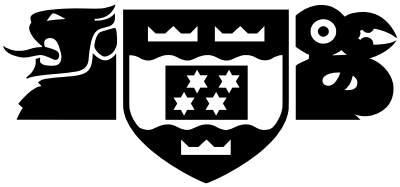
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Engineering and Computer Science
Te Kura Matai Pukaha, Purorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Lively JavaScript

Michael Homer

Supervisor: James Noble

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

Abstract

In this project we evaluate the use of JavaScript as a general-purpose programming language, using the Lively framework. We implemented several applications within the Lively system to seek insights into both the language and the programming environment, exploring claims and techniques found in the literature. We examine the prototype-based paradigm and the benefits and drawbacks of interactive programming for application development.

Contents

Chapter 1

Introduction

JavaScript, or ECMAScript, is often used as a secondary programming language embedded in other systems, but has rarely been used as a language in its own right [10, 11]. Because JavaScript has primarily been used as a "glue" language in the past, and was built for this purpose at Netscape, it has had little attention as a general-purpose language until recently. Using JavaScript for top-level development uncovers areas of the language that do not appear in its most common usage as part of web pages, and may show strengths and weaknesses of its approach [18, 26].

Lively for Qt [2] is an environment in which applications can be written in JavaScript and run alongside each other in a visual workspace. These applications are cross-platform and dynamic, and can be run both via the web and on a client machine. Many applications can run alongside one another in a "world" and can be manipulated, added, and removed by the end user. Each application executes in a shared namespace and is able to see the applications around it.

In this project we evaluated the suitability of the JavaScript language and the Lively platform for general development. We sought an overview of the capabilities of the system, and to establish for which tasks it is suitable or unsuitable. We also explored the advantages and disadvantages to software development that resulted from such a highly dynamic programming language and environment.

We carried out the project by implementing several applications within the Lively framework. We chose several relatively small applications to implement, with the aim that they be independently useful and address different aspects of the system. We also wanted to experiment with the reflexive nature of Lively and "interactive programming", so we implemented some programming tools which we were then able to use in developing other applications.

Because the Lively system allows many applications to run alongside one another in a shared space, we explored the idea of linking multiple smaller applications together within the environment. In this way our applications are able to use the functionality provided by one another as a part of their own operation. This facility is not available in many other systems and is one of the novel aspects of Lively, so our applications provide the maximal opportunity for connection to each other. We extended this connectivity to modifying built-in applications, developing a standard interface to allow unrelated software to communicate with minimal coupling.

We chose our target applications in order to test the claims of the literature, and in response to the successes and obstacles we found in implementing earlier applications. In particular, we wanted to explore the claimed high interactivity in the Lively system, and what advantages or disadvantages it provides to the programmer. We also wanted to test how easily an existing program written in JavaScript could be extended with new function-

ality using the dynamic and prototype-based features of the language, as this is a key claim of advocates of prototype-based and dynamic languages [24, 29, 18].
Our objectives in conducting this research project were:

- To evaluate JavaScript as a language for general-purpose development. We considered the language from the perspective of development speed, correctness, and maintainability, and made comparisons to some other languages.
- To evaluate the Lively system as an environment for application development. We examined the system from the perspective of interactivity and accessibility, and investigated what unique opportunities it provides for connecting applications together.
- To experiment with interactive programming, and to utilise the reflexivity and dynamism of the programming environment given by Lively and JavaScript to produce an interactive tool for examining and developing applications in the system.

Contributions

This project makes the following contributions:

- Tested the claim of rapid prototyping when using JavaScript for development.
- Evaluated Qt and the QtScript bindings for development.
- Showed practical situations in which the `that` pointer is a useful addition to a programming environment.
- Demonstrated that single-purpose GUI applications can be usefully connected to each other.
- Built a reflexive introspection and interactive development tool for the Lively system, and showed that interactive programming is a useful approach.
- Constructed a partial aspect-orientation system for JavaScript.

This report is structured as follows:

Chapter 2 describes the distinguishing features of the JavaScript language and its history, and the Lively environment. We also discuss some related work that forms the background for Lively.

Chapter 3 discusses the approach we have taken to the project, and why we chose this approach over other possibilities. We describe the applications we built from a user perspective.

Chapter 4 contains implementation details of the applications, and observations on JavaScript and Lively we made while writing them. We also discuss the partial aspect-oriented programming system built as part of our introspection tool.

Chapter 5 contains our judgements about JavaScript and Lively from implementing those applications.

Chapter 6 has our conclusions.

Chapter 2

Background and related work

We use JavaScript for all development in this project and build on past research into the language. Our applications run under the Lively environment. In this chapter we describe the background of both the language and the environment, and the JavaScript bindings to the Qt toolkit that we use.

2.1 JavaScript

JavaScript is a prototype-based dynamically-typed object-oriented imperative programming language developed at Netscape to be embedded into its browser and used in web pages. It was standardised as ECMAScript [10], and all non-Netscape implementations are technically of ECMAScript.

The name "JavaScript" is a trademark of Oracle Corporation (as the successor to Sun Microsystems), and is used under licence by Netscape and its successors to bless their reference implementation of the JavaScript language under a 1995 cross-marketing agreement between Sun and Netscape. The Mozilla Foundation's version is the only JavaScript, and they periodically release new versions independently of the standardisation process. Other implementations are of ECMAScript, including the QtScript used in this project. The language is conventionally known as JavaScript, including in the IETF MIME type[13], and we follow this convention here.

In a prototype-based language objects share behaviour by being based on another "prototype" object, rather than by having a static class distinguished from instance objects [24, 17], so inheritance is from another object in the system instead of from a class. The JavaScript object system is prototype-based, inspired by Self [30, 29, 31], in contrast with the classical approach of almost all other popular object-oriented languages. It is also possible to have a non-classical object-based language without prototypes [20], but discussing this situation is not within the scope of this project.

There are two methods for implementing prototypes: concatenative and referential prototyping [25, 9]. Concatenative prototyping as used in Self involves conceptually copying the prototype at the time of creation, and applying instance-level changes to this new object. With referential prototyping, the object stores a reference to its prototype, and forwards incoming messages that it does not handle itself. In a language with referential prototyping subsequent modifications to the prototype object affect all children, while with concatenative prototyping the entire prototype for a new object is frozen at the time of creation.

JavaScript uses referential prototyping, with a hidden reference to the prototype in each object instance created implicitly on creation. The prototype is assigned as a property on a constructor function, which is in other respects an ordinary function, and the reference to

this object is copied as a side effect of the `new` operator as follows:

```
1  function SomeConstructor(arg1, arg2) {
2    // ... initialise object ...
3  }
4  // { key: val, ... } is a pure object literal
5  SomeConstructor.prototype = {aProperty: 42};
6  var x = new SomeConstructor('hello', 'world');
7  x.aProperty == 42; // true
8  x.aProperty = 1729;
9  var y = new SomeConstructor('java', 'script');
10 y.aProperty == 1729; // false
```

Figure 2.1: Code showing property behaviour of prototypes

Object properties can be looked up in two ways: using the "." operator as above, or as a table lookup in the form `obj[key]`. Any member lookup on an object for a property it does not have itself will be looked up in the prototype. The prototype object can itself have a prototype, creating a chain of inheritance. When an assignment is made to an object property, the value is stored in the target object itself rather than the prototype, even for existing members which were defined only in the prototype.

Methods in JavaScript are simply first-class functions assigned as members of an object. Functions that are called via object dereference, using the "." or "`[]`" operators, have access to a special "`this`" pointer which refers to the receiving object. "`this`" always refers to the object on which the lookup was first performed, and not to the prototype containing the function. It is a run-time error for a function to attempt to access "`this`" when it was not called as a method, and the pointer is not available to other functions defined or called within the method's lexical scope.

A function can also act as a closure. Functions retain access to local variables defined in the enclosing scope when they are later invoked from outside that scope. The function will see any subsequent changes to the values of variables and can update them itself. Functions do **not** close over `this`, so functions needing access to methods or data on the surrounding object require additional bookkeeping to create a viable closure, even when called locally.

Subclasses in this structure are copies of a prototype object (which may be an instance of an existing "class") with some of their properties changed. These are cheaply created and require only a function definition for the constructor [24]. There is no distinction between an object created from a constructor and one that has had its properties modified to match directly, which also makes singleton subclasses trivial. Properties may include both data and functions (methods), and any that are not explicitly set directly on the object will be retrieved from the prototype chain when required.

JavaScript values can be other objects, numbers (double-precision floating-point), strings, booleans, functions, or undefined (a primitive type). Numbers, strings, booleans, and functions are defined primitives, while all other values are objects. JavaScript `Arrays` are objects with array-like methods defined on their prototype. Arrays are treated identically to other objects with the exception of the "length" property, which is always greater than the largest integer key in the array. Arrays are sparse, storing elements in the table in the same way as other members, and are not arrays with contiguous memory in the sense found in other languages. All keys in arrays and other objects are converted to strings using the built-in `toString` method before storage. All properties are mutable and may be set and changed on an object.

While JavaScript is ubiquitous for HTML scripting, it has only recently begun to be used

for more general-purpose tasks outside the browser environment. node.js is a JavaScript-based system for writing applications based on the V8 engine from Chromium [4]. node.js is intended principally for building the server side of web-based applications, but provides a general-purpose environment for event-based development. The Adobe Integrated Runtime is used to build desktop applications using Adobe's JavaScript implementation [6]. As well as these uses the language is deployed in the browser for increasingly complicated web-based applications that stretch the boundaries of the "scripting" categorisation.

In this project we use an implementation of ECMAScript 3. The most recent ECMAScript standard [11] defines a number of extensions to the language, predominantly in the standard library. These extensions are not part of QtScript and are not available for our code. ECMAScript 5 also includes a "strict mode", a variant of the language where the runtime engine is less forgiving of errors that the language otherwise allows to pass for compatibility reasons [11, p4]. The strict mode of the language is optional, and enabled at the level of code objects (programs and functions). We have noted where these changes in the language would be relevant to our analysis.

2.2 Lively

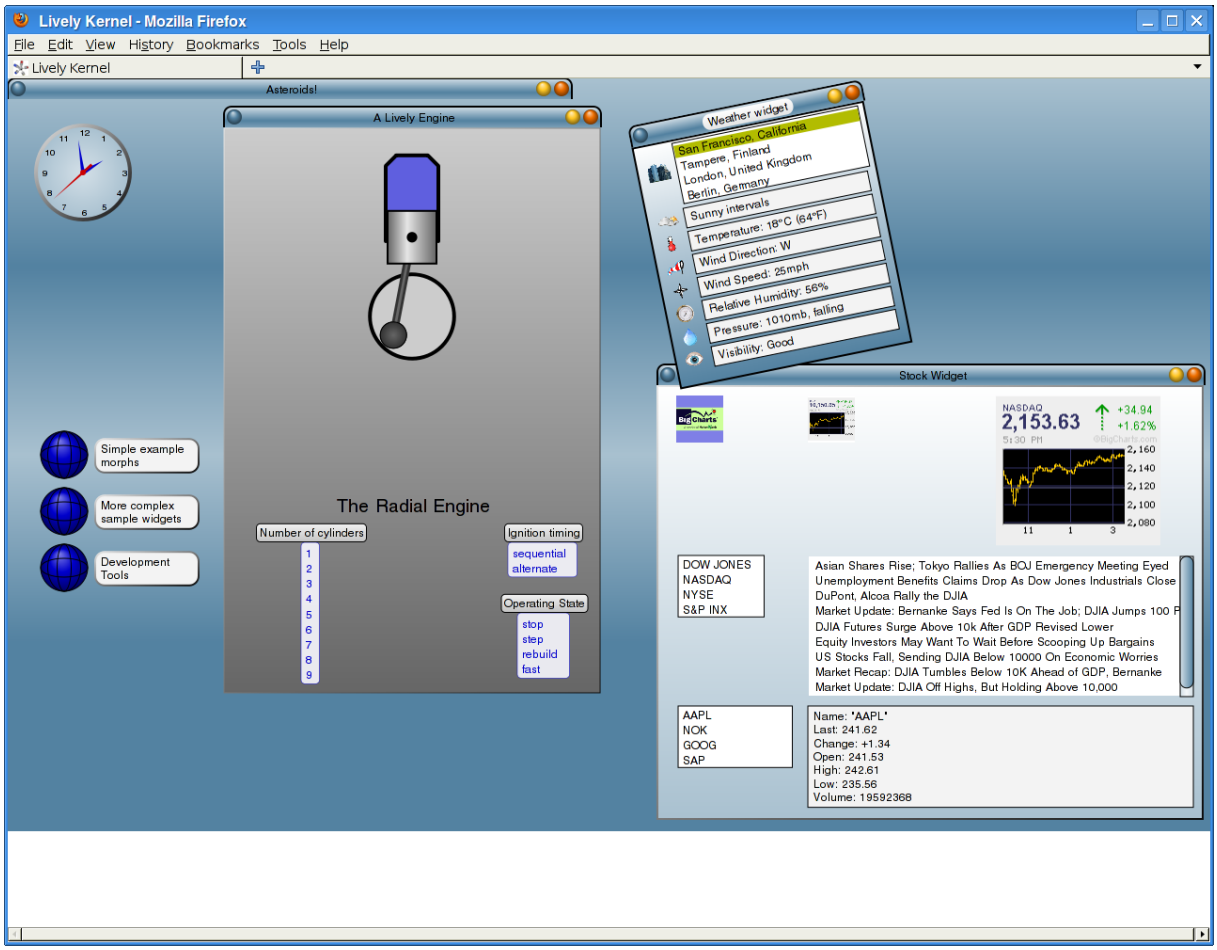


Figure 2.2: The Lively Kernel

The Lively Kernel [3, 15] (figure 2.2) is a programming environment for JavaScript applications. These applications run alongside each other in a visual environment inspired by

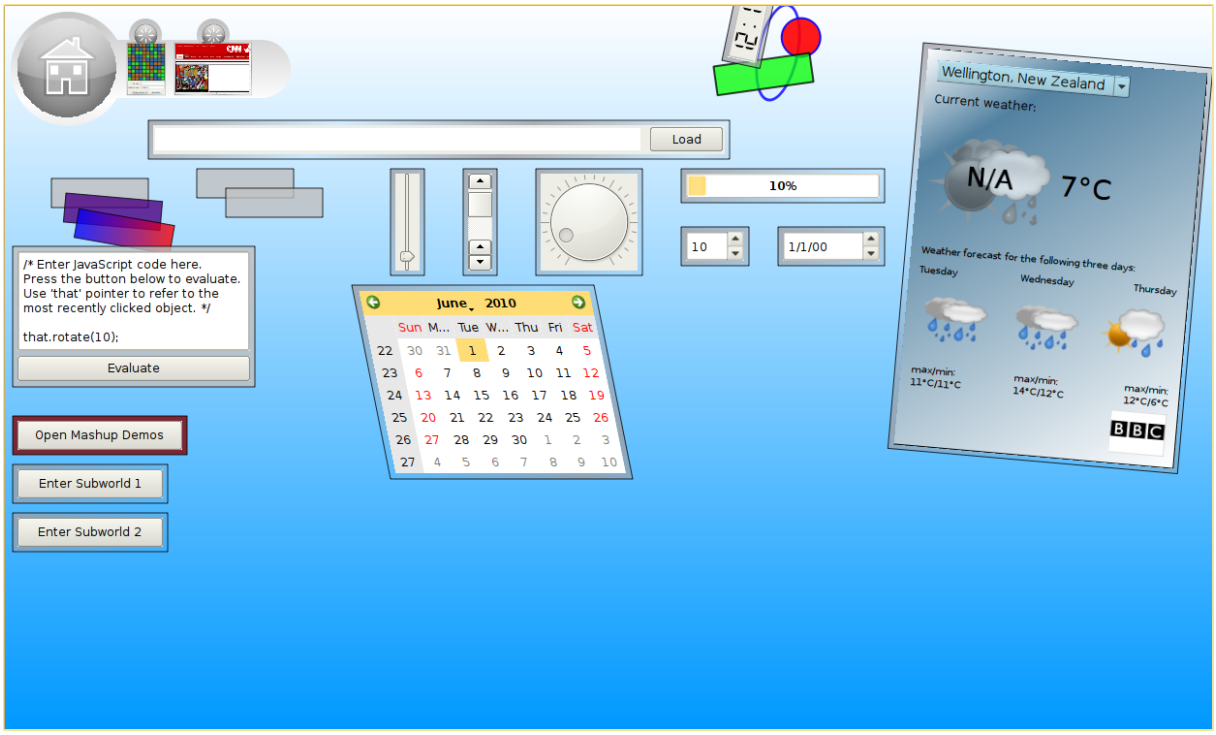


Figure 2.3: The default Lively for Qt environment

the Squeak Smalltalk programming environment [5, 7, 14] and Self before that [30], running within a commodity web browser [28]. The Lively Kernel is a very dynamic and interactive environment, presenting an interesting area for development.

Applications built with Lively are cross-platform, dynamic, and securable, and can be run both via the web and independently on a client machine [26]. Many unrelated applications may run simultaneously in the same Lively "world" (similar to a desktop), and may be manipulated, added, and removed by the user. All applications and Lively itself run inside the same JavaScript interpreter environment. The way that applications may run in the same environment makes for a different pattern of development from the standard desktop, server, or web application, as the applications are neither entirely self-contained nor under shared control.

Lively for Qt [2, 19] (figure 2.3) borrows most of the structure of the Lively Kernel but runs inside the environment of the Qt toolkit [21], including bindings to native widgets [22]. Qt ("cute") is a cross-platform C++ widget toolkit now developed by Nokia, which uses it as part of its mobile development platforms. Qt provides a complete set of user interface widgets, using native widgets where available. QtScript is Nokia's implementation of ECMAScript, and includes bindings for most Qt classes. The goal of the Lively for Qt port is to create cross-platform applications with native appearance everywhere, including mobile environments [19]. Operating with native bindings allows new functionality that is not possible in the browser embedding of the Lively Kernel, in particular filesystem access, and avoids a jarring change of appearance between Lively and native applications. Building applications that can run equally on desktop computers and mobile phones was a goal of both Sun and Nokia, and Lively for Qt aimed to support native appearance rather than HTML [19].

Lively for Qt includes a number of prebuilt applications, both ports of existing applications from the Lively Kernel and new applications to make use of the Qt functionality.

Figure 2.3 shows some of these applications from the default home screen. Figure 2.3 includes several examples of the native user interface widgets that Qt makes available.

Interactive programming is the approach of developing software while it is running, as in Lisp, Self [31], and Squeak [5]. Both the Lively Kernel and Lively for Qt aim to aid interactive programming, providing a built-in tool for the user to execute JavaScript code in the scope of the environment. Also in pursuit of this goal is one of Lively's novel features, the `that` keyword [27]. "`that`" refers to the application currently with the focus. The user may programmatically manipulate and inspect the object in the system with which they are interacting, to help rapid development of new functionality. `that` is in the global scope and available from any code. Aside from the "Evaluator" there is no editor. The Lively Kernel includes an Inspector with some facility for viewing, but not modifying, the methods and properties of a class, but the Inspector is not available in the version of Lively for Qt we used.

2.3 Qt Bindings

Qt provides a complete set of user interface widgets, as well as some other classes to abstract access to the filesystem and other aspects of the host. The native Qt widgets have JavaScript bindings corresponding to the structure of the native C++ bindings. They are namespaced in JavaScript objects to parallel the structure, so `Qt.KeyLeft` in JavaScript corresponds to the `Qt::KeyLeft` constant, and the same for other functions and methods. Top-level widgets are exposed as constructor functions: a QPushButton is created as `new QPushButton("Label")`, and similarly for other widgets and classes.

Qt widgets provide zero or more "signals", named events which the widget can raise in response to user actions. The JavaScript bindings expose signals as properties of the widget object, and callbacks can be bound to the event using the `bind` method:

```
1 var button = new QPushButton("Click me");
2 // 'clicked' is one of the signals provided by QPushButtons.
3 button.clicked.bind(function() { /* do something in response to
      click */ });
```

Figure 2.4: QtScript signal binding example

A signal may have multiple callbacks associated with it, which are triggered in an unspecified order. Some signals do not correspond to user actions, but rather timer or paint events, but all follow the same mechanism for registration and invocation.

Where Qt's C++ interface has overloaded methods or constructors the JavaScript bindings generally provide access to all variants. Although JavaScript does not have overloading, strong typing, or fixed method arity, the script bindings attempt to resolve arguments into suitable native types and dispatch to the correct method. In some cases only a subset of the overloaded versions of a method is available, which we believe to be an artifact of limitations in the bindings' dispatch infrastructure, but the documentation does not provide a definitive answer. Some native classes are also missing entirely: parts may have been omitted by choice or not yet implemented.

Chapter 3

Design and Methodology

Our approach to this project was dictated by pragmatic concerns of flexibility and practicality within the time available. In this chapter we outline the approaches we considered, and describe the details of the method we chose.

3.1 Approach

We considered several possible methods for evaluating the JavaScript language before settling on a suitable method for meeting our objectives. All methods had advantages and drawbacks, but the balance of these traits and practicality within the time available guided our decision. The selected approach needed to allow us to assess both JavaScript and the Lively environment, and also needed to incorporate experimentation with interactive programming, which is one of the claims made about Lively.

Case studies One approach is to examine already-existing case studies to see what information they can provide about JavaScript. These have the potential to provide good insights into the language, but there are few if any such studies available that use JavaScript as a general-purpose language.

Algorithm contests We could examine entries in algorithm contests written in JavaScript, and potentially compare them to entries in other languages. There are many contests and suitable entries to use here, but algorithm contests are not representative of normal usage of a language and could be misleading.

Lab experiments Another possibility is to conduct lab experiments, where the test subjects would write some JavaScript code under observation. A major design constraint on this is finding a suitable test population of programmers with JavaScript knowledge, and evaluating the language based on these observations and feedback may not be reasonable.

Usability tests We could run usability tests on Lively in the same way as lab experiments on JavaScript, with largely the same constraints. It would also be impossible to get an accurate picture of Lively without also evaluating JavaScript, and separating the two would be problematic.

Corpus analysis We could conduct a corpus analysis to determine when the language is being used for scripting and when as a general-purpose language. That classification is fraught, especially when imposed after the fact, and a suitable corpus does not appear to exist.

Development experience In this approach we would write code, implementing some applications in the system to see what we could learn from them. The major drawback of this approach is generalisability; the programming evaluation is limited to the actual programmer's perspective, which may not apply in other circumstances.

3.2 Application development

Our chosen approach was to develop applications. We considered that reflecting on development experience was reasonable to conduct within the scope of the project and should be flexible enough to adapt to new information we discovered while carrying out the project. To this end, we have implemented a number of applications in JavaScript and Lively. We experimented with different techniques to discover what works and what does not, and to make a qualitative analysis of the language and environment. We chose six broadly-defined applications. These are:

- Calculator
  - A simple desktop calculator.
  - A Reverse Polish Notation (RPN) calculator, extending the simpler calculator.
  - A scientific calculator capable of working with functions, also extending the simple calculator.
- A charting application.
- An implementation of Conway's Game of Life for profiling purposes.
- A versioning text editor
- A syntax-highlighting in-process JavaScript code editor
- An introspection and profiling tool.

Figure 3.1 shows of some of these applications in operation. We selected the applications from those that did not already exist as part of the Lively system, focusing on variety of approach and the potential to explore different areas of the system. We also wanted to experiment in interactive programming, as this was one of the claims made about the Lively environment, so the introspection and editing applications were important selections. If there were more time available in the project we would have chosen more applications, but these gave a good balance of aspects of the system: some are an interface around relatively simple code, some have complex algorithms inside, and some aid interactive programming.

In the rest of this chapter we present the applications we developed, from the user's perspective. We reflect on the implementation in the next chapter, and evaluate the JavaScript language and Lively based on our experience in chapter 5.

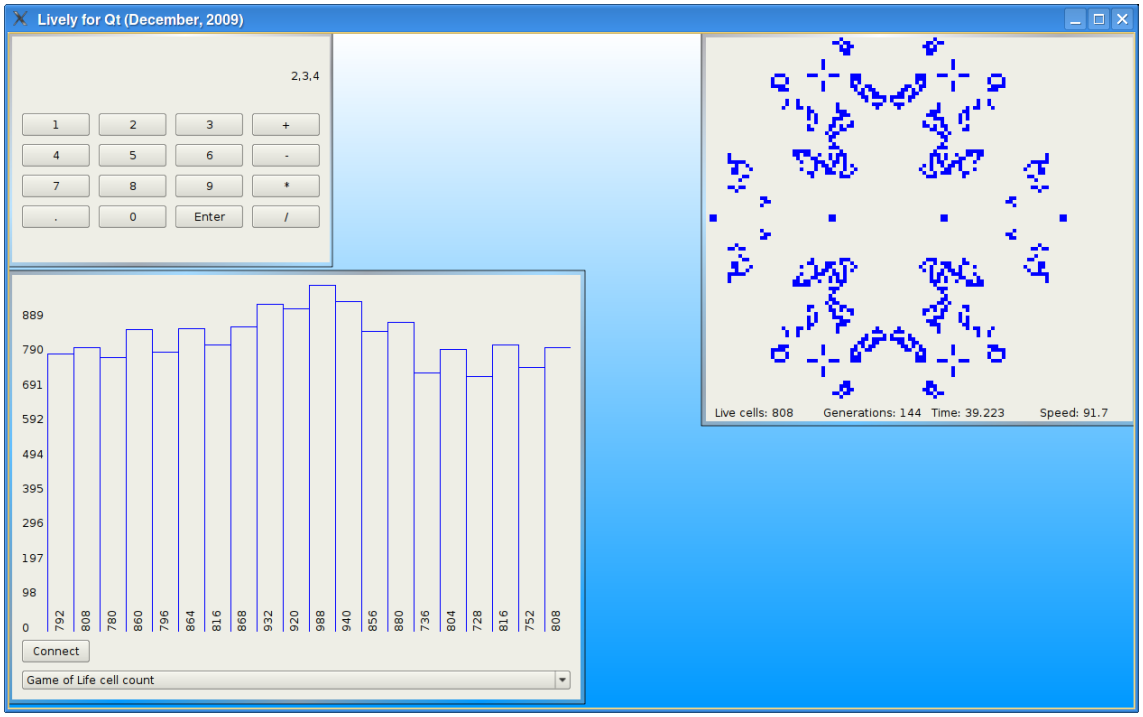


Figure 3.1: RPN calculator, Charter, and Game of Life

3.3 Calculators

We chose to begin by building a simple calculator, as a way to explore the capabilities of the system and learn to use the Qt GUI library. A calculator was a well-defined and relatively simple project, which had room for extension later on. As there was no previously-existing calculator application in Lively for Qt this subproject was also immediately useful.

The simple calculator uses the stock Qt widgets for buttons and text rendering. The visual button layout uses one of the toolkit's built-in layout managers to mimic the structure of a physical calculator. Each button press immediately triggers an action. The simple calculator is only able to perform simple arithmetic calculations with strict left-to-right evaluation order.

Following the simple calculator we built a Reverse Polish Notation calculator by extending the simple calculator, as a first test of the prototype-based inheritance mechanism. The RPN calculator stores operands on a stack, and so has quite different internal functioning than the simple calculator, but still uses an instance of the simple calculator as its prototype.

The scientific calculator evaluates an expression rather than performing operations instantly, and supports including variables in this expression. Like the RPN calculator this application extends the simple calculator, making many more changes to the internal workings but still using the framework of it. The user interface in particular is almost entirely shared apart from some additional controls. This calculator provides a source of continuous data for the charting application. Figure A.1 shows all three calculators in action.

3.4 Charter

A charting application was part of the original plans for the project, but we expanded its scope while the project was underway. This application makes visual plots of data provided from other applications in the system. The Charter is able to plot continuous data (provided

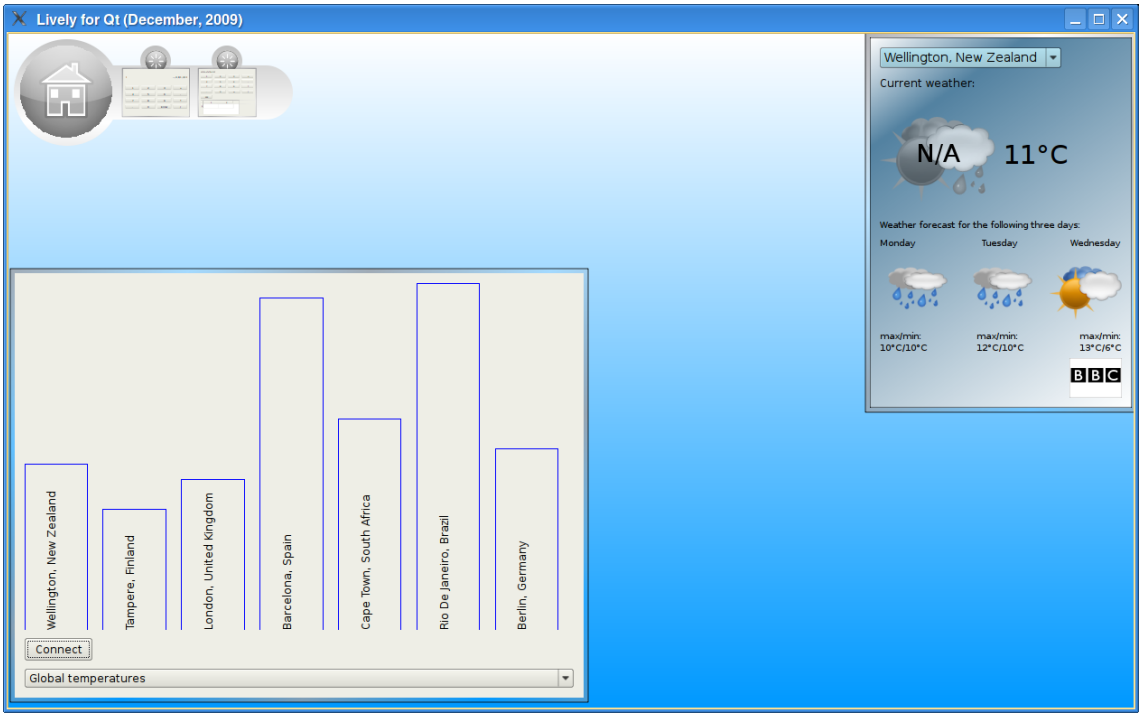


Figure 3.2: Charter displaying data from weather applet

by the scientific calculator, figure A.3), discrete data (provided by the RPN calculator and by the weather applet, figure A.2), and time-series data (provided by the Game of Life, figure A.4). The application accesses this data by way of a standard and flexible interface, so it can potentially connect to any other application in the system. Applications may provide notification that their data has changed, which will automatically trigger a refresh of the visualisation.

3.5 Introspector

One goal of the project was to experiment with the language and environment's interactivity by producing a full profiling, visualisation, and monitoring tool for the Lively environment, to aid debugging and interactive programming. The Introspector examines other applications in the system, recording and displaying the available control-flow information.

Figure 3.3 shows the Introspector examining an instance of the RPN calculator. The user selects an application by clicking the "Watch" button and then selecting the other application, in the same way the Charter finds a data source. The methods of the application are displayed as the blue rectangles in a grid formation, with arrows between them indicating calls from one to the other. Yellow arrows show very recent calls, and will fade back to green over time. The methods automatically rearrange to move related methods (those which call each other) closer together, resulting in a natural partitioning betweeen different areas of functionality in the visualisation. When there are more methods than fit in the display the Introspector can be resized, with the method grid expanding and adding new rows and columns when space allows. The visualisation updates in real time, whenever the outermost method triggered by an external action terminates. The user can watch the effects of their actions as they occur, with the highlighting showing the full chain of method calls they have triggered. This can help to track bugs or document the behaviour of an existing

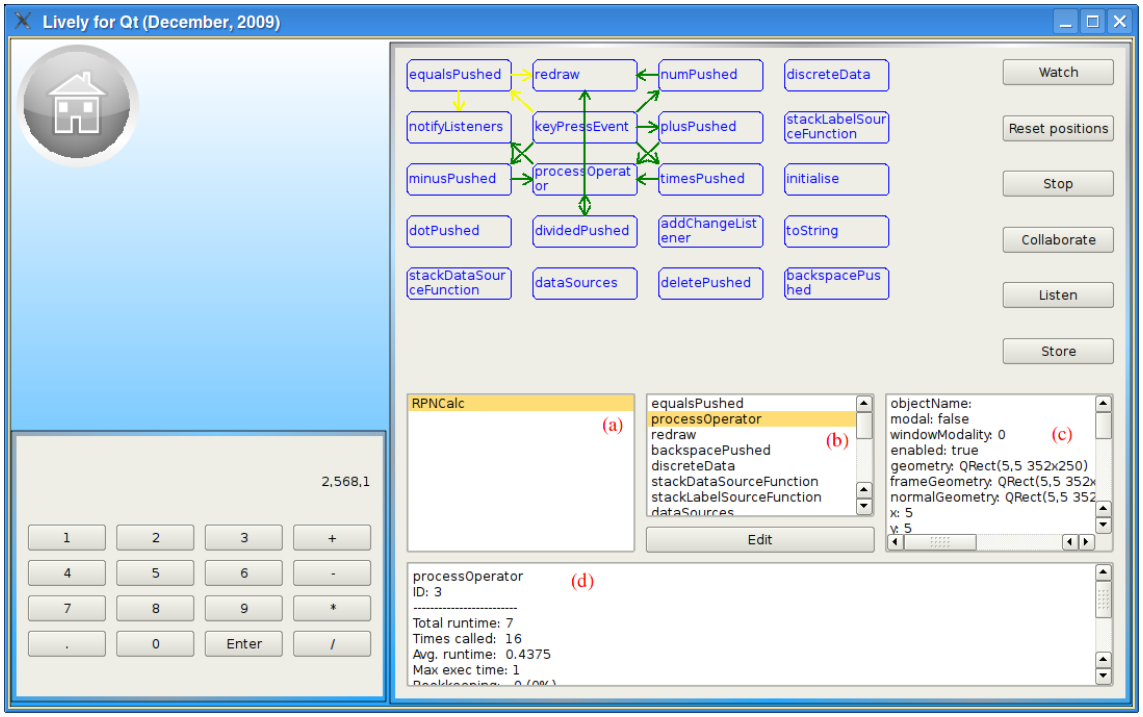


Figure 3.3: Introspector examining an RPN calculator instance

(a) lists the objects being inspected; (b) shows the methods on the selected object, while (c) shows its non-method properties and their values; (d) is used to display any textual data required by other parts of the system, and currently shows statistics about the execution of a selected method.

system.

As a large part of this project was exploring connections between applications, the Introspector is also able to watch multiple applications at once. The user clicks the "Collaborate" button and then selects the other application to examine. The methods from each application display in different colours, making the inter-application method calls obvious. These points of interface are difficult to track otherwise. The left-hand list box, labelled (a) in figure 3.3, lists the applications currently being observed. The user can select one to see its methods and properties listed in boxes (b) and (c), and can select one of the methods in (b) to see all of the collected statistics about that method in the text area at the bottom. These statistics include the number of times the method is called, how long it has run for, and which other methods it has called from this or other observed applications. These statistics can be plotted by connecting the Charter application (figure A.5). Clicking "Edit" will provide the source code of the method (if it is JavaScript) for the user to view, edit, and save into the running application. The code is displayed in the text area, or in a connected text-editing application with more advanced features if one is present. The Introspector can also eavesdrop on all method calls, parameters, and return values, and display them to the user. This mode is enabled with the "Listen" button, and shows the structure and nesting of all detected method calls in the text area. Listening in this fashion can produce a lot of data very quickly, and slows down the target application significantly, so it is not enabled by default.

3.6 Conway's Game of Life

This application is an implementation of Conway's Game of Life, simulating a cellular automaton and rendering its state to the screen with each iteration. The application acts as a time-series data source to the charting tool, giving the number of living cells at each iteration (figure A.4). The implementation performs very many calculations each iteration, so it has the potential for optimisation and performance measurement. We implemented the simulation using a fairly simplistic approach involving many nested method calls, providing both interesting information and a challenge to the Introspector.

The application led to the discovery of one of the major drawbacks in the implementation of Lively for Qt: it is transparently multi-threaded, and runs GUI and timer events in different threads. We will discuss the implications of this fact in more detail later, but for this application it rendered the entire system unusable above a particular scale, and led to unpredictable failures at smaller scales. The Introspector was a great help in discovering this behaviour, which also explained some previous errors in the Introspector itself. These errors were unavoidable within the framework in use.

3.7 Versioning text editor

The versioning text editor provides a basic text editing environment with the ability to roll back to previous versions of the document. It uses Qt's native textbox widget for the basic editing functionality, and several other widgets and dialogue boxes for other functions. It makes the widest and most varied use of the native Qt functionality of any application implemented in this project, and also makes heavy use of the ability to access the filesystem.

This application is able to connect to the Introspector and be used to edit methods on a watched object (figure A.6). The Introspector implements the same interface used for connecting the Charter and other applications together, extended to allow string and function data sources. While the function can be edited in the application's textbox otherwise, when connected to the versioning editor it is possible to roll back to previous versions of the function (even those whose changes were not made by the versioning editor). This is possible because the Introspector keeps track of the chain of original methods when it updates them, and is able to retrieve the code in order to pass it on through the data source interface. This feature is an aid to interactive programming on the system that is not otherwise available from Lively.

3.8 Syntax-highlighting code editor

This syntax-highlighting editor uses Qt's HTML rendering functionality to display coloured code text. The application includes a simple JavaScript tokeniser and parser and updates the highlighted rendering on changes to the document. This editor can also be used to edit code in the running system, and we used it for development during the later stages of the project.

The editor can manipulate code in the running system in two ways: editing methods on existing application objects, and evaluating fresh code. This editor can connect to the Introspector in the same way as the versioning editor, displaying the highlighted code of the currently-selected method for editing and able to save it back when changes have been made. This editor can also load and save code from files, and can evaluate the code in the global scope. Evaluating code directly allows editing constructors and prototypes, adding new functionality to existing applications, and creating new applications.

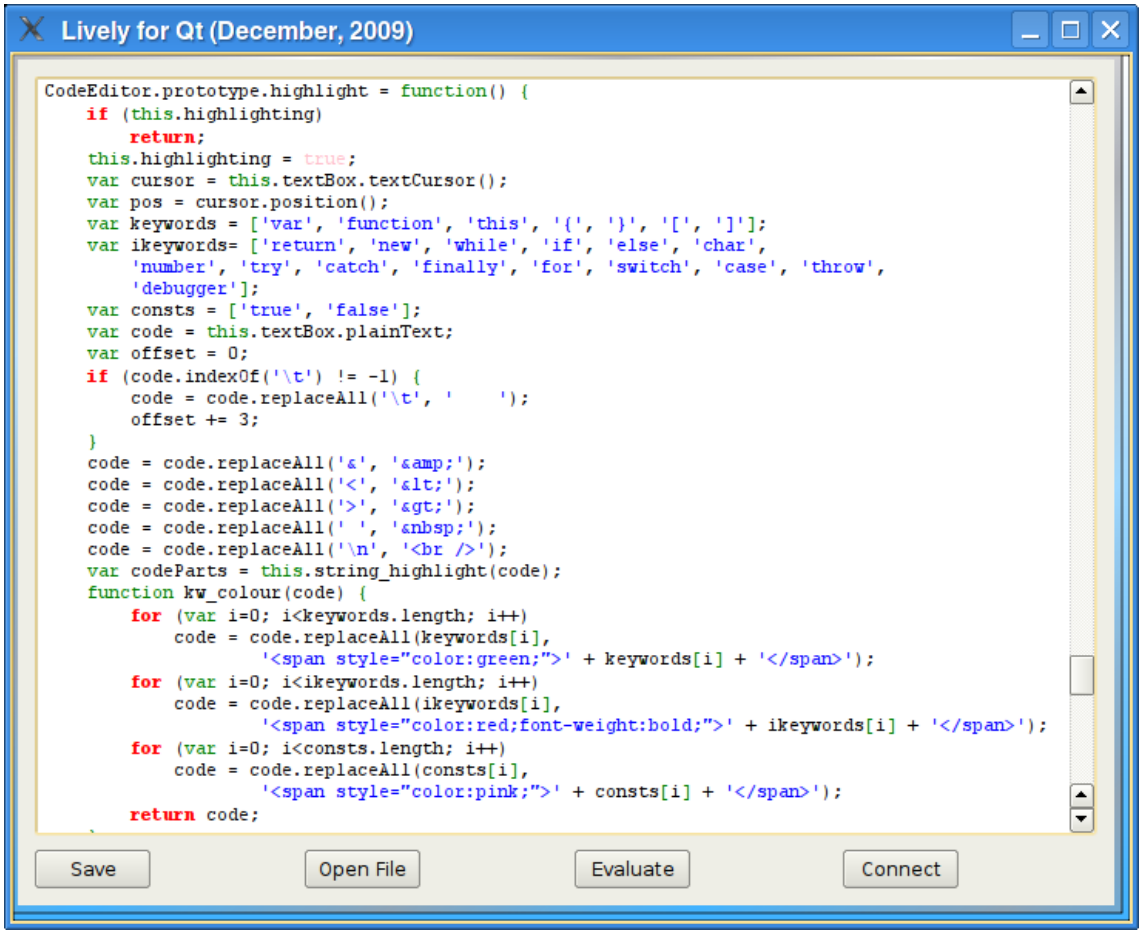


Figure 3.4: Highlighting code editor editing its own code

Once this editor was usable we used it for all our subsequent coding in the project. While there was a large adjustment involved in switching from Vim to the internal editor, we found that being able to evaluate changes immediately, without terminating and restarting the Lively for Qt system, was very helpful for development. We were able to make small changes and test them alone, rather than making several changes to test together. We could also edit the code of a particular instance of an application, allowing us to compare the behaviour of two different versions of the code.

We found that this editor filled a gap in the Lively environment. The built-in applications from Lively for Qt do not include an editor, which made interactive development challenging. In the earlier stages of the project all our development was outside the system, and even once the Introspector was in place we only made small modifications while the system was running. This editor provided most of the essential functionality of a programming editor and allowed us to modify whole applications. If a useful editor were part of Lively from the start we might have followed a different development path.

While this editor is conceptually similar to the versioning editor it has a very different implementation with virtually no code in common. The highlighting editor implements its own text-handling and transforms the code into HTML for display, while the versioning editor delegates all text functionality to the underlying Qt text area widget. The two applications are not fundamentally incompatible and could be combined in one, but we did not have the time in this project.

Chapter 4

Implementation Discussion

In this chapter we describe aspects of the language and environment we encountered during implementation that do not pertain to the applications themselves. These aspects are facts about the systems that weigh on our evaluation. We also describe how we addressed the issues we mention where appropriate.

4.1 JavaScript

During the project we encountered some aspects of the JavaScript language which raised unexpected issues. In this section we describe the unexpected behaviours and limitations of the language, including what methods we found for working with or around the functionality.

4.1.1 Prototype-based programming

The RPN calculator uses as its prototype an instance of the basic calculator, and so obtains all of the structure and functionality of the original calculator through code reuse. For the initial development of the calculator we also worked by extending an existing calculator **instance**, which the dynamic system allows. The way in which prototypes are specified lends itself to this type of development:

```
1 function RPNCalculator() {
2   // Constructor
3   ...
4 }
5 RPNCalculator.prototype = new Calculator();
6
7 RPNCalculator.prototype.equalsClicked = function() { ... }
8 ...
```

Figure 4.1: Prototype definitions

The prototype object here is a fully-functioning RPNCalculator-like application. It has the appearance and behaviour of an RPNCalculator, built up on top of the base Calculator instance. The prototype instance is usable during the development process and we can test the behaviour we have added on the running instance, modifying it in place if it is not correct. When the object has the behaviour we want we can set it as the prototype, adding the facility to create more similar objects on demand once we have a functioning system.

4.1.2 Constructors

A subclass must trigger the constructor of its superclass in order to initialise itself properly. JavaScript itself does not require this in all cases, but initialising the native objects corresponding to their JavaScript analogues **does** require that the QWidget constructor is called before any other Qt methods are called on an application. The subclass can call this directly, or it can invoke its superconstructor and perform all the necessary initialisations indirectly. While JavaScript has no syntactic support for accessing the superconstructor there are two other methods available:

- The constructor can be invoked as a function from within the subclass constructor. JavaScript function objects have a `call` method which invokes the function with a given recipient binding and parameter list. This is how the QWidget constructor must be invoked:

```
1 function MyWidget(parent) {
2     QWidget.call(this, parent);
3     ...
4 }
```

Figure 4.2: Invocation of superconstructors with `call`

  This method invokes the QWidget function as it exists at the time of the constructor call. In the case of QWidget, this is probably the expected behaviour, but for a user-defined object it may not be, particularly in the presence of interactive programming. When programming interactively, any changes to the required initialisation of an object means replacing the constructor function, but existing subclasses will have prototypes built with and expecting the setup of the original function and may break when data is different from what they expect.

- We can store the constructor used for the prototype as a member of the object, and simply invoke it using ordinary method dispatch when required. The constructor for our object can invoke `this.SuperConstructor()`, which will have the correct recipient binding without using `call`.

  If we invoke the superconstructor by name and a new function of the same name has been defined since we may encounter unexpected errors. The function may initialise the object differently than our prototype expects. These errors can be particularly annoying during an interactive programming session, when we can lose progress in an interpreter crash. Interactive programming is also the most likely time for a function to be re-bound in this way.

We have used the second method for most of our superconstructor invocations as it more often gives the expected behaviour. We use the first method for QWidget calls as it has less overhead and it is never desirable or useful to overwrite the base QWidget constructor, as it would no longer perform the necessary native initialisation.

4.1.3 Algorithmic performance

The versioning editor implements a version of the Patience Diff algorithm[8] for both displaying the differences between versions to the user and saving the versioned file more efficiently. This is a well-defined nontrivial algorithm based around longest-common-subsequence

calculation, which we chose for these properties as a means of evaluating JavaScript's algorithmic suitability. Implementing the algorithm in JavaScript was more difficult than expected because of limitations in its array and mapping functionality, but the final algorithm calculates differences correctly and is functional for moderately-sized inputs. Diffing texts larger than around a hundred lines is noticeably slow, and inputs much over 200 lines become intractable to the point that we are not sure whether they eventually terminate or have crashed the runtime engine.

We ran performance tests on the diff algorithm, calculating differences between two randomly-generated lists of "lines" of varying sizes, testing each size 90 times. The lines consisted of one of four symbols chosen at random. These lines are much shorter than real text lines, and incur less performance penalty from string manipulation, but still show a marked slowdown as the size increases. There is a notable jump between 150 and 175 lines, and beyond that point the increases in average time are much larger than before. For 250 lines and above the calculation did not terminate within three minutes.

# lines	Avg. time (ms)
25	18
50	24
75	31
100	33
125	38
150	45
175	97
200	125
225	133
250	Did not terminate

Figure 4.3: Average running times of our diff implementation for different sizes of input

4.1.4 Standard library

The diff algorithm used in the versioning editor relies on many array manipulations, which are expensive with JavaScript's string-keyed pseudoarrays, and also depends on a degree of memoisation to make calculation tractable. Some parts of the algorithm would have been much simpler with the new native features in recent versions of Mozilla's JavaScript and the ECMAScript 5 draft. Mozilla's JavaScript 1.6 and ECMAScript 5 include functional array extensions including `map`, `filter`, and `reduce` operations, which are executed as native code by the runtime engine and able to be much more efficient than JavaScript-implemented versions. These features are not available in the version of the Qt Script Engine used for this project.

We would also have appreciated some additional extensions to the Function type. Creating bound functions (with a defined receiver object) is complicated in JavaScript, as closures do not include `this` and the function to be bound must be re-invoked dynamically with the parameters forwarded through the `apply` method. ECMAScript 5 includes a `bind` method, which addresses much of this difficulty, but any other manipulations on functions are challenging to implement from JavaScript and would benefit from native methods. We would have found it especially useful to be able to clone a function, and to manipulate the closure of a function directly rather than having to use the `with` structure at definition time.

4.1.5 Qt callbacks

When binding callback functions to Qt signals (section 2.3) we have added a layer of indirection around the methods of our application. In our code signal bindings have this form:

```
1 var openButton = new QPushButton("Open");
2 openButton.clicked.bind(this, function() {this.openClicked();});
3 // Does not work correctly:
4 openButton.clicked.bind(this, this.openClicked);
```

Figure 4.4: Sample code binding an application method to a Qt signal

While we want the callbacks to invoke behaviour from the application, two traits of the callback system make binding the methods themselves problematic:

- Qt events bound using the built-in `connect` method bind a function. Functions do not include a binding to `this`, so the callback would not be able to access our application data.

  We later discovered an undocumented behaviour that allowed specifying the receiver as an additional parameter (as used in figure 4.4), rendering this part of our original reasoning invalid. This is one of many examples of incomplete documentation for the Qt Script Engine that we encountered.

- Function bindings from `connect` are early-bound at the time of the `connect` call: it is the provided function object which is called, not the named method on the application that exists at the time of callback. If the method in that slot is subsequently changed, which happens during prototype-based inheritance and in interactive programming, the signal will continue to call the original function.

  Method changes in an inheritor will always occur **after** the prototype's constructor has run, so any callbacks created within the constructor refer to the method from the unmodified prototype. If these callbacks trigger mutating behaviour on the receiver this would affect the prototype itself, and so propagate to every instance of the subclass.

  If code executes with the following effect:

```
1 button.clicked.bind(this, this.buttonClicked);
2 this.buttonClicked = function() { // ... new behaviour ... }
```

Figure 4.5: Early-binding of Qt signals in code

  then the final `buttonClicked` method will never be called, because the existing `buttonClicked` function was registered with the Qt event. While code exactly like the above is rare, prototype inheritance often has exactly this structure, and run-time changes to methods by interactive programming always suffer this problem.

JavaScript allows us to construct first-class functions inline, so there is no great inconvenience in creating this indirection function: we need only add some brief boilerplate to each of our signal bindings as in figure 4.4, but it is not the natural way of binding a callback (shown in figure 4.5). Because the problem only appears later on in inheritance corner cases and interactive debugging it can easily slip in unnoticed, creating difficult-to-track bugs.

4.1.6 Threading

One of the major unresolved problems we encountered in the project was directly related to the Qt bindings. Qt is inherently multithreaded, allowing it to provide non-blocking methods and perform graphics painting on demand. These repaint events trigger JavaScript code, if custom painting behaviour is defined, which can be running simultaneously with other code. Timer events similarly run out-of-band in another thread. JavaScript is a single-threaded language and provides no synchronisation or locking primitives; the combination of these two factors caused data corruption and untrappable errors that we were unable to work around. The QMutex class did not work from JavaScript code in the bindings used in this project, but is conceivably one way of addressing this issue.

Threading also caused issues with the built-in debugger. Ordinarily, the QtScript Debugger would launch when a JavaScript error occurred, showing which line triggered the error and allowing us to inspect the state of local variables. However, during a repaint event in any thread the debugger is suppressed, because the paint event was treated as uninterruptible. As well as making debugging the painting code itself nearly impossible this behaviour allowed errors in other code to pass silently at unpredictable times.

4.2 Lively

The Lively environment presents many novel features which were discussed in chapter 2. In this section we describe the functionality we found that was not described in the literature, and discuss the implications of this additional functionality.

4.2.1 Inter-application connectivity

Connections between applications are possible because of the shared application environment that Lively provides, and the object model of JavaScript. All applications execute in the same JavaScript namespace, and are able to see and access each other directly. The Charter defines an interface for other applications to act as sources for particular kinds of data, which can be supported simply by adding a method to the application object. The Charter accesses this method directly and does not need any intermediary to interpret or pass on its method calls.

4.2.2 "`that`" pointer

The Charter obtains a reference to the user's specified application using the `that` pointer (section 2.2). When the user wants to chart data from an application they need only start the charter and click on a "Connect" button. If the charter is able to detect the other application immediately (because the `that` pointer already refers to another application supporting the necessary interface) it will update immediately. Otherwise, it uses a repeated timer callback to detect changes to the `that` pointer and will adjust as soon as it detects a suitable referent implementing the necessary interface. This gives the intuitive behaviour for the general case, using the inbuilt capabilities of the Lively application environment.

Although the `that` feature was designed as an aid to interactive programming [27], as a way to refer to and programmatically manipulate a particular object in the system without naming it, it is also very useful for this type of inter-application interaction functionality. When one application is able to obtain a first-class reference to another application, knowing that the user is currently interacting with it, it can adapt itself to the other in ways that are much more difficult otherwise. We use this ability in both the Charter and editing

applications, and some others provide target-application functionality. We discovered this application of the feature during the project, and changed some of our application plans to take advantage of it and explore its implications.

4.2.3 Inter-application interface

To implement the inter-application interface an application must provide a `dataSources` method on its application object. This method returns an array of the data sources that the application provides. Each data source is an object containing at least the type ("discrete", "continuous", "text") of data and a function which can be invoked to obtain the current data for charting. There are several optional fields as well, providing for labels and display suggestions. Many applications have only one data source (such as the calculators), and so return a singleton array. Others, like the Introspector developed later, provide many different sets of data and of varying types. We needed to accomodate both of these use cases, and this combination of arrays and first-class functions fit this requirement. An earlier interface used distinguished `discreteData` and `continuousData` methods on the application, but this structure was unable to cope with multiple data streams from the same application.

The application should also provide an `addListener` method, which implements part of the Observer pattern[12], for requesting notification when the available data changes. If an application does not provide this method the rendered chart will update only intermittently, when fundamental paint events are triggered by the underlying windowing system. Only the `dataSources` method is required for an application to be treated as meeting the interface.

To test the suitability of this interface for general use, meaning with applications other than those developed as a part of this project, and to discover any necessary modifications to make it suitable, we implemented it in an existing application that ships with Lively itself. Figure 3.2 shows the built-in weather applet, which we had already extended to display our local weather, and a Charter instance plotting the current temperatures of various locations. The weather application is well-suited for this test as it naturally works with the correct type of data, key-value with a numeric range and arbitrary keys. We had some familiarity with its coding style from the previous modifications, so we could focus on only the essential difficulties of implementing the interface rather than needing to learn the structure of the code at the same time.

Adapting the weather application to serve its data took under two dozen lines of code, much of which was to deal with the newly-required bookkeeping of keeping track of its previously-fetched temperature data instead of discarding it. Total implementation and testing time was only a few hours. When new weather data arrives the application triggers an update of the chart, a facility which required only four lines in the weather applet and which is available to all other data sources as well.

4.3 Aspects

In this section we discuss our partial implementation of Aspect-Oriented Programming (AOP) for JavaScript, used in the Introspector to instrument methods. While the structure was designed for the purpose of instrumentation, the implementation is more general and many parts of it have broader application.

4.3.1 Introspector implementation

The Introspector replaces all existing methods on the target application with instrumented versions, calling the original method alongside some additional logging code encapsulated

in before and after functions. Constructing these functions accurately is complicated, and discussed later in this section. Once a replacement function for a given method is defined a reference to the original function is assigned to an `originalMethod` property on the replacement. This allows the tool to revert its changes when introspection is complete and aids some programming features developed later. The function is assigned to the correct slot on the watched object and enumeration continues.

When all of the replacement methods are in place the Introspector begins collecting data. The before and after functions collect and create both temporary and permanent records of aspects of execution. They maintain a call stack, containing one element for each function call currently executing. The before function initialises this element with the start time and blank entries for recording internal method calls, and returns the arguments unmodified.

The after function calculates the execution time of the method and adds it to the total recorded for that method. It also examines the call stack and adds a reference to the current method in its caller's entry if there is one, allowing the tool to track which methods call which others, along with timing and call-count information. The data from any internal method calls of the terminating method is transferred to the permanent record, and the call record is popped off the stack.

4.3.2 Reflection

When examining an application the Introspector uses JavaScript's built-in property enumeration to trawl through the members of the object. Code of the following form:

```
1 for (var key in object) {
2   value = object[key];
3   ...
4 }
```

Figure 4.6: JavaScript property enumeration

will loop over every key in the object, which is accessed as if it is simply a string-keyed associative array. The enumerated values include all methods of the object, including most methods inherited from the prototype (some built-in methods are marked "non-enumerable" by the runtime environment). As a method in JavaScript is simply a function stored as an object member, the methods can be found by testing `typeof object[key] == 'function'`. The first pass of introspection builds two lists partitioning the key set into methods and non-method properties. Both lists are displayed to the user, but the methods of an object require additional processing to be useful.

4.3.3 Instance-level method modification

In order to collect statistics about method execution the Introspector instruments the individual functions. Every method of the object is replaced by another performing the same function but also causing other logging to occur. The structure of this replacement is somewhat complicated, but putting it into place once created is not. All JavaScript properties are mutable and may be updated by simple assignment; assignment always occurs in the given object, even when setting a property previously obtained from the prototype. The instrumentation does not affect any other instances of the watched application. An instrumented application is no longer suitable for use as a prototype, but the necessary closures cannot be saved to disk so it would be a poor choice of prototype in any case.

4.3.4 Dynamic method replacement

To construct a suitable replacement method the Introspector must run the original behaviour alongside the instrumentation. The system can keep a reference to the original function to be invoked, but doing so for a method that is not fixed at the time the introspection code is written is nontrivial as passing on the parameters correctly to the inner invocation requires some knowledge about what arguments the function expects, and JavaScript functions do not provide this information.

The approach we have used combines four JavaScript features: first-class functions, dynamic invocation, variadic arguments, and closures. A function object can be invoked within a provided "this" context using its `call` method: `func.call(this)` will invoke the function as if it were a method on the current object. Arguments can be passed as `func.call(this, arg1, arg2, ...)`. For an unknown function this is not sufficient: the arguments are not known when the code is written. There are two potential ways to handle this:

1. Use the function's `length` property and dynamic code evaluation with `eval`. The length of a function is its arity, and dynamic evaluation can define new variables in the local scope so the created function binding will be accessible after `eval` has completed.

```
1  // Functions stringify to 'function (a,b,c) { body }'
2  funcStr = func.toString();
3  paramStr = ...;
4  var codeString = "return function(" + paramStr + ") {";
5  codeString += "// ... perform pre-execution actions ...";
6  // 13 is the length of 'function () {', the beginning of the
7  // stringification of a function. There is a closing '}' at the
       end.
8  codeString += funcStr.substr(13 + paramStr.length, funcStr.length
       - 14 - paramStr.length);
9  codeString += "// ... perform post-execution actions ...";
10 codeString += "}";
11
12 replacementFunc = eval(codeString);
```

Figure 4.7: Constructing function wrapper inline using `eval`

2. Use the `apply` method and the `arguments` pseudo-array for variadic arguments. The apply method on a `Function` object invokes the function using a provided recipient (bound to `this`) and an **array** of parameters:

```
1  // func and receiver are set appropriately in the enclosing scope
       .
2  function wrapper() {
3    // ... perform pre-execution actions ...
4    func.apply(receiver, arguments);
5    // ... perform post-execution actions ...
6  }
7
8  replacementFunc = wrapper;
```

Figure 4.8: Dynamic method invocation using variadic arguments

`arguments` is a special object created by the JavaScript engine, which contains the arguments to the function stored like an array. The object is not a true array and has none of the common methods from the Array prototype, but it does support the `length` field and element retrieval in the same form. The object is available in any function, but is most useful for implementing variadic arguments.

The first method is error-prone, because it requires generating code as a string. Our pre- and post-execution code runs in the same scope as the wrapped function, and may interfere with the original code unless we examine it very carefully. That method does not incur any invocation-time performance penalty.

The second method addresses all functions equally, but will be slower at runtime. Any mention of the `arguments` object in a function body attracts a performance penalty, regardless of how it is used, as the runtime must create and track the expensive object [10]. Given the existing penalty of dynamic invocation and some of the necessary logging operations, and the flexibility it allows, we chose to use the second approach in the Introspector.

4.3.5 Aspect wrapping

For each method on the object to be observed we create a replacement. The replacement function calls another provided function before executing the original, and passes it the object, method name, arguments, and the internal method ID used by the Introspector to identify each wrapped method. This pre-invocation function has the opportunity to modify the passed arguments, but the current implementation makes no use of this facility. The original function is invoked via `func.apply(obj, args)`, and another provided function is called afterwards, passed the return value (also modifiable) instead of the arguments. This structure allows the instrumentation to record method timings and call chains. The overall structure of the replacement function is:

```
1 function() {
2   var args = callBefore.call(this, obj, prop, arguments, methodid
      );
3   var returnValue = func.apply(obj, args);
4   returnValue = callAfter.call(this, obj, prop, returnValue,
      methodid);
5   return returnValue;
6 }
```

Figure 4.9: Structure of Introspector's method replacement functions

Adding these before and after functions is akin to aspect-oriented programming. The structure for applying them allows inserting arbitrary behaviour, not only the statistical logging we use, and it was no more complicated to implement this generic wrapping than it would have been to wrap in fixed logging code. While we do not use the ability to provide arbitrary wrapping layers in our applications, the structure does allow for it. During debugging we did use some different variants of the logging code to test the behaviour, which was much simpler with the function-passing structure than a hard-coded system.

In a true aspect-oriented language like AspectJ these pieces of wrapping code would be specified declaratively, with "pointcuts" identifying the relevant methods, and before and after "advice" declared for applying behaviour at those pointcuts[1]. The structure we use in this project is imperative instead, explicitly replacing each function found in an object with another. The modification is also applied at the object instance rather than class level, although the distinction between the two is less clear in JavaScript than in Java or AspectJ. We

were able to add this behaviour directly within the JavaScript language, rather than needing to extend the language, alter the source code programmatically, or run on a customised virtual machine. Our extension to the system does not provide all of the functionality of AspectJ, however, and is not able to intercept variable accesses or object instantiation. Variable and field accesses cannot be intercepted in JavaScript without modifying the runtime engine, but we can intercept any method calls easily. For the limited purposes we have in this project our restricted aspect-like functionality is sufficient.

4.3.6 Scoping and closures

Maintaining the necessary state for the replacement function is also a nontrivial problem. JavaScript functions act as closures, storing the enclosing lexical scope for the function body to access. The language does not have block scope, however, so creating these functions in a loop does not behave as would be expected for a similar feature in another C-like language. A local variable in JavaScript is scoped to the entire function that contains it, so any closures created within the function have a reference to the same variable even if it is declared inside a loop body. Creating many closures inside a loop, all having different bindings for their outer variables, is complicated by this trait of the language.

JavaScript keeps variable scope as a stack of objects, with variables looked up as keys in each object in turn. Reads are made from the innermost definition (the closest to the top of the stack), and writes to the nearest location where the key is defined, but otherwise in the top-level (probably global) scope. The `var` declaration creates a variable binding in the local scope that shadows an outer variable of the same name. A variable assigned without a `var` declaration is implicitly global, regardless of the surrounding scope, which can lead to some unexpected bugs. There is no error in redeclaring a variable multiple times, however, so we found it best to be promiscuous with `var` declarations and include them almost everywhere possible.

Any modifications that are made to a variable in a scope are visible anywhere else with that scope in its stack, and anywhere with that scope in the stack can modify variables in it at any time. A loop over method names that creates a function inside, for example, will in fact create functions with access to only the last method name in the loop, and not to the value of the variable at the time of definition:

```
1  function(methods) {
2    var closures = [];
3    for (var method in methods) {
4      var methodName = method;
5      closures.push(function() {
6          print(method);
7          print(methodName);
8        });
9    }
10   for (var i=0; i<closures.length; i++)
11     closures[i]();
12 }
```

Figure 4.10: Demonstration of JavaScript scoping and closure behaviour in loops

The above function will simply output the last method name in the parameter array repeatedly. Each function created refers to the same scope stack, and so when the binding of `method` and `methodName` is changed on the next iteration the value accessible through the

closure changes with it. In many other languages with closures of this form, such as Python and Smalltalk, local variables are scoped to their containing block, and so `methodName` would not be updated by subsequent assignments. JavaScript does not allow this technique, so more complicated manipulations are required.

There are two ways to create a new entry on the scope stack programmatically[1]: function invocation and the `with` statement. When a function is invoked, a new scope is placed on top of its stack containing its parameters, appropriately bound, and which ceases to exist at the exit of the function unless preserved in a closure. As JavaScript has first-class functions this function can define **another** function inside itself and return it. The inner function will inherit the entire scope stack of its parent as a closure, including the function parameters. Calling such a higher-order function with parameters to close over is safe within a loop, and was the original method we chose for this tool.

The `with` statement pushes an object onto the scope stack for its accompanying block. Any variable lookups for members defined in that object will access the slot in it instead of in any of the parent scopes. While this can be problematic with arbitrary objects, as it is unclear where a given lookup or assignment will actually occur, it is safe and predictable with an object literal:

```
1 for (...) {
2   with ({originalmethod: meth, methodid: id}) {
3     obj[meth] = function() {
4         ...
5     }
6   }
7 }
```

Figure 4.11: Use of `with` to open a new scope

The function defined in this scope has access to the "`originalmethod`" and "`methodid`" variables in its outer scope, which will not vary with reassignment on subsequent iterations. This approach avoids an extra function invocation and keeps related code together.

Although the `with` statement is useful in this context, it is prohibited in the "strict mode" of the most recent ECMAScript version, because it can be unsafe unless used with a literal [11, p93]. As strict mode is optional, this technique is still applicable if the program is willing to forego universal strictness. A non-strict function can be defined within strict code using the Function constructor [11, p51], which takes a string containing code and creates a function with that code as its body. A function created this way is only evaluated with strict mode on if the function body itself requests it, with no care for the status of the surrounding scope.

Mozilla's JavaScript 1.7 includes a `let` declaration that closely resembles block-scoped local variables in other languages, along with a `let` statement having essentially the same function as the above use of the `with` statement. Neither of these is available in the implementation we used in this project, nor were they included in ECMAScript 5.

[1] The `catch` block of a `try ...catch` statement also creates a new lexical environment [11, pp96-7], but using it for this purpose is so pathological that we do not consider it seriously.

Chapter 5

Evaluation

In this project we set out to assess the JavaScript language and the Lively application environment, with particular reference to the claims made about them in the literature. From the development of our applications we have learnt about many aspects of both systems, as well as about the Qt toolkit and the interactive programming approach that Lively encourages.

5.1 JavaScript

JavaScript did not present us any major obstacles in using it as a general-purpose language. Despite being designed as an embedded scripting language the language is not restricted in any way that prevents its deployment for other purposes, although it has some idiosyncratic traits that make large programs more challenging than they would be in a language designed with such usage in mind. Some features almost invite the introduction of bugs, like the default global scope for variables and semicolon insertion. If a `var` declaration for a variable is omitted assignments are made in the global scope, leading to subtle bugs when multiple functions use the same variable or one is recursive. The JavaScript parser implicitly inserts semicolons at line breaks in some (but not all) circumstances, so statement boundaries may not be where the programmer expected. Disciplined use of sensible coding practices mitigates the harm from these design choices. Although there may be difficulties with generalising such practices across many programmers, we do not consider that to be more of a problem for JavaScript than other languages. Projects involving many programmers will always need to have coding standards in place.

JavaScript's dynamic nature allowed us to impose new structures and behaviour upon programs that were not part of the language. We could add AOP-like extensions without modifying the underlying runtime, for example, which enabled us to express our program in a more natural way for the task at hand. The ability to make this type of change without significant alterations to the compiler or runtime system does not exist in many languages, and is usually found in "scripting" languages. Building some of the tools we constructed in this project would have been much more difficult without the ability to make these extensions, and the code would have been less maintainable. These "scripting" features can be useful for general-purpose development.

The language has no built-in module system or way of separating conceptually-distinct blocks of code. We did not experience this as a major issue in this project because all of our subprojects were relatively small and the application boundaries sufficed, but larger projects might encounter more of a problem. Although there is no inherent module system in the language one could be provided by the runtime system or imposed by code inside the program. Lively's application separation from Lively is in essence a partial module system

imposed by application design.

5.2 Lively

The shared application execution environment that Lively provides gives a unique platform for building applications. The ability for independent graphical applications to connect to and interact with each other allows an intriguing and useful structure which opens up new functionality almost for free to other applications. The Charter application in this project allows any application with suitable data to expose it through a simple interface and have it charted in a variety of ways, without any significant implementation cost.

Structuring small special-purpose applications in this fashion also limits the effect of one of the shortcomings of the JavaScript language, its lack of a module system. Building a large monolithic application without formal structure is difficult, but application boundaries are a natural break in structure. We found that our smaller applications were simpler to keep mental track of than our larger ones, even where they were interacting with other applications in the system. The Lively literature underplays this advantage but it is the most significant feature of the system observed in this project.

5.2.1 Implications of connections

Of all our applications the Charter makes the most, and the most varied, use of the ability to access others, because its role is very suited to obtaining data from other sources. The text editors similarly have reasonable connections to other applications, but most interesting is the development style the ability encourages. We built several small special-purpose applications, all able to interact with each other, rather than monolithic multipurpose systems. We originally planned to build charting functionality into the calculator and use this experience and relevant code to build the same function into the Introspector, but instead we could build one more general application able to plot data from all sources.

The program design that this ability enables resembles that espoused by the "UNIX philosophy" [23, 16]. Applications do one thing and do it well, rather than attempting to encompass all the related functionality which might be required, and applications work together. This move away from monolithic applications to smaller single-purpose ones able to operate independently made the codebase easier to manage, and is an ability that does not usually exist in other environments.

5.3 Prototype-based programming

We found prototype-based programming as found in JavaScript to be a useful paradigm. In particular, being able to build up a suitable prototype instance that was fully functional throughout before assigning it as a prototype to the constructor was helpful when developing new applications, especially those based on an existing application. We were able to create multiple prospective prototypes and evaluate them alongside one another, allowing for very rapid experimentation.

The variant of prototyping that JavaScript uses has some unusual aspects that other prototype-based languages do not have, which have both advantages and drawbacks. Prototypes must be associated with a constructor function and cannot be assigned arbitrarily, or to existing objects at all. We would have found it useful to be able to set prototypes on existing objects and without a constructor, particularly for some of our data objects. Some

parts of the Introspector implement manual "prototyping" for data objects, retrying lookups in another object when the result of one is undefined.

While we used many features of the referential prototyping of JavaScript, particularly during interactive programming, at times we would have appreciated a primitive "clone" operation making a full copy of an object, as found in concatenative prototyping languages. Both kinds of prototyping can be useful, and JavaScript (like all other prototype-based languages we are aware of) includes only the one type. We would also have found it useful to have more explicit access to object prototypes, so that we could access and change the prototype reference of an existing object. The runtime engine in use provided an undocumented and non-standard `__proto__` property referring to the prototype, but this property was not writeable.

5.4 Qt and QtScript Bindings

The Qt toolkit was very useful, not only for its graphical widgets but for its non-GUI classes dealing with file access and other features of the underlying system. The widgets were at least as easy to use as in any other toolkits we had experience with.

The JavaScript bindings were reasonably natural, but largely undocumented. When documentation for the bindings did exist it often consisted only of a list of members, without even distinguishing them into properties and methods. Often we were forced to refer to the (comprehensive) C++ documentation for Qt and attempt to find parallels with the JavaScript bindings. Finding the correct parallels was often a matter of guessing, as some Qt accessor methods were transparently translated to properties in JavaScript and some not, and often only one or two variants of an overloaded method were accessible from JavaScript. Some of the bindings were automatically generated, and a few of these existed but were unusable because other methods they required to be useful were unavailable, particularly those relating to the underlying graphics contexts.

As Qt and its bindings are multi-threaded and JavaScript is single-threaded we encountered problems with some of our applications. In large part this arose from a mismatch between the expectations of Lively, designed for a single-threaded event-driven browser environment, and Qt. Applications written entirely to one model or the other would be unproblematic, but the mixture of the two caused difficulties. Some of these could have been avoided were there synchronisation primitives in the language, or even library-based locking from Qt.

5.5 Interactive programming

While interactive programming is a claimed strength of the Lively environment, its built-in support is incomplete. The environment includes an "Evaluator" widget to execute user-provided JavaScript code, but no tool for modifying the code or state of other objects. The `that` pointer was useful from this evaluator for reasonably trivial manipulations, mostly those affecting the Lively rendering of the application rather than the application itself, but it was difficult to compose any substantial modifications this way. With only the evaluator to use we were limited to "toy" changes, rather than real development.

Once we had built the Introspector and associated tools we found interactive programming to be a useful model for development. Directly modifying existing objects and immediately observing the effects in a system that was already running allowed us to make rapid progress when developing new features, and to verify attempted bug fixes quickly.

The major drawback of the interactive approach was that we would occasionally lose work when our interactive changes caused interpreter crashes. Causing destructive crashes required very particular circumstances, usually editing one of the applications taking part in the interactive programming process itself, so we were able to avoid them almost entirely once we realised what the problem was. We were able to avoid the issue because we had already written the bulk of our tools, and so we could avoid reflexive editing, but working interactively from the earliest stages would have been difficult. Successful interactive programming requires a comprehensive suite of preexisting development tools.

Chapter 6

Conclusions

JavaScript and Lively present an interesting environment for application development. Each presents some novel features that are not found elsewhere, and bears out some of the claims made in its literature.

JavaScript was designed and remains primarily used as an embedded scripting language, but is not unsuitable for general-purpose development. No aspect of the language renders it unusable for purposes other than scripting, although using it in large-scale projects would require careful attention to coding standards to avoid certain poor language design choices. Maintaining a large codebase without such standards would be very difficult, particularly given the lack of a module system, and some behaviours that were sensible in the original application of the language invite bugs if not avoided.

The language allows very rapid development with its prototype-based inheritance mechanism. When developing a new object type a functional instance of the object can be built up, and then used directly as the prototype for creating further instances. Constructing a new subtype with shared functionality involves modifying an instance of the existing object to use as a prototype, with no more complexity than is inherent in achieving the desired behaviour. The lack of distinction between classes and objects enables development to focus on the desired behaviour rather than the type structure without requiring subsequent refactoring. JavaScript's form of prototyping is imperfect and does not expose all of the prototype-related behaviour that can be useful for development.

Lively's interactive application environment allows some novel types of application. Applications can see and access each other, and discover which application the user is interacting with through the `that` pointer. First-class method calls on other applications allow new kinds of inter-application collaboration, and the transfer of the "UNIX philosophy" of many small special-purpose applications to a GUI environment in ways that are not otherwise possible.

Lively also aims to enable interactive programming. In interactive programming the running program is manipulated, modified, and extended at runtime. Lively's `that` pointer provides a reference to the application selected by the user without requiring a named reference, and allows code to be evaluated within the system to manipulate this or other objects. The built-in capabilities of Lively for Qt do not provide significant useful support for this programming model, allowing only "toy" manipulations to be practical. "`that`" was not as directly useful as claimed. The system required further extension before practical interactive programming was possible, but with our introspection application interactive programming became the primary mode of development in the later stages of the project. The Introspector built on the facilities of Lively and JavaScript, and required almost all of the novel features of both. Interactive programming was both useful for development and allowed us to proceed more rapidly than we had previously, and was a strength of the Lively environment

once the supporting functionality was complete.

Our project built many applications using JavaScript and Lively in order to test the claims of the literature about both. The range of applications covered the important features and claims about both systems, and allowed us to assess the interactivity at the heart of Lively. The applications are independently useful, not previously existing in Lively, and demonstrate different aspects of the two systems. We found JavaScript to be a suitable language for general-purpose development with some caveats arising from its historical application, while Lively allowed novel interactions between different applications that are not common elsewhere, and interactive programming was a useful technique within such a dynamic environment.

Future work

We believe that further investigation of the potential of connecting single-purpose GUI applications together as we did in Lively here would be useful. While the mechanism we used is probably not applicable outside a shared environment like Lively the behaviour we uncovered seemed useful. Similarly, the "`that`" pointer provided new avenues for functionality and is deserving of more research.

The use of JavaScript for top-level desktop development also merits further examination. This project did not address top-level development, only applications running inside an existing JavaScript-based window system. This work could include investigating adding synchronisation primitives to the language, or alternative mechanisms of supporting responsive GUI programming. We also believe it would be useful to investigate how large-scale JavaScript development might work, and whether the limitations of the language make it unsuitable for large many-programmer projects.

Acknowledgements

James Noble for the concept, and his feedback throughout the project.

Lena Herrmann for the JavaScript highlighting settings for `listings`[1].

Roma Klapaukh, Hugh Davenport, Christabel Marshall, and Alexandra Donnison for their notes and feedback on this report.

Samantha Garcia, Amy Chard, and Bruce Mills for their contributions to the oral presentation.

[1] `http://lenaherrmann.net/2010/05/20/javascript-syntax-highlighting-in-the-latex-listings-package`

Appendix A

Application screenshots

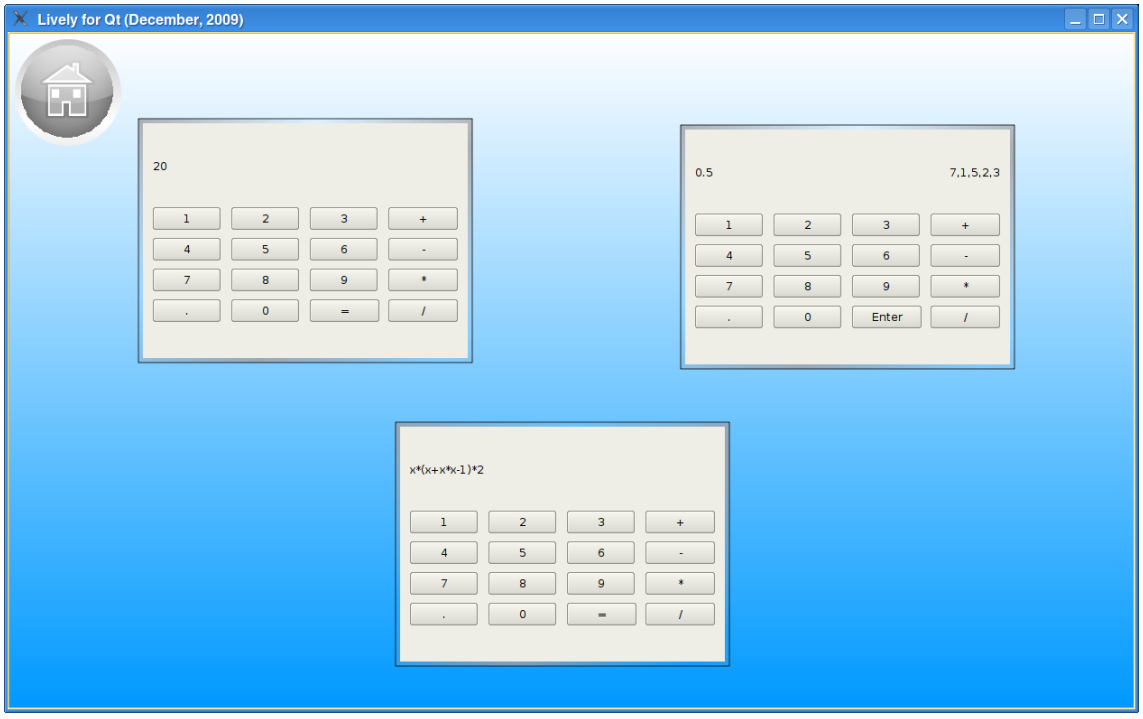


Figure A.1: Basic calculator (top left), RPN calculator (top right), and scientific calculator.

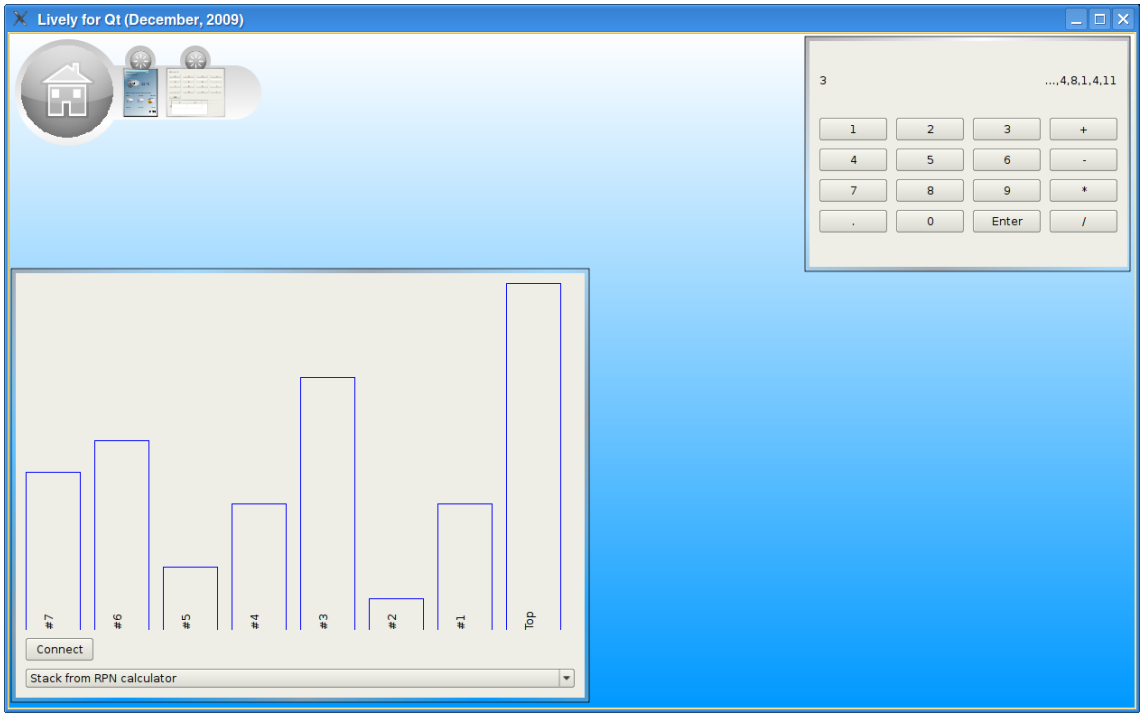


Figure A.2: Charter plotting stack from RPN calculator

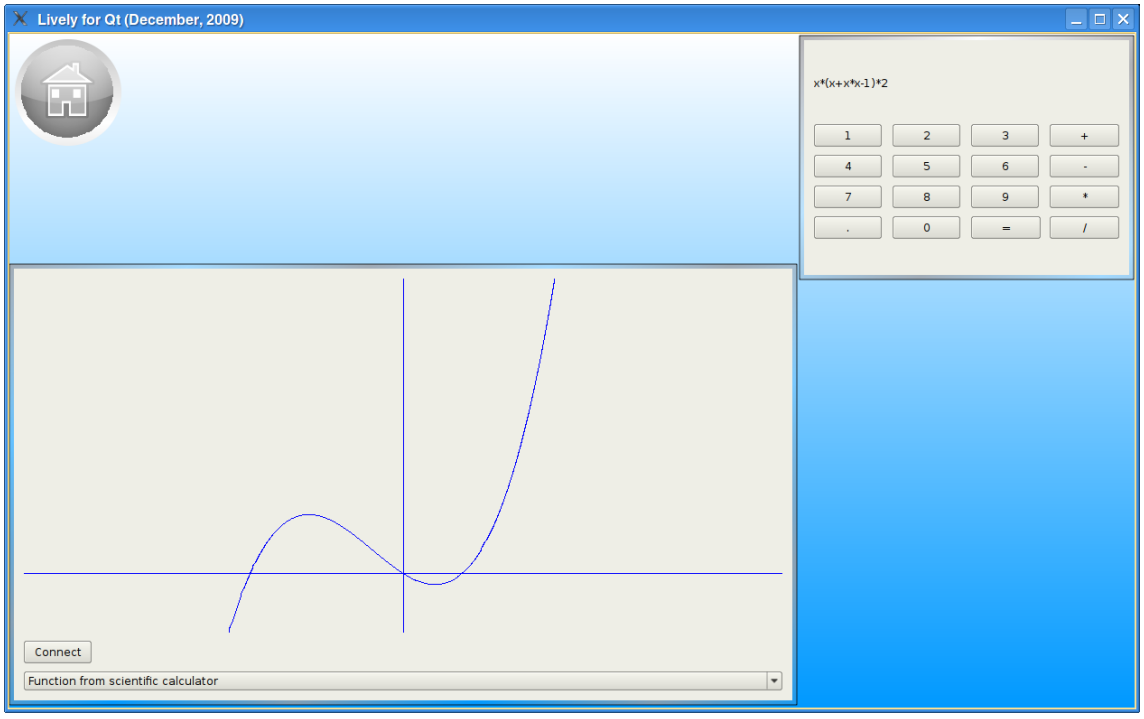


Figure A.3: Charter plotting function from scientific calculator

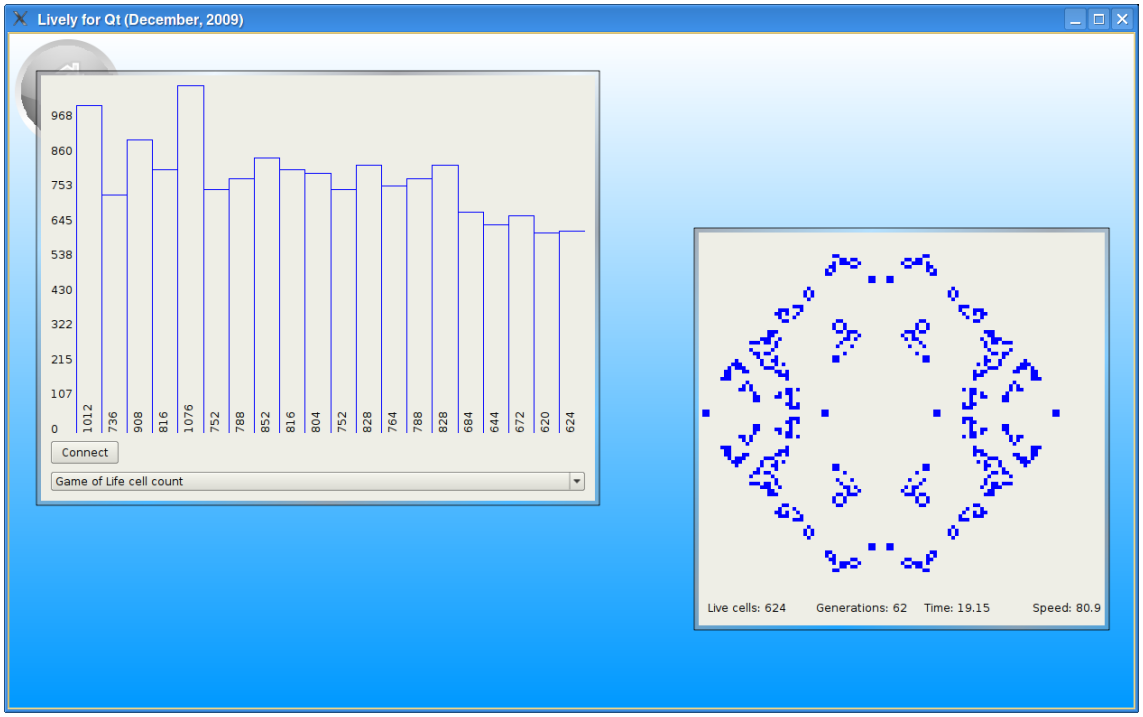


Figure A.4: Charter plotting live cells from Game of Life

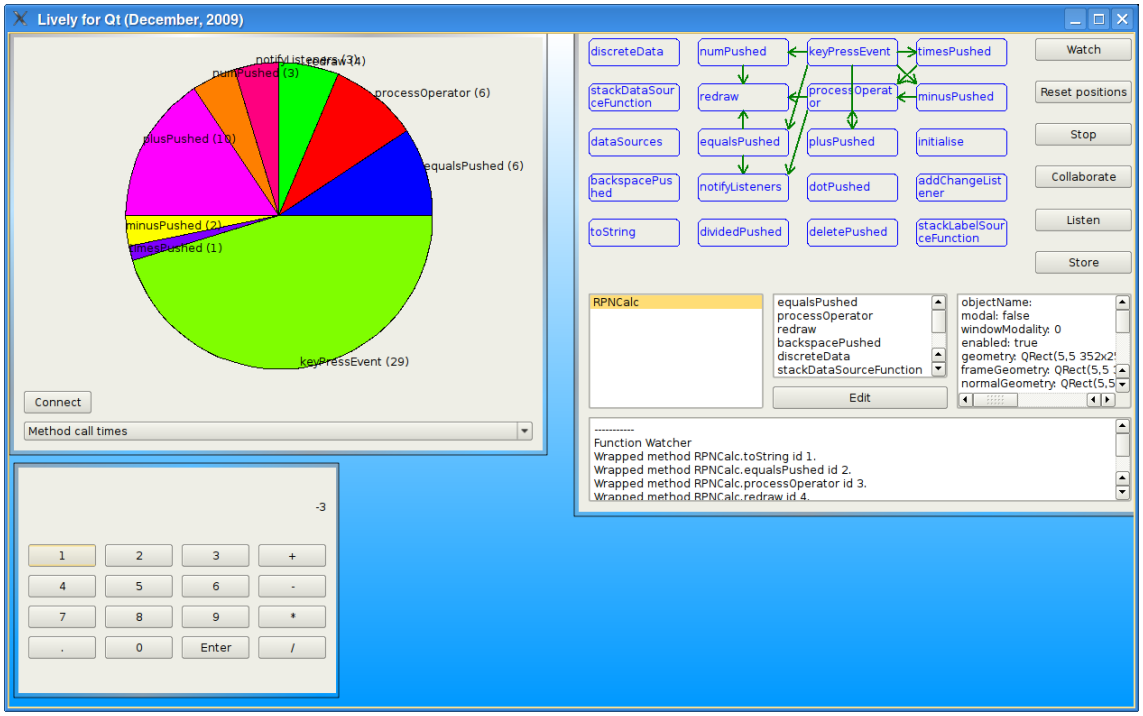


Figure A.5: Introspector examining RPN calculator instance while providing Charter method call statistics.

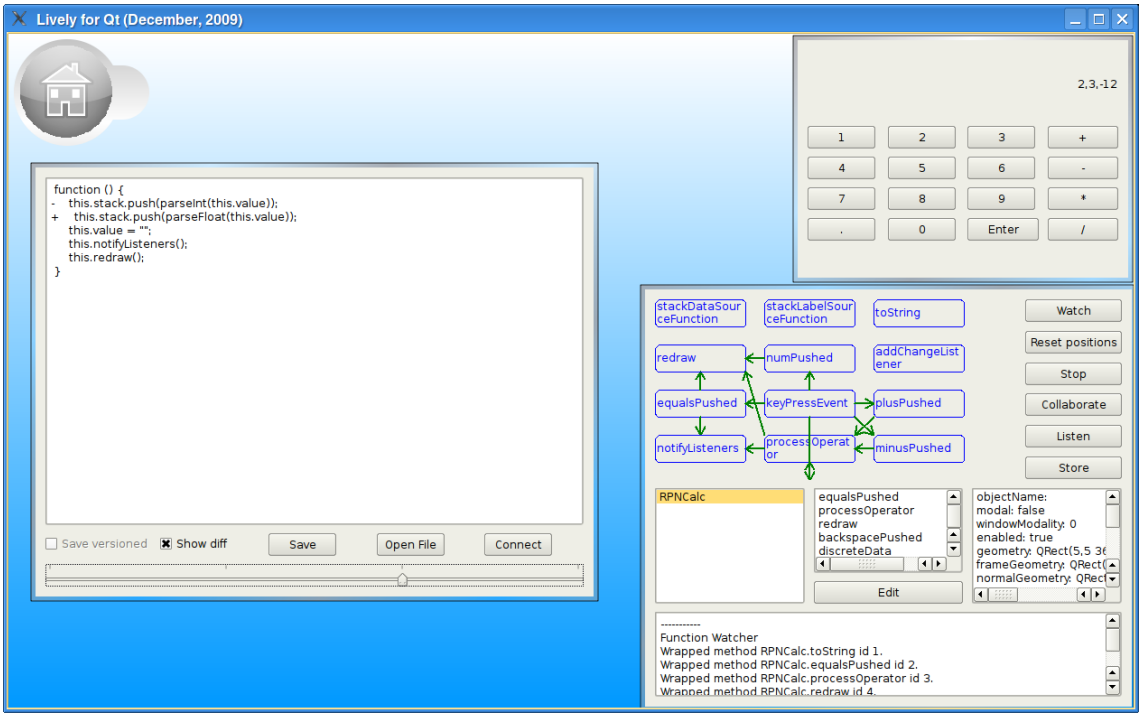


Figure A.6: Versioning editor viewing changes in RPN calculator method

Bibliography

[1] The AspectJ programming guide. `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`, October 2010.

[2] Lively for Qt. `http://lively.cs.tut.fi/qt/`, October 2010.

[3] Lively Kernel. `http://www.lively-kernel.org/`, October 2010.

[4] node.js. `http://nodejs.org/`, October 2010.

[5] Squeak Smalltalk. `http://www.squeak.org/`, October 2010.

[6] ADOBE. rich Internet Applications | Adobe AIR. `http://www.adobe.com/products/air/`, October 2010.

[7] CARDELLI, L., AND PIKE, R. Squeak: a language for communicating with mice. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques* (1985), pp. 199–204.

[8] COHEN, B. Patience diff advantages. `http://bramcohen.livejournal.com/73318.html`, October 2010.

[9] DONY, C., MALENFANT, J., AND BARDOU, D. *Prototype-Based Programming: Concepts, Languages, and Applications*. In Noble et al. [20], 1999, ch. Classifying Prototype-based Programming Languages, pp. 17–45.

[10] ECMA. Standard ECMA-262 ECMAScript Language Specification, 3rd Edition. Tech. rep., Ecma International, 1999.

[11] ECMA. Standard ECMA-262 ECMAScript Language Specification, 5th Edition. Tech. rep., Ecma International, 2009.

[12] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995, ch. Observer Pattern, pp. 293–304.

[13] HOEHRMANN, B. RFC 4329 Scripting Media Types. Tech. rep., Internet Engineering Task Force, 2006.

[14] INGALLS, D., KAEHLER, T., MALONEY, J., WALLACE, S., AND KAY, A. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (1997), pp. 318–326.

[15] INGALLS, D., PALACZ, K., UHLER, S., TAIVALSAARI, A., AND MIKKONEN, T. The Lively Kernel a self-supporting system on a web page. In *Self-Sustaining Systems: First Workshop, S3 2008 Potsdam, Germany, May 15-16, 2008 Revised Selected Papers* (2008), pp. 31–50.

[16] KERNIGHAN, B. W., AND PIKE, R. *The Unix Programming Environment*. Prentice Hall, 1984.

[17] LIEBERMAN, H. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of the 1986 conference on Object-oriented programming systems, languages, and applications* (1986), pp. 214–223.

[18] MIKKONEN, T., AND TAIVALSAARI, A. Using JavaScript as a real programming language. Tech. Rep. 2007-168, Sun Microsystems Laboratories, 2007.

[19] MIKKONEN, T., TAIVALSAARI, A., AND TERHO, M. Lively for Qt: A platform for mobile web applications. In *Proceedings of the 6th ACM Mobility Conference* (2009).

[20] NOBLE, J., TAIVALSAARI, A., AND MOORE, I., Eds. *Prototype-Based Programming: Concepts, Languages, and Applications*. Springer-Verlag, 1999.

[21] NOKIA. Qt - a cross-platform application and UI framework. `http://qt.nokia.com/`, October 2010.

[22] NOKIA. Qt Labs - Projects/QtScript/Generator. `http://labs.trolltech.com/page/Projects/QtScript/Generator`, October 2010.

[23] PIKE, R., AND KERNIGHAN, B. Program design in the UNIX environment. In *USENIX Summer Conference* (1983).

[24] SMITH, R. B. Prototype-based languages (panel): object lessons from class-free programming. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications* (1994), pp. 102–112.

[25] TAIVALSAARI, A. *Prototype-Based Programming: Concepts, Languages, and Applications*. In Noble et al. [20], 1999, ch. Classes vs. Prototypes, pp. 3–16.

[26] TAIVALSAARI, A. Mashware: The future of web applications. Tech. Rep. 2009-181, Sub Microsystems Laboratories, 2009.

[27] TAIVALSAARI, A., AND MIKKONEN, T. Simplifying interactive programming with keywords 'that' and 'those'. In *To appear in Proceedings of the 36th Euromicro Conference on Software Engineering and Advanced Applications* (2010).

[28] TAIVALSAARI, A., MIKKONEN, T., INGALLS, D., AND PALACZ, K. Web browser as an application platform: The Lively Kernel experience. Tech. Rep. 2008-175, Sun Microsystems Laboratories, January 2008.

[29] UNGAR, D., CHAMBERS, C., CHANG, B.-W., AND HÖLZLE, U. Organizing programs without classes. *Lisp and Symbolic Computation 4*, 3 (1991).

[30] UNGAR, D., AND SMITH, R. B. Self: The power of simplicity. In *OOPSLA '87 Conference Proceedings* (October 1987), pp. 227–241.

[31] UNGAR, D., AND SMITH, R. B. Self. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages* (2007), pp. 9–1–9–50.