May 1991.

[2] A. Benveniste, P. L. Geurnic, and C. Jacquemot. Synchronous programming with events and relations: the Signal language and its semantics. In *Science of Computer Programming*, 1991.

[3] M. Burnett, J. Atwood, R. Walpole, H. Gottfried, J. Reichwein, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. In *Journal of Functional Programming*, pages 155–206, Mar. 2001.

[4] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang, and P. van Zee. Scaling up visual programming languages. In *IEEE Computer*, pages 45–54, Mar. 1995.

[5] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *Proc. of POPL*, 1987.

[6] C. Chambers. Object-oriented multi-methods in Cecil. In *Proc. of ECOOP*, pages 33–56, June 1992.

[7] C. Chambers. Predicate classes. In *Proc. of ECOOP*, pages 268–296, July 1993.

[8] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proc. of of ESOP*, 2006.

[9] J. Edwards. Subtext: Uncovering the simplicity of program. In *Proc. of OOPSLA Onward*, 2005.

[10] C. Elliott and P. Hudak. Functional reactive animation. In *Proc. of ICFP*, volume 32 (8) of *SIGPLAN Notices*, pages 263–273. ACM, 1997.

[11] R. W. Floyd. Syntactic analysis and operator precedence. In *Journal of the ACM*, volume 10 (3), pages 316–333, 1963.

[12] A. Goldberg and D. Robson. *SmallTalk-80: The Language and its Implementation*. Addison Wesley, Boston, MA, USA, 1983.

[13] C. M. Hancock. *Real-time Programming and the Big Ideas of Computational Literacy*. PhD thesis, Massachusetts Institute of Technology, Sept. 2003.

[14] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer, 2002.

[15] D. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph, and K. Doyle. Fabrik, a visual programming environment. In *Proc. of OOPSLA*, pages 176–190, Nov. 1988.

[16] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination queueing. *SIAM Journal for Applied Mathemetics*, 14(6):1390–1410, Nov. 1966.

[17] P. O. Latour. *Quartz Composer*. Apple Computer, 2005. http://developer.apple.com/graphicsimaging/quartz/-quartzcomposer.html.

[18] J. H. Maloney and R. B. Smith. Directness and liveness in the Morphic user interface construction environment. In *ACM Symposium on User Interface Software and Technology*, pages 21–28. ACM, 1995.

[19] Y. Matsumoto. *Ruby: Programmers' Best Friend*. http://www-.ruby-lang.org/en/.

[20] S. McDirmid and W. C. Hsieh. SuperGlue: Component programming with object-oriented signals. In *Proc. of ECOOP*, June 2006.

[21] National Instruments Corporation. *LabVIEW User Manual*, 1990.

[22] M. Odersky and et. al. The Scala language specification. Technical report, EPFL, Lausanne, Switzerland, 2007. http://scala.epfl.ch.

[23] M. Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Comm. of the ACM*, pages 33–44, June 1995.

[24] A. Repenning. AgentSheets: an interactive simulation environment with end-user programmable agents. In *Proc. of Interaction*, 2000.

[25] E. Sandewall. Programming in an interactive environment: the lisp experience. *Computing Surveys*, 10(1), Mar. 1978.

[26] O. Shivers. The semantics of Scheme control-flow analysis. In *Proc. of PEMP*, pages 190–198, June 1991.

[27] I. B. Sutherland. SKETCHPAD, a man-machine graphical communication system. In *Proc. of the Spring Joint Computer Conference*, pages 329–346, 1963.

[28] T. Teitelbaum and T. W. Reps. The Cornell program synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

[29] D. Ungar and R. B. Smith. Self: the power of simplicity. In *Proc. of OOPSLA*, pages 227–242, Oct. 1987.

[30] D. Ungar and R. B. Smith. Programming as an experience: the inspiration for Self. In *Proc. of ECOOP*, pages 227–242, Aug. 1995.

[31] G. van Rossum and F. L. Drake. *The Python Language Reference Manual*, Sept. 2003.

[32] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.