Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelor's Thesis

IDE Support for Lexical Effects and Handlers

Tim Neumann

29.1.2022
Last modified 23.2.2022

Gutachter / Betreuer

Jun. Prof. Dr. Jonathan Brachthäuser
Eberhard Karls Universität Tübingen
Wilhelm-Schickard-Institut für Informatik
Lehrstuhl für Software Engineering

Neumann, Tim:
IDE Support for Lexical Effects and Handlers
Bachelorarbeit Informatik
Eberhard Karls Universität Tübingen
Bearbeitungszeitraum: 01.12.2021-31.03.2022

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Tim Neumann (Matrikelnummer 4137034), 29. Januar 2022

Abstract

Algebraic effects and handlers offer a new way to express computational effects in functional programming languages. Using the Effekt language and Microsoft Visual Studio Code, we discuss and partially implement ideas for novel IDE features to support the usage of effects and handlers. We conclude that research in the overlapping field of effects and handlers and user support through programming environment tools is sparse. Supportive features common for other language constructs may be adaptable to effects and handlers. To some users the proposed features may be helpful in reasoning about and writing code with effects and handlers. Further research is needed to ascertain the usefulness of our proposals.

Acknowledgments

My greatest thanks are due to Jun. Prof. Dr. Jonathan Brachthäuser, who not only suggested the topic of this thesis and supervised it, but also showed patience and understanding in what was a difficult time for me. Thank you and also Prof. Dr. Klaus Ostermann and M.Sc. Philipp Schuster for deepening my interest in programming languages.

As this thesis marks the provisional completion of my studies, I would like to thank the University of Tübingen and all people involved that made this study possible for me. Thank you to the department of computer science and all my professors, lecturers and tutors. Thank you to the examination office. In particular, a big thank you to Renate Hallmayer, who helped me and hundreds of other students navigate through the Prüfungsordnungen.

My heartfelt thanks to my girlfriend, my friends, my family and my girlfriend's family for supporting me throughout my studies and beyond!

Last but not least, I would like to thank my brother: I could not have done all this without you.

Contents

1 Introduction **1**
1.1 Effects and handlers . . . 1
1.1.1 Algebraic effects . . . 2
1.1.2 Effect handlers . . . 2
1.1.3 The Effekt language . . . 4
1.2 Language support in integrated development environments . . . 7
1.3 Related work . . . 8
1.4 Conclusion . . . 11

2 Implementation **13**
2.1 Technical details . . . 13
2.1.1 Visual Studio Code . . . 13
2.1.2 Language Server Protocol . . . 14
2.1.3 Previously implemented features . . . 15
2.2 Proposed features . . . 16
2.2.1 Implemented features . . . 16
2.2.2 Outlined features . . . 23
2.3 Summary . . . 29

3 Discussion **31**
3.1 Implemented IDE features . . . 31
3.1.1 Inferred type and effect decoration . . . 31
3.1.2 Effect origin and binder hint . . . 31
3.1.3 Automatic handler creation . . . 32
3.2 Outlined IDE features . . . 32
3.2.1 Insert type and effects . . . 32
3.2.2 Display effects as requirements and handled effects as arguments 33
3.2.3 List handlers of an effect . . . 34
3.3 Limitations . . . 34
3.4 Conclusion . . . 35

4 Bibliography **37**

1 Introduction

In the year of 2020 the History of Programming Languages (HOPL) counted 8945 different programming languages in existence [Pigott, 2020]. The multitude of programming languages is caused by a combination of different programming paradigms, use cases and requirements. The goal of most programming languages is to improve expressiveness in their field of application. In order to achieve improved expressiveness, many concepts have been invented and implemented in a wide range of languages. One famous and fundamental concept for example is that of data types: a data type is assigned to computational data in order to allow the computer programmer to express the intended use or interpretation of that data. As mentioned before, concepts like this derive from use cases or requirements. In the case of data types, the requirements could be type safety during program execution, expression of contracts and improved comprehensibility for developers. Concepts like data types may improve the way a computer programmer reasons about code. They may also allow an enriched user experience by providing additional information to the programming environment. For example, it is common for programming environments of typed languages to allow type specific syntax highlighting, highlighting of type mismatches or missing type information and more precise *jump-to-definition* and *code completion* features.

Another, rather young concept in programming languages is that of effects and handlers. Just as the concept of data types, the concept of effects and handlers addresses certain use cases and requirements and tries to increase the expressiveness of a programming language. And just as with data types, the concept of effects and handlers may be able to improve the way of reasoning about a computer program. Also the concept may benefit from advanced programming environment features to support its understanding and intuitive usage. In this thesis we want to discuss the latter.

1.1 Effects and handlers

In order to be able to talk about supportive programming environment features for effects and handlers, we first want to introduce algebraic effects and effect handlers. We will give a short introduction to the used language *Effekt*. Even though examples are given in the Effekt language, most concepts discussed here are not specific to Effekt but carry over to other languages that implement effects and handlers.

1.1.1 Algebraic effects

The underlying problem that spawned the concept of effects are so called computational (side) effects. A computational effect occurs whenever a function in a computer program has some effect besides the pure calculation of an output from its input. In other words, a function in a computer program has effects when it violates some basic characteristics of mathematical functions. Functions like these are called *impure*. Common examples of effects are the occurrence of exceptions, modification of mutable state, or the performance of input / output operations [Forster et al., 2017].

In purely functional programming languages, computational effects pose a special problem. On the one hand they may violate some core properties of functional programming, for example referential transparency. On the other hand, they may be necessary to implement real world tasks such as user input or realisation of stateful computations. Probably the most common way to model computational effects in functional languages are monads. These are based on Eugenio Moggi's expression of computational notions, including side effects, by means of category theoretical monads [Moggi, 1991].

In 2001, Plotkin and Power introduced a novel approach to model computational effects called *algebraic effects* [Plotkin and Power, 2001, Plotkin and Power, 2003]. In their work the authors picked up the idea that computational effects may only arise from a well defined set of operations. Therefore, these effects can be tracked as part of a computations type, expanding the classical type systems to type and effect systems. The usage of one such operation introduces the algebraic effect, hence they are also referred to as *effect constructors* [Plotkin and Pretnar, 2009]. In subsections 1.1.2 and 1.1.3 we will look at practical examples of effects in the Effekt language.

1.1.2 Effect handlers

The concept of handlers may be familiar to many programmers in the context of exceptions. An exception is meant to describe the occurrence of unintended or abnormal behavior. Thus an exception can be understood as a computational side effect. An exception handler offers the ability to catch and treat the occurrence of such an unexpected event during run time. Plotkin and Pretnar adapted the concept of exception handlers to their more general concept of arbitrary algebraic effects [Plotkin and Pretnar, 2009]. As stated above, their model of effects assures that an effect can only arise from certain operations, commonly called *effect operations*. However, these operations only offer trackable information on effects. They do not offer an actual implementation of the computational effect. Handlers may offer a dynamic interpretation of the effect operation. In other words, an algebraic effect and its effect operations act like an interface and handlers act as implementations of that interface. As effect operations are constructors of effects, handlers are *effect deconstructors*. Figure 1.1 shows an example program in the Effekt language that

incorporates an effect with an effect operation and a matching handler.

```
effect Logging {
  def logResult[A](s: A) : A
}

def add(a: Int, b: Int) : Int / { Logging } = {
  val res = a + b;
  logResult(res)
}

def main() = {
  try {
    add(23, 42)
  } with Logging {
    def logResult(s) = {
      println("Logging result: " ++ s.show);
      resume(s);
    }
  };
}
```

Figure 1.1: Example program defining, utilising and handling an effect. The code is written in the Effekt language.

On lines 1-3, the *Logging* effect is defined. Its only effect operation *logResult* is generic in some type *A*. It receives some value of type A, returns a value of type A, and gives rise to the *Logging* effect. Through the usage of the *logResult* effect operation on line 7, the *Logging* effect is introduced. Notice that the signature of *add* in line 5 denotes the *Logging* effect in its return type:

```
Int / { Logging }
```

On line 12 we call *add* with two arguments. However, the *Logging* effect has no implementation and thus the call to *logResult* in the *add* function would fail. Therefore we wrap the call in a

```
try { ... } with ... { ... }
```

block to provide a handler for the occurring *Logging* effect. The actual implementation for the *logResult* effect operation is provided on line 14 to 17. More information on the Syntax of Effekt and especially the *resume* keyword on line 16 will be given in subsection 1.1.3.

In the above example we can see a special property of effect handlers that separates them from exception handlers: effect handlers allow resumption, i.e. after an effect

operation call hands the control flow over to the handler, the execution can be resumed at the effect operation call site. This allows handlers to generalise over many other concepts like expressing complex control flow or modeling dependencies [Brachthäuser et al., 2018, Brachthauser and Leijen, 2019]. Another value of effects and handlers lies in their ease of composition and modularity. Composing monads is seen as a hard problem in general [Lüth and Ghani, 2002]. In contrast to that, composing programs with effects and handlers is simple, because the required effects will always be reflected in a computations type [Brachthäuser et al., 2020b]. Modularity is maintained as each effect handler only offers semantics to its associated effect [Schrijvers et al., 2019].

1.1.3 The Effekt language

The Effekt[1] programming language implements effect handlers as described by Plotkin and Pretnar [Brachthäuser et al., 2020b, Plotkin and Pretnar, 2009]. Effekt was designed around the idea that effects may not only be seen as the side effects a computation may yield; they can also be understood as "capabilities a computation requires from its context" [Brachthäuser et al., 2020a]. Effekt uses so called lexical effect handlers which allow that "each use [...] of an effect can be singled out by name, bound by an enclosing handler and tracked in the type of the expression" [Biernacki et al., 2019]. This concept transfers the common way one reasons about lexical scoping to effects and handlers, where traditional effect handlers correspond to dynamic binding.

In figure 1.2 we see how effects are defined in Effekt; the *effect* keyword on line 1 defines an effect signature with the name *Logging*. On lines 2 and 3 we define two effect operations. As they are part of the effect signature of *Logging*, calling them will always give rise to the *Logging* effect. We do not implement the effect operations inside of the effect signature, as the implementation will be served by a corresponding handler. Effekt supports generic typing, as can be seen in the type parameter *[A]* on line 2.

```
1  effect Logging {
2    def logResult[A](s: A) : A
3    def logMessage(s: String) : String
4  }
```

Figure 1.2: Definition of an effect called *logging* along with two effect operations that may give rise to the *logging* effect. Effekt supports generic typing: *logResult* is generic in some type *A*.

[1] Official website: https://effekt-lang.org/

In figure 1.3 we call the *logMessage* effect operation inside of a function definition (line 3). As we can see in the type of the *add* function definition on line 1, the *Logging* effect has become part of *add*'s type and effects.

```
1 def add(a: Int, b: Int) : Int / { Logging } = {
2     val result = a + b;
3     logMessage(result.show);
4     result;
5 }
```

Figure 1.3: Definition of a function in Effekt. Required effects are part of the signature, denoted after the return type.

Effekt's syntax separates the return type and the effect set with a forward slash:

```
Int / { Logging }
```

The curly braces around *Logging* indicate that this effect may be just one of a set of effects. If a function has only a single effect, we can omit the curly braces:

```
Int / Logging
```

Pure functions have an empty effect set, for example:

```
def add(a: Int, b: Int) : Int / {} = a + b
```

In figure 1.4 we call the *add* function defined in figure 1.3. As explained in subsection 1.1.2, we provide the call to *add* with a handler using the *try-with* construct.

```
1  def main() : Int / {} = {
2    try {
3      add(23, 42)
4    } with Logging {
5      def logMessage(s) = {
6        println("Log message: " ++ s);
7        resume(s);
8      }
9      def logResult(s) = resume(s)
10   };
11 }
```

Figure 1.4: Call to an effectful function and handling of an effect in the Effekt language.

The provided handler implements every effect operation of the *Logging* effect; lines 5 to 8 define what *logMessage* is supposed to do in this context, line 9 defines the behavior of *logResult* in this context. Though *add* only uses the *logMessage* effect operation, we must provide a fully implemented handler to guarantee handling.

Effekt implements type inference: definitions of variables or functions can, but do not have to be explicitly typed. Figure 1.5 shows an example of type inference. The return type *Int* is inferred from the return type of the integer addition on line 2.

```
1 def sum(a: Int, b: Int) = {
2     a+b
3 }
```

Figure 1.5: Example code demonstrating type inference in Effekt.

Effekt also supports holes and typed holes; programmers may leave some expressions as unimplemented holes in their code but Effekt will type check these programs successfully by assuming that the type of the hole fits the type required by its context. Figure 1.6 gives several commented examples of holes.

```
1  val a = <> // type of a remains unknown
2
3  val b = 5 + <> // type of b is assumed to be Int
4
5  val c = <{ // type of c is unknown
6      42 // type of the hole is Int
7  }>
8
9  val d = <{ // type of d is unknown
10     // type of the hole is Unit with some effect
11     someEffectOperation()
12 }>
13
14 // function passes type checking even tough type
15 // definition collides with the holes type
16 def doSomething(s: String) : Int = <{
17     s
18 }>
```

Figure 1.6: Commented examples of type inference and typed holes in Effekt. Lines 27 to 29 demonstrate how typed holes may be used to pass type checking in spite of conflicting return types.

We use Effekt as an exemplary language to design and implement development environment features for effects and handlers. This choice of language was made due to the closeness of the original authors to the faculty. In addition to that, the original Effekt publication already provided a language extension to the Visual Studio Code editor and a language server that could be used as a starting point. The technical details will be discussed in section 2.1.

1.2 Language support in integrated development environments

Computer programs are usually encoded as written text. Thus a text editor most often is necessary to write a computer program. With increasing complexity of programs, the requirements to editors may grow as well. As programming not only consists of writing code and executing it, the programmers environment is forced to fulfill additional needs. These needs include file, version and project management, the finding and avoidance of coding mistakes (debugging and testing), restructuring and rewriting of existing code (refactoring), help in reading and understanding foreign code, and translating or deploying it to a working program.

An integrated development environment (IDE) tries to unite a multitude of the needed tools in a single application to avoid unnecessary mental context switches. Thus, the more modern development environments are exactly that: a complete environment tailored to the needs of a certain domain, programming language or user group.

One core aspect of IDEs is that it has has means to translate or run the currently loaded code. To be able to do this, the IDE needs an interface to interact with the language in terms of an interpreter or compiler. Therefore the IDE has means to interact with the language itself as interpreters and compilers offer features such as static code analysis to provide error messages or type checking. These interfaces can be used to offer programming aides such as those listed in table 1.1. It is worth mentioning that these features focus on the text editing part in programming. Language support is mostly given through these editor features because the editor is the main tool to interact with the language.

Feature	Explanation
Syntax highlighting	Source code symbols are coloured differently in order to transport semantics on a visual level
Auto-completion	Suggestion of completions for already typed symbols
Jump-to definition	Navigating to its definition by clicking on a symbol
Code folding	Hide or reveal code blocks to reduce distraction
Automatic code indentation	Automatically apply indentation rules to improve readability of source code
Code refactoring	Automated code changes that keep semantics intact but may lead to improved code
Compilation error reporting	Display error messages and further information, e.g. error position indication using squiggly underlines.
Find all references	Search for references to a symbol based on its semantics instead of a mere textual search for the symbol's string.

Table 1.1: List of common IDE editor features as mentioned in [Dyke, 2011, Heinonen et al., 2014, Masci and Munoz, 2019]

The features listed above commonly rely on the generation of an abstract syntax tree of the underlying program in order to offer information on the code. This is an algorithmic way of extracting information on the source code. Recent advancements in artificial intelligence allow for novel ways to generate information such as auto-completion responses or even whole code snippets [Bruch et al., 2009, Zhang et al., 2019]. These machine learning models are able to include contextual information and infer information using statistical methods [Svyatkovskiy et al., 2019]. Due to their foundation in natural language processing, some of these models may even generate useful code for languages they were not trained with [Chen et al., 2021].

Research in the field of language support in IDEs reaches from topics like general IDE user experience to support of novel programmers through interactive learning environments. We list some examplary research in the following section.

1.3 Related work

Both the field of effects and handlers and the field of IDE language support and usability are under active research. However, there seems to be little to no research available on the intersection of these topics.

Even the most common research languages that implement effects and handlers do not offer advanced IDE features common to major programming languages.

Languages such as Eff [Bauer and Pretnar, 2015], Koka [Leijen, 2014], Helium [Biernacki et al., 2019] and Links [Cooper et al., 2006] all support syntax highlighting through different code editor extensions. However, at the time of writing none of these language extensions implement advanced IDE features such as code completion, code refactoring, hover information or jump to definition. The Unison[2] language features an extension for the Vim editor that offers code completion. None of these language extensions implement features that could support the process of thinking and programming with new language constructs such as effects and handlers. While the aforementioned languages all offer new ways to increase expressiveness, the increased expressiveness is only reflected in syntax. Figure 1.7 gives an example of a Koka program in the official language extension[3] for Visual Studio Code. As we can see, effects, effect operations and handlers are syntactically highlighted. No other common IDE features are offered.

```
1   // A generator effect with one operation
2   effect yield<a>
3     fun yield( x : a ) : ()
4
5   // Traverse a list and yield the elements
6   fun traverse( xs : list<a> ) : yield<a> ()
7     match xs
8       Cons(x,xx) → { yield(x); traverse(xx) }
9       Nil        → ()
10
11  fun main() : console ()
12    with fun yield(i : int)
13      println("yielded " ++ i.show)
14    [1,2,3].traverse
```

Figure 1.7: Screenshot of a program in Koka involving effects and handlers, displayed in Visual Studio Code using the official Koka extension.

Research in the general field of language support in IDEs reaches from topics like user experience to support of novice programmers through interactive learning environments. Authors such as Fan et al., Svyatkovskiy and Rask et al. focused on the introduction of new features, improvement of known features or language extension implementations [Fan et al., 2019, Svyatkovskiy et al., 2019, Rask et al., 2021]. There seems to be little data on the general quality and benefits of existing, commonly implemented and used IDE features.

The most studied feature seemingly is syntax highlighting. Many methods have been used to gain insight into the effects of syntax highlighting on code comprehension,

[2] Unfortunately there is no paper on Unison, yet. Information can be found on the official website: https://www.unisonweb.org/

[3] Downloadable at the VS Code marketplace: https://marketplace.visualstudio.com/items?itemName=koka.language-koka

programming tasks, and others [Sarkar, 2015, Beelders and du Plessis, 2016a, Beelders and du Plessis, 2016b, Hannebauer et al., 2018, Häregård and Kruger, 2019]. The general verdict seems to be that syntax highlighting has little to no positive impact on the selected tasks, although methodological limitations were taken into account.

Some authors investigate the effect of programming aides on novice programmers. For example, Dillon et al. could show that novice programmers struggled more with programming tasks when using a low assistive environment [Dillon et al., 2012]. The authors compared those novice programmers to other novice programmers who used a moderately assistive environment that offered common IDE features like syntax highlighting, error highlighting and auto completion. The correlation persisted despite similar levels of prior experience. Kelleher and Pausch constructed a whole taxonomy of programming environments and languages that focus on novice programmers [Kelleher and Pausch, 2005]. Most of the environments and languages presented focus on simplifying core programming aspects, e.g. by using syntax and grammar that is closer to a natural language, or by using alternatives for entering code, such as drag-and-drop of predefined code blocks. No information is given on the actual effect on novice programmers.

Parker et al. list the availability of an "easy to use development environment" as a language selection criteria for introductory programming courses [Parker et al., 2006]. They refer to the assistive effect of well designed development environments on language learning as shown by Eisenstadt and Lewis [Eisenstadt and Lewis, 2018].

Another work that analyses students' behavior within a programming environment is the work by Dyke, where the analysis focused on students' use of IDE features [Dyke, 2011]. Dyke found that the use of features such as code generation had a high correlation with course success. However, he did not discuss the students' backgrounds. It could be possible that the results are at least partially explained by previous experience [Vihavainen et al., 2014].

Assuming that every person learning a new language can be seen as a novice programmer to some degree, this may hint at the importance of assistive development environments for the introduction of new programming languages and language features. As pointed out by Medeiros et al., learning of syntax is one of the barriers programmers face when learning a new language or programming in general [Medeiros et al., 2018]. Tools that help with syntax thus may be crucial when trying to avoid frustration of novel developers.

The availability of language support features and tooling in IDEs may also contribute to the acceptance of a language. Meyerovich and Rabkin found in a survey among software developers that the availability of tools may be one important criteria for the choice of a programming language: around 40% of respondents described the availability of language tools as an aspect of "medium or strong importance" [Meyerovich and Rabkin, 2013].

1.4 Conclusion

We have seen that no research exists on the intersection of IDE language support and effects and handlers. It is shown that tools embedded in the development environment can improve the understanding and use of language features and concepts. Effects and handlers are a relatively new concept that provides high expressive power and composability for computational effects in programming languages. Novel IDE features could facilitate the understanding, exploration, and use of effects and handlers.

2 Implementation

In the following we present concepts for IDE features that may support programmers in reasoning about and programming with effects and handlers. First we give some insight into the technologies used for the implemented features. We then list all IDE features, differentiating between features we implemented and features that are just theoretical and not yet part of the Effekt language extension. For the former we give details on the implementation and their requirements. For the latter we only discuss requirements.

2.1 Technical details

2.1.1 Visual Studio Code

Visual Studio Code (VS Code)[1] is a program primarily intended for editing text files, especially source code. VS Code was first released in April 2015 by the Microsoft Corporation and is since developed as open source software under the MIT license. VS Code is developed using web technologies such as TypeScript, JavaScript, CSS and the Electron framework to allow cross-platform development for Microsoft's Windows, Apple's macOS and the Linux operating system.

In contrast to the development environments of the "Visual Studio" series, also developed by Microsoft, VS Code provides only basic functionalities of a text editor along with basic project and file management capabilities. Additional functionalities can be added through plug-ins, so-called extensions. These extensions allow the integration of functionalities that are usually found in full development environments: the integration of compilers, debuggers and tools for static code analysis, version management systems and many more. As VS Code itself is was not designed for a specific programming language, extensions are used to add language-specific support features such as those mentioned in table 1.1. The development of extensions is made possible through an extension API[2]. To allow seamless communication with language tools such as compilers or code analysis tools, VS Code offers a client implementation for the Language Server Protocol[3] (LSP).

[1]Official website: https://code.visualstudio.com/
[2]Official API documentation: https://code.visualstudio.com/api
[3]Official website: https://microsoft.github.io/language-server-protocol/

2.1.2 Language Server Protocol

The Language Server Protocol defines a unified interface for communication between a language client (e.g. an IDE) and a language server. The language server represents a provider of language services for a programming language. These language services can include, for example, the provision of additional semantic information about the program code represented in an IDE. The information provided by the LSP server can be used by the client to enrich the development environment informatively, for example by highlighting program parts of certain semantics in color (commonly called "semantic syntax highlighting"). The information can also be used to facilitate the work of the developer. Examples would be code completion suggestions, automated code changes or easier navigation in the program code.

Figure 2.1 shows the general way of bidirectional communication between language clients and language servers. Once a connection is established between parties, the LSP uses JSON encoded remote procedure calls (RPC) for communication.

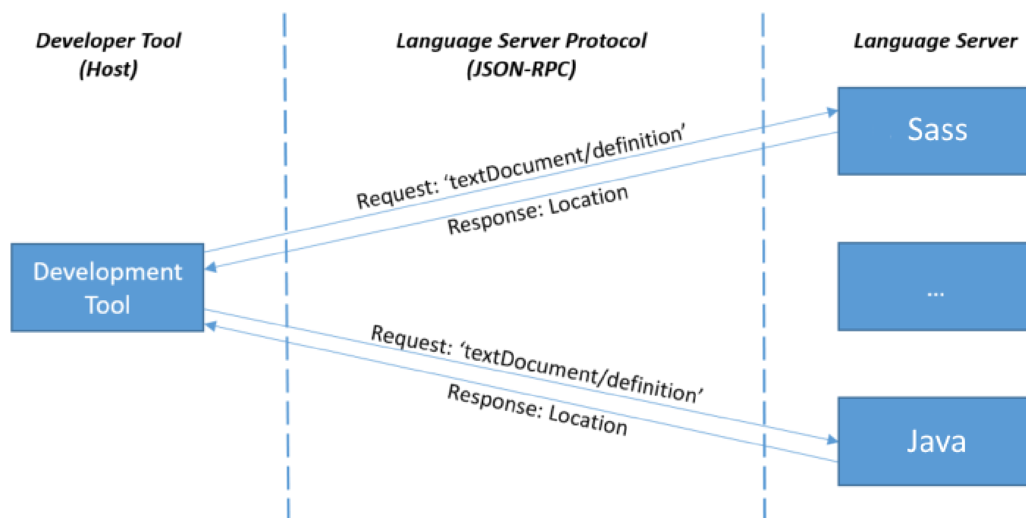


Figure 2.1: Outline of communication between a developer tool and multiple language servers via the LSP. Image source: Microsoft Corporation [Microsoft Corporation, 2021].

The advantage of using the LSP and a client-server architecture to provide language services is its flexibility. Once a language provides an LSP server, many different clients can directly benefit from the provided language services without needing an own implementation of a language service provider. A general LSP client implementation is sufficient. Today, many common code editors and IDEs feature an LSP client. Among them are the Atom editor, Eclipse IDE, Sublime text, Visual

Studio and many more[4].

We used the official "VSCode Language Server - Node"[5] (Microsoft Corporation) library as LSP implementation on the client side. On the server side, the LSP was already implemented through the usage of "LSP4J"[6] (Eclipse Foundation) in the Kiama[7][Sloane, 2008] library. An own version of Kiama was derived from the official repository to implement missing LSP features.

In our implementation we made repeated use of LSP's *workspace/executeCommand* command. This command allowed us to implement functionality that is not covered by other specifications of the LSP. For example, LSP offers a *textDocument/hover* command to provide a unified interface to request information upon hovering text in the client. Some of our features proposed in section 2.2 need information that is not offered upon a pre-defined actions such as hovering a token. For these actions a manual call to *workspace/executeCommand* is triggered.

2.1.3 Previously implemented features

As mentioned in section 1.1.3, the original Effekt publication already provided a language extension for VS Code. We used this extension as a starting point and implemented our own features on top of the existing ones. The original Effekt extension implemented syntax highlighting, jump-to-definition, error reports, additional information about symbols when hovered over, and a code refactoring to update the type and effects of an explicitly typed function.

[4]More complete list: https://microsoft.github.io/language-server-protocol/implementors/tools/
[5]Official project repository: https://github.com/microsoft/vscode-languageserver-node
[6]Official repository: https://projects.eclipse.org/projects/technology.lsp4j
[7]Official repository: https://github.com/inkytonik/kiama

2.2 Proposed features

2.2.1 Implemented features

The following features were implemented in the Effekt VS Code extension.

Inferred type and effect decoration

To a programmer it might be of interest to see the effects of a function that is not explicitly typed. Take the following example depicted in figure 2.2. Here we call *someFunction* that we imported from *SomeModule*. What is the type of *mainFunction* and does it bind effects? To answer this question without any IDE feature, a programmer would have to either look at the documentation or look at the definition of *someFunction* in *SomeModule*.

```
import SomeModule

def mainFunction() = {
  someFunction("foo")
}
```

Figure 2.2: Example program for the inferred type and effect decoration feature.

Effekt offers a static type and effect system that is capable of type inference. This means that for a given piece of not explicitly typed code the type may be deducible from the context. The programmer can omit the type and effect information in definitions. We propose to make use of type inference to display type and effect information on a given function declaration that is not explicitly typed. Figure 2.3 shows an example of the implemented proposal. An implicitly typed function is added a non editable piece of information that reflects the currently inferred type and effect set of the function. As we can see, the *sum* function does not only return an integer but it also requires a *Console* effect.

```
def sum(a: Int, b: Int) = {
  val res = a + b;
  println(res);
  res;
}
```

(a) Example function without inferred inline type and effect decoration

```
def sum(a: Int, b: Int) : Int / Console = {
  val res = a + b;
  println(res);
  res;
}
```

(b) Example function with inferred inline type and effect decoration

Figure 2.3: Exemplary screenshots demonstrating the inferred type and effect decoration feature.

Implementation details
The inferred type hints are displayed using the Decorations API of VS Code. The API offers the capability to display non editable characters at any given position inside the editor. Existing code at that position is not altered as the decorations are only part of the visual representation. To obtain regions of interest that may need an inferred type decoration, the language server is requested whenever the current file is saved. The abstract syntax tree of the currently displayed source file is generated by the server. Every function declaration is searched in the tree. Inferred types and effects are gathered for function declarations that are not explicitly typed. The source code position for the type information is calculated. Both the source code position and the inferred type and effects are sent back to the VS Code extension in JSON format. The *DecorationRenderOptions* of an inline decoration are adjusted for the given source code position and text content. The result is a non editable type and effect information that integrates seamlessly into the original source code.

Effect origin hint

In a larger code base it might be difficult to see immediately what piece of code has given rise to an effect or, in terms of the Effekt language, what causes the requirement of an effect. Figure 2.4 shows an example. A function *ex* is defined that requires several effects: *Fail*, *Next*, *Error* and *Flip*. If we wanted to handle, for example, die *Error* effect inside of *ex*, we have to find the function calls that give rise to the effect. Thus we would have to look at the definition of each function called inside of *ex*.

```
1  def ex() : Int / { Fail, Next, Error, Flip } = {
2      or {
3          accept("do");
4          commit {
5              accept("foo");
6              0;
7          }
8      } {
9          accept("do");
10         accept("bar");
11         1
12     }
13 }
```

Figure 2.4: Example program for the effect origin hint feature.

We propose an option in the IDE that allows to highlight the origin of a given effect inside of a function definition. Figure 2.5 shows the workflow of the implemented proposal. Whenever the cursor is moved inside the scope of a function definition

that binds effects, clickable items are displayed above that function definition. For each effect that is part of the functions type and effect set one such item is created. Clicking an item highlights the function or effect operation call that requires the corresponding effect. For the example above, clicking the *Error* item reveals that *commit* requires the effect: the call to *commit* is highlighted by shading the background and putting the text in cursive.

```
Fail | Next | Error | Flip
def ex() : Int / { F 1. ext, Error, Flip } =
  or {
    accept("do");
    commit {
      accept("foo");
      0;
    }
  } {
    accept("do");
    accept("bar");
    1
  }
```

(a) Clickable items are placed on top of functions that bind effects (1.).

```
Fail | Next | Error | Flip
def ex() : 1. / { Fail, Next, Error, Flip } =
  or {
    accept("do");
    commit {
      accept("foo"); 2.
      0;
    }
  } {
    accept("do");
    accept("bar");
    1
  }
```

(b) Clicking an option (1.) slightly highlights the origin of the underlying effect (2.).

Figure 2.5: Workflow of the effect origin hint feature. Inferred type and effect decorations are used as described in subsection 2.2.1. Moving the cursor inside a function definition reveals an inline menu with the names of all required effects above the definition (a). Clicking one of the names highlights code that requires the corresponding effect (b).

Implementation details

We implemented the inline clickable options using the CodeLens feature of the

VS Code extension API. CodeLenses are clickable links inside the editor that intersperse the displayed code[8]. Whenever the currently active source file is saved, the Effekt extension requests CodeLens information for that file from the LSP server. Information on each effect, its binder and its origin position is gathered for the file and sent back to the client. CodeLenses are created for each piece of information. The on-click action is set to a function that receives the effect information and highlights the range of the effect's source. Each CodeLens creates an extra line inside the editor. To avoid visual clutter, we decided to only show CodeLenses for the scope the cursor resides in. Whenever the cursor is moved, it is checked whether it moved inside the range of a an effect. If it does, the correspondig effect CodeLens is displayed. By default, no CodeLenses are shown.

Effect binder hint

Effekt uses lexical effects: effects are lexically bound to the surrounding scope but can also be bound by a corresponding handler. Due to this dynamic behavior it can be hard to grasp where a certain effect is bound or handled. Consider for example the program in figure 2.6.

```
1 def mainFunction() = {
2
3   effectfulFunction("foo")
4
5   someApiFunction(){ (s: String) =>
6     effectfulFunction(s)
7   }
8 }
```

Figure 2.6: Example program for the effect binder hint feature.

We want to know where the effects are bound that each call to *effectfulFunction* may introduce. For the first call on line 3 the reasoning is simple: as no handler is defined, any effect must be bound lexically by *mainFunction*. Handing *effectfulFunction* over to *someApiFunction* on line 6 makes reasoning more complicated. We would have to look at least at the type of *someApiFunction* to find out what happens to the effects introduced by *effectfulFunction*.

The effect binder hint feature tries to give the answer immediately: given an effect, the feature highlights the range of the corresponding binder. Figure 2.7 (a) shows the example program from figure 2.6 without effect origin or binder hints. Moving the cursor over the first call to *effectfulFunction* reveals an inline menu indicating

[8]An introduction to CodeLenes can be found at https://code.visualstudio.com/blogs/2017/02/12/code-lens-roundup

that the function call introduces the *logging* effect. Clicking the *logging* effect option highlights the whole definition of *mainFunction* by softly underlining and shading it, indicating that the logging effect is bound by this function (Figure 2.7 (b)). Clicking on the *logging* effect option inside the call to *someApiFunction* softly underlines and shades the block argument of the call, indicating that the *logging* effect is handled inside of *someApiFunction* (Figure 2.7 (c)).

```
def mainFunction() : Int / { Console, logging } = {

  effectfulFunction("foo")

  someApiFunction(){ (s: String) ⇒
    effectfulFunction(s)
  }

}
```

(a) Example program without binder hints.

```
def mainFunction() : Int / { Console, logging } = {   2.

  logging {}  1.
  effectfulFunction("foo")

  someApiFunction(){ (s: String) ⇒
    effectfulFunction(s)
  }

}
```

(b) The *logging* effect is bound by *mainFunction*. Upon clicking the annotation (1.), the whole function definition is highlighted (2.).

```
def mainFunction() : Int / { Console, logging } = {

  effectfulFunction("foo")

  lo  1.
  s   piFunction(){ (s: String) ⇒
    logging {}                     2.
    effectfulFunction(s)
  }

}
```

(c) The *logging* effect is bound inside of *someApiFunction*. Clicking the annotation (1.) slightly highlights the block argument of *someApi-Function* (2.).

Figure 2.7: Example screenshots of the effect binder hint feature. Effect origin hints and inferred type and effect decorations are used as described in 2.2.1.

Implementation details

The effect binder hint feature was implemented using the same technique as for the effect origin hint feature. The LSP server is requested for information on all available effect origins in a source file. For each origin of an effect the position of that origin and the range of the corresponding binder in the source code are extracted and sent back to the client. The client creates CodeLenses for each effect origin. CodeLenses are only shown when the cursor enters the origin of an effect to avoid visual clutter. By default, no CodeLenses are shown.

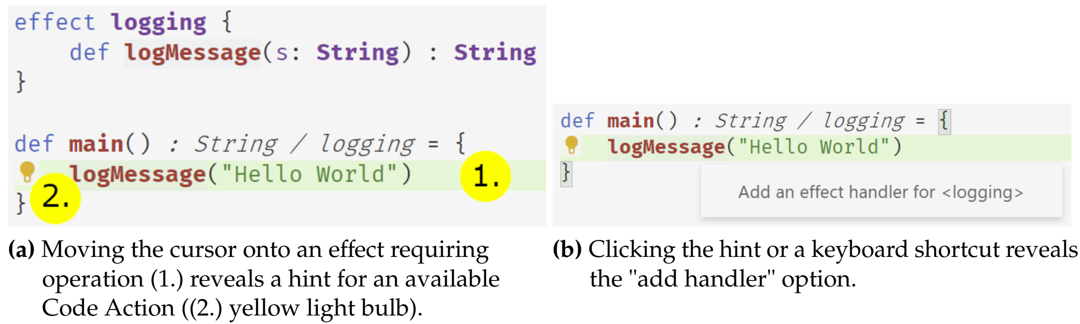
Automatic handler creation

Adding a handler to a certain effect is a common task in a programming language that implements effects and handlers. However, typing the necessary code is bothersome and unnecessary. Take for example the code shown in figure 2.8. It involves the call of several effectful functions. If we want to make *main* a pure function, we have to offer at least two handlers, one for *Async* and one for *Logging*.

```
1  def main() : Unit / { Async, Logging } = {
2
3    logMessage("Starting");
4    runAsync {
5      logMessage("Running thread 1");
6      // ...
7    } {
8      logMessage("Running thread 2");
9      // ...
10   }
11
12 }
```

Figure 2.8: Example program for the automatic handler creation feature.

Offering a handler for each effect would involve typing nearly identical code and manual indentation for every effect. For each effect, the corresponding handlers blueprint is well defined. Hence we propose to add an option to the IDE to automatically insert handler blueprints for selected effects. Figure 2.9 demonstrates the proposed workflow on a single effect in the implemented Effekt VS Code extension. We insert an empty handler that omits the actual implementation and replaces it with a typed hole to allow type checking without a valid implementation.



(a) Moving the cursor onto an effect requiring operation (1.) reveals a hint for an available Code Action ((2.) yellow light bulb).

(b) Clicking the hint or a keyboard shortcut reveals the "add handler" option.

(c) Triggering the operation wraps the source of the *logging* effect in a handler (1.). A hole is used to allow type checking without implemented effect operations (2.).

Figure 2.9: Workflow of the automatic handler creation feature. The effect handling is reflected in the updated return type of the surrounding function in subfigure (c).

Implementation details

The *add effect handler* operation is implemented as a so called *Code Action*[9] in VS Code. Code Actions are supposed to provide both quick fixes and refactorings for detected issues in the source code. The information of available Code Actions is provided by the active language extension in VS Code. In our case, the Effekt extension provides the Code Action by requesting the LSP server for possible Code Actions in the current file. After creation of the abstract syntax tree of that file, all nodes in the tree that indicate an effect operation without a handler in the current scope are added to the list of nodes that have effects a potential user may want to handle. For each of these nodes a matching Code Action is created, consisting of the effect name and range in the source code and an appropriate handler dummy. Whenever the cursor enters that range, an indicator is shown to signal the availability of a code action. Triggering the action results in the replacement of the original node with a new one representing the effect issuing code, enclosed by the handler.

[9] An introduction to Code Actions is given at https://code.visualstudio.com/docs/editor/refactoring

2.2.2 Outlined features

The following features pose ideas and were not implemented in the Effekt VS Code extension. Implementation may follow at a later point in time.

Insert type and effects

Accidentally contaminating a function with an unwanted effect may happen easily. As Effekt supports type inference and functions don't need explicit typing, the accidental introduction of a new effect to a function may remain unnoticed. Take for example the code in figure 2.10. The *add* function on line 9 is not explicitly typed.

```
1  effect logging {
2    def logResult[A](s: A) : A
3  }
4
5  effect async {
6    def fetch(s: String) : String
7  }
8
9  def add(a: Int, b: Int) = {
10   val res = a + b;
11   logResult(res)
12 }
13
14 def main() = {
15   try {
16     add(23, 42)
17   } with logging {
18     def logResult(s) = {
19       println("Logging result: " ++ s.show);
20       resume(s);
21     }
22   };
23 }
```

Figure 2.10: Example program for the *insert type and effects* feature.

If we were to introduce the *async* effect to *add* by calling *fetch* inside of it, the program would break as the effect is never handled. This behavior would not be obvious until compilation, especially in a larger code base or in deeply nested code. The *inferred type and effect decoration* feature from 2.2.1 could reflect the newly introduced effect. However, the possibly unwanted behavior is not prevented. In order to prevent

the behavior, the programmer needs to explicitly type the *add* function as shown in Figure 2.11.

```
1 def add(a: Int, b: Int) : Int / logging = {
2   val res = a + b;
3   logResult(res)
4 }
```

Figure 2.11: Explicitly typed version of the *add* function from figure 2.10.

The introduction or requirement of the async effect would now conflict with the explicit type and effect set of *add*. The type of some expression may often be quite obvious. For example, the type of a function is commonly determined by what is returned through return statements. The introduction of effects however may happen by accident and in multiple places.

We propose an IDE operation to insert the inferred type and effects of an untyped function, making it explicitly typed and thus invulnerable to accidentally introduced effects.

Requirements for implementation

In order to implement the *insert type and effects* feature in an LSP environment, the LSP server would require type inference and the ability to obtain all expressions that are not explicitly typed. Inferred type and effects and their theoretical position in the source code could be calculated and transmitted to the LSP client. The client could then show a hint close to these expressions. Clicking the hint would exchange the implicit type with the explicit one received from the server. A *Code Action* as seen in 2.2.1 *Automatic handler creation* could be used to implement this part in VS Code.

Display effects as requirements

As stated in section 1.1.3, the Effekt language revolves around the idea to understand effects as capabilities that are required from its context. We propose to introduce semantic hints that transport this concept visually using common notation of functions. These hints may be visualized using non-editable inline text as in 2.2.1 *Inferred type and effect decoration*. Figure 2.12 shows one idea of how effects could be displayed as requirements.

```
1 effect logging {
2   def logResult(s: Int) : Int
3 }
4
5 def add(a: Int, b: Int) : Int / logging = {
6   val res = a + b;
7   logResult(res)
8 }
```

```
1 def add(a: Int, b: Int) : Int / logging = {
2   val res = a + b;
3   (logResult => logResult(res))
4 }
```

Figure 2.12: First idea for a visualization of effects as requirements. Top: effect operation usage in *add* without representation as a requirement. Bottom: the same function definition but on line 3 the *logResult* effect operation is displayed as a capability that is required from the context.

Here we use the notation of functions to express that the effect operation *logResult* is not defined in some global scope but in fact must be handed in explicitly if we want to use it. Another way of visualizing effects as requirements is shown in figure 2.13. Here we stress the fact that we hand in the whole effect as a requirement and that the effect operation *logResult* gives rise to the *logging* effect. However, property access on effects via a dot-operator '.' is not actually part of Effekt's syntax. Thus this representation may also be misleading.

```
1 def add(a: Int, b: Int) : Int / logging = {
2   val res = a + b;
3   (logging => logging.logResult(res))
4 }
```

Figure 2.13: Second idea for a visualization of effects as requirements. The same function definition as in figure 2.12 is used, but on line 3 the *logging* effect is displayed as a capability that is required from the context and the *logResult* function is obviously an effect operation of the *logging* effect, denoted by a fictional property access operator '.'.

Requirements for implementation

In order to implement this feature using the LSP, a client could request all positions of effect requirements in the currently opened source file from the server. The server could respond with a set of effects along with the source code positions of the effect operation calls that require the effect. In case of VS Code, the client could then utilise the Decorations API mentioned in subsection 2.2.1 to display the inline decorations we propose above.

Display handled effects as arguments

Whenever code is wrapped in a *try-with* block to offer handlers, it may not be directly visible what piece of code actually requires the handled effect. Take for example the code in figure 2.14. If we don't know the definitions of *foo, bar* and *baz*, we can not know which function requires which of the two handled effects.

```
1  def main() = {
2      try {
3          foo(42);
4          bar(23){
5              p => baz(p)
6          }
7      } with Logging {
8          def logMessage(s) = resume(s)
9      } with Error {
10         def fail() = logMessage("Aborting!")
11     }
12 }
```

Figure 2.14

In analogy to the *display effects as requirements* feature, we propose to use the notation of function arguments to transport the requirements visually. Non-editable inline text could be used to visualize this feature. Figure 2.15 shows the example code from figure 2.14 with visualized effect requirements. Each handled effect is appended as a mock argument to the requiring function call. We instantly see that *foo* requires both handled effects while *bar* and *baz* only require *Error* and *Logging*, respectively.

```
1  def main() = {
2      try {
3          foo(42)(Logging, Error);
4          bar(23)(Error){
5              p => baz(p)(Logging)
6          }
7      } with Logging {
8          def logMessage(s) = resume(s)
9      } with Error {
10         def fail() = logMessage("Aborting!")
11     }
12 }
```

Figure 2.15

Requirements for implementation

We propose the same requirements as for the implementation of the *display effects as requirements* feature. For each handler, the server would search for the requirement of every handled effect and extract the according source code positions for function arguments. The client could then utilise non-editable inline text to annotate the effects as arguments to these function calls.

List handlers of an effect

Reading and understanding foreign source code can be a complex and time consuming task. In a language that implements effects and handlers, it might be hard to understand what a certain effect is actually meant to express and how matching effect operation implementations could look like. Take the code in figure 2.16 as an example. It defines two effects, *Logging* and *State*. Let us assume that many lines of code are following the effect definitions. To a person that did not write the code it may be hard to understand what the *Logging* and *State* effects are supposed to model. One may assume the intended functionality of *logString* by its name, arguments and type. However, the *State* effect may be a mystery to a person that is not familiar with the concept of state and the way it is implemented in functional programming.

```
1  effect Logging {
2    def logString(s: String) : String
3  }
4
5  effect State {
6    def get() : Int
7    def put(n : Int) : Unit
8  }
9
10 // Lots of code is following
11 // [...]
```

Figure 2.16: Example code for the *list handlers* feature. The reader of this code might be wondering how and where both effects are handled in the code base.

We propose an IDE feature that lists all available handler implementations of a given effect, along with their position in the source code. This feature would especially allow readers of foreign source code to quickly jump from an effect definition to an implementation that may serve as an example. For the *State* effect in figure 2.16 the effect requirement and handler implementation depicted in figure 2.17 might be found.

Requirements for implementation

In order to create a list of all handlers to all effects defined in the current file, the client could send the current file name to the language server. After creating the abstract syntax tree for that file, the server can extract all effect definitions and all handler definitions, along with their position in the source file. A mapping could be constructed from effects to handlers. The mapping could be sent back to the client. The client may then offer a list view or other means to display the source code positions of effect handlers to a selected effect. Upon selecting a handler from that list, the client may quickly put the corresponding handler code into view.

```
// takes a program with state and
// offers a local mutable state to it
def state { prog: Unit / State } = {
  var s = 0;
  try {
    // by definition, prog is a program with state
    // therefore we can define a handler in this closure
    prog()
  } with State {
    // whenever we need to read the state
    def get() = resume(s)
    //whenever we need to write to the state
    def put(n) = {
      s = n;
      resume(())
    }
  };
  //return the state
  s
}
```

Figure 2.17: Usage of the *State* effect from figure 2.16 and definition of a matching handler that may serve as an example to a programmer. The *state* function executes a program that requires the *State* effect and offers a handler that implements local mutable state.

2.3 Summary

We implemented four language features in a VS Code extension to support the usage of effects and handlers in the Effekt language. Furthermore, we collected ideas on novel language support features. We can conclude that the concept of effects and handlers allows for completely new IDE features that may benefit a developer.

3 Discussion

We found that there is no research on language extensions that support the use of effects and handlers in IDEs. We presented several ideas for supporting IDE features, demonstrated use cases and implemented four of these functionalities. In what follows, we discuss and theorise about the usefulness of each of our proposals. We distinguish between implemented and outlined features. Due to the lack of similar research, we draw parallels to more common IDE features.

3.1 Implemented IDE features

3.1.1 Inferred type and effect decoration

We visualized the inferred type and effects of functions that are not explicitly typed. It is questionable whether the visibility of inferred types and effects has influence on the understanding of effects and handlers in particular. However, Meyerovich and Rabkin found that when querying software developers, 45% (+/- 10) agreed that "Using types helps improve readability" of source code [Meyerovich and Rabkin, 2013]. Fischer and Hanenberg confirmed this when they compared programmers performance on tasks in statically typed TypeScript with dynamically typed JavaScript. Their verdict is that "the effect of static type systems is larger than often assumed, at least in comparison to code completion" [Fischer and Hanenberg, 2015]. This could hint to the usefulness of the presented feature in settings where code comprehension is of great importance. One such situation could be the learning and understanding of novel language features.

3.1.2 Effect origin and binder hint

The effect origin hint and binder hint features show similarities to other common IDE features. Our implementation allows to quickly find the function call that requires a bound effect, or to identify the corresponding binder to a given effect. This is similar to the common *jump-to-definition* feature where a shortcut is provided to navigate from a usage location of a source symbol to its location of definition. The inverted behavior is commonly offered as a *find-usage* feature that yields all usage sites of a symbol. Navigation in the code is considered a frequently occurring aspect of programming and seen as an important skill of a programmer [Murphy

et al., 2006, Jones and Burnett, 2007, Mader and Egyed, 2011]. Even though jump-to-definition is commonly mentioned in papers on IDEs and IDE features, there seems to be no data describing the impact of this feature on programmers [Masci and Munoz, 2019, Németh and Brunner, 2019, Szalay et al., 2018]. Therefore we can not draw parallels to other findings.

3.1.3 Automatic handler creation

Automatic generation of handler code for a given effect primarily reduces typing. This feature may be ranked among other code generating utilities such as *auto-completion*, *refactoring tools* or *code snippet insertion*. Auto-completion (see table 1.1), also referred to as code completion, is assumed to be one of the common features for a code editor [Robbes and Lanza, 2008]. Robbes and Lanza cite Murphy et al.'s empirical finding that among 41 Java developers, every single one made use of the code completion feature during the study and that it ranked sixth after more general editing tasks such as copy and paste [Murphy et al., 2006, Robbes and Lanza, 2008]. This not only indicates how common the feature is but also how actively it is used among programmers. Refactoring tools also try to reduce typing during restructuring of source code. While refactorings maintain functionality, they are supposed to increase maintainability, the level of abstraction and reusability of code. Refactoring tools are often criticized for being underutilized [Murphy-Hill and Black, 2007]. Campbell et al. list lack of trust in automatic refactorings, bad discoverability of tools and the habit of manual coding as reasons [Campbell and Miller, 2008]. Our implementation may avoid these problems by utilising Code Actions in VS Code to display the automatic handler creation option only when it is applicable. Code Actions are used by multiple language extensions in VS Code and we assume them to be familiar to most users of the editor.

We theorized that automatic handler code generation could help in lowering the barrier of learning involved syntax. Coding errors are one predictor for frustration in students during coding assignments [Rodrigo and Baker, 2009]. Ford et al. found that lack of syntactical knowledge poses a source for frustration among programmers [Ford and Parnin, 2015]. The automatic creation of handler code could decrease coding mistakes for novel programmers which may lead to a lower rate of frustration while learning the concept of effects and handlers.

3.2 Outlined IDE features

3.2.1 Insert type and effects

This feature automatically writes the inferred type and effects of implicitly typed functions into the source code, making the type and effects explicit. We think that this feature could help in avoiding accidental introduction of unwanted effects. There

is one implication involved in our argumentation: we derived this feature from the presence of a type system that allows type inference. One may argue that type inference allows a statically typed language to feel more like a dynamically typed one. Meijer and Drayton argue in the exact same way [Meijer and Drayton, 2004]. Their reasoning includes that explicit typing often means extra work in order to state the obvious but in some cases a programmer definitely wants the benefits of a static type system. They conclude that dynamic and static typing both offer benefits and may co-exist in programming languages. Some attempts exist to implement both static and dynamic typing in the same language [Abadi et al., 1991, Ortin et al., 2010].

One can imagine to automatically apply this feature to every function definition in a project. In this way, all effects would be "fixed" and the project would be secured against accidentally introduced effects. The feature would avoid the tedious work of setting each functions type and effects manually. As a benefit the code may gain readability: we mentioned in subsection 3.1.1 that the availability of missing type and effect information may improve code comprehension in readers. The Metals[1] language server for Scala implements a similar feature called *insert type annotation* to add explicit typing to any implicitly typed expression.

3.2.2 Display effects as requirements and handled effects as arguments

The underlying intent of these features is to transport the way the original authors of Effekt tried to look at effects and handlers: effects pose requirements to their surrounding environment and handlers dynamically offer implementations to fulfill these requirements. Effects are seen as capabilities. Due to the direct relation to the Effekt language, this feature may not be transferable to other languages that offer effects and handlers. However, the general idea of transporting semantics in non-editable text blocks woven into the source code could be interesting for other languages with effects and handlers and other language concepts in general. The idea of enriching source code with secondary information is not new: source code comments have existed in COBOL since at least the 1960s [Sammet, 1978]. Additional information accompanying the source code is considered both important for understanding the program and problematic due to the lack of formalism; for example, code comments provide the flexibility of natural language to express intentions, but could be vulnerable to misinterpretation [Van De Vanter, 2002]. Using formal language constructs to express intent or meaning in code-accompanying content could prevent misinterpretation. Enriching the programming environment with such constructs could be a way to keep computer programs free from natural language while providing the programmer with supporting information.

[1] https://scalameta.org/metals/

3.2.3 List handlers of an effect

Reading and understanding of source code is considered a crucial aspect of software development [Raymond, 1991, Busjahn and Schulte, 2013, Busjahn et al., 2014]. We assume that it can be of benefit to a programmer to see exemplary handler implementations for an effect at hand, especially when reading foreign source code. Exemplary code is considered helpful during programming [Zagalsky et al., 2012, Nasehi et al., 2012]. This leads us to the following conclusions. By allowing a programmer to quickly find all handler implementations of an effect, we might be able to provide better insight into what the effect is supposed to express. We assume that algebraic effects in code are harder to grasp than ordinary functions because their definition and implementation are separated from each other. In addition to that, an algebraic effect may have multiple handler implementations whereas a function is only defined once. The intention behind an algebraic effect can be partially expressed in its name and the names and types of its effect operations. However, an implementation always provides additional insight. If a handler implementation is defined somewhere in the source code, there is no reason not to provide the reader with this additional source of insight when she needs it.

3.3 Limitations

The implemented features were limited by the design of the LSP and VS Code's extensions API. We made use of LSP's *workspace/executeCommand* command to request information that is not defined by any command in the LSP standard. Therefore the demonstrated features do not comply with the LSP and will inly work with specially designed clients. Visualisation and interaction with the requested information was limited by the APIs of VS Code. For example, inline text decorations used for the *inferred type and effect decoration* are not interactive. If they were so, the *effect origin and binder hint* feature could have been implemented with less visual clutter by not using CodeLenses.

There is a limitation in the transferability of the features presented. The Effekt language was developed with a close relationship between effects and their binding sites in mind. In the Effekt compiler the binding sites of each effect are annotated. This makes accessing that information for the *effect origin and binder hint* feature very easy. Other languages may be designed differently, making the implementation of similar features more difficult.

3.4 Conclusion

We discussed both the concept of effects and handlers and IDE language support features. While considering related work, we found that research in the intersection of these topics is basically non-existent. We theorized that a novel language feature like effects and handlers allows for novel IDE language support features. Furthermore, we assumed that such IDE features can help in understanding effects and handlers and programming with them. We presented several ideas for such novel IDE features and implemented four of them. While discussing these features, we have found that they share similarities with other common IDE features. Scientific knowledge on the usefulness of other features may give an indication of the usefulness of our proposals.

We conclude that effects and handlers are a novel language concept and allow for novel IDE features. These features could be useful in understanding and programming with effects and handlers. Further research is needed to ascertain the usefulness of our proposals.

4 Bibliography

[Abadi et al., 1991] Abadi, M., Cardelli, L., Pierce, B., and Plotkin, G. (1991). Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268.

[Bauer and Pretnar, 2015] Bauer, A. and Pretnar, M. (2015). Programming with algebraic effects and handlers. *Journal of logical and algebraic methods in programming*, 84(1):108–123.

[Beelders and du Plessis, 2016a] Beelders, T. and du Plessis, J.-P. (2016a). The influence of syntax highlighting on scanning and reading behaviour for source code. In *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 1–10.

[Beelders and du Plessis, 2016b] Beelders, T. R. and du Plessis, J.-P. L. (2016b). Syntax highlighting as an influencing factor when reading and comprehending source code. *Journal of Eye Movement Research*, 9(1).

[Biernacki et al., 2019] Biernacki, D., Piróg, M., Polesiuk, P., and Sieczkowski, F. (2019). Binders by day, labels by night: effect instances via lexically scoped handlers. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–29.

[Brachthauser and Leijen, 2019] Brachthauser, J. and Leijen, D. (2019). Programming with implicit values, functions, and control (or, implicit functions: Dynamic binding with lexical scoping). Technical Report MSR-TR-2019-7, Microsoft. Submitted to ICFP'19.

[Brachthäuser et al., 2018] Brachthäuser, J. I., Schuster, P., and Ostermann, K. (2018). Effect handlers for the masses. *Proc. ACM Program. Lang.*, 2(OOPSLA).

[Brachthäuser et al., 2020a] Brachthäuser, J. I., Schuster, P., and Ostermann, K. (2020a). Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30.

[Brachthäuser et al., 2020b] Brachthäuser, J. I., Schuster, P., and Ostermann, K. (2020b). Effekt: Lightweight effect polymorphism for handlers. Technical report, Technical Report. University of Tübingen, Germany.

[Bruch et al., 2009] Bruch, M., Monperrus, M., and Mezini, M. (2009). Learning from examples to improve code completion systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT*

Symposium on The Foundations of Software Engineering, ESEC/FSE '09, page 213–222, New York, NY, USA. Association for Computing Machinery.

[Busjahn et al., 2014] Busjahn, T., Bednarik, R., and Schulte, C. (2014). What influences dwell time during source code reading? analysis of element type and frequency as factors. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 335–338.

[Busjahn and Schulte, 2013] Busjahn, T. and Schulte, C. (2013). The use of code reading in teaching programming. In *Proceedings of the 13th Koli Calling international conference on computing education research*, pages 3–11.

[Campbell and Miller, 2008] Campbell, D. and Miller, M. (2008). Designing refactoring tools for developers. In *Proceedings of the 2nd Workshop on Refactoring Tools*, pages 1–2.

[Chen et al., 2021] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

[Cooper et al., 2006] Cooper, E., Lindley, S., Wadler, P., and Yallop, J. (2006). Links: Web programming without tiers. In *International Symposium on Formal Methods for Components and Objects*, pages 266–296. Springer.

[Dillon et al., 2012] Dillon, E., Anderson, M., and Brown, M. (2012). Comparing feature assistance between programming environments and their" effect" on novice programmers. *Journal of Computing Sciences in Colleges*, 27(5):69–77.

[Dyke, 2011] Dyke, G. (2011). Which aspects of novice programmers' usage of an ide predict learning outcomes. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 505–510.

[Eisenstadt and Lewis, 2018] Eisenstadt, M. and Lewis, M. W. (2018). Errors in an interactive programming environment: Causes and cures. In *Novice Programming Environments*, pages 111–131. Routledge.

[Fan et al., 2019] Fan, H., Li, K., Li, X., Song, T., Zhang, W., Shi, Y., and Du, B. (2019). Covscode: A novel real-time collaborative programming environment for lightweight ide. *Applied Sciences*, 9(21).

[Fischer and Hanenberg, 2015] Fischer, L. and Hanenberg, S. (2015). An empirical investigation of the effects of type systems and code completion on api usability using typescript and javascript in ms visual studio. *ACM SIGPLAN Notices*, 51(2):154–167.

[Ford and Parnin, 2015] Ford, D. and Parnin, C. (2015). Exploring causes of frustration for software developers. In *2015 IEEE/ACM 8th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 115–116. IEEE.

[Forster et al., 2017] Forster, Y., Kammar, O., Lindley, S., and Pretnar, M. (2017). On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.*, 1(ICFP).

[Hannebauer et al., 2018] Hannebauer, C., Hesenius, M., and Gruhn, V. (2018). Does syntax highlighting help programming novices? *Empirical Software Engineering*, 23(5):2795–2828.

[Häregård and Kruger, 2019] Häregård, E. and Kruger, A. (2019). Comparing syntax highlightings and their effects on code comprehension.

[Heinonen et al., 2014] Heinonen, K., Hirvikoski, K., Luukkainen, M., and Vihavainen, A. (2014). Using codebrowser to seek differences between novice programmers. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 229–234.

[Jones and Burnett, 2007] Jones, S. J. and Burnett, G. E. (2007). Spatial skills and navigation of source code. *ACM SIGCSE Bulletin*, 39(3):231–235.

[Kelleher and Pausch, 2005] Kelleher, C. and Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2):83–137.

[Leijen, 2014] Leijen, D. (2014). Koka: Programming with row polymorphic effect types. *arXiv preprint arXiv:1406.2061*.

[Lüth and Ghani, 2002] Lüth, C. and Ghani, N. (2002). Composing monads using coproducts. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, page 133–144, New York, NY, USA. Association for Computing Machinery.

[Mader and Egyed, 2011] Mader, P. and Egyed, A. (2011). Do software engineers benefit from source code navigation with traceability? – an experiment in software change management. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, page 444–447, USA. IEEE Computer Society.

[Masci and Munoz, 2019] Masci, P. and Munoz, C. A. (2019). An integrated development environment for the prototype verification system. *arXiv preprint arXiv:1912.10632*.

[Medeiros et al., 2018] Medeiros, R. P., Ramalho, G. L., and Falcão, T. P. (2018). A systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education*, 62(2):77–90.

[Meijer and Drayton, 2004] Meijer, E. and Drayton, P. (2004). Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. Citeseer.

[Meyerovich and Rabkin, 2013] Meyerovich, L. A. and Rabkin, A. S. (2013). Empirical analysis of programming language adoption. *SIGPLAN Not.*, 48(10):1–18.

[Microsoft Corporation, 2021] Microsoft Corporation (2021). What is the language server protocol? `https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/`, accessed on 2022-01-18.

[Moggi, 1991] Moggi, E. (1991). Notions of computation and monads. *Information and computation*, 93(1):55–92.

[Murphy et al., 2006] Murphy, G. C., Kersten, M., and Findlater, L. (2006). How are java software developers using the elipse ide? *IEEE software*, 23(4):76–83.

[Murphy-Hill and Black, 2007] Murphy-Hill, E. R. and Black, A. P. (2007). Why don't people use refactoring tools? In *WRT*, pages 60–61.

[Nasehi et al., 2012] Nasehi, S. M., Sillito, J., Maurer, F., and Burns, C. (2012). What makes a good code example?: A study of programming q&a in stackoverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 25–34. IEEE.

[Németh and Brunner, 2019] Németh, B. and Brunner, T. (2019). Haskellcompass: Extending the codecompass comprehension framework for haskell. In *2019 IEEE 15th International Scientific Conference on Informatics*, pages 000149–000154. IEEE.

[Ortin et al., 2010] Ortin, F., Zapico, D., Pérez-Schofield, J. B. G., and Garcia, M. (2010). Including both static and dynamic typing in the same programming language. *IET software*, 4(4):268–282.

[Parker et al., 2006] Parker, K. R., Ottaway, T. A., and Chao, J. T. (2006). Criteria for the selection of a programming language for introductory courses. *International Journal of Knowledge and Learning*, 2(1-2):119–139.

[Pigott, 2020] Pigott, D. (2020). Online historical encyclopaedia of programming languages.

[Plotkin and Power, 2001] Plotkin, G. and Power, J. (2001). Adequacy for algebraic effects. In *International Conference on Foundations of Software Science and Computation Structures*, pages 1–24. Springer.

[Plotkin and Power, 2003] Plotkin, G. and Power, J. (2003). Algebraic operations and generic effects. *Applied categorical structures*, 11(1):69–94.

[Plotkin and Pretnar, 2009] Plotkin, G. and Pretnar, M. (2009). Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer.

[Rask et al., 2021] Rask, J. K., Madsen, F. P., Battle, N., Macedo, H. D., and Larsen, P. G. (2021). Visual studio code vdm support. *John Fitzgerald, Tomohiro Oda, and Hugo Daniel Macedo (Editors)*, page 35.

[Raymond, 1991] Raymond, D. R. (1991). Reading source code. In *Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research*, pages 3–16.

[Robbes and Lanza, 2008] Robbes, R. and Lanza, M. (2008). How program history can improve code completion. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 317–326. IEEE.

[Rodrigo and Baker, 2009] Rodrigo, M. M. T. and Baker, R. S. (2009). Coarse-grained detection of student frustration in an introductory programming course. In *Proceedings of the fifth international workshop on Computing education research workshop*, pages 75–80.

[Sammet, 1978] Sammet, J. E. (1978). The early history of cobol. In *History of Programming Languages*, pages 199–243.

[Sarkar, 2015] Sarkar, A. (2015). The impact of syntax colouring on program comprehension. In *PPIG*, page 8.

[Schrijvers et al., 2019] Schrijvers, T., Piróg, M., Wu, N., and Jaskelioff, M. (2019). Monad transformers and modular algebraic effects: What binds them together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, page 98–113, New York, NY, USA. Association for Computing Machinery.

[Sloane, 2008] Sloane, T. (2008). Experiences with domain-specific language embedding in scala. In *Domain-specific program development*, page 7.

[Svyatkovskiy et al., 2019] Svyatkovskiy, A., Zhao, Y., Fu, S., and Sundaresan, N. (2019). Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*, KDD '19, page 2727–2735, New York, NY, USA. Association for Computing Machinery.

[Szalay et al., 2018] Szalay, R., Porkoláb, Z., and Krupp, D. (2018). Symbol clustering: Resolving ambiguous symbol references of large-scale c/c++ projects based on linkage information.

[Van De Vanter, 2002] Van De Vanter, M. L. (2002). The documentary structure of source code. *Information and Software Technology*, 44(13):767–782. Special Issue on Source Code Analysis and Manipulation (SCAM).

[Vihavainen et al., 2014] Vihavainen, A., Helminen, J., and Ihantola, P. (2014). How novices tackle their first lines of code in an ide: Analysis of programming session traces. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, pages 109–116.

[Zagalsky et al., 2012] Zagalsky, A., Barzilay, O., and Yehudai, A. (2012). Example overflow: Using social media for code recommendation. In *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 38–42. IEEE.

[Zhang et al., 2019] Zhang, X., Jiang, Y., and Wang, Z. (2019). Analysis of automatic code generation tools based on machine learning. In *2019 IEEE International Conference on Computer Science and Educational Informatization (CSEI)*, pages 263–270. IEEE.