Error tolerance for live programming environments

Dissertation for the degree Master of Science

Colin de Roos

February 3, 2022

Supervisor:
dr. P.W.M. Koopman

Second reviewer:
Prof. dr. S.B. Scholz

Radboud University

Abstract

Live programming environments promise to close the gap between writing programs and observing their runtime behaviour. Live programming support needs to keep functioning for incomplete programs, i.e. it needs to be error-tolerant, to keep this promise.

In this thesis, we present a live programming environment prototype that (1) provides a foundation for error-tolerant IDE services in general and (2) offers error-tolerant support for live programming by showing up-to-date runtime values of relevant expressions.

To accomplish this, we developed a structure editor that maintains the flexible and language-independent interface of standard text editors, by incorporating a novel notion of incompleteness, called construction sites.

We formalize the meaning of programs with syntax, binding and type errors by extending the notion of (weak head) normal form and giving a denotational call-by-need semantics. Based on this foundation, an interpreter can provide meaningful runtime information in practically all editor states. Furthermore, the programming environment remains responsive and even provides partial runtime information for lengthy or non-terminating computations.

This work suggests that there is a powerful synergy between error tolerance and live programming support that could greatly benefit developer productivity.

Contents

1 Introduction **4**

2 Shortcomings of existing IDE technology **11**
2.1 Shortcomings of error-correcting parsers . . . 12
2.2 Shortcomings of strict structure editors . . . 13

3 Construction sites **15**
3.1 Concept and terminology . . . 15
3.2 ASTs with construction sites . . . 16
3.3 ASTs with multiple AST node types . . . 16

4 Motivating examples **18**

5 Error-tolerant structure editors **25**
5.1 Architecture of a standard IDE . . . 25
5.2 Frugel's architecture . . . 26
5.3 Decomposition . . . 30
5.4 Partial linearization . . . 36
5.5 Parsing . . . 39
5.6 Limitations . . . 41
5.7 Other implementation decisions . . . 42
5.7.1 Cursor and whitespace . . . 43

6 Error-tolerant IDE services and live programming support **45**
6.1 Formatting . . . 45
6.1.1 Formatter implementation . . . 46
6.2 Live programming support . . . 47
6.2.1 General approach to evaluation . . . 47
6.2.2 Error-tolerant interpretation . . . 49
6.2.3 Presenting normalized expressions . . . 51
6.2.4 Collecting runtime information . . . 52
6.2.5 Interlude: dynamic type checking . . . 54
6.2.6 Explicit sharing with references . . . 54
6.2.7 Presenting runtime information . . . 57
6.2.8 Non-termination tolerance . . . 58

6.2.9 Maintaining responsiveness 61

7 Related work **62**
7.1 Grammar cells 62
7.2 Proxima 63
7.3 Lamdu 63
7.4 Hazel 64
7.5 Grounds for increased developer productivity in live programming environments . . 64

8 Future work **66**
8.1 Supporting I/O 66
8.2 Performance on larger programs 67
8.3 Integration with mainstream programming environments and other text-oriented tools 67
8.4 Improved interface for runtime values 68
8.5 Improved explicit empty construction sites 68

9 Conclusion **69**

A Decomposition for ASTs with multiple node types **73**

Chapter 1

Introduction

Programs often do not work as intended by the programmer the first time they are written. Even if they do, they often need to be modified later because of changing requirements.

Furthermore, programs contain syntax and type errors that prevent their execution during a large part of their development. Yoon and Myers find 44% of programs were syntactically malformed based on 1460 hours of editor logs [26].

In systems where syntax and type errors prevent program execution, we typically find the familiar edit-compile-run cycle illustrated below.

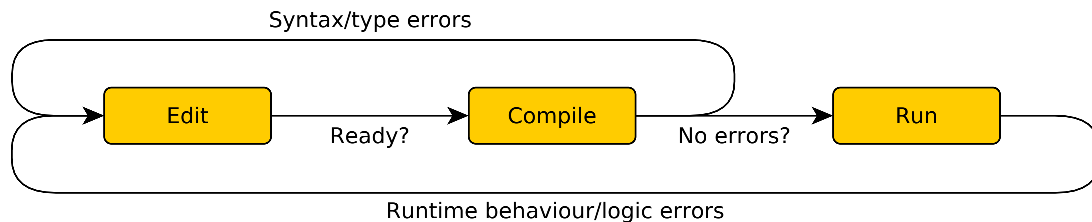

Figure 1.1: The traditional edit-compile-run cycle

Because syntax and type errors prevent programs from being executed, a perceptive gap exists between the writing and execution of programs: when we write (a part of) a program. we cannot immediately see how it behaves at runtime. This impairs the learning experience of novices and the productivity of professionals [2] because there is a delay in receiving runtime behaviour feedback. Receiving this feedback as soon as possible can prevent the programmer from taking a flawed approach to their task and thus spare them the time they would have needed to implement the approach in full and resolve all the errors they encounter in doing so.

Furthermore, the lack of continuously available runtime behaviour feedback forces the programmer to "play computer in their head", which is time-consuming and error-prone.

Programmers are often assisted in their tasks by the application they use to write code, called the programming environment/integrated development environment (IDE). Live programming envi-

ronments promise to close this gap by offering uninterrupted information on the program's runtime behaviour, thus leading to the workflow illustrated in figure 1.2.

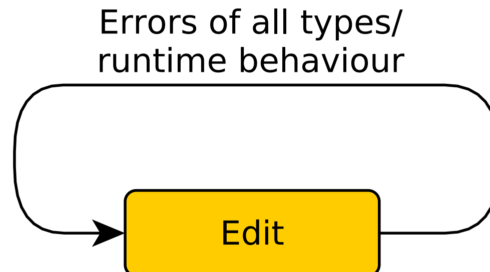

Figure 1.2: The live programming cycle

In order to keep this promise, live programming environments need to be able to evaluate code in any editor state. Furthermore, evaluation must not abort with an exception or result in an uninformative "undefined" value when errors are encountered. Such results would defeat the purpose of live programming features because the programmer receives very little new information. Instead, the evaluation result should support the programmer in finding the source of the problem, for example by showing context, in the form of the values of relevant variables and expressions at the time the error occurred (e.g. an array index being out of bounds).

In other words, it should not only be possible to run incomplete programs, but the result should also include any information that could help the programmer find the source of the issue. Gathering this information entails evaluating code that is computationally independent of the expression in which the error occurred. For example, the left operand e_l of an addition term $e_l + e_r$ can be evaluated independently of the right operand and the resulting value should be available to the programmer if the right operand e_r does not evaluate to a number and thus prevents the values from being added together.

This property of IDE service is called *error tolerance*: *to continue working in the presence of errors to provide the most useful results possible.* Ideally, evaluation should tolerate any kind of error, e.g.

1. Syntax errors, such as a missing parenthesis
2. Type errors, such as find a function where a number is expected.
3. Binding errors, such as an undefined variable
4. Logic errors, such as
   (a) an array index being out of bounds
   (b) diverging evaluation of an expression (e.g. the Ω-term from lambda calculus)

Note that we do not mean to argue that live programming environments should show a value where there is none (i.e. in the case of diverging evaluation). Instead, the environment should

show a maximally informative result, for example, the values that are independent of the diverging branch of evaluation (ideally relevant in the context of the diverging computation) and/or a partial result that shows how far the diverging evaluation has progressed. Of course, it is impossible to know whether the evaluation of an expression will eventually halt or not [22], so the programming environment should produce such results for every long-running evaluation. In fact, such results would make the programming environment more useful in cases of long-running computations as well, as long as full results are still provided when they become available.

To provide maximally useful results, it is important to isolate errors. For example, if a function definition contains an opening parenthesis that lacks a closing counterpart, this should not affect (the evaluation of) later definitions in the file. In other words, the syntax error should be isolated to the definition or expression it is located in.

This isolation allows a live programming environments to provide partial results for syntactically malformed (i.e. containing syntax errors) or ill-typed (i.e. containing type or binding errors) programs as well. Any reductions of expressions that cannot be performed due to an error can be included in the evaluation results to show the programmer exactly what went wrong.

Error tolerance and isolation may also benefit other services that IDEs commonly provide. For instance, an automatic formatter may indent code following the unmatched opening parenthesis too far if the syntax error is not isolated. If it is, the following code could be indented properly. Additionally, if there is an error in a type definition in a certain file or program module, error-tolerant type systems could support the type checking of dependent files/modules (modules that import the one with the errors).

Tolerance of syntax errors is commonly provided by error-correcting parsers, but these are forced to guess at the programmer's intent (where is the missing parenthesis supposed to go?) when IDE services that rely on them demand error-free results. Even when IDE services technically could function with a partial parse results, they are often automatically disabled in fear of generating spurious errors. For example, a type checker might report that a variable lacks a definition if the parser has not recovered in time from an error preceding a syntactically correct definition for the variable. We explain the issues with error-correcting parsers in more detail in section 2.1.

In general, we find a lack of error tolerance and live programming support in programming environments, despite the benefits discussed. With regards to live programming support, we see problems with current approaches that limit its use to certain application domains (e.g. data science or game development). In other words, these approaches are not *domain-agnostic*. In other cases, we find problems that harm the usability of the programming environment. A non-exhaustive list of problems with existing live programming environments follows below:

1. Assumption of infinite loops that keep the program running. In this case, the "liveness" originates from updates to a running program, such as a game or a program that plays a melody on repeat (e.g. Fluxus [9] and Impromptu [20]). This approach excludes programs with short typical run times (such as many of those invoked at a command line) because there is no time to see the effect of changes.

   Additionally, updating a continuously running program may cause it to enter states that it could not have entered otherwise. This may be beneficial in some cases, but it can also be dangerous if unintended side effects occur.

2. Assumption of data with a natural visual representation. Some live programming environments intentionally choose a limited application domain to make demonstrating live programming features easier [13]. Typically, a natural way of visualizing results exists in these

domains. In data science, for example, data can often be visualized using diagrams, like in Observable [1]. In the domain of computer science education, drawings and animations such as those found in ALX [13] can be helpful because they are easier to inspect than complex data structures.

However, in live programming environments that rely on this assumption, a decent presentation of complex data-structures is often lacking. Of course, a good domain-specific representations are often more important for the domains for which these tools are intended, but in most domains a decent generic data presentation is required as a foundation. Mechanisms built on this assumption, such as the visualization of changes, are hard to extend to general data structures and thus prevent adoption of the live programming environment in other domains.

3. The only information about runtime behaviour that is presented, is the program output. Fluxus [9], Impromptu [20] and Observable [1] have this property. It is problematic because for non-trivial programs, the output does not provide detailed insight into the runtime behaviour. While the impact of small changes may be clear, the only way to derive information about the runtime values around the location of a change is reasoning backwards from the presented output. Therefore, any programming environment with this property does not free the programmer from "playing computer in their head".

4. The requirement of syntax error-tolerance is circumvented by replacing a standard text editor with a structure editor that prevents syntax errors completely, as in Hazel [15], Lamdu [12] and ALX [13] for example. Structure editors maintain a structured representation of the document during editing. The replacing of text editors with structure editors is problematic because structure editors that preserve syntactical well-formedness are less flexible than text editors [23].

   We will discuss this in more detail in section 2.2, but in short: the simplest method for transforming an expression to a new one sometimes requires syntactically malformed (i.e. containing syntax errors) intermediate states For example, when moving the opening parenthesis in $(3*2+1)$ to enclose the addition term (i.e. $3*(2+1)$), we would normally delete and reinsert the opening parenthesis, but this is not allowed because deleting it would introduce a syntax error. Some structure editors have evolved to specifically make moving parentheses easier than it is with text editors, but the general problem persists. We discuss more advanced examples in section 2.2.

Problem statement Before live programming environments can become a part of mainstream software development practice, *domain-agnostic* and *error-tolerant* live programming environments must be developed.

The aim of this thesis is to contribute towards their development by examining the use of a novel notion of incompleteness, called *construction sites*, in a live programming environment.

In essence, constructions sites are (possibly empty) sequences of plain characters and other structured pieces of programs called *complete nodes*. Construction sites are designed to isolate syntax errors while allowing a structure editor to retain the flexibility of standard text editors. The notion of construction sites is further developed in chapter 3.

Specifically, we attempt to answer the question:

Can a programming environment provide live programming support, and generally offer error-tolerant IDE services without interruption, regardless of present errors or application domain, based on (1) a structured representation of programs that incorporates construction sites and (2) formal semantics extended to give meaning to syntactically malformed and ill-typed programs?

Methods To answer the research question, a structure editor and error-tolerant interpreter were developed and combined in a live programming environment named "Frugel". The structure editor maintains a structured representation of the program that is used by the interpreter to provide continuous information on the runtime behaviour of the program and thus supporting live programming.

The name is derived from "frugal", but misspelled for disambiguation. It is inspired by how careful the environment is with discarding information that might be useful to the programmer.

Chapter 4 contains many examples of the usage of the programming environment and the unique ways in which it assists the programmer.

Overview of our approach to the structure editor A common approach to structure editing is inserting holes into the program based on the programming language's grammar. However, structure editors based on this approach sometimes insert holes that the programmer wants to fill with an existing part of the program. Moving that part of the program into the hole requires additional effort from the programmer.

Instead of inserting holes that can be filled in later, our structure editor *decomposes* the language construct at which the cursor is located into a construction site. This construction site contains the node's components (characters and complete nodes) according to the grammar of the language construct. Any character can be inserted into a construction site. In contrast to many structure editors, this allows syntax errors to be introduced and thus enables our structure editor to retain the flexible interface of text editors (resolving issue 4 of the list of problems we find in existing live programming environments). We detail the design and implementation of the structure editor in chapter 5.

Moreover, keystrokes are not coupled to the insertion of specific language constructs. In combination with allowing arbitrary characters in construction sites, this enables a user interface that works similarly for all (textual) programming languages.

Even if the insertion of a character does not introduce a syntax error, the AST node under the cursor is still decomposed. Syntactical correctness is checked afterwards using a traditional parser. If the parser can parse the program without finding any errors, the construction site is resolved. If it does find an error, this must be due to the last edit action and the construction site remains. Because the AST is only affected at a single node, there is always an AST to fall back on if a syntax error is found.

This preservation of child nodes distinguishes this approach from others where the editor simply allows unstructured text at some locations.

If there are multiple construction sites, the parser may skip their contents to verify syntactical correctness of the construction site that was last introduced. All combinations of skipping construction site contents should be considered to resolve all construction sites that do no longer contain a syntax error. We describe the various options for realizing this in section 5.4.

As a bonus, the integration of construction sites in the structure editor supports a clear distinction between the characters that cause parse errors and the rest of the program text. This way, our combination of careful decomposition and reparsing support a more transparent form of error resilience than what is provided by error-correcting parsers.

Construction sites are a language-agnostic concept, i.e. they can be incorporated in the structured representation of any language. Furthermore, the structure editor technology developed is language-agnostic, in the sense that it should be possible to instantiate the editor for any structured language that admits a textual representation. The main requirements to instantiate our structure editor for a language are a (slightly modified) parser and a way to traverse the structured representation used to represent programs. These requirements are encapsulated in an interface and described in detail in chapter 5. In this thesis, we instantiate the structure editor with a small pure programming language as an example. This language is λ-calculus extended with numbers, booleans and recursive binders.

IDE services can use the preserved structure of unaffected parts of the structured program representation and the complete nodes inside construction sites to continue operating properly. This is demonstrated with a formatter, which has the unique property that syntax errors never affect the formatting of parts of the program below or above (in terms of lines of code) of the language construct in which the syntax error is located.

The live programming support in our programming environment serves to demonstrate the adequacy of this approach to tolerating syntax errors as well. We describe the implementation of the formatter and the formal foundations of the live programming support in chapter 6.

Overview of our approach to live programming support Instead of supporting updates to a continuously running program, we think it is often more useful to show the effect of code changes on the behaviour of the program in circumstances specified by the programmer. This removes the need for repeatedly triggering the relevant circumstances for the runtime behaviour of interest (the runtime behaviour corresponding to the part of the program the programmer is working on) and thus resolves issue 1 from the list presented before. This approach is relatively simple to implement for the aforementioned extended λ-calculus, but tackling major challenges such as handling I/O and testing with real-world programs did not fit in the time constraints of this thesis. Because our example language still lacks I/O capabilities (and the environment lacks mechanisms to handle this), specifying circumstances for runtime behaviour of interest is limited to specifying an instance of a function application. For example, if f is a function and the expression $f\ 1 + f\ 2$ is evaluated, the programmer can pick the instance where f is applied to 1 or the instance where f is applied to 2 when inspecting f's runtime behaviour.

We show the effect of code changes in a user interface similar to that of a time-travel-debugger that is updated automatically after every edit. It shows the values of variables in scope at the cursor (the evaluation environment), the value of the expression under the cursor and the result of the full program. The specification of runtime behaviour of interest as described above is similar to the way a time-travel-debugger supports browsing through different occurrences of a break-point.

This user interface resolves issues 2 and 3 from the list presented earlier, but it is also still rudimentary. For example, the size that certain evaluation results take up on the screen is only configurable using a number picker that the determines the depth to which the AST underlying the result is rendered. Whether some nodes are rendered is determined one AST level at the time, while it would be very useful to be able to pick nodes to render individually as well.

A more advanced interface could offer further benefits by supporting exploration of the compu-

tation in general, instead of just the results, and support the display of information from multiple breakpoints simultaneously. We discuss this in more detail in section 8.4.

The data presented in our user interface is obtained using an interpreter based on a semantics of the example language extended to syntactically malformed and ill-typed programs (an error-tolerant semantics). When the program or cursor location changes, the interpreter runs again on the new program and collects runtime information based on the new cursor location. While this is acceptable for the current prototype, it is not clear that this approach performs well enough in real-world applications. See section 8.2 for further discussion.

To realize error-tolerance, evaluation does not stop at an error (as with exceptions) nor do errors "absorb" other values (as with "undefined" in many languages). Instead, errors only prevent the evaluation of expressions that depend on the result of the evaluation during which the error occurred. We formalize this principle in a denotational call-by-need semantics presented in section 6.2. Thus, we give a useful meaning to (incomplete) programs with errors.

The interpreter also has a "limited" mode where recursion and the number of performed reductions is limited by a configurable number. We call this mode "fuel-limited evaluation". This crude mechanism guarantees that the programming environment can also provide some runtime information when normal evaluation diverges or is only taking a long time to produce results. Furthermore, evaluation of the program does not need to be restarted if any additional evaluation needs to be performed to present additional interactively requested runtime information.

Remarkably, we find that the measures providing this robustness can help to find the sources of non-termination in some cases. In essence, the reductions that fail due to a shortage of fuel point out the expressions which causes evaluation to diverge (when there is enough fuel to evaluate the rest of the program). We show an example of this at the end of chapter 4.

Summary of contributions In summary, we make the following contributions:

1. We present the concept of construction sites, a novel notion of incompleteness that supports the precise isolation of syntax errors.

2. We detail and develop an editor that uses the inherent structure in programming languages and the flexibility of construction sites to provide a sound technical basis for error-tolerant IDE services, while retaining the familiar flexible interface of text editors.

3. To support this claim, we develop and describe two completely error-tolerant IDE services: a formatter and live programming support. Moreover, these form a domain-agnostic live programming environment that aids the programmer in unique and intriguing ways.

4. We formalize the error-tolerant dynamic semantics of a pure functional language with a call-by-need reduction strategy. To the best of our knowledge, there are no other live programming environments using this reduction strategy at the time of writing. We identify several issues with the combination of live programming support and these semantics and describe our solutions.

The reader may experiment with the programming environment online at https://cdfa.github.io/frugel/ or download a (better performing) native executable from https://github.com/cdfa/frugel/releases.

Chapter 2

Shortcomings of existing IDE technology

In most IDEs of today, IDE services are dependent on the error-free completion of certain program analysis steps. If an analysis step completes without finding any errors, this allows dependent IDE services to make certain assumptions about the analysed program. However, if errors are found, the IDE service is disabled or may not function properly.

For example, when a parser produces an abstract syntax tree without reporting any errors, we can assume this AST is a complete representation of the program. Automatic type checking depends on this because it may report spurious errors when the representation is incomplete. If the parser finds an unmatched opening parenthesis, a formatter may indent code following the opening parenthesis to far. Note that it is not always obvious when an IDE service is disabled because an IDE may continue displaying outdated results or new changes may not affect the functioning of IDE services in specific cases.

Often, analyses are performed per file or software module. With IDE services such as automatic type checking, there are also dependencies between the analysis of these modules. A type checker can only check a module for type errors if the type definitions imported from other modules are well typed. Inconsistencies between function implementation and its assigned type can usually be tolerated, though, because type checking of other functions does not depend on this.

The general problem is that the design of these IDE services requires assumptions (such as a complete structural representation or correct type definitions) to hold true for entire software modules, instead of performing their tasks on a software module partially, based on what parts of the program the assumptions hold for. In short: IDE services are (partially) error-intolerant. This prevents programmers from using the IDE services, while programmers could still benefit from their partial (but proper) functioning.

This was already recognized as problematic by van de Vanter in 1994 [23]. He phrases this error intolerance as a failure to "degrade gracefully" in the presence of errors. He also summarizes the situation in which this occurs with the I3 conditions: incorrectness, incompleteness and inconsistency. Their paper explains why intolerance for these conditions is problematic in more detail than we have here.

There are two common approaches of ensuring IDE services that depend on a structured representation of the program can always keep functioning: error-correcting parsers and projectional

editors/structure editors.

Both typically have drawbacks that will be discussed in depth in section 2.1 and 2.2 respectively. For our programming environment, we choose the structure editor option. We show how we can avoid the inflexibility that is usually paired with these in chapter 5.

2.1 Shortcomings of error-correcting parsers

Error-correcting parsers are a popular option for realizing (partial) tolerance of syntax errors. However, strings of characters fundamentally contain less information than the structured representations maintained by structure editors. This limits the completeness and accuracy of the results of error-correcting parsers in comparison to those of structure editors that prevent syntax errors.

Results from error-correcting parsers may be less complete than those of structure editors in cases where the parser discards parts of its input when it encounters a parse error to return to a consistent state. Results from error correcting parsers may be less accurate, because when a parser corrects its input, it often needs to guess at the programmer's intent.

As a concrete example of this, consider the case of a missing parenthesis: `f (g x z`. In this case, there are three strong candidates for correction:

1. Removing the opening parenthesis, resulting in `f g x z`.
2. Adding a closing parenthesis after `x`, resulting in `f (g x) z`
3. Adding a closing parenthesis after `z`, resulting in `f (g x z)`

There is simply no information in the text that indicates which of these corrections is the intended one and the number of options increases quickly with the size of the expression and the number of errors.

In structure editors that preserve syntactical well-formedness, examples such as this one are not possible, so their accuracy is 100%. They also do not need to discard a part of their input, so their completeness is 100% as well.

However, as mentioned in the introduction, structure editors may also allow parse errors if the structure they maintain is flexible enough. Therefore, it is not true in the general case that structure editors can produce more accurate and complete results. However, the lack of available information in comparison to structure editors does make it difficult to realize results of similar quality.

Various heuristics have been developed to improve the quality of results. As an indication of their general performance, consider the following recently developed error-correcting parsers.

1. The error-recovering PEG-based (Parsing Expression Grammar) parser by Medeiros et al.[18] (2020). It only performs the intended recovery in 37% to 64% of cases (depending on the grammar; or 63% to 81% with manual intervention).
2. The SGLR parser by Kats et al. (2009). It only performs the intended recovery in 60% of cases and produces spurious errors in 10% of cases.

Of course, better heuristics will be found in the future, but people have been working on this since at least 1963[11]. This motivates us to look into different approaches. Similar studies to the one listed above will have to show how the accuracy and completeness found in these studies compares to those of the results produced by the structure editor we developed.

However, there are cases where our structure editor performs at least as good as any error-correcting parser could. For example, when a parser encounters a syntax error that is followed by a (syntactically correct) variable definition, it might not recover in time to include the variable definition in the result. A type checker would then report that the variable is not defined, while there is nothing wrong with the variable definition itself.

With the information contained in the structured representation our editor maintains, it can determine that the syntax error is located before the variable definition and isolate it. Thus, the variable definition is included in the result and the type checker does not produce a spurious error message.

In general, syntax errors cannot cause spurious error messages to be reported, because they are always isolated to the language construct they are located in. They are even isolated from other language constructs that the one containing the error was composed of because these component constructs become complete nodes. This is an important property for error-tolerant programming environments and the main reason we chose to develop a structure editor.

An error-recovering parser could also be combined with our structure editor, because we do not place requirements on the parser technology used. However, the possible introduction of spurious errors should be weighed carefully against the higher completeness that this combination could produce.

2.2 Shortcomings of strict structure editors

Structure editors attempt to improve upon the editing experience provided by standard text editors by using the structured nature of a document. Structure editors are language-specific and may offer an interface that is fine-tuned to the language in use. For example, they may allow for the insertion of language constructs with fewer keystrokes than there are characters in the language construct's textual representation or support an intuitive way of resolving ambiguities in documents composed of multiple languages.

However, their general interface is often less familiar or intuitive than that of a text editor and therefore they often have a steep learning curve. Their choice for a structured representation of the document also often inhibits integration with the large ecosystem of text-based tools.

They distinguish themselves from traditional text editors by performing edits directly on a structured representation of the document instead of on a textual one. If this structured representation allows no states with syntax errors, we call the structure editor *strict*. While it might sound great to have it be "impossible to make syntax errors", this often causes usability issues in practice because the document has become inflexible or its manipulation relatively complicated[23].

An example of these issues is the *tunnelling problem*. We attribute its identification to Hazel [15] contributors, but cannot find a proper reference. In many cases, the shortest (or at least simplest) sequence of edits that would lead to the desired document state goes through one or more syntactically incorrect document states.

This problem is often demonstrated by trying to move parentheses, but some structure editors have evolved to handle this use case well. So instead, we will give a more advanced example: consider the expression `<x> + <y>` where `<x>` and `<y>` are both large subexpressions. The objective is to convert it to a record with the left and right summand as fields, i.e. `{left: <x>, right: <y>}`. What the easiest approach is, depends on the editor being used, but it often comes down to first constructing a new record with the desired field names, cut-and-pasting `<x>` and `<y>` into the right positions and finally deleting the old expression (if cut and paste is supported at all, which is often

not the case because the operating system's clipboard cannot be used because it does not preserve structure). In any case, it seems difficult to make it simpler than the process in a text editor: add curly braces at the start and end, replace the plus sign with a comma and add the field names, all in any order that seems convenient.

Thus, restricting the user to edit actions that preserve syntactic correctness makes the document inflexible and therefore complicates editing instead of easing it. Structure editors which are strict in this correctness-preserving requirement might rather be called structure *constructors*. The report by Voelter et al. provides a more extensive overview of the issues that plague them [24]. Voelter et al. report that these issues can be effectively mitigated by emulating the experience of standard text editors, which is the approach Frugel takes as well. The differences between their approach and the one taken by Frugel are explained in more detail in section 7.1

Another drawback of the structure-editor-approach in general is that due to their language-specific nature, it requires more work to make the editor usable for a new language. To mitigate this drawback as much as possible, we keep the technology developed for our structure language-agnostic and formally parameterize the editor. We discuss the requirements for instantiating our structure editor for a new language in chapter 5. We discuss the integration of our structure editor with text-based tools in chapter 8.3.

Chapter 3

Construction sites

Due to the issues mentioned above, a non-strict structure editor was developed for Frugel. This structure editor incorporates construction sites to retain the flexible and intuitive interface of text editors. Essentially, where the structured representation cannot be preserved, construction sites are inserted.

What these construction sites are and how they can be integrated into the structured representation of any language is the topic of this chapter.

3.1 Concept and terminology

The concept of construction sites is inspired by Hazel's non-empty holes [15]. Similarly to Hazel's holes, construction sites act as a "membrane" around an error to isolate it from the rest of the program. In our structure editor, empty construction sites inserted into the program with their own syntax, but there is no syntax for non-empty construction sites. Instead, non-empty construction sites are created by the editor when the programmer performs a transformation that leads to a syntax error.

In contrast to the holes in many structure editors, the main purpose of empty construction is not to be inserted as a placeholder where a language construct requires some component that has yet to be written. Instead, their main purpose is to appear as placeholders when all the characters representing such a component are removed, so the large language construct can be left intact. Because this is their main purpose, empty construction sites disappear when they can be removed without introducing a syntax error.

The contents of a construction site are called node components. These are either plain characters or other language constructs, called complete nodes. These complete nodes may still be analysed and contribute to the feedback presented by the programming environment.

When a character is inserted, the language construct under the cursor is decomposed into a construction sites with the node components in order of the grammar production rule corresponding with the language construct. A construction site is resolved when a parser can parse its contents without encountering a syntax error. The parse result then replaces the construction site in the structured representation of the program.

In the rest of this thesis, we will discuss construction sites in the context of their use in abstract syntax trees, the kind of structured representation often used in structure editors. However, we

will use a common grammar notation to represent ASTs in a way that is independent of an implementation language. A proper grammar for the parsing of construction sites is presented in section 5.5.

Construction sites are a language-agnostic concept. Therefore, it should be possible to integrate them into any structured representation. To facilitate concrete examples, we will start with the λ-calculus extended with numbers and an addition operation. In section 3.3 we will extend this further to include auxiliary definitions.

3.2 ASTs with construction sites

We will first review an AST for the λ-calculus extended with numbers and addition. In the aforementioned generic notation:

$$\mathsf{Expr} \quad e \quad ::= \quad x \mid \underline{n} \mid \lambda x \doteq e \mid e\ e \mid e + e$$

We have the usual forms for variables (x), application ($e\ e$) and addition ($e + e$). The form $\underline{n}$ ranges over numbers and $\lambda x \doteq e$ represents abstraction.

We will now incorporate construction sites into this AST. With c ranging over characters, we get:

$$\begin{array}{lll l}
\mathsf{Expr} & e & ::= & x \mid \underline{n} \mid \lambda x \doteq e \mid e\ e \mid e + e \mid \colorbox{yellow}{nc} \\
\mathsf{Cmps} & nc & ::= & i^* \\
\mathsf{Comp} & i & ::= & c \mid \boxed{\mathsf{Expr}}
\end{array}$$

Construction sites are denoted with a yellow background, as in $\colorbox{yellow}{$x$}$. Node components are represented with the non-terminal nc. When an AST node is decomposed into node components, all literal parts (such as "λ") become normal characters and all child nodes become complete nodes. The notation for empty construction sites and complete nodes is $\colorbox{yellow}{\ }$ and $\boxed{\mathsf{Expr}}$ respectively.

We use $\boxed{\mathsf{Expr}}$ instead of $\boxed{e}$ because complete nodes occur in the parser input stream as AST nodes and their textual representation does not need to be parsed.

As an example, consider $1 + \colorbox{yellow}{$\boxed{\colorbox{yellow}{\ } + 2 + 3}\lambda + x$}$. This expression contains two construction sites: one enclosing the right summand of the first $+$ and one empty construction site. The first item in the first construction site is a complete node representing the sum of the second construction site, 2 and 3. Then follow three characters, λ, $+$ and x.

The reason for the first construction site is the occurrence of $\colorbox{yellow}{$\lambda$}$. If it were removed, $\colorbox{yellow}{$x$}$ would become the fifth summand in the expression. Note that because $+$ is left-associative, the $\colorbox{yellow}{$+$}$ in the construction site would belong to the root node of the parsed expression and $1 + \colorbox{yellow}{\ } + 2 + 3$ would be the left child of the root node. Thus, the root of the expression shifted from the first $+$ to the last.

3.3 ASTs with multiple AST node types

The language presented above is unrealistically simple; all real-world languages have multiple types of AST nodes. Since extension of ASTs with construction sites to languages with multiple types of AST nodes is not completely trivial, we will extend our example language slightly.

Our extension consists of `where`-clauses and auxiliary definitions. The new AST looks as follows:

$$
\begin{array}{llll}
\textsf{Prog} & p & ::= & e[w] \mid \boxed{nc} \\
\textsf{Expr} & e & ::= & x \mid \underline{n} \mid \lambda x \doteq e \mid e\ e \mid e + e \mid \boxed{nc} \\
\textsf{Where} & w & ::= & where\ d^{+} \mid \boxed{nc} \\
\textsf{Def} & d & ::= & x = e \ \mid \boxed{nc} \\
\textsf{Cmps} & nc & ::= & i^{*} \\
\textsf{Comp} & i & ::= & c \mid \boxed{\textsf{Expr}} \mid \boxed{\textsf{Where}} \mid \boxed{\textsf{Def}}
\end{array}
$$

We use a^+ and $[a]$ to denote one or more of an item and an optional item, respectively. Aside from the obvious additions of Prog, Def and Where, the notation for Comp has also been extended.

Adding a new type of node to an AST with construction sites raises two questions: should the node have a construction site form and should it be possible for the node to be a node component. To answer the first question, we consider what information can be derived from a node's child nodes in comparison to a construction site with only their components. For the forms of Expr this is the most clear: more information can be derived from an application term than from two separate expressions and similarly for an abstraction term compared with a sequence of characters and an expression. Because of this, the nodes that have expressions as child nodes (Programs and auxiliary Definitions) have a construction site form.

For auxiliary definitions, the extra information lies in their purpose of binding: a complete definition node can add a new variable to the scope, but an identifier followed by an expression in an unstructured list does not have the same semantics. Therefore, where clauses should have a construction site form as well.

Where clauses should also not be decomposed without reason, because they delimit a scope and can therefore prevent name clashes where a series of definition would not. This is an extra reason for program nodes to have a construction site form.

For the second question, we should consider whether the node we are adding can result from a decomposition of other nodes. Only for the top-level Prog node this is not the case and therefore it does not need a form for the Cmps node.

Chapter 4

Motivating examples

In this chapter, we discuss some examples that demonstrate the features of our programming environment. The screenshot in figure 4.1 provides an overview of the interface. We describe its elements in the caption.

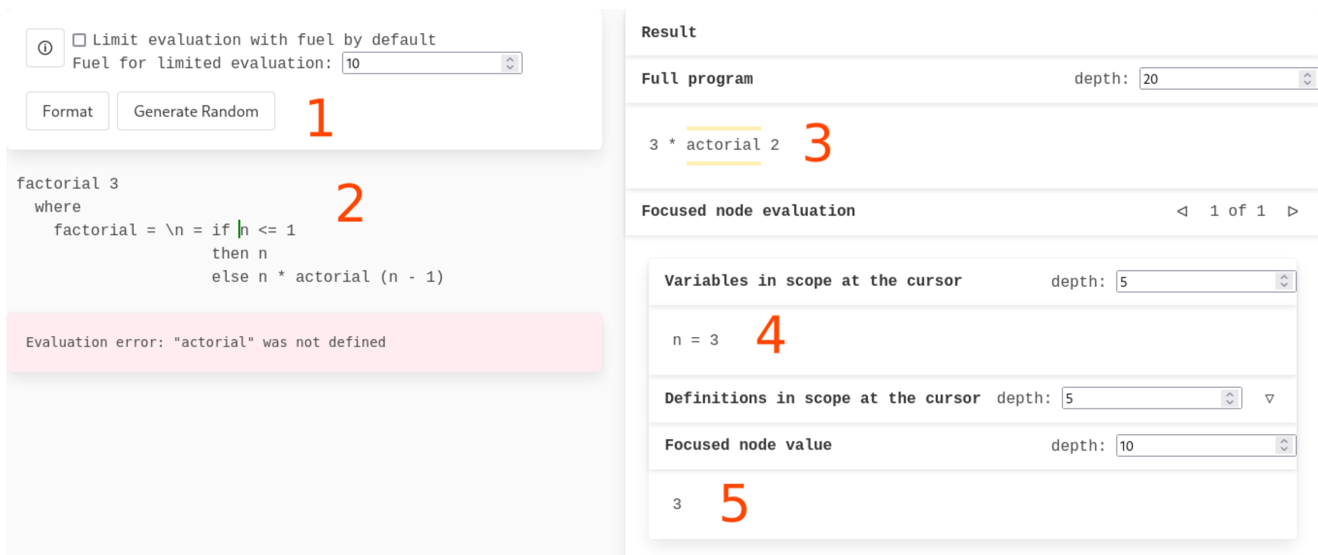

Figure 4.1: The complete interface of our programming environment. Various panels are annotated: (1) a panel with a button to format (a.k.a. pretty-print) the program and other controls; (2) the program being edited with a list of errors below; (3) the normal form of the main expression (`factorial 3`); (4) the normal form any variables in scope at the cursor (currently on `n` in the condition of the if-term); and (5) the normal form of the expression under the cursor. Between 4 and 5, there is a panel similar to 4 for definitions in scope (only `factorial` in this case), but this panel is collapsed. The program shown is a definition of the factorial function where the recursive application is misspelled. Therefore, it cannot be reduced.

Tunnelling For our first example, consider again the instance of the tunnelling problem from the introduction: moving the multiplication term in $(3 * 2 + 1)$ outside the parentheses, changing it to $3*(2+1)$. To recapitulate, this is a problem for strict structure editors because the simple approach of deleting and re-inserting the parenthesis is not possible because it would lead to a syntax error.

Because our structure editor is not strict, this approach is possible and we observe $3 * 2 + 1)$ as the intermediate state. Because the rest of the programming environment is error-tolerant, its features keep functioning for these intermediate states. For example, the formatter would still insert whitespace in the complete node if it was missing and the interpreter will evaluate this expression to $7)$. As expected, inserting a parenthesis in front of the "2" results in $3 * (2 + 1)$.

Syntax error isolation We claimed before that we can isolate the syntax errors we allow in the structure editor. We will now demonstrate this with an example: consider writing a function g that takes an argument x, adds 4 and then multiplies this with a larger term. Writing the function left-to-right, we encounter the state .. $g = \lambda x \doteq x + 4 * f\ \lambda d \doteq$. Here, we elide the preceding part of the program for brevity.

We are in the process of writing a lambda term as an argument to f, but we forgot to add parentheses around the addition term.

Assuming the function is encountered during the evaluation of the main expression, the live programming features of our editor also help to catch this error. For example, if the main expression is $g\ 4$, the body of the function evaluates to $3 + 4 * f$ instead of $7 * f$. This would be visible in the main expression normal form (panel 3 in figure 4.1).

We can insert the missing parentheses like we would in a standard text editor. Because this happens one by one, this will introduce a new construction site containing the expression the cursor was on. Because the syntax error from the unfinished lambda term is isolated in a different construction site, the new construction site can be resolved when the second parenthesis is inserted. This confirms that all parentheses are now balanced and the unfinished lambda term is the only remaining issue.

The benefits of this kind of detailed and reliable syntactic feedback are small in a trivial example such as this. We think it can be helpful during larger refactorings nevertheless, but have not found the time to confirm this with a user study.

The larger benefit of this approach as a whole is that IDE services dependent can use the recovered AST to provide more advanced feedback despite the program's incomplete state. In the case above, the interpreter can show that the body of the function now indeed evaluates to $7 * \langle y \rangle$. In the case of more complex refactorings, this enables the programmer to catch mistakes before the refactoring reaches a syntactically correct state. Furthermore, this feedback may help the programmer work out how to complete the rest of the program (e.g. what the body of the unfinished lambda term should be) and prevent further errors.

Error-tolerant evaluation We would not get the benefits mentioned above if evaluation was not tolerant of type and binding errors as well. Specifically, evaluation does not stop in the event of an error (as with exceptions) nor do errors absorb other values (as with "undefined"), but instead evaluation continues around errors.

This is already visible in the previous example because $(3 + 4)$ in $(3 + 4) * \langle y \rangle$ is evaluated to 7 even though the multiplication term is not reducible (we assumed $\langle y \rangle$ was not a number).

We will now give two more examples of when this is useful. First, consider the expression $(\lambda f \doteq f\ 2 + 8)\ (\lambda numerator \doteq \lambda denominator \doteq numerator/denominator)$. This expression contains a type error because f is given too few arguments.

This evaluates to $\boxed{\lambda denominator \doteq 2/denominator} + 8$. In this case, the construction site is used as a means of annotation of the error instead of isolation. The reason we use construction sites instead of a traditional red underline is that construction sites are much clearer when they are nested, e.g. $1+\boxed{2+\boxed{3}}$ instead of $1+2+3$. A dedicated annotation may replace these construction sites in the future.

The key benefit this example demonstrates is that the programmer gets type errors even without static type checking. However, unlike static type checking, the code is only checked when it is evaluated by the main expression. Consequently, this method only proves there are no type errors when the main expression covers alls branches of the program. Therefore, static type checkers remain useful. We refer the reader to the Hazelnut calculus by Omar et al. [15] for an excellent example of an error-tolerant type system.

The type errors generated by evaluation do have an advantage over the statically generated ones, though. The ill-typed value and the surrounding successfully evaluated values provide information that can help resolve the issues. The variable names in the example above make clear that we are specifically missing the denominator argument. In contrast, a statically generated type error would only say we need a number.

One of the largest benefits of having a static type checker is that it makes type errors readily available. Error-tolerant evaluation can do the same for errors that would typically require a dependent type system (and therefore extra effort from the programmer in the form of proofs). Suppose the programming language supports array literals in the form of $[1, 2, 3]$ and a primitive operation $index : Int \to Array(a) \to a$[1]. The programming environment would automatically report that the expression $(\lambda xs \doteq index\ 0\ xs + index\ 2\ xs)[1, 2]$ evaluates to $1 + \boxed{index\ 2\ [1, 2]}$. This works even when the literal values are replaced with variables and the value's properties become complex (as long as the program actually makes use of the property).

Seamless debugging We will now continue with examples specifically demonstrating the live programming features. The feedback provided by the live programming features of our programming environment provide similar information to that provided by a time-travel-debugger. However, in contrast to traditional debuggers, the information is available seamlessly, i.e. the programmer does not need to take any action to receive it. Additionally, the execution of the program is navigated automatically based on cursor position, which removes the need for adding break-points every time. However, the approach we developed for this is not yet sufficient for real-world usage because our example language lacks I/O and this significantly complicates debugging (see section 8.1). Additionally, we are still missing stack-trace-based navigation (i.e. stepping out of or back into function applications).

Consider the implementation of Euler's algorithm below:

[1] These are currently not implemented in our example language)

$$
\begin{aligned}
&gcd\ 12\ 9\\
&\quad where\\
&\qquad gcd = \lambda a \doteq \lambda b \doteq if\ \ b == 0\\
&\qquad\qquad\qquad\qquad\quad then\ |b\\
&\qquad\qquad\qquad\qquad\quad else\ gcd\ b\ (a\ \%\ b)
\end{aligned}
$$

Here, we use % for the modulo operation. Our programming environment shows that the result of the main expression is 0, which is incorrect. To inspect, we move the cursor to the only non-recursive case, annotated with | above. Panel 5 from figure 4.1 shows the value of this node is indeed 0. A quick look in panel 4 (which we call the context inspector) will also show that $b = 0$. It also shows $a = 3$, which is the correct answer. The programmer may now see that simply replacing the b under the cursor with a resolves the issue. After this edit, the programming environment immediately confirms this by showing 3 for the result of the main expression.

The *gcd* function is evaluated 3 times in total, but with the cursor at its current location, the programming environment only shows the last instance because this is the only one where the branch with the cursor is evaluated. As this example demonstrates, this can help the programmer find the relevant instance more quickly.

If we want to explore the other evaluations, we can move the cursor to the other branch in the if-expression and browse through the evaluations with the triangles in the head of the "focused node evaluation" panel. Doing this, we would see that a has the values 12 and 9 when this branch is evaluated.

Additionally, this feature may help narrow down the source of the issue because it cannot be in branches that are not evaluated.

Non-termination robustness One fundamental issue that arises when we evaluate potentially unfinished code automatically is that we frequently encounter expressions without normal forms. These cases must not make the programming environment unresponsive because the live programming features could become a nuisance more than a delight otherwise. Ideally, the impact of these cases on the displayed runtime information is also minimized. In our solution, we distinguish between expressions only lacking a full normal forms and expressions lacking weak-head normal forms as well.

First, all panels that show normal forms can be configured (using the number picker in the header) to limit the rendering of expressions to a certain depth. Combined with the "lazy" normalization of expressions (the normalization of parts of the normal form that are not shown is deferred until those parts are requested), this constitutes a crude solution to the problem for expressions lacking full normal forms.

Consider the following implementation of the factorial function:

$$
\begin{aligned}
&factorial\ 6\\
&\quad where\\
&\qquad factorial = \lambda n \doteq if\ \ n \leq 1\\
&\qquad\qquad\qquad\qquad\quad then\ n\\
&\qquad\qquad\qquad\qquad\quad else\ n * factorial\ (n-1)
\end{aligned}
$$

With the cursor on *factorial* in the main expression, panel 5 will show its definition. However, since the function is recursively defined, the definition is substituted for the recursive application

forever, yielding an infinitely large expression. But when the rendering depth is limited to 20, subexpressions are elided after 4 iterations, showing:

$$
\begin{aligned}
\lambda n \doteq & \, if \; n \leq 1 \\
& then \; n \\
& else \; n * (\, if \; (n-1 \leq 1 \\
& \qquad then \; n-1 \\
& \qquad else \; (n-1) * (\, if \; n-1-1 \leq 1 \\
& \qquad \qquad then \; n-1-1 \\
& \qquad \qquad else \; (n-1-1) * (\, if \; ... \\
& \qquad \qquad \qquad else \; ...))) \\
& \qquad \qquad \qquad then \; ...
\end{aligned}
$$

While this technique's primary purpose is keeping the programming environment responsive, this example shows that it can also reveal patterns generated by the recursion (i.e. the repeating $n - 1 - 1 - ...$). There are more benefits from having a tool to interactively explore values, but we leave this to the imagination of the reader since this is not the subject of this thesis.

To minimize the effects of expressions that also lack a weak head normal form, the programming environment allows the programmers to limit evaluation to a number of steps using the controls in panel 1. We call this number the fuel limit and this mode of evaluation *fuel-limited* evaluation. But instead of threading this number through the various branches of evaluation (e.g. the expressions on each side of a $+$) we distribute it over the branches. Thus, one branch burning all the fuel does not prevent evaluation of another. This still guarantees termination, but has the nice property that subexpressions without normal forms are "pointed out" in the larger normal form.

We demonstrate this with the following example:

$$
\begin{aligned}
& evilFactorial \; 6 \\
& \quad where \\
& \qquad id = \lambda x \doteq x \\
& \qquad evilFactorial = \lambda n \doteq if \; n \leq 1 \\
& \qquad \qquad then \; n \\
& \qquad \qquad else \; if \; n == 3 \\
& \qquad \qquad \quad then \; evilFactorial \; n * n \\
& \qquad \qquad \quad else \; evilFactorial \; (n-1) * id \; n
\end{aligned}
$$

In this program, there is a special case in the factorial definition where n does not decrease. This is meant to simulate an unintentionally created expression without weak head normal form. Nevertheless, the main expression evaluates to $\boxed{\langle OutOfFuel \rangle \; \langle OutOfFuel \rangle} * 3 * 3 * 3 * 4 * 5 * 6$. Here, we use a construction site to annotate the failed β-reduction of an application term. The subexpressions in this term are replaced by placeholders because their normalization would require more fuel. Remarkably, this result leads the programmer to the case that contains the expression without a normal form: when $n = 3$.

Generally, the partial results produced in the event fuel runs out show a "computation tree", where the redex at which the fuel ran out prevents the reduction of all its parent redexes, but not its siblings (in the example above, $id \; n$ is still reduced). These partial results expose patterns in the computation tree that can lead the programmers to the source of the issue.

The standard and fuel-limited mode of evaluation are both applied automatically by the programming environment. If the standard mode has not yielded any results within half a second, fuel-limited evaluation is started in parallel. If the standard mode yields results later, these replace the results obtained by fuel-limited evaluation.

In most cases, the evaluation running in standard mode will consume more and more memory as it goes on. To prevent this from becoming a problem, we also limit the memory usage. This limit is configurable through GHC's runtime system options[2].

In summary, these features keep the programming environment responsive and can even provide information that is essential to solving non-termination issues, instead of leaving the programmer to guess and wonder where they went wrong.

Exploring computations The fuel-limited evaluation mode from the previous section also provides a crude mechanism for exploring computations because it allows us to "step through" the computation by varying the fuel limit. Our approach of distributing the fuel over evaluation branches is less suited for this use case than threading the fuel through evaluation branches would be, but we think the use case is worth showing nevertheless.

By default, the partial results the fuel-limited evaluation mode produces are replaced by those from normal evaluation (when available). Panel 1 from the interface includes a checkbox to disable this and use fuel-limited evaluation by default.

Additionally, it would be more helpful to see the values of subexpresssions instead of placeholders. Since evaluation is intentionally halted early in this use case, we can reset the fuel limit for the evaluation of subexpressions when fuel runs out. Intuitively, redexes where the fuel runs out are skipped.

Consider again the correctly defined $factorial$ function:

$$\begin{aligned}
&factorial\ 6\\
&\quad where\\
&\qquad factorial = \lambda n \doteq if\ n \leq 1\\
&\qquad\qquad\qquad then\ n\\
&\qquad\qquad\qquad else\ n * factorial\ (n-1)
\end{aligned}$$

If we make fuel-limited the default, we get the following results for fuel limits 12, 11 and 10:

$$6 * (5 * (4 * (3 * \boxed{factorial\ 2})))$$

Figure 4.2: Factorial with fuel limit 12.

$$\begin{aligned}
6 * (5 * (4 * (\ &if\ \boxed{3 <= 1}\\
&then\ 3\\
&else\ \boxed{3 * 2})))
\end{aligned}$$

Figure 4.3: Factorial with fuel limit 11.

[2] https://downloads.haskell.org/~ghc/8.10.7/docs/html/users_guide/runtime_control.html#rts-flag--M%20%E2%9F%A8size%E2%9F%A9

$$
\begin{aligned}
6 * (5 * (4 * \; & \boxed{if\ \boxed{4-1} <= 1} \\
& then\ \boxed{4-1} \\
& else\ \boxed{4-1} * (if\ ... \\
& \qquad\qquad\quad then\ ... \\
& \qquad\qquad\quad else\ ...)))
\end{aligned}
$$

Figure 4.4: Factorial with fuel limit 10.

Figure 4.2 clearly shows how the factorial of 6 is computed. In cases such as this, where fuel has run out at an application term, the subexpressions in function position are not expanded, even though the fuel limit is reset. This produces more digestible results in this use case of fuel-limited evaluation.

Because fuel is distributed over evaluation branches, we see two construction sites in figure 4.3. The first one prevents the reduction of the if-term. In the second, the reduction of the multiplication term is skipped, but $factorial$ $(3-1)$ is still evaluated to 2 because the fuel is reset after skipping a redex.

Skipped redexes prevent parent redexes from being reduced in the same way that an $\langle OutOfFuel \rangle$ placeholder does. This exposes the computation tree, but it can also "blow up" results when they are substituted in a subexpression of another skipped redex, which is demonstrated in figure 4.4. In this case, fuel ran out *immediately after* the reduction of $factorial$ $(4-1)$. Consequently, $4-1$ was not reduced and substituted into the body of the factorial function. Fuel also runs out at the reduction of the if-term in the body of the factorial function. The construction site that is used to annotate this spans multiple lines, which is depicted by open sides at the line breaks.

The skipped reduction of $4-1$ also prevents the reduction of the condition in the if-term, the argument of the recursive application of $factorial$ and thus also the reduction of the if-term in its body (the subexpression of which are elided by the limited rendering depth).

In addition to threading the fuel through evaluation branches instead of distributing it over them, this problem may also be solved by resuming the reduction of skipped redexes when they are found in subexpressions of other skipped redexes. Unfortunately, we did not have time to experiment with this solution.

Chapter 5

Error-tolerant structure editors

In this chapter, we will show precisely how construction sites can be used in a structure editor to preserve information from the AST that is essential to isolating errors. This flexibility of construction sites allows for the implementation of an editor that retains the interface of standard text editors (and thus alleviates the inflexibility issues of strict structure editors discussed in section 2.2), while the preserved information supports analysis by IDE services (presented in chapter 6. Furthermore, it provides a more complete structured representation of the program than any error-correcting parser could (at least in interactive settings).

In this chapter, we first review the architecture of an IDE with a standard text editor to establish a base to work from. Subsequently, we give an overview of Frugel's architecture and detail its components and their implementation.

5.1 Architecture of a standard IDE

The architecture of an IDE with a standard text editor is illustrated in figure 5.1.

The architecture has two components: an editor and a language engine. The term language engine is chosen to mean anything that provides language specific operations on a program. It could be a plugin for the editor, a standalone program like a compiler, a server implementing the Language Server Protocol[1] or it could be an internal component of the editor if the editor is specifically built for the language.

At startup, the editor typically loads a file from the file system into a text buffer. The arrow in figure 5.1 corresponding with this operator is thin because it does not transform the data significantly. It is also dashed because the operation only happens once for the entire editing session (or until the file is modified by another program and the programmer wants to load the new version) or not at all if the programmer creates a new file through the editor. The contents of the text buffer are then updated based on the current contents and keystroke input. The arrow denoting the reuse of the text buffer contents is wider because it represents a significant transformation.

The programmer may receive feedback about the program by providing it as input to the language engine. Usually, the text is then parsed into an AST which can be analysed. Finally, the resulting analysis is rendered into a human friendly form. In practice, the feedback from the

[1] https://microsoft.github.io/language-server-protocol/

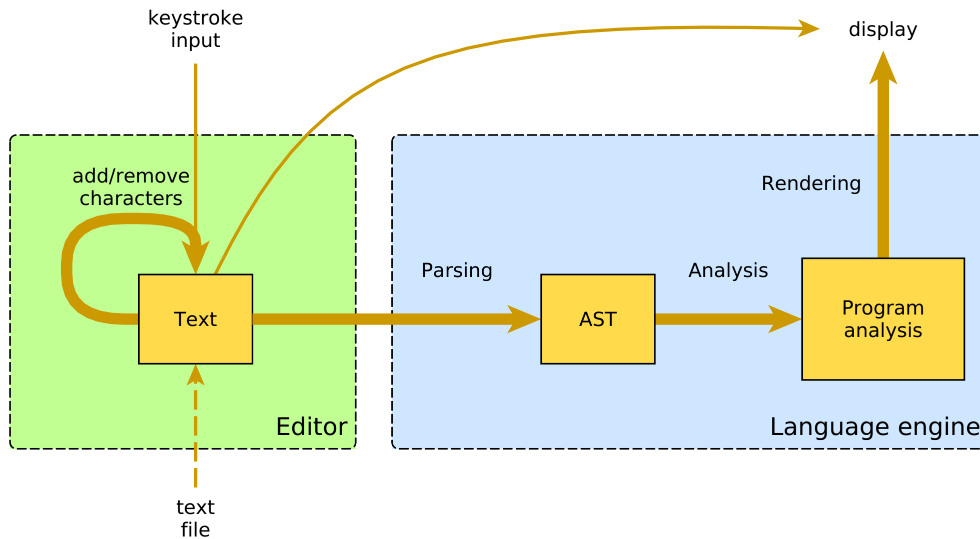

Figure 5.1: The architecture of a standard IDE.

language engine is presented to the user by the editor and the editor may also do some rendering work, but this is omitted from the diagram because it is irrelevant for the comparison with Frugel's architecture.

5.2 Frugel's architecture

Figure 5.2 gives an overview of Frugel's architecture. We discuss its components below and give some examples to illustrate the various transformations at the end.

Shared component The standard IDE's architecture (figure 5.1) has changed in a few ways. First, a new shared component is added which overlaps both the editor component and the language engine. The editor requires operations on the document (such as decomposition) that are encapsulated in the interface this shared component represents. However, in contrast to most structure editors, the bulk of the work of most operations is defined generically and the language-specific parts are relatively simple. To signify this, the labels of these operations labels are placed within the shared component.

The shared component is encoded by the `Editable` type class:

```
import Data.Data

class ( Data n, Decomposable n, CstrSiteNode n, Parseable n) => Editable n
```

This class splits the interaction into multiple more specific type classes. Except for `Data`, these will be discussed in detail throughout the rest of this section. `Data` can be derived by GHC

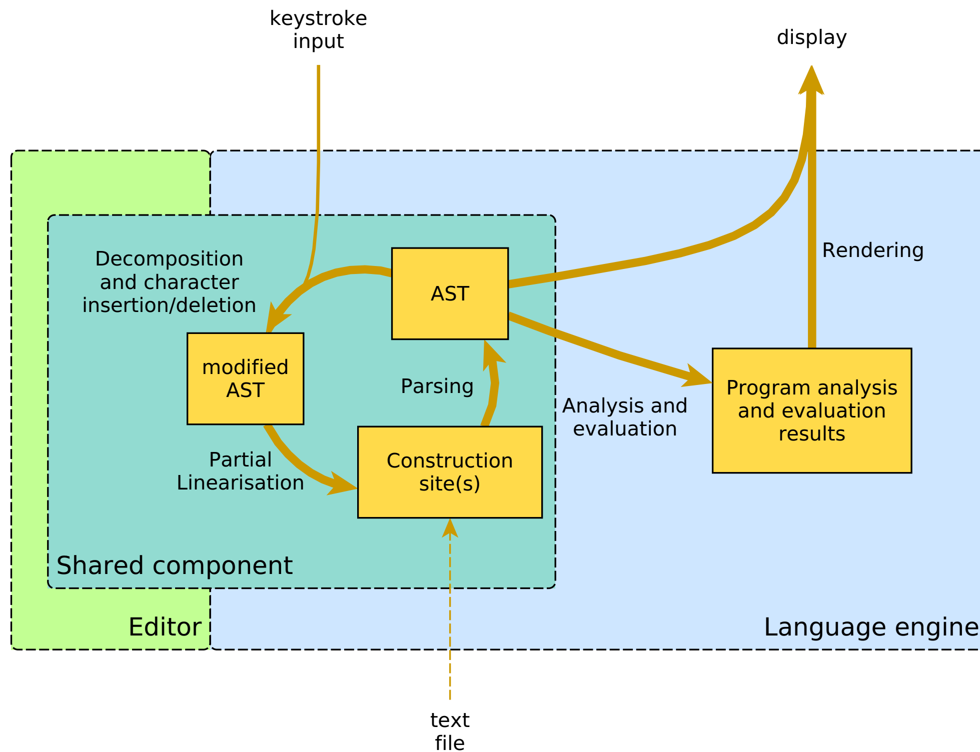

Figure 5.2: Frugel's architecture.

automatically using the `DeriveDataTypeable` language extension.

There is one law that demands a kind of consistency between the functions from htese more specific type classes. We discuss this law after the `Decomposable` class is introduced in section 5.3.

The edit loop Next, the most significant difference is the composition of the edit-loop. In the standard IDE diagram, the loop only consists of text and a transition inserting/removing characters from it. In a structure editor, edit actions are performed directly on the AST. Frugel's edit loop consists of three steps: decomposition (with character insertion/deletion), linearization and parsing. These operations will be explained in more detail in section 5.3, 5.4 and 5.5, respectively. In short: The decomposition step finds and appropriate AST node for the edit action and decomposes that node into a construction site. The "plain-text"/textual edit action triggered by the keystroke is then performed on this construction site (e.g. a character is inserted or deleted). The decomposition is an *in-place* transformation, so the result of this step is a modified AST where at most one more node than before is decomposed and as much information as possible is preserved.

Finding an appropriate AST node to decompose also requires information on the location of the cursor. This will be represented with a caret | in the examples below. How this information is actually represented in the system's implementation is described in section 5.7.1.

Generally, the system tries to decompose the lowest AST node that (1) the cursor is in and (2) allows for the edit action that needs to be performed. In the case of the language from chapter 3, this means that insertion happens in leaves in most cases and deletion happens at the node to which the deleted character belongs (e.g. + in $1 + 2$).

While all construction sites in the AST that is rendered will always contain syntax errors, this is not necessarily true for the modified AST or the construction sites resulting from linearization.

The linearization step flattens most of the modified AST into various "root construction sites" based on the location of construction sites in the modified AST. These root construction sites vary in which syntax errors are still isolated in nested construction sites and which are not. This is necessary, to allow isolated characters from different construction sites from the modified AST to match grammar rules together (e.g. matching parentheses) without giving up syntax error isolation altogether.

Finally, the parser tries to parse these root construction sites in order of least remaining isolated syntax errors, thus resolving all construction sites that do no longer contain parse errors and producing a new AST.

The AST In contrast to many structure editors, programmers are allowed to manage whitespace themselves, just like in a text editor. This means they can freely separate code with spaces, tabs or newlines and apply a formatter whenever they wish (possibly automatically after every edit, if replicating the behaviour of most structure editors is desired).

To accomplish this, whitespace needs to be included in the AST and preserved throughout the edit loop. Section 5.7.1 describes how this is accomplished in detail.

Semantic edit actions In addition to textual edit actions, the system could also support *semantic* edit actions. These are edit actions that take the semantic information or structure from the AST into account, like refactoring actions or structure-based navigation as it if often found in strict structure editors. This is possible because our editor maintains an AST just like strict structure editors. The flexibility of construction sites simply allows our editor to support both. However, no semantic edit actions have been implemented yet.

Rendering ASTs In Frugel, the default display representation is enhanced with markings to indicate the presence of construction sites. These markings show exactly what characters and nodes cause a syntax error and constitute a transparent form of error resilience.

However, some extra work needs to happen to allow these markings to span multiple lines. During rendering, the markings are split at line breaks, which allows them to flow over line them. This can cause many markings to be stacked on top of each other when nested construction sites span multiple lines, which unfortunately has a confusing effect. We had to leave improving upon this method to future work because it is not the focus of this thesis.

Loading a text file Finally, the operations of loading a text file should be clarified. A text file is simply converted to a construction site containing the characters in the file[2]. Parsing this construction site may fail, in which case the advanced level of feedback the editor can normally provide drops away. This situation is called a *cold start*. Since reloading the file with syntax errors

[2]Trivial, but not implemented yet

is usually not performed very frequently, this is not seen as a great drawback. Still, Frugel's architecture does not put any constraints on the parsing technology used, so error recovery techniques may still be implemented to alleviate this issue if desired.

Examples Before each step of the edit loop is described in detail, we give a few simple examples to allow the reader to build an intuition for the system. It is not necessary to understand why everything happens the way it does at this moment. The goal is only to show the system's behaviour.

Consider the expression $1|+2$. The programmer could insert a "3" using the keyboard, resulting in $13| + 2$. Some construction sites are used internally while traversing the edit loop, but these can be removed before the edit loop is completed because the resulting document is syntactically correct.

Pressing the "Delete" key in the previous result will remove the "+" and result in $13|2$. Again, no construction sites appear, because the result is syntactically correct. So far, the system appears to work exactly like a text editor. In contrast to some structure editors, merging the 1 and 2 nodes into one does not require any specific machinery in Frugel.

Furthermore, the programmer can enter the expression $1+2$ linearly (as they are used to in text editors), i.e. by entering "1", "+" and then "2". This contrasts with some other structure editors where expression need to be entered "outside-in", i.e. entering "+", "1", moving to the right and finally entering "2".

Suppose now that the programmer introduces a syntax error, e.g. by entering "(" in $|1 + 2$. This would result in $(|1 + 2$. The syntax error is contained in the left child of the plus node and most of the AST is preserved. When the programmer enters a matching parenthesis behind the "2", internally the "2" and new parenthesis are placed in a construction site in a similar way, but because the document is syntactically correct as a whole, both construction sites are removed. The programmer only sees $(1 + 2)$ as the result.

Insertion happens in leaves of the AST in most cases, so any construction sites that are inserted will be small. With deletion, it is more often necessary to decompose a non-leaf node, leaving the children of the node as complete nodes in the AST. Consider the expression $f(\lambda x = x+)|\ g\ y$. The syntax error is isolated to a leaf node while most of the structure remains intact. If we were to press the "Backspace" key, the closing parenthesis before the caret is deleted. The node enclosed in the parentheses is still syntactically correct and it would be wasteful to discard the information contained in its structure. To preserve this information, the enclosed node is placed in the resulting construction site as a complete node, like so: $f\ (\lambda x = x +|\ g\ y$.

The first advantage of the system is that it *rules out* a significant part of the document as a possible source of the syntax error, which is especially helpful to novices. Furthermore, the way the system handles syntax errors is *transparent*, in contrast to the heuristics used by error-correcting parsers. However, most importantly: analysis can be performed on every edit state without the risk of producing spurious errors (no guessing of the programmer's intent is required). Instead, the analysis simply loses some information.

In some cases, showing outdated information may be preferable, but at least this system allows for a preference. In any case, the drawbacks of showing outdated information weigh increasingly heavier with further code changes because any cached information becomes increasingly outdated. For example, if the programmer would press "Backspace" again, this would remove the "+" and resolve the syntax error contained by the inner construction site and lead to an improved analysis

again. The resulting expression would be f ($\lambda x = x$) $g\ y$.

In summary, these examples demonstrate how the system *preserves the interface of a text editor* while employing a *transparent* and spurious-error-free form of error tolerance and supporting analysis of the document *on every edit state*.

5.3 Decomposition

Decomposition is arguably the most interesting step in Frugel's edit loop. During this operation, a character is inserted or removed from the program while as much of the AST as possible is preserved.

Some edits can be performed in any construction site, but others can fail. For example, insertion is always possible, but a character cannot be deleted from an empty construction site.

Before describing the operation in detail, both cases will be illustrated with an example: If the programmer insert a "3" in $1 + 2$, this always succeeds in the leaf node 1. 1 become 13 and the operation results in $13 + 2$. In this case, only a leaf node of the AST is decomposed into a construction site, which is the minimum for any system that allows insertion of arbitrary text.

However, trying to delete the next character from 1 would fail, so its parent $1+2$ is considered. Deleting the next character in $1 + 2$ succeeds and results in 12. In this case, a non-leaf node is decomposed into a construction site, but all child nodes are left in their structured form. In this example, the complete nodes do not provide much additional information, but the child nodes may be much more extensive in realistic cases. Preserving their structure is what gives Frugel its name and what distinguishes it from other non-strict structure editors that just allow arbitrary text (but no child nodes) to replace a node in the AST.

Again, only the node which the edit action truly acted upon is affected, which also makes it the smallest change possible when allowing the program to be edited as its textual representation.

In both the examples given above, the construction sites will be resolved by the parser and never presented to the programmer because they are both syntactically correct.

Decomposition implementation The operation is split in a generic part, which is based on the structure of construction sites and a AST specific part.

The generic part requires two functions from the language-specific part: one to traverse the components (characters and child nodes) of a node with applicative effects (`traverseComponents`) and one to create a node of the desired type from a construction site given the node before decomposition (`setCstrSite`).

This first function resembles a monomorphic version (the argument function preserve the type of their input) of Haskell's `bitraverse` from the `Bitraversable` class. The second function should correspond to a constructor of the node data type.

Recursive application of `traverseComponents` allows the generic part of the implementation to project out a concrete syntax tree from the AST. This allows us to find the node under the cursor based on the current cursor offset (its index in the textual representation). The node under the cursor is converted to a construction site and the textual edit action (e.g. character insertion) is performed. Then it can use the second function to obtain a construction site node which can be left in the AST.

Before we make this process precise, we discuss the language-specific implementation of these functions.

Language-specific part of implementation These functions are encapsulated with two type classes: `CstrSiteNode` and `Decomposable`:

```
class Decomposable n where
    traverseComponents :: Applicative f
        => (Char -> m Char)
        -> (n -> f n)
        -> n
        -> f n

class CstrSiteNode n where
    setCstrSite :: CstrSite n -> n -> n
```

To recapitulate, the purpose of `traverseComponents f g n` is to allow for effectfully traversing the components of an AST node `n`. It should apply `f` to any characters belonging to the node (e.g. "+") and apply `g` to its child nodes in the order they occur in the corresponding production rule from the language's grammar. For $\langle x\rangle + \langle y\rangle$, that order would be `g <x>`, `f "+"` and `g <y>`. This class becomes more complicated when an AST has multiple node types, as in the example from 3.3. This is discussed in further detail in appendix A.

Similarly to `bitraverse`, a definition of `traverseComponents` should satisfy the following laws:

1. it should preserve **identity**: `traverseComponents Identity Identity` $\equiv$ `Identity`. This ensures the AST nodes are not changed by other means than the argument functions.

2. it should preserve **composition** of its second argument function, but (in contrast to `bitraverse`) not necessarily the first:

   ```
   Compose . fmap (traverseComponents gChar gNode)
       . traverseComponents fChar fNode
    ≡
   traverseComponents (\c -> Compose $ fmap gChar (c <$ fChar c))
       (Compose . fmap gNode . fNode)
   ```

   This ensures that the nodes produced by `fNode` are actually preserved in the result of `traverseComponents`. This requirement is relaxed for the first argument because characters belonging to the notation of a term (e.g. the +-sign for an addition term) should be traversed, but cannot be changed without changing the term to a construction site (which is up to the generic part of decomposition)) Additionally, this ensures nodes and characters are not traversed twice.

3. **Naturality**:

   ```
   traverseComponents (t . fChar) (t . fNode)
       ≡ t . traverseComponents fChar fNode
   ```

where `t :: (Applicative f, Applicative g) => f a -> g a` is a applicative transformation, i.e. one that preserves the applicative operations:

```
t (pure x) ≡ pure x
t (f <*> x) ≡ t f <*> t x
```

This ensures `traverseComponents` is independent of the applicative functor used.

We can now also discuss the law on the `Editable` type class: **Editor parameter consistency**. Suppose we have `decompose :: Decomposable n => n -> CstrSite n` that accumulates components traversed by `traverseComponents` in a construction site and `parse :: CstrSite n -> Either String n` that parses a construction site according to the language's grammar (the function in the `Parseable` class is slightly different, but has the same functionality). Editor parameter consistency mandates that the following equation holds:

```
parse . decompose ≡ id
```

This ensures a consistency between the grammar used by the parser and the node components traversed by `traverseComponents`.

The function `setCstrSite` should create a node of the desired type from a construction site. This corresponds directly to a constructor of the datatype representing the AST in most cases, but it is also given the node from before decomposition for cases where there is no single way to create the node from a construction site. This is the case for the `Node` type in appendix A.

An instance of these classes for expression language from section 3.2 could be defined as in the listing below (omitting whitespace and parentheses for clarity). For an example for the language from 3.3 with multiple node types, see appendix A.

```
newtype CstrSite n = CstrSite [Either Char n]

data Expr =
    = Variable String
    | Abstraction String Expr
    | Application Expr Expr
    | Sum Expr Expr
    | Integer Int
    | ExprCstrSite (CstrSite Expr)

instance Decomposable (CstrSite n) where
    traverseComponents traverseChar traverseNode (CstrSite cstrMaterials)
        = CstrSite <$> bitraverse traverseChar traverseNode cstrMaterials

instance Decomposable Expr where
    traverseComponents traverseChar _ (Identifier name)
        = Identifier <$> map traverseChar name
    traverseComponents traverseChar traverseNode (Abstraction name body)
        = Abstraction <$> map traverseChar name <*> traverseNode body
    traverseComponents traverseChar traverseNode
        (Application function whitespace argument)
```

```
        = Application <$ traverseChar '\\' <*> traverseNode function
            <* traverseChar '=' <*> traverseNode argument
    traverseComponents traverseChar traverseNode (Sum left right)
        = Sum <$> traverseNode left <* traverseChar '+'
            <*> traverseNode right
    traverseComponents traverseChar traverseNode (Integer i)
        = Integer i <$ traverse traverseChar (show i)
    traverseComponents traverseChar traverseNode (ExprCstrSite cstrSite)
        = ExprCstrSite <$> traverseComponents traverseChar traverseNode
                cstrSite

instance CstrSiteNode Expr where
    setCstrSite = const . ExprCstrSite
```

The instance declarations are quite repetitive and may be automatically derived from an encoded grammar, but this is left to future work.

Generic part of implementation Provided with these `Decomposable` and `CstrSiteNode` instances, the generic part of the operation can do most of the heavy lifting. In addition to the AST, this operation also requires information on the position of the cursor. There are multiple approaches for tracking the cursor in a document and these are discussed in more detail in section 5.7.1. In short, the approach taken by Frugel is maintaining an integer representing the cursor's offset in the text representation of the current document. For example, $x| + y$ corresponds with a cursor offset of 1.

Pseudocode for the generic part and an explanation follows below:

```
modifyNodeAt ::
    (MonadError InternalError m, Decomposable n, CstrSiteNode n)
    => (Int -> CstrSite n -> m (CstrSite n))
    -> Int -> n -> m n
modifyNodeAt f cursorOffset program
    = runStateT (traverseNode program) $ initialDecompositionState
    cursorOffset
  where
    traverseChar = ... -- Updates to internal state
    traverseNode n = ifM passedCursor (pure n) $ do
        incrementCstrSiteOffset
        withLocalCstrSiteOffset $ do
            newNode <- traverseComponents traverseChar traverseNode n
            ifM (cursorAtCurrentNode && editNotYetPerformed)
                (catchError
                    (setCstrSite <$> transform n ?? n <* editPerformed)
                    (const (pure n))
                (pure newNode)
    transform n = do
        cstrSiteOffset <- gets stateCstrSiteOffset
        lift . f cstrSiteOffset $ decompose n
```

```
decompose :: Decomposable n => n -> CstrSite n
decompose n = ...
```

The function `modifyNodeAt` uses an internal state to count how many characters have been processed, the number of components of the current node (traversed by `traverseComponents`) that have been processed and whether the edit has succeeded or not. The number of components processed is also called the construction site offset (`cstrSiteOffset`) because this number corresponds with an offset within the current node's construction site if it would be decomposed.

Counting the number of characters processed allows for finding the AST node the cursor is currently at (further explained below). This could be optimized by caching the length of nodes.

The error monad is used to handle failure of the edit action (represented by `f`) gracefully (e.g. trying to delete characters when the cursor is at the end of the document). In this case, the edit action is discarded.

To kick off the process, `traverseNode` is applied to the root of the AST. This function is applied recursively to traverse the AST. Thus, changes to nodes occur in-place. This function `traverseNode` first checks that we have not passed the cursor yet using the internal state. In that case, we can just use the original node as a result. Else, the components of the current node are traversed and processed. This processing maintains a local value of the construction site offset, i.e. it is reset to zero before starting and restored to its original value when all components are processed. Due to this local processing, we can conveniently increment the outer value of the construction site offset in advance. Processing the current node and its components consists of applying `traverseComponents` to the current node to attempt to perform the edit action on its children and possibly applying the edit action to the current node. Processing the current node's children first like this gives rise to a "bottom-up" manner of processing, where decomposition occurs as low in the tree as possible.

Before continuing, some terminology should be introduced. We say the cursor is at a node when the cursor offset corresponds to any location within the first character and last character (inclusive) belonging to a node. For example, for $1| + 2$, this means the cursor is considered to be at both the nodes $1|$ and $1| + 2$.

If the cursor is at the current node and performing the edit on the children failed, an attempt is made to perform the edit at the current node `n`. Otherwise, the updated node is returned. Performing the edit is encapsulated in the function `transform` described below.

If editing the current node is successful, the state is updated accordingly and the new node is returned, but, if this fails, the original node is returned.

The function `transform` first retrieves the offset of the cursor within the construction site that would result from decomposing the node. Then, the node is decomposed, the edit is attempted and the resulting construction site is converted back into a node using `cstrSiteOffset`. Thus, a successfully edited node is always in construction site form. This is how construction sites reflecting the edits are inserted into the AST.

The function `decompose` simply decomposes a node using `traverseComponents`. For example, decomposing the term $1 + 2$ would result in $\boxed{\boxed{1} + \boxed{2}}$.

With a little modification, `modifyAt` could also be used to implement the *semantic* edit actions described before.

In summary, this operation always decomposes a node into a construction site if the edit action can be applied. However, the number of construction sites in the tree may stay the same because

decomposing a construction site results in the same construction site.

Conservative decomposition With the algorithm described above, entering a "(" in $|\lambda x = x$ would decompose the expression into $(\lambda x = x)$. This is suboptimal because a node is decomposed while nothing has changed inside the node. To improve this, we change the above algorithm slightly by adding *conservative decomposition*.

The idea behind this kind of decomposition is to include a node as a complete node in the result construction site when the modification is performed on the edge of a node (i.e. at the start or end). This would have the decomposition example above result in $(\lambda x = x)$. Note that this has nothing to do with the inserted character being a parenthesis. The same would happen for any character. Decomposition still happens as low in the AST as possible (in/close to the leaves).

Notably, conservative decomposition can be implemented completely generically for the language from section 3.2 and with minimal additional work from the language engine for the language from section 3.3 with multiple types of AST nodes. This work effectively consists only of annotating the types of AST nodes that cannot occur in construction sites (e.g. the root node).

Such a generic implementation consists of a new definition for `transform` (in the definition of `modifyNodeAt`) and the function `conservativelyDecompose`:

```
modifyNodeAt f cursorOffset program = ...
  where
    ...
    transform n = do
        cstrSiteOffset <- gets stateCstrSiteOffset
        lift $ case conservativelyDecompose cstrSiteOffset n of
            Just (cstrSiteOffset', cstrSite) -> catchError
                (f cstrSiteOffset' cstrSite)
                $ const (f cstrSiteOffset $ decompose n)
            _ -> f cstrSiteOffset $ decompose n

class Decomposable n where
    traverseComponents :: ...
    conservativelyDecompose :: Int -> n -> Maybe (Int, CstrSite n)
    conservativelyDecompose cstrSiteOffset n = case cstrSiteOffset of
        0 -> Just (0, singletonCstrSite)
        l | l == length (toList $ decompose n) -> Just (1, singletonCstrSite)
        _ -> Nothing
        where
            singletonCstrSite = fromList [ Right n ]
```

In the new definition of `transform` we first check that conservative decomposition is possible for the node in question and the current construction site offset (case analysis on `conservativelyDecompose cstrSiteOffset n`). If so, we do not just get a singleton construction site with the node, but also a new construction site offset corresponding to either the start or end of that construction site. Then, we attempt to perform the edit action with that construction site and offset. IF this fails, we fall back to attempting to perform the edit action with the construction site resulting from standard decomposition and the old construction site offset. If conservative decomposition is not possible, we fall back to standard decomposition as well.

The function `conservativelyDecompose` implements the logic deciding when conservative decomposition is possible and returns a corresponding construction site and offset. This function has to be generalized slightly for the case of multiple types of AST nodes. This is expanded upon in appendix A.

The astute reader may notice that, without restrictions, conservative decomposition would hinder reuse of construction sites, e.g. entering "f" in |(1 would result in f|(1 instead of f|(1. Conservative decomposition could be restricted to non-construction site nodes, but this issue already gets resolved by a small operation that improves conservativeness of the system in the face of ambiguity. This is described in detail in 5.6.

The remainder of the edit loop The performed edit action might not actually introduce a syntax error. It may even resolve one that was already present. The construction sites that do not/no longer contain a syntax error should be resolved again before the new AST is presented to the user or used for analysis. This is the purpose of the next two steps.

5.4 Partial linearization

The term linearization is borrowed from Voelter et al.'s work on grammar cells[25]. It means to convert something with a non-linear format (e.g. a tree) into a linear one (e.g. a list or a construction site). It can also be seen as a recursive decomposition of the nodes in a tree. For example, the term $1 + 2 * 3$ could be linearized to the characters "1", "+", "2", "$*$" and "3".

A linearization step is required because a parser is usually not capable of constructing a new AST from an older AST with some unstructured parts (i.e. construction sites). Therefore, the AST resulting from the previous step needs to be converted to such a linear format before it can be parsed.

However, if the AST would be converted to a plain sequence of characters, information that helps contain possible syntax errors would be lost. If the AST is instead *partially* linearized to a construction site, this information could be retained by wrapping construction sites in the AST in complete nodes. For example, (1 $+ 2$ could be linearized to (1 $+ 2$. Since the cursor position is not relevant any more from this point on, the caret will be omitted. Linearizing to a construction site does require changing the token type of the parser to a sum type of characters and nodes, but fortunately parsing algorithms are usually independent of the token type.

In the above example, a construction site is contained in another construction site. For disambiguation, we will call the outer construction site the *root construction site* and all others *nested construction sites*. The root construction site serves as the input for the parser.

Since the nested construction sites could again contain complete nodes with nested construction sites, there is still a kind of tree structure in the data. In general, the complete AST is linearized, except for the construction site nodes. Therefore, this operation is named partial linearization. From this point on, we will simply use linearization to refer to this kind of partial linearization.

Note that before parsing, it is unclear whether any of the construction sites in the AST contain a syntax error or not. All construction sites in the AST that was last presented to the programmer contained one at that point (because the edit loop guarantees this), but the addition or removal of a character from the decomposition step might have resolved one or more of these syntax errors.

For example, if the left parenthesis in (1 + 2) was just inserted, the second construction site does not contain a syntax error any more.

To achieve the desired final AST (nested construction sites that do contain a syntax error do not grow in size, but the ones that do not contain one are resolved) the parser would have to attempt parsing the contents of all construction sites, but if this fails, fall back to a parse state where the parser "skips over" the contents of a nested construction site.

However, recovering like this is challenging because as with the example above, the parser has to look beyond individual construction sites to see whether it contains a syntax error or not. To the best knowledge of the author, the only existing parsing technology that could do this efficiently is GLR or GLL parsing because it is capable of "cloning the head of the parser" and maintaining multiple parser states at once.

It might also be possible to adapt a different existing parsing algorithm to handle nested construction sites this way or develop an entirely new one, but this was not a priority for this thesis because it is not needed for a proof of concept.

In fact, Frugel implements a much more naive approach where much of the burden of finding the desired result is offloaded to the linearization step. This approach allows for the use of any parsing technology, but does multiply the asymptotic computational complexity of the parser with a factor that grows exponentially with the number of nested construction site.

It also requires a slight change to the grammar, which is described in detail in section 5.5. In short: for each non-terminal that represents a type of node (e.g. an expression or a definition), a production rule is added that produces a complete node of that type. This allows the parser to accept a complete node of the right type when it encounters one in the root construction site and skip over the contents of any construction site nested in that node.

For the sake of this explanation, we will first limit ourselves to the case of ASTs with only a single construction site: (1 + 2. Before wrapping a construction site in a complete node, we could first attempt to parse a root construction site where the contents of the nested construction site are simply spliced into the root construction site, e.g. (1 + 2. In this example, the parser would fail, but if we have the parser skip the contents of nested construction site, parsing of (1 + 2 succeeds (because all characters read originate from complete nodes).

However, when multiple construction sites are present in the AST, we have to consider all combinations of nested construction sites because some may only provide the correct syntax for a node together (e.g. two parentheses in different construction sites) and there may still be others whose syntax errors are not resolved (so they must be excluded). Then we attempt to parse all the resulting root construction sites in order of most nested construction sites spliced in (described in more detail in the next section).

Concretely, for the AST (1 + 2) the following root construction sites will be generated (in the order they will be attempted to be parsed): (1 + 2), (1 + 2), (1 + 2) and (1 + 2). Since parsing of the first variation will succeed, there is no need to attempt to parse the others. In fact, they are not even generated because the system is implemented in a lazy programming language.

In the past examples, all nested construction sites consisted only of characters. However, it is also possible that the nested construction sites contain complete nodes with more deeply nested construction sites. To be able to resolve these deeply nested construction sites while the syntax error from the parent construction site is not yet resolved, all complete nodes are linearized in the

same way as the root AST node. Thus, linearization of these nodes may also generate multiple construction sites, but parsing of these only happens for the selected variation of the root construction site.

While this approach increases the asymptotic computational complexity of the parsing step considerably, this may not be a critical issue for real-world systems because the number of construction sites seems to remain relatively small during normal usage. Furthermore, this approach does lend itself very well to parallelization, which may reduce the impact on real use performance even further. However, this was not a priority for the prototype.

Alternatively, requirements on the parser can be added to mitigate this inefficiency. A GLR or GLL parser could be used to efficiently manage the various root construction sites that we should attempt to parse. An incremental parser could also offer a significant performance improvement because there is a lot of overlap in the contents of construction site variations.

It is also important to note that the granularity of error-tolerance in this system is at the level of construction sites, i.e. the parser attempts to parse a construction site completely, or not at all. If a construction site contains multiple syntax errors, this means that it cannot be resolved until all of them are fixed. This limitation is discussed in more detail in section 5.6.

Partial linearization implementation In addition to requiring an instance of `Decomposable` for the node types, this operation requires another language-specific function: one that checks if the argument node is a construction site. Since this should be possible for exactly the same nodes that have a construction site form (instances of `CstrSiteNode`), this function is added to the `CstrSiteNode` class. This function is usually trivial to implement:

```
class CstrSiteNode n where
    setCstrSite :: CstrSite n -> n -> n
    isCstrSite :: n -> Bool

instance CstrSiteNode Expr where
    setCstrSite = ... -- given above
    isCstrSite ExprCstrSite{} = True
    isCstrSite _ = False
```

The generic part of the operation is defined as follows:

```
linearise :: (Decomposable p, IsCstrSite n) => p -> CstrSite n
linearise program
    = foldr addItem (singleton $ CstrSite []) rootCstrMaterials
  where
    (CstrSite rootCstrMaterials) = decompose program
    addItem item@(Left _) variations = cons item <$> variations
    addItem item@(Right node) variations
        = (if isCstrSite node
              then cons item <$> variations
              else mempty)
        <> (mappend <$> linearise (decompose node) <*> variations)
```

The operation is kicked off by applying `decompose` to the root node of the AST. This results in a construction site that is folded into a list of construction sites (variations where different nested

construction sites are spliced in) using a singleton list with an empty construction site as starting value and the following folding operation:

1. If a character is encountered, it is added to all variations.
2. If a node is encountered, we check if it is a construction site with `isCstrSite`. If this is true, a list of variations with the node prepended as is, is created. This new list of variations is concatenated to the list of variations that results from concatenating elements of a "cross-product". The first factor of the cross product is the list of variations obtained by applying `linearise` recursively to the node. The second factor is simply the existing list of variations.

   A node that is prepended as is, may include new construction sites and should therefore be reparsed as well. The linearization that this requires is postponed until it is picked by the parser.

In Frugel's real implementation, many of the lists in this definition are replaced by sequences from the *containers*[3] package because these have $\mathcal{O}(\log(\min(n1, n2)))$ concatenation.

5.5 Parsing

While Frugel does not impose any requirements (with the naive approach described in the previous section) on the type of parser used (LL, LR, with or without lookahead, based on parser combinators or generated by a parser generator), the input type of the parser must be changed to a construction site.

Doing this is easier with some technologies than others, but parser algorithms are generally independent of the input type, so this is not regarded as a significant requirement.

Additionally, the parser needs to be run on the various construction site variations from the previous step. The interface between the parser and the editor is once again encapsulated in a type class (introduced later in this section).

Finally, the language's grammar should be adapted to the new input type. First, consider the following grammar for the simple extended λ-calculus described in section 3:

```
⟨expr⟩ ::= '\' ⟨ident⟩ '=' ⟨expr⟩
  | ⟨expr⟩ ⟨expr⟩
  | ⟨expr⟩ '+' ⟨expr⟩
  | '(' ⟨expr⟩ ')'
  | ⟨ident⟩
  | ⟨number⟩

⟨uppercase⟩ ::= 'A'
  | 'B'
  | ...
  | 'Z'

⟨lowercase⟩ ::= 'a'
  | 'b'
  | ...
  | 'z'

⟨ident⟩ ::= ⟨letter⟩ ⟨alphanumeric⟩

⟨alphanumeric⟩ ::= ⟨letter⟩
  | ⟨digit⟩

⟨letter⟩ ::= ⟨uppercase⟩
  | ⟨lowercase⟩

⟨number⟩ ::= ⟨digit⟩*

⟨digit⟩ ::= '0'
  | '1'
  | ...
  | '9'
```

[3] https://hackage.haskell.org/package/containers

To each non-terminal that is represented by a node in the AST a production rule should be added that produces complete nodes of that type (the other type of "token" in construction sites). We denote these nodes with terminals in curly braces, e.g. `{Expr}`. These production rules allow the parser to accept a complete node of the right type for the corresponding non-terminal.

Additionally, syntax for empty construction sites should be added. The grammar for expressions now looks as follows:

$$
\begin{aligned}
\langle \mathit{expr} \rangle ::= &\ \text{`\textbackslash'}\ \langle \mathit{ident} \rangle\ \text{`='}\ \langle \mathit{expr} \rangle \\
| &\ \langle \mathit{expr} \rangle\ \langle \mathit{expr} \rangle \\
| &\ \langle \mathit{expr} \rangle\ \text{`+'}\ \langle \mathit{expr} \rangle \\
| &\ \text{`('}\ \langle \mathit{expr} \rangle\ \text{`)'} \\
| &\ \langle \mathit{ident} \rangle \\
| &\ \langle \mathit{number} \rangle \\
| &\ \text{`...'} \\
| &\ \text{`\{Expr\}'}
\end{aligned}
$$

Note that the added production rule is different from the notation for construction sites introduced in section 3. The production rule produces a complete node (which can be a construction site), but the notation added in section 3.2 refers directly to construction sites, which may contain complete nodes of any kind. No new construction sites are created during parsing.

Finally, one should pay attention to cases where lookahead is used because the targeted characters may also occur in a complete node.

Parsing implementation As with all language-specific components, there is an interface by which the interaction with the editor is encapsulated:

```
class Ord (ParseErrorOf p) => Parseable p where
    type ParserOf p :: * -> *
    type ParseErrorOf p :: *
    programParser :: (ParserOf p) p
    runParser :: (ParserOf p) p
        -> CstrSite p
        -> Either (NonEmpty (ParseErrorOf p)) p
    errorOffset :: Lens' (ParseErrorOf p) Int
```

We use the `TypeFamilies` language extension to associate two type families with the class: `ParserOf` and `ParseErrorOf`. These type families allow the implementer to choose the parser type and the datatype to represent parse errors. These types are parametrized by the type variable `p` introduced by the type class. This variable should be instantiated with the type of (the root of) the AST, i.e. `Expr` in the running example.

Besides a function (`runParser`) to run a parser for some node `n`, instances of the class should provide a parser for a complete program (`programParser`) and a lens[4] for adjusting a parse error's location. We also require the datatype representing parse errors to be an instance of the **Ord** class for deduplication using sets.

[4] https://hackage.haskell.org/package/lens

This class is generalized slightly further in appendix A to account for ASTs with multiple types of nodes. The reader is referred to Frugel's source code for an implementation using parser combinators.

As mentioned in the previous section, multiple variations of root construction sites are generated by partial linearization, representing different combinations of nested construction sites. From these variations, one that is parsed successfully should be picked and parse errors should be collected. Due to time constraints, only a simple suboptimal approach was implemented.

First, the root construction site variations are sorted and grouped into "buckets" with the same number of nested construction sites. These buckets are then considered in order of least remaining nested construction sites. If only a single variation in a bucket is accepted by the parser, no other buckets will be considered. However, if multiple variations are accepted, this indicates an ambiguity in how construction sites can be resolved. In this case, we act conservatively and simply move on to the next bucket with more remaining construction sites.

If this process does not produce a new AST, the original root construction site (with all nested construction sites intact) is used to continue the operation. Using this original construction site or the new AST produced by a successful parse, the parsing (and partial linearization) process is run recursively for all remaining nested construction sites, so any more deeply nested construction sites may be resolved as well.

This top-down order of processing construction sites is unnecessarily inefficient for context-free grammars. Due to their context-independence, processing construction sites in a bottom-up order should produce correct results as well. However, almost all real-world languages do have a form of context-dependence: lookahead. A bottom-up order of processing construction sites would still be possible by adding as much context to the parser input as required, but this would require some clever engineering that did not fit in our time constraints.

Parse errors from all attempted parses are collected and deduplicated. Extensive testing to show whether this is a good approach or not is left to future work.

5.6 Limitations

While Frugel improves on error-correcting parsers and strict structure editors overall, its approach is still imperfect and in some cases inferior to some error-correcting parsers. Consider the expression $(f \; (1 \; + 2|$. In the system described so far, if the programmer enters a closing parenthesis, the result will be $(f \; (1 \; + 2)|$. The new parenthesis is matched with the parenthesis in the outer construction site, but it is not clear that this is the intention of the programmer.

This behaviour does not fit Frugel's philosophy: it cannot know the intention of the programmer, so it should act conservatively in the face of ambiguity.

While we have not yet found a way to resolve this issue generally, we can force the intended behaviour in some cases: if we have a construction site that is the only item in another construction site, the inner construction site's contents can be spliced into place in the outer construction site.

This would transform $(f \; (1 \; +2$ into $(f \; (1 +2$, but not $f \; (\; \lambda x = x+ \; g \; y$ into $f \; (\lambda x = x+ \; g \; y$. This does discard some information on what parts of the expression contain syntax errors, but no information from complete nodes is discarded. Still, some ambiguous cases are still not handled conservatively.

With this operation added after the parsing step, entering a closing parenthesis in $(f\ (1 + 2$ would result in $(f\ (1 + 2)$ because all errors contained in a construction site (or any parent construction sites) need to be resolved before the construction site can be removed. This might confuse programmers initially, but fortunately the rule is relatively simple.

This new case, however, also has issues. Suppose that the programming language also allows for curly braces instead of parentheses. If we have the expression $(f\ \{1 + 2$, entering a curly brace will still introduce a new construction site, i.e. result in $(f\ \{1 + 2\}$.

Note that even without the splicing operation described above, this case would be problematic, since the outer construction site still has an unmatched parenthesis which prevents the removal of the inner construction site with the curly brace.

This is indeed a case where Frugel's approach is inferior to an error-correcting parser. We believe there are ways that this issue could be resolved, but we have not been able to spend a significant amount of time on it yet.

5.7 Other implementation decisions

Frugel is implemented in Haskell using the Miso[5] framework. When considering an implementation framework, the following properties were considered:

- Allows for writing in a functional language with a strong type system.
- Level of integration in a development environment, e.g. auto-complete and automated analysis (type errors and type information).
- Support for step-debugging with breakpoints.
- Allows for an idiomatic way of creating GUIs.
- Portability of interface. Ideally, it would compile to the web, which makes it easy to share and present.

Three candidates were given extensive consideration: iTasks on Clean[16][17] or Haskell on either Reflex[6] or Miso. After large parts of the implementation were finished, a fourth candidate called Schpadoinkle[7] was discovered. However, rewriting the implementation for his framework was not deemed worth the potential benefits.

From the original trio, iTasks was dismissed first due to low level of support in development environments and lack of step-debugging with breakpoints. Miso and Reflex mainly differ in proposed application architecture: Miso is a faithful implementation of The Elm Architecture (TEA)[4], while Reflex proposes that the programmer designs their own architecture relying on Functional Reactive Programming (FRP).

With TEA, user input is handled through updates to a model that constitutes the complete state of the application. The "view" that is presented to the user can then be derived from this model by relatively straight-forward functions. This architecture's strength is in its simplicity, but

[5]https://github.com/dmjio/miso
[6]https://reflex-frp.org/
[7]https://shpadoinkle.org/

it does have the downside that code pertaining to even relatively simple stateful components like a counter is spread across at least three places.

Functional Reactive Programming was originally formulated in 1997 [5] by Conal Elliott and Paul Hudak. Since its introduction, it has taken many forms and interpretations.

Without going into too much detail, FRP in the context of Reflex is all about combining functional programming with data that includes a dimension of time. For example, instead of setting an event handler on a button, a button returns the stream of events it generates. In this case, the dimension of time is discrete (in the same way that integers are).

Mouse position on the other hand, has a value for every point in time (at least theoretically) and its time dimension is therefore continuous. Data with a continuous dimension of time is also called a "behaviour".

Event streams and behaviours can also be combined into a "Dynamic", which has the same continuous time dimension as a behaviour, but also provides a way to handle changes at discrete points in time.

FRP allows for more cohesion in UI components by handling user input locally. This greatly improves their composability because there is no global model or type representing updates that needs to be updated when a new component is added. Therefore, FRP is especially suited for GUIs with many isolated components. However, handling this temporal dimension complicates component definitions. In the case of Reflex specifically, recursive `do`-blocks are required. This is an extension to Haskell's do-notation which allows for recursive bindings (similar to **let**). This can be problematic because it makes it easy for non-terminating recursion to occur. As the documentation says: "Basically for doing anything useful one has to introduce a feedback {loop} in the event propagation graph. And often this can lead to either an infinite loop or a deadlock."[3]

This feedback propagation only occurs once in Miso (though updates of the global application model) and it is shown to work properly in this instance. Our prototype has relatively few components, so the benefits of component cohesion are limited, but so is the drawback of using recursive `do`-blocks. Ultimately, Miso was chosen because of its simplicity. Hazel has also demonstrated that TEA can work well for structural editors[8].

5.7.1 Cursor and whitespace

Hazel uses Huet's zipper[10] for representing an AST with a cursor[15]. This solution does not work well for Frugel because we would lose the cursor location when parsing a construction site. Additionally, implementing text selection is more difficult because you need two cursor positions.

Instead, the cursor location is represented by its integer offset in the program's textual representation. This requires a method for finding and modifying an AST node based on such an offset. One option would be to attach source locations (start and end line and column number) to AST nodes.

However, to provide an editor interface that feels as simple as that of a standard text editor, the programmer should be allowed to manage whitespace themselves. This means separating code with spaces, tabs or newlines as they wish and applying pretty printing whenever they wish. With the attached source location model, character insertion would amount to "moving over" all the following AST nodes by changing their source locations and taking these source locations into account when rendering the code.

[8] https://github.com/hazelgrove/hazel/

This would not only be inefficient, but also needlessly complicated when whitespace is not discarded by the parser. When whitespace is not discarded, the AST becomes *lossless*: the complete source text can be reproduced from just the AST. If the whitespace is saved in the AST, whitespace inserted in the text can simply be inserted in the corresponding place in the AST. We can also find AST nodes by their source location based on this "complete view" of the source text.

The problem with this approach is that it "clutters" the AST with text details, while an AST should just represent a structured view of the program. This same problem occurs when representing parentheses in the AST. Parentheses are used to denote binding precedence of operators, but are redundant once the expression is parsed. To solve both problems (and some others regarding other parts of the programming environment) all AST nodes have a metadata field that contains such "text-only" information.

Saving whitespace in AST node metadata begs the question: which whitespace belongs to which AST node. When discarding whitespace, grammar terminals usually consume all following whitespace, but this approach is not ideal for Frugel because it leads to larger construction sites. For example, you could get $\lambda\quad y$. This is not ideal, because we prefer to keep construction sites as small as possible.

Instead of trailing whitespace, *interstitial* whitespace is saved. Interstitial whitespace is whitespace occurring between the children of an AST node. Concretely, we keep a list of whitespace fragments (strings containing only whitespace) for each node. For example, the expression

```
1     +
1
```

would have `["     ", "\n"]` as interstitial whitespace list. In the previous example, we would have a single fragment belonging to the application node, which makes it easy to display it as $\lambda\quad y$.

Manually inserting whitespace non-terminals in the parser definition would be error-prone and tedious. This can usually be alleviated by including whitespace consumption in the primitives used in the parser definition. In the case of interstitial whitespace, the primitives that need to be modified are the basic parser combinators (⟨$⟩ and ⟨*⟩ in the case of Haskell parser combinators). Unfortunately, this does mean most higher level parser combinators must also be adapted to use these primitives.

Chapter 6

Error-tolerant IDE services and live programming support

Based on the ASTs produced by the structure editor from the previous chapter, IDE services can assist the programmer. These services can be available without interruption, because there is always an AST to work from, but to make them error-tolerant requires additional effort.

In this chapter, we present two error-tolerant IDE services: a formatter (a function that rearranges the whitespace in the program in order to make it easier to read) and an interpreter that provides information about the program's runtime behaviour. This is far from a full-featured IDE, but especially the second service suggests that most IDE services can be adapted to become live and error-tolerant.

We emphasize that the goal is not to guess what the user intends to happen in the event of an error. Instead, the goal is to make the IDE services perform their tasks correctly for the correct parts of the program and incorporate in the incorrect parts in the result in a reasonable way. Thus, the utility of the result depends on how much it depends on the correctness of the erroneous part of the document.

In contrast to the structure editor technology detailed in the previous chapter, these services are largely language-specific by nature. However, there are still some parts that can be implemented generically, especially those related to the service is integrated into the programming environment. The details of this will be discussed separately.

6.1 Formatting

Whitespace plays an important role in any document that is intended to be read by humans. When an AST is available at all times, automatically manage whitespace based on the structure of the tree, but we choose not to do this because it can lead to sudden shifts in the layout of the document during editing.

Still, manually managing whitespace is boring and repetitive work, so instead, Frugel allows the programmer to run the formatter with a key-combination (like in many popular programming environments). However, due to their isolation in construction sites, syntax errors do not affect the formatter's behaviour outside the AST node containing the error. To the best of our knowledge,

this is a unique feature.

We give an example below:

```
(\argument
    = argument    ok
    theArgument
```

Figure 6.1: Input program

```
(\argument
    = argument ok
    theArgument
```

Figure 6.2: Formatted program

The formatter produces the program on the right from the one on the left. Despite the missing counterpart of the first parenthesis, `theArgument` is only indented once because it is an argument to the construction site. It would be indented twice if it was parsed as a second argument to `argument`, like an error-tolerant parser might do. Our formatter does not have to guess where the closing parenthesis is meant to go, because there is a structured document to work from. Despite the construction site, the excessive whitespace in the complete node is removed.

The appropriate whitespace is based on the kind of node.

6.1.1 Formatter implementation

Usually, formatters output plain text. This would discard the construction sites, including information on the origin of syntax errors, so an error-tolerant formatter needs a different approach.

We need to perform a process similar to the partial linearization and reparsing steps detailed in section 5.6 and 5.5, but instead of linearizing to various construction sites (where the inlining of nested construction sites is varied), the formatter linearizes to a single construction site where no nested construction sites are inlined. In other words, the degree of linearization performed by the formatter is the maximum degree that allows all complete nodes to be recovered.

However, if the AST is only partially linearized, we need a layout algorithm (an algorithm that determines where whitespace should be inserted or removed) that produces a tree as output instead of plain text. Furthermore, we would like this algorithm to be reusable, because ensuring a line does not exceed a certain length works the same way for all languages. Fortunately, such algorithms already exist and are implemented by the *prettyprinter*[1] package.

We can now describe the concrete steps performed by the formatter:

1. Instead of using node decomposition to linearize nodes, the formatter uses a function that converts AST nodes to a language-agnostic tree structure specific to the layout algorithms from the *prettyprinter* package. While this structure is almost completely linearized by the layout algorithms, it does allow some structure to be preserved through annotations. These are what we use to limit linearization.

2. It runs one of the layout algorithms from the *prettyprinter* package to obtain a formatted and partially linearized tree from the tree from the previous step. Through this process, complete nodes are linearized, but no construction sites are inlined.

[1] https://hackage.haskell.org/package/prettyprinter

3. The tree from the previous step is still specific to the *prettyprinter* package, so we need to convert back to the tree of construction sites that is required by the parser. In this tree, a construction site was either a construction site in the original AST or it does not contain a syntax error, so complete nodes can be recovered from it.

4. It reparses the construction sites in the tree, to recover the original AST with adjusted whitespace and a new list of syntax errors with updated locations. Because construction sites from the original AST are separated from those that resulted from the linearization of complete nodes, all complete nodes can be recovered.

Admittedly, this is more complicated than formatting usually is, but fortunately, the implementation of this approach can be reused for different languages. The only language-specific step in this process is step 1. Using the functions and combinators included in *prettyprinter*, the definition of this conversion is similar to what you would need for a traditional formatter.

6.2 Live programming support

As promised in the introduction, we now demonstrate how the internalization of construction sites into language semantics can enable live programming in the presence of errors. To this end, we develop a formal calculus that allows for evaluation to "continue around errors" on all levels of analysis (syntax, binding and types). This is implemented in an interpreter that also gathers runtime information that is relevant in the context of the current cursor position.

Since there is no I/O in the example language, there is no need to support pausing the interpreter when the program is edited. Instead, the interpreter is simply restarted and the effects of the code changes on the runtime behaviour of interest will become apparent when the in interpreter re-evaluates those parts of the program.

Because of the multitude of requirements and interactions between them, we develop the calculus and interpreter step by step. First, we discuss our general approach to evaluation and our motivations. Second, we take on the requirement of error-tolerance by defining corresponding notions of weak head and full normal form and an interpretation function. Third, we add the requirement of serializable and comprehensible normalization results, which requires post-processing of the results of the interpretation function and variable name disambiguation. This step also completes our definition of the normalization function. The collection of runtime information is introduced in the fourth step. To this end, we extend the interpretation function from the second stage and introduce a simple dynamic type system, which we elaborate on in a subsequent interlude. Fifth, we adapt the post-processing step from the third stage to the new interpretation function from the fourth stage and incorporate non-termination isolation technique. In the sixth and final step, we tackle non-termination issues more directly by developing an alternate mode of evaluation that is limited to a configurable number of reductions. As a bonus, this provides a crude method for exploring the evaluation process of any expression. Finally, we discuss a few noteworthy engineering challenges related to maintaining responsiveness of the programming environment.

6.2.1 General approach to evaluation

There are many approaches to evaluation in λ-calculus. Aside from supporting the functional requirements we discuss in the following sections, we also need to consider ease of implementation. Ideally, the chosen approach would be computationally efficient as well.

In the end, we developed an approach based on normalization by evaluation for untyped λ-calculus [6]. In this approach, normal forms of expressions in the object-language (the example language established in section 3) are obtained through their denotational semantics in a meta-language (Haskell in our implementation; mathematics in this report). A denotational semantics formally describes the meaning of expressions in the object language in terms of the meta-language. Most importantly, functions and function application in the object language are described by functions and function application in the meta language, respectively. Usually, this involves defining a new representation for object-language expressions in addition to the syntactical one. This additional semantic representation is called the *denotation* of an expression.

Formally, we would need to take a domain-theoretic approach with mathematics as our meta-language. This would amount to accounting for non-termination with $\perp$, but we think this would do more harm than good here, because it does not affect results observed by the programmer and we do not formally prove any properties of the semantics. We refer the reader to Filinski and Rohde's article [6] for a proper domain theoretic approach to a denotational semantics for untyped λ-calculus.

Our approach has the following qualities:

1. Most importantly, by combining this approach with Haskell, the object language inherits its call-by-need semantics, at least with the simple version described in section 6.2.2. When we add runtime information collection in section 6.2.4, we need to fall back to big-step dynamic semantics to implement sharing. The laziness of these semantics is important because expressions without normal form occur more often in the live setting of our programming environment. This laziness enables us to provide useful runtime information to the user in many of those cases nevertheless. The reasons for this are discussed in detail in section 6.2.7. Because laziness plays such a important role for the usability of the programming environment, we also annotate it in the formal specification of the approach. As a bonus, these semantics make the interpreter relatively efficient as well.

2. With a little modification, it supports the preservation of variable names, which makes the results presented to the user more comprehensible.

3. Instead of variable-capture-avoiding substitution, we only need to perform variable disambiguation after normalization. Variable capture happens when a free variable in one expression becomes bound in an expression it is substituted into. Performing this after the expression is reduced is both more efficient and simpler.

4. The lack of substitution makes it relatively efficient.

Countering these qualities are three drawbacks:

1. Usually, this approach is only used to obtain full normal forms, but this causes unnecessary cases of non-termination when the programmer is only interested in part of the result. In our most significant modification to the approach from Filinski and Rohde's article [6], we incorporate deferred computations explicitly in the denotation and add a third representation for object-language expressions: a partially reified representation. This modification will be discussed in more detail in section 6.2.3.

2. The fact that this approach requires additional representations is a drawback on its own because it requires either duplicating definitions and code or merging the various implementation into a single complex one. We apply the former in the report and the latter in the implementation.

3. Handling the results obtained with this approach requires more care than those obtained with other approaches because non-termination can be hidden deep in the structure of the result. Notably, normal forms in the object-language are translated to normal forms in the meta-language, so if an object-language expressions does not have a normal form, the corresponding denotation does not have one either. However, we cannot have the first quality without this drawback.

Whether this is the optimal approach to achieve our goals is difficult to determine, because the interactions between requirements, qualities and drawbacks are often not immediately obvious. However, even after the implementation, we have not found an approach that is obviously better.

Normalization by evaluation consists of two steps: first, interpretation and second, reification. The interpretation step converts an expression in the syntactical representation to its denotational one and the reification step converts it back. The first step will be described in the next section and the second in section 6.2.3.

6.2.2 Error-tolerant interpretation

Before the interpretation step can be defined, the denotation of expressions from the example language and a formalized notion of weak head normal form (WHNF) for the example language are needed. Because the denotation incorporates deferred computations based on the definition of WHNF, we start with the latter.

We naturally extend the standard notion of β-redexes to general redexes that include reducible addition terms. The reduction for addition terms simply adds the two numbers from its subexpressions together to produce a new number term.

The addition of numbers and construction adds two new kinds of non-redex application terms. Furthermore, the addition term can be irreducible when one of its sub-expression is not a number term. Combined, we get the following definition for WHNFs:

Definition 6.2.1 (Weak head normal form)**.** A term is in weak head normal if it is of one of the forms:

1. x, a free variable
2. $\underline{n}$, a number
3. nc, a construction site
4. $\lambda x \doteq e$, an abstraction term
5. $e_l + e_r$ where either or both e_1 and e_2 are in WHNF, but not a number.
6. $e_f\ e_{arg}$, an application term where e_f is in WHNF, but not an abstraction term

Full normal form is defined as standard, i.e. a term e is in full normal form if and only if there are no redexes in e (it is irreducible).We start with the denotation of expressions:

The denotation incorporates deferred computations in the places where there are no constraints on the subexpressions of an expression in WHNF. We write $\overline{a}$ for a deferred computation of a and $force(\overline{a})$ for forced computation of a deferred value to WHNF. In the definitions below, these are mere annotations, but they determine the object-language calling semantics if the interpretation function is implemented in a strict meta-language. This is discussed in more detail at the end of this section. The denotation follows:

$$\begin{array}{llll} \mathsf{DExpr} & de & ::= & x \mid \underline{n} \mid \boldsymbol{\lambda} mv.me \mid de\ \overline{de} \mid de + de \mid \boxed{dcm} \\ \mathsf{DCsMt} & dcm & ::= & di^{*} \\ \mathsf{DCsIt} & di & ::= & c \mid \boxed{\overline{de}} \end{array} \tag{6.1}$$

Other than the deferred computations of expression in complete nodes and arguments in application terms, the main difference with the syntactic representation from section 3 is the abstraction term. Here, it is represented with a meta-language function (signified by the bold lambda $\boldsymbol{\lambda}$) of the type $\overline{\mathsf{DExpr}} \rightarrow \mathsf{DExpr}$. In the notation, mv and me range over meta-language variables and expressions, respectively.

What meta-language function is used in this representation depends on the operations performed on the expression. For example, given the meta-language function f as representation of an object-language function, the result of performing an operation g on f is simply the composition of f and g, i.e. $\boldsymbol{\lambda} x.g(fx)f$.

Normally, denotations for λ-expressions lack the application (and addition) terms because these turn into operations in the meta-language. However in our case, they cannot be omitted, because we need to give meaning to syntactically malformed and ill-typed expressions to be error-tolerant and such erroneous application and addition terms cannot always be reduced.

For disambiguation of object-language addition and meta-language addition, we will use $\boldsymbol{+}$ for the latter from this point on.

Now, we can define an interpretation function $[\![e]\!] : (V \rightarrow \overline{\mathsf{DExpr}}) \rightarrow \mathsf{DExpr}$ that interprets an expression e in the environment gives as the first argument:

$$\begin{array}{rll} [\![x]\!](\rho) & = force(\rho(x)) & \\ [\![\lambda x \doteq e_{body}]\!](\rho) & = \boldsymbol{\lambda}\overline{de_{arg}}.[\![e_{body}]\!](\rho[x \mapsto \overline{de_{arg}}]) & \\ [\![e_f\ e_{arg}]\!](\rho) & = \begin{cases} f(\overline{[\![e_{arg}]\!](\rho)}) & f = \boldsymbol{\lambda} mv.me \\ f\ \overline{[\![e_{arg}]\!](\rho)} & \text{otherwise} \end{cases} & \text{where } f = [\![e_f]\!](\rho) \\ [\![\underline{n}]\!](\rho) & = \underline{n} & \\ [\![e_{left} + e_{right}]\!](\rho) & = \begin{cases} \underline{n_1 \boldsymbol{+} n_2} & [\![e_{left}]\!](\rho) = \underline{n_1} \wedge [\![e_{right}]\!](\rho) = \underline{n_2} \\ [\![e_{left}]\!](\rho) + [\![e_{right}]\!](\rho) & \text{otherwise} \end{cases} & \\ [\![\boxed{c_0^0..c_{i_0}^0 \boxed{e_0}..\boxed{e_n} c_0^{n+1}..c_{i_{n+1}}^{n+1}}]\!](\rho) & = \boxed{c_0^0..c_{i_0}^0 \boxed{\overline{[\![e_0]\!](\rho)}}..\boxed{\overline{[\![e_n]\!](\rho)}} c_0^{n+1}..c_{i_{n+1}}^{n+1}} & \text{for } n \geq 0 \end{array} \tag{6.2}$$

There are a few cases of interest regarding error tolerance: (1) a failed match on the denotation of the expression in the function position in an application term, (2) a failed match in the denotation of addition and (3) construction sites. The first can occur when there is any kind of error that prevents the expression from being interpreted as a meta-language function. In such a case, the argument is nevertheless interpreted and is used to construct a new syntactic application together with f.

This way, interpretation (and therefore normalization) does not stop in the event of an error (as with exceptions) nor does the expression at which the error occurred absorb other values (as with `undefined`). Instead, the programmer gets a partial result that shows exactly where something went wrong and most importantly: the context in which something went wrong. With the other error handling approaches, the programmer would either have to simulate the computation in their head or start debugging and run and navigate the program again.

Additionally, the programmer can see whether other values in the context match their expectations and thus verify other parts of the program. This kind of feedback is remarkable when compared to other methods of verification because (1) the programmer did not have to expend any effort to receive it (as is sometimes the case with type systems), (2) it is very detailed in comparison to running the program as a whole and (3) it is probably more relevant in the current editing context than a whole-program result.

However, this can generate immense expressions when the error occurs far into normalization. At the moment, we mitigate this issue by limiting the rendering of expressions to a configurable depth (see also section 6.2.7), but more advanced methods of managing what part of an expression should be displayed need to be developed before this approach is useful for real-world programs (see section 8.4).

The second case works similarly. With a concrete function such as addition, it is easier to imagine verifying the unaffected part of the result (the sub-expression that could be interpreted as a number). For example, $x + \langle\text{some complicated expression}\rangle$ can be normalized to $x + 37$.

In the case of construction sites, all characters are preserved and all complete nodes are interpreted. This interpretation is deferred to prevent unnecessary cases of non-termination.

Deferring and forcing interpretation We will now further detail the distinction between deferred and forced interpretation of values. Depending on meta-language semantics, using the result of an application of the interpretation function $[\![e]\!](\rho)$ will either compute the result of this interpretation immediately (with strict meta-language semantics) or only when the result is needed for output to the outside world (with lazy meta-language semantics), e.g. displayed on the screen.

However, one may still implement lazy object-language semantics in a strict meta-language by replacing all deferred values with functions that take no arguments and produce the deferred values, e.g. `() => interpret(e, env)` in JavaScript. In turn, the $force$ function calls this function and thus the embedded interpretation function.

This way, object-language function arguments are still only evaluated to WHNF when they are required to compute the result of the program. In a lazy meta-language, all function calls are deferred implicitly, which makes these annotations superfluous.

6.2.3 Presenting normalized expressions

An essential part of live programming environments is the presentation of runtime information. However, expressions in the semantic representation from the previous section cannot be directly presented to the user, because they contain meta-language functions. The simplest solution would be to show a placeholder in the place of function terms. This is a fair option when the information that could be shown has become unrecognizable due to internal transformation (for example, in a compiled language). In our case, the only transformation that has taken place is the evaluation of the function body. The user has a mental model of the function body and can therefore recognize

and use it to reason about the program. Then, the main challenge is preserving the variable names from the syntax representation.

For this, we need to extend the denotation of function terms to a tuple containing the original name of the variable it binds (the x in $\lambda x \doteq e$) and the meta-language function that was already there:

$$\mathsf{DExpr} \quad de \quad ::= \quad \ldots \mid (x, \boldsymbol{\lambda} mv.me) \mid \ldots \tag{6.3}$$

In the interpretation function, these variable names are simply copied from the syntax representation in the lambda term case and they are discarded when a function is eliminated in the application term case.

When we preserve variable names, we also preserve the variable-capture problem. However, because the interpretation function is independent of them, it is sufficient to disambiguate names referring to different variables in the result of the interpretation function (instead of performing capture-avoiding substitution during interpretation). We incorporate this disambiguation in the function that converts expressions in the semantic representation back to the syntax representation.

This process is called reification because meta-language expressions that represent expressions from the object-language are reified back to a syntactical representation. With denotational semantics, reification is usually defined together with an inverse process called reflection in a mutually recursive manner. However, reflection is superfluous in our system, because the only case we need is the one for variables and our representation of variables is the same in both expression representations.

The reification function $\downarrow de : \mathsf{Expr}$ is defined as follows:

$$\begin{aligned}
\downarrow x \quad &= x \\
\downarrow \underline{(x, f)} \quad &= \lambda y \doteq \downarrow f(y) \quad \text{where } y = fresh(x) \\
\downarrow (de_f \ \overline{de_{arg}}) \quad &= \downarrow de_f \ \downarrow force(de_{arg}) \\
\downarrow \underline{n} \quad &= \underline{n} \\
\downarrow (e_{left} + e_{right}) \quad &= \downarrow e_{left} + \downarrow e_{right} \\
\downarrow \boxed{c_0^0 .. c_{i_0}^0 \boxed{\overline{e_0}} .. \boxed{\overline{e_n}} c_0^{n+1} .. c_{i_{n+1}}^{n+1}} \quad &= \boxed{c_0^0 .. c_{i_0}^0 \boxed{\downarrow force(e_0)} .. \boxed{\downarrow force(e_n)} c_0^{n+1} .. c_{i_{n+1}}^{n+1}} \quad \text{for } n \geq 0
\end{aligned} \tag{6.4}$$

This function uses $fresh$ to obtain a fresh variable name based on a given variable name for the disambiguation mentioned before. We omit a formal definition of this function because normalization is independent of its exact workings. The approach we took requires the distribution of a variable name environment through the reification function and appends numbers to variable names based on this environment. This environment is initialized with the set of free variables from the original program (before normalization) to prevent name collisions with them as well.

This function also completes the full definition of normalization: $norm(e) = \downarrow [\![e]\!](empty))$ where *empty* maps all variable names to variables with the given name (so free variables are mapped to themselves).

6.2.4 Collecting runtime information

In this section, we add the collection of runtime information to the interpretation function from section 6.2.2. This runtime information is added to the result of the function, but the normalized expressions it returns stay the same. The interpreter collects two kinds of runtime information:

encountered errors and values relevant at the cursor. We extend the interpretation function to include these in the return value: $[\![e]\!](\rho) : (V \to \overline{\mathsf{DExpr}}) \to (\mathsf{DExpr}, \mathsf{RInf})$. We use the following notation for runtime information:

$$\begin{array}{llll} \mathsf{RInf} & ri & ::= & err \mid rci \mid ri \cup ri \mid \epsilon \\ \mathsf{EvEr} & err & ::= & \tau \not\Leftarrow de \mid x \not\twoheadrightarrow \\ \mathsf{Type} & \tau & ::= & \tau_{\to} \mid \tau_{num} \\ \mathsf{RecI} & rci & ::= & \rho \vdash de \end{array} \tag{6.5}$$

Runtime information RInf is either an error or a recorded interpretation, a composition of other runtime information or an empty runtime information collection. An interpretation error err can be a type error $\tau \not\Leftarrow de$ or a free variable error $x \not\twoheadrightarrow$. The first of these is used when an expression de is encountered of which the form does not match the expected type τ. For example, if the interpretation of e_f in the interpretation of $e_f\ e_{arg}$ results in an expression of the form $e_1 + e_2$, it is trivial to see that this does not match the type for functions, $\tau_{\to}$. In this case, the type error $\tau_{\to} \not\Leftarrow e_1 + e_2$ will be produced. This kind of dynamic type analysis is formalized in section 6.2.5.

Currently, the set of values relevant at the cursor is chosen to be the value of the smallest expression the cursor is on (de) (determined with the same process that is used to determine what node should be decomposed by an edit) and all variables that are in scope at the cursor (ρ). We call a tuple of these values a *recorded interpretation*, denoted $\rho \vdash de$.

The interpreter records the interpretation of an expression every time it encounters the expression with the cursor on it. For instance, if the cursor is in the body of the identity function $i = \lambda x \doteq |x$, interpreting the expression $i\ i\ a$ will produce the recorded interpretations $x \mapsto \lambda x \doteq x \vdash \lambda x \doteq x$ and $x \mapsto a \vdash a$.

Sadly, runtime information collection has complicating interactions with other properties of our interpreter:

1. The expressions in semantic representation in the type errors and recorded interpretations need to be shown to the user as well. Therefore, they also need to be reified and have variables renamed for disambiguation. This is no problem for environment-less disambiguation as described before, but if disambiguation does depend on an environment, this environment needs to be available when the error is produced or the interpretation is recorded. This complication will be discussed in further detail in section 6.2.7.

2. Normalizing the expressions in type errors and recorded interpretations could evaluate parts that would not be evaluated otherwise. This extra evaluation may take a noticeably long time, or it may never even terminate. However, the user is often only interested in specific parts of these values, so it is sufficient to perform this extra evaluation on-demand. The details of this will be discussed in section 6.2.7 as well.

3. If we keep translating sharing in the object-language to sharing in the host-language directly, runtime information will be duplicated when a value is shared, e.g. in $\lambda x \doteq x + x$. Because the evaluation of x (and therefore of the corresponding runtime information) is deferred, the runtime information is produced at both occurrences of x in the body of the function and then combined by the addition term. Hence, this issue is due to the lazy semantics of our object-language (the runtime information would be produced only once with strict semantics, specifically at the point when the function is applied to an argument). This duplication would fit a call-by-name semantics, but we think this is unintuitive. For instance, if we calculated

the factorial of 3 with the church-encoding for numerals and placed the cursor in the body, we would get 27 recorded evaluations instead of 4. To resolve this, we redefine sharing in big-step-operational-semantics-style using references in section 6.2.6.

6.2.5 Interlude: dynamic type checking

Technically, the untyped lambda calculus is a uni-typed calculus. A free variable may block reduction, but this is not a type error. Our object-language extends the untyped lambda calculus with numbers, which makes it bi-typed and therefore there can be type errors.

We define a rudimentary type system for our object language that is sufficient for reporting helpful type errors (using dynamic type checking), but not static type checking. Remarkably, much of the feedback a programmer could get from a static type checker can be provided by dynamic type checking during evaluation because the code can be evaluated continuously and evaluation continues past type errors without entering inconsistent states. However, type systems remain useful for proving properties for all branches in the program control flow. An excellent example has been developed by Cyrus et al. [15].

Currently, our approach only ensures an error when an application or addition term cannot be reduced, but it does not report on all the type errors that it could theoretically find, because this was not a priority for this thesis.

Definition 6.2.2 (Type matching)**.** Recall that $\tau_{\rightarrow}$ is the type of functions and τ_{num} the type of numbers. An expression de does *not* match the type $\tau_{\rightarrow}$ (written $\tau_{\rightarrow} \not\Leftarrow de$) if and only if is it of the form $\underline{n}$ or $de_l + de_r$. An expression does *not* match the type τ_{num} if and only if it is of the form $(x, \boldsymbol{\lambda} mv.me)$. All other expressions match both types.

We implement type matching with the function $match : \mathsf{DExpr} \rightarrow \mathsf{Type} \rightarrow \mathsf{RInf}$:

$$\begin{aligned} match(de, \tau) = {} & \text{if } (\tau = \tau_{\rightarrow} \wedge (de = \underline{n} \vee de = de_l + de_r)) \vee (\tau = \tau_{num} \wedge de = (x, \boldsymbol{\lambda} mv.me)) \\ & \text{then } \tau \not\Leftarrow de \\ & \text{else } \epsilon \end{aligned} \tag{6.6}$$

6.2.6 Explicit sharing with references

As discussed in section 6.2.4, we need to prevent duplication of collected runtime information due to sharing. We cannot simply pick one of the occurrences of a shared variable because the variables can occur in different branches of interpretation and we cannot always know ahead of time whether any branch will always be evaluated. We first attempted to deduplicate runtime information based on variable names, when an expression that joins interpretation branches (such as addition and application) is interpreted, but this approach turned out to be complex and error-prone. In our final approach, we thread a state $\sigma : \Sigma$ through the interpretation process that carries a counter, which we can use to create unique references $\phi : \Phi$. These references are used to create an additional level of indirection in the map from variables to values. The environment that is distributed across interpretation branches (ρ) maps from variable names to references, thus preserving standard scoping rules. The state that is threaded through the interpretation also carries the values (with optional runtime information) these references refer to. When a variable is evaluated, any associated runtime information is added to the interpretation output and removed from the state. Thus, subsequent evaluation will find no associated runtime information.

This approach solves the problem, but has a few drawbacks. Firstly, branches of interpretation that are conceptually independent, e.g. in $de_l + de_r$, become operationally dependent. Furthermore, it detracts from the initial beauty of our general approach to evaluation, because we need to manage sharing ourselves instead of inheriting it from the meta-language. Moreover, we now *need* to be explicit in the locations where laziness is required in the denotation of expressions. Specifically, instances of $\overline{de}$ in de are turned into references to partially applied instances of the interpretation function. These instances have already received an environment ρ, but not yet a state σ. We call these instances *interpretation closures* and they have the type $\Sigma \to (\mathsf{DExpr}, \mathsf{RInf}, \Sigma)$. This is necessary to preserve laziness of the object-language because σ will become undefined if the computation of de diverges. Incorporating these closures into the semantic representation also allows us to perform further evaluation of collected runtime information without restarting.

Additionally, the type of meta-language functions in the denotation of object-language functions is changed to $\Phi \to \Sigma \to (\mathsf{DExpr}, \mathsf{RInf}, \Sigma)$ where $\phi : \Phi$ now represents a reference to the argument of the object-language function.

In the redefinition of the interpretation function below, we assume the following operation on the state:

- $newState : \Sigma$, which creates an empty state.
- $store : (\Sigma \to (\mathsf{DExpr}, \mathsf{RInf}, \Sigma)) \to \Sigma \to (\Phi, \Sigma)$, which stores an interpretation closure in the given state and returns a reference to the closure.
- $retrieve : \Phi \to \Sigma \to (\mathsf{DExpr}, \mathsf{RInf}, \Sigma)$, which retrieves the interpretation closure associated with the given reference and replaces it with $\boldsymbol{\lambda}\sigma.(de, \epsilon, \sigma)$ where de is the expression the retrieved interpretation closure produces.

We redefine the interpretation function $[\![e]\!] : (V \to \Phi) \to \Sigma \to (\mathsf{DExpr}, \mathsf{RInf}, \Sigma)$ in equation 6.7 and discuss the various cases below.

In the case of variables, we check if the variable is in the environment using the domain (a.k.a. keys) in the map ρ. If it is in the environment, we *retrieve* the expression and runtime information associated with it. Otherwise, the variable is returned unchanged along with a free variable error.

The cases for abstraction terms and numbers work similarly to the initial version.

In the case of application terms, e_f is interpreted first (in the first auxiliary definition). The state σ_f this results in is used to *store* the interpretation closure of the argument ($[\![e_{arg}]\!](\rho)$ is partially applied; a shorthand for $\boldsymbol{\lambda}\sigma_{arg}.[\![e_{arg}]\!](\rho, \sigma_{arg})$). If e_f was interpreted to a meta-language function f, we obtain the interpreted body of the function and the associated runtime information by applying f to the reference and state that resulted from the *store* operation. Else, the application term is reconstructed with in_f and the reference to the interpretation closure of the argument. If *match* determines there is a type error, this is included in the result.

The case for addition term is quite verbose, but not very complicated. The left and right summands are interpreted in order in the first two auxiliary definitions. If both interpretations result in numbers, the sum of the numbers is returned along with the combined runtime information that resulted from the interpretations. Otherwise, the return value is the same, except for the addition term being reconstructed from the interpreted summands and any type errors are included in the returned errors.

In the case of construction sites, the interpretation closures of any complete nodes are stored in the state in order and the resulting references replace the corresponding complete nodes in the construction site.

$$\boxed{[\![e]\!] : (V \to \Phi) \to \Sigma \to (\mathsf{DExpr}, \mathsf{RInf}, \Sigma)}$$

$$
\begin{aligned}
[\![x]\!](\rho, \sigma) &= \begin{cases} retrieve(\rho(x), \sigma) & x \in dom(\rho) \\ (x, x \not\twoheadrightarrow, \sigma) & \text{otherwise} \end{cases} \\
[\![\lambda x \doteq e_{body}]\!](\rho, \sigma_1) &= ((x, \boldsymbol{\lambda}(\phi, \sigma_2).[\![e_{body}]\!](\rho[x \mapsto \phi], \sigma_2)), \sigma_1) \\
[\![e_f\ e_{arg}]\!](\rho, \sigma_1) &= \begin{cases} f(\phi_{arg}, \sigma_{store}) & \pi_1(in_f) = f = \boldsymbol{\lambda} mv.me \\ (in_f\ \phi_{arg}, match(f, \tau_{\to})\ rci_f, \sigma_{store}) & \text{otherwise} \end{cases} \\
&\quad \text{where} \\
&\qquad (in_f, ri_f, \sigma_f) = [\![e_f]\!](\rho, \sigma_1) \\
&\qquad (\phi_{arg}, \sigma_{store}) = store([\![e_{arg}]\!](\rho), \sigma_f) \\
[\![\underline{n}]\!](\rho, \sigma) &= (\underline{n}, \epsilon, \sigma) \\
[\![e_{left} + e_{right}]\!](\rho, \sigma) &= \begin{cases} (\underline{n_1 + n_2}, ri_{left} \cup ri_{right}, \sigma_{right}) & in_{left} = \underline{n_1} \wedge in_{right} = \underline{n_2} \\ (in_{left} + in_{right}, ri_{all}, \sigma_{right}) & \text{otherwise} \end{cases} \\
&\quad \text{where} \\
&\qquad (in_{left}, ri_{left}, \sigma_{left}) = [\![e_{left}]\!](\rho, \sigma_1) \\
&\qquad (in_{right}, ri_{right}, \sigma_{right}) = [\![e_{right}]\!](\rho, \sigma_{left}) \\
&\qquad ri_{all} = ri_{left} \cup ri_{right} \cup match(in_{left}, \tau_{num}) \cup match(in_{right}, \tau_{num}) \\
[\![\boxed{c_0^0..c_{i_0}^0 \boxed{e_0}..\boxed{e_n} c_0^{n+1}..c_{i_{n+1}}^{n+1}}]\!](\rho, \sigma) &= (\boxed{c_0^0..c_{i_0}^0 \boxed{\phi_0}..\boxed{\phi_n} c_0^{n+1}..c_{i_{n+1}}^{n+1}}, \epsilon, \sigma_n) \quad \text{for} \quad \begin{matrix} n \geq 0 \\ 0 \leq l \leq n+1 \\ i_l \geq 0 \end{matrix} \\
&\quad \text{where} \\
&\qquad (\phi_0, \sigma_0) = store([\![e_0]\!](\rho), \sigma) \\
&\qquad (\phi_k, \sigma_k) = store([\![e_k]\!](\rho), \sigma_{k-1}) \quad \text{for } 1 \leq k \leq n
\end{aligned}
\tag{6.7}
$$

We left out the recording of interpretations from this equation to reduce clutter and because it happens in the same way for all cases. The expression is first interpreted using the equations above and if the cursor was on the given expression, its denotation and environment are added to the collection of recorded interpretations.

Another difference between this formalization and the implementation is that the implementation uses singleton construction sites to point out where in the resulting expression type errors were encountered. For example, $[\![\rho, \sigma]\!](0\ 1)$ would result in $\boxed{0}$ 1 (after reification) with the error that 0 could not be matched with $\tau_{\rightarrow}$. As mentioned before, these construction sites are merely used for annotations and may be substituted for a different method of annotation, as long as it clearly shows any nesting of the annotations (which red underlining usually does not).

6.2.7 Presenting runtime information

In this section, we show how the reification operation from section 6.2.3 needs to be adapted to account for the references and interpretation closures now included in the denotation and preserve laziness in the presentation of the collected runtime information.

Rendering of reified expressions is limited by a maximum depth because the user is often only interested in the part close to the root of the expression and rendering would diverge in the case of expressions without normal forms otherwise. However, to retrieve expressions and runtime information from the references in the expression returned from the interpretation function, we have to thread the final interpretation state through the reification operation as well. If all references in an expression need to be eliminated (as the type of $\downarrow$ demands), this imposes an evaluation order on the subexpressions that is "depth first". Consequently, reification may diverge due to reification of subexpressions far from the root expressions and therefore prevent the programming environment from showing other parts of the expression or even the root expression itself, despite the rendering depth limit. For example, if normalization of the expression in the first complete node in a construction site diverges, the construction site can never be rendered. If we define a non-tail-recursive function such as the Fibonacci sequence and apply it to a construction site, we get an infinitely large expression because evaluation continues after failed reductions and the recursive function applications will keep being substituted for instances of the function body. In this case, rendering would diverge at the term that adds the two previous numbers in the sequence together.

Instead, we want to enable arbitrary exploration of the reified expression, where there are only no results rendered when the computation of a specifically requested WHNF diverges. Because we could not manage to develop a proper user interface for this in the time constraints of this thesis, we only allow the user to specify the rendering depth and develop a reification approach that can be generalized for arbitrary exploration.

Our solution is to enable partial reification of an expression. This requires a third representation for expressions where the representation is allowed to "switch" at any sub-expression. The reification function can then be modified to reify expressions up to a specified depth and an additional reification function that continues reification for a partially reified expression can be defined.

We give the notation for the partially reified representation below:

$$\begin{array}{llll} \mathsf{RExpr} & re & ::= & x \mid \underline{n} \mid (re) \mid \lambda x \doteq re \mid re\ re \mid re + re \mid \boxed{rcm} \mid \lceil de \rceil \\ \mathsf{RCsMt} & rcm & ::= & rit^{*} \\ \mathsf{RCsIt} & rit & ::= & c \mid \boxed{\mathsf{RExpr}} \end{array} \tag{6.8}$$

This representation is isomorphic to the syntax representation, except for the extra denotation

form. Using this form, the partially reified representation can switch to the semantic representation at any (sub-)expression.

The partial reification function $\downarrow_i de \ : \mathbb{N} \to \Sigma \to (\mathsf{RExpr}, \mathsf{RInf}, \Sigma)$ (with subscript i for initial) also collects runtime information from *retrieve*d references. In its definition, we use an auxiliary function $\downarrow_{ic} ic \ : \mathbb{N} \to \Sigma \to (\mathsf{RExpr}, \mathsf{RInf}, \Sigma)$ that combines the retrieval and reification of an interpretation closure. We defined these functions in equations 6.9 and 6.10, respectively.

In general, $\downarrow_i de$ goes into recursion on the subexpressions of de, threading the state left-to-right and collecting the resulting runtime information. However, the depth n is decremented with each recursive application of the function, until it reaches 0. In that case, we use the new form of the partially reified representation to switch to the semantic representation.

In the cases where interpretation closures occur in the argument denotation, these are reified with the function defined in equation 6.9. In the case of lambda terms, we first create a reference to a fresh variable (first auxiliary definition) and then obtain the reified body of the meta-language function by reifying the interpretation closure obtained by applying the meta-language function to the new reference (second auxiliary definition).

The additional reification function $\downarrow_c re \ : \mathbb{N} \to \Sigma \to (\mathsf{RExpr}, \mathsf{RInf}, \Sigma)$ that continues reification of an expression that was already partially reified, is very similar to the initial reification function, so we omit its formal definition. It works analogously in all cases, except those of the representation switch and lambda terms. In the case of lambda terms, it simply goes into recursion on the body. In the case of the representation switch, we use the initial reification function to continue reification.

Variable disambiguation in runtime information We will now discuss the first complication from the list from section 6.2.4. This approach of collecting runtime information complicates variable disambiguation if this depends on the variables in scope because the scope of expressions in type errors is not included in the type errors. In our case, it is not an option to start variable disambiguation with an empty scope for these expressions, because we need the set of free variables as an initial environment for disambiguation and we cannot obtain this from the semantic representation without reifying it.

Instead, we separate variable disambiguation from the reification step and move it to the interpretation step where the scope is naturally available. Disambiguation is then performed after every beta reduction, but only on the runtime information. Disambiguation on the main expression can still be performed during reification, which is more efficient.

We omit a redefinition of the interpretation function because we do not think this is an elegant solution and we do not see it as a core part of our approach.

6.2.8 Non-termination tolerance

Non-termination issues are some of the hardest to find the cause of because there is no error-message or output that can be used to reduce the number of places the error could be. In the previous section, we described a technique to isolate such issues, but this technique does not provide any insights into the issues themselves when a term lacks a WHNF.

To realize non-termination tolerance, we applied a second technique we call fuel-limited evaluation depth. The general technique is common in software engineering: we start interpretation with a configurable amount of "fuel" (represented by an intefer), which is "burned" with every reduction or recursive function application. When the fuel runs out, we return a failed application

$$\boxed{\downarrow_{ic} ic \ : \mathbb{N} \to \Sigma \to (\mathsf{RExpr}, \mathsf{RInf}, \Sigma)}$$

$$\begin{aligned}
(\downarrow_{ic} ic)(n, \sigma_1) \ &= (re, ri_{in} \cup ri_{re}, \sigma_{re}) \\
&\text{where} \\
&\quad (de, ri_{in}, \sigma_{in}) = ic(\sigma_1) \\
&\quad (re, ri_{re}, \sigma_{re}) = (\downarrow_i de)(n, \sigma_{in})
\end{aligned} \tag{6.9}$$

$$\boxed{\downarrow_i de \ : \mathbb{N} \to \Sigma \to (\mathsf{RExpr}, \mathsf{RInf}, \Sigma)}$$

$$\begin{aligned}
(\downarrow_i de)(0, \sigma) \ &= (\lceil de \rceil, \epsilon, \sigma) \\
(\downarrow_i x)(n, \sigma) \ &= (x, \epsilon, \sigma) \\
(\downarrow_i (x, f))(n, \sigma_1) \ &= (\lambda y \doteq re_{body}, ri, \sigma_{re}) \\
&\text{where} \\
&\quad (\phi_y, \sigma_{store}) = store(\boldsymbol{\lambda} \sigma_y.(fresh(x), \epsilon, \sigma_y), \sigma_1) \\
&\quad (re_{body}, ri, \sigma_{re}) = (\downarrow_{ic} (f(\phi_y)))(n-1, \sigma_{store}) \\
(\downarrow_i (de_f \ ic_{arg}))(n, \sigma_1) \ &= (re_f \ re_{arg}, ri_f \cup ri_{arg}, \sigma_{arg}) \\
&\text{where} \\
&\quad (re_f, ri_f, \sigma_f) = (\downarrow_i de_f)(n-1, \sigma_1) \\
&\quad (re_{arg}, ri_{arg}, \sigma_{arg}) = (\downarrow_{ic} ic_{arg})(n-1, \sigma_f) \\
(\downarrow_i \underline{n})(n, \sigma))(n, \sigma) \ &= (\underline{n}, \epsilon, \sigma) \\
(\downarrow_i (e_{left} + e_{right}))(n, \sigma) \ &= (re_{left} + re_{right}, ri_{left} \cup ri_{right}, \sigma_{right}) \\
&\text{where} \\
&\quad (re_{left}, ri_{left}, \sigma_{left}) = (\downarrow_i e_{left})(n-1, \sigma_1) \\
&\quad (re_{right}, ri_{right}, \sigma_{right}) = (\downarrow_i e_{right})(n-1, \sigma_{left}) \\
(\downarrow_i \boxed{c_0^0..c_{i_0}^0 \boxed{de_0}..\boxed{de_m} c_0^{m+1}..c_{i_{m+1}}^{m+1}})(n, \sigma) \ &= (\boxed{c_0^0..c_{i_0}^0 \boxed{re_0}..\boxed{re_m} c_0^{m+1}..c_{i_{m+1}}^{m+1}}, ri_{all}, \sigma_m) \quad \text{for} \quad \begin{array}{l} m \geq 0 \\ 0 \leq l \leq m+1 \\ i_l \geq 0 \end{array} \\
&\text{where} \\
&\quad (re_0, ri_0, \sigma_0) = (\downarrow_i de_0)(n-1, \sigma) \\
&\quad (re_k, ri_k, \sigma_k) = (\downarrow_i de_k)(n-1, \sigma_{k-1}) \quad \text{for } 1 \leq k \leq m \\
&\quad ri_{all} = ri_0 \cup .. \cup ri_m
\end{aligned} \tag{6.10}$$

as when type errors are encountered. This guarantees termination by limiting the only two sources of divergence.

However, instead of threading the number of fuel units remaining through the interpretation function like a state, we distribute it like an environment. This prevents any diverging computation from burning all the fuel and causing “false positive” out-of-fuel errors in expressions that are evaluated later.

Since this is a common and simple technique for guaranteeing the termination of algorithms in general, we omit a formal redefinition of the interpretation function. There is one complicating interaction between this technique, call-by-need semantics and recursive binders, which will be described later in this section.

This alternate mode of evaluation is especially useful in our case because it enables partial normal forms that can be used to find the source of the non-termination issue. Specifically, the failed reduction will point out the general location of the issue because the fuel will run out in the non-terminating loop (if there was enough fuel to reach it). Furthermore, the collected runtime information can be used to trace the control flow of the program and see what values may be causing the issue.

However, when the expression does have a WHNF, but no full normal form, the source of the issue is not pointed out as clearly. It would be only recognizable as a repeating pattern in the partial normalization result.

This technique supersedes the non-termination tolerance technique from the previous section, but enabling it permanently can be detrimental because there can be false positives with long-running programs. In our programming environment, we combine both modes of evaluation by running the limited variant in parallel if the unlimited one does not provide results within half a second (this is sufficient for small examples, but of course, this should be configurable in a mature programming environment). Therefore, the first technique remains useful.

Additionally, this second technique can be useful in more general cases than non-terminating computations. The programming environment has the option to start with limited evaluation by default, which provides a crude method for exploring the computation of any value. This makes the technique useful for generally gaining an understanding of unfamiliar programs and debugging programs that do not generate the desired results, but also no errors. The last example in chapter 4 shows an example of this use case.

As mentioned earlier, there is a complicating interaction between this technique, call-by-need semantics and recursive binders. Call-by-need semantics demands that when a value is computed, it is reused in all places it is referenced. Therefore, when (part of) the value of a recursive expression is computed, it should be reused in that expression.

At the same time, we limit recursion by burning one unit of fuel with every recursive application and when the fuel runs out, this should result in a failed application.

These two properties are in conflict because the latter implies a change in the value that is supposed to be constant, according to the first. Our current solution is to take the performance penalty for not sharing the value in recursive definitions when this alternate mode of evaluation is used. More complicated approaches to guaranteeing termination may allow for increased sharing, but we have not found the time to look into this.

6.2.9 Maintaining responsiveness

The previous sections have described the theoretical issues we encountered along with their solutions. Aside from these problems, we also encountered several engineering challenges which deserve to be discussed.

First, we have to ensure edit actions and evaluation can be interleaved because the environment would otherwise be unresponsive during evaluation (possibly forever if the evaluation does not terminate). We solve this by using separate threads (specifically, Haskell runtime system (RTS) threads) for edit actions and evaluation and allowing them to run concurrently. An edit action stops the current evaluation (if any) and (re)starts it with the updated program/changed configuration. To ensure the displayed evaluation results belong to the currently displayed program, we keep track of the number of edit actions in a version number and only display evaluation results when their source version number matches the current number.

Secondly, long-running evaluations can start to consume a significant amount of memory, which can slow down the entire computer or even lead the operating system to terminate the process. Ideally, the evaluation is performed as a separate process of which the memory usage can be monitored and which can be terminated separately from the programming environment. However, this requires a significant amount of platform-dependent code and is not possible when the environment runs in a web browser.

Instead, we keep everything in a single process and set a memory limit with Haskell's `-M` RTS argument. When we reach the limit, we catch the exception that gets thrown and stop the evaluation thread, which should free enough memory to continue operating.

Chapter 7

Related work

Structure editors and live programming have been the subject of research for many years now. In this chapter, we discuss the similarities and differences between our work and the related systems we have referred to throughout this thesis.

7.1 Grammar cells

The issues with strict structural editors are also recognized by Voelter et al.[24]. In response, they develop an approach for the development of projectional editors based on a new concept named grammar cells. They show that this approach yields a text-like editor experience similar to Frugel. Conceptually, the main difference is how editing in their system is directed by the specific syntax of the document, while in Frugel editing is directed only by the general tree structure of the document.

For example, their system would be aware that a newly entered `+` belongs to the language-specific notion of a binary expression and would modify the AST accordingly. Frugel, on the other hand, only regards it as a character and inserts it in the right place while preserving most of the AST. What this means for the structure of the document is left for the parser to decide.

In Voelter et al.'s approach, the edit actions emulating a text-editor experience are dependent on the type of grammar cell under the cursor, which demands special grammar cells for specific edit actions in cases like `splittable`. This grammar cell allows its child to be split in two, for example in the case of changing `11` into `1+1`.

Contrastingly, Frugel supports any edit action on any type of AST node by construction, thus eliminating the possibility of missing one.

Our editor is a more general system than theirs, because it allows for the integration of any kind of parser, while the language developer is limited to the parser features supported by the `rule` grammar cell in Voelter et al.'s approach. For example, look-ahead does not seem to be supported yet.

Furthermore, the grammar-cell-based editor is not generally error-tolerant. It solves the problem of inflexibility for the case of matching parentheses with a special `brackets` grammar cell, which allows for the occurrence of an unmatched bracket by storing it in an annotation node. This can be regarded as a specific instance of Frugel's construction site.

The largest advantage of their approach is the support for language composition. This is a result of how the programmer is forced to resolve ambiguities in the grammar when entering an expression.

Frugel has no such mechanism and simply relies on the provided parser to resolve ambiguities, but does integrate with existing text-based tools more easily because handling of plain text is already built into the system.

7.2 Proxima

Proxima [19] is “a presentation-oriented editor for structured documents”. In this case, presentation-oriented means supporting edit actions targeting the presentation of a document, i.e. text in the case of a source code editor.

Proxima employs the Utrecht University's attribute grammar system[1] to support error-correction during parsing. However, all editor functions depending on the structure of the document are disabled in the presence of a parse error because this system may generate spurious errors or perform unintended corrections.

7.3 Lamdu

Lamdu [12] is a live programming environment with very similar goals to Frugel. It contains a strict structure editor that is tuned to support left-to-right writing of code in Lamdu's internal programming language and inserts holes (a kind of placeholder) where necessary. In fact, Lamdu goes even further than syntactic correctness in its strictness: it attempts to keep ill-typed expressions from being constructed.

For example, variables with the wrong type appear below expressions with a correct type in the list of expressions from which the programmer must choose. Additionally, argument holes are inserted according to the function's type when we enter the name of a function where a number is expected. If the choice of expression to fill a hole does not match the expected type, the expression is placed in a “fragment”, which isolates the type error.

In our (admittedly limited) experience with the editor, this complicates code changes because it magnifies the tunnelling problem.

Lamdu supports type inference for and evaluation of incomplete programs. Holes are typed as free type variables and evaluation continues around holes used as operands in binary operations in a similar fashion as in Frugel. However, evaluation is not continued around hole arguments in application terms or under partially applied lambda terms and function values are not displayed. Additionally, evaluation exits with an exception if a certain stack size is exceeded. Together, these limitations avoid the presentation and non-termination issues we solved in sections 6.2.3, 6.2.7 and 6.2.8. Consequently, their system lacks some of the debugging assistance we describe in the later examples in chapter 4.

Type mismatches and evaluation results are displayed in a stack under expressions, where each layer spans the corresponding AST node. This “inline” view shows more runtime values than our approach (with a separate panel for the focused expression only), but takes up a lot of space with larger values.

If a function is applied multiple times, controls appear to cycle through the different evaluations. They may avoid the runtime information duplication problem we tackled in section 6.2.6 by using strict calling semantics in the object-language, but we failed to find any documentation that confirms this.

[1] https://hackage.haskell.org/package/uuagc

Remarkably, Lamdu supports I/O in the object-language by requiring explicit confirmation before I/O actions are executed. When the code is modified, the recorded runtime values disappear and the action needs to be confirmed again. This approach resolves the safety issues tied to automatically running I/O actions, but at the cost of liveness. Nonetheless, it may serve as a foundation for more advanced approaches that preserve liveness better.

7.4 Hazel

Hazel [15] [14] is the most similar programming environment to ours at the time of writing. It has been a significant inspiration for our work.

It is based on a formal structure editor calculus that also aims to preserve the interface of text editors. However, the structure editor also suffers from the tunnelling problem because it is strict and the editor calculus is specific to their internal language. We built our programming environment from scratch to see if these problems can be solved with a different foundation.

Notably, this formal specification supports the definition of various structure editor properties such as reachability (that is possible to reach any term in an expression through the defined movement actions), sensibility (that all expressions created through the editor actions can be typed) and continuity (that all typeable expressions can be evaluated to a kind of normal form). They also verify their calculus regarding these properties by providing mechanized proofs in Agda.

In further contrast to our work, Hazel includes a bidirectional type system that internalizes incomplete programs and type errors with empty and non-empty type holes. Initially, we wanted to diverge from this approach by implementing a unification-based type system, but this idea was abandoned due to a lack of time and the discovery that a live interpreter can provide much of the same feedback.

Hazel also automatically evaluates programs and makes evaluation continue around holes and inside ill-typed expressions. In other words: their evaluation process is tolerant of type errors and incomplete programs. It displays runtime values in a separate panel, but only for variables that are in scope at a selected hole (instead of at any selected expression), which limits the use of runtime values. Similarly to our recording of evaluations, they track *hole closures*. The duplication of this "collected runtime information" we encountered in our system is avoided by their choice of strict object-language calling semantics.

Much like Lamdu, their system does not continue evaluation under partially applied lambda terms, thus avoiding the presentation and non-termination issues we solve in 6.2.3, 6.2.7 and 6.2.8. However, their system completely freezes when expressions without normals form are encountered.

Remarkably, they also define a fill-and-resume operation that allows evaluation to continue from its previous result when a hole is filled, which may improve responsiveness of the programming environment significantly when working with larger programs.

7.5 Grounds for increased developer productivity in live programming environments

That live programming environments have been the subject of research since at least 1990 [21] proves there is some allure to them. Crowdfunding for a live programming environment called LiveTable received $316,720 from more than 7 thousand people in 2014 [8].

However, there have been almost no empirical studies that verify the predicted productivity gains. We are only aware of a study by Krämer et al. that found participants working in live programming environments fixed bugs faster and switched between debugging and editing more frequently than those working in the same environment with live programming features disabled. However, their study only had 10 participants.

We think this lack of empirical studies has two main reasons: (1) there is a lack of mature live programming environments and (2) productivity is hard to measure. We hope interest in this area increases and enough resources become available that these issues can be resolved.

Chapter 8

Future work

The programming environment we developed is a promising prototype, but we should be weary of the assumptions we made while designing it. There are many hurdles that still need to be overcome before it can become part of widespread development practices.

In this chapter, we discuss these assumptions and hurdles and what future work is required to validate or remove them.

8.1 Supporting I/O

One clear hurdle that needs to be overcome before our programming environment can be used for real-world programs is the support of I/O in the object-language. This is not trivial, because programs are run automatically and running I/O actions with unfinished parameters can have disastrous consequences.

We already mentioned Lamdu's solution to ask for confirmation from the programmers before running I/O operations, but this severely limits the use of live programming features. This could be mitigated by allowing individual operations or entire calls to I/O performing functions in external libraries to be permanently marked as safe. However, this option still carries risks because programs can behave in unexpected ways during development. This risks grow with the size of the project and the number of people working on it. Nonetheless, it may be the best solution for the prototyping phase of a project.

For more mature projects, a better solution would be to require and help the programmer define software container images that only provide the I/O operations required. A program can then run in this container in isolation from the rest of the system. This brings "the outside world" under control of the programming environment, which allows it to be easily reset in the event of unintentional consequences of I/O operations and improves reproducibility.

In fact, setting up test environments like this is already common software development practice, rising hand-in-hand with remote automated testing and continuous integration practices.

In summary, this approach requires an initial up-front investment and a lower continuous investment of time and effort, but the benefits are already well-established.

8.2 Performance on larger programs

Real-world programs are a lot larger than the ones that will probably be written in our programming environment. Until more extensive languages are supported, we cannot use any existing programs either.

Our programming environment might not perform well enough on two fronts: editing and evaluation. Currently, our structure editor will always be slower than a parser in an environment with a text editor because the structure editor reparses the entire program at least once after every edit. If there are construction sites in the program, the number of reparses increases exponentially, as discussed in section 5.4. Use of a GLR or GLL parser may mitigate this, but this would significantly reduce the number of existing parsers that can be used in our programming environment. Incremental parsers may also offer a sufficient performance improvement because there is a significant overlap in the contents of construction site variations.

Alternatively, the problem may be mitigated by employing a technique common in error-recovering parsers: recognizing non-terminals that delimit certain AST nodes. For example, by analysing a language's grammar, we may find that identifiers followed by an =-sign cannot occur in expressions. This syntax uniquely belongs to definitions, which provides an outer bound on what characters can be relevant to a definition with a syntax error. This allows for the AST to be "partitioned" in smaller sections that define limits on the effects of edits. Now, we only have to linearize and reparse the text in this section to find the effect of an edit. In most cases, these sections should be small enough that our approach is feasible. In this case, our structure editor can compete with incremental parsers regarding performance.

On the evaluation front, we foresee the issue that running the program to the runtime behaviour of interest simply takes too long, either because of computational intensity or blocking I/O actions. This could be mitigated by creating snapshots of the running program that can be used as starting points after edits. Another option may be to employ "editing closures" similar to Hazel's hole closures that allow evaluation to continue from a previously computed partial evaluation result.

We are aware that the possibility of performing live updates to running programs is also being researched (see for example the taxonomy by Giuffrida and Tanenbaum [7]), but we think the domain of programs this approach applies to is too narrow.

However, if it turns out this problem cannot be mitigated adequately, the approach to live programming we developed may still be useful in educational settings, where examples are naturally small.

8.3 Integration with mainstream programming environments and other text-oriented tools

Structure editors are notorious for being incompatible with traditional programming environments and other text-oriented tools. Since our approach by definition includes a parser, storing code as text and reparsing it after it has been modified by external tools is no problem, as long as there are no syntax errors. If there are any, they will simply have to be resolved before the IDE features can serve their purpose, just like in error-intolerant IDEs.

It would even be possible to integrate the structure editor with text oriented tools built into the IDE, by finding the bounds on the text modified and only linearizing and reparsing the affected AST nodes. However, the purely textual presentation of code in traditional IDEs may be a problem in

itself. Often, this presentation uses text field widgets that are built into the IDEs GUI framework or even the operating system itself. These widgets typically only allow for limited markup of the text and are incapable of showing nested and overlapping annotations like those we use for construction sites in a clear way.

We do not see these changing easily, but we think there is a way to generalize the traditional red underline that would also satisfy our requirements. If these widgets supported additional composable patterns in additional to the traditional zigzagging pattern, these patterns could be combined in the same line where annotations overlap. For example, the span of one construction site could be indicated by a line with regularly spaced circles and another with regularly spaced diamonds. Where the construction sites overlap, a line with both circles and diamonds would be displayed.

We think this would still be less clear than the presentation we used in our implementation, but the compromise could be worth it to increase adoption.

8.4 Improved interface for runtime values

There are also challenges remaining regarding interface design. The configurable rendering depth keeps the size of displayed values in check, but does little to help the programming find the parts of interest.

Developing an intuitive interface that supports interactive exploration of runtime values where irrelevant parts are elided automatically is a very challenging problem that deserves its own research project.

Furthermore, the exploration of computations instead of just the results, for which we achieved crude support through fuel-limited evaluation, deserves further development. Our system only allows very limited control of which reductions should be prevented. We think that allowing programmers to control this more precisely could make this into a valuable tool for debugging.

8.5 Improved explicit empty construction sites

The grammar defined for our object-language in section 5.5 allows manual insertion of empty construction sites with the ...-syntax. However, these construction sites disappear when a linearization variation is found where it is inlined and it does not cause a parse error, which is counter-intuitive.

This could be resolved by distinguishing construction sites inserted by the structure editor and those inserted using the syntax. The disappearance of manually inserted construction sites can then be prevented by only inlining the former in the linearization step.

However, this would also require that manually inserted construction sites can be removed using a "delete" edit action. Future work could look into how this problem is best resolved.

Chapter 9

Conclusion

The question that inspired this thesis is “are we live yet?”, an impatient inquiry into the lack of live programming features in mainstream software development practice. This question turned into: what are the remaining issues that prevent support for live programming from being adopted by mainstream development environments? With this thesis, we tackled the issues of limited domains and error intolerance and demonstrated the powerful synergy between error tolerance and live programming.

As a first step, we presented the concept of construction sites and give a detailed description of a structure editor that uses these to maintain a structured representation of code without sacrificing usability.

To show that this constitutes a sound technical basis for continuously available error-tolerant IDE services, we also developed a formatter and live programming support. We presented a formalization of the error-tolerant dynamic semantics underlying the latter and discuss fundamental issues regarding non-termination and the combination with a call-by-need reduction strategy. Additionally, we find that our solutions to the issues regarding non-termination can assist the programmer with finding the source of non-termination issues, which is notoriously difficult in traditional programming environments.

Together, this answers the precise research question:

Can a programming environment provide live programming support, and generally offer error-tolerant IDE services without interruption, regardless of present errors or application domain, based on (1) a structured representation of programs that incorporates construction sites and (2) formal semantics extended to give meaning to syntactically malformed and ill-typed programs?

As is also demonstrated by the examples in chapter 4, the answer is yes. However, our approach does not perform optimally when implemented with standard text fields and it is not yet clear that it is performant enough for real-world application. Additionally, the issues of safe I/O and the design of an intuitive interface for runtime values still pose a barrier for live programming environments in general.

Because of these issues, it is not yet certain that our approach generalizes to larger programs and real-world applications, but we do not think any of the issues pose an insurmountable challenge.

If our approach turns out to be infeasible for these applications, we hope our contributions may inspire future work that finds one that is.

Bibliography

[1] Mike Bostock and Melody Meckfessel. Observable - explore, analyze, and explain data. as a team. https://observablehq.com/.

[2] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It's alive! continuous feedback in ui programming. *SIGPLAN Not.*, 48(6):95–104, June 2013.

[3] Reflex contributors. Application development with reflex-dom. http://docs.reflex-frp.org/en/latest/app_devel_docs.html.

[4] Evan Czaplicki. The elm architecture · an introduction to elm. https://guide.elm-lang.org/architecture/.

[5] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[6] Andrzej Filinski and Henning Korsholm Rohde. A denotational account of untyped normalization by evaluation. *BRICS Report Series*, 10, 12 2003.

[7] Cristiano Giuffrida and Andrew S Tanenbaum. A taxonomy of live updates. *Proc. of the 16th ASCI Conf*, 2010.

[8] Chris Granger. Light table by chris granger — kickstarter. https://www.kickstarter.com/projects/ibdknox/light-table.

[9] Dave Griffiths and Gabor Papp. (fluxus). http://www.pawfal.org/fluxus/.

[10] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[11] Edgar T. Irons. An error-correcting parse algorithm. *Commun. ACM*, 6(11):669–673, November 1963.

[12] Eyal Lotem and Yair Chuchem. Lamdu. http://www.lamdu.org/.

[13] Sean McDirmid. "a live programming experience" by sean mcdirmid - youtube. https://www.youtube.com/watch?v=YLrdhFEAiqo.

[14] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live functional programming with typed holes. *PACMPL*, 3(POPL), 2019.

[15] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*, 2017.

[16] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pages 141–152, Freiburg, Germany, Oct 1–3 2007. ACM.

[17] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. Task-oriented programming in a pure functional language. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*, PPDP '12, page 195–206, New York, NY, USA, 2012. Association for Computing Machinery.

[18] Sérgio Queiroz de Medeiros, Gilney de Azevedo Alvez Junior, and Fabio Mascarenhas. Automatic syntax error reporting and recovery in parsing expression grammars. *Science of Computer Programming*, 187:102373, 2020.

[19] Martijn M Schrage. *Proxima - A presentation-oriented editor for structured documents*. PhD thesis, Utrecht University, 10 2004.

[20] Andrew Sorensen. Impromptu: An interactive programming environment for composition and performance, 2005.

[21] Steven L. Tanimoto. Viva: A visual language for image processing. *J. Vis. Lang. Comput.*, 1:127–139, 1990.

[22] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42:230–265, 1 1937.

[23] Michael L. Van De Vanter. Practical language-based editing for software engineers. In *Proceedings of the Workshop on Software Engineering and Human-Computer Interaction*, ICSE '94, page 251–267, Berlin, Heidelberg, 1994. Springer-Verlag.

[24] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards user-friendly projectional editors. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8706:41–61, 2014.

[25] Markus Voelter, Tamás Szabó, Sascha Lisson, Bernd Kolb, Sebastian Erdweg, and Thorsten Berger. Efficient development of consistent projectional editors using grammar cells. In *SLE 2016 - Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, co-located with SPLASH 2016*, pages 28–40. Association for Computing Machinery, Inc, 10 2016. Similar ideas to Hazel, but less formal and doesn't support proving formal properties.

[26] Young Seok Yoon and Brad A. Myers. A longitudinal study of programmers' backtracking. *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, pages 101–108, 2014.

Appendix A

Decomposition for ASTs with multiple node types

Allowing for multiple node types requires generalizing the edit loop significantly. We first define some types:

```
newtype CstrSite n = CstrSite [Either Char n]

data Node = ExprNode Expr | DeclNode Decl | WhereNode WhereClause

data Expr
    = Variable String
    | Abstraction String Expr
    | Application Expr Expr
    | Sum Expr Expr
    | ExprCstrSite CstrSite

data Decl = Decl String Expr | DeclCstrSite CstrSite

data WhereClause = WhereClause [Decl] | WhereCstrSite CstrSite

data Program = Program Expr WhereClause | ProgramCstrSite CstrSite
```

In this example, `Program` is the root of the AST. The purpose of `Node` is to collect all node types into a sum type for inclusion in construction sites, i.e. `CstrSite Node`.

To allow us to ensure all types of node in an AST have the same type of node (and therefore construction site), we introduce an open type family:

```
{-# LANGUAGE TypeFamilies #-}

type family NodeOf a :: *

type instance NodeOf (ACstrSite a) = a
type instance NodeOf Node = Node
```

```haskell
type instance NodeOf Identifier = Node
type instance NodeOf Expr = Node
type instance NodeOf Decl = Node
type instance NodeOf WhereClause = Node
```

Using this type family, we can now define a more general version of `Decomposable`:

```haskell
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE DefaultSignatures #-}

class (FromNode a, SetCstrSite a) => IsNode a

class FromNode a where
    fromNode :: a -> NodeOf a

class NodeOf n ~ NodeOf (NodeOf n) => Decomposable n where
    traverseComponents :: Applicative f
        => (Char -> f Char)
        -> (forall n'.
            (Decomposable n', IsNode n', NodeOf n ~ NodeOf n')
            => n'
            -> f n')
        -> n
        -> f n
    conservativelyDecompose :: Int -> n -> Maybe (Int, CstrSite (NodeOf n))
    default conservativelyDecompose :: FromNode n
        => Int
        -> n
        -> Maybe (Int, CstrSite (NodeOf n))
    conservativelyDecompose cstrSiteOffset n = case cstrSiteOffset of
        0 -> Just (0, singletonCstrSite)
        1 | 1 == length (toList $ decompose n) -> Just (1, singletonCstrSite)
        _ -> Nothing
        where
            singletonCstrSite = fromList [ Right $ fromNode n ]
```

First, due to the recursive nature of decomposition, applying `NodeOf` twice may not change the node type (`NodeOf n ~ NodeOf (NodeOf n)`). Otherwise, requiring an instance of some type class `C` for `NodeOf n` for some `n`, would require an infinite number of instances (`C (NodeOf n)`, `C (NodeOf (NodeOf n))`, etc...).

Secondly, the function gets a rank 2 type to allow for processing of any kind of node, while maintaining the node's type. We do require that these nodes are instances of `IsNode`, which simply requires an instance of `SetCstrSite` and `FromNode`, the latter of which allows for converting a node to its `NodeOf` type.

Finally, the `Decomposable` instance for `Program` overrides the default definition for

`conservativelyDecompose` to always return **Nothing** (i.e. `conservativelyDecompose _ _ = Nothing`.

The simple methods from `IsNode` are all that is required by the general part of the decomposition step (aside from the `Decomposable` instances, which stay the same):

```
modifyNodeAt ::
    (MonadError (InternalError p) m, Decomposable p, SetCstrSite p)
    => (Int -> CstrSite (NodeOf p) -> m (CstrSite (NodeOf p)))
    -> Int
    -> p
    -> m p
```

Note that we distinguish the type of the root of the AST this function is called on `p` and any node in the AST `n`. This is also due to `Program` not being an instance of `FromNode`.

Finally, we can show where the second argument of `SetCstrSite` is required. The `SetCstrSite` instance of `Node` needs to see from the old node which type of node it should create. The instance is given below:

```
instance SetCstrSite Node where
    setCstrSite cstrSite = \case
        ExprNode expr -> ExprNode $ setCstrSite cstrSite expr
        DeclNode expr -> DeclNode $ setCstrSite cstrSite expr
        WhereNode expr -> WhereNode $ setCstrSite cstrSite expr
```

The last place where the presence of multiple types of AST nodes needs to be taken into account is in the `Parseable` class. To parse nested construction sites of any type, we add the `anyNodeParser` method. This parser should succeed on any type of node. The complete class then looks as follows:

```
class Parseable p where
    type ParserOf p :: * -> *
    type ParseErrorOf p :: *
    programParser :: (ParserOf p) p
    anyNodeParser :: (ParserOf p) (NodeOf p)
    runParser :: (ParserOf p) n
        -> ACstrSite (NodeOf p)
        -> Either (NonEmpty (ParseErrorOf p)) n
    errorOffset :: Lens' (ParseErrorOf p) Int
```