Live Programming with Code Portals

Alexander Breckel and Matthias Tichy

Institute of Software Engineering and Compiler Construction
Ulm University, Germany
{alexander.breckel,matthias.tichy}@uni-ulm.de

Abstract

Programming is often a cycle between programming activities and understanding the results of those activities. Hence, modern development environments support many different tools to increase the speed of development, e.g., for refactoring or to visualize variable values or type information in a running program. However, current development environments provide these different tools in various ways, from wizards for refactorings with special preview pages, to tooltips and watch views for showing specific information. In this paper, we present code portals as a generic technical concept to enable showing live and continuously updated information about the program and its state. We illustrate code portals on three different types of programming languages, procedural, functional and homoiconic. A qualitative user study shows that code portals and their applications are seen helpful by users. A video demonstration can be found at: https://goo.gl/PumhQR

Keywords live evaluation, refactoring, function inlining, code portals, source code editor

1. Introduction

Contemporary integrated development environments (IDEs) often provide, unlike the name suggests, only little integration between different editor features. Information like, for example, debugging state or code structure, is often accessed through a multitude of visually and conceptually separate views, instead of being displayed within the main source code view. Furthermore, in many cases this information needs to be requested and updated explicitly.

Source code changes, like various types of refactorings, are often implemented in wizards, which only provide specific preview capabilities and only support one refactoring at a time. If developers identify a mistake when using such a wizard, they typically need to start from scratch or go back several steps, which makes performing refactorings a rather heavy-weight process.

Different approaches already exist which support live preview of various development activities. For live-evaluation of expressions, some approaches use a separate view [6], whereas others embed results directly under [10], beside [7], or near the [1] respective expression in the source code view.

In previous work [3], we introduced *code portals*, a mechanism to integrate arbitrary textual information into source code documents. Code portals provide a generic and versatile way to provide both, live-evaluation for expressions, and live-previews for editor features, from right within the main source code view. In contrast to other techniques of embedding additional information inside the source code view, however, code portals maintain complete editability of embedded content, while providing a low visual overhead and a handling already familiar to programmers. Integrated content therefore can be selected, copied, and modified using existing text editing operations.

This paper builds upon this previous work [3], where we present how code portals can be utilized to present programming context and improve code comprehension. In this paper, we extend our previous work on live-evaluation to three different types of programming languages (functional, imperative, homoiconic) to show the genericity of the code portal concept. Further, we show how code portals can be used for live-previews of program refactorings, including metaprogramming in homoiconic languages. Finally, we present a generic mechanism to enable live-previews using code portals for various typical editor features.

We have implemented all features presented in this paper in a prototypical source code editor. Based on this implementation, we have furthermore performed an initial qualitative usability study, which suggests that the presented techniques can be quickly grasped by programmers and help in understanding and writing source code.

After an introduction to code portals in the next section, we describe different types of live-information which can be embedded into source code in Sections 3 and 4, and illustrate them with concrete examples. Section 5 provides a general-

CONF 'yy Month d–d, 20yy, City, ST, Country

ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00
DOI: http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

ization of these use-cases that facilitates live-previews for a wide set of editor features. In Sections 6 and 7, we present our prototypical implementation of a standalone text editor, and initial results of a qualitative study. After a discussion of related work in Section 8, we conclude and give an outlook on future work.

2. Code Portals

Existing programming editors support a wide variety of ways to display additional information inside the main source code view. For example, many IDEs display API documentation or compilation errors with tool-tips or overlays. While this in principle represents a non-intrusive way to display optional information, the fact that tool-tips and similar concepts obstruct the view on underlying code makes them unsuitable for a permanent display of live-evaluation results. Therefore, some tools [7, 10] with live-evaluation capabilities choose to embed live-information in between existing source code locations. This minimizes the visual distance between evaluation results and respective code locations and clarifies the pairing of code and results in the case of multiple simultaneous embeddings. However, these additional embeddings are not part of the surrounding code document and cannot be selected and manipulated directly with existing editor operations.

Code portals follow a different approach. Similar to non-obstructive embeddings known from other tools, code portals provide a way to embed textual content into the source code document. However, the content is – character by character – integrated into the surrounding source code. This makes it possible to select, copy, and in some cases even manipulate embedded content, just as if its characters were part of the original code document. Analysis tools, and the compiler itself, though, can safely ignore all embedded content to avoid syntactic or semantic errors.

To illustrate this, the following example, taken from [3], shows a definition of an empty Java class `PersistentDB`, which derives from an interface `DB` further down in the file:

```
1  class PersistentDB implements DB {
2  ····
3  }
   ...
76 interface DB extends Serializable {
77 ····void store(String k, String v);
78 ····String retrieve(String k);
79 }
```

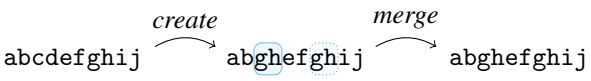
In order to implement the class `PersistentDB`, a programmer likely needs access to the interface definition of `DB`. This is normally achieved by opening a second editor view, or navigating back and forth between the corresponding code locations. With our approach, however, the programmer can instead open a code portal showing the definition of `DB` right within the code of `PersistentDB`:

```
1  class PersistentDB implements DB {
2  ····
76 interface DB extends Serializable {
77 ····void store(String k, String v);
78 ····String retrieve(String k);
79 }
3  }
   ...
76 interface DB extends Serializable {
77 ····void store(String k, String v);
78 ····String retrieve(String k);
79 }
```

The code portal, visible at the top, is highlighted with a blue border and background, but otherwise uses the same font and mono-space grid as the surrounding code. Line numbers are adjusted to indicate where the content originates from. The dotted area at the bottom represents the source region of the code portal further down in the file and is displayed using a matching blue color.

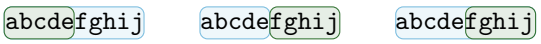
The Link between the source and destination area of the portal is maintained, and modifications within a code portal also affect the original code location, which inspired the name *portal*. Conversely, changes to the original code locations, e.g., when performing refactoring operations, are also immediately visible within the code portal. This allows the programmer to extend the interface definition, or work on both locations in parallel without having to navigate back and forth.

Unlike similar visualization mechanisms, code portals represent first-class citizens within a document and can therefore be nested, which allows a composition of different editor features. After creation, code portals can subsequently be moved, resized, closed, and have their contents merged into the underlying code location. Merging works by replacing the code underneath a portal, which is empty in the aforementioned example, with the code inside the portal. This has the effect that all visible characters stay in place while the portal borders disappear:



Here, a portal is created hiding the characters `cd` and showing the characters `gh` from further right in the string instead. Merging the portal replaces `cd` with `gh`.

Portals are displayed by a combination of borders and colors. Rounded corners allow the user to distinguish between different nesting levels and overlappings:



In the left example, the green portal is nested within the blue portal. In the middle example, both portals are side-by-side, whereas the right example shows an inverted nesting.

Colors indicate linked code fragments of the same origin. The source and target regions of a portal are surrounded by dotted and solid frames respectively. To further emphasize this linkage, we also display ghost cursors and ghost selections in locations affected by active cursors and selections:

```
let x=42; y=13 in 42 + 13
```

Here, two portals were created on the right showing the respective values from the left. The main cursor, shown in black, is contained within the blue portal `42` on the right. A corresponding ghost cursor, shown in grey, is displayed in the source region `42` on the left. Performing a modification, like adding a new character `5`, will not only affect the main cursor location, but also the ghost cursor location within the source region:

```
let x=452; y=13 in 452 + 13
```

In addition, portals containing active cursors are highlighted, so that the user can distinguish between cursor positions at the edges of nested portals when moving the cursor around in the document:

```
outerinner outerinner outerinner outerinner
```

First, the cursor is contained within the outer portal behind the character `e`. Moving the cursor to the right first places it behind the character `r`. A subsequent movement then switches up into the green portal before the character `i`. As the cursor did not visibly move between the two middle positions, we instead help the user distinguish between such positions by highlighting the currently active portal.

Because of the inline representation, a multi-line portal loses the original alignment of its contents if the source and target column offsets differ. This complicates the usage of indentation-aware programming languages. We compensate this without inserting or removing any characters by locally indenting the whole editing area by the column difference. For cases where the target column is further left than the source column, the editing area extends to the left over the edge of the main document area:

```
1  f = n => if n == 0
2  ·····then 1
3  ·····else f (n - 1) * n;
4  print((n => if n == 0
   2  ·····then 1
   3  ·····else f (n - 1) * n)(7));
5  g(n => if n == 0
2  ·····then 1
3  ·····else f (n - 1) * n);
```

Here, code portals were used to inline a function definition, as will be further explained in Section 4. Note the consistent alignment of `if`, `then` and `else`, which would not have been maintained if all lines started at the same column.

Among other use-cases, we utilize code portals to display various types of live-evaluation and live-previews for editor features. The following two sections describe these types in more detail and provide corresponding examples.

3. Live-Evaluation

In order to explore the limits and possibilities of live-evaluation in conjunction with code portals, we wanted to work with different programming languages and paradigms. Our goal was to support example-driven development [6] and provide extensive debugging functionality. Unfortunately, the lack of libraries providing a syntactical and semantical analysis of mainstream programming languages complicates the development of general purpose editors with diverse language support. We have therefore developed and integrated three custom programming languages as feasibility studies for different programming paradigms:

SLambda A pure functional language with Hindley-Milner type-inference and a Haskell-like syntax.

Imp A procedural dynamically typed scripting language with Javascript-like syntax.

Quasi A homoiconic functional language based on Lisp's S-Expressions with support for macros.

All three languages follow lexical scoping rules, which simplifies the live-evaluation of nested sub-expressions as references to non-local identifiers can be inferred without having to execute all surrounding code.

The following sections will first describe the differences in evaluation strategies for different programming paradigms. Afterwards, we will present concrete ways to realize live-evaluation with code portals, both on value and type level, as well as on a meta-programming level.

3.1 Live-Evaluation for different paradigms

Function and procedural programming paradigms are – in varying extents – suited for evaluation and exhibit distinct properties. While functional languages, in particular purely functional ones excluding side-effects, offer many opportunities to evaluate sub-expressions locally, the execution of procedural code often requires more context information due to its state-based nature. Take, for example, the following code written in our functional language SLambda:

```
1  sum = xs => case xs
2  ···········of Nil       => 0
3  ···········of Cons h t => h + sum t;
4  main = let x = [1, 2, 3];
5  ·····in sum x;
```

The code contains a function definition for `sum`, which takes a list of numbers `xs` and recursively calculates the sum of all elements, based on whether the list is empty or non-empty. A second definition `main` contains a call to `sum` with a locally defined list `x`, which, when evaluated, returns the value `6`. This code example contains several highlighted sub-expressions that can be evaluated statically:

All number literals are trivially evaluable. The list expression `[1, 2, 3]` can also be evaluated statically, as it does not reference any non-local identifiers. Although the expressions `x` and `sum x` in line `5` reference the identifier `x`, which is locally defined within the let-expression, as well as `sum`, which is defined globally, this does not prevent evaluation, as all referenced identifiers taken by themselves are also transitively evaluable.

In contrast to this, the expressions in line `3` contain identifiers that depend on the value of the formal function parameter `xs`, which, when looking only at the function definition itself without a corresponding function application, can not be inferred statically.

The same functionality, implemented in our procedural scripting language Imp, results in the following code:

```
1  var x   = [1, 2, 3];
2  var sum = 0;
3  var i   = 0;
4  while(i < x.length) {
5  ····sum += x[i];
6  ····i   += 1;
7  }
8  print(sum);
```

Here, the sum of all numbers is calculated by sequentially iterating over all list elements of `x` with a counter variable `i` in a while-loop. While this code still contains a multitude of evaluable sub-expressions, again highlighted with a green border, most of them contain constant values, which provide fairly useless evaluation results. The actual result of the program, namely the value of the variable `sum` in line `8`, can only be inferred by executing the whole example code. In some cases a smart runtime implementation might be able to reduce this amount of code with slicing and advanced program analysis. However, this quickly gets complicated with growing code examples. Note that, although most modern procedural programming languages also support a functional style for writing code without depending on mutable state, this is not (yet) widespread among programmers.

3.2 Live-Evaluation using Code Portals

We will now present how code portals can help to integrate live-evaluation results into the code document. Similar to existing tools, our approach is to display evaluation results in close proximity to their corresponding expressions. In the following example, the user has requested an evaluation of the three expressions `xs`, `[1, 2, 3]`, and `sum x`:

```
1  sum = xs => case xs = value of xs unknown
2  ····------····of Nil       => 0
3  ····------····of Cons h t => h + sum t;
4  main = let x = [1, 2, 3] = [1, 2, 3];
5  ····----····in sum x = 6;
```

Values are displayed to the right of their corresponding expressions and are highlighted using the same color. Highlighting the original expressions is also important to allow a precise subsequent modification of the code. The expression `xs` can not be evaluated in isolation due to the missing function argument and results in an error message. The other expressions `[1, 2, 3]` and `sum x`, however, are evaluable. All results are kept up-to-date by recalculating them after each modification within the whole source code. Individual evaluations can be created and closed in arbitrary order.

When working with functional languages, programmers are often also interested in the types of sub-expressions, as types are usually inferred and therefore not explicitly visible in the code. Similar to a live-evaluation, we therefore also offer a live-display of inferred types:

```
   sum :: [Int] -> Int
1  sum = xs => case xs :: [Int]
2  ····------····of Nil       => 0
3  ····------····of Cons h t => h + sum t;
4  main = let x :: [Int]
   ·····------·····x = [1, 2, 3];
5  ····----····in sum x :: Int;
```

Following the usual convention of Haskell-like syntax, we display types of expressions as post-fix annotations, and types of definitions in a separate line above. Just like with values, all types are freshly inferred after each modification anywhere in the code, which can be useful when debugging type errors.

For our procedural language Imp, a live-evaluation triggers the execution of the complete source code. While this of course is not practical for real world code, it provides a proof-of-concept with unnoticeable delays for most small code examples. Serious implementations could instead use slicing to reduce the amount of code needed to produce correct results. During the execution, all values of requested expressions are collected and displayed at the corresponding locations. Expressions evaluated multiple times during a single execution produce traces instead of singular values, as can be seen in the following code example:

```
1  var x   = [1, 2, 3];
2  var sum = 0;
3  var i   = 0;
4  while(i < x.length = [t, t, t, f]) {
5  ····sum += x[i] = [1, 3, 6];
6  ····i = [0, 1, 2]   += 1;
7  }
8  print(sum = 6);
```

Note that the `while`-condition in line 4 is evaluated 4 times, whereas the loop-body is executed only 3 times. The evaluation result of the `while`-condition contains three `true` values, followed by one final `false` when exiting the loop. Also, the tracing for `sum` in line 5 shows the values *after* the increment assignment, whereas the tracing in line 6 happens *before* each increment.

Having quick access to live-evaluation, tracing and types enables an example-driven development of code, where a programmer first writes a set of simple test cases for a not-yet-written functionality, triggers a live-evaluation (which probably only displays a set of errors), and then proceeds to implement the actual functionality. As the implementation grows, more and more test cases produce meaningful results. In our experience, this immediate – and always present – feedback allows programmers to detect and fix mistakes earlier in the development cycle, and reduces the typical roundtrip-time during debugging.

In contrast to other tools, the live-results are not merely displayed at the appropriate code locations. Instead, the information is integrated into the surrounding source code and can be accessed and manipulated using familiar editing operations. This makes it possible to, for example, select and copy the characters `sum = 6` in line 8 and replace the original definition in line 2.

Programmers can use this deliberately to write temporary code that, when evaluated, produces a value they need to include into the surrounding code. If, for example, one needs to sort the elements in a list literal, one can wrap the literal with a call to a sorting function, evaluate the call, and replace the original literal with its sorted result. In order to do this, one does not have to learn a separate scripting language provided by the IDE, but instead can use the same language as the surrounding code, with full access to its standard library and all user-defined functions. This gives programmers access to a novel form of meta-programming, even for languages that do not natively support it.

3.3 Live-Evaluation for Meta-Programming

On the other hand, languages that *do* support native meta-programming can use live-evaluation to support source-level transformations. Take, for example, the following code written in our homoiconic language Quasi:

```
1  (defmacro (swap x y) `(,y ,x))
2  (swap '(a b c) print)
```

The syntax is based on Lisp's S-expressions, which explicitly encode syntax trees as nested lists. The first line defines a macro that transforms an expression of the form `(swap x y)` for arbitrary expressions `x` and `y` into the form `(y x)`. The symbols `` ` `` and `,` make sure that `x` and `y` remain unevaluated in the resulting expression. In the second line, the macro is applied to the list `(a b c)` as `x`, and the function `print` as `y`, which transforms the code to `(print '(a b c))`. Executing this code therefore prints `(a b c)`.

Macros are used extensively in Lisp-like languages to perform code transformations at runtime. However, using live-evaluation, programmers can perform these transformations at source-level. For example, a programmer might have actually written the expressions `print` and `(a b c)` in the wrong order. Instead of reversing the order by hand or trying to find an appropriate IDE refactoring feature to perform the swapping, the programmer can simply prefix the code with `swap` and trigger a live-evaluation:

```
1  (defmacro (swap x y) `(,y ,x))
2  (swap '(a b c) print = print '(a b c))
```

The old code can then be quickly replaced by the corrected version using familiar editing operations. Note that the macro transformation automatically maintained the links between the old and new version by referencing the original code fragments through the red and green code portals. This means that further modifications, like appending a forth list element `d`, can also be performed in the evaluation-result instead of the original code.

By supporting user-defined macros, this live-evaluation facilitates complex and highly specific code transformations, that would otherwise be impossible using pre-defined refactoring operations provided by a development environment. For languages without native meta-programming, however, such syntax-level code transformations still must be performed through manual and tool-assisted refactoring operations. The following section describes ways to apply the ideas of live-evaluation also to such editor operations.

4. Live-Preview of Refactoring Operations

Modern IDEs offer many features to perform high-level modifications to source code, like advanced search/replace functionality or refactoring operations. In order to apply such features, IDEs often implement a two-step process, where the programmer first has to provide several configuration values, like setting the new name for an identifier renaming or other options to adjust the behavior. After pressing `OK`, the corresponding code transformation is applied to the document. If the user notices later on, that the new version does not match his expectations, the changes can be

undone and re-applied with a different configuration. In fact, Vakilian et al. [13] have found that this workflow is quite common among programmers working with IDEs.

A more user-friendly approach is to provide a live-preview of all expected changes. However, providing this usually involves a large overhead for tool-developers, which is likely avoided for rarely used features.

Code portals offer a simple and versatile way to provide such live-previews for code transformations. Instead of applying the changes to the document, code portals displaying the expected result can be created instead. After each change to the document or the feature-configuration, all portals can be closed and reopened in order to keep the preview always up-to-date.

We have applied this concept to string replacements based on regular expressions. The following HTML-code contains an integrated regular expression in Perl-syntax that adds the path prefix `img/` to all image URLs matching the pattern `="(.*?\.jpg)"`:

```
1  <html>
2  ·<body>
3  ··<img src="img/header.jpg" />
4  ··Our time in Rome:
5  ··<a href="img/gusto.jpg">Gusto</a>
6  ··<a href="img/piazza.jpg">Piazza</a>
7  ··s/="(.*?\.jpg)"/="img/\1"/
8  ·</body>
9  </html>
```

The replacement has not yet been applied to the underlying source code. However, even while typing, the results are immediately visible. This also obviates the need for an OK-button: instead of confirming the replacement, the user can simply merge the preview-portals into the document.

The use of portals enables a complete linkage of replacement expression and results. The replacement string `="img/\1"` is visible in all results and can be modified everywhere simultaneously, similar to linked editing [12]. This is also indicated by three ghost cursors in lines 3, 5, and 6. Additionally, every occurrence of the sub-pattern `\1` is recursively replaced by a portal to the respective substring.

Despite their utility, regular expressions are often shunned by users because of their perceived complexity and the unpredictability of their results. As our live preview shows both, the matching section and the resulting code, the overhead of using regular expressions compared to manual operations is reduced, making them convenient even for small adjustments.

The effect that the visible document-state is always up-to-date in respect to the underlying source code and feature-configuration, is achieved by a combination of two different aspects: First, all code portals representing the textual replacement are continuously re-created after each modification. And secondly, modifications performed within a code portal are – by definition – immediately visible in all other corresponding locations.

This second aspect can be used to provide a behavior similar to live-previews for some code transformations, without the overhead of continuous re-calculation. Take, for example, the following code fragment containing a definition of the factorial function and a corresponding function call:

```
1  f = n => if n == 0
2  ·····then 1
3  ·····else f (n - 1) * n;
4  main = f 7;
```

The underlying functional programming language fulfills the property of referential transparency, which allows us to replace identifiers with their respective values. In this instance, we provide a feature to *inline* the selected function call, by replacing it with the definition of `f`, while simultaneously replacing all occurrences of the formal parameter `n` with the provided function call argument `7`, resulting in the following document state:

```
1  f = n => if n == 0
2  ·····then 1
3  ·····else f (n - 1) * n;
4  main = if 7 == 0
2  ·····then 1
3  ·····else f (7 - 1) * 7;
```

All green portals reference the original application argument `7` in the underlying source code. Modifications to one of these instances are immediately visible in all other instances. Furthermore, changes to the function definition are also visible in both locations. By further inlining the recursive call to `f` in the last shown line, the programmer can proceed to unroll the recursion and perform a manual symbolic execution by combining inlining with live-evaluation.

5. Generic Live-Previews

So far, we have described how code portals can be used to display live-previews for expression values and types, function inlinings, and refactoring operations. All these examples share a common structure: Results are integrated into the document as non-destructive previews, which are then kept up-to-date by a continuous re-calculation of the originating feature. In the case of live-evaluation, the result consists of a single insertion of a dynamically inferred value, whereas for inlining and refactoring the results consist of multiple portals referencing existing code locations. In all cases, however, the non-destructive aspect of code portals allowed for a clean removal of all portals, followed by a re-calculation. By treating these two aspects separately, we can derive a generic way to facilitate live-preview capabilities for IDE features.

The only requirement for an editor feature to be suitable for a generic live-preview is that instead of modifying the ac-

tual code, it displays all modifications through the creation of code portals. This does not limit expressiveness, as sequences of textual modifications can always be represented by corresponding sequences of portal creations. Having a feature that fulfills this requirement, we can then automatically perform the following adjustments to keep the results always up-to-date:

- When first applying the feature, we collect all portals $\mathcal{P}$ that are created during its execution.
- Multiple invocations of the feature are kept on a stack.
- After each modification to the source code, we first close all portals $\mathcal{P}$ for each entry on the stack, and then reapply the features in the right order while updating $\mathcal{P}$.
- If a modification occurs within a code portal, the semantics of portals makes sure that the modification is performed "through" the portal onto the referenced code location. Therefore, all changes are already committed to the source code *before* all portals $\mathcal{P}$ are closed.

Without the concept of code portals, a development environment would somehow have to revert and re-apply all changes performed by live-features, without undoing modifications performed afterwards by the user. This process is prone to conflicts, which in the case of code portals are already handled cleanly by the semantic definition of editing operations within portals.

In our implementation, all aforementioned features realize their live-preview through this generic process. In the case of live-evaluation, the underlying non-live variant simply creates a portal containing the evaluated result at the current moment, whereas for regular expression replacements, the non-live variant performs the replacement only once.

We have further experimented with live-previews of various editor features, creating interesting results:

- A feature we implemented that shows the diff between the current file and its last revision in a version control system, displays, when activated "live", an always up-to-date diff while typing, which proved quite useful when preparing changes for a commit.
- A search function displaying its results through a list of portals provides live-results with entries appearing and disappearing while typing.
- Displaying compiler errors through a code portal instead of a separate view produces live error-reporting, which reduces the time overhead of manually triggering the compilation process.

In general, we have found that many IDE features can easily be adapted to use this process, and can benefit greatly from live-preview capabilities.

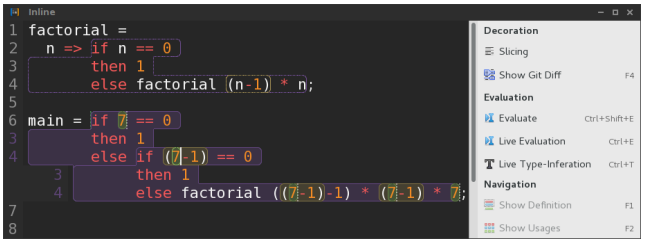


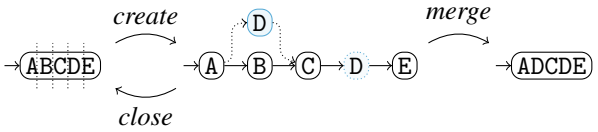
Figure 1. Screenshot of our implementation containing two nested function inlinings. The sidebar provides quick access to commonly used features.

6. Implementation

We have implemented our approach in a prototypical text editor called INLINE [2, 9] to demonstrate feasibility and further explore the possibilities of live evaluation. Figure 1 shows a screenshot of the editor.

Most features mentioned in this paper have been implemented in the presented ways, namely regular expressions, function inlining, live-evaluation, type-inference, and effects. All of these features consist of a backend part, which calculates the corresponding results or transformations, and a frontend part, which handles the (live-)presentation of the results within the editor view. By using portals as a uniform representation, the frontend code is very concise across all implemented features and in some cases consists of only a few lines of code. The whole project consists of about 13,000 lines of code written in the D programming language.

We implemented code portals by transforming the conventional linear representation of text documents into a non-linear graph structure, where each node references an atomic text fragment. Text fragments can be referenced multiple times and can therefore occur in multiple instances within the visible document. A simple code document without any portals consists of a single node referencing the whole file contents. Creating a code portal splits the nodes containing the start and end positions of the selected range, and replaces this range by a new sequence of nodes. All further operations are defined using similar basic graph transformations:



Unfortunately, most features of existing IDEs like Visual Studio or Eclipse are built with an inherent assumption that textual content is linear, and would require major invasive changes to support code portals. We therefore chose to implement our approach as a standalone text editor as part of a feasibility study.

7. Evaluation

We have conducted a qualitative usability study with 9 participants to evaluate whether our approach is usable, quickly learnable, and improves program comprehension. Each participant had 40-60 minutes of hands-on work to solve a set of 6 common programming tasks using INLINE.

All participants, even those not familiar with the concept of code portals, quickly learned their usage and gained productivity within this time frame. They reported unanimously that the concept of code-portals in its current look and feel is intuitive and the provided features significantly helped them to understand the source code. Especially the live-evaluation of expressions was used as a quick way to infer the semantics of functions. One participant, having little experience with functional languages, noted that the combination of live-evaluation helped him understand the functional programming paradigm.

Some participants raised usability concerns for large or deeply nested code-portals, which we will address and re-evaluate in a further update. Overall, the feedback was very positive, with multiple participants expressing their surprise that these features were not more common in modern IDEs. More details and evaluation results can be found in [3].

8. Related Work

The concept of providing an immediate feedback to programmers through the display of live-values is commonly found in academic research and programming tools. In this regard, our work is closely related to Edwards [6, 7] and McDirmid [10], who also integrate non-visual live-evaluation into the main source code view. Edwards uses an omnipresent display containing the values of all relevant expressions, whereas McDirmid allows programmers to toggle individual *probes* and *traces* according to their needs. In contrast to their work, however, we focus on making all values textually accessible to allow programmers to re-use and integrate them into their code.

Our use of code portals introduces partial structure into the source code, that is not required to follow the syntactical structure to allow temporary malformed programs. This is similar to the approach of Homer and Noble [8], who allow users to arbitrarily switch between a textual and syntax-oriented view while actively supporting syntactically invalid code. Our visualization of portals, however, maintains the original mono-space grid of conventional source code editors.

By using portals, code fragments appear multiple times in a document and are edited simultaneously, similar to linked editing. SIMULEDIT [11] allows the selection and simultaneous editing of text fragments via temporary pattern definitions. In our editor, such links are formed by opening persistent portals within the document with support for nesting. Following a different approach, CODELINK [12] identifies code clones and performs modifications to all clones in parallel. Our approach uses similar visual concepts, like ghost cursors and identifying background colors. However, portals are not restricted to clones and handle a much wider set of use cases.

Different approaches like Code Canvas [5], Code Bubbles [1] and Jasper [4] allow a free placement of small, encapsulated editor views on a canvas. While they in principle address a similar problem, namely the need to organize multiple code artifacts on a limited screen space, their approach differs from ours by providing a canvas for multiple editors, whereas we provide a single editor that can be used for multiple canvases.

9. Conclusion and Future Work

In this paper, we have described how live-evaluation results and feature previews can be presented and subsequentially modified within the main source code view using code portals. We have applied this technique to several programming languages from different paradigms, and presented multiple examples and use-cases.

Our evaluation shows, that code portals provide an intuitive and versatile extension to regular source code documents. A plain-text representation of evaluation results, combined with the ability to apply regular editing operations within the results, increases usability by limiting the amount of concepts programmers have to learn.

As future work, we plan to integrate code portals into existing IDEs, and implement evaluation support for more common languages, particularly Haskell due to its strict purity. Furthermore, we plan to use the presented techniques to provide live-previews for code transformations that go beyond refactorings, like displaying the weaved results of aspect oriented programming.

References

[1] A. Bragdon et al. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2503–2512. ACM, 2010.

[2] A. Breckel and M. Tichy. Inline: Now you're coding with portals. Proceedings of the *International Conference on Program Comprehension (ICPC 2016)* [accepted], Available online: `https://goo.gl/cSAeHa`, .

[3] A. Breckel and M. Tichy. Embedding programming context into source code. Proceedings of the *International Conference on Program Comprehension (ICPC 2016)* [accepted], Available online: `https://goo.gl/cSAeHa`, .

[4] M. J. Coblenz, A. J. Ko, and B. A. Myers. JASPER: An eclipse plug-in to facilitate software maintenance tasks. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*. ACM, 2006. ISBN 1-59593-621-1.

[5] R. DeLine and K. Rowan. Code canvas: Zooming towards better development environments. In *Proceedings of the International Conference on Software Engineering (New Ideas and Emerging Results)*, May 2010.

[6] J. Edwards. Example centric programming. *ACM SIGPLAN Notices*, 39(12):84–91, 2004.

[7] J. Edwards. Subtext: uncovering the simplicity of programming. In *ACM SIGPLAN Notices*, volume 40, pages 505–518. ACM, 2005.

[8] M. Homer and J. Noble. Combining tiled and textual views of code. In *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, pages 1–10. IEEE, 2014.

[9] Inline Editor. `http://www.uni-ulm.de/en/in/pm/research/projects/inline.html`.

[10] S. McDirmid. Usable live programming. In A. L. Hosking, P. T. Eugster, and R. Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 53–62. ACM, 2013.

[11] R. C. Miller and B. A. Myers. Interactive Simultaneous Editing of Multiple Text Regions. In *USENIX Annual Technical Conference, General Track*, pages 161–174, 2001.

[12] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 173–180. IEEE, 2004.

[13] M. Vakilian et al. Use, disuse, and misuse of automated refactorings. In *ICSE 2012*, pages 233–243. IEEE, 2012.