The Larch Environment - Python programs as visual, interactive literature

G. W. French

A Thesis submitted for the degree of Master of Science

School of Computing Science
University of East Anglia

January 2013

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisory team, Dr. Richard Kennaway, and Prof. Andy Day for their support, advice, and help throughout this project.

I would like to thank Jim Huginin, Frank Wierzbicki, and all other Jython project contributors. Jython [70] is a Java implementation of the Python language. Without Jython, our work would have been very difficult. Our system utilises Martin Jericho's 'Jericho' HTML parsing library, and Mark McKay's 'svgSalamander' library.

I would like to thank Dr. Joost Noppen his advice positioning and contextualising our work.

Special thanks to Dr. Gregory V. Wilson, whose ACM Queue article [84] provided the initial inspiration for our project.

Abstract

'The Larch Environment' is designed for the creation of programs that take the form of interactive technical literature. We introduce a novel approach to combined textual and visual programming by allowing visual, interactive objects to be embedded within textual source code, and segments of source code to be further embedded within those objects. We retain the strengths of text-based source code, while enabling visual programming where it is beneficial. Additionally, embedded objects and code provide a simple object-oriented approach to extending the syntax of a language, in a similar fashion to LISP macros. We provide a rapid prototyping and experimentation environment in the form of an active document system which mixes rich text with executable source code. Larch is supported by a simple type coercion based presentation protocol that displays normal Java and Python objects in a visual, interactive form. The ability to freely combine objects and source code within one another allows for the construction of rich interactive documents and experimentation with novel programming language extensions.

Chapter 1

Introduction

Plain text remains the dominant medium for source code despite the development of visual programming languages that represent programs in a diagrammatic form. There are two reasons for this. Firstly, diagrammatic representations of programming constructs have met with varying success; some hinder comprehension [25], while others (such as SWYN [5]) have been beneficial. Secondly, visual programming languages have been found to suffer from the *scaling up problem* [10]; diagrammatic representations of large and complex programs tend to be unwieldy, and difficult to understand and navigate [53, 83].

Modern IDEs provide an efficient interface for editing and navigating through source code. Unfortunately, the range of visual cues available is limited to the use of colours, text style and indentation. As a consequence, professional programmers miss out on the opportunity to use visual forms to improve the readability of their code. While visual programming languages are cumbersome to work with, carefully designed visual notations can be helpful [83]. To better serve the needs of programmers, a development environment should combine both forms. To this end, we have developed an approach to partially visual programming inspired by technical literature. Technical literature is primarily a textual medium, in which diagrams, spatial arrangement (e.g. tables) and other visual forms are used where prose is insufficient to convey the desired information. Our approach to partially visual programming is based on embedding normal objects within source code and rendering them visually, so that programmers can use visual constructs within code where they are beneficial. Our technique is a simpler alternative to those introduced by Racket [60] and ETMOP [19].

The ability to freely mix textual and interactive visual source code representations supports the development of in-line interactive development tools (such as the program visualisation tool in section 8.1.4) and customised forms of the host programming language that are aimed towards supporting a specific domain (such as the MIPS CPU simulator in section 8.1.5). In section 8.1.7 we demonstrate a way in which the work-flow of unit-testing can be improved. We have developed table based unit tests that simplify the test development process and juxtapose the test results with the test code.

For partially visual programming to be truly useful, a programmer must be able to develop their own visual constructs. In aid of this, we have designed a simple protocol to control the compile and run time behaviour of visual con-

structs.

The development of visual programming constructs is supported by an object presentation system based on *type coercion* (the process of converting objects to a specific type; explained further in section 4.2). It eliminates much of the boilerplate code that is found in many Model-View-Controller [9] implementations. Besides the development of visual programming constructs, it is also used to implement the GUIs for the various tools and components that make up Larch.

We provide a simple approach to developing a visual representation for an object, allowing developers to visualize the state of their objects for either user interface or debugging purposes. The careful use of spatial arrangement (e.g. tables), visual cues and diagrams can improve comprehension [83], when contrasted to plain-text based representations.

These components operate within the context of the active document based Larch programming environment. Larch provides a REPL (read-eval-print-loop) based console that displays objects in a visual form using the object presentation protocol discussed above. Like other REPL based consoles, it offers an effective experimental programming environment, thanks to its rapid edit-run-debug cycle. Unfortunately the simple command-line interaction model limits its effectiveness when working with more than a few lines of code.

For complete modules, we provide a system based on active documents. Active document systems augment rich text with procedurally generated interactive content. Notebook systems such as Vital workbooks [27], Mathematica notebooks [88] and IPython notebooks [58] interleave rich text and executable source code, whose output is displayed in-line. Our worksheet active documents operate in a similar fashion. Two modes of operation allow them to function as both a user facing application and a developer facing programming environment. Drawing inspiration from Smalltalk [22], our system encourages programmers to frequently execute and test their programs throughout the development process.

The components of Larch are designed to work in harmony in order to provide a useable interactive environment. The visual object presentation system acts as the core of Larch, allowing applications to be developed as objects, some of which are displayed to form a user interface. The development environment GUI, program source code and execution output all consist of objects that reside within the same object space. This approach was pioneered by Smalltalk [22].

In summary, Larch combines the embedding of interactive objects within source code with an active document based programming environment. It is built upon a visual object presentation system that operates on similar principles to the live object based development environments introduced by Smalltalk. Like Smalltalk, Larch is inherently dynamic; objects can be modified interactively or programmatically, and code within Larch documents can be immediately executed.

We will cover our approach from two angles. It will be considered from the perspective of a developer looking to implement our approach within a different system, such as an existing Integrated Development Environment (IDE). We will also briefly discuss our implementation within the active document based programming environment within Larch.

Our novel contributions are the type coercion based approach to object presentation — which will be discussed in chapter 4 — and our embedded object

based approach to partially visual programming — which will be discussed in chapter 7.

We present a survey of background work in chapter 2. Our presentation system is discussed in chapter 3. A rich content editing tool kit is discussed in chapter 5. The interactive programming environment is described in chapter 6. We evaluate our system with proof of concept examples and performance measurements in chapter 8. We present our conclusions in chapter 9. A conceptual overview of the design of our system is shown in figure 1.1.

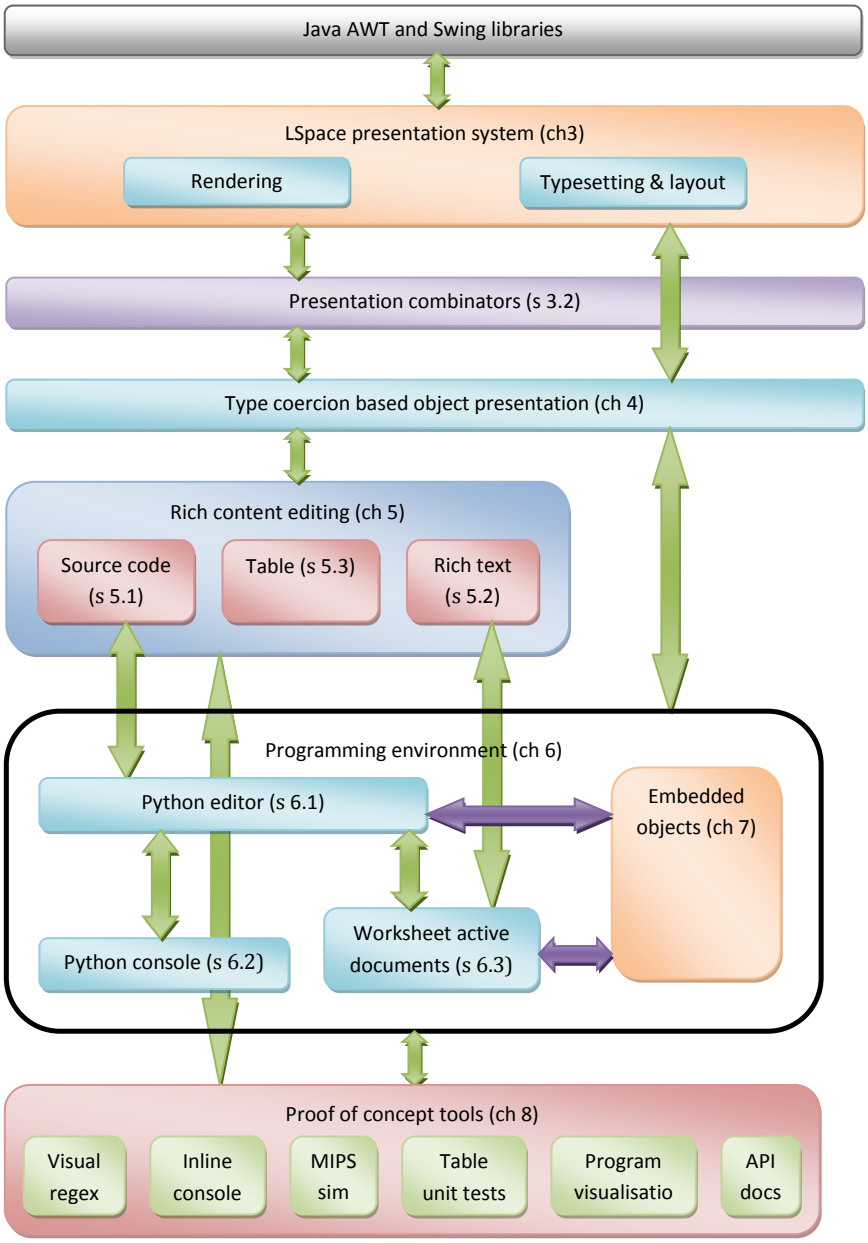


Figure 1.1: Larch system conceptual overview

Chapter 2

Background

2.1 Presentation systems

2.1.1 Document mark-up systems

Document mark-up systems render plain text documents that are marked up with tags or control structures that provide structural and styling information.

TEX[39] — the document and mathematical typesetting system — is often considered to represent the 'Gold Standard' in automated typesetting due to it's unsurpassed quality of output. The TEXmark-up language defines a Turing-complete macro system that permits the development of user-defined mark-up macros. Unfortunately, TEXmacros are difficult to understand and write, making macro development a task that is undertaken by few. VorTeX [13] was an incremental implementation of TEXthat incrementally updates a typeset document in response to modifications to the underlying source document. It was developed for the purpose of constructing a WYSIWYG TEXeditor. Its success was limited as it could re-typeset a minimum of one page in response to modifications. No smaller amount of content could be incrementally typeset, due to the semantics of the TEXsystem. The full internal workings of TEX, are described in [39]. A functional description of TEX's box-based formula layout algorithms is presented in [28]. TEXutilizes inherited attributes (first described in the context of attribute grammars in [37]) to allow styling attributes applied to an element to effect child elements as well.

The HyperText Mark-up Language (HTML) is the publishing language of the World Wide Web [77]. Cascading Style Sheets (CSS) [75] is a style sheet language that is used to specify style information the control the presentation of structured documents, namely those that are written in HTML or XML [76]. A CSS style sheet consists of descriptions of styles, each of which consists of a selector — to identify the elements to which the style applies — and a collection of styling attributes, which control the appearance, style and layout of the elements. CSS attributes may operate in an inherited or cascading fashion. An inherited attribute applies not only to its target element, but also to the child elements enclosed within it. A cascading attribute operates in a cumulative fashion, inheriting the value from the parent and applying a modification, such as a scale factor.

Modern web browsers provide facilities for implementing dynamic content,

enhancing the capacity of web applications for providing rich user interfaces. This lies behind the recent trend in web development, known colloquially as Web 2.0. Web browsers can execute code that is written in the Javascript programming language [18], and allow Javascript programs to receive user input, and modify the contents of the document through an API.

2.1.2 User interface tool kits

User interface tool kits — such as GTK+ [69] and Java Swing [56] — provide a library of widgets that are instantiated and combined to form a tree structure. Simple widgets such as text labels or images are placed at the leaves, while container elements are placed at the branches.

Modern tool kits perform automatic layout, spatially arranging widgets according to layout rules determined by the containers. Container elements can arrange children horizontally, vertically or in a matrix layout. Modifications to the content of widgets or to the structure of a widget tree will cause the tool kit to incrementally update the layout to account for the changes. These systems typically use a box-based layout system, similar to that found in TEX.

2.1.3 Combinatorial APIs

Direct element construction — as employed in GUI toolkits such as Swing [56] — results in verbose and inflexible code, as it is necessary to configure each element individually. This in part underlies the recent rise in popularity of presentation and GUI description languages such as XAML [33] and QML [55]. The use of a domain specific syntax or XML schema results in presentation descriptions that are terser than the equivalent direct element construction code. They can be further simplified through the use of a style sheet system such as CSS. A further benefit of XML based systems is that their similarity to HTML opens their use to a wider audience, as user interface designers are often familiar with HTML. Unfortunately, by using a presentation language the developer loses the power and flexibility of a general purpose programming language; functions, loops and control structures which can assist in the presentation construction process are not usually available. Within web development frameworks, template languages [16] attempt to address this problem by mixing elements of the host programming language with HTML.

Combinatorial APIs with style sheets bridge the gap between presentation languages and general purpose languages, allowing source code to be used to create terse, logical presentation descriptions, while retaining the full power of the host programming language.

Hopscotch [12] is a document-centric application framework that is a component of the Newspeak [7] environment. Its appearance bears resemblance to that of a web browser, due to its browser-like navigation controls, and the frequent use of hyperlinks. The reduction in code verbosity achieved through the use of a combinatorial API underlies the use of combinator methods within Hopscotch presenters (a presenter constructs a presentation of an underlying data model). Their invocation results in the construction of the necessary elements. This indirect specification of the interface offers significant flexibility, as the layout and appearance of a presentation can be altered by sub-classing the presenter and overriding the necessary methods. The resulting system provides a style sheet

facility, which resembles the operation of the text typesetting system employed by TEX.

XPREZ [63] is the presentation system developed for the Proxima editor [62]. It is implemented in the Haskell programming language [59]. XPREZ is able to produce rich presentations and is sufficiently powerful to cover much of the mathematical typesetting capabilities provided by TEX, as described in [28]. The functionality of XPREZ is exposed to programmers though a combinatorial API. Its design allows simple, understandable code to be used to construct rich document-centric presentations.

Mathematica [86] provides a powerful combinatorial API that can be used to compose presentations from plots, formulae, and user interface controls.

2.2 Object presentation

There are various approaches for object presentation, most of which are based on the MVC architecture.

2.2.1 Model View Controller (MVC) architecture

The Model-View-Controller (MVC) architecture [9] has become the dominant design pattern for the development of GUI driven applications. It describes three components that work together in order to present editable data to the user. The model component encapsulates the application state (the data that is to be edited) and provides an API for accessing and modifying it. It also notifies the view of state changes, so that it can automatically refresh in order to maintain consistency with the model. The view component creates a user interface that presents the data to the user, usually in the form of a GUI. It acquires values from the model that guide the construction of the GUI components. The view passes user actions to the controller. The controller component responds to user actions by making the appropriate modifications to the model.

Many modern user interface tool kits utilise the MVC architecture. GTK+ [69] uses the model/view separation for complex controls such as list and tree views, but not for simple controls such as check boxes and text entries. Swing [56] uses this approach even for simple controls.

The MVC architecture is also used in many web application frameworks (e.g. Django [17]). The data is typically stored in a relational database, with model objects providing an object oriented interface for accessing it. A view that accessible at a specific URL responds to a request from the client by accessing the data model and generating an HTML document that will present the interface to the user when rendered on the client. The client responds to user actions by sending a request to a URL that maps to a controller, that converts the user's actions into modifications to be performed on the data model. The stateless model of the World Wide Web provides a natural fit for MVC, with requests for data being handled by a view and responses being handled by a controller.

2.2.2 Editing environments

Citrus is a language and toolkit designed to simplify the development of graphical editors for structured data and code [40]. Models and views within citrus

are described using the declarative Citrus language. Citrus enforces a 1:1 correspondence between model class and editor description. The tight static binding between data and editor schemas allows Citrus to automatically construct views of data and maintain consistency between them in response to changes. Furthermore, the data model schemas are statically typed and can specify constraints for fields that limit their range of values. These constraints are automatically applied by the editor.

The Ensemble project [24] was an editing environment that operated on generic multimedia structured documents. It was aimed primarily at software development, and was therefore intended to be used for editing source code, user documentation, design notes and project management information. Document models were represented using a tree structure. A variety of presentation systems were developed for Ensemble[1], each using a different technique to present a document. Proteus [51] uses tree elaboration; the document tree is projected to form a presentation tree, with the presentation schema defining additional elements which are inserted into the presentation tree, along with attributes that affect the elements' appearance. A tree transformation based Ensemble presentation system — described in [48] — visits the nodes in a document tree, using re-write rules to transform document tree nodes into presentation tree nodes.

2.2.3 Programming environments

The interactive development environments within Smalltalk systems [22] allow a developer to explore the object graph of a running application and interact with their state. Objects are live, in that they can be inspected by a developer as they form part of a running program. A Smalltalk object inspector normally displays an object in the form of a collapsible tree. This interface is now employed by the GUI-based debuggers that are packaged with most IDEs.

Hopscotch (mentioned previously in section 2.1.3) uses a *subject* to identify the data that is to be presented in the browser window. A subject consists of a location marker to identify the object that is to be presented and a viewpoint that describes how the data should be presented. Hopscotch does not provide support for automatic synchronisation of models and views. The developer must specifically implemented code to update the contents of a view in response to state changes in the model.

Vital [27] is a document-centred environment for the Haskell programming language [59]. It presents values visually, by using type coercion to convert a value into a visual representation. Visuals are composed through the use of combinator functions that construct graphical primitives (shapes and text, with options for colour), combine them by applying transformations (translation, scale, rotation, and flipping), and superimpose them, in order to construct composite diagrams.

[1]The definition of the term *presentation system* in the context of the Ensemble project is slightly different to the one used within Larch; the Larch presentation system encompasses only the system used to construct visual content to display to the user. An Ensemble presentation system must also create and maintain the model-view mapping

2.3 Structured source code editors

The development of structured source code editors has a rich history in the research community Structured source code editors represent code in a structural form (an abstract syntax tree (AST) for example) rather than a textual form. They can be divided into two classes: syntax-directed editors; and syntax-recognising editors.

2.3.1 Syntax directed editors

Syntax directed editors expose the user to the structure of the underlying representation. While they may present the document as text on the screen, they typically use a direct manipulation interface. The user must first choose a node within the document model to edit (nodes normally correspond to syntactic constructs, such as variable access expressions or additions). The user is then presented with a range of modification operations, accessible via GUI controls such as menus, toolbars or keyboard shortcuts. In practice, the direct manipulation interfaces offered by syntax directed editors have proven to be cumbersome to use, and have not achieved widespread use among software developers as a result [42, 80].

Redwood [82] is a syntax directed editing environment in which programs are developed by combining code snippets. It uses a direct manipulation interface in which snippets are dragged from a library into the code pane. Snippets are combined to form a tree, whose structure is similar to that of an AST. A code generator converts the program into Java or C++ source code.

Within the Anastasia editor [31], the user edits functional programs by inserting code segment templates. Anastasia validates functions that are written within it using 'proofs-as-programs'. This work-flow is helpful for beginners who are unfamiliar with the language syntax, but can act as a hindrance to more experienced programmers, who will achieve greater productivity with a text editor.

Syntax directed editors implemented using Citrus [40] (first discussed in section 2.2.2) behave in a fashion that is a hybrid of syntax-directed editing and syntax-recognizing editing. When editing textual elements such as identifiers, literal values or tokens, Citrus behaves in nearly the same fashion as a text editor. Structural modifications can be performed in response to keyboard short-cuts. With carefully chosen short-cuts, the editor will behave in a similar fashion to a text editor. For instance, pressing the '+' key could result in the currently selected expression being placed within an addition expression.

2.3.2 Syntax recognising editors

Syntax recognizing editors provide a free-form text editing interface. The document is represented internally in structural form, but presented to the user in a textual form. Edit operations undertaken by the user alter a temporary textual representation of the document. A parser is used to convert the modified text into a structural representation, which is inserted into the document model. While the internal data model is structural, the editor behaves in much the same way as a text editor.

The Synthesizer Generator [43] displays source code to the user in a textual form. The user can then select a segment of code corresponding to an AST node, at which point it is displayed in a separate pane in plain text form, which the user can modify. The user may only commit the modified code back if it conforms to the language grammar rule that is associated with the selected AST node.

Pan [2] and Ensemble [24] were two editing environments developed at Berkeley in the 1980's and 1990's. Pan was a syntax recognizing source code editor, which improved on prior structured source code editors by permitting unrestricted text editing. Novel incremental lexical and syntactic analysis techniques were developed as part of Pan. Their error handling and recover techniques form the basis of those used in Barista [41] and Larch. Where Pan displayed source code in a textual form, Ensemble permitted the development of richer presentations. It was described in more detail in section 2.2.2.

Harmonia [6] is an incremental source code analysis system, designed to support the development of tools that enhance an existing source code editor. The Harmonia project built on lessons learned during the Pan and Ensemble projects. Harmonia's structural representation retains all information from the textual form so that it can be accurately recreated after structural modifications. It therefore preserves white-space and comments due to their importance for layout and documentation. The structural model is self versioning, with each node maintaining its own edit history. The versioning system allows different versions of a document to be available to different parts of an application simultaneously.

UQ$\star$ [81] is a software document editor in which code is represented internally as an abstract syntax tree. It is presented to the user in textual form, and utilizes a syntax-recognizing editing model. Source code can be viewed at different levels of detail; for example, the user can zoom between the module level, where only function declarations are visible and the function level where only the implementation of the selected function is visible, with other functions hidden. The diagrammatic capabilities of UQ$\star$ (discussed in [36]) allow the development of interactive editable diagrams, composed of graphical and textual elements. Edit operations performed on a diagrammatic representation cause the underlying document to be modified. UQ$\star$ textual and graphical presentations, while synchronized, remain separate. While graphical presentations can contain some textual elements, textual presentations do not contain graphical elements; the two presentation types cannot be freely intermixed.

Proxima [62] is a generic editing environment written in the Haskell functional language. Proxima uses a layered architecture. At each level, the document is represented in a different form, with the document model at the top level and the rendered visual representation at the bottom. Each level represents a different stage in the presentation process. Layers between each level convert data from the level above to the level below and propagate the effects of edit operations upwards. A Proxima editor description consists of a document schema and a presentation schema. The document schema takes the form of statically typed node structure descriptions, implemented in a dialect of Haskell. A presentation schema is described using an attribute grammar. An editor must be combined with the Proxima runtime and compiled with a Haskell compiler.

Barista [41] is a syntax recognizing source code editor for the Java programming language, implemented using the Citrus framework. As with Citrus, mod-

els and editors have a close correspondence. Barista allows a structural model to be edited in a textual fashion by 'fluidly changing code between structured and unstructured text'. Barista editors also offer a direct manipulation interface that allows code to be constructed by dragging and dropping templates, or via auto-complete menus.

Barista takes advantage of its structural data model by offering features that are not normally possible with a plain text editor. Comments can contain rich text and images, eliminating the need for ASCII art. It offers an alternative approach to code folding; instead of hiding a region of code, it can be scaled to half size, thus providing a higher level view of the code without hiding any of it.

2.4 Active documents

A number of active document systems employ a notebook interface, which are common in computer algebra and mathematics systems such as Sage [61], Mathematica [86] and IPython [58]. A notebook consists of a vertically arranged sequence of cells which can contain either text or code. Text cells may support styling. Code is executed with its output displayed in an output cell immediately below.

Sage and IPython support visual output to the extent that they can display plots in output cells. Mathematica takes this much further, providing a combinatorial API for constructing and combining visual elements, which can include plots, user interface controls, typeset mathematical formulae, and more. The new CDF (Computable Document Format) system [85] from Wolfram Research is a publishable document format based on Mathematica. It is worth noting that Sage and IPython use a web browser to present the interface to the user.

Vital [27] — first discussed in section 2.2.3 — is based on ideas from spreadsheet programming, and operates in a similar fashion to the notebook oriented interfaces of programs such as Mathematica [88], and Maple [47]. A document consists of a number of cells, each of which contains a Haskell declaration, and can be accompanied by a comment. Vital declarations (cells) are freely positioned by the user, in contrast to the automatic vertical layout used in Sage, Mathematica, IPython and Larch. This has the benefit of giving the user has precise control over the layout of the document (e.g. allowing for a multiple column layout). Unfortunately, difficulties arise when altering declaration code such that the visual space requirements of its output changes. Vital's presentation system does not perform automatic layout, so it will not rearrange the declarations to prevent them from overlapping or to fill any gaps that are created.

Tilescript [79] is a web-based notebook system which uses Javascript as both its implementation and user programming language. Code can take the form of Javascript text, tile scripts (code composed by dragging and dropping tiles representing Javascript programming constructs) or HTML expressions. The elements created by HTML expressions are available through the DOM API and are therefore accessible to code within code cells; this allows the creation of complex interactive presentations.

Chalkboard [89] is the successor to the Tilescript system. It is designed for the creation of active essays (an active essay — a term coined by Alan Kay

— is a written essay, mixed with a computer program. The program code is executed within the document, providing dynamic content). Chalkboard divides the workspace into three areas: an editor, into which documentation and code can be entered; a transcript which displays any textual output that results from evaluating and executing code within the editor; and a play area, which is an HTML 5 canvas element which is used for displaying graphical output. Chalkboard provides a number of controls, which are used for executing code, and altering the style of rich text.

TreeCalc [68] is an editing environment designed for the construction of programmable structured documents. A TreeCalc document is an XML document, with an associated presentation specification, which describes how the content should be displayed. Documents can contain embedded CDuce [3] code, which can acquire, manipulate and generate content. The nature of these alterations is unrestricted; the authors describe a number of scenarios, including an automated exam system to be used in a teaching environment, in which the system will check the answers entered by a student and assign a grade accordingly.

2.5 Live programming environments

Programmers need to evaluate their problem-solving process at frequent intervals; this is known as progressive evaluation within the Cognitive Dimensions framework [26]. There are a variety of ways in which a programming environment can support this. At the most basic level, a REPL based console allows small snippets of code to be immediately tested. More sophisticated environments such as those found in modern Smalltalk systems [22] allow the programmer to enter a partially complete method implementation. It invokes the debugger in place of yet to be written statements. The programmer may then complete the code within the debugger and test it in place, while having it function as if it was in the context of the original incomplete method. The code must then be copied and pasted back into the original method.

The Javascript environment presented in [35] takes live programming a step further by converting edit operations into transformations that are propagated though the intermediate compiler representations all the way to the interpreter and its call stack, resulting in modifications to live stack frames.

Spreadsheets are of particular interest, as they are widely used by non-expert users. Continuous evaluation provides instant feedback to the user in response to changes, allowing them to rapidly assess their progress.

Forms/3 [11] is a research language based on the spreadsheet paradigm. Like prior systems, Forms/3 supports data types other than numeric values and provides visual presentation capabilities. Cells are not constrained to a grid-based layout as with most spreadsheets; they can be freely positioned, allowing the user to determine the form and layout of a document. Forms/3 can therefore be used to construct complete interactive applications. Forms/3 defines a custom spreadsheet language that supports the definition of new data types and functions. A notion of time and sequences of events allow for the development of animated and interactive content.

SuperGlue [49] is a development environment and language which mixes object orientation and spreadsheet style incrementally maintained values (called signals). Visual presentations of values can be defined, allowing for the develop-

ment of visual, interactive software. The language features are tightly integrated with its development environment, which provides the user with a continuously up-to-date view of the state of the program under development.

2.6 Visual source code extensions

Visual extensions to source code can take two forms: augmented rendering and visual programming constructs.

Augmented rendering alters the appearance of existing constructs to improve comprehension. Barista [41] allows comments to include rich text and images, eliminating the need for ASCII art. Boolean AND expressions and fractions are both vertically arranged for the purpose of improving comprehension. ETMOP [19] allows the development of a pattern matcher that selects specific constructs to be presented differently (e.g. displaying an invocation of a square root method using mathematical notation). Unfortunately, developing a reliable pattern matcher is often challenging since semantic analysis is usually necessary, even for a simple task such as determining the target of a method call.

Visual programming constructs are visual elements that are embedded within textual source code. ETMOP [19] allows such constructs to be embedded within Java code. They are implemented by creating new ETMOP AST node types along with a custom presentation and code generator. Similarly, the Racket environment [60] allows racket boxes to be inserted into textual scheme source code. Boxes evaluate to a value, which depends on their purpose; image boxes evaluate to the image itself, while fraction boxes (displayed as vertical fractions) evaluate to a numeric value.

Mathematica's *generalized input* [87] allows visual output (results from a computation) to be copied and pasted as visual input, as part of a Mathematica expression. The way in which it is evaluated can be customised through the use of the `Interpretation` function. It operates in a similar fashion to a LISP macro, transforming the visual content into values.

Most of the aforementioned approaches are inspired or influenced by LISP macros. Macros are invoked like ordinary LISP functions, except that their arguments are not evaluated; they are passed to the macro in AST form, which processes them and returns a result AST that is to be evaluated in its place.

2.7 Visual programming languages

Visual programming languages (VPLs) exchange the formal syntax of a textual language for visual constructs that are placed by the programmer, usually with a direct manipulation interface. Scratch [46] — a popular educational VPL — uses a drag and drop interface in which programs are constructed by dragging blocks representing familiar programming constructs into a scripting area. They snap together like LEGO bricks, with their shape suggesting how they can be connected to one another, thus providing a convenient and natural interface. Scratch blocks are combined in much the same way as the equivalent textual constructs (statement blocks can be placed within conditional blocks, expressions can be placed into slots within statements or other expressions, etc). Execution and feedback is immediate; Scratch scripts are executed by clicking

on the relevant blocks, affecting sprites that are visible in the stage pane.

Diagrammatic languages — such as LabVIEW [54] — present programs in a completely diagrammatic form. While they have achieved popularity among non-programmers, they unfortunately suffer from usability problems [26], hence their lack of adoption among professional programmers.

The recent popularity of touch enabled tablet computing devices has spurred the development of touch based programming languages. TouchDevelop [71] employs a menu driven direct manipulation interface to create and edit programs which appear as text in their final form. The language structure and syntax is simple, as it is designed as an end-user scripting language for creating simple applications. YinYang [50] is designed to allow children to create simple games. Its language and programming model is 'based on tile and behaviour constructs'.

Direct manipulation programming environments (DMPEs) are often slow and cumbersome in contrast to free-form text editors. JPie [4] introduces approaches to alleviate these issues by allowing the programmer to enter incomplete or incorrect programming constructs, which many structured editors disallow. Statements and constructs can also be edited in textual form.

2.8 Domain specific languages

Domain specific languages are programming languages or specification languages designed for expressing solutions to a specific problem domain. Their syntax and semantics are designed so that solutions can be expressed in a terse and efficient form. They may closely match the form of any existing notation that is already used by domain experts.

Existing DSLs come in many forms, depending on their purpose. QML [55] is a DSL for describing user interfaces to be built with the Qt user interface tool kit. The Django template language [16] mixes Python code with HTML for web page generation. ANTLRWorks [57] is an IDE for developing parsers that uses DSLs for describing lexical and syntactic analysers (DSLs are commonly used within parser generators).

The Intentional Domain Workbench [65] is a commercial system designed for the creation of domain specific languages (DSLs) aimed at non-programmer domain experts. Intentional DSLs can mix textual and visual constructs as needed. The Intentional Domain Workbench is not publically available, so little is known of its internal workings, with the exception of a description of its internal storage model [64]. In [66], Intentional Software demonstrate a pension language, developed in collaboration with pension analysts. Prior to its development the analysts used an informal spreadsheet and word-processor based notation to communicate model designs to a programming team for implementation. The pension language was a partially visual programming language which blended spreadsheet style tables, formulae and executable source code in a fashion that was close to the analysts' notation. It allowed the analysts to implement their models directly.

The Jetbrains Meta Programming System [34] is a programming tool aimed at the development of DSLs. Like the Intentional Domain Workbench, MPS eschews the use of parsers and purely textual languages, opting for a partially visual editor. MPS languages can utilise textual-style source code and GUI

controls.

2.9 Canvas based development environments

Recent research efforts have resulted in a number of development environments in which blocks or modules of code can be freely positioned on a zoom-able, pan-able canvas. As a consequence, a programmer can use their spatial memory to assist in locating a desired code segment. Larch does not currently make any attempt to support such an interface. We discuss these systems here as we consider it to be a valuable line of future inquiry.

Code Canvas [14] allows the user to conceptually group files, classes and methods by freely positioning them on a zoom-able canvas. They can be further contained within visual containers or attached to Photoshop-style layers, whose visibility can be toggled, further supporting the use of spatial memory.

Code Bubbles [8] offers a new and innovative take on the browser interface employed by many IDEs. Method or class implementations are displayed within freely positioned bubbles. Navigating to the target of a method call displays the target method implementation in a new adjacent bubble. As a result, the browsing history is visible as a chain of bubbles in the workspace. Similarly, the Code Bubbles debugger uses bubble chains to represent the call stack.

The Fluid source code editor [15] is not strictly a canvas based environment, as it is an enhanced text editor. It does, however, re-arrange source code in a novel way, providing an in-line exploration interface that arranges segments of source code in context with related surrounding code. For example, method call sites can be expanded to show the complete body of the target method implementation. Task specific source code re-arrangement is a valuable avenue for future work.

Chapter 3

Presentation system

In this chapter, we describe a presentation system called *LSpace*. It is used for constructing interactive visual presentations, which can combine rich text, images, GUIs and interactive documents. Presentations are displayed within a web-browser style interface.

3.1 Presentation tree

Presentations are composed of elements which are arranged to form a tree, with text, shape and image elements at the leaves, and container elements at the branches. Container elements combine and spatially arrange their children in rows, columns, paragraphs (flow layout), and tables. Mathematical containers arrange child elements to form fractions, superscript and subscript. The elements are implemented as a Java class hierarchy.

LSpace uses an automatic spatial layout system that incrementally re-arranges elements in response to modifications. Our layout process uses a four-phase request-allocate algorithm. Many toolkits, such as GTK+ [69] use a two-phase approach. In the request phase, the spatial requirements are accumulated from the leaves of the presentation tree towards the root. The requirements of a leaf element are determined by its content, e.g. the size of a small piece of text. Branch elements combine the spatial requirements of their children according to branch layout rules, e.g. row elements accumulate space horizontally, where columns accumulate vertically. This process continues to the root of the tree. The allocation phase divides the space available — normally determined by the size of the window — among the elements in the presentation tree. A branch element will divide its allocated space among its children and position them according to its layout rules, e.g. row and column elements respectively arrange their children horizontally and vertically. We use a four-phase layout approach — request and allocate horizontally, then request and allocate vertically — in order to support flow layouts. A flow layout splits its sequence of children into multiple lines when there is insufficient horizontal space available to arrange them into one line. Its vertical space requirements therefore depend on the number of lines required, which in turn depends on the amount of horizontal space available, which is only available after the horizontal allocation phase is complete.

3.2 Presentation combinators and style sheets

Direct element construction and configuration is a cumbersome and low-level process that requires long-winded and inflexible code [12]. As a consequence, developers are encouraged to use the presentation combinator API to construct LSpace presentations.

Presentation combinators act as a declarative description from which concrete presentation elements are built. The appearance of elements is controlled by using an extensible style sheet system that offers inherited style attributes (style attributes which propagate from parent to child, as in TEX[39, 28], HTML, XPREZ [63]).

LSpace provides a library of primitive combinators, which have a close correspondence with the underlying presentation elements (the `Text` combinator creates an `LSText` element[1], the `Row` combinator creates an `LSRow`, etc). Presentation combinators are compose-able. They can be combined to form more complex combinators. The rich text and GUI control combinators are all implemented in this way. Programmers are encouraged to implement their own in the same fashion. Presentation combinators are implemented as a Java class hierarchy, with the abstract base class `Pres` at the root.

An example of the presentation combinator and style sheet API is shown in figure 3.1. It presents a small Python program that constructs a visual explanation of a Gaussian blur, along with the result of executing it. The source and result images are pre-computed and saved as image files; the code presented constructs the diagram, it does not perform a blur operation. The blur kernel is converted from a 2D list of floating point numbers to the 2D (grid) representation in the centre. The images and the blur kernel are presented as figures (combined with a caption below) and arranged with arrows. The following code was elided for brevity: 6 lines of import statements; 3 mathematical function definitions (3, 2 and 3 lines) and 1 function call to build `kernel` (a 2D list of floats).

3.3 GUI controls

A range of common GUI controls is provided. They can be incorporated into presentations alongside other content. All of the controls are constructed using lower-level combinators. LSpace provides buttons, hyperlinks, check boxes, option menus, text entry boxes, numeric entry boxes, multi-line text areas, and scroll bars.

3.4 Incremental modification

LSpace is designed for the construction of editable documents. Modifications to a presentation frequently entail altering the structure of the presentation tree. LSpace supports efficient incremental structural modification by allowing presentation fragments to be replaced, instead of entire sub-trees. A tree fragment F is defined as the set of nodes (elements) that are within a sub-tree S, but

[1] Presentation element class names are prefixed with `LS` and derive from the `LSElement` base class.

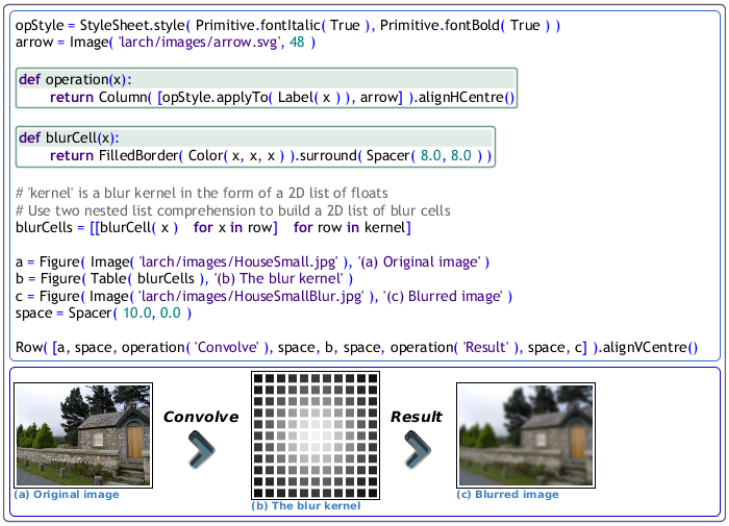


Figure 3.1: Presentation combinator and style sheet API example

not within sub-trees $T_0...T_N$, where $T_0...T_N$, are contained within S. Fragment replacement is accomplished by (1) creating F', a small tree of new presentation elements to replace those in F, (2) adding the roots of $T_0...T_N$ as children to the appropriate elements within F', and finally (3) replacing the element at the root of S (that is also the root of F), with the element at the root of F'[2].

3.5 Event handling

3.5.1 User input

The LSpace event handling system follows an established design pattern of separating input handling from the main UI tool kit, as in [52]. In this way, support for new user actions, interaction techniques and input devices can be implemented without needing to modify the presentation tree implementation.

Application code needs to be informed of user actions that affect specific elements, so that it can respond appropriately. All presentation elements maintain a list of action handlers that receive user action events. A action handler is a Java interface that defines methods that receive events relating to a gesture. Each type of handler handles a different kind of action (e.g. button push/release, pointer motion, drag, etc).

Each supported input device has a representation within LSpace. The process of input handling starts with a device, e.g. a mouse pointer. The device processes the input and detects actions (e.g. drag). At the beginning of the action, the presentation tree is traversed to find the element to which it applies. For a mouse pointer, the element under the pointer is found. For keyboard input, the element containing the caret is used. Presentation elements along the path from the target element towards the presentation tree root are tested,

[2]It is worth noting that if possible, layout information is retained when a fragment is detached and later re-used

until an element is found that has a action handler of the appropriate type (e.g. a drag handler). Subsequent action events are sent to the handler, until the action ends.

3.5.2 Application events

LSpace allows arbitrary application generated events to be emitted at specific presentation elements. Elements maintain a list of application event handlers. When an application event is emitted, LSpace tests each element along the path from the source element to the presentation tree root until one is found that has a handler that successfully consumes the event.

3.6 Structured document support

LSpace is designed for presenting structured documents, in contrast to plain or marked up text, e.g. HTML. Within Larch, structured documents are normally represented by normal Java and Python objects (this will be discussed in chapter 4). In aid of this, structural values (objects) can be associated with presentation elements, and later retrieved, allowing application code to request the data represented by a specific part of a presentation.

3.7 Targets, selections and regions

LSpace uses targets and selections to represent the choice of content that will be affected by user edit operations. The current target is the point in the document at which edit operations are performed. The selection is the range of content that is affected by copy and paste operations. LSpace's target and selection system is extensible; new types of target and selection can be implemented for new forms of context and their associated styles of interaction.

When editing textual content, the current target is a caret; the point at which text typed by the user is inserted. It is rendered visually as a blinking vertical bar. The selection is a highlighted range of sequential text.

When editing content displayed within a spreadsheet style table (see section 5.3) the target is the chosen cell, while the selection is a rectangular block of cells.

LSpace is designed to allow a presentation to mix different types of content. For example, a presentation may consist of a rich text document, into which is embedded a block of source code (stored in a structural form; see section 5.1) and a spreadsheet style table. In this example, the table uses block-style selections and targets, while the rich text and source code use text-style sequential selections and targets. Additionally, the rich text and the source code use different document schemas and corresponding editors; a rich text editor and a structured source code editor.

In aid of allowing different content types to reside within the same presentation, LSpace allows a presentation to be divided into regions. A region is denoted by placing a region element at the root of a presentation sub-tree that is to reside within it. A region element is given a clipboard handler whose task is to handle copy and paste operations that affect the content within it. When the user invokes a copy or paste action, the clipboard handler is retrieved by

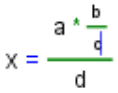


Figure 3.2: Fraction presentation example

acquiring the region that contains the target or selection that is to be affected. The clipboard handler performs the necessary modifications to the underlying data model.

3.8 Caret behaviour

Constructing useable, editable presentations that are richer than styled sequential text presents additional challenges concerning the behaviour of the caret. The caret must move through the presentation in response to user actions (mainly the use of the cursor keys) in a way that maximises usability.

To illustrate this problem, let us consider the shortcomings that become apparent with the most basic approach to managing the caret; positioning it with respect to characters that form the textual content of the presentation tree. This would work for plain text, but would fail for more complex layouts, e.g. mathematical fractions. Figure 3.2 shows a nested fraction presentation, the textual representation of which is `x = a + b/c/d` (note that parentheses are missing as they are not present in the visual form). The caret is shown just to the right of `c`. Imagine that the user moves the caret one place to the right. Moving it one character to the right will position it between the last `/` and the `d` in the denominator. This would not allow the user to insert additional content into the outer numerator after the nested fraction. The correct behaviour requires that the caret stop just to the right of the nested fraction, while still in the numerator. At this point, the position of the caret is visually distinct from its original position, while its position relative to the textual content has not changed.

The desired behaviour requires that the caret should be able to be positioned at the start or end of any horizontal span of content within the presentation. Often, a number of horizontal spans can start or end at the same boundary between two characters from the textual representation.

The desired behaviour is achieved through the use of caret slot elements and segment elements; presentation elements specifically designed to control caret motion. Segments are simple containers that surround their content with a caret slot on either side. Caret slots are empty elements which can stop and capture the caret on its way to the next character. Caret slots will stop the caret if it is moving from one segment into another. By wrapping the numerator and denominator of a fraction in segment elements, the desired behaviour can be achieved. Segments can also be employed to control caret behaviour in the context of other visual constructs.

3.9 Editable text elements

LSpace text elements can be marked as editable using a style sheet attribute. An

editable text element that contains the caret will respond to keyboard input from the user by inserting or deleting the appropriate characters from the text that it contains. It will also emit an application event (section 3.5.2) that describes the edit operation. These events are handled by the application, which responds to the user action in the appropriate way. The structured source code editing system (section 5.1) and the rich text editing system (section 5.2) both respond to text edit events by initiating updates.

3.10 Implementation

3.10.1 Presentation combinators

Presentation combinators are implemented as factories. The presentation combinator class hierarchy is rooted at the abstract base class `Pres`. It defines a method called `present` that builds the concrete presentation elements that are described by the combinator.

Presentation combinators are constructed and combined so that they form trees. In the final line of listing 3.1, the `Text` combinators are combined using a `Column` and a `Row`. The resulting combinator will retain the tree structure built within the code. When the `present` method of the `Row` combinator is invoked, it will invoke the `present` methods of the child combinators and then place the elements that they create within an `LSRow` presentation element.

```
1 t0 = Text('Hello ')
2 t1 = Text('world. ')
3 t2 = Text('The end')
4 helloWorldTheEnd = Column([Row([t0, t1]), t2])
```

Listing 3.1: Presentation combinators forming a tree

When a presentation combinator constructs a concrete presentation element, it will configure the element in order to control its appearance (font, size, colour, etc.). The `present` method takes a presentation context (discussed in section 4.12) and a *style value table* (discussed below) as parameters. It acquires values for configuring the element from the style value table, by accessing the values for the appropriate style attributes.

3.10.2 Style sheets

The design of LSpace style sheets is inspired by CSS [75] and the style system within Xprez [63]. Our system uses two kinds of table; style sheets and style values tables, both of which map style attributes to values.

Style sheets and style value tables are immutable. Attribute value immutability ensures that a style sheet cannot be modified after construction, thus ensuring that LSpace does not need to modify a presentation in response to style modifications. Attribute values are altered in the same way that objects are modified in a purely functional programming language; a *base style sheet* is cloned, creating a *derived style sheet* which has the necessary attribute value modifications applied.

An example can be seen in listing 3.2. `StyleSheet.instance` is the root, empty style sheet. `Primitive.foreground` is the foreground colour attribute, declared within the `Primitive` class.

```
redStyle =
    StyleSheet.instance.withAttr(Primitive.foreground,
    Color.RED)
```

Listing 3.2: Creating a derived style sheet

Style value tables are used by the presentation combinator system to propagate and accumulate the effects of style changes through a presentation tree, from parent to child (the style value table is passed by a combinator's `present` method to the `present` method of child combinators). They are the mechanism through which styles are inherited.

A style sheet is applied using the `ApplyStyleSheet` combinator. Its `present` method takes the style values table passed as a parameter and creates a derived style values table, taking attribute values from the style sheet that it applies. The derived style values table is passed to the child combinator, causing the style sheet to be applied to the child presentation. For the purpose of convenience, the style sheet class defines the `applyTo` method that creates the `ApplyStyleSheet` combinator, as seen in listing 3.3. The style sheet applies red foreground to the text 'Hello world' but not to 'The end'.

```
t0 = Text('Hello ')
t1 = Text('world. ')
t2 = Text('The end')
helloWorldTheEnd = Row([redStyle.applyTo(Row([t0, t1])),
    t2])
```

Listing 3.3: Applying a style sheet

3.11 Comparison to existing work

The design and operation of LSpace presentation elements is similar to that of the systems discussed in section 2.1. Our spatial layout algorithm was originally inspired by the simple two phase approach used in GTK+ [69]. We first extended the approach to use four phases in order to support flow layouts (as stated in section 3.1). Later, it was enhanced by allowing the horizontal request phase to compute both minimum and preferred horizontal space requirements. This was inspired by Java Swing [56] in which minimum, preferred and maximum sizes are used. Our element alignment strategy is inspired by Xprez [63], in which horizontal and vertical reference points are computed for alignment purposes, although LSpace only uses vertical reference points.

The design of our combinatorial API was influenced by Hopscotch [12] and Xprez. We started out using direct element construction. This was improved upon using a combinatorial method approach inspired by Hopscotch, and later improved again by the use of presentation combinators and style sheets, inspired by Xprez. An example of a commercially successful system that employs a combinatorial presentation API is the Mathematica computer algebra system [86].

Unfortunately, the combinator based systems presented so far do not work with mainstream programming languages (although Haskell comes close, given its popularity within the functional programming community). This leaves many software engineers out in the cold. Our system leverages this approach and demonstrates its feasibility for popular object-oriented languages. The main benefit of our approach however, is the integration with the object presentation system, effectively giving the programmer MVC functionality, almost for free.

The upcoming HTML 5 standard [77] would appear to be a suitable presentation system for an environment such as Larch. The new `contenteditable` attribute directs a web browser to allow the user to edit the content of tags to which it applies. Unfortunately, its behaviour (in terms of how a document is modified and the behaviour of the caret) is not precisely defined; the application would need to account for the variety of behaviours that exist among the various browser implementations. This would significantly complicate the implementation of a syntax recognizing editor (see section 5.1); this was one of the most challenging components of Larch. Additionally, the application would have to be split into two parts: client side and server side, with careful consideration given as to where the separation should lie.

Chapter 4

Type coercion based object presentation

In this chapter, we describe a type coercion driven approach to presenting normal Java and Python objects. It is integrated with the presentation combinator API presented in the previous chapter. This system enables the development of interactive visual representations for objects that are required for partially visual programming (chapter 7) and for the active document based programming environment which we have implemented within Larch.

4.1 Overview

Our object presentation system uses type coercion to drive the object presentation process. Views of objects are automatically created, destroyed and maintained in response to state changes. Perspectives are responsible for choosing which kind of view to use for a given model, and can be said to represent the intention of the view (e.g. view, edit, or debug). In effect, they perform a form of dispatch; they may use an if-else block or invoke a method on the model object, depending on their implementation (section 4.6).

4.2 Implicit type coercion

Type coercion is the process by which an entity is converted from one data type into another. It is the basis for Java's `toString` protocol, and its equivalent in other programming languages (e.g. `__str__` in Python). Vital [27] used the rich type system of the Haskell language [59] to provide a type coercion based method of visually presenting Haskell values. Our system seamlessly integrates this approach with the presentation combinator system described in section 3.2.

Support for implicit (automatic) type coercion augments our presentation combinator API by allowing normal Java and Python objects to be directly incorporated into presentations. For instance, the objects `x`, `y`, and `z` within the expression `Row([x, y, z])` will be automatically coerced into visual form[1], and

[1]In our implementation, general objects are coerced into presentable form by wrapping them in a special combinator, called `InnerFragment` (see section 4.12.1), whose `present`

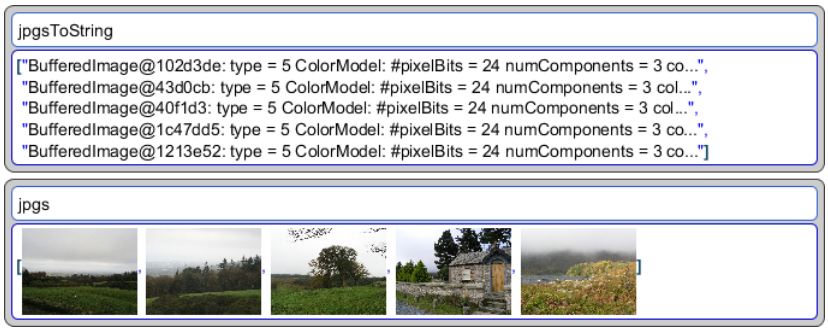


Figure 4.1: Python list containing Java `BufferedImage` objects, displayed in a Larch console

arranged horizontally by the `Row` combinator. It is not necessary to explicitly create views of `x`, `y`, and `z`, as is required in many MVC based systems. As a result, presentations can be composed of normal Java and Python objects, with presentation combinators used for basic content and spatial arrangement. While the code in figure 3.1 composes the diagram using only presentation combinator types, one could replace the figures that are placed in the variables `a`, `b` and `c` with normal objects; the object presentation system would automatically coerce them to presentation combinator types for display.

The coercion process is applied recursively; an object may use other objects to define its presentation, each of which in turn will undergo coercion. We use the term *presentation fragment* (see section 3.4) to describe the set of elements created to represent an object. Figure 4.1 shows five Java `BufferedImage` objects in a Python list. The list is presented with its items separated by commas, and surrounded by brackets. Each item is presented individually; the image objects are presented as thumbnails. A textual representation is shown at the top of figure 4.1 for contrast.

It should be noted that the type coercion system does not affect presentation combinators; objects that are instances of classes that derive from `Pres` (the presentation combinator base class) are not affected. As a result, a `Text` combinator will always create text, and a `Row` combinator will always arrange its children in a row.

4.3 Incremental consistency maintenance

During the type coercion process, an automatically maintained model-view relationship is established between the object and its visual representation. Visual representations of objects are updated automatically in response to state changes. An object notifies the system of state changes by creating an incremental monitor and informing it when the object's state is accessed (during presentation) or modified. A dynamic incremental computation system (section 4.4) — which drives automatic incremental consistency maintenance — propagates state changes throughout the application, scheduling refreshes for appropriate parts of a presentation.

method establishes a model-view relationship between the object and its visual representation.

Our use of automatic incremental consistency maintenance relieves the programmer of the need to update views (normally by mutating presentation elements) in response to model state changes, as is necessary in many MVC implementations. While not an especially difficult problem, it can be time consuming to implement reliably. In contrast, objects presented by Larch need only report that a state change has occurred to trigger a refresh, during which elements within its corresponding presentation fragment will be removed and replaced.

Automatic consistency maintenance can be disabled for specific fragments (views of objects) where it is desirable to maintain state across modifications (state which would normally be lost when a fragment is re-created). This is only necessary in particular circumstances, such as responding to a continuous user gesture.

4.4 Dynamic incremental computation system

Incremental consistency maintenance is driven by a dynamic incremental computation system, whose operation is similar to that of a spreadsheet. It is used to propagate the effects of state changes throughout the application and schedule updates. This simplifies the construction of interactive applications, since modifying a value in the underlying document model will cause change events to propagate through derived values (incrementally maintained values computed using other incrementally maintained values), until they reach the user interface, at which point a refresh will be requested. Conceptually, this model is very similar to that of a spreadsheet; a spreadsheet cell can contain either a literal value, or a formula in the form of a mathematical expression that computes a derived cell value, using the values of other cells that are referenced by the formula. Modifying a cell in the spreadsheet causes the values of all formula cells that utilise its value to refresh their values, thus restoring consistency.

Our system dynamically maintains a dependency graph, through which changes are propagated. Our implementation consists of two incremental monitor classes; `IncrementalValueMonitor`, and `IncrementalFunctionMonitor`, used to monitor values and functions respectively. Informing an incremental monitor each time its associated value (evaluation result in the case of a function) is accessed allows the system to discover computational dependencies. Informing an incremental monitor of a modification results in change events being propagated though the dependency graph, and listeners being informed that a refresh is necessary.

Conceptually, one can consider the visual representation of an object to reside within a spreadsheet cell whose formula creates the presentation. This cell will be refreshed when Larch is informed of modifications to cells on which it depends (the objects whose states are accessed in order to construct the presentation).

4.5 Live values and functions

Live values and functions build on the lower level incremental monitors provided by the dynamic incremental computation system. They are conceptually equivalent to spreadsheet cells that contain either literal values or formulae,

respectively.

A live value is a mutable container that contains a value; the value may be accessed and modified through the use of get and set methods. A live function uses a function (either a Python function or a Java object that implements a single method interface) to compute its value. Should that function access the value of any other live value or live function, a computational dependency will be discovered and established between them. As a consequence, modifying live values and live functions will cause change events to propagate through the dependency graph, scheduling updates; they operate in the same fashion as spreadsheet cells.

Visually, they are represented as the value that they contain or evaluate to. They do not affect the visual appearance of their contents at all. Modifications will result in their visual representations being refreshed. As a consequence, they operate much as spreadsheet cells, but without the constraint of being placed within a grid (as with Forms/3 [11]).

The `LiveValue` and `LiveFunction` classes both derive from the presentation combinator base class `Pres`. This was done to ensure that the type coercion process does not affect them[2].

Programmers are encouraged to use incremental monitors, live values and live functions within their applications to simplify the process of developing interactive interfaces. Larch uses them within GUI controls (section 3.3), the programming environment (chapter 6) and proof of concept tools (section 8.1).

4.6 Perspectives

The MVC architecture allows multiple view types to be implemented for the same type of model, with each view type representing a different intention (e.g. view, edit and debug). We use different *perspectives* to achieve the same ends. Perspectives are applied in a presentation description when the programmer wishes to change intent; applying a perspective `p` to a presentation expression `x` (with the code `p.applyTo(x)`) will cascade through all the recursive applications of the type coercion process that result from presenting `x`, causing them to use `p`. Perspectives can therefore be said to operate as inherited style attributes. This is in contrast to typical MVC architectures, which require the programmer to explicitly create the right kind of view, in terms of both model type and view intention. As an example, let us consider the process of viewing a model that describes a person (`PersonModel`), which contains a model that describes an address (`AddressModel`). When using a typical MVC architecture, a `PersonView` must be created to view the person. The `PersonView` will in turn explicitly create an `AddressView` to view the address. If we intend to edit the person, a `PersonEditView` will be created, which will in turn create an `AddressEditView`. In contrast, one would declare two perspectives — a view perspective and an edit perspective — one for each intent. The view perspective would create (the equivalent of) the `PersonView` or `AddressView`, depending on

[2] A prior implementation had them implement the `Presentable` interface, which would allow their contents to be displayed when using the default perspective (see section 4.6). Unfortunately, this meant that they would not work when other perspectives were used, forcing the programmer to switch perspectives in order to display live values and functions. Deriving from `Pres` eliminates this inconvenience.

which model is being presented. The edit perspective would create the corresponding edit views. The implementations of the person views would not need to explicitly create the view of the address. Incorporating a reference to it within their presentation description invokes the type coercion process, that uses the perspective currently in use to create the desired view.

In our implementation, a perspective provides a method that takes an object as input and returns a presentation combinator as output. The method is invoked during the type coercion process, to create the presentation combinators that describe an objects visual representation.

Programmers familiar with object oriented (OO) programming are accustomed to using methods and inheritance to define object behaviour. *Object presentation perspectives* provide an OO approach for presentation by delegating the responsibility of presenting an object to a method defined by it. For classes that cannot provide a presentation method (due to being defined within standard or external libraries, and therefore not modifiable), an alternate approach is available. Object presentation perspectives will use the object's class as a key to lookup[3] an object presenter (a function that defines a presentation method external to the class). If no object presenter is found, an object presentation perspective will fall back on another perspective.

The *default perspective* is used within Larch to present an object by default, unless a different perspective has been specified. It defines the `Presentable` Java interface and `__present__` Python method for implementing presentation methods. If no presentation method can be found, it falls back on the inspector perspective (see section 6.5.3), which displays the contents of an object's fields in the style of a debugger.

4.7 Functional and compositional approach to GUI development

The design of our object presentation system encourages an approach to user interface development that is both functional and compositional. The functional characteristics derive from our approach to incremental consistency maintenance, whereby changes to an object's state causes its presentation fragment (section 3.4) to be rebuilt from scratch by re-invoking the presentation process.

The compositional characteristics are due to the way in which the developer is encouraged to implement a presentation for an object, which can be used within the presentation of a containing object, and so forth; user interfaces can be composed from smaller parts in a piecewise fashion.

4.8 Caret behaviour

Caret behaviour must be carefully handled when implementing editors for textual content within Larch. Both our source code editors (section 5.1) and our rich text editors (section 5.2) use the object presentation system to present a structured data model. As stated previously, our system encourages a functional approach to presenting objects, in which fragments of the presentation tree are

[3]Inheritance rules are applied, so that if a presenter is not found for the object's class, its super classes will be tested.

removed and replaced in response to data model changes. In instances where the caret resides within an element that is to be replaced, a new position must be found within the replacement content.

Consider the fraction shown in figure 3.2. Imagine that the caret is in the position shown, and that a modification to the data model representing the outer fraction results in its presentation being removed and replaced; only the left hand side of the assignment statement remains in place. Given that the presentation element that contains the caret has been removed, a new position for the caret must be found. In aid of usability, the caret should maintain its position with respect to the content in which it is placed; if it should disappear or move to an unexpected position, the user's work-flow will be disrupted.

We maintain the caret's position by creating textual representations of the old and new content (created by concatenating the textual content of the elements), and using the Levenshtein distance algorithm [45] to generate a set of differences between them. Given the position of the caret within the old content, we use the differences computed to offset the caret position so that it is relative to the replacement content. We then place the caret at this new position. In contrast, Proxima [62] places the caret at the point closest to its original physical position.

Automatic caret maintenance allows the programmer to implement an interface that frequently replaces content, without needing to be concerned with disrupting the caret. The majority of GUI tools and web browsers would require the programmer to manually intervene in order to maintain the position of the caret during such modifications.

4.9 Browser navigation; subjects and locations

Larch mimics the appearance of a web browser, with a location bar, and forward, back and reload buttons. Locations — entered into the location bar — are used to identify the subject that is to be displayed. A subject contains a focus (the data model object that is the target), the perspective used to present the focus, and a title that is displayed by the browser window title bar. A location is a text string, in the form of a dotted identifier. It is evaluated in much the same way as the equivalent Python expression[4] would be. In some cases, the location expression evaluates to a value that is not a subject. If it is a browser page its contents are displayed. Otherwise, the default perspective is used to construct a presentation of the value.

4.10 Change history

The change history provides undo and redo functionality by recording a sequence of changes. A change represents a reversible operation that alters the application state.

An object that supports the change history protocol is said to be track-able. A change history can be asked to track a track-able object. From then on, the

[4]Creating the appearance of nested content is a simple matter of adding attributes to the subject objects. For example, given a subject `X`, accessible at location `x`, setting the attribute `y` of `X` to the subject `Y`, will make `Y` accessible at `x.y`.

object will notify the change history of changes, which it will record. A tracked object may also ask the change history to start or stop tracking child objects as they are added or removed.

When the user invokes the undo or redo commands, the change history will revert or perform the approach changes within its sequence, restoring tracked objects to the appropriate state.

Having tracked (data model) objects assume the responsibility of notifying the change history of state changes removes the need to implement this functionality within the GUI. This approach would require the developer to implement a change type for every type of user action, and add them to the change history as they are performed.

4.11 Clipboard behaviour

Our object presentation system allows normal Java and Python objects to be used as the data model within document centric applications. We allow objects to be embedded within rich text documents (section 5.2), source code (section 5.1, chapter 7) and editable tables (section 5.3). As a consequence, the user is able to duplicate and move them throughout the document using copy and paste operations.

How an object should react to copy and paste depends on its purpose. Some objects are designed to share an underlying piece of data that should be copied by reference, while others should be deeply copied. In aid of this, we provide a clipboard copier protocol that allows objects to control their behaviour in response to copy and paste operations. The protocol consists of a single method (in the form of an interface for Java objects or a named method for Python objects) which is invoked in order to create a 'clipboard copy' of the object.

4.12 Implementation

4.12.1 Presentation combinator integration

The object presentation system is tightly integrated into the presentation combinator system described in section 3.2.

Objects are presented through the use of the `InnerFragment` combinator. `InnerFragment` keeps a reference to the data model object that it is to present. When its `present` method is called, it invokes the presentation process. This creates elements that display the object and returns them.

Finally, `Pres` — the presentation combinator base class — defines a static method called `coerce`, which coerces an object into a presentation combinator. If the given object is already a combinator (it is an instance of a class that inherits from `Pres`), it is returned as is. Otherwise, it is wrapped in an `InnerFragment` combinator. All presentation combinator class constructors take object types (as opposed to presentation combinators) as parameters and pass them through `coerce` in order to convert them into visual form; this is the mechanism for our automatic type coercion.

4.12.2 Presentation process

The `present` method takes a presentation context as a parameter (as stated in section 3.10.1). Besides some internal state, it contains a reference to the current perspective.

When the presentation process is invoked (from within the present method of `InnerFragment`) it uses the supplied perspective to create a presentation combinator that describes the visual presentation of the object (data model). During this process, any computational dependencies between the data model and the presentation are recorded, so that the appropriate parts of a presentation can be updated in response to state changes. The present method of the new presentation combinator is called in order to create concrete presentation elements. These elements are inserted into the presentation tree.

4.13 Comparison to related work

A core benefit of the MVC architecture is the separation between data model and view logic. Our design sacrifices this due to (often) placing the presentation method in the same class as the model. We made this compromise as our main priority was to reduce the complexity of implementing visual presentations by taking an approach inspired by `toString` methods.

Figure 4.2 shows our object presentation system in action. Two classes are defined (14 lines of import statements were elided for brevity). `AnnotatedImage` displays an image with an annotation. Five of these can be seen in the example at the bottom. Note that its `__present__` method sets up a drop target allowing the image to be changed by dropping a file from an external application. The `ImageCollection` class displays a list of images in a flow grid arrangement, below a 'Drop images here' prompt. It sets up drop targets for files and for objects from elsewhere within Larch. Files dropped onto the prompt will create an `AnnotatedImage` for each file and add it to the list. Objects dropped will be added to the list.

At the start only the prompt was visible. The user dropped five images; Ireland1.JPG to Ireland5.JPG. The arrow.svg file was dropped into the first image, changing it from Ireland1. After that, the user dropped a simple Python object, followed by a cellular automata object, both from different demo applications. Note that after each action the user interface was refreshed to reflect the changes made. The cellular automata object defines a `__present__` method that displays an animated, evolving cellular automata; this animation continued to run in the image list. The simple Python object did not define a `__present__` method. The default perspective was therefore unable to display it, so it fell back on the inspector perspective which created the debugger style object inspector. `ImageCollection` does not need to implement any logic for displaying content besides `AnnotatedImage` instances (or even logic to display `AnnotatedImage` instances for that matter); when the items in the images list (which may not even be images) is passed to `FlowGrid`, they are automatically coerced to visual form. Deciding how to display them (invoke `__present__` or display an object inspector) is handled by the perspective.

There are similarities between our object presentation process and tree transformation process described in [48]. Instead of using re-write rules to transform

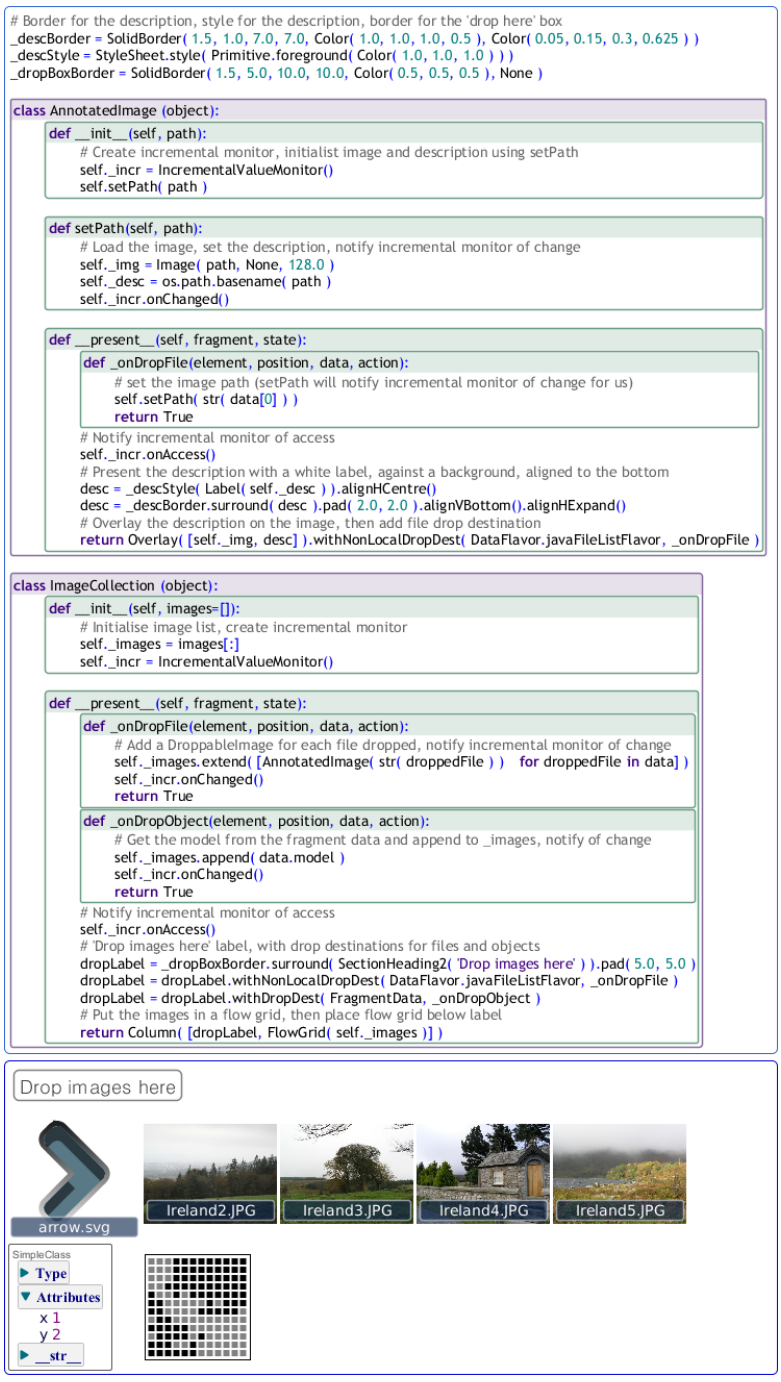


Figure 4.2: Image collection implementation and example.

an Ensemble tree node into a presentation tree node, Larch uses normal Java or Python code to transform a normal object into a combinatorial presentation description. Ensemble presentation systems also defined custom languages for defining presentation schemata. These languages typically disallowed generalised recursion and iteration, therefore allowing full static analysis of their operation and semantics. This allowed computational dependencies (used for determining which parts of a presentation are to be updated in response to a modification to the document model) to be determined statically, therefore optimising processor and memory usage. Larch permits the programmer to use the full range of capabilities of Java and Python, making this kind of static analysis almost impossible. As a consequence, we maintain computational dependencies dynamically, at a cost to memory usage and performance.

In contrast to systems such as Citrus [40] our design sacrifices the benefits of the tight coupling between model and view, but provides far more flexibility. There is no enforced correspondence between model and view; deciding how to present an object is handled by the perspective and can therefore be controlled by the programmer. Perspectives provide a lot of flexibility in determining how an object is displayed: they may delegate to a method implemented by the model (such as `__present__`); use the model's class to look up a presentation function; or use a simple if-else block if the situation requires it. As a consequence, the programmer is largely free to determine the design of their system, without constraints placed by the presentation system. In fact, one can even develop visual representations for objects provided by the standard library.

The use of popular object oriented programming languages for presentation description eases the learning curve faced by developers as they no longer need to learn a new programming language. It also avoids the tricky problems that face language designers, namely those of designing appropriate language syntax and semantics, and developing a sufficient standard library.

The design of Hopscotch influenced the design of Larch in two important ways. Hopscotch viewpoints are notionally equivalent to Larch perspectives; different viewpoints present an object in different ways. Larch subjects have the same purpose as Hopscotch subjects; to identify the data that is to be presented within the browser window.

Systems (such Vital [27] and Proxima [62]) that are developed in purely functional languages such as Haskell do not require consistency maintenance on a per value/object level, since all values are immutable. It is worth noting that presentation by type coercion fits the Haskell language naturally due to its pattern matching constructs and powerful static type system. These are the main reasons why Vital's approach to presentation by type coercion is simpler than our own.

The primary factors motivating the design of our system were code brevity, and providing the programmer with the freedom and flexibility to design their data models and views as they see fit. A combination of type coercion, automatic incremental consistency maintenance and presentation combinators achieves these goals. Our approach is also sufficiently flexible to be implemented for other platforms; we have developed a proof of concept implementation of our object presentation system that runs as a web application.

Chapter 5

Rich content editing

We provide three sub-systems to support the development of structured content editors for source code, rich text and spreadsheet style tables.

Each of these sub-systems was designed to support the programmer during the development of the relevant kind of structured editor, while placing as few constraints as possible on the design of the underlying data model and the visual representation.

In terms of design, all three sub-systems are data model agnostic; the programmer can design the data model to suit their needs. Source code editors and rich text editors also require the programmer to implement the visual representation; in MVC parlance, only the controller is provided. Both of these systems translate user actions into a sequence of modifications to be performed on the data model.

5.1 Structured source code editing system

Larch provides a framework for implementing syntax recognizing structured source code editors that mimic the appearance and behaviour of a text editor.

5.1.1 Approach

Overview

Our editors mimic the behaviour of a text editor by employing a syntax recognizing approach, that changes code between a structural tree model, and unstructured text as needed, in a similar fashion to Barista [41]. A similar approach was used in Proxima [62].

Parsing library

The Larch parsing library is a recursive descent parser, which permits grammars to be described using the Parsing Expression Grammar (PEG) formalism [21]. The PEG technique has been extended to permit left-recursive grammars, and uses Packrat style memoization [20].

In contrast to other systems we do not use an incremental parser to maintain consistency between edited text and corresponding AST nodes. Our parser is

able to use any rule within the grammar as the start rule, allowing our editors restore consistency by applying the grammar rule that corresponds to the AST node type. Like Barista, Larch reattempts the process at the parent node when the parse fails.

For the purpose of permitting language extensions we provide an object oriented grammar system whose design is based on that of OMeta [78]. Grammars are declared as classes and rules are declared as methods, allowing a language to be extended by sub-classing a grammar and overriding the appropriate rules[1].

Basic operation

The object presentation system (section 4) constructs an incrementally maintained presentation of the model, in which most of the presentation fragments correspond to AST nodes.

The editing process is initiated when the user enters or deletes some text, thus altering the content of the presentation tree. The editor must now modify the document model in order to restore consistency with the altered presentation. This is done by attempting to parse the presentation content, converting it to a structural form, which can be inserted into the model.

The presentation tree sends edit notification events (section 3.5.2) in response to edits performed by the user. *Editable fragments* respond to these events by invoking the consistency restoration process; they acquire the content of the presentation sub-tree (in the form of a rich string; a string which mixes text and objects) rooted at the fragment, and parse it using the appropriate parser rule. The choice of rule is normally determined by the type of AST node that is represented by the editable fragment.

Rich strings will contain structural items (objects) for parts of the presentation sub-tree that are not affected by an edit operation and textual content for parts that are. Structural items within a rich string can be processed by the parser with far less work than would be required to process the equivalent textual representation, thus reducing the computational cost involved in parsing large sub-trees of a document. Barista uses a similar optimisation [41].

Syntactically invalid content

Prior research has demonstrated that programmer productivity is impeded by editors which disallow syntactically invalid content [2]; while editing, programmers frequently take source code through a number of invalid states, before arriving at the final valid state. Larch source code editors support syntactically invalid content by defining a special node within the document model schema, and including the appropriate parsing rules for creating such nodes when the normal parsing rules cannot process the given input.

Incomplete constructs

Many languages use composite statements, such as if-statements, while-loops, and function definitions. Such statements are composed of a header (e.g. the

[1] Grammar ambiguities are avoided through the use of the Parsing Expression Grammar formalism, which uses an ordered choice operator.

if-keyword, followed by a condition), and a body, consisting of a sequence of child statements.

Supporting free-form editing of source code requires that the programmer should be able to construct incomplete compound statements (statements with a header but no body). To this end, a Larch language schema should define nodes for representing compound statement headers (a compound statement with no body), and indented blocks (a sequence of statements, with no header). The language parser should attempt to join compound statement headers with indented blocks of code — to form complete compound statements — when it encounters them together.

Operator precedence and parentheses

When handling expression operators, a source code editor must mimic the process of un-parsing. It must take operator precedence into account by inserting parentheses into a presentation, to ensure that it faithfully represents the underlying document tree. Programmers may also wish to insert syntactically unnecessary parentheses for aesthetic reasons. To account for this, an expression AST node stores the number of additional parenthesis pairs — beyond what is syntactically necessary — that surround it.

Comments

Comments are an important part of a source code document and must be retained by a Larch source code editor. To this end, a source code document schema will explicitly define nodes for representing comments and blank lines. This is in contrast with most programming language ASTs that do not retain any comments or formatting information at all.

5.1.2 Framework

To simplify the process of developing a structured source code editor, we have implemented a framework that employs the approach described above.

Edit filters

Edit filters respond to edit notification events by attempting to handle modified presentation tree content and convert it into a structural form. The presentation tree content comes in the form of a rich string, which is parsed using an appropriate rule from the language grammar. If the content parses successfully, the resulting structural content is inserted into the document model. If it fails, the edit event is passed on to the next edit filter in the sequence of filters used by the edit rule (see below). Edit filters that handle parsed, incomplete or unparsed content are created using method calls.

Edit rules

An edit rule defines how a fragment (see sections 3.4 and 4.2) responds to edits performed by the user. Edit rules are associated with specific types of node in the document model. For instance, our Python editor defines an edit rule for expressions, one for statements, another for unparsed statements, etc. An edit

rule uses a sequence of edit filters that are applied in turn to modified content in attempt to convert it to a structural form. When a filter successfully converts modified content into a structural form and inserts it into the document model, the process is complete, and the edit operation has been handled successfully. Otherwise, it is passed to the next filter in the sequence. If no more filters are available, the edit rule hands the operation to the parent fragment, where it propagates up the presentation tree until an ancestor fragment (with associated ancestor node from the document model) is encountered that has an edit rule associated with it, at which point the process starts over, attempting to convert a larger segment of modified content, possibly using a different edit rule. An edit rule can also produce any parentheses necessary, according to the precedence rules of the language. Like edit filters, edit rules are constructed using method calls.

Document data model

A simple dynamically typed document data model is provided for the purpose of representing structured documents, which take the form of a tree. It was designed with structured source code editors in mind. A document schema declares a number of node classes, each of which declares a list of named fields. A single-inheritance model allows node class to inherit fields from a super-class. Document data model nodes provide support for incremental computation, change tracking for undo and redo, and serialisation.

5.1.3 Implementing a structured source code editor

In MVC parlance, our structured source code editor framework only provides the controller; it translates user actions into modifications to be applied to the data model.

The programmer must implement the data model that represents the underlying document. The data model will take the form of an AST-style structure and can use the document data model described above.

The programmer must also implement the visual representation. The normal approach is to define a perspective that uses the type of AST node to look-up a presentation function. Each presentation function should construct the visual representation of the underlying data and attach the appropriate edit rule.

Edit rules — and the edit filters which they use — must be defined in order to allow the editor to translate user edit operations into structural modifications to be applied to the document tree. The programmer must supply commit functions to the edit filters. Commit functions are simple functions that modify a data model; normally by replacing a node within a document with a new replacement one.

5.1.4 Structured source code editors; evaluation

The syntax-recognizing structure editor approach enables our Python editor to provide some support for mathematical typesetting *as-you-type* and visual cues to indicate structure (boxes surrounding function and class definitions). Additionally, our visual regular expression editor (see section 8.1.6) is able to

use visual cues to enhance the presentation of regular expressions, improving their readability.

Given that developing a structured source code editor involves considerably more effort than developing the equivalent text editor, we must now ask: 'was it worth the effort?'. In order to achieve the functionality provided by the Python editor, the answer is *no*. In terms of the effects on the overall project, the picture is more positive. It was the first part of Larch to be implemented. Doing so forced us to tackle difficult challenges that arose during the development of the presentation system and our object presentation techniques. Additionally, we feel that a structured model offers some as of yet unrealised benefits; refactoring tools can be more easily developed by programmers, as they can traverse and modify an abstract syntax tree (AST), rather than plain text; the feasibility of developing customised refactoring tools which can rapidly modify a large code base is an attractive proposition. Furthermore, richer storage models can be investigated, such as graphs which explicitly store name-based references and other relationships or models that include richer commenting and documentation.

Python has a terse and easily understandable syntax. As a result there is little room for improvement, in our opinion. This is not the case for all languages and notations however. Our visual regular expression editor (see section 8.1.6) is able to use visual cues to enhance the presentation of regular expressions, improving their usability. The usability problems that affect regular expressions were the motivation for the development of SWYN [5]

In summary, the primary benefits offered by our syntax recognizing editing framework are in the areas of enhanced presentation of complex notation and avenues for future research.

5.1.5 Comparison to related work

Our approach to maintaining consistency between the structured data model and the on-screen presentation is very similar to the approach used by Barista [41]. Barista in turn based its approach on the techniques developed over the course of the Pan [2], Ensemble [24] and Harmonia [6] projects. Like UQ⋆ [81], Larch allows syntax-recognizing editors to be developed at run-time within an interactive programming environment (see chapter 6).

The process of developing a structured source code editor within Larch is somewhat ad-hoc in comparison to other systems. The programmer must implement the data model, view, parser and edit rules. The parser must explicitly provide patterns for matching pre-parsed structural items and constructing incomplete compound statements. While Proxima [62] editors are also developed in a similarly ad-hoc fashion, systems such as Pan [2], Ensemble [24], and Barista [41] automate more of this development process. For instance, Barista is largely able to automatically derive a parser from the model and editor schema definitions; this is possible due to the support for statically typed data and value constraints within the Citrus [40] data model, along with its tight coupling with the editor definition.

5.2 Rich text editors

Our rich text editor sub-system provides a generic framework that supports the development of rich text editors that operate on paragraphs and spans of styled text.

Our rich text editor system represents a rich text document as a sequence of paragraphs, each of which contains a sequence of text items. A text item is either a string or a text span. A text span contains a sequence of text items. Spans can be arbitrarily nested. Attributes can be attached to either paragraphs or spans. They are normally used as style attributes and affect the appearance of the text contained within the entity to which they are attached. Normal Java or Python objects — that are presented in an interactive, visual form — can be embedded within rich text documents, either as paragraphs or as text items. In this way, images or interactive content can be inserted into a rich text document.

As with the structured source code editor framework, only the controller (in MVC parlance) is provided. User actions are translated into modifications to be applied to the data model. We chose this approach, as we wanted to give the programmer the freedom to design their data model and the visual appearance of their editor with as few constraints as possible. Our system takes care of the tricky procedures involved in joining and splitting paragraphs, and applying style attributes to segments of text, while ensuring that the optimal structure of spans is created. Our system creates the event listeners that receive user input. It responds to user input by directing the editor implementation (provided by the programmer) to construct document nodes and modify existing ones.

The rich text editing system models attributes (attached to paragraphs and spans) as a key-value table. It does not attempt to interpret their meaning (e.g. by setting the style of the text to which they apply). Its only interaction with attributes is to manage and maintain them during modifications. When applying attributes to segments of text, it constructs the most optimal structure of nested span entities to faithfully represent them, by joining adjacent spans that share common attributes.

The task of interpreting their meaning falls to the programmer. Given that the rich text editing system is data model agnostic, the programmer may choose their own storage model for attributes. The only requirement is that they can be translated to and from a key-value table. For instance, a text span model with a boolean field that controls italics will create a key-value table mapping the string 'italic' to the state of the field. This table will be given to the editing system to represent the state of the span's attributes. In response to user edit operations, the editing system will build an optimal nested span layout. It will use it to direct the application to create the necessary span and paragraph models, and then insert them into the relevant place in the document. When the editing system directs the application to create a span model, it will supply an attribute table, which the application should interpret by accessing the value associated with 'italic' and set the relevant field in the model appropriately.

Given that the programmer must design the data model — along with its representation of style attributes — they must also implement the visual representation, rendering the content as needed. The programmer must also implement the user interface that allows the user to initiate operations that modify the style of paragraphs or selected text.

This design affords a significant amount of flexibility. The editing system's model of rich text documents is intentionally abstract, so that the developer can interpret it according to the needs of their application. The contents of an attribute table could for instance control semantic meaning instead of styling.

5.3 Table editors

Tables are frequently used in technical literature to spatially arrange information in order to improve its comprehensibility. Spreadsheets offer a convenient interface for viewing and manipulating data in tabular form that is familiar users. Our table editor system provides a spreadsheet style interface for manipulating data contained within Java and Python objects. Individual cells can be selected and edited, or rectangular blocks of cells can selected, after which copy and paste operations can be used to duplicate and move data within the table. An HTML filter allows data to be exchanged between Larch and external applications, such as Microsoft Excel and Google Docs spreadsheets, or tables within web browsers.

In contrast to the source code and rich text editing systems, the table editing system provides much of the view functionality, in addition to that of the controller. The programmer must implement the data model and describe its interface to the table editor so that it is able to access and store data within it. The table editor constructs the view, presenting the data extracted from the model in tabular form. It responds to user actions by modifying the model as appropriate.

The table editing framework provides facilities for editing data in two forms. A *generic table editor* operates on a two-dimensional array, or list-of-lists. The rows (lists of cells), and list of rows will grow and shrink as necessary when data is inserted or deleted. In cases where the lengths of the rows are inconsistent a ragged table will be displayed with the length of each row reflecting that of its underlying list. An *object list table editor* operates on a list of Java or Python objects, where each object appears as a row in the table. The columns are defined within a table definition. Each column definition consists of a name, and functions for transferring data between table cells and the underlying row objects. A python attribute column specialises this behaviour by providing a name that is displayed in the column header and the name of the python attribute whose value should be displayed in the column cells. As a consequence, normal Python objects can be displayed and manipulated by a table editor simply by specifying the columns and the attributes that they represent. Adding or deleting rows from the table will result in new objects being added or removed from the list.

Figure 5.1 shows a definition (left) and example (right) of an object list table editor for a representation of a simple solar system. The definition declares a the Body class, whose constructor sets three attributes: `name`; `mass`; and `orbitalRadius`. The `SolarSystem` class derives from `LiveList`; a sequential container that supports incremental computation. It also defines a `__present__` method which uses the table editor to display its contents. Below, the table columns are defined; one for each attribute. Each definition consists of the column name, the name of the attribute within Body that contains the data, a value constructor (for converting values from textual form) and a default value.

```
class Body (object):
    def __init__(self, name='', mass=1.0, orbitalRadius=1.0):
        self.name = name
        self.mass = mass
        self.orbitalRadius = orbitalRadius

class SolarSystem (LiveList):
    def __present__(self, fragment, inheritedState):
        return _editor.editTable( self )


name = AttributeColumn( 'Name', 'name', str, '' )
mass = AttributeColumn( 'Mass', 'mass', float, 1.0 )
orbitalRadius = AttributeColumn( 'Orbital radius', 'orbitalRadius', float, 1.0 )

_editor = ObjectListTableEditor( [name, mass, orbitalRadius], Body, True, False )
```

```
data = []
data.append( Body( 'Sol', 332900.0, 0.0 ) )
data.append( Body( 'Mercury', 0.055, 0.4 ) )
data.append( Body( 'Venus', 0.815, 0.7 ) )
data.append( Body( 'Earth', 1.0, 1.0 ) )

SolarSystem( data )
```

	Name	Mass	Orbital radius
0	Sol	332900.0	0.0
1	Mercury	0.055	0.4
2	Venus	0.815	0.7
3	Earth	1.0	1.0

Figure 5.1: A simple solar system table definition and example

The example shows a solar system object presented in tabular form.

Chapter 6

Programming environment

The programming environment within Larch consists of: a Python editor; a visual console; the worksheet active document system (the most important component); the project system and some introspection tools. They are implemented as objects that reside within the same process and object space as the code and documents on which they operate; an approach pioneered by Smalltalk [22].

6.1 Python editor

Larch represents Python source code as an AST and presents it to the user for editing with a syntax recognizing editor (see section 5.1). In contrast to a plain text editor it features some visual enhancements: division expressions are presented as vertical fractions and exponentiations are presented as superscripts (our editor performs mathematical typesetting as you type). Class and function definitions are surrounded by borders. Escape sequences within strings are highlighted by surrounding them with a border. Multi-line strings are displayed within an embedded text area, in contrast to textual Python which breaks the flow of indentation of the source code to represent the multi-line string contents faithfully. Figure 6.1 contrasts the appearance of Python source code in two forms: (a) a textual form within the PyCharm IDE (33) and (b) a partially visual form within Larch.

Larch allows Python source code to be embedded within other objects, including those implemented by the user. Python expressions, suites (sequences of statements) and modules can be easily instantiated and incorporated into an object. Utilising them within an object's presentation code causes editable Python source code to be embedded within the object's visual representation. Examples of this can be seen in section 8.1.

6.2 Python console

The Larch Python console operates as a read-eval-print-loop (REPL). It provides a quick testing and exploration environment in which a programmer can test small segments of code with minimal up-front effort. It also provides introspection facilities that can be used to explore and debug Larch applications.

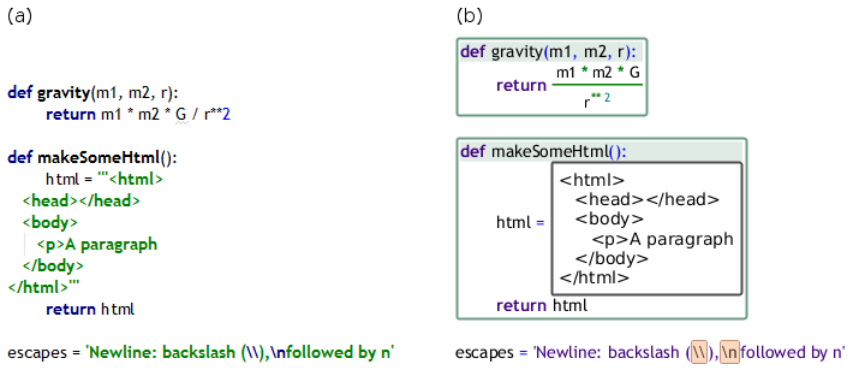


Figure 6.1: Python visual enhancements, contrasting the Larch Python editor with plain text

A typical textual Python console executes each statement after it is entered. An exception is made for compound statements (if, while, function definition, etc); the programmer is allowed to enter additional lines of code to complete the statement, until the indentation level decreases (Python uses indentation rather than braces to delimit blocks of code) or a blank line is entered. It is not possible to return to previously entered lines of code and edit them. In contrast, the Larch Python console allows the programmer to enter a complete block of code and execute it with a short-cut (Control-Enter). Prior to execution, the programmer may freely move the caret throughout the block of code and edit it. Previously executed blocks may not be modified.

It presents computation results visually, using the object presentation system (chapter 4). It uses the Python editor to provide the multi-line code editor. Text sent to the standard output or standard error streams is displayed in a box, along with a visual representation of any exceptions that were raised.

The Larch console provides introspection capabilities through the *data model dragging* facility (see section 6.5.1). The console responds to data model drops by prompting the user to name a local variable in which to store the acquired value. This allows the programmer to inspect and manipulate the data that underlies any visual presentation within Larch.

6.3 Worksheet active document system

Worksheets form the core of the Larch programming environment. They are active documents that interleave rich text and executable Python code. Execution output is presented immediately below the code that created it, maintaining locality between the two. Execution results are presented visually and can include interactive content. Larch worksheets are similar in nature to Vital workbooks [27], Mathematica notebooks [88], and IPython notebooks [58]. Figure 6.2 shows part of a worksheet that implements a simple virtual machine. At the top are two lines of rich text; a heading and a line of normal text. The source code constructs an AST (`'Mul( Add( ... ))'`) and compiles it to assembly. A object that represents a virtual machine is then instantiated, and given the assembly code to interpret. The visual representation of the VM is interactive; the 'Step'

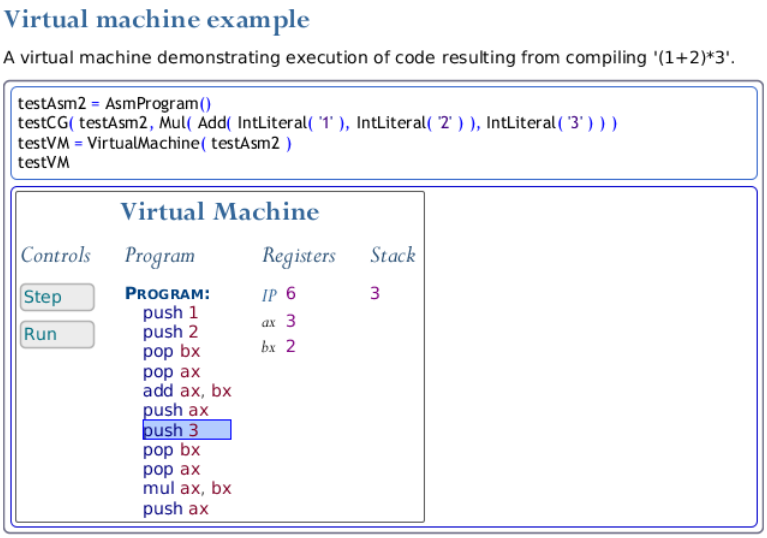


Figure 6.2: Interactive virtual machine in a worksheet

and 'Run' buttons can be pressed to advance execution. The current instruction is highlighted and the contents of the registers and stack are shown.

Worksheets are implemented using the rich text editing system (section 5.2), with code blocks inserted between paragraphs.

Worksheets build on the fast edit-run-debug cycle offered by a Python console. Where a console can only operate on small snippets of code, one at a time, a worksheet provides the same convenient work-flow, but for a complete Python module. All of the code blocks within a worksheet can be re-executed — updating all visual results — with a key-stroke (Ctrl-Enter).

Worksheets support an incremental programming work-flow. Prior work has shown that the problem solving process is non-linear in nature; the programmer experiments with a variety of solutions before settling on a final choice [44]. Programmers also benefit from the use of progressive evaluation, whereby partially finished programs or small segments of code are tested [26]. In addition to progressive evaluation, source code editing, execution and testing are combined within the same interface.

Worksheets can operate as document-centric applications and are designed for application users as well as programmers. In aid of this, worksheets can operate in two modes: *user mode* and *developer mode*. The difference between the two is simple; user mode elides content that the developer has marked as hidden (usually code that is internal to the operation of an application). Switching between modes is achieved by clicking a hyperlink that appears at the top left of the worksheet. Similar ends can be achieved within Mathematica Notebooks [88] by marking cells as hidden or non-editable.

6.3.1 Building and editing a worksheet

In developer mode, a worksheet behaves like a simple word processor that allows the user to enter paragraphs of text. Further functionality is accessed through the context menu, show in figure 6.3. The paragraph style hyperlinks set the

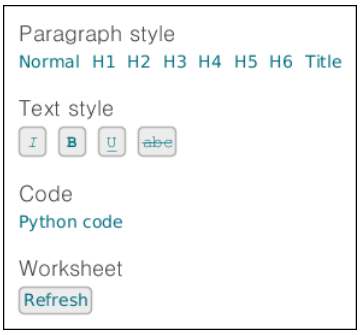


Figure 6.3: Worksheet developer mode context menu

style of the current paragraph (to one of: title, heading 1/2/3/4/5/6 or normal text; these are rich text styles provided by the presentation system, named after styles provided by HTML). The text style buttons apply styles to the currently selected text. The Python code hyperlink inserts a block of Python code below the current paragraph.

A style chooser in the header of a Python block provides options for controlling its appearance in user mode. The style options allow for controlling the visibility of the code, making the code editable (by default, code is not editable in user mode) and controlling how the result is displayed (either not at all, with a decorative border, or with no decoration). Alternatively, the code block can be hidden entirely.

6.3.2 Viewing a worksheet

User mode displays the contents of a worksheet. Code blocks that are marked as hidden (via the style option) are elided. Code blocks that show only a result can be used for displaying a GUI; the GUI is visible, but the code that creates it is not. Code marked as editable can be modified by the user; this can be used to create interactive API documentation or programming tutorials which allow the user to experiment with the code fragments.

6.4 Project system

A Larch project has a structure that mirrors the file system structure used by a typical Python program. A Python program consists of a number of Python modules (`.py` files) residing within packages (directories), which can form a nested structure. A Larch project consists of a number of pages (either worksheets or plain Python modules), that reside within packages. The project editor displays the project contents in a hierarchical tree, as seen at the bottom of figure 6.4. The context menu provides functions for creating, deleting and renaming packages and pages. Drag and drop gestures can be used to re-arrange them.

Larch projects function as complete Python programs, within which pages can import functionality from other pages and packages, as if they are regular Python modules. This is achieved through the use of Python import hooks

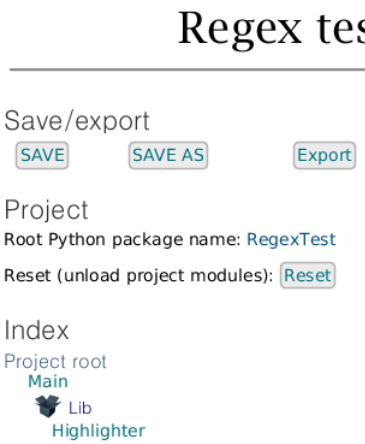


Figure 6.4: Project editor

[72], which allow a Python programmer to provide additional mechanisms for importing modules. Our implementation allows subjects (section 4.9) to define methods which allow Larch to search for and load named modules.

The root python package name (seen in figure 6.4) controls the name through which project contents are imported. In the example shown, the `Highlighter` page can be imported using the name `RegexTest.Lib.Highlighter`. The Python interpreter caches modules as it imports them, so that it does not have to load and execute a module more than once. The reset button removes all modules from the module cache that were imported from the project; this is used when the user modifies the contents of a page, so that an up to date version of it's contents can be imported.

6.4.1 Special pages

The Larch project system can utilise two special pages; the index page and the start up page. If a page named `index` is present at the root of the project, it will be displayed instead of the project by default. If a page named `__startup__` is present, it will be executed at start time. The start-up page allows a project to register extensions to the editing environment. For example, new commands can be added to the command bar, to allow new types of embedded content to be inserted into Python code and worksheets. As a consequence, a Larch project can extend the editing environment used to create and interact with it. The use of special page names was inspired by the way in which web servers look for a file named `index.html` if no file name is supplied in the URL.

6.5 Introspection tools

Larch provides introspection tools to assist during development and debugging.

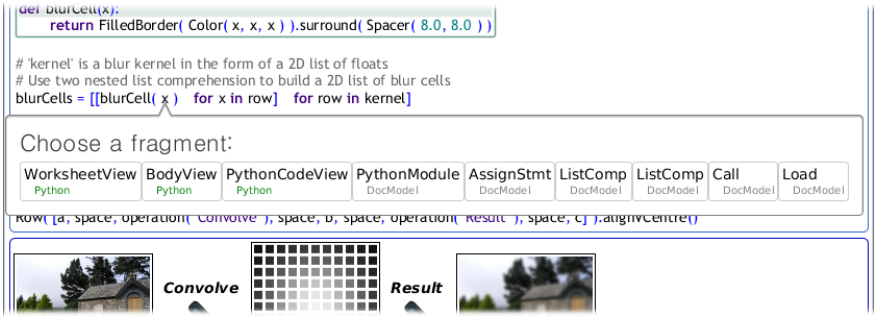


Figure 6.5: A fragment inspector

6.5.1 Data model dragging

The data model dragging facility (a feature provided by the object presentation system) allows a drag gesture to be used to acquire the data model object that underlies any part of a presentation within Larch. The acquired data model can be dropped into any Larch application or tools that supports it, e.g. the Python console. Models can be acquired from anywhere within Larch; AST nodes representing Python source code within a Python editor can be dragged to a console, where they can be inspected or manipulated. A common use involves dragging objects that result from computations within a worksheet or a console, and dropping them into source code or worksheet text in order to embed them.

6.5.2 Fragment inspector

The fragment inspector is used to inspect the data model and presentation elements that are associated with a presentation fragment and its ancestors along the path to the root. The programmer activates it with a click gesture (Alt-Shift-Right click) over the presentation content that they wish to inspect. At this point, a pop-up appears, allowing the programmer to select the fragment that they are interested in. Each fragment is described in terms of the name of the class of its associated data model, as seen in figure 6.5. Upon selecting a fragment, a new window is opened, that contains two tabs. The first tab contains a Python console (section 6.2) in which the data model is bound to the local variable `m`. The programmer may use the console to inspect and manipulate the data model. The second tab contains an element tree explorer (section 6.5.4) that displays the sub-tree of elements that form the presentation rooted at the chosen fragment.

Figure 6.5 shows a fragment inspector displaying the fragment ancestry of a load expression within a nested list comprehension, from the code shown in figure 3.1.

6.5.3 Inspector perspective

The inspector perspective presents objects in the style used in visual debugging environments; the objects attribute and field values are displayed allowing the user to explore an object's internal state. The inspector perspective can be

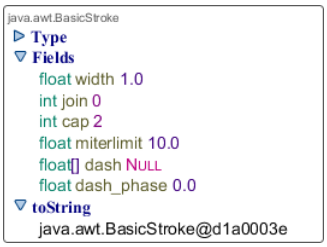


Figure 6.6: An object inspector

invoked by calling it, e.g. `inspect(x)`. Figure 6.6 shows an object inspector displaying a `java.awt.BasicStroke` object.

6.5.4 Element tree explorer

The element tree explorer is a debugging tool which allows the programmer to explore the elements within a presentation sub-tree. It is similar to the HTML element tree explorer provided by the Google Chrome [23] web browser. It can be used to display the presentation tree for an entire page by activating a menu item or a presentation sub-tree by using the fragment inspector.

6.6 Comparison to related work

The tools presented in this chapter are very similar to prior work in the area. We don't claim to have made significant novel contributions in this area; novel contributions reside in other parts of Larch.

Our programming environment has some minor novel features that we will now discuss. The Python editor adds some minor visual and usability enhancements to Python source code. The console allows multiple lines of code to be entered at a time and presents results in a visual, interactive form; one can display complete, interactive interfaces within a console. Our worksheets operate as notebooks that can function as a user facing GUI, as well as a programmer facing development environment. Our project system ties the other components together, providing a simple way of creating interactive applications.

Larch provides some of the benefits of live programming environments — primarily continuous evaluation — while utilising Python, an imperative object-oriented language. This is in contrast to much prior work in the field, in which custom languages have been used. The main benefit of using Python is that programmers do not need to learn a custom language and programming paradigm in order to use our system. We also avoided the mammoth task of designing a programming language that programmers would want to learn and use.

Like prior live programming environments, Larch encourages a rapid edit-test cycle, while the integration of incremental computation with the object presentation system eases the development of interactive, visual content.

Chapter 7

Partially visual programming with embedded objects

Our visual programming approach is based on embedding objects in-line within source code. We use the object presentation system described in chapter 4 to present them in an interactive visual fashion, thus permitting interactive or GUI-driven visual content to be embedded within textual source code. By allowing code to be contained within objects as well, our approach permits visuals and code to be freely mixed with one another. Functionality similar to that of LISP-style macros is achieved by using the *embedded object protocol* to control the way in which embedded objects are compiled or executed. Consequently, they can be used to develop new kinds of interactive programming tools and partially visual languages. They are an editor-based construct that provides a novel object-oriented approach to extending a language, without the necessity of modifying the language compiler or interpreter.

Embedded objects are fully functional objects. They are mutable and can contain references to one another, allowing them to co-operate. In addition, they cross the boundary between edit/compile-time and run-time. An object that is presented to the user at edit time can be accessed, used and modified by the code in which it is embedded at run-time; often causing it's visual representation within the editor to update in response.

LISP and its derivatives (such as Scheme) support language extensions by the use of simple S-expression based syntax and the use of macros, which traverse and manipulate LISP code in abstract syntax tree (AST) form. Smalltalk [22] takes the approach of incorporating it's development tools (including the compiler) into the programming environment, where they can be modified and extended as needed.

Embedded objects also provide a novel approach for developing interactive documents and technical literature by allowing an object to be embedded in both source code and rich text paragraphs within a worksheet. As a consequence, user actions performed on the object's visual representation that lies within the worksheet text will also affect its operation when it is executed as part of the code in which it is placed.

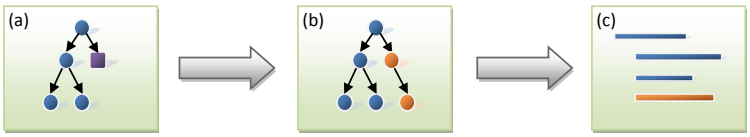


Figure 7.1: Code generation with embedded objects.

In contrast to prior work such as ETMOP [19] and Racket [60], embedded objects are easier to use, as they do not have to conform to any specific design, beyond implementing particular methods to control their behaviour. Mathematica's *generalised input* [87] allows visual output to be copied and pasted into code. Its `Interpretation` function provides behaviour customisation.

7.1 Embedded object protocol

The embedded object protocol defines a set of methods which a Python object can implement in order to control the way in which it is compiled or executed when embedded within source code.

Process overview

As stated in section 6.1, Larch represents Python source code using an AST-like structure. The Python data model schema defines special AST nodes that contain a reference to an embedded object.

Executing Python source code defined within Larch requires two phases; code generation and run-time execution. During the code generation phase Larch AST nodes[1] are converted to a form that the Python interpreter can process[2]. Embedded objects are processed at code generation time by replacing them with generated code that will have the intended effect. As a consequence embedded objects provide a mechanism for extending the language syntax without needing to modify the Python interpreter. The code generation process is shown in figure 7.1. We start with (a) an AST composed of normal code and an embedded object (the box). In (b) the embedded object protocol is used to replace the embedded object with AST nodes that perform the desired functionality. Finally, in (c) the resulting AST is converted to a form which can be processed by the language compiler or interpreter.

Embed as a literal

The object acts as an expression, that evaluates to a reference to the embedded object. This allows visual interactive values to be embedded within source code, as seen in figure 7.2 where an interactive polygon is embedded within a

[1] We define our own AST nodes (using the document data model mentioned in section 5.1.2) instead of using the standard Python AST nodes, as the standard AST nodes cannot represent comments and do not emit change events; these features are required by our Python editor.

[2] We currently convert Larch AST nodes to textual source code, which is executed by the interpreter. In the future we intend to convert Larch AST nodes to Python AST nodes directly as this removes the un-parse (to text, within Larch) and parse (within the Python interpreter) steps that are currently used.

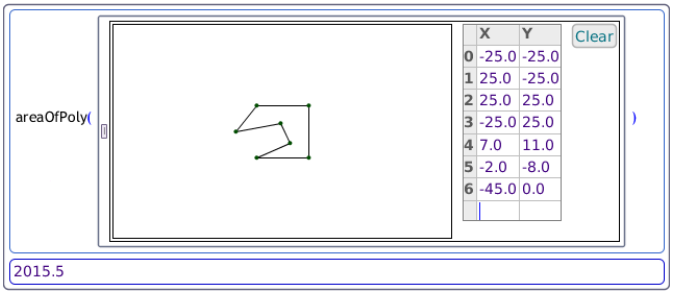


Figure 7.2: An interactive polygon embedded within source code

call to a function called `areaOfPoly` (the result of evaluating the expression is shown below). Internally, the code generator replaces the embedded object with a Python expression that evaluates to a reference to the embedded object. Objects embedded as literals do not need to implement any embedded object protocol methods; any Python object can be embedded as a literal.

`a.x = f(`  `)`  `a.x = f(box)`

Embed as an expression

The object acts as an expression and is able to determine how it is evaluated (for an example, see the visual regular expression tool in section 8.1.6). Its evaluation result is acquired by invoking its `__py_eval__` method. The code generator replaces the embedded object with an expression that invokes its `__py_eval__` method at run-time.

`a.x = f(`  `)`  `a.x = f(box.__py_eval__())`

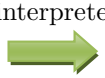
Embed as a statement

The object acts as a statement and is able to determine how it is executed. It is executed by calling its `__py_exec__` method. The code generator replaces the embedded object with a statement that invokes its `__py_exec__` method at run-time.

  `box.__py_exec__()`

Embed as a macro

The object functions as a LISP-style macro (for an example, see the table based unit test tool in section 8.1.7). The `__py_evalmodel__` (for expressions) or `__py_execmodel__` (for statements) methods create AST nodes which are used in place of the embedded object. The code generator invokes the `__py_evalmodel__` or `__py_execmodel__` method at code generation time and replaces the embedded object with the AST nodes that they return, before converting the resulting tree for use by the Python interpreter.

`a.x = f(`  `)` 

`a.x = f(≪AST from box.__py_evalmodel__()≫)`

It is worth noting that in the examples above, we have placed the embedded object in a variable called `box`. This was done for clarity; in our implementation, embedded objects are placed in a list — accessible via a global variable — which is initialised when the module is instantiated.

7.2 Performance impact

Embed as literal

The overhead for embedding as a literal is minimal. In our implementation, an object that is embedded as a literal is replaced by a global variable access and an array access when the code is passed to the interpreter. Our implementation keeps an array of embedded objects within each module. Other implementations could generate a new global variable for each embedded object, eliminating the array access.

Embed as expression or statement

In addition to the global variable access and the array access, embedding an object as an expression or as a statement adds a method invocation and the computational cost of executing the method. Given that the method may incur a significant cost, a production-ready implementation of an embedded object system should allow the embedded object developer to provide easily accessible documentation describing the embedded object's computational costs, so that the programmer could assess its effects on their program.

Embed as a macro

Given that an object embedded as a macro is replaced by the code that it generates (when it is handed to the interpreter or compiler), the effects on performance depend on the nature of the code generated. Once again, a production-ready implementation would need to provide facilities for documenting computational costs.

7.3 Implementation

7.3.1 Within Larch

As mentioned in section 7.1, the Larch Python data model schema defines special AST nodes for representing embedded objects. Three embedded object node types are defined: literals; expressions and statements. The literal node type is required as the programmer can force an object to be embedded as a literal. When an object is not embedded as a literal, the editor needs to be able to determine which kind of syntactic construct it represents; an expression or a statement. This is so that the parser within the syntax recognizing editor can determine whether a given segment of code — that includes an embedded object — is valid or not.

7.3.2 Within other environments

While our prototype editor is quite usable, our choice of a structural editor deviates significantly from typical code editors which represent code in a textual form. Given that a textual form is most likely to be chosen by other implementers, we will describe an approach to implementing embedded objects as an extension for existing text-based programming environments, as opposed to our own system.

We would suggest that the editor should represent source code using an enriched string; a sequential container that stores interleaved characters and object references (see 'Basic Operation' in section 5.1.1). The text pane would render the characters as normal text and the object references visually. While type coercion based object presentation (chapter 4) is not necessary for the implementation of embedded objects, we would consider it to be helpful, as it provides a convenient API for developing the object's user interface. Implementers are however free to use other systems as they see fit. The only other requirement is that the source code rendering system would display an embedded object's visual representation in-line with the source code, arranging the source text around it.

The code generation process would replace embedded object references with the appropriate textual source code, according to a protocol similar to the one described above. The resulting text would be passed to the language compiler.

It is worth noting that supporting the embed-as-macro functionality necessitates the availability of a parser to translate textual source code into AST nodes for further processing. Python provides such a parser, accessible through its standard library. Other languages may require the use of external tools or libraries to enable this functionality.

7.4 Creation

There are two ways in which an object can be embedded within source code.

Drag and drop

An object can be dragged from a different part of the application and dropped into source code. The data model dragging facility mention in section 6.5.1 is often used to acquire an object from elsewhere within Larch (e.g. from an object created within the Python console (section 6.2), or a worksheet (section 6.3)), after which it is dropped into a Python source code editor. Dropping an object displays a context menu that asks the user if they wish to insert the object as a copy, a reference, a literal copy or a literal reference. Inserting an object as a copy will create a fully deep copy of the object and embed the copy. Inserting as a reference allows an object to be embedded in multiple locations throughout a document. Inserting as a literal copy or literal reference in that the copy of reference is embedded as a literal, allowing the surrounding code to access the object itself, rather than the result of evaluating it.

Commands

Embedded objects can be inserted by invoking commands from the *vi*-style

command bar that is provided by the presentation system. Developers can implement new tools that insert embedded objects to extend the functionality of Larch editors (source code editor, worksheet text editor, etc.). An object may be inserted at the position of the caret, or may be wrapped around a selected expression, statement or range of statements. The wrapping procedure replaces the selected code with a newly created object containing the contents of the selection (recall that objects can contain source code within them; section 6.1).

7.5 Isolation Serialisation System

Our system permits embedded objects to be instances of classes, which are defined within other parts of the same document/project. Consequently, special care must be taken when serializing and deserializing Larch documents.

Larch documents are represented in memory using standard Python objects, and Java objects that implement the Python serialization protocols. We use the Python *pickling* system (the standard Python serializer) as the basis for our serializer.

Problems arise during deserialization, when objects must be recreated. The first step in deserializing an object, is to acquire a reference to its class. The class is used to create a blank instance, whose field/attribute values are loaded from the serial stream. The class is normally identified by a name, which is used by the *pickler* to locate it. Our requirements pose an additional challenge; the class identified can be defined within another part of the same project, which is not yet completely available due to deserialization still being in progress!

Figure 7.3 shows the kind of document structure that inspired the development of our isolation serialisation system. It shows three modules, each of which contains an embedded object that is an instance of a class defined in a different module. The structure results in the dependency chain:

$$C \rightarrow Z \rightarrow B \rightarrow Y \rightarrow A \rightarrow X \tag{7.1}$$

Modules that contain embedded objects need them to be fully deserialized in order to operate, and deserialization of embedded objects needs a class defined in another module to proceed. The standard unpickler (Python deserializer) would attempt to deserialize all the objects that make up the document in one go; both the objects representing the code in the modules (in AST form), and all the embedded objects. Y and Z cannot be deserialized immediately, since the code in modules A and B (respectively) has not been executed, since the deserializer has not yet fully re-created the object graph within which the code is contained.

Our solution is to divide the object graph into distinct partitions, which can be deserialized as needed. When a document is loaded from a file, many partitions will remain in serialized form, in memory. Referring back to Figure 7.3, assume that X, Y, and Z are in different partitions, and that we need module C. Referring to the dependency chain in Equation 7.1, we can now deserialize (objects) and execute (modules) in reverse order. Partitioning allows us to deserialize objects when all the necessary information is available.

We built our system upon the standard Python serialisation system (the *pickler*), in order to maintain compatibility with the Python serialisation protocols.

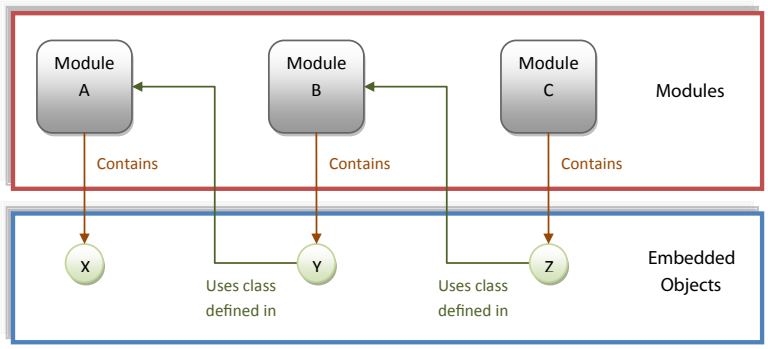


Figure 7.3: Motivation for isolation serialisation system

Definitions

An object graph can be represented as a digraph G, consisting of set of nodes N, where each node represents an object, and a set of edges E, where each edge indicates the presence of a reference from one object to another. We define T_x to be the set of nodes that are reachable from node x (transitive closure).

Approach

Our approach involves placing embedded objects behind *isolation barriers*. An isolation barrier is an object that contains a reference to an *isolated object* that lies 'behind it'. Upon serialisation, the reference is converted to an integer that indexes into an *isolated object table*. Upon de-serialisation, the isolation barrier keeps the index, until the object reference is requested, at which point, the isolation serialisation system re-creates the isolated object, and the index is replaced with a reference. Please note that we do not consider G to contain edges from isolation barriers to the isolated objects that are behind them.

The diagram in Figure 7.4 shows an object graph divided into partitions. The circles represent objects in the object graph, and the thin boxes represent isolation barriers. Objects that are reachable from the same set of isolated objects are in the same partition. Consequently, deserialising an isolated object requires that only the partitions that are reachable from it are deserialised; other partitions may remain in serialised form.

Formal description

Given a root object r, we discover T_r, the set of all objects reachable from r. During this process, when we encounter an *isolation barrier*, we add the *isolated object* that lies behind it to I, the set of all *isolated objects*. Note that more than one *isolation barrier* can refer to the same *isolated object*, in which case they will be given the same index into the *isolated object table*.

We can now divide N into two disjoint sub-sets T_r and T_I where:

$$T_I = \bigcup_{i \in I} T_i$$

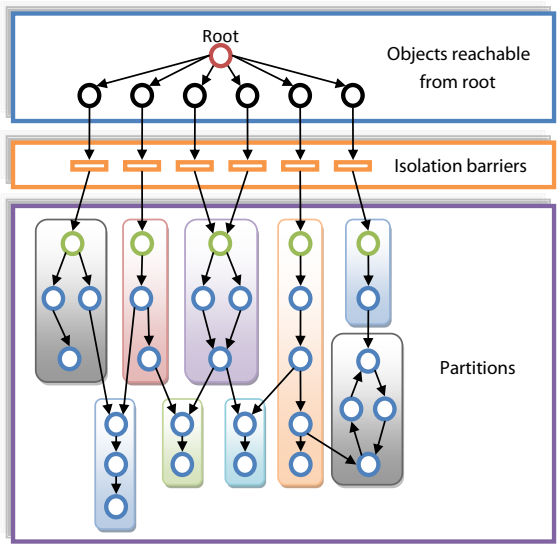


Figure 7.4: Isolation serialization system - partitioning.

where T_i is the set of all objects reachable from i.

We now divide T_I into a number of disjoint sub-sets, each of which forms a partition of G. We first define the *source set* S_x to be the set of isolated objects from which x is reachable:

$$S_x = \{i \in I : x \in T_i\}$$

We now define a partition P_a as the set of all objects for which S_x is the same for each element x of P_a, along with its associated source set Q_a:

$$P_a = \{x \in T_I : S_x = Q_a\}$$

We have now partitioned T_I into partitions $P_0, P_1, \ldots P_M$, where M is the number of partitions, by the set of isolated objects from which each partition is reachable. We now serialise each partition into a string. Note, that when serialising objects within a partition P_a, we can encounter references to objects not within P_a. We must serialise these references specially; we record an external reference, consisting of the identity (an index) of the partition P_b within which such an object y resides, along with the index of y in P_b. We may also encounter references to objects within T_r; we record an identify for these too.

In order to de-serialise an isolated object i, we will first need to de-serialise each partition for which i is a member of its source set:

$$\{P_a : i \in Q_a\}$$

We may now de-serialise i itself, handling all external references as we go.

Alternative approach

Using the pickler guarantees compatibility with existing code, but necessitates the use of a more complex approach than would be required, if we developed our own serialization system from scratch. The main requirement of our system is the ability to defer the deserialization of specific objects, until all the necessary resources (classes, functions, etc) have been instantiated. This could be achieved by storing the serialized objects in a simple table, that maps indices to objects. Inter-object references would be represented by indices. When loading a document, objects behind isolation barriers would remain in serialized form within the object table, only being deserialized on request. Consequently, this simpler approach would achieve the same results as our partitioning system.

Limitations

It is possible to use construct a Larch document which has circular module dependencies. If a user embeds within a module M, an object whose class is defined in a module N, which in turn imports (and is therefore dependent on) module M, the resulting document will not load properly, as module M will fail to load, due to the deserializer not being able to re-create the embedded object. We propose to address this issue in the future, by detecting dependency cycles at save time, and warning the user.

7.6 Limitations

Embedding objects from one document into another can result in unexpected behaviour or invalid documents. Within the same Larch session the documents will remain linked, as they will share the same embedded object within memory. Upon reloading, this relationship will not be preserved as each document will receive its own copy of the embedded object at load time. If the embedded object's class is defined within the source document, the destination document will become invalid, as the required source code will be unavailable, unless the source document is loaded first.

From the perspective of the programmer, objects embedded as literals must be treated as if they are global variables. This is particularly relevant for objects embedded within functions or class methods, as their appearance within the code may incorrectly suggest that a new instance of the object will be created for each function invocation or class instance.

7.7 Comparison to related work

7.7.1 Visual languages and DMPEs

The usability problems that hamper the widespread acceptance of visual languages and DMPEs among professional programmers are the motivation for our choice to develop a partially visual programming system. Developing visual languages that scale up beyond toy examples has proven to be very challenging. The cumbersome work-flow of DMPEs further discouraged us from taking a either a purely visual approach or from developing a DMPE based editor.

7.7.2 Visual source code extensions

As stated in section 2.6, augmenting rendering of standard source code requires the development of robust pattern matchers that select the programming constructs that are to have their presentations enhanced. The difficulties involved in developing reliable pattern matchers underlie our decision not to explore this approach.

We chose to base our approach on visual programming constructs. Our object presentation system provides a simple way of displaying objects in visual form in-line within textual source code. Continuing our philosophy of giving freedom and flexibility to the programmer by placing few constraints on the design of their code, we designed the simple protocol described in section 7.1. It places less of a burden on the developer when contrasted against the approaches employed by ETMOP [19] and Racket [60], since developing new ETMOP AST node types or new kinds of Racket boxes requires the developer to work within the constraints of their respective designs.

When objects are embedded as macros, our approach offers some of the capabilities to LISP macros, except that embedded objects are presented visually, rather than appearing as normal function calls. Objects that are embedded as (non-macro) literals, expressions or statements offer powerful visual source code extensions without the need to manipulate AST nodes.

7.7.3 Domain specific languages

The two DSL development environments discussed in section 2.8 (the Intentional Domain Workbench [65] and Jetbrains MPS [34]) are geared towards language development. In comparison, Larch's language development facilities are quite primitive. Defining a syntax recognizing editor within Larch is certainly more complex than defining an editor within MPS. The behaviour of a Larch editor will be closer to that of a text editor and therefore more familiar to most programmers. MPS provides a library of generic language constructs (e.g. arithmetic operations, function calls), eliminating the need to develop common constructs from scratch. Both MPS and Intentional software provide version control and support for developing type systems.

Rather than providing a framework for developing new languages and notations, Larch focuses on allowing a developer to customise an existing one through the use of embedded objects. By doing so, one avoids the tricky issues faced by language designers. Choosing syntax and semantics that stand the test of time is a very challenging task, not to mention the effort required to develop a sufficient standard library.

Chapter 8

Evaluation

8.1 Proof of concept

We have used the embedded objects system in concert with the other components of Larch to develop examples of potential applications. These tools are only intended as a proof of concept.

8.1.1 Interactive explorable documents; embedded editable values

Embedded editable values are inspired by Tangle [73], a Javascript library designed for the creation of reactive HTML documents. Reactive documents allow a reader to experiment with the scenario presented by the document; modifying interactive values causes textual and visual elements to change in response. For instance, Ten Brighter Ideas [74] — based on Tangle — allows a reader to explore the effects of a variety of energy saving measures.

Embedded editable values (EEVs) allow interactive, user-editable values to be embedded within Python code. Visually, they appear as GUI controls (figure 8.1). A frequent use case involves using an EEV to control value of a parameter that affects the operation of the surrounding code, by embedding it in place of in place of a literal value (e.g. an integer or string). Drag and drop is then used to embed a reference to it within a text paragraph within a worksheet. The control that appears within the worksheet text can now be used to alter the parameter. EEVs provide a simple approach to rapidly connecting parameters within code to GUI controls.

EEVs inform the incremental computation system (section 4.4) of value changes. Incrementally maintained functions — such as live functions (section 4.5) — that access the values off EEVs will therefore be automatically updated (along with their visual representations) in response to user modifications. This allows EEVs to be used in the creation of live interactive documents. The same functionality can be achieved in Mathematica through the use of its `dynamic` and `Interpretation` functions [87].

Figure 8.1 shows a very simple demonstration of the use and underlying principles of EEVs. The top line, 'Define x=...' is worksheet text. The outer blue box denotes the boundary around a code block, the lower part of which

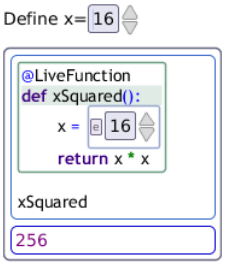


Figure 8.1: Square function with embedded editable value

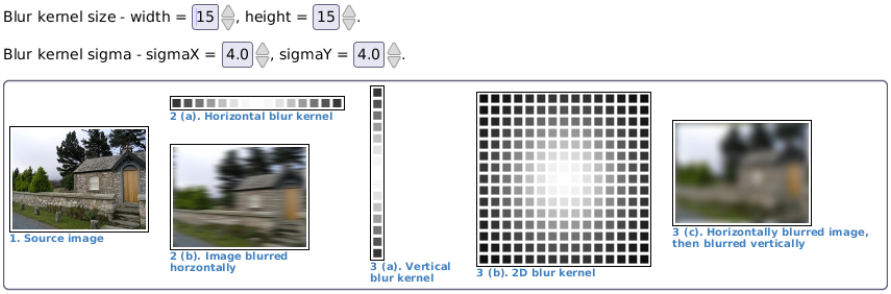


Figure 8.2: Separable Gaussian blur using embedded editable values

displays the result of the expression `xSquared`. The two spin boxes (numeric value with arrows) are presentations of the same embedded object; changing the value in one affects the other. The `@LiveFunction` decorator applied to the `xSquared` function converts it into a live (incrementally maintained) function. As a consequence, altering the value of either spin box causes Larch to automatically re-invoke `xSquared` and update the value shown at the bottom[1]. If the code block was marked as hidden, the result would still be visible, as would the spin box at the end of the line of text 'Define x=...'. The user could change the value in the spin box and see the result update in response.

Figure 8.2 shows a more complex example; a visual demonstration of a separable Gaussian blur. Altering the parameters controlling the blur kernel size and sigma causes the visuals below to automatically update; the size and shape of the blur kernels are altered, and the blurred images change to demonstrate the effect. Figure 8.3 shows the separable Gaussian blur worksheet in developer mode, revealing the implementation. 11 lines of import statements were elided, including statements that imported functions for computing the blur kernel and rendering a Gaussian blur.

EEVs take the place of values that are typically represented as literals or constants within textual source code. EEVs have a number of advantages. Firstly, they can use controls specific to a certain domain. An embedded file path displays a text entry field alongside a button that opens a file dialog window. In contrast, one would use a string to represent a path within textual source code. No domain specific behaviour would be provided by the interface.

EEVs are implemented as objects that present themselves as GUI controls.

[1]Recall that live values and live functions are presentation combinators that display a live presentation of their values.

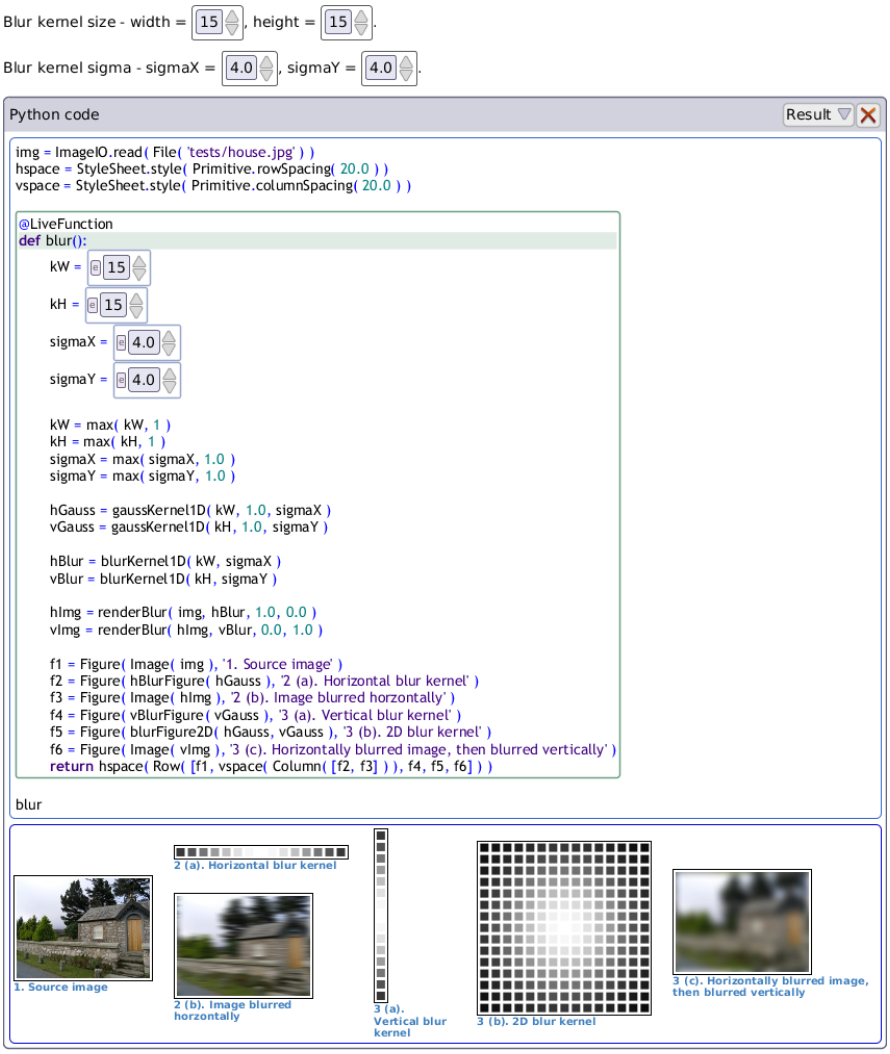


Figure 8.3: Separable Gaussian blur implementation

Internally, their values are stored within a live value (section 4.5), hence they can participate in incrementally maintained computations.

8.1.2 Simplified interactive literate programming

Literate programming [38] is an approach to programming in which source code is organised in an order suited to human understanding, as opposed to the order demanded by the programming language. Source code within literate programs is divided into small named segments which are placed in context with documentation that describes their operation. Code segments reference one another by name; it is this web of references that defines the structure and order of the compilable source code. A tool called Tangle generates the compilable source code by resolving these references in a similar fashion to the C-language macro system; the complete source of a referenced segment is inserted in place of the reference in the referring segment. The Weave tool generates printable documentation via TEX. Tangle and Weave operate as batch-oriented command line tools.

The Larch literate programming tools operate interactively. Code segments are contained within embedded objects which can be referenced in multiple places throughout a project. All visible representations of a code segment are editable and synchronised; any modifications made to one will affect the others. Code segments are executed in place when embedded within source code. They are not executable when embedded within worksheet text. Literate code segments come in two types: expressions and suites. Suites have an editable name and are visually collapsible. Literate code segments are created by selecting a segment of code and invoking a command from the command bar, which will cause the selected code to be wrapped within a literate expression or suite (depending on the command invoked). Drag and drop gestures can now be used to embed new instances of the segments within source code or worksheet text.

The interactivity of the Larch literate programming tools facilitate applications beyond the production of *programs as literature*; the original inspiration for Literate Programming. Small segments of code (e.g. expressions) can be taken from a large body of code, and displayed in editable form within a worksheet, allowing a reader to edit small segments that are relevant to the worksheet's topic, without the distractions of the rest of the supporting code. An example is seen in Figure 8.5 where the uncompressed input is contained within a literate expression, seen embedded within the worksheet text. Additionally, the implementation of the LZW algorithm and the program trace visualisation (explained in the next sub-section) is also shown within a literate suite. The worksheet contains only the information that is salient to the operation of the LWZ algorithm, the supporting code has been elided (it can still be seen and explored by the user by switching the worksheet to developer mode).

Our approach has a number of advantages when compared to the standard textual literate programming tools. The Tangle and Weave tools can be used to generate either form of output (either compilable source code or human readable documentation). Both of these forms are read-only. It is not possible to modify the source document by editing the typeset documentation. A detangle tool was later developed that could transfer modifications to the compilable source code to the original source document. Within Larch, both source code and documentation forms are simultaneously available and editable, with modifications

to one form immediately affecting the other.

8.1.3 In-line console

The in-line console tool is an experimental programming and debugging aid that takes the form of a simple console that can be embedded within source code. It contains an editable block of Python code, which is executed in the context of the surrounding code into which the console is placed. The code within the console has access to variables that were available to the surrounding code at the time of execution. Results created by executing the code block are displayed below it. The programmer may then modify the code within the console and refresh the result with a key-stroke. As a consequence, the in-line console tool allows a programmer to explore the values of variables and objects that were available to the surrounding code. The programmer may also wish to incrementally develop and test a segment of code within the in-line console. Once finished, the in-line console can be dissolved with a key-stroke, replacing it with the code that was contained within it.

The worksheet system (section 6.3) allows the values of variables to be displayed, provided that they are variables available to the global (module level) scope. Displaying values available within the body of a function or within an event handler is not directly supported[2]. The in-line console tool fills this gap; placing one within the body of a function or event handler allows the programmer to inspect or interact with the state of local variables.

The in-line console works by taking a copy of the local and global scopes when the surrounding code is executed. These copies are stored as dictionaries that map variable names to values. The code within the console is executed in the context of these scopes. Its results are retrieved and displayed.

Figure 8.4 shows an in-line console within the step method of the virtual machine that was shown in figure 6.2. The `step` method is invoked in response to pressing the 'Step' button. The program is in the same state as seen in figure 6.2. The in-line console is being used to inspect the instruction pointer, the name of the instruction and the contents of the registers.

The closest analogue to the in-line console in a text based environment would be a procedure that can be used within a Smalltalk environment, first discussed in section 2.5. Within Pharo Smalltalk [1], inserting the code `self halt` into a method will halt execution and summon the Smalltalk debugger. Within the debugger, the programmer can execute segments of code within the context of the method. After implementing the desired functionality, the code can be copied and pasted into place.

Visually, our in-line console displays the code segments under development in-line within the surrounding code. The dissolve function is quicker than the copy and paste actions needed within Smalltalk.

Our in-line console does not interrupt execution. This can be a disadvantage, as subsequent statements may modify the state of objects accessible from within the in-line console, so it may not be a true representation of the state of the application at the time the in-line console was encountered within the flow of

[2]This can however be achieved through the use of a workaround; a container must be created and bound to a global variable and displayed within the worksheet. Within the function body, the value that is to be inspected should be placed within the container. While not difficult to achieve, the work-flow is cumbersome.

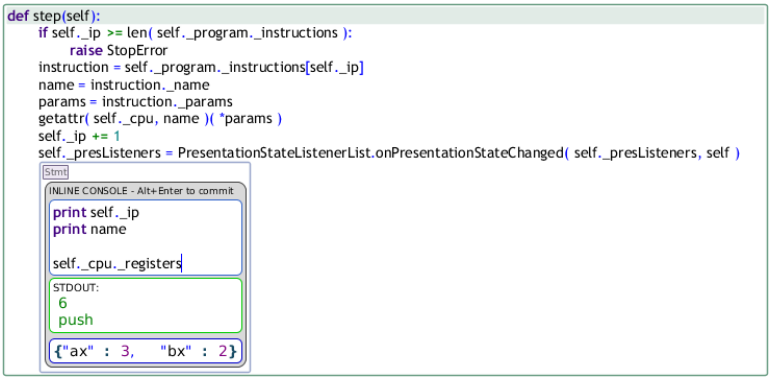


Figure 8.4: An in-line console

execution. This problem could be addressed in the future by executing the program under development in a separate thread that could be halted.

8.1.4 Simple static software visualisation; program trace visualisation tool

The program trace visualisation (PTV) tool is a simple program visualisation system. Given that it is intended as a proof of concept, it is quite primitive in comparison to past state of the art techniques [67] and even more so when compared to current work. Jype [30] is a recent system that displays function activation records and data structures in a visual form within the context of an educational programming environment for the Python/Jython platform. It would serve as a good starting point for readers interested in pursuing this area further.

The PTV tool generates a static program visualisation by tracking the execution of a sequence of executable statements. The user chooses a number of expressions whose values are to be logged by tagging them as *monitored expressions*. As the statements contained within the PTV are executed, the values of monitored expressions are recorded. The PTV can display them in a table (as in figure 8.5), or in an activation tree (as in figure 8.6). The PTV tracks the invocation of the code contained within it, and is able to detect iterative and recursive invocation. A limitation of the PTV tool must be noted; the logged values are stored as references. As a consequence, mutable objects may be modified by code that is executed subsequence to logging, resulting in the visual representations depicting their current, modified state rather than their state at logging time. This is an issue that we would need to address in the future.

Figure 8.5 shows a PTV within an implementation of the LZW compression algorithm. The variables `r`, `g` and `b` are used in the input data and are bound to objects which display as the coloured boxes seen in the table. At the end of each iteration of the LZW loop, the PTV (the table, which is embedded as a statement) is executed, which in turn executes the code contained within it. In this case, the code is the tuple expression '`x, P, Q, y`'. The tuple is admittedly difficult to see, since each element is wrapped in a 'Monitored

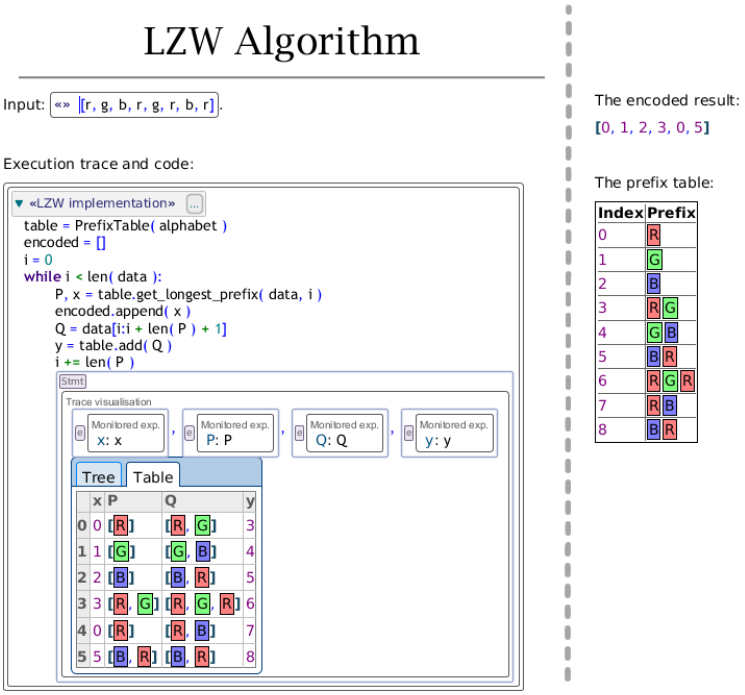


Figure 8.5: A program trace visualisation of the LZW compression algorithm

expression' embedded object, which gathers the value of the expression (`x`, `P`, `Q`, or `y`) within it at evaluation time. The PTV displays each log entry as a row in the table. As a result, the progress of the algorithm is visualized in the table. The PTV was very easy to apply; the line '`x, P, Q, y`' was entered and selected, after which the PTV command was invoked. Each of the `x`, `P`, `Q`, and `y` variables were selected, and the monitored expression command was invoked.

The worksheet shown in figure 8.5 uses both the PTV tool and the literate programming tools. The literate programming tools are used to display the input expression (`[r,g,b,r,g,r,b,r]`) within the worksheet text at the top. The input expression — originally within the source code — was wrapped in a literate expression (a Python expression contained within an object). A reference to it was embedded within the worksheet text resulting in what is seen. The implementation of the LZW algorithm — containing the PTV — was wrapped in a literate suite (a Python suite contained within an object), that can be seen with the title 'LZW Implementation'. Again, a reference to it was embedded within the worksheet, this time below 'Execution trace and code:'. The encoded result and the prefix table built during execution is shown further down, although these have been placed to the right of the main body in the figure to save space.

Figure 8.6 shows (the bottom half of) a PTV used to monitor the construction of a KD-tree (a binary spatial partitioning structure used in graphics) as it partitions a cloud of points. The construction algorithm is recursive; the two recursive invocations can be seen at the top of figure 8.6 (a) (the code '`self.lower = ...`' and '`self.upper = ...`'). The PTV has tracked the recursive invoca-

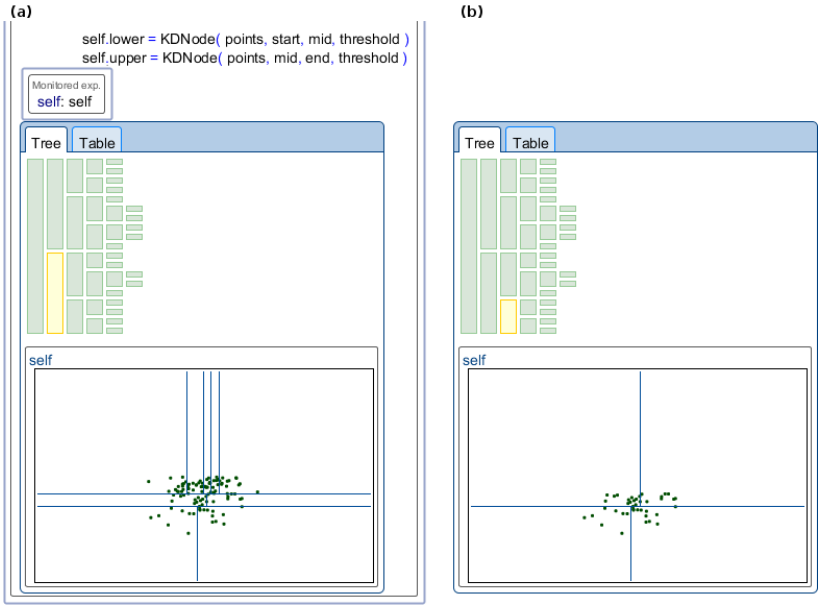


Figure 8.6: A program trace visualisation of a recursive algorithm

tion and generated the activation tree, seen just below the 'Tree' tab. Each box in the tree represents an activation record; a record of a recursive invocation. Selecting a box takes the values recorded during the associated invocation and displays them below the activation tree. In this case only one value is recorded; the KD-tree node itself. The KD-tree node is presented as a diagram that shows the points within the selected node along with partitioning lines used by the node and its children. Figure 8.6 (b) shows the activation tree with a different record selected.

In contrast, textual source code would wrap the expressions tagged as monitored expressions in logging method calls. The execution results would presented in a different pane within the IDE, as textual code editors do not allow execution output to be interleaved with the source code (although some sophisticated IDEs — such as Visual Studio [32] — can display the values of variables in tooltips while debugging). They would also be presented in textual form, relying on careful use of spaces and tabs with a fixed width font to align the content.

8.1.5 Language extensions and domain specific programming tools

The combination of the language extension capabilities and the ability to embed code within objects allows for the development of domain specific programming tools and customised languages. We hesitate to use the term *domain specific language* as it would suggest the creation of a new programming language. Instead, we propose enhancing the host language (Python in this case) by allowing segments of code to be combined and augmented by interactive visual forms in order to customise the language to better support the target domain.

This is achieved by implementing an object that is to be embedded within

code (such as the table shown in figure 8.7). Segments of code are further contained within the object. The object displays itself and the code segments within it in an editable, interactive form. At code generation time, the code segments are manipulated at the AST level, such that they are utilised and executed as needed.

Figure 8.7 shows one such example; a specification of an instruction set for a MIPS CPU simulator. It is presented as a spreadsheet style table. Each row represents an instruction supported by the simulator. The columns represent the prefix bits, suffix bits, mnemonic, operands (r - register, imm - immediate value) and simulation code. The simulation code is a suite of Python statements that simulate the effect of the instruction. New instructions can be added to the simulator by using spreadsheet style block copy operations to duplicate existing rows, after which the appropriate fields and simulation code are modified.

The instruction set table is used to drive an assembler, which parses instructions in textual form and produces binary machine code output. It is also used to drive the simulator. The simulator's instruction decoder uses the binary prefixes and suffixes along with the operand descriptions to determine which instruction is represented by a particular word from memory, and extract the operands. It then invokes the corresponding simulation code to simulate the instruction's effect.

In contrast, a typical textual implementation of a CPU simulator would rely on an if-else block to select the instruction to execute. The body of each if-clause would extract the operands and simulate the effects of the instruction. The table seen in figure 8.7 represents the intentions and concepts that underlie the simulator far more faithfully as implementation specific details are elided, leaving only salient information; its visual form is similar to that which would be used in a textbook. It is also more obvious to the user how to extend the simulator to support new instructions.

Our visual instruction table shares similarities with domain specific languages (DSLs) developed using the Intentional Domain Workbench [66, 65], in which tables can be used to augment code in a similar fashion.

8.1.6 Visual regular expression editor.

Our visual regular expression editor is a syntax recognizing editor that uses visual cues and spatial arrangement to enhance the readability of regular expressions. Its development was inspired by the use of a visual representation in SWYN [5]. Unlike SWYN however, it does not incorporate a PBE (programming by example) based system that infers regular expressions from examples, or provide incremental evaluation and testing. It is focused solely on enhancing comprehension while maintaining compatibility with Python's textual regular expression syntax. The use of a syntax recognizing editor allows it to recognise and display structure *as you type*. Figure 8.8 shows a regular expression for matching a date in both textual and visual form.

Visual regular expressions embed as expressions within Python source code. They evaluate to their string representation, allowing them to inter-operate with the regular expression tools within the Python standard library.

The visual regular expression editor can be a useful tool for a programmer in its own right, even if they are using an external editor as their primary development environment. The visual cues provided by our editor can simplify

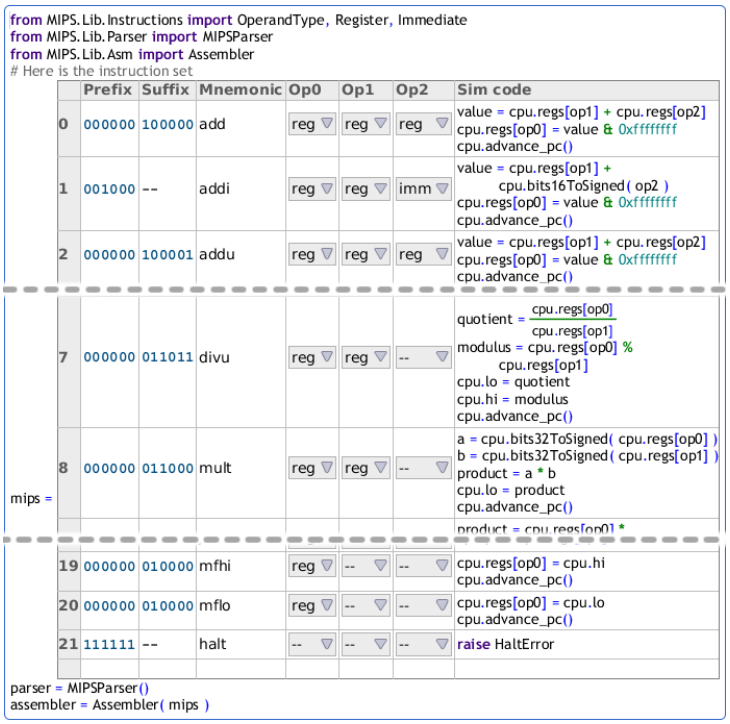


Figure 8.7: MIPS simulator instruction set, shown as an editable table

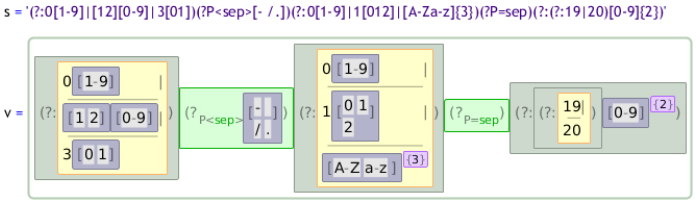


Figure 8.8: Visual regular expression editor, with textual form above for comparison

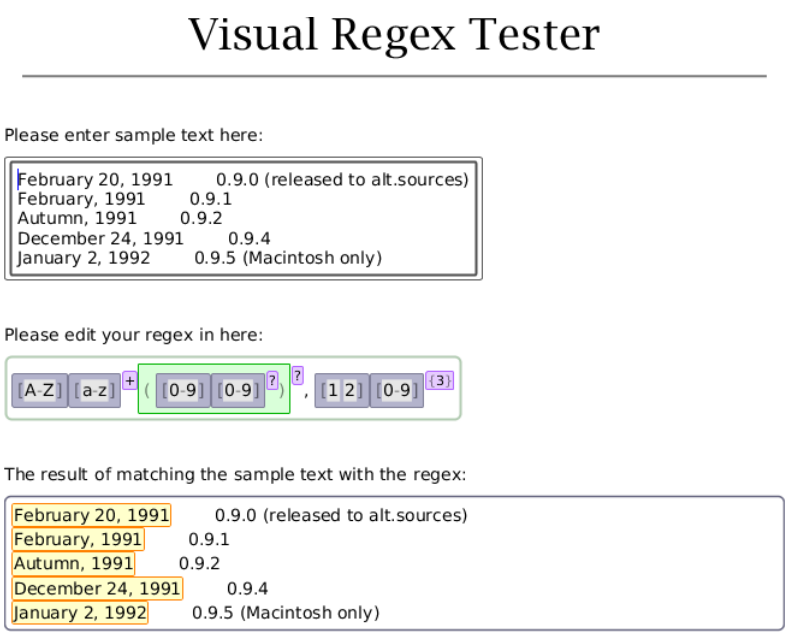


Figure 8.9: Regular expression tester

the process of building a regular expression, after which copy and paste can be used to transfer the regular expression in textual form to and from external applications.

We have used our regular expression editor, in concert with embedded editable values and worksheets to develop a regular expression testing application, shown in figure 8.9. Both the regular expression and the sample text area visible at the top were originally embedded within source code, as seen in the implementation in figure 8.10. References to both were embedded within the worksheet text, resulting in the interface that is seen. The source code declares the `regexTest` function, that matches the sample text against the regular expression, and returns a visual representation of the sample text with highlights around the matched segments. The `regexTest` function is converted into a live function by decorating it with `LiveFunction` (section 4.5, another example can be seen in figure 8.1). As a result, the highlighted text shown at the bottom updates in response to modifications to either the sample text or regular expression.

8.1.7 Table based unit tests

Unit tests are not usually located near the code they test; they are often in a different module or in a distant part of the same module. Developing the code alongside the tests requires the programmer to switch between the two. Additionally, testing code that generates complex data structures can be cumbersome, as the test must explicitly create the data structure in the expected form in order to compare it with the execution result. The results of running the tests are usually displayed in a separate window (such as a command shell) or within a different pane of an IDE.

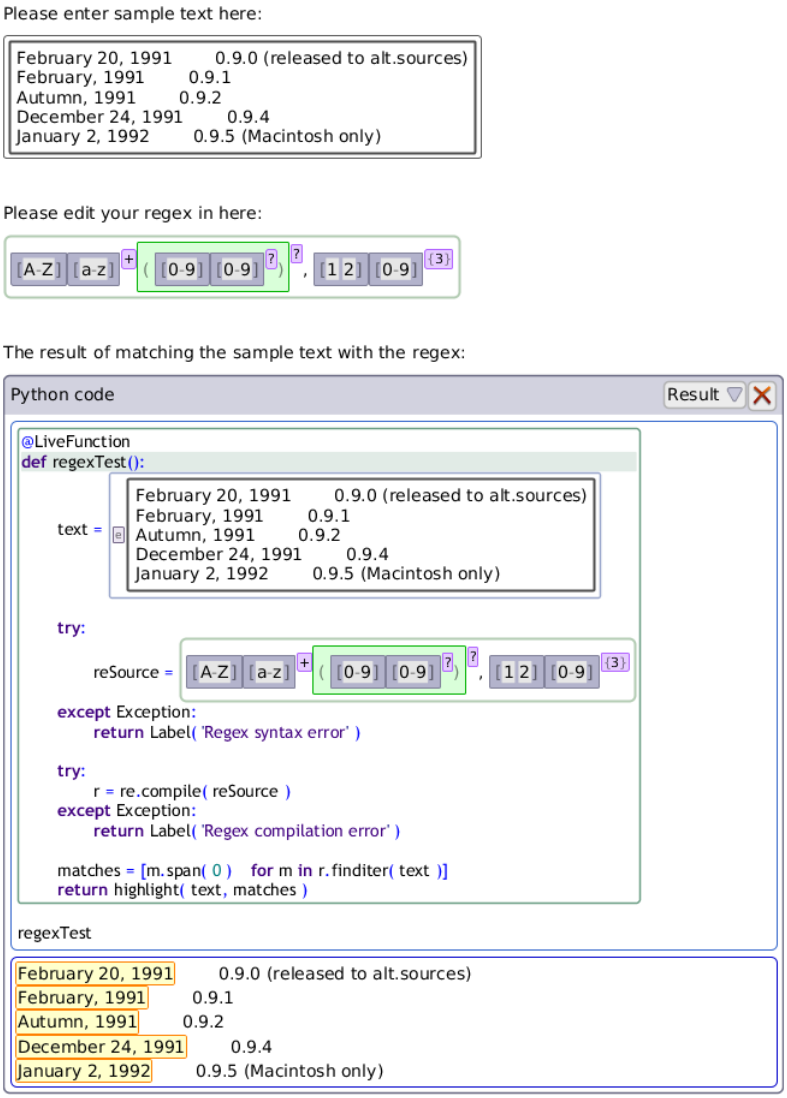


Figure 8.10: Regular expression tester implementation

(a)

```
@Rule
def add_sub(self):
        return (self.mul_div() + ((Literal('+') | Literal('-')) +
                self.mul_div()).zeroOrMore()).action(_leftInfixOpAction({'+':Add, '-':Sub}))
```

```
def test_addsub2(self):
        self.assertEqual( _g.add_sub().parseStringChars( 'a+b' ).getResult(),
                Add(Load('a'), Load('b')) )
        self.assertEqual( _g.add_sub().parseStringChars( 'a+b*c-d' ).getResult(),
                Sub(Add(Load('a'), Mul(Load('b'), Load('c'))), Load('d')) )
        self.assertEqual( _g.add_sub().parseStringChars( 'a-b*c-d' ).getResult(),
                Add(Sub(Load('a'), Mul(Load('b'), Load('c'))), Load('d')) )
```

(b)

```
@Rule
def add_sub(self):
    return (self.mul_div() +
        ((Literal( '+' ) | Literal( '-' )) + self.mul_div()).zeroOrMore()).action( _leftInfixOpAction( {'+':Add, '-':Sub} ) )
```

Unit tests `add_sub2`

	Result	Parser code	Input code	Expected
0	PASS Trace	_g.add_sub()	'a+b'	Success Add(Load(a), Load(b))
1	PASS Trace	_g.add_sub()	'a+b*c-d'	Success Sub(Add(Load(a), Mul(Load(b), Load(c))), Load(d))
2	No expected result, received: Success Add(Sub(Load(a), Mul(Load(b), Load(c))), Load(d)) Set expected result Trace	_g.add_sub()	'a-b*c+d'	

Figure 8.11: Parser unit test in textual and tabular form

Figure 8.11 (a) shows the textual form of a method defining a rule (`add_sub`) from a calculator grammar, and the corresponding unit test method (the dashes separating them indicate that the test method is located in a different part of the module). Notice that the unit test explicitly constructs the expected result structure (`Add(Sub(Load('a'), .....))`) in order to check the output generated by parsing the input text. In figure 8.11 (b), we see a table based form of the test, placed immediately below the method that it tests. The code that generates the parser and the code that generates the input are displayed in the *Parser code* and *Input code* columns. The expected result is displayed visually in the *Expected* column, with the structure highlighted with concentric boxes. The *Result* column tells the user whether the test passed or failed. The small *Trace* hyperlink (in each Result cell) is a debugging tool that displays a trace of the parsing process in the form of a graph. This assists the programmer in determining how the parser reached its result.

The table based test is easier to comprehend; the spatial layout guides the readers eye, while implementation details are elided, leaving only the salient information. The unit testing tool objects are embedded as macros (see section 7.1); the contents of the tables are converted into code which is appended to the module.

In row 2, we see a test that is not complete; no expected result has been supplied. In order to save the programmer from having to type out the structure by hand (manually entering the expressions in (a) can be cumbersome, especially for deeply nested structures more complex than those shown), the test can be executed without an expected result. The programmer is notified of this with the message shown in the *Result* column along with the received output; the programmer is invited to visually validate the output, and can choose to use it as the expected result in future test runs by clicking the *Set expected result* button. Doing so transfers the received output to the *Expected* column. This is how the unit tests in the table were created.

8.1.8 Live API Documentation

As they stand, worksheets can be used to create live API documentation. Rich text is used to explain the material under discussion. Examples take the form of code blocks. Within a worksheet, the examples are executed, with the results being displayed below. This simplifies the creation of API documentation, as there is no need to develop and execute examples in a separate environment, and insert results via cut and paste or as screenshots. Furthermore, a code block can be marked as editable, permitting the reader to modify and experiment with it, testing the effects of changes.

We have expanded on this approach by developing table based API examples, as seen in figure 8.12. In this case, we are documenting the API of LSpace, the Larch presentation system. In our example, the first column contains code that creates a style sheet that is applied to the presentation created in the second column. The result is displayed in the third. The user may edit the code and refresh to see the effect with a key-stroke. The API tables are implemented as objects that are embedded within a worksheet. The cells in the first two columns contain objects that represent Python code (section 6.1).

HTML based web pages are currently a popular format for API documen-

Style parameters

The style sheet attributes that affect text elements are found within the *Primitive* namespace. They are demonstrated here:

Style sheet code	Presentation code	Result
Press Ctrl+Enter to execute StyleSheet.style()	Press Ctrl+Enter to execute Label('Default style')	Default style
Press Ctrl+Enter to execute StyleSheet.style(Primitive.fontFace('Serif'))	Press Ctrl+Enter to execute Label('Set font face')	Set font face

Press Ctrl+Enter to execute
StyleSheet.style(Primitive.fontFace('Times New Roman; Serif'))
	Press Ctrl+Enter to execute
	# See note (1) on multiple font faces
	Label('Multiple faces given')
		Multiple faces given

Press Ctrl+Enter to execute StyleSheet.style(Primitive.fontSize(24))	Press Ctrl+Enter to execute Label('Set the font size')	Set the font size
Press Ctrl+Enter to execute StyleSheet.style(Primitive.fontSize(24), Primitive.fontScale(0.75))	Press Ctrl+Enter to execute Label('Scale the font size')	Scale the font size
Press Ctrl+Enter to execute StyleSheet.style(Primitive.fontBold(True))	Press Ctrl+Enter to execute Label('Bold')	**Bold**
Press Ctrl+Enter to execute StyleSheet.style(Primitive.fontItalic(True))	Press Ctrl+Enter to execute Label('Italic')	*Italic*
Press Ctrl+Enter to execute StyleSheet.style(Primitive.fontUnderline(True))	Press Ctrl+Enter to execute Label('Underline')	Underline
Press Ctrl+Enter to execute StyleSheet.style(Primitive.fontStrikethrough(True))	Press Ctrl+Enter to execute Label('Strikethrough')	~~Strikethrough~~
Press Ctrl+Enter to execute StyleSheet.style(Primitive.fontSmallCaps(True))	Press Ctrl+Enter to execute Label('Small Caps')	Small Caps
Press Ctrl+Enter to execute StyleSheet.style(Primitive.foreground(Color(0.0, 0.5, 0.0)))	Press Ctrl+Enter to execute Label('Change the colour')	Change the colour
Press Ctrl+Enter to execute StyleSheet.style(Primitive.foreground(Color.BLUE), Primitive.hoverForeground(Color.RED))	Press Ctrl+Enter to execute Label('Red when pointer hovering')	Red when pointer hovering

Notes:

(1). *Primitive.fontFace* - multiple font faces can be named; they are separated by semi-colons. Larch will pick the first one from your list that it finds on the system.

Shapes

Box creates a filled box. The two parameters are the width and height respectively.

Style sheet code	Presentation code	Result
Press Ctrl+Enter to execute StyleSheet.style()	Press Ctrl+Enter to execute Box(30.0, 30.0)	

Press Ctrl+Enter to execute
# FillPainter - see note (1) 	
StyleSheet.style(Primitive.shapePainter(FillPainter(Color.RED)))
	Press Ctrl+Enter to execute
	Box(30.0, 30.0)	

Press Ctrl+Enter to execute # hoverShapePainter sets the painter that is used to draw a shape when the pointer hovers over it StyleSheet.style(Primitive.shapePainter(FillPainter(Color.RED)), Primitive.hoverShapePainter(FillPainter(Color.BLUE)))	Press Ctrl+Enter to execute Box(30.0, 30.0)	

Notes:

(1). Painters will be discussed later

Figure 8.12: Live API documentation for LSpace

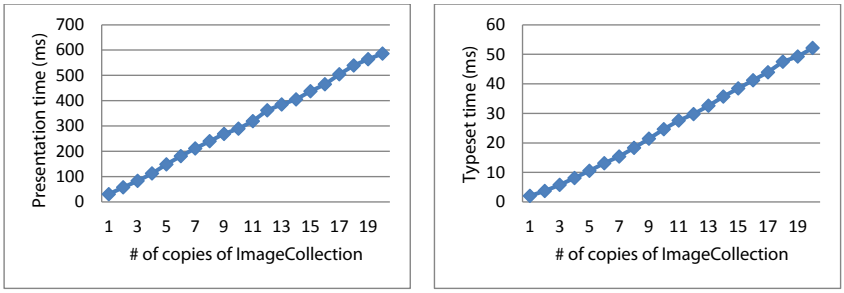


Figure 8.13: Performance measurements

tation[3], as it is convenient to access and navigate. The API is explained with prose, while source code and example output are presented as static text. The static nature of standard rich text documents imposes a number of limitations on their capabilities. The reader must accept on faith that documentation is accurate (this is not necessarily the case with some projects). Experimenting with the source code requires the user to copy and paste it into an external development environment in order to execute it. Results or output that are interactive can only be explained using a combination of screenshots and prose.

Live documents address these limitations. The output will always be consistent with the API implementation, as it is created by running the example code shown in the document; any inconsistencies between the API and the examples will be visible as either incorrect output or exceptions displayed in the output area. The user can experiment with the code shown by making small changes and refreshing the output to see their effect. Example output may include interactive content, allowing the user to experiment with code that creates interactive, GUI driven content within the API documentation.

8.2 Programming environment performance

To assess the performance of Larch we measured the amount of time required to present the contents of a worksheet while varying its size. We varied the size by appending copies of the `ImageCollection` code presented in figure 4.2, which consists of 2 lines of text and two code blocks containing 73 lines of code between them (including the import statements that were elided for brevity, 13 lines of comments, 5 blank lines). Each additional copy resulted in the creation of 1775 presentation elements to display it[4]. Each test was performed 43 times. The first three results were discarded; they typically took longer due to JVM JIT compiler warm-up. The median time of the remaining 40 results was selected and used. The test setup consists of an Intel Core 2 Quad 2.4Ghz

[3]Documentation for the standard libraries for the Java, C# and Python programming languages are all available in this form.

[4]Recall that Larch represents source code in an AST-style structured form (see section 6.1). Presenting source code involves walking the AST in order to present each node. The resulting presentation tree contains an element for each source code token, along with elements for spatial arrangement, hence the large number of elements.

Doc size	Lines of code	Code SLOC	Lines of ws text	# of elements	Presentation time (ms)	Typeset time (ms)
1	73	44	2	1813	30.9	2.1
2	146	88	4	3588	58.0	3.7
3	219	132	6	5363	83.5	5.8
4	292	176	8	7138	112.9	8.1
5	365	220	10	8913	148.7	10.6
6	438	264	12	10688	181.3	13.1
7	511	308	14	12463	211.5	15.4
8	584	352	16	14238	240.3	18.3
9	657	396	18	16013	268.9	21.4
10	730	440	20	17788	289.7	24.6
11	803	484	22	19563	319.4	27.6
12	876	528	24	21338	362.3	29.8
13	949	572	26	23113	385.3	32.6
14	1022	616	28	24888	405.3	35.7
15	1095	660	30	26663	437.7	38.5
16	1168	704	32	28438	465.1	41.2
17	1241	748	34	30213	504.6	43.9
18	1314	792	36	31988	539.3	47.5
19	1387	836	38	33763	564.1	49.3
20	1460	880	40	35538	586.2	52.2

Table 8.1: Performance measurements in numeric form

CPU, 8GB RAM, Windows Vista OS, 32-bit Java Hotspot 1.6.0-31 JVM and Jython 2.7-alpha1.

The graphs in figure 8.13 show the presentation and typesetting time with respect to the worksheet size. The presentation time is the time required to walk the document model and construct the complete presentation, while the typesetting time is the time required by the presentation system to spatially arrange the elements for display. Analysis of the results indicates that the presentation time is linear with respect to input size while the typesetting time is quadratic. The linear appearance of the typesetting graph is due to fact that the quadratic portion of our typesetting system is not sufficient to significantly affect the run-time at the document sizes tested. The performance measurements are also shown in table 8.1.

To assess the performance of Larch when handling larger worksheets, we imported the Larch Python language parser module, which consists of 2,524 lines of Python code. Using the same approach as above, we found the median presentation time to be 1,636ms, requiring the creation of 103,880 presentation elements. The performance of edit operations is variable, with presentation refresh times varying between 50ms and 200ms (when working on the aforementioned 2.5k lines of code). The variability in performance is due to the variability in the scope of the effects of an edit operation; many edits will affect only a single line of code, resulting in a quick refresh as only a small section of the structural source code model will be modified. Other operations will require rebuilding the structure of substantial parts of a block of code, requiring larger parts of the visual representation to be refreshed. A delay of 1.6s between choosing to open

a worksheet and seeing it, along with edit response times between 50ms and 200ms makes Larch feel sluggish when working with worksheets of this size.

We would suggest that the performance of Larch could be improved with the following techniques. Content could be presented incrementally. Rather than generating presentation elements for a complete page in one go, the page could be presented incrementally. Pages would appear — in an incomplete form — almost instantaneously, after which the remaining content would be gradually presented until the page is complete. This is done by most web browsers so that a page can be displayed in an incomplete form before it has finished downloading. The remainder of our proposed performance improvements focus on syntax recognizing editors. Our parsing library is quite slow, as it is based on recursive descent parsing and backtracking. We would propose implementing a tokenizer (the current approach consumes content directly) and using a finite state machine based parsing algorithm. We would also propose extending the parser so that it supports a form of incremental parsing, by allowing entries recorded into the parser memo table during a previous parse to be re-used when parsing a larger segment of a document that includes the content parsed previously.

8.3 A discussion of Cognitive Dimensions

When considered within the context of the Cognitive Dimensions Framework [26], the use of visual constructs within source code yields benefits in three dimensions. Closeness of mapping is improved as a visual representation can map more closely to the programmer's mental model of the problem domain. Hard mental operations are reduced by using visual cues to communicate the structure and relationships inherent in the textual code that they enhance or replace. Understanding complex textual code requires a reader to remember the relative sequential positions of elements and structures within the text. A good example of this can be seen in our visual regular expression editor (section 8.1.6) or SWYN [5]. Improvements in the terseness dimension are achieved by eliding implementation specific details, leaving only salient information visible to the user, as seen in the MIPS simulator (section 8.1.5) and the table based unit tests (section 8.1.7).

The Larch programming environment provides benefits in the progressive evaluation cognitive dimension, as programmers are encouraged to develop the programs incrementally. Worksheets — supplemented with the in-line console tool — provide an environment for testing incomplete programs and experimenting with segments of code; a block of code can be executed with a key-stroke, with the results generated being displayed immediately below. Given that objects (results) are displayed in a visual form, visual cues can be used to enhance the comprehensibility of results that are displayed. Our worksheets are only helpful for segments of code that execute in a linear fashion. This shortcoming prevents them from being used to understand the inner workings of functions or event handlers. This is addressed through the use of the in-line console tool (section 8.1.3) that allows these areas to be explored.

In the future, we would like to evaluate our system with respect to the Cognitive Dimensions Framework with an empirical study, with a view to guiding future development.

Chapter 9

Conclusions

The Larch Environment is a visual interactive programming environment that introduces a novel form of partially visual programming based around embedding visual objects within source code. The presence of our worksheet active document system allows Larch to be used to create programs that operate as interactive technical literature.

9.1 Discussion

9.1.1 Visual object presentation

Our use of type coercion and incremental consistency maintenance for visual object presentation eliminates much of the boilerplate code that is required in a typical MVC implementation. An object can be displayed simply by incorporating it into a presentation description. Views of objects are automatically created, maintained and destroyed by the presentation system as required. In contrast to systems such as Citrus [40] which require that the model and view must conform to a specific design, Larch requires very few design constraints. Any Java or Python object can be used as a data model and anything that can act as a function (e.g. a method) can create the view. As a consequence, presentations can be implemented for objects that are not designed with presentation in mind, such as those provided by the standard library. This is how the image objects in figure 4.1 are displayed. We have also implemented a presentation for Java `ResultSet` objects, displaying the results of SQL database queries in tabular form.

Implementing a presentation method is sufficient for specifying a basic visual representation for an object. Implementing a simple non-interactive visual representation that uses spatial layout is therefore not significantly more complex than defining a `toString` method.

Presentation by type coercion also provides a consistent and compose-able approach to presentation construction. In addition to enabling embedded objects, it allows programmers to easily develop visual representations for their own objects. An example of this can be seen in section 8.1.7, where a normal Python object defines a visual representation that is incorporated into the interface of a unit testing tool.

Our approach is sufficiently flexible to allow presentations to mix a variety of different content types. For example, Larch worksheets can contain a combination of source code displayed using a syntax recognizing structure editor, editable rich text, spreadsheet style tables and GUI based content, all freely mixed within one another.

9.1.2 Programming environment

Active document systems such as Mathematica notebooks and our worksheet system improve on the experimental programming environment provided by a REPL console. They retain the rapid edit-run-debug cycle while allowing the programmer to develop a complete module. The mix of rich text and executable source code provides an environment that naturally lends itself to the development of programs that take the form of interactive live documents.

Our worksheets can simultaneously serve as both development environment and user facing GUI. In developer mode, the code that underlies the functionality of a program is visible, while in user mode it is hidden, with only the rich text and the GUI (created by the code) shown to the user.

A standard GUI based Python program will display its user interface within a separate window that is created during the program's execution. As a consequence there is an apparent disconnect between the visible GUI and the code that creates it. Our worksheets reduce this disconnect, by juxtaposing them with one another.

Worksheets encourage an incremental approach to GUI development, in which small components of a GUI can be developed and tested individually. A programmer can create test worksheets whose purpose is to instantiate a part of a GUI and display it, in order to test the functionality of the component, separate from the rest of the application.

9.1.3 Partially visual programming

Partially visual programming by embedded objects allows source code to take on the appearance of interactive technical literature; programs remain to be mostly textual, with interactive visual content employed only where it is beneficial. Our approach is simpler to use than those of prior work in the field [19], as doing so does not require significant knowledge of interpreter and compiler internals or semantic analysis (although knowledge of AST manipulation is necessary for the use of the LISP-macro style facilities). A programmer with a good understanding of object-oriented programming and GUI development can implement their own visual constructs to be embedded within their programs. Our examples in section 8.1 demonstrate the flexibility of our approach. Embedded objects can improve the comprehensibility of complex textual code (section 8.1.6, also see figure 7.2). They can also customise the form of a program to suit a specific domain. This opens avenues for the development of domain specific, partially visual programming tools (section 8.7) and customised languages.

Our visual object presentation system, in concert with worksheets provide an environment suitable for the development of programs as visual, interactive literature. One piece of the puzzle remains missing however; while the output created by source code can be visual, interactive and dynamic, source code itself remains to be mostly static and textual in nature. This shortcoming is

addressed by our approach to partially visual programming. As a consequence, programs within Larch take the form of living, breathing, interactive documents. By blurring the boundaries between code and the interactive content that it creates, we support the development of new kinds of programming tools and environments.

9.2 Limitations

We have not yet determined how a version control system could manage source code that uses embedded objects, as it no longer takes the form of plain text. A textual source file with embedded object references — which refer to objects in an external file — could be handled by extending the comparison and merging tools to handle the modified format. Alternatively, the references could be stored in a plain text form, and specially handled and displayed by an embedded object aware editor. Comparing and merging the contents of the embedded objects themselves however is a different problem that has yet to be addressed.

It would also be difficult for a programmer using plain text based tools to work on a code base that contains embedded objects. Using the references as plain text approach described above is only a partial solution since the programmer would be unable to view or interact with the embedded objects, rendering the code base unreadable.

Our object presentation system is not designed for presenting large amounts of data; attempting to construct a presentation of millions of objects would take a considerable amount of time and most likely exhaust the available RAM. Dedicated data visualisation tool kits such as prefuse [29] are better suited for such tasks. Indeed, we are considering integrating prefuse with Larch in the future.

The program visualisation tool (section 8.1.4) and the unit testing tool (section 8.1.7) both rely on the editor and the program under development executing within the same process space (an approach pioneered by Smalltalk [22]), so that values generated at execution time can be displayed within the editor's code pane. In the future, we would like to extend our system so that the editor and the user's program can reside in separate processes, so that the program under development cannot modify the editor's internal data structures.

A number of design choices made during the development of Larch limit the scale of projects for which it can be used. The Python language, and therefore Larch is inherently dynamic; the interactive programming experience comes at the cost of static tools and analysis, and compile-time error detection. This precludes the development of large scale static program visualisations and automated refactoring tools, the likes of which are present in commercial IDEs.

Our system omits some important features, namely search and replace. This hampers a programmer's ability to modify and refactor their code. Such features are not infeasible; their omission is due to our choice of development priorities during the creation of this prototype.

9.3 Future work

Recent research into canvas based programming environments has yielded sys-

tems that allow the user to arrange source code on a canvas, utilising their spatial memory to assist in locating code segments during the development process. Code Canvas [14] allows a user to spatially arrange files, classes and methods on a zoom-able canvas, and use visual containers and Photoshop-style layers to conceptually group them together. Code Bubbles [8] offers a new and innovative take on the browser interface employed by many IDEs; code segments — namely classes and methods — are displayed within bubbles. Navigating to other code segments opens the target segment in a new adjacent bubble, causing the browsing history to take the form of a chain of bubbles in the workspace. The Fluid source code editor [15] takes an alternative approach; call sites can be expanded to show the source code of the target method in-line. In the future, we would like to extend Larch so that it can support free-form canvas based programming environments. Such environments offer a number of exciting possibilities, especially when considered for use on multi-touch devices.

Bibliography

[1] Pharo Open Source Smalltalk. http://www.pharo-project.org/home.

[2] Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter. The pan language-based editing system. *ACM Trans. Softw. Eng. Methodol.*, 1:95–127, January 1992.

[3] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. Cduce: an xml-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ICFP '03, pages 51–63. ACM, 2003.

[4] Benjamin E. Birnbaum and Kenneth J. Goldman. Achieving flexibility in direct-manipulation programming environments by relaxing the edit-time grammar. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 259–266. IEEE Computer Society, 2005.

[5] Alan F. Blackwell. *SWYN: a visual representation for regular expressions*, pages 245–270. Morgan Kaufmann Publishers Inc., 2001.

[6] Marat Boshernitsan. Harmonia: A flexible framework for constructing interactive language-based programming tools. Technical report, Berkeley, CA, USA, 2001.

[7] Gilad Bracha. Newspeak — The Newspeak Programming Language. http://newspeaklanguage.org/.

[8] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 2503–2512. ACM, 2010.

[9] Steve Burbeck. Applications programming in Smalltalk-80^{TM}: How to use model-view-controller (MVC). http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html, 1992.

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

[11] Margaret Burnett, John Atwood, Rebecca Walpole Djang, James Reichwein, Herkimer Gottfried, and Sherry Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11(02):155–206, 2001.

[12] Vassili Bykov. Hopscotch: Towards user interface composition. Technical report, Cadence Design Systems. San Jose, CA 95143, 2008.

[13] John Coker. Internals of vortex: The source editor. Technical report, Berkeley, CA, USA, 1988.

[14] Robert DeLine and Kael Rowan. Code canvas: zooming towards better development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 207–210. ACM, 2010.

[15] Michael Desmond and Chris Exton. An evaluation of the inline source code exploration technique. In *Psychology of Programming Interest Group 2009*, PPIG'09, 2009.

[16] Django Software Foundation. The Django template language. https://docs.djangoproject.com/en/1.4/topics/templates/.

[17] Django Software Foundation. The web framework for perfectionists with deadlines — django. https://www.djangoproject.com/.

[18] ECMA International. *ECMA-262: ECMAScript Language Specification.* Third edition.

[19] Andrew D. Eisenberg. *Presentation Techniques for More Expressive Programs.* Phd thesis, University of British Columbia, Vancouver, Canada, 2008.

[20] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time (functional pearl). In *ICFP'02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pages 36–47, 2002.

[21] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122. ACM, 2004.

[22] Adele Goldberg and David Robson. *Smalltalk-80 - The Language and its Implementation.* Addison-Wesley, 1983.

[23] Google. Google chrome - get a fast new browser. for pc, mac, and linux. http://www.google.com/chrome/.

[24] Susan L Graham. Language and document support in software development environments. Technical report, Darpa Software Technology Meeting, 1992.

[25] T. R. G. Green and M. Petre. When Visual Programs are Harder to Read than Textual Programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6 (6th European Conference Cognitive Ergonomics)*, 1992.

[26] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7:131–174, 1996.

[27] Keith Hanna. A document-centered environment for haskell. In *Implementation and Application of Functional Languages*, volume 4015 of *Lecture Notes in Computer Science*, pages 196–211. Springer Berlin / Heidelberg, 2006.

[28] R. Heckmann and R Wilhelm. A functional description of tex's formula layout. *Journal of Functional Programming*, 7:451–485, September.

[29] Jeffrey Heer, Stuart K. Card, and James A. Landay. prefuse: a toolkit for interactive information visualization. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '05, pages 421–430. ACM, 2005.

[30] Juha Helminen and Lauri Malmi. Jype - a program visualization and programming exercise tool for python. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 153–162. ACM, 2010.

[31] Ianthe S. A. Hind. Extending the capabilities of anastasia. Technical report, Edinburgh, Scotland, UK, 2006.

[32] Microsoft Inc. The Official Site of Visual Studio 2010. http://www.microsoft.com/visualstudio/en-gb.

[33] Microsoft Inc. XAML Overview (WPF). http://msdn.microsoft.com/en-us/library/ms752059.aspx.

[34] Jetbrains. Jetbrains :: Meta programming system - language oriented programming environment and dsl creation tool. http://www.jetbrains.com/mps/.

[35] Olov Johansson. Describing live programming using program transformations and a callstack explicit interpreter. Master's thesis, Linköping University, 2006.

[36] Tim S Jones. *Seamless Interaction Facilities in Structure-Oriented Software Development Environments*. Phd thesis, The University Of Queensland, Queensland, Australia, 2000.

[37] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[38] Donald E. Knuth. Literate programming. *Computing Journal*, 27:97–111, May 1984.

[39] Donald E. Knuth. *The TeXbook*. Addisson Wesley, 1984.

[40] Andrew J. Ko and Brad A. Myers. Citrus: a language and toolkit for simplifying the creation of structured editors for code and data. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*, UIST '05, pages 3–12. ACM, 2005.

[41] Andrew J. Ko and Brad A. Myers. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, pages 387–396. ACM, 2006.

[42] Andrew Jensen Ko. Designing a flexible and supportive direct-manipulation programming environment. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, VLHCC '04, pages 277–278. IEEE Computer Society, 2004.

[43] Barbara S Lerner. Contrasting approaches of two environment generator: The synthesizer generator and pan. Technical report, Amherst, MA, USA, 1993.

[44] Catherine Letondal, Stphane Chatty, W. Greg Philips, Fabien Andr, and Stphane Conversy. Usability requirements for interaction-oriented development tools. In *Psychology of Programming Interest Group 2010*, PPIG'10, 2010.

[45] V I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics-Doklady*, 10:707–710, February 1966. Wikipedia: http://en.wikipedia.org/wiki/Levenshtein_distance.

[46] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *Trans. Comput. Educ.*, 10:16:1–16:15, November 2010.

[47] Maplesoft. Math software for engineers, educators, and students. http://www.maplesoft.com/.

[48] Vance Maverick. *Presentation by Tree Transformation*. Phd thesis, Computer Science Division, University of California. Berkeley, CA, 1997.

[49] Sean McDirmid. Living it up with a live programming language. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 623–638. ACM, 2007.

[50] Sean McDirmid. Coding at the speed of touch. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, ONWARD '11, pages 61–76. ACM, 2011.

[51] Ethan Vincent Munson. *Proteus: An Adaptable Presentation System for a Software Development and Multimedia Document Environment*. Phd thesis, Computer Science Division, University of California, Berkeley. Berkeley, CA, 1994.

[52] B. A. Myers. Encapsulating interactive behaviors. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Wings for the mind*, CHI '89, pages 319–324, New York, NY, USA, 1989. ACM.

[53] Brad A. Myers. Taxonomies of visual programming and program visualization. *J. Vis. Lang. Comput.*, 1:97–123, March 1990.

[54] National Instruments. NI LabVIEW - Improving the productivity of engineers and scientists. http://www.ni.com/labview/.

[55] Nokia. Qt 4.7: Introduction to the QML language. http://doc.qt.nokia.com/4.7-snapshot/qdeclarativeintroduction.html.

[56] Oracle. Swing (java foundation classes). http://docs.oracle.com/javase/6/docs/technotes/guides/swing/.

[57] Terence Parr. ANTLRWorks: The ANTLR GUI Development Environment. http://www.antlr.org/works/.

[58] Fernando Pérez and Brian E. Granger. IPython: a System for Interactive Scientific Computing. *Comput. Sci. Eng.*, 9(3):21–29, May 2007.

[59] Simon Peyton-Jones and J. Hughes. Haskell 98: A non-struct, purely functional language. http://www.haskell.org/onlinereport/.

[60] Racket. Racket. http://racket-lang.org/.

[61] Sage. Sage: Open Source Mathematics Software. http://www.sagemath.org/.

[62] Martijn Michiel Schrage. *Proxima: A presentation-oriented editor for structured documents*. Phd thesis, University of Utrecht. Utrecht, Netherlands, 2004.

[63] Martijn Michiel Schrage and Johan Jeuring. XPREZ: A declarative presentation language for XML. Technical report, Institute of Information and Computing Sciences, University of Utrecht, The Netherlands, 2003.

[64] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional software. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 451–464. ACM, 2006.

[65] Intentional Software. Intentional software. http://intentsoft.com/.

[66] Intentional Software and Microsoft Corporation. Intentional software at dsl devcon 2009. http://msdn.microsoft.com/en-us/data/dd727740.aspx, 2009.

[67] J. Stasko, J. Domingue, M. H. Brown, and B. A. Price. *Software Visualization*. The MIT Press, 1998.

[68] Masato Takeichi, Zhenjiang Hu, Kazuhiko Kakehi, Yasushi Hayashi, Shin cheng Mu, and Keisuke Nakano. Treecalc: towards programmable structured documents. In *In Japan Society for Software Science and Technology*, 2003.

[69] The GTK+ Team. The GTK+ Project. http://www.gtk.org.

[70] The Python Software Foundation. The jython project. http://www.jython.org.

[71] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. Touchdevelop: programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, ONWARD '11, pages 49–60. ACM, 2011.

[72] Just van Rossum and Paul Moore. Pep 302 – new import hooks. http://www.python.org/dev/peps/pep-0302/.

[73] Bret Victor. tangle. http://worrydream.com/#!/Tangle.

[74] Bret Victor. Ten brighter ideas. http://worrydream.com/#!/TenBrighterIdeas.

[75] W3C. Cascading style sheets level 2 revision 1 (css 2.1) specification. http://www.w3.org/TR/2009/CR-CSS2-20090908.

[76] W3C. Extensible markup language (xml) 1.1. http://www.w3.org/TR/2006/REC-xml11-20060816/.

[77] W3C. HTML 5 - W3C Editor's Draft. http://dev.w3.org/html5/spec.

[78] Alessandro Warth and Ian Piumarta. Ometa: an object-oriented language for pattern matching. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, pages 11–19. ACM, 2007.

[79] Alessandro Warth, Takashi Yamamiya, Yoshiki Ohshima, and Scott Wallace. Toward a more scalable end-user scripting language. In *Proceedings of the Sixth International Conference on Creating, Connecting and Collaborating through Computing (c5 2008)*, pages 172–178. IEEE Computer Society, 2008.

[80] Richard C. Waters. Program editors should not abandon text oriented commands. *SIGPLAN Not.*, 17:39–46, July 1982.

[81] Jim Welsh and Jun Han. Software documents: Concepts and tools. *SOFTWARE — CONCEPTS AND TOOLS*, 15:12–25, 1995.

[82] Brian T. Westphal. The redwood programming environment. Master's thesis, University of Nevada. Reno, 2004.

[83] K. N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, 8:109–142, 1996.

[84] Gregory V. Wilson. Extensible programming for the 21^{st} century. *ACM Queue*, 2:48–57, 2004.

[85] Wolfram Research Inc. Computable document format. http://www.wolfram.com/cdf/.

[86] Wolfram Research Inc. Mathematica, technical and scientific software. http://www.wolfram.com/.

[87] Wolfram Research Inc. *Dynamic Interactivity - Wolfram Mathematica Tutorial Collection*. Wolfram Research Inc., 2008.

[88] Wolfram Research Inc. *Notebooks and Documents - Wolfram Mathematica Tutorial Collection*. Wolfram Research Inc., 2008.

[89] Takashi Yamamiya, Alessandro Warth, and Ted Kaehler. Active essays on the web. In *C5 '09: Proceedings of the 2009 Seventh International Conference on Creating, Connecting and Collaborating through Computing*, pages 3–10. IEEE Computer Society, 2009.