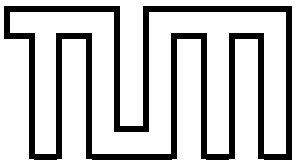


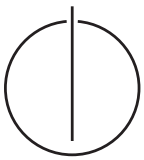
DEPARTMENT OF INFORMATICS

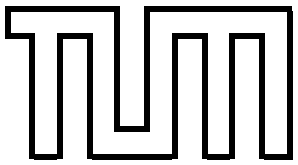
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

Database Code Generation Explorer

Simon Kapfberger





DEPARTMENT OF INFORMATICS

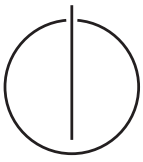
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

Database Code Generation Explorer

Visualisierung von dynamisch erzeugtem Datenbankencode

Author:	Simon Kapfberger
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Maximilian Bandle, M.Sc.
Submission Date:	December 15, 2020



I confirm that this bachelor's thesis in information systems is my own work and I have documented all sources and material used.

Munich, December 7, 2020 Simon Kapfberger

Acknowledgments

I would like to express my gratitude to my advisor Maximilian Bandle and my supervisor Prof. Neumann for the opportunity to work on this interesting topic. Furthermore, I would like to thank Max for the thesis template and his continuos support. Special thanks to my dad Ralf for providing ideas, proofreading and his encouragement throughout the whole process of writing.

Abstract

The Umbra database system uses the Umbra Intermediate Representation for efficient query compilation. Especially for large requests, the data layout of the generated UmbraIR can be expansive. This makes it difficult to understand the logical structure of the code and complicates the debugging process.

In the context of this thesis, an explorer application is developed that is optimized to display UmbraIR code. The UIR-Editor is a visualization tool for UmbraIR that supports code interaction through intuitive routines. The primary goal of the program is to make UmbraIR more understandable by providing helpful features. The UIR-Editor focuses on increasing the productivity for analysis and debugging workflows. Therefore the application presents UmbraIR visually and improves transparency by showing context information. In addition, convenient navigation features make the code more accessible.

This thesis describes the basic structure of UmbraIR upon which the UIR-Editor is built. The implemented features of the tool are first documented and then applied in example use case scenarios.

Kurzfassung

Das Datenbanksystem Umbra verwendet die Umbra Intermediate Representation für eine effiziente Abfragekompilierung. Insbesondere bei großen Anfragen kann das Datenlayout des generierten UmbraIR sehr umfangreich sein. Dies macht es schwierig, die logische Struktur des Codes zu verstehen und erschwert den Debugging-Prozess.

Im Rahmen dieser Arbeit wird eine Explorer-Anwendung entwickelt, die für die Darstellung von UmbraIR-Code optimiert ist. Der UIR-Editor ist ein Visualisierungstool für UmbraIR, das die Interaktion mit dem Code durch intuitive Routinen unterstützt. Das primäre Ziel des Programms ist es, UmbraIR durch hilfreiche Funktionen verständlicher zu machen. Der UIR-Editor konzentriert sich darauf, die Produktivität für Analyse- und Debugging-Workflows zu erhöhen. Daher stellt die Anwendung UmbraIR visuell dar und verbessert die Transparenz durch das Anzeigen von Kontextinformationen. Darüber hinaus machen komfortable Navigationsfunktionen den Code leichter zugänglich.

Diese Arbeit beschreibt die Grundstruktur von UmbraIR, auf der der UIR-Editor aufbaut. Die implementierten Funktionen des Tools werden zunächst dokumentiert und dann in Beispielszenarien angewendet.

Contents

Acknowledgments **v**

Abstract **vii**

Kurzfassung **ix**

1. Introduction **1**
1.1. Motivation and Goal 2
1.2. Thesis Structure 3

2. Background: Umbra Intermediate Representation **5**
2.1. Component 6
2.1.1. Global Variable 6
2.1.2. Function Declaration 7
2.1.3. Function Definition 7
2.2. Block 7
2.3. Instruction 8
2.3.1. Assignment 8
2.3.2. Control Flow 8
2.3.3. Other 9
2.4. Operand 9
2.4.1. Constant 10
2.4.2. Variable 10
2.4.3. Target 10

3. Implementation: UIR-Editor **11**
3.1. Editor 13
3.2. Language 14
3.3. Data 14
3.4. Content 15
3.5. Interface 16
3.6. App 17

4. UIR-Editor: Features **19**
4.1. User Interface . . . 19
4.1.1. Query Dropdown . . . 19
4.1.2. Status Display . . . 20
4.1.3. Input Bar . . . 20
4.1.4. Keyboard Button . . . 20
4.2. Code Editor . . . 21
4.3. Syntax Highlighting . . . 21
4.4. Basic Navigation . . . 22
4.5. Node Selection . . . 22
4.6. Advanced Navigation . . . 23
4.7. Variable Flow . . . 24
4.8. Query Annotation . . . 25
4.8.1. Bookmark . . . 25
4.8.2. Note . . . 25
4.8.3. Comment . . . 26
4.8.4. Rename . . . 27

5. UIR-Editor: Use Cases **29**
5.1. Code Presentation . . . 29
5.2. Code Analysis . . . 31
5.2.1. Variable Assignment . . . 31
5.2.2. Control Flow . . . 34
5.3. Generator Code Comparison . . . 36
5.4. Finding Keywords . . . 40

6. Conclusion & Future Work **41**

A. Appendix **43**
A.1. Keyboard Shortcuts . . . 43
A.2. Keyboard Shortcuts - Legend . . . 44
A.3. Hash Function Annotations . . . 45

List of Figures **51**

List of Tables **53**

Bibliography **55**

1. Introduction

Databases are the backbone of the growing infrastructure of digital systems and services we use today. Most applications use some kind of database system as their foundation in order to store information. For example, modern technologies like machine learning and data-mining rely on large amounts of processed data to create new insights. Services might use different approaches, but they all rely on the database to handle requests at reasonable speeds. Therefore the factors *data volume* and *request speed* are crucial for database usability. If data volume and/or reading and writing speed can be improved, it expands the horizon for better services in the future. As Moore's law suggests, it can be expected, that microprocessing power and data storage volume will advance further [20]. To support this development, database systems have to be able to handle a growing amount of data effectively. Consequently working processes need to be continuously optimized for new storage technology innovations in order to pave the way for technological advances.

To meet this demand, *Umbra* [11] provides an environment to implement and test new ways of efficiently processing database requests [16]. Generally databases like Umbra have multiple options when it comes to efficient query execution: Traditionally an *interpreter* is used to translate the query into a physical plan of multiple algebraic expressions [21]. This plan is then iteratively executed by the bytecode interpreter inside of a virtual machine (VM). Each individual instruction needs to be fetched and processed by the VM. As a result interpretation time is higher the more iterations have to be handled. Therefore this approach archives optimal performance for short-running queries and small data sets. Alternatively a *compiler* can convert the query into a machine code program that can be interpreted more efficiently. This step requires the database system to wait for the compiler to start up and finish the compilation. As a result, compilation time is usually significantly higher than the interpretation time of the compiled program. Therefore this approach archives optimal performance for long-running queries and large data sets. Umbra is built to combine the benefits of both approaches. It implements an adaptive execution framework that makes use of efficient query compilation as a basis and a VM to reduce query latency [17]. Umbra first creates a query plan from the users SQL request through efficient interpretation. After that, the request is translated into an equivalent program representation before creating the executable that calculates the query result. The generated code is called

Lines of code (UIR)	Operators (SQL)	Lines per Operator
1353,36	9,64	140,44

Table 1.1.: Average UIR code lines per SQL operator from 22 TPC-H requests

Umbra Intermediate Representation (UmbraIR/UIR). Umbra uses the efficient UIR-code to optimize internal processes. UmbraIR is either used for interpretation inside the VM or compiled for assembly by one of Umbra's two compiler backends [16]. This means UmbraIR is the last common language in the Umbra stack. Therefore it is highly important to improve and debug UIR-code to increase performance.

This Thesis aims to help users of Umbra to better understand the Umbra program representation language. It provides a reference point for the basic structure of UmbraIR. Furthermore, it focuses on the documentation of the associated software project called the UIR-Editor. The editor's many features were developed alongside this thesis to enhance the users understanding of UIR-code and make it more accessible.

1.1. Motivation and Goal

Analyzing raw UIR-code can be challenging for novice as well as experienced users. The TPC-H queries compiled in Umbras program representation average around 1400 lines of code. The underlying SQL requests average around 10 operators per query. Therefore one SQL operator is usually translated to around 140 lines of code in UmbraIR (Table 1.1). The code logic of a single SQL operation is spread across the UIR-code over many lines. This makes it difficult to understand the logical structure of large UmbraIR programs. Due to the overall query size, UIR-code can also be convoluted and difficult to traverse. So far available code editors are not optimized and do not offer functionality to navigate expansive UmbraIR queries effectively. The best a user can do currently, is to simply scroll through the code line by line, which makes it hard to recognize important connections. This can lead to unrecognized mistakes, for example by mixing up similar values when tracing the control flow. Additionally, there are no tools available to increase perception of context or detail awareness. This significantly increases the time spent by the user when looking for specific code parts. These factors are problematic for users of Umbra trying to understand or debug the generated UIR-code and may lead to a bad experience.

That is why in the context of this thesis a code visualization tool - *UIR-Editor* - is implemented, specifically tailored to the UmbraIR language. It is designed to help novice users to get familiar with UmbraIR and help experienced users to get insights faster. The UIR-Editor's features enable the user to explore the details of UmbraIR

intuitively and makes it much easier to work with UIR-code. All of the features aim to improve the user experience, make the code more understandable and increase productivity. UIR-Editor delivers UmbraIR in colors that represent the code's logic. It provides context based information and quick content navigation, following routines that are familiar to the user. The UIR-Editor highlights related code parts and enables the user to interact with the code, while avoiding the risk of unwanted side effects. The goal of this thesis is to give insights into the most important aspects of UmbraIR, document the UIR-Editor's features and show how to use it optimally.

1.2. Thesis Structure

Chapter 2, *Background: Umbra Intermediate Representation*, lays the foundation for this thesis by presenting the underlying logical structure of UmbraIR. The Sections explain each of the different building blocks of the data layout.

Chapter 3, *UIR-Editor: Implementation*, documents how the software is implemented by covering the most important classes and interfaces as well as their relationships.

Chapter 4, *UIR-Editor: Features*, presents all the implemented features of the UIR-Editor. The Sections each highlight related features and explain how to use them.

Chapter 5, *UIR-Editor: Use Cases*, applies the features described in Chapter 4 and combines them in practical use cases. Building on the insights about UmbraIR presented in Chapter 2, these use cases show different application scenarios of the UIR-Editor.

Chapter 6, *Conclusion & Future Work*, ends the thesis with a retrospection, adresseses the softwares limitations and offers some ideas for future work.

2. Background: Umbra Intermediate Representation

The *Umbra* [11] database system uses a custom intermediate program representation (IR). It constitutes the interim step between the code generator, that interprets the users request, and the compilation backend, that creates the machine language program. UmbraIR is a subset of the popular *LLVM IR* [18]. In order to improve performance, it contains a similar and less generalized operation library with minimal overhead of unused code. UIR is optimized for fast reading and writing speeds. It includes only the basic functionality necessary to support Umbra's requests [22]. The data structures are designed to make code generation and compilation quicker by reducing the layout to be as compact as possible. This improves query latency across the compilation pipeline and helps with memory allocation, but comes at the cost of loosing some flexibility [16]. Similar to LLVM IR, the static single assignment form (SSA) is used. Therefore variables in UmbraIR can only be assigned once, to simplify compiler optimizations.

On a basic level UmbraIR operates by defining function components that contain a number of successive code blocks. These blocks represent the underlying requests program logic through sequences of predefined instructions [16]. In this Chapter the basic structure of the Umbra Intermediate Representation language is documented in detail. The associated UIR-Editor project implements the UmbraIR structure as classes to provide content-based functionality and features. Each of the following sections covers the relevant classes in the UML diagram - Figure 2.2 - and provides examples for the UIR-code syntax in reference to the query in Figure 2.1.

```
1  const %698[8] = "\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000"
2
3  declare void @umbra::Dummy::init(umbra::Dummy*, void const*)(object umbra::Dummy* %807, int8* %821)
4
5  define int32 @_123_dummyStep(int8* %trampoline, int8* %localState) [] {
6  body:
7    %7429 = getelementptr int8 %localState, i64 32
8    call void umbra::Dummy::init(umbra::Dummy*, void const*) (%7429, global %698)
9    return 0
10 }
```

Figure 2.1.: Umbra Intermediate Representation: Reference Query

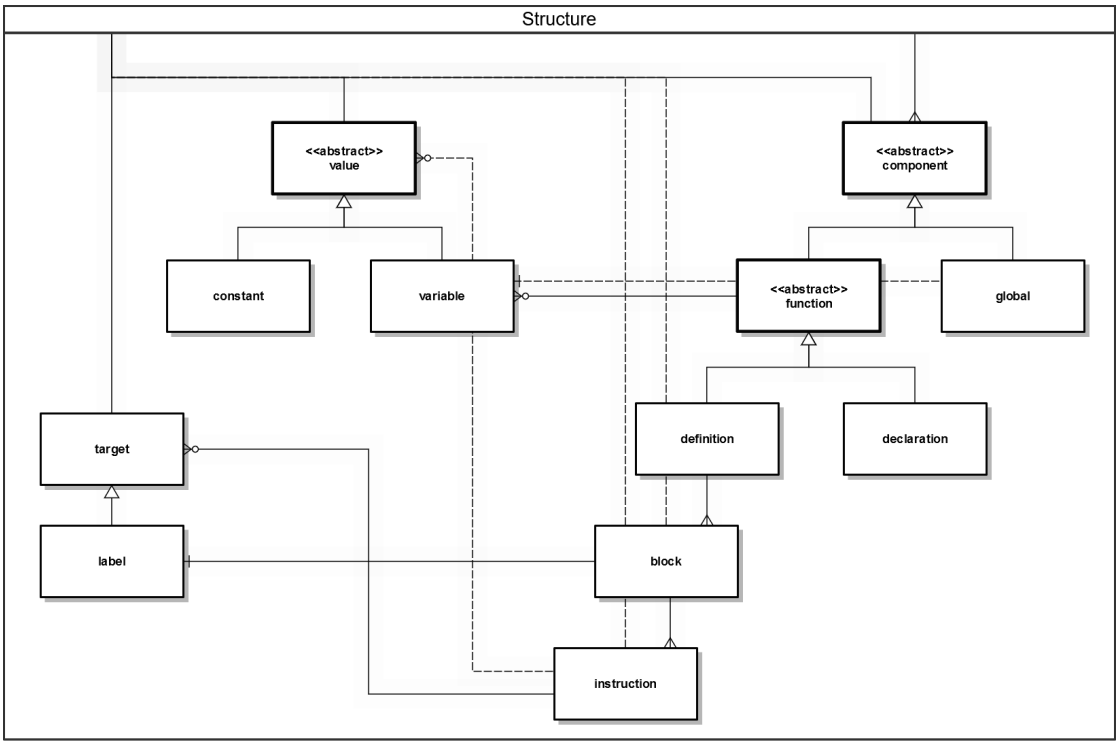
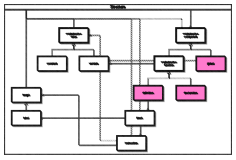


Figure 2.2.: Umbra Intermediate Representation: Language Structure

2.1. Component



On the highest level each UmbraIR query consists of a number of successive *components*. The UIR-query in Figure 2.1 contains one component of each class. At the start of any query the *global variables* are initialized (line 1). The rest of the query consists of *function declarations* (line 3) and *function definitions* (line 5-10).

2.1.1. Global Variable

Umbra IR uses *global variables* to improve memory allocation by avoiding redundant data storage [16]. The initialization of a *global* is expressed with the *const* statement across a single line of code. The size of the data stored is specified in square braces after the *variable* (Section 2.4.2) name. This is followed by the assignment operator = and the stored data formatted in unicode [3] surrounded by quotes.

```
const <variable>[<size>] = "<data>"
```

2.1.2. Function Declaration

In order to reference and call an external function in an UmbraIR query, it first needs to be declared. Functions can be declared anywhere within the query, even after being called by another instruction, because the whole query will be available to the compiler at runtime. The *declaration* of an external function is expressed with a *declare* statement across a single line of code. After that, the return type of the function is specified. Next is the function character @ followed by the function's name. Two sets of braces conclude a *declaration*. The first set specifies the types of arguments that are accepted by the external function. The second set contains the applicable UmbraIR argument types and variables for the first set.

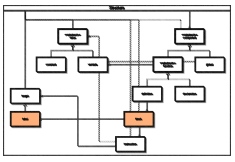
```
declare <type> @<function>(<external types>)(<argument types & variables>)
```

2.1.3. Function Definition

Definitions contain multiple *basic blocks* (Section 2.2), that hold the generated code and make up the underlying request's logic. The first *block* after the initialization is always the entry point for the sequential execution [16]. A new *function definition* is expressed with the *define* statement. This is followed by the return type, the function character @ and the function name. Next is a single set of braces specifying the argument types and *variables* (Section 2.4.2). The square braces contain the field for the function variables, which is empty in most of the UIR-queries. In the following line the curly braces are opened for the body of the *definition*. It is made up of one or more code *blocks* and terminated with closing curly braces.

```
define <type> @<function>(<argument types & variables>) [] {
  <blocks>
}
```

2.2. Block

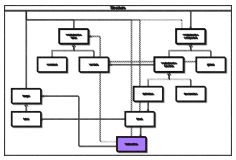


Umbra generates *basic blocks* inside of *function definitions* (Section 2.1.3) to organize *instructions* (Section 2.3). The *instructions* inside a *block* are executed sequentially. To implement control flow constructs like conditions and loops, UmbraIR can determine the execution order of the code *blocks* dynamically. Therefore each *block* inside of a *definition* can be identified by the *block label*. This *label* represents the entry point for the *basic block*. The *label* of a *block* can be targeted by *instructions* of other *blocks* using specific control flow operations. After that, the targeted *block* is executed,

which is called branching. To terminate a *basic block*, a control flow instruction must be used [16]. For example, the UIR-query in Figure 2.1 contains a single *block* spanning from line 6 to line 9. *Blocks* are represented by the *label* followed by the *:* character. A *block* ends after the last *instruction* followed by a line-break.

```
<label>:
  <instructions>
```

2.3. Instruction



Every *basic block* contains one or more *instructions* that make up the code's logic. Currently there are 104 different kinds of single line *instructions* in the UmbraIR language [16]. They are defined by the operation they execute and the corresponding code (OpCode). It determines the form of the *instruction* and which *operands* (Section 2.4) are accepted as arguments.

2.3.1. Assignment

One central aspect of UmbraIR is assigning variables and executing operations on them. *Assignment instructions* first evaluate the result of their containing operation and assign it to a new *variable*. They are used for arithmetic, loading, comparisons, casts and calls of *declared functions* [16]. For example, in line 7 of Figure 2.1 the *variable %7429* is assigned with the result of the *getelementptr* operation. First the assigned *variable* is defined at the start of the *assignment*. After that, the assignment operator = separates the *variable* on the left from the operation on the right. Next is the OpCode, followed by the return type of the operation and the *operands*.

```
<variable> = <opcode> <type> <operands>
```

2.3.2. Control Flow

UmbraIR implements control flow constructs using multiple *basic blocks* (Section 2.2) and specific *instructions*. The control flow *instructions* determine (conditionally), which *block* is executed next or if a result is returned. They are used for branches, conditional branches and returns from *defined functions* [16]. Such control flow *instructions* are located in the last line of a *basic block*. For example, line 9 of Figure 2.1 returns the value *0*. The OpCode specifies the operation to be executed. The following *operands*, specify the return value or reference the *label* of the next to execute *block*.

```
<opcode> <operands>
```

One important *instruction* for managing the control flow is the *PHI node*. It is used due to the SSA property of the UmbraIR language. Umbra generates *PHI nodes* as a substitute for multiple variable assignment, for example to translate loops into UIR. The *PHI instruction* is always located at the beginning of a *basic block*. It chooses its value from the provided argument values depending on which *block* was executed prior to the *PHI instruction*. Then a new *variable* (Section 2.4.2) is assigned, which contains the resulting value [16]. The *PHI instruction* is represented similar to the previously described *assignment* syntax. The *operands* are enclosed in square brackets and the possible argument values are listed, each followed by their destination *label*.

```
<variable> = phi <type> [<value>, <label> <value1>, <label1> ...]
```

2.3.3. Other

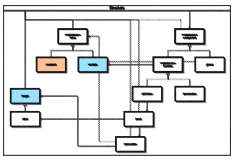
Some UmbraIR *instructions* are neither used to assign new *variables* nor to manage the control flow. For example, the *store* and *atomicstore instructions* are used to modify memory and write a value to a specific register.

```
store <type> <value>, <pointer>
```

Another unique *instruction* is *call*, which can be used to execute an external procedure. This allows Umbra to keep the UIR-code efficient by executing complex parts of the request's logic externally. For example, the *call instruction* in line 8 of Figure 2.1 calls the external function previously *declared* in line 3.

```
call void <function>(<external types>) (<operands>)
```

2.4. Operand



At the root of the UmbraIR code's structure are the *operands*, which are processed by the *instructions* (Section 2.3) to implement the program logic. *Operands* can be categorized into the following classes: *Values* can be basic *constants* of a specific data type or registers containing SSA *variables* representing a previously calculated result. Registers can be used to represent the *label* of a *basic block* (Section 2.2). These *block targets* appear as *operands* in branching control flow *instructions*.

2.4.1. Constant

The most basic *operand* is the *constant* value. *Constants* are usually integers, but can also represent bool values or simple objects. Arithmetic *instructions* can use *constants* to calculate a result. *PHI instructions* can implement conditions using a boolean *constant*. *Constants* can specify positions in memory to access and store information. For example, in line 7 of Figure 2.1 a 64-bit integer *constant* value is used to specify the register for the *getelementptr instruction*. *Constant* values always follow after their type specifier.

```
<type> <constant value>
```

2.4.2. Variable

UmbraIR follows the SSA form. Therefore *variables* can only be assigned once [16]. In this context the term "variable" is not used in the traditional sense. Here it refers to a register in memory containing a value. As a result multiple related *variables* are assigned to conform to the SSA property. Every new *variable* contains an incremental result of the code's logic. The *variables* that are used in *instructions* leading up to the current *variable's* value are called *Parents*. All following *variables*, that are assigned by an operation containing the current *variable* value, are called *Children*. *Variables* appear in all *instruction* types except for simple control flow operations. For example, in line 8 of Figure 2.1 the *variable %7429* is used as an *operand* for the *call instruction*. Assigned *variables* adopt the type specified by the assignment operation. Argument *variables* of *function declarations*, *definitions* and *global variables* include a type specifier. A *variable* can be identified in the code by the register character %.

```
%<variable name>
```

2.4.3. Target

Every *block* inside a *function definition* can be referenced by the unique *block label* to direct the control flow. Due to the SSA property of UmbraIR, registers targeting *block labels* are used to reuse code *blocks* and avoid redundancy. For this purpose *targets* with the same name as the *block labels* are used. To trace the control flow of a complex UmbraIR function, the *targets* at the end of each *block* and inside the *PHI node* must be considered. A *target* operand appears similar to a *variable*, as it is also identified by the register character %. *Targets* can be differentiated from *variables* by the *instruction* context. They always refer to one of the available *block labels* inside the current *definition*. Therefore only *definitions* with multiple *blocks* contain *targets*.

```
%<block label>
```

3. Implementation: UIR-Editor

The UIR-Editor web application is built for the *Umbra* [11] web interface. In order to improve compatibility and to enable simple integration, the implementation is based on frameworks that are applied by the website. Therefore the program is built upon the *React* [14] software library and written in the *TypeScript* [8] language. The UIR-Editor implements its main features on top of the basic functionality of the *Monaco Editor* [6] framework. Furthermore, the project uses the markup language *HTML* [5] and the style sheet language *CSS* [4], to manage the visual rendering. For advanced visualizations the *D3.js* [1] library is used. To run the application on a local system, the runtime environment *Node.js* [13] is required. It includes the package manager *npm* [15], which can be used to install the project's dependencies and start the UIR-Editor.

In this Chapter the implementation of the UIR-Editor web application is documented. To increase the reusability of components and make the program easier to maintain, the software design follows a modular object-orientated approach. The project can be divided into groups of classes that correspond to different parts of the UIR-Editor:

The main class of the application is the *Editor* component that renders the editor window and implements most available features that are presented in Chapter 4. The color scheme to represent the UmbraIR language syntax is defined in the *Language* classes. Before a query can be displayed, the raw UmbraIR is fetched from the Umbra webserver by the *Data* classes. After that, the query is converted into a *Content-Graph* that follows the UmbraIR structure as described in Chapter 2. The components of the *Interface* control the editor's features and display additional information. Finally, all the parts of the UIR-Editor are connected inside the *App* component.

The following Sections describe the project's architecture as well as the most important classes and interfaces. As a reference, the relationships and dependencies between the classes are shown in the UML diagram of Figure 3.1. Additionally, the relevant parts of the UML are highlighted at the start of each section.

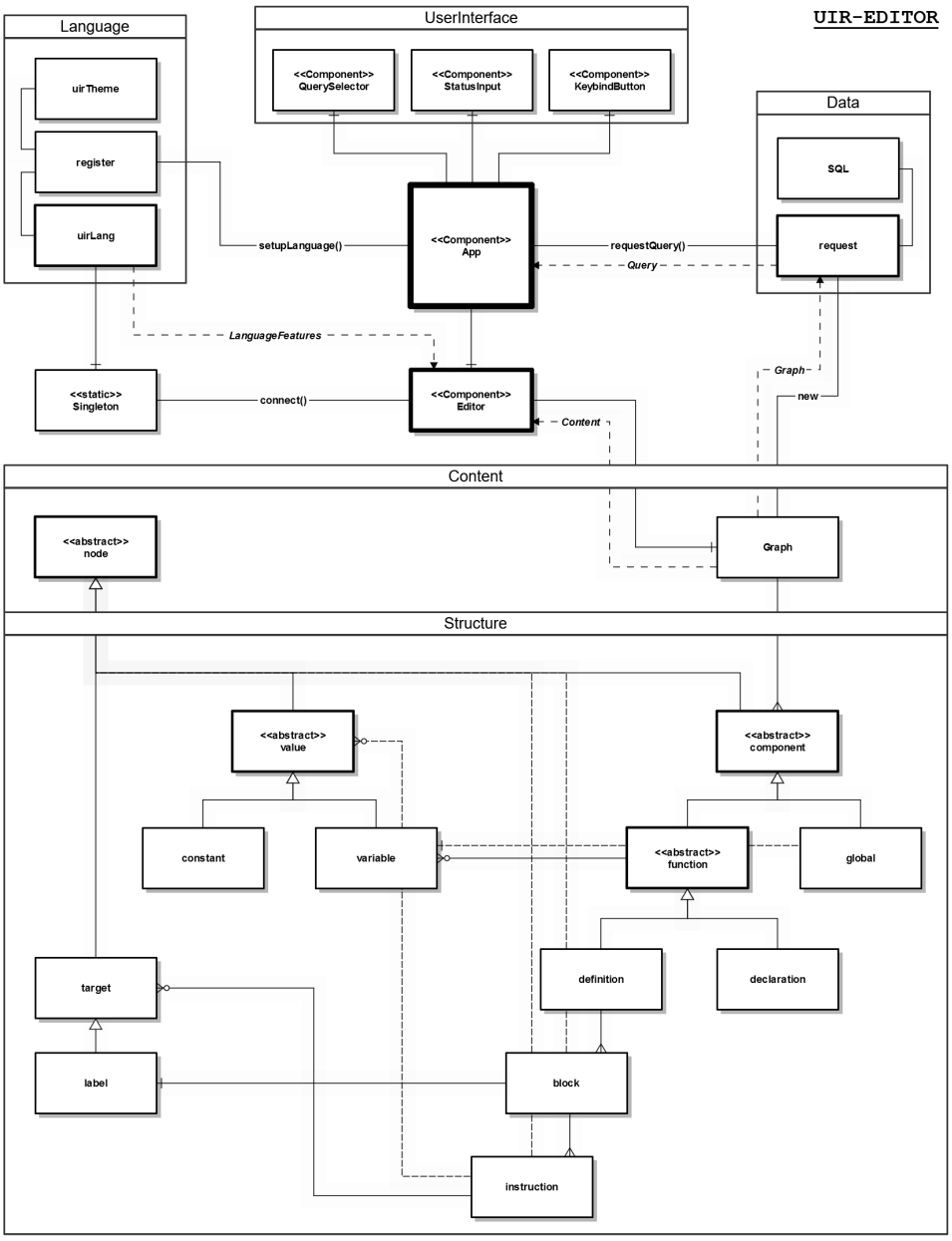
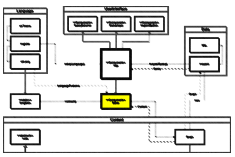


Figure 3.1.: UIR-Editor: Project Architecture

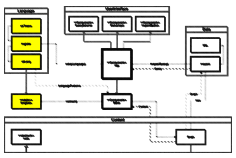
3.1. Editor



The *Monaco Editor* [6] framework is the foundation of the *Editor* class. It includes basic text editor functionality, supports custom languages, enables text *Decorations* and handles *Keyboard Shortcuts*. The implemented editor is a *React* component that imports the *Monaco* API and builds on it. It adds support for the UmbraIR *Language* (Section 3.2) and manages the features shown in Chapter 4. Once the *App* component (Section 3.6) is rendered, the representation of the currently loaded UmbraIR is the passed to the editor. The *Content-Graph* (Section 3.4) informs the editor about the content of the UIR-query. This allows the editor to implement contextual functionality based on the code's logic. Therefore it can focus on controlling the surface level with the information that the graph provides. The editor's methods apply this information to implement the features of the application. They call reusable helper-methods that manage the interaction of the current editor instance and the *Content-Graph*. Together the methods of the editor implement the following functionality:

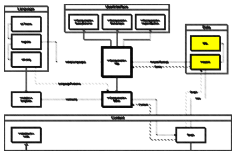
- Display the text representation of the current UmbraIR
- Assign, handle and toggle the keyboard shortcuts
- Move the cursor and handle mouse clicks
- Update the internal position to match the current *Node* (Section 3.4)
- Communicate with the *User Interface* (Section 4.1)
- Manage the *Annotations* (Section 4.8) of the graph
- Add decorations to highlight parts of the code
- Provide support for the *Language* features

3.2. Language



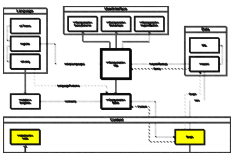
The *Language* classes add support for the UmbraIR language to the *Editor* (Section 3.1), by using the *Monarch* [7] library. The class *uirTheme* specifies the color scheme for the custom language based on the *VS-Dark* theme. Inside the *uirLang* class, the tokens for the language syntax are defined by regular expressions. Each token is assigned a color to enable *syntax highlighting* (Section 4.3). Additional language features are introduced by defining rules for hovering, highlighting and folding of the text. These features operate on the information about the displayed UmbraIR that is returned by methods of the editor class. In order apply the language features to the currently rendered editor instance, a static connection is required. Therefore a static *Singleton* is implemented to link the language and the editor. Finally, the *register* class exports the *registerLanguage()* function that is called by the *App* component (Section 3.6) to register the custom language in the environment. This enables the syntax highlighting and makes the defined language features accessible in the editor.

3.3. Data



The *Data* classes allow the UIR-Editor to load UmbraIR queries from the Umbra webserver [11] using predefined SQL requests. By default, the 22 TPC-H [23] requests are stored inside the string array of the *SQL* class. Additional SQL requests can be directly appended to the array. The requests are managed by the *request* class that exports the asynchronous *requestQuery(index)* function. It is called by the *App* (Section 3.6) component on startup of the application and every time a new UIR-query is selected. Before rendering other components, the App displays a loading animation and waits for the request to be resolved. During this time, a POST-request is sent to the Umbra webserver that contains the value of the SQL array at the specified index. If the connection to the webserver is successful, the UmbraIR is fetched in raw *.json* [10] format. Otherwise local files are loaded to prevent runtime errors. After that, a new *Graph* (Section 3.4) representation is created from the resulting raw UmbraIR. Finally, the result is returned to the App component and passed to the *Editor* for display.

3.4. Content

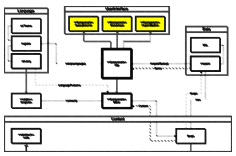


The UIR-Editor implements dynamic features that depend on the content of each individual UmbraIR query. To represent queries in the project, the *Strcuture* classes implement the different types of nodes. These match the UmbraIR classes described in Chapter 2. The *Content* classes create and manage the *Content-Graph* data structure, which is based on the nodes of the *Strcuture* classes. The *Graph* class implements methods to iterate over the connected nodes and informs the *Editor* (Section 4.2) about the content of the associated UIR-query. A new graph instance is built from the *.json* [10] file that is fetched by the *request* class (Section 3.3). When the graph is built, it creates multiple nodes that contain one unit of the UmbraIR query each.

Node The abstract *Node* class is inherited by every *Structure* class. It defines the basic attributes to identify a node and to store the necessary information. Methods are defined to manage the node's attributes and to provide information to the graph. Every node subclass modifies the inherited methods to represent each structural component. For example, *toString()* is an abstract method that is implemented by every subclass. It returns a string representation of the corresponding node and all contained nodes.

Graph The *Graph* class implements the advanced functionality of the editor class. It contains all node instances that make up the UmbraIR query representation and provides the methods to iterate over them. In order to create a new graph, a *.json* file is required that contains the UmbraIR. In the constructor, the *build()* method is called that parses the provided *.json* file. It recognizes the identifying keywords for each *Component* class (Section 2.1) and creates a new node object of that class with the associated part of the *.json* file. Once all components are defined, they also call a *build()* method to parse the assigned part of the *.json* file. This process is repeated for all subclasses, until all nodes are built. Therefore the graph can reference any node in the query by iterating over the different levels of subclasses. For example, the *print()* method iteratively calls the *toString()* method of every component in the graph. The *toString()* method of each component returns its string representation that also includes calls of the *toString()* method of its subclasses. Most of the methods in the graph operate by iteratively calling related methods in the component nodes and all underlying subclasses.

3.5. Interface



The classes of the *Interface* are *React* components, that support interaction with the UIR-Editor. The user interface is rendered by the *App* component (Section 3.6) and is displayed above the main *Editor* (Section 4.2). In order to exchange data with the editor, the corresponding methods of the App are called. Together the following components make up the user interface:

QuerySelector (Feature 4.1.1)
: The *QuerySelector* dropdown can be controlled using the mouse or by a keyboard shortcut in the editor. It shows one "UIR-Query" option for every request inside the array of the *SQL* class (Section 3.3). Selecting an option in the menu changes the current array index and issues the *request* class to fetch the associated query.

StatusInput (Feature 4.1.2 & 4.1.3)
: The *StatusInput* contains two connected input fields. The first is set to read only mode, displays the current status and changes its color accordingly. The second is only accessible by the keyboard shortcuts in the editor. Depending on the status, a different method is called to pass the input string to the editor instance.

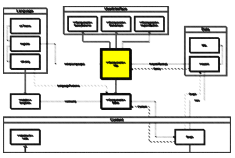
KeybindButton (Feature 4.1.4)
: The *KeyboardButton* can be toggled using the mouse or by a keyboard shortcut in the editor. It has two alternating states represented by the color of the button. If the button is green, all keyboard shortcuts of the editor are enabled. Otherwise the button is red and all keyboard shortcuts of the editor are disabled.

TargetTreeModal (Feature 4.6)
: The *TargetTreeModal* window can be displayed by a keyboard shortcut in the editor. Depending on the current *Node* in the *Content-Graph* (Section 3.4), it shows a *TargetTree* representation of the surrounding code. First a string representation of the tree is created from the data provided by the graph. Then functions of the *D3.js* [1] framework are applied to generate a graphical visualization.

InfoModal (Feature 4.5)
: The *InfoModal* window can be displayed by a keyboard shortcut in the editor. It shows a text view with contextual information about the current node.

KeybindModal
: The *KeybindModal* window can be displayed by a keyboard shortcut in the editor. It shows a picture containing all available keybinding (Section A.1).

3.6. App



The *App* class is the main *React* [14] component of the program. It renders the components to the *React DOM* and enables the communication between them during the *React Lifecycle*. The *App.css* file is imported to control the visual appearance. At startup, no UmbraIR query is available for the *Editor* 3.1 and the a loading animation is rendered. In the background, the App calls the asynchronous *requestQuery(0)* function from the *request* class (Section 3.3) that fetches the UmbraIR for the first request in the *SQL* array. The resulting *Content-Graph* (Section 3.4) is returned and saved in the state of the App. After updating the state, the components of the UIR-Editor are rendered:

The *registerLanguage()* function is called to register the imported custom UmbraIR language (Section 3.2). Then the components are defined, the property methods are assigned to them and the UmbraIR is passed to the editor. A reference of each component is saved to make them accessible during the *React Lifecycle*. In order to communicate with each other, the subordinate components can call the property methods of the App that manage the exchange of information. For example, the editor can call the *nextSelectionQuery()* method of the App. As a response, the App calls the *nextItem()* method in the *QuerySelector* component (Section 3.5). This changes the array index, issues the App to request a new query and finally pass the result to the editor.

4. UIR-Editor: Features

In this Chapter the different features of the UIR-Editor software are documented. They are designed to help the user to intuitively understand and traverse UmbraIR queries by following familiar routines. The UIR-Editor's features improve *clarity* of the code's logic by highlighting important parts. Some make convoluted parts more *transparent* through code visualization. Other increase productivity and provide *convenient* content-based navigation by utilization of the UmbraIR structure (Chapter 2). Additional features enable user *interaction* with the code or provide *contextual* information. Each feature is implemented to contribute to one or more of these design goals.

Within the UIR-Editor application, most features can be accessed via the *Context Menu* by right clicking on the editor window or using a *Keyboard Shortcut*. An image containing all the available shortcuts is included in Chapter A.1 together with a legend explaining the icons in Chapter A.2. The *Keybind Modal* window can be opened in the UIR-Editor by pressing Shift + Alt + S, to display all custom keybindings.

4.1. User Interface

The *User Interface* above the main editor window (Section 4.2) displays relevant information about the current query. It allows the user to interact with the code and supports the *annotation* features described in Section 4.8. The UIR-Editor's basic user interface layout is shown in Figure 4.1. It consists of the following four components:

4.1.1. Query Dropdown

The *Query Dropdown* on the left can be used to change the currently displayed UIR-query. Alternatively the shortcut Q / Shift + Q can be used to switch to the next / previous query. It selects the last query after the first and vice versa. Once a new query is selected, a loading animation is displayed until the next query is available.

Figure 4.1.: UIR-Editor: User Interface

Status	Display	Action
Position	[19/1]	Display the node at the current cursor position
Search	SEARCH NODE	Find the first Node that matches the input
Comment	ADD COMMENT	Add a comment to the current line
Note	ADD NOTE	Add a note to the current node
Rename	RENAME NODE	Add an alias to the current node

Figure 4.2.: UIR-Editor: User Interface - Status Options

4.1.2. Status Display

The *Status Display* next to the dropdown menu highlights the status of the *Input Bar*. It switches depending on the shortcut used to access the *Input Bar*. The status indicates which action will be performed once the input text is confirmed by pressing Enter. If no input is currently expected, it displays the cursor position inside the code editor window (Section 4.2) by default. All available status options are listed in Figure 4.2.

4.1.3. Input Bar

The *Input Bar* at the right side of the *Status Display* changes its functionality according to the current status. It displays or inputs information depending on the status as shown in Figure 4.2. By default, it displays the name of the currently selected node (Section 4.5). Otherwise, if the *Input Bar* is accessed to perform an input action, Enter can be used to confirm the action and return to the main editor window. The features which depend on the *Input Bar* for user input are documented in a following sections.

4.1.4. Keyboard Button

The *Keyboard Button* can be used to toggle the UIR-Editor's *Dumb-Mode* on and off. It disables all custom keybindings and changes the button color accordingly from green to red and vice versa. Alternatively the shortcut Tab implements the same functionality. An usual application scenario for *Dumb-Mode* is provided in Section 5.4.

```
1  const %698[8] = "\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000"
2
3  declare void @umbra::Dummy::init(umbra::Dummy*, void const*)(object
4
```

Figure 4.3.: UIR-Editor: Code Editor Window

4.2. Code Editor

The code editor window (Figure 4.3) below the user interface is the UIR-Editor's centerpiece. It contains the UIR-code and implements basic text-editor functionality. *Line Numbers* are introduced on the left to enable code referencing. Alongside this, the *Glyph Margin* provides space to highlight important code parts. For example, the current cursor position is indicated with a yellow glyph. Cursor highlighting can be toggled on or off by using Shift + 2. The code is displayed in *Monospace Font* to improve code readability. To avoid changes and preserve the code's logic of the generated UIR-query, the editor is set to *read only* mode. On the right side, a colorful *Minimap* shows a query overview and allows for quick code navigation. Additionally, *Folding* can be used to hide parts of the code. The code of *function definitions* (Section 2.1.3) and *basic blocks* (Section 2.2) can be folded by clicking the arrow icon next to the corresponding line number. Alternatively the shortcut Shift + F / Shift + Alt + F folds all *blocks* / *definitions* inside the query and the F key unfolds the whole query.

4.3. Syntax Highlighting

The UIR-Editor provides *Syntax Highlighting* for the UmbraIR language following the UmbraIR structure as described in Chapter 2. The *VS-Dark* theme of the *Visual Studio Code* editor [9] is used as a baseline. Additionally, tokens and colors are defined (Figure 4.4) to represent the data layout of UmbraIR.

Token	*Color*	*Example*
Syntax	#FFFFFF	=
Component	#E06CB0	define
Variable	#9CDCFE	%state
Label	#E09C6C	body:
Constant	#FEBF9C	128
Function	#7FE06C	@_263_init
Type	#5F73E5	int32
Operator	#9C6CE0	call
Statement	#D66CE0	return
String	#CE9178	"ECONOMY"
Comment	#229977	// comment

Figure 4.4.: UIR-Editor: Syntax Highlighting

4.4. Basic Navigation

To enable effective code interaction, the UIR-Editor takes inspiration from the popular *Vim* editor [19]. It is well known for its vast amount of shortcuts, which increase productivity and improve code editing workflows. Similarly to Vim, the UIR-Editor implements keyboard shortcuts that make code navigation quick and intuitive. For basic navigation the Left, Up, Down, Right arrow keys can be used in addition to the H, J, K, L keys respectively. The U / I shortcut moves the cursor multiple positions to the left / right at once. To move to the start / end of the query, the Shift + U / Shift + I shortcut can be used. Space centers the screen at the current cursor position, which is displayed inside the *User Interface* (Section 4.1). The UIR-Editor also includes the *Cursor Grid* feature, known from other text-editors like *VS Code* [9]. The grid is set to the current line's column, every time the cursor is moved in horizontal direction. Additionally, the grid is set every time a navigation shortcut is used. Once the cursor is moved vertically into a new line, the line column is set to the previous grid value. In case the next line contains less characters than the current grid, the cursor moves to the end of that line but retains the grid value. This allows the cursor to move across empty lines without defaulting to the first column of each following line.

4.5. Node Selection

In the UIR-Editor each code part belongs to a *Content-Node* (Section 3.4) that can be selected by using the cursor. The cursor can be moved using the basic navigation keys (Section 4.4) or by clicking with the mouse. If multiple layered nodes are available at the cursor's position, because one is contained within the other, the inner most node is selected. Once a node is selected by the cursor, the name is displayed inside the *Input Bar* (Section 4.1.3) of the user interface. For more information on the current node, an *Info Modal* window can be opened with the Shift + S shortcut. It displays contextual information for each individual node. This includes the node's name, UmbraIR class, range within the editor and additional information, like explanations for all available UmbraIR OpCodes. Furthermore, any node inside the query can be found and selected by the *Node Search* feature using the Backspace / US-Backslash key. This triggers the *Status Display* (Section 4.1.2) in the user interface to change and reflect the search request. After that, the cursor is moved to the input bar. A (case sensitive) search string can be specified there. The Enter key submits and executes the search request. If any matching node exists in the current UIR-query, the cursor will be updated to the node's position. Otherwise the cursor defaults to the previous position. If a node is selected by a mouse click, the *Node Highlighting* feature Additionally, highlights all similar nodes in

grey. Similar nodes either have the same name, are global variable instances or are calls of a declared function. Their positions are also marked in the scroll bar on the right. Node highlighting can be toggled on and off using the shortcut Shift + 1.

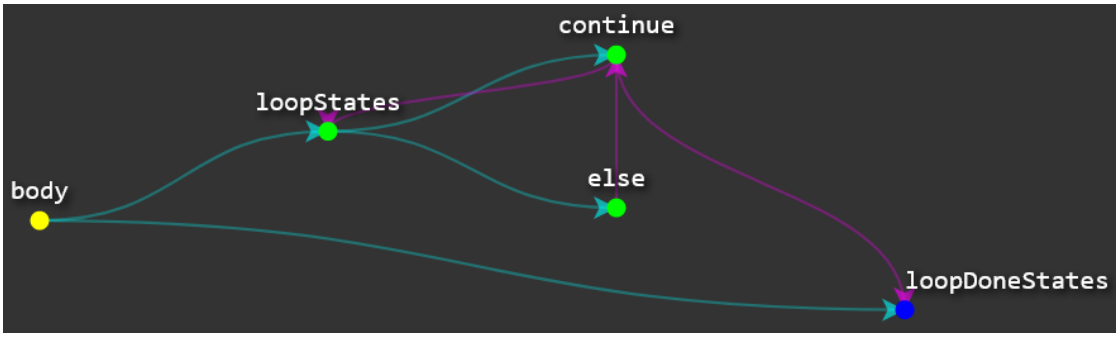


Figure 4.5.: UIR-Editor: Target Tree Modal

4.6. Advanced Navigation

The UIR-Editor implements content-based navigation features tailored to UmbraIR queries. Before advanced navigation can be used, a node has to be selected by the cursor (Section 4.5) inside the editor window. The G key moves the cursor back to the previously selected node. To move the cursor to the next / previous node with the same name, Enter / Shift + Enter can be used. This also works for custom names defined with the *Rename* feature described in Section 4.8.4.

The following features apply only, if the cursor is currently located inside of a *function definition* (Section 2.1.3). To move the cursor to the next / previous *block target* (Section 2.4.3), the T / Shift + T shortcut can be used. If a *block target* is currently selected, the shortcut R / Shift + R moves the cursor to the corresponding *block* (Section 2.2). Otherwise the cursor is moved to the next / previous *block* in relation to the current position. To provide an overview of the control flow inside the *definition*, a *Target Tree Modal* window can be displayed for using the S key. It shows the *Target Tree* - Figure 4.5 - representing the function's *basic blocks* as nodes, starting at the *block* containing the cursor. The directed edges indicate that a *target* is directing the control flow from the outgoing *block* to the targeted *block*. A string representation of the *Target Tree* is available when hovering over any *block label*. This string also shows the conditional variables for each control flow *instruction* (Section 2.3) in braces. *Target Tree* hovering can be toggled on or off with the shortcut Shift + 8.

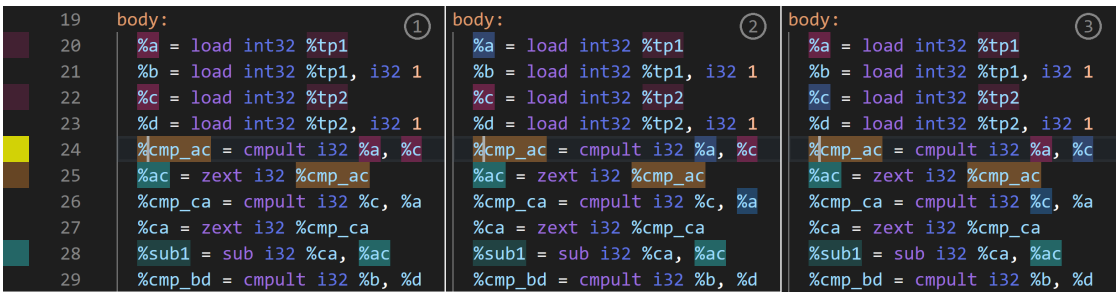


Figure 4.6.: UIR-Editor: Child and Parent Highlighting

4.7. Variable Flow

Due to the SSA form of UmbraIR, multiple *variables* (Section 2.4.2) are needed to express the code's logic. As a result it can be difficult to determine the current value of any variable across the query. The UIR-Editor implements functionality to facilitate working with code that contains SSA variables. To trace the value of a specific variable, the parent and child variables should be considered. When hovering over a variable with the mouse, all occurrences of the same variable are highlighted in blue. This feature can be toggled on or off with the shortcut Shift + 3. If a variable node is selected by the cursor (Section 4.5), the editor highlights all occurrences of the same variable in orange. Additionally, all variables related to the current variable are highlighted. Section 1 of Figure 4.6 shows the relationships of the current variable *%cmp_ac* across the *body* block. By default related variables include the next three ancestor levels of parents and children. Parent - *%a*, *%c* - and grandparent variables - *%tp1*, *%tp2* - incrementally contribute to the current value and are highlighted in red. Child - *%ac* - and grandchild variables - *%sub1* - are assigned partially with the current value and are highlighted in green. The color saturation indicates, how close the relationship is. Relative decorations can be toggled on or off using the Shift + 4 shortcut for child and Shift + 5 for parent highlighting. The UIR-Editor also implements multiple shortcuts to quickly navigate across the related variables. Shift + J / Shift + K can be used to move the cursor to the current child / parent variable. If there are multiple relatives of the current variable available to move to, the first one from the left is selected by default. To highlight the currently selected child / parent options in blue and cycle through them, the Shift + H / Shift + L shortcut can be used. This feature is showcased in Section 2 and Section 3 of Figure 4.6, each highlighting one of the respective parents.

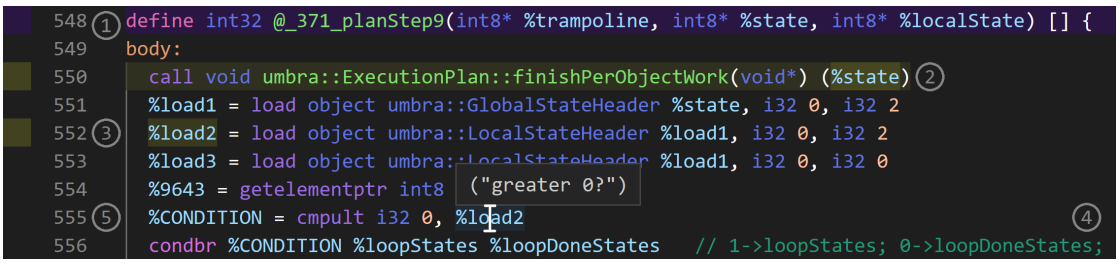


Figure 4.7.: UIR-Editor: Query Annotations

4.8. Query Annotation

The UIR-Editor enables the user to adjust each query to the current workflow by providing convenient annotation features. Multiple tools are implemented to allow user interaction with the code, while preventing accidental changes to the UmbraIR query. Therefore the following features operate only on the surface-level of the query representation that is displayed by the editor. Additionally, all annotations are persistent across user sessions. They are saved in the localStorage of the UIR-Editor's webbrowser and are recovered on startup of the application. The basic functionality of each feature is shown at the five numbered postions of Figure 4.7.

4.8.1. Bookmark

In order to quickly find a specific line inside a UIR-query, a bookmark can be added using the B shortcut. A bookmark is represented by a purple highlighted line at position 1 of Figure 4.7. When a bookmarked query is loaded, the cursor will immediately move to the corresponding code line. Every query in the UIR-Editor can contain a single bookmark. Therefore using the B key again moves the bookmark to the new line. Additionally, the shortcut Shift + B can be used to move the cursor to the current bookmark. Shift + Alt + B removes the current bookmark. Bookmark decorations in the code can be toggled on and off using Shift + 6.

4.8.2. Note

To reference interesting subtleties of the UIR-code or highlight important parts, text notes can be added. Every query can contain multiple notes, represented by a dark yellow background at the associated node or line. The shortcut N can be used to attach a note to the currently selected node (Section 4.5) or code line. This triggers the *Status Display* (Section 4.1.2) in the user interface to change and reflect the request.

After that, the cursor is moved to the *Input Bar* (Section 4.1.3). There the note string can be specified. The Enter key submits the request and attaches the note. One string note at a time can be attached to any node or code line inside a single UIR-query. The content of a note is displayed when hovering over the corresponding node or line with the mouse. Additionally, all string notes for the currently selected node are shown inside the user interface input bar after the node's name inside braces. Notes that are attached to higher level nodes, which contain other nodes, also apply to all contained nodes. Layered notes can be identified by the less saturated color of the more general note, as shown in Figure 4.7. At position 2, the note of the *call* instruction is layered with the *%state* variable's note. In this case, the note string of the more general note will be displayed after the other. If a node has multiple occurrences across a query, the associated note string is shown at all occurrences. This is shown in Figure 4.7, as the note string assigned to the variable *%load2* at position 3 is shown when the mouse is hovered over the related variable in line 555. If multiple notes are attached to related occurrences, the string of the closest related note will be displayed first. A note can be edited by selecting the corresponding node with the cursor and using the N key. To remove the current note, the shortcut Shift + N can be used. Shift + Alt + N removes all notes across the current query. Note decorations in the code can be toggled on and off using Shift + 7. In order to quickly find the next / previous note in the current query, the M / Shift + M shortcut moves the cursor to the associated position. Using this feature, notes can serve as quick access reference points for specific code parts.

4.8.3. Comment

The UIR-Editor features readily apparent code annotations in the form of line comments. Inside a query the start of a comment is indicated with two leading slash characters (//) followed by the comment string. For example at position 4 of Figure 4.7, a comment is displayed to clarify the control flow instruction. Every line inside a query can contain a single comment. A new comment can be added using the C shortcut. This triggers the *Status Display* (Section 4.1.2) in the user interface to change and reflect the request. After that, the cursor is moved to the *Input Bar* (Section 4.1.3). There the comment string can be specified. The Enter key submits the request and adds the comment to the line. A comment can be edited by navigating the cursor to the corresponding line and using the C key. To remove a line comment, the shortcut Shift + C can be used. Shift + Alt + C removes all comments in the current query.

4.8.4. Rename

UmbraIR code is generated automatically for the compiler and is not designed for manual programming. Therefore most of the code is named inconclusively. To make the code more readable, the UIR-Editor allows surface-level renaming. For example in Figure 4.7, the variable at position 5 was given a meaningful custom name. This feature can only be used for *block labels* (Section 2.2), *block targets* (Section 2.4.3) and *variables* (Section 2.4.2). Once a suitable node is selected (Section 4.5) inside the editor window, the rename feature can be used with the V shortcut. This triggers the *Status Display* (Section 4.1.2) in the user interface to change and reflect the request. After that, the cursor is moved to the *Input Bar* (Section 4.1.3). There the new name can be specified. The Enter key submits the request and changes the display name of the node. New names apply to all occurrences of the same variable, target or label across a query. In order to ensure correct *Syntax Highlighting* (Section 4.3), the allowed characters for new names are limited. Names can only include the *Ascii* [12] characters for the numbers from 0 to 9, the upper and lower case latin alphabet and the underscore character. Additionally, names of already existing nodes are excluded. If the input name is already in use or includes an illegal character, the renaming will fail and the cursor is returned to the previous position. A node can be renamed multiple times using the V key. To restore the name of the current node, the shortcut Shift + V can be used. Shift + Alt + C removes all custom names in the current query.

5. UIR-Editor: Use Cases

This Chapter constitutes a reference guide for the effective application of the UIR-Editor. In the following sections, the software is applied in different use case scenarios that show workflow routines for the typical user. They can help to guide professionals that want to improve UmbraIR as well as novice users that want to get familiar with the code. The examples rely on the background knowledge of the UmbraIR structure according to Chapter 2. Each use case highlights some of the editor's features (Chapter 4) and combines them to improve the user experience when working with UmbraIR.

The most evident application for the editor is to display and present UIR-code. Futhermore the software provides many helpful tools to analyze queries on different code levels. In this context, the editor's features are used to highlight connections between the generating C++ code and the generated UmbraIR. Finally it is explained how the *Dumb-Mode* feature can be applied to effectively find keywords in the code.

5.1. Code Presentation

UmbraIR code is usually generated and immediately processed by the compiler [16]. In this context, the visual appearance of the code is irrelevant. However, if users want to analyze or present UmbraIR, it is important that the code is displayed in an appealing way. Especially large queries can be overwhelming and difficult to look at in raw text format. So far users are required to fetch the UmbraIR from the Umbra [11] backend and manually highlight the important parts of code. This is a tedious and inefficient process that can be susceptible to errors. To counteract this and to provide an alternative, the UIR-Editor introduces an environment to automatically fetch and display UmbraIR. This use case shows the implemented features that allow users to effectively present UIR-code. For reference the Figure 5.1 emphasizes the contrast of the raw UmbraIR on the left and same code inside the UIR-Editor on the right.

In order to display any UmbraIR query, it first has to be loaded in the editor. The UIR-Editor enables the user to easily choose and switch the currently presented query. By default the UmbraIR for the first TPC-H [23] query is loaded. The application includes the SQL code that is required for the UmbraIR of all 22 benchmark TPC-H requests. To load additional queries, the corresponding SQL request has to be added to the project, as described in Section 3.3. Any query can be loaded using the *Query Selector*

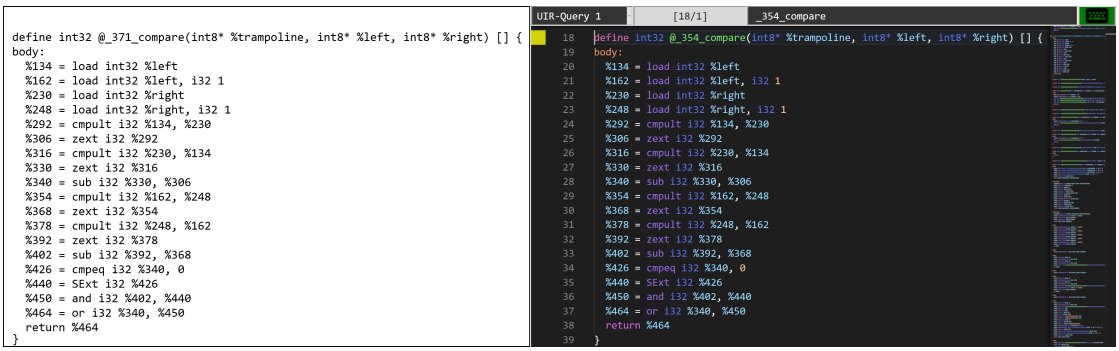


Figure 5.1.: Code Presentation: Before and After

(Section 4.5) or the available keyboard shortcuts. Once a query is selected, the associated SQL request is remotely executed in Umbra, the resulting UmbraIR is fetched from the webserver and the code is displayed in the editor window.

After any query is loaded in the editor, multiple visual enhancements are applied to guide the user through the code. Above the main editor an interactive *User Interface* (Section 4.1) is added to provide contextual additional information to the user. It includes the dropdown menu for the available queries, shows the cursor position in the *Status Display* (Section 4.1.2) and the current selection in the *Input Bar* (Section 4.1.3). Inside the editor window (Section 4.2) basic features enhance the clarity of the code by improving readability and provide an overview of the query. These include *Line Numbers*, the *Glyph Margin* on the left as well as a colorful *Minimap* on the right side. Code *Folding* can be used to hide the less relevant code of *function definitions* (Section 2.1.3) and *basic blocks* (Section 2.2). Furthermore, similar parts of the code are displayed in matching colors to visualize the logical structure of UmbraIR. The automated *Syntax Highlighting* (Section 4.3) clarifies the code's logic and helps the user to recognize logical connections. For example in Figure 5.1 it can be seen, that most of the code between line 20 and 37 look similar. These lines all start with a light blue word followed by the assignment operator =, a purple and a dark blue word. Such patterns help users intuitively understand the syntax and identify parts of the logical structure.

As a result the previously described features make UmbraIR more accessible and help users to easily find intricacies in the code. By presenting the code in this way, the UIR-Editor improves workflows with UmbraIR for novice and professional users.

5.2. Code Analysis

Analyzing functions step by step can lead to new insights about the details of an UmbraIR query. The following Sections include two workflow examples for effective manual code analysis with the UIR-Editor. They provide a guide on how the editor can be used to make the code flow more transparent and how to clarify connections in the code. Novice users can apply the shown techniques to get familiar with UmbraIR, interact with the code and gain a deeper understanding. Professionals can use them in their efforts to improve and debug UIR-code. In the first example, the editor's features are used to verify the result of a function by tracing the variable assignment process. The second example shows how the user can quickly navigate through convoluted control flow constructs and how to visualize them.

5.2.1. Variable Assignment

Due to the SSA property of UmbraIR, variables can only be assigned once [16]. Therefore multiple related variables are generated to calculate the final result of a function. By default these variables also use inconclusive names. Because of this, it can be difficult to determine the content of any individual variable and to follow the code flow. To make variables easier to trace and gain insights about the inner workings of UmbraIR, multiple features are implemented by the UIR-Editor.

For the following example the *@_371_Compare* function of UIR-Query 1 is considered as shown in Figure 5.2 It is used to compare two tuples, which contain two values each. The function is expected to return a different result depending on the argument tuples. More precisely there should be three different return values, if the first tuple is less than, greater than or equal to the second tuple. In order to verify this assumption, the function is analyzed line by line three times with different sets of arguments. Starting at the argument tuples, the value of each following variable is determined using manual bitwise calculations. To quickly find the child variables that are assigned next, the *Variable Flow* (Section 4.7) navigation features are used. If the functionality of any *instruction* (Section 2.3) is unknown, the *Info Modal* Shift + S (Section 4.5) can be displayed for clarification. The value of every assigned variable is annotated with a line *Comment* C (Section 4.8.3). Additional explanations for interesting parts of the code are attached using a *Note* N (Section 4.8.2). Furthermore, the assigned variables are given a more descriptive name to emphasize their contents by using the *Rename* V (Section 4.8.4) feature. This workflow is repeated until the final instruction of the function is reached and all variable assignments are clarified.

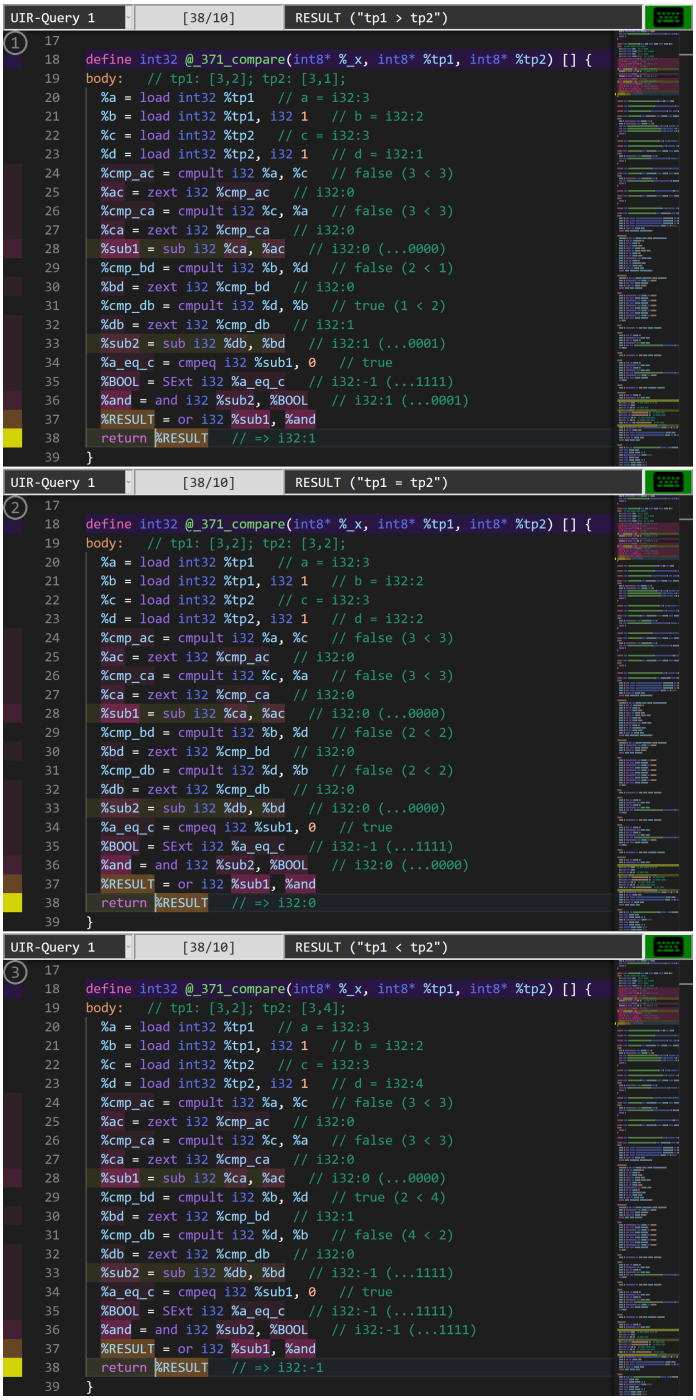


Figure 5.2.: Variable Assignment: Compare Function

As a result of the three analysis iterations, the *@_371_Compare* function is annotated each time as shown in Figure 5.2. In line 19 the argument tuples are defined. From lines 20 to 23 the values from the tuple registers are loaded. In lines 24 to 28 the first value of each tuple is compared. In lines 29 to 33 the second value of each tuple is compared. The rest of the lines calculate the final result by comparing the two interim results. From this detailed manual analysis of the variable assignment process it can be concluded, that the function works as expected and returns the following result: If the first tuple value (*tp1*) is greater than the second value (*tp2*), the function returns "1" as a 32-bit integer. If both tuples are the same, "0" is returned and "-1" otherwise.

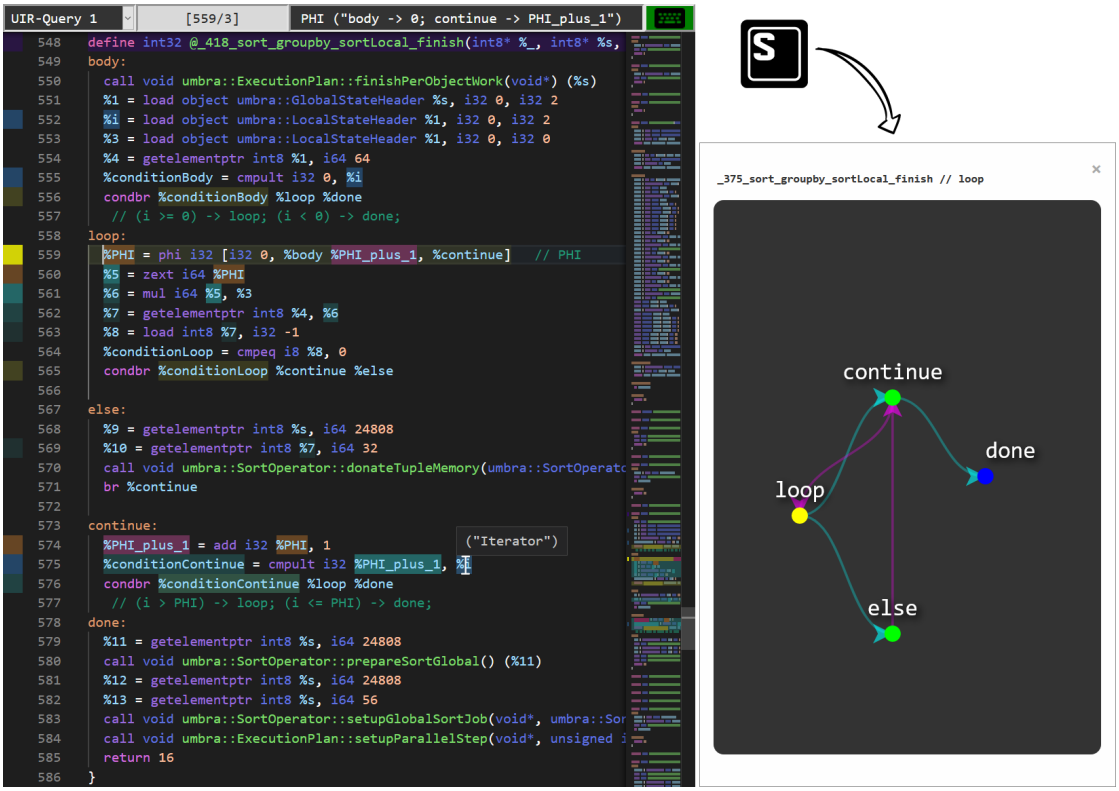


Figure 5.3.: Control Flow: Sort Loop Function

5.2.2. Control Flow

Due to the SSA property of UmbraIR, control flow constructs are implemented without reassigning variables [16]. Therefore the code of the *function definitions* (Section 2.1.3) is separated into multiple *basic blocks* (Section 2.2). UmbraIR allows the current block to target any other basic block for execution. Additionally, the *PHI node* (Section 2.3) changes its value depending on the previously visited block. These control flow constructs can be combined, to generate loops and conditional functionality in UmbraIR. However, some functions contain convoluted loops that span across multiple basic blocks. This makes the control flow move back and forth between the blocks, which complicates tracing the code. As a result, the UIR-Editor implements features that clarify the control flow and enable efficient movement between blocks.

For this example the function *@_418_sort_groupby_sortLocal_finish* of UIR-Query 1 is considered as shown in Figure 5.3. It sorts the provided arguments by utalizing control flow constructs across multiple basic blocks. Like any other UmbraIR function [16], the execution starts at the first block inside the *body*. At line 556 the *conditional branch* instruction is used to target a different block for execution depending on a conditional variable. Once *%conditionLoop* is selected (Section 4.5) by the cursor, the *Variable Flow* (Section 4.7) is highlighted in the editor. Now it is apparent that the loop inside the function is only executed, if the variable *%i* is greater than or equal to "0", otherwise the control flow is transferred to the *done* block. To clarify this, a line *Comment* C (Section 4.8.3) is added. After that, the *loop* block can be reached using *Advanced Navigation* (Section 4.6). First the *%loop* target is selected T, before the cursor is moved directly to the targeted block R. At Line 559 of the *loop* block the variable *%PHI* is assigned by the *PHI node*. It either contains the 32-bit integer "0" or the value of *%PHI_plus_1* depending on the previous block. To emphasize this, a text *Note* N (Section 4.8.2) is added to the PHI instruction. At this point it is helpful, to visualize the function by using the *Target Tree Modal* S (Section 4.6). The *Target Tree* is a representation of the control flow starting at the current block, as shown on the right side of Figure 5.3. It is visible that all outgoing branches of the *loop* block eventually lead to *continue*. Inside the *continue* block, the variable *%PHI_plus_1* is assigned with the value of the *PHI node* incremented by "1". Then it is evaluated, if the assigned value is less than the variable *%i*. Depending on the result, the loop is either terminated in the *done* block or continued, by moving the control flow back to the *loop* block. Therefore *%i* represents the maximum amount of repetitions and *%PHI* is used for the current loop index.

This workflow example shows, how the UIR-Editor can be applied to visualize the control flow of UmbraIR functions. As a result of this manual analysis of the sort loop function, it is shown how the control flow constructs are clarified (Figure 5.3) and how productivity is increased by the convenient navigation features.

```cpp
UInt64 Hash::hashValues(vector<SQLValue> values) {
    // {...}
    UInt64 hash1(5961697176435609000);
    UInt64 hash2(2231409791114444000);
    for (UInt64 v : concatenatedValues) {
        hash1 = hash1.crc32(v);
        hash2 = hash2.crc32(v);
    }
    UInt64 hash = hash1 ^ hash2.rotateRight(32);
    hash *= 2685821657736339000;
    // {...}
    return hash;
}
```

Figure 5.4.: Code Comparison: C++ Hash Function [16]

```
UIR-Query 1    [158/1]    cont4
158  cont4:
159    %3316 = ashr i64 %3294, 63
160    %3330 = builddata128 d128 %3294 %3316
161    %3344 = call d128 umbra::BigNumericRuntime::mulTrap(umbra:
162    %3366 = zext i64 %2676
163    %3376 = zext i64 %2734
164    %3386 = shl i64 %3376, 32
165    %3400 = or i64 %3366, %3386
166    %3442 = crc32 i64 5961697176435609000, %3400
167    %3456 = crc32 i64 2231409791114444000, %3400
168    %3470 = rotr i64 %3456, 32
169    %3484 = xor i64 %3442, %3470
170    %3512 = mul i64 %3484, 2685821657736339000
UIR-Query 1    [165/3]    v ("concatenatedValues")
158  cont4:
159    %3316 = ashr i64 %3294, 63
160    %3330 = builddata128 d128 %3294 %3316
161    %3344 = call d128 umbra::BigNumericRuntime::mulTrap(umbra:
162    %int1_64 = zext i64 %int1   // Zero extend to 64 bit
163    %int2_64 = zext i64 %int2
164    %int2_64_l = shl i64 %int2_64, 32   // Shift int2
165    %v = or i64 %int1_64, %int2_64_l   // Combine integers
166    %hash1 = crc32 i64 5961697176435609000, %v   // 1. crc32
167    %hash2 = crc32 i64 2231409791114444000, %v   // 2. crc32
168    %hash2_r = rotr i64 %hash2, 32   // Shift hash2
169    %hash = xor i64 %hash1, %hash2_r   // Combine hash parts
170    %result = mul i64 %hash, 2685821657736339000   // Mix
```

Figure 5.5.: Code Comparison: UmbraIR Hash Function - Before and After

5.3. Generator Code Comparison

Umbra automatically generates UIR-code from a C++ application for each new database request [16]. Therefore manual changes to any query are lost once the same request is executed again. To make structural changes to UmbraIR, the C++ generator code needs to be updated and debugged. In this context, it can be helpful to compare the related code parts side by side. For example, when analyzing the runtime performance of specific function such a comparison may identify bottlenecks in the UIR-code. To find them, the user can try to identify unnecessary loops or redundant instruction calls in the generated UmbraIR. The resulting conclusions can then be applied to change the C++ code and improve the performance of UmbraIR.

This use case explains how the UIR-editor's features support the optimization process of the C++ code. More precisely it is shown how the user can emphasize relevant parts of UmbraIR in relation to the C++ code and add annotations to them. As an example Figure 5.4 and Figure 5.5 both show parts of the same typical hash function. Figure 5.4 depicts the most important lines of the generator function in C++. The first UIR-query of Figure 5.5 shows the generated code without annotations in UmbraIR. Both code parts in C++ and UIR contain the same code logic and compute a hash value for two given 32-bit integers [16]. Based on this knowledge, the second UIR-query of Figure 5.5 is annotated by the UIR-Editor's different features. The following workflow example shows how the UIR-Editor software helps the user to detect and emphasize the similarities of UmbraIR and the C++ generator code. To accomplish this, the editor makes UmbraIR more accessible and provides user guidance by pointing out important connections in the code. As a result the user can gain deeper insights and apply them to the higher-level code. Additionally, the annotations are persistent across user sessions. This makes woking with previously analyzed parts of code much easier.

Example scenario The runtime analysis of the benchmark TPC-H Query 01 [23] shows bad results for the hashing process during execution. The problem can be narrowed down to the UIR-code generated by the C++ hash function shown in Figure 5.4. From the function's context is apparent that two 32-bit integers are used as arguments for the hash. Additionally, it is known that the logic of the hash function is translated to the block *cont4* of UIR-Query 01. An Umbra developer is tasked to investigate the runtime problem. In order to get familiar with the problematic code, the UIR-Editor application is used as described in Section 5.1 to present the relevant basic block. The following 17 steps describe the workflow of the developer inside the editor. All relevant keyboard shortcuts are included. The goal is to make the problematic UIR-code of the hash function easier to understand. All steps are visualized in Section A.3.

1.

At startup of the UIR-Editor, the Umbra program representation for TPC-H 01 is loaded by default. To navigate to the relevant basic block the developer uses *Search* Backspace (Section 4.5). The name "cont4" is typed into the *Input Bar* (Section 4.1.3) and confirmed Enter. Now the editor's cursor is moved to the first node with the specified name. To reach the desired block, the cursor is moved to the next occurrence of "cont4" using *Advanced Navigation* Enter (Section 4.6). After the relevant basic block *cont4* is reached, a *Bookmark* B (Section 4.8.1) is added at line 158. This allows the user to move the cursor directly to the specified line Shift + B, once the user scrolls to a different line or reloads the query.

2.

The developer knows from the context of the C++ hash function that two 32-bit integers are used as arguments. When taking a look at the code of Figure 5.4, it can be seen at line 1 that the function returns a *UInt64* value. It can also be noted that the calculations in the following lines only contain 64-bit integer values. Therefore it can be concluded that the arguments of the hash function have to be converted inside UmbraIR to the appropriate 64-bit integer type.

Back inside the UIR-Editor, the developer notices the two similar *zext i64* instructions. The cursor is moved down to the corresponding line 162 using *Basic Navigation* J (Section 4.4). To recall the exact functionality of the *zext* instruction, the *Info Modal* Shift + S (Section 4.5) is used for clarification. It is now apparent, that *zext* is generated by UmbraIR to extend the length of the 32-bit arguments to 64-bit in order to prepare them for the hashing process. After that, the cursor is moved right L to the operand variable *%2676* of the first *zext* instruction. To give this variable a more recognizable name the *Rename* V (Section 4.8.4) feature is be used. The new name "int1" is specified in the *Input Bar* and confirmed Enter. Now the renamed variable *%int1* is selected. By default the *Variable Flow* (Section 4.7) is displayed. This means that the selected variable's children and parents are highlighted. Therefore the first child variable *%3366* can now be identified by the bright green background color.

3.

At this point the developer can move the cursor directly to the first child Shift + J (Section 4.7) of the selected variable *%int1*. The result of the *zext* instruction in line 162 with the operand *%int1* is assigned to the selected variable *%3366*. Therefore it now contains a 64-bit value consisting of the zero extended 32-bit hash argument. To reflect the variable's contents, *%3366* is renamed V to *%int1_64*.

4.

Now the developer wants to select the second 32-bit integer argument of the hash function. First the cursor is moved from the current variable *%int1_64* back to the first parent variable *%int1* Shift + K (Section 4.7). To select the desired variable *%2734*, the cursor is moved down J. Then the variable is renamed V to *%int2*.

5.

Similar to Step 3, the developer moves the cursor to the first child Shift + J of the selected variable *%int2*. The variable *%3376* in line 163 contains a 64-bit value consisting of the zero extended 32-bit hash argument inside *%int2*. To reflect the variable's contents, *%3376* is renamed V to *%int2_64*.

6.

The editor highlights all occurrences of the selected variable in orange color (Section 4.7). Now the developer recognizes that *%int2_64* is used as an operand in the following line 164. It contains the *shl* instruction, which also takes a constant value of *32* as an operand and assigns the result to *%3386*. Next the cursor is moved to the assigned child variable Shift + J. To reflect that it contains the value of *%int2_64* shifted to the left, *%3386* is renamed V to *%int2_64_l*.

7.

So far the developer has clarified the argument variables for the hash. In the next step of the hashing process these argument are combined into a single value. This is represented in line 165 of the UmbraIR by the logical *or* instruction. Here the previously renamed variables *%int1_64* and *%int2_64_l* are concatenated and the result is assigned to the variable *%3400*. After moving to the current first child *%3400* Shift + J, both occurrences inside the next two *crc32* instructions are highlighted. The developer notices that both *crc32(v)* function calls in the C++ code contain the same argument. Therefore it can be concluded, that *%3400* represents the variable *v* defined in line 5 of the C++ function. To reflect this connection in the UmbraIR, the variable is renamed V to *%v*.

8.

The currently selected variable *%v* contains the argument values that need to be hashed. To emphasize the important contents of *%v* at all following occurrences, a text *Note* N (Section 4.8.2) is used. The note's string "concatenatedValues" is specified in the *Input Bar* and confirmed Enter. Now the note is attached and displayed after the current variable's name in the *Input Bar*. Furthermore, the note is shown, when the developer hovers over *%v* in line 166.

9.

Next the developer notices, that the *crc32* instruction in line 166 of the UmbraIR uses the same constant value ("5961697176435609000") as the *hash1* variable in line 3 of the C++ code. As a result the assigned child *%3442* is selected Shift + J and renamed V to *%hash1* in order to represent the similarity.

10.

Now the developer notices the same constant value ("2231409791114444000") inside the next *crc32* instruction as the *hash2* variable in line 4 of the C++ code. To select the assigned child variable *%3456*, the cursor can simply be moved down to line 167 J. Alternatively the desired variable can be accessed by moving back to the parent *%v* Shift + K. From there the next child variable *%3456* is highlighted in blue by cycling through all available children Shift + H (Section 4.7).

11.

Once *%3456* is declared as the next child, the cursor can be moved to the variable Shift + J. After that, it is renamed V to *%hash2*.

12.

Before both parts of the hash can be combined into the *hash* variable in line 9 of the C++ code, the function *rotateRight(32)* is called for *hash2*. Similarly in line 168 of UmbraIR the *rotr* instruction contains the operand *%hash2* and a constant value of *32*. Therefore the developer moves the cursor to the assigned child variable *%3470* Shift + J and renames V it to *%hash2_r* to represent the right shift.

13.

The *xor* instruction in line 169 of the UmbraIR contains the previously renamed variables *%hash1* and *%hash2_r* as operands and assigns the result to *%3484*. In line 9 of the C++ code the developer recognizes the matching exclusive-or operator (^). Therefore it can be concluded that the C++ variable *hash* corresponds to the assigned variable *%3484*. As a result, the cursor is moved to the child variable Shift + J and renamed V to *%hash*.

14.

Finally, the C++ variable *hash* is multiplied with a constant value to mix the hash parts. This is represented by the *mul* instruction inside line 170 of UmbraIR, which contains the same constant operand ("2685821657736339000") and the *%hash* variable. The developer moves the cursor to the assigned child Shift + J *%3512* and renames V it to *%result*. This variable contains the final hash value.

15. To emphasize the end of the hashing process, the developer moves the cursor left [H] to the instruction of line 170 and attaches a note ("hashDone").

16. To emphasize the start of the hashing process, the developer moves the cursor up [K] to the instruction of line 162 and attaches a note ("hashStart").

17. In order to explain all the previous steps, *Line Comments* [C] (Section 4.8.3) are added to the UmbraIR. Each comment string is specified in the *Input Bar* and confirmed [Enter]. After that, the variable *%v* is selected to highlight the flow of the concatenated values in the hashing process.

These previous steps show how the UIR-Editor's features can be effectively combined to annotate an UmbraIR query. The code with all annotations can be seen in the bottom picture of Figure 5.5. The final result improves the readability, highlights connections in the code and makes the parallels to the generator code visible. As a result, this helps any developer to detect abnormalities and improve the UmbraIR.

5.4. Finding Keywords

Most ot the UIR-Editor's many features can be conveniently accessed through the keyboard shortcuts shown in Figure A.1. While this increases productivity when analyzing UmbraIR code, it leads to some complications for the integrated *Find* [Ctrl + F] feature. Once the *Find Dialog* is opened, the editor's keybindings are not automatically disabled. Therefore the keys, which are reserved by the shortcuts, can not be typed regularly. This use case shows a workaround to solve this problem.

In order to find any sequence of characters in the UIR-Editor, the *Find* [Ctrl + F] feature can be used. Due to software limitations, the editor's keyboard shortcuts need to be disabled before opening the *Find Dialog*. For this purpose, *Dumb-Mode* can be toggled by clicking the *Keyboard Button* (Section 4.1.4) inside the *User Interface* or by using the [Tab] key. After all shortcuts (except [Tab]) are disabled, the *Keyboard Button* changes its color from green to red, which indicates that *Dumb-Mode* is active. Now the *Find Dialog* [Ctrl + F] can be opened and the desired string can be typed without triggering the editor's features. If the cursor is inside the dialog window, the different occurrences can be highlighted in the editor using the [Enter] key. After the desired string is found, [Tab] can be used to directly move the cursor from the *Find Dialog* to the result inside the UIR-Editor and disable *Dumb-Mode* in the process.

6. Conclusion & Future Work

This thesis presented the UIR-Editor application, an interactive explorer tool for the Umbra Intermediate Representation. First the logical structure of UmbraIR was described, upon which the program is built. Then the implemented features were documented and applied in different use case scenarios. In this context it was shown, how the features improve the experience for users of UmbraIR:

The UIR-Editor presents UmbraIR effectively and enables intuitive understanding of the data layout by emphasizing logical connections in the code. UmbraIR becomes more accessible through visual clues and intuitive interaction techniques that guide users through the code. Advanced features increase productivity for the debugging process of UmbraIR by providing convenient navigation routines and additional information about the context of the code.

In conclusion, the UIR-Editor simplifies working with UmbraIR and lays the foundation for the implementation of further code visualization and debugging tools.

Future Work

The current implementation of the UIR-Editor works well but can still be improved and additional features can be added in the future. To determine which features should be developed, a field study can be conducted on the requirements of regular users.

For example, a "SQL-Inspector" feature would be feasible that displays the underlying SQL request side by side with the UmbraIR code and highlights the connections. Furthermore, a feature that prints the editor's content as an image could be used to share annotated UmbraIR queries. Additionally, submenus can be added in the context menu to group the features by category. The next step is to make the UIR-Editor available to all users of UmbraIR by integrating it into the Umbra web interface.

A. Appendix

A.1. Keyboard Shortcuts

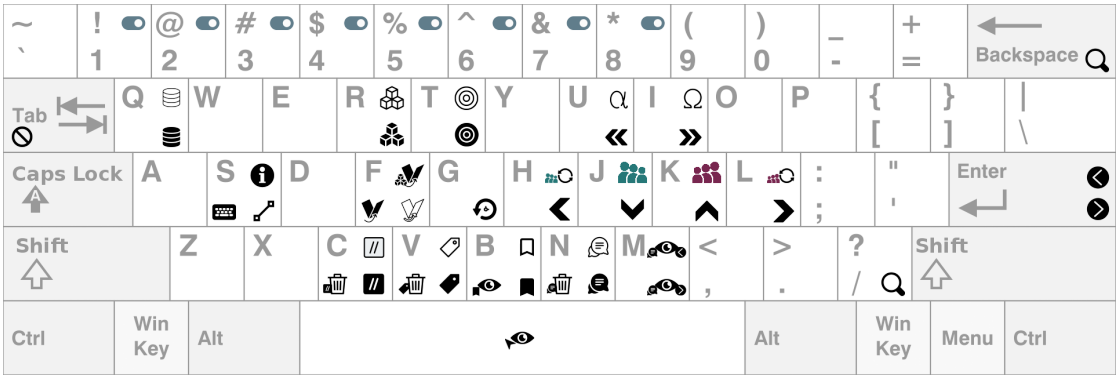


Figure A.1.: UIR-Editor: Keyboard Shortcuts (Icons by Flaticon [2])

A.2. Keyboard Shortcuts - Legend

Keybind	*Icon*	*Command*
H		Left
Shift + H		Cycle Children
J		Down
Shift + J		Current Child
K		Up
Shift + K		Current Parent
L		Right
Shift + L		Cycle Parents
U		Jump Left
Shift + U		Jump Start
I		Jump Right
Shift + I		Jump End
Q		Next Query
Shift + Q		Prev Query
Enter		Next Occurrence
Shift + Enter		Prev Occurrence
T*		Next Target
Shift + T*		Prev Target
R*		Next Block
Shift + R*		Prev Block
G		Go Back
/ (US) || Back		Search Node
F		Unfold All
Shift + F		Fold Blocks
Shift + Alt + F		Fold All
S*		Target Tree
Shift + S		Node Info
Shift + Alt + S		Keybinds
C		Comment
Shift + C		Del. Comment
Shift + Alt + C		Del. Comments
V**		Rename
Shift + V		Del. Rename
Shift + Alt + V		Del. Renames
B		Bookmark
Shift + B		Del. Bookmark
Shift + Alt + B		To Bookmark
N		Note
Shift + N		Del. Note
Shift + Alt + N		Del. Notes
M		Next Note
Shift + M		Prev Note
Space		To Cursor
Tab		Toggle Keybinds
Shift + (1, ..., 8)		Toggle Feature
*** : Active only inside a Definition**		
**** : Active only for Variables and Targets**		

Figure A.2.: UIR-Editor: Keyboard Shortcuts - Legend (Icons by Flaticon [2])

A.3. Hash Function Annotations

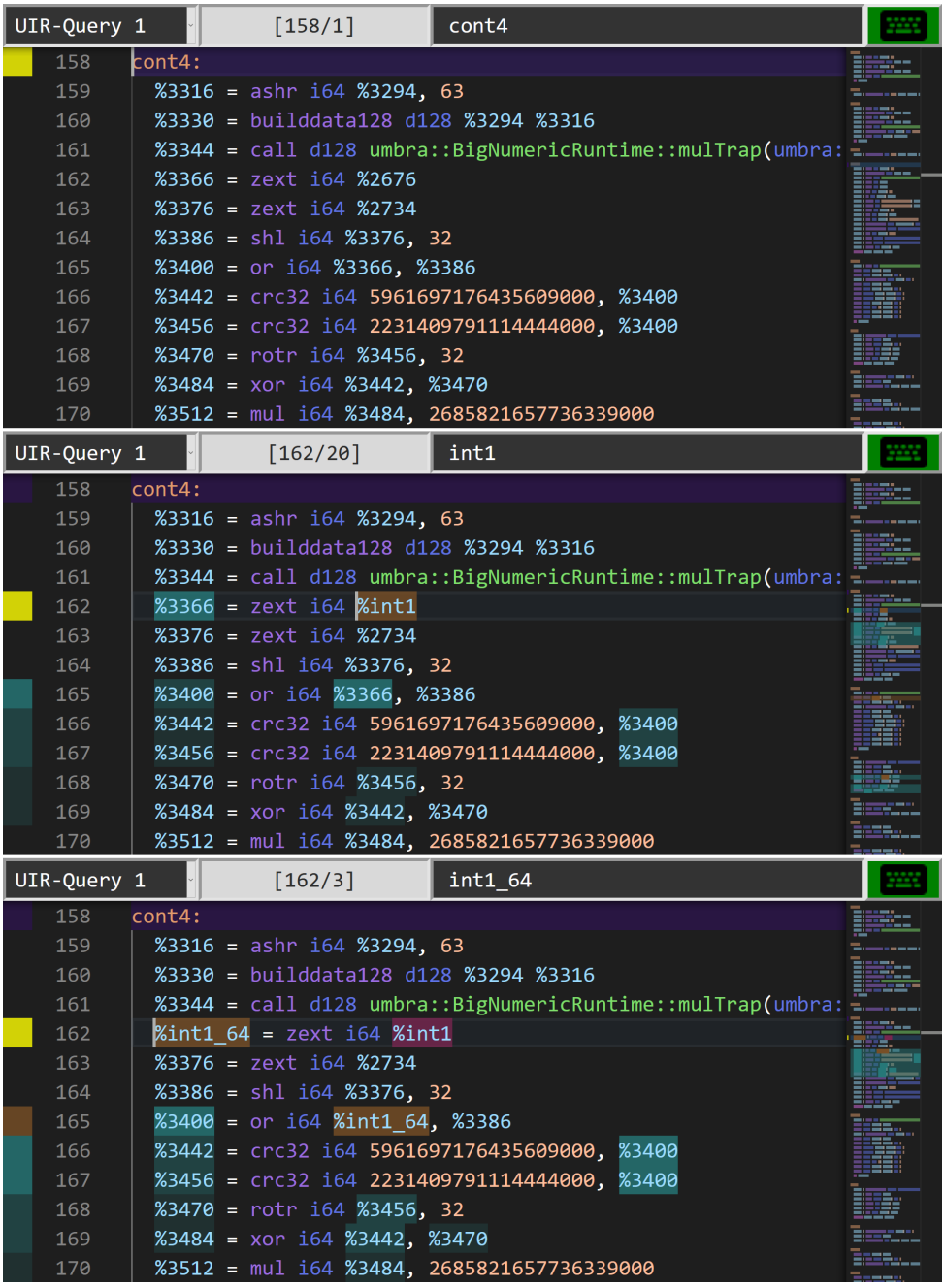


Figure A.3.: UmbraIR Hash Function Annotations: Steps 1, 2, 3

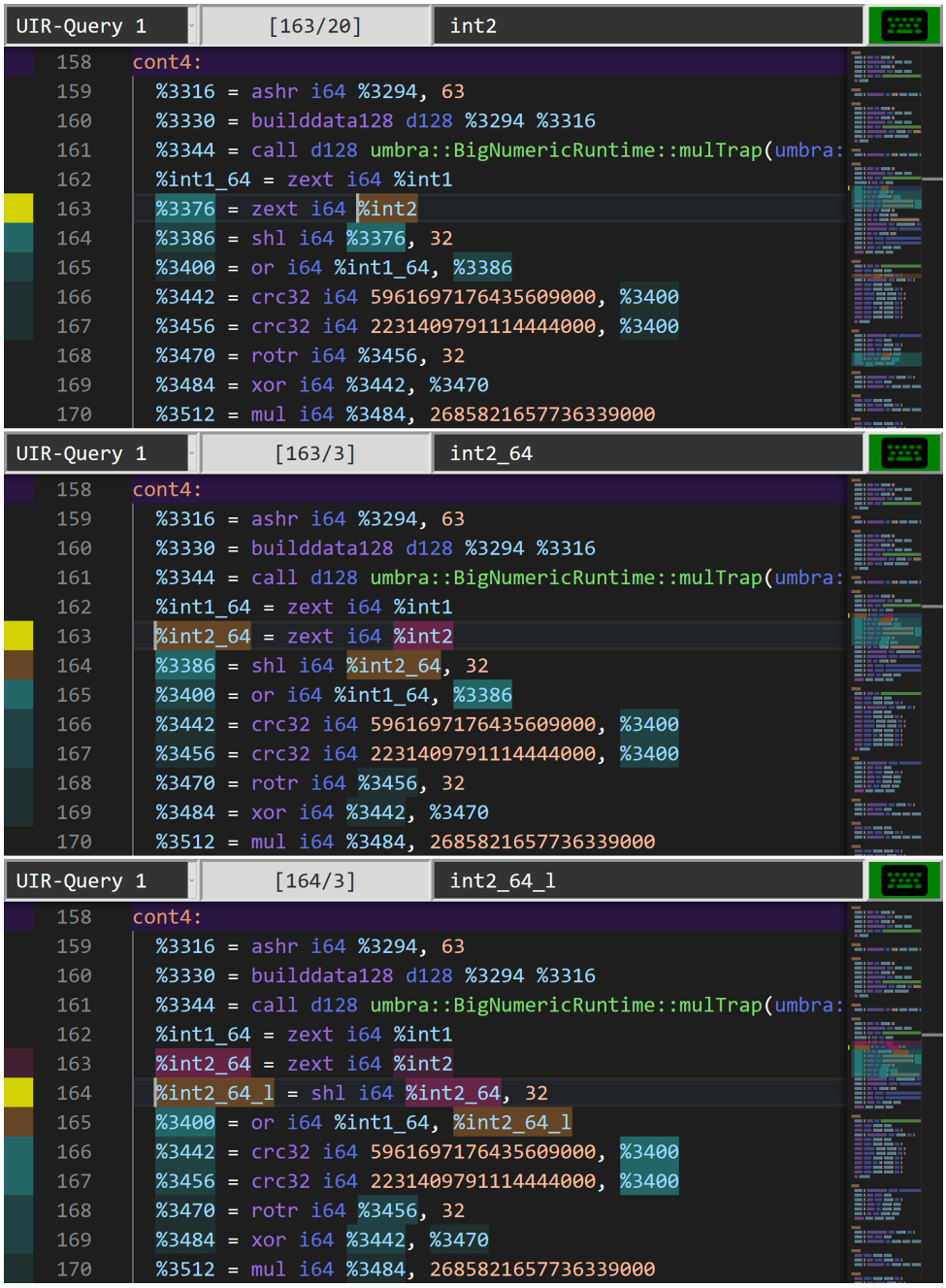


Figure A.4.: UmbraIR Hash Function Annotations: Steps 4, 5, 6

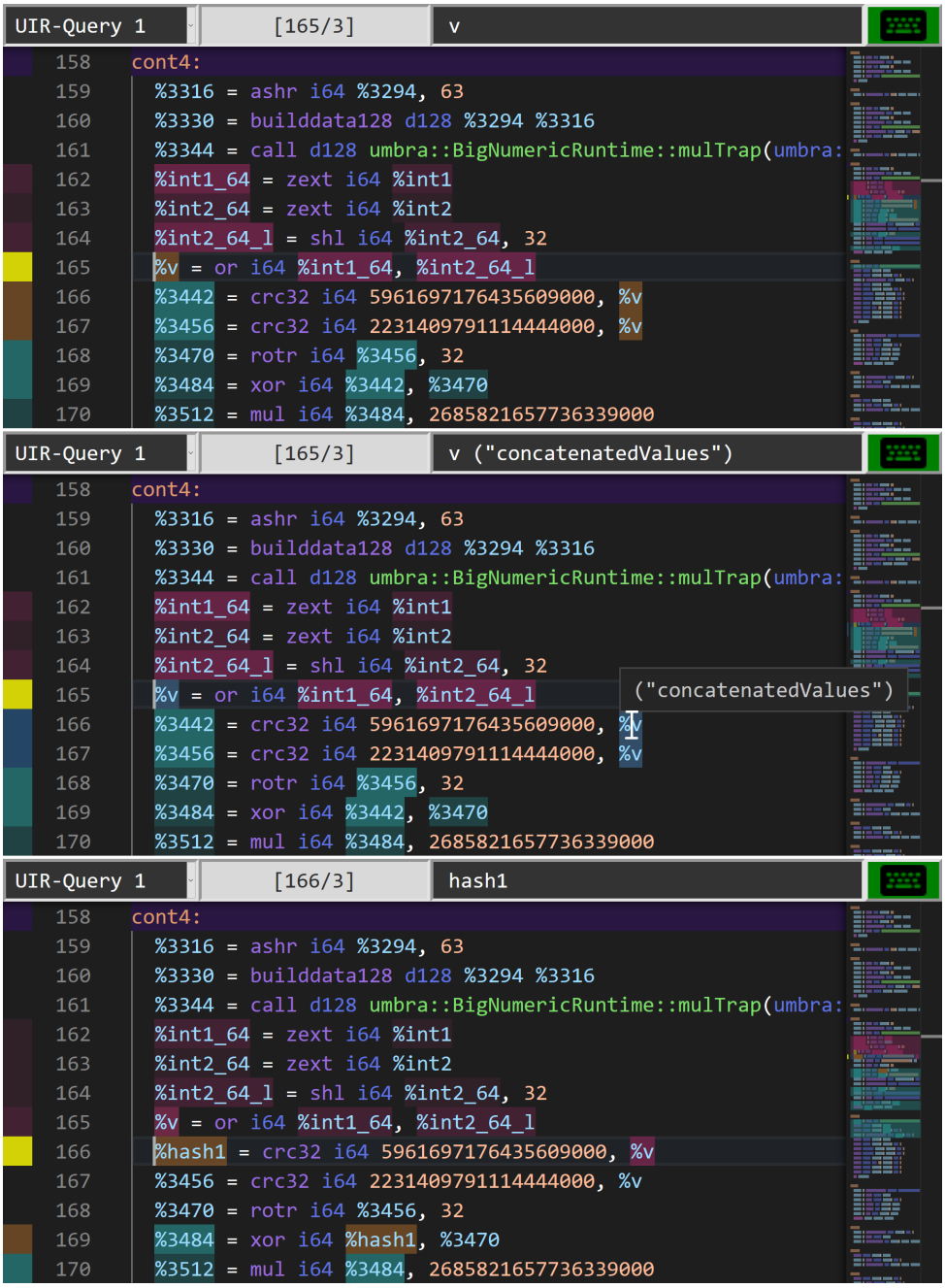


Figure A.5.: UmbraIR Hash Function Annotations: Steps 7, 8, 9

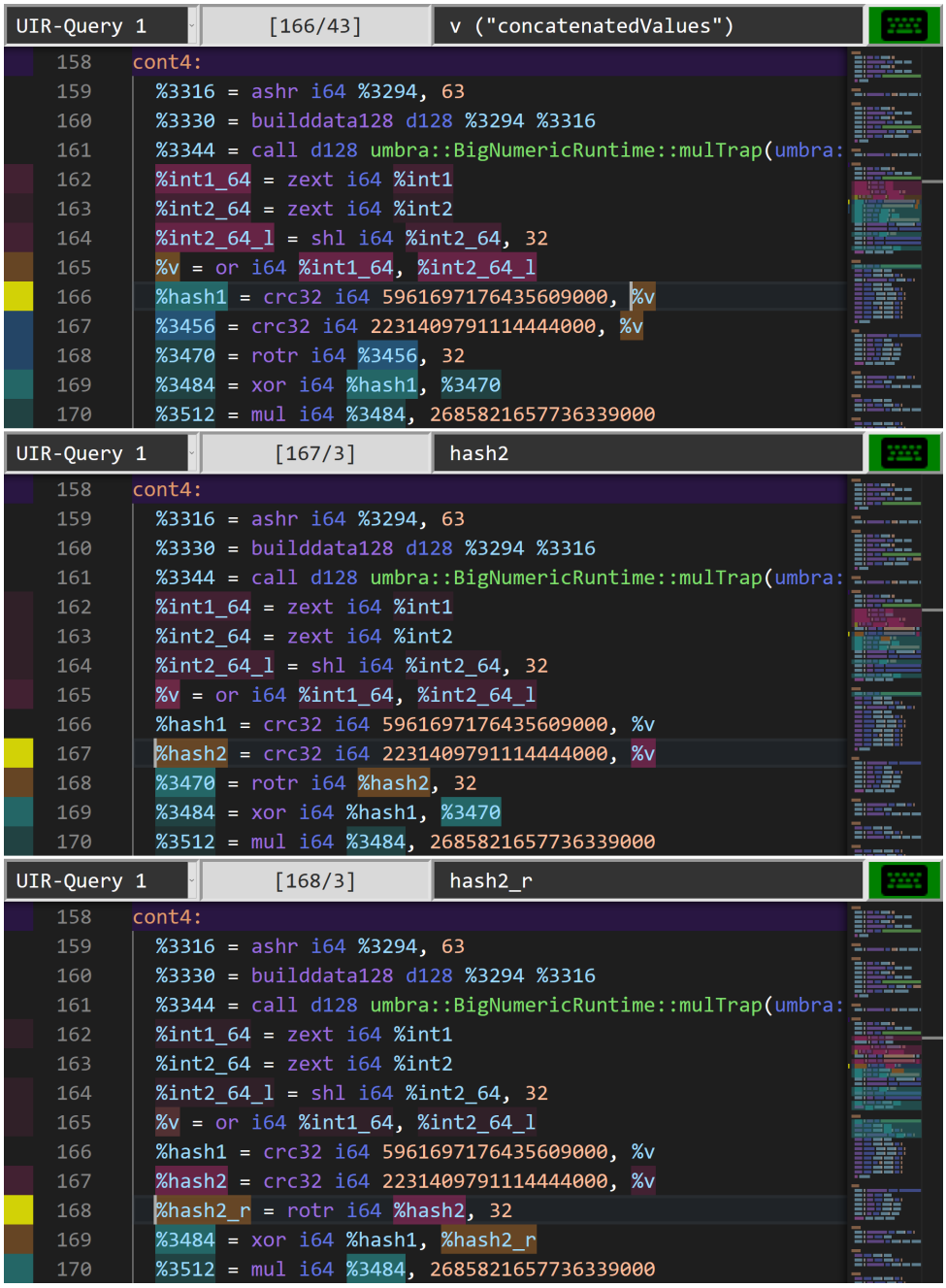


Figure A.6.: UmbraIR Hash Function Annotations: Steps 10, 11, 12

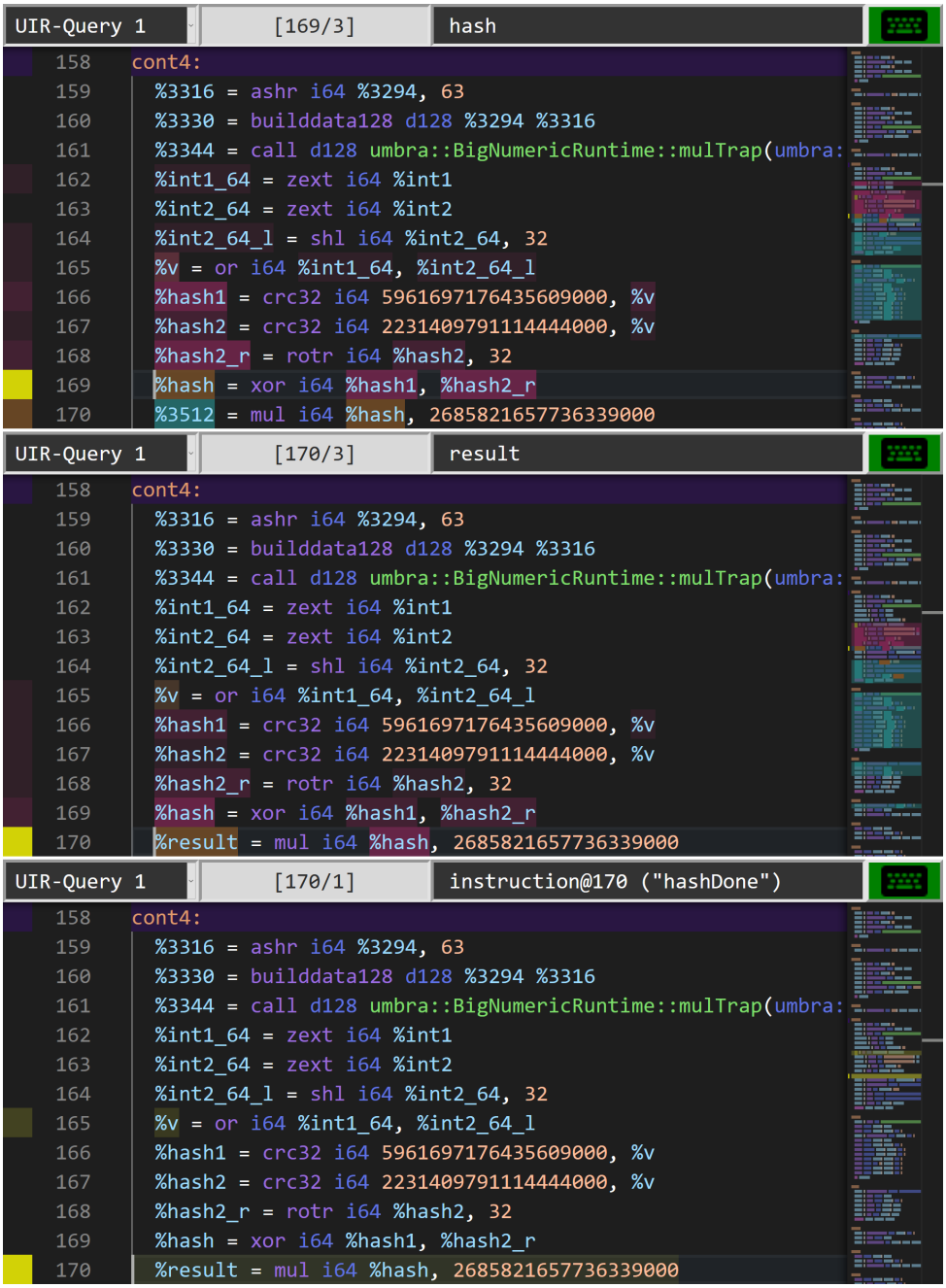


Figure A.7.: UmbraIR Hash Function Annotations: Steps 13, 14, 15

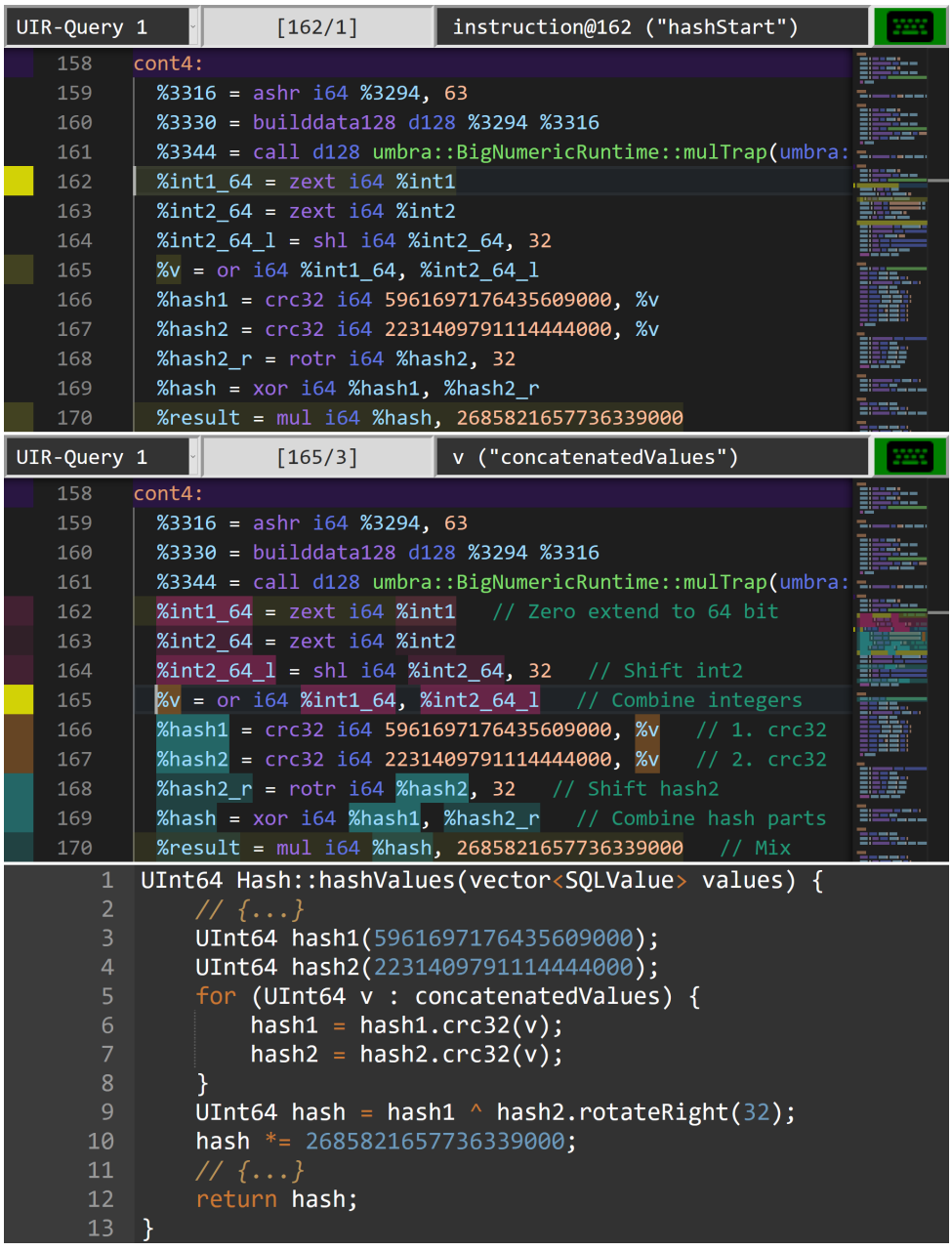


Figure A.8.: UmbraIR Hash Function Annotations: Steps 16, 17 & C++

List of Figures

2.1. Umbra Intermediate Representation: Reference Query 5
2.2. Umbra Intermediate Representation: Language Structure 6

3.1. UIR-Editor: Project Architecture 12

4.1. UIR-Editor: User Interface 19
4.2. UIR-Editor: User Interface - Status Options 20
4.3. UIR-Editor: Code Editor Window 20
4.4. UIR-Editor: Syntax Highlighting 21
4.5. UIR-Editor: Target Tree Modal 23
4.6. UIR-Editor: Child and Parent Highlighting 24
4.7. UIR-Editor: Query Annotations 25

5.1. Code Presentation: Before and After 30
5.2. Variable Assignment: Compare Function 32
5.3. Control Flow: Sort Loop Function 33
5.4. Code Comparison: C++ Hash Function [16] 35
5.5. Code Comparison: UmbraIR Hash Function - Before and After 35

A.1. UIR-Editor: Keyboard Shortcuts (Icons by Flaticon [2]) 43
A.2. UIR-Editor: Keyboard Shortcuts - Legend (Icons by Flaticon [2]) 44
A.3. UmbraIR Hash Function Annotations: Steps 1, 2, 3 45
A.4. UmbraIR Hash Function Annotations: Steps 4, 5, 6 46
A.5. UmbraIR Hash Function Annotations: Steps 7, 8, 9 47
A.6. UmbraIR Hash Function Annotations: Steps 10, 11, 12 48
A.7. UmbraIR Hash Function Annotations: Steps 13, 14, 15 49
A.8. UmbraIR Hash Function Annotations: Steps 16, 17 & C++ 50

List of Tables

1.1. Average UIR code lines per SQL operator from 22 TPC-H requests . . . 2

Bibliography

[1] Mike Bostock. *D3js API Documentation*. 2020. URL: https://github.com/d3/d3/blob/master/API.md (visited on 09/16/2020).

[2] Freepik Company. *Flaticon Website*. 2020. URL: https://flaticon.com/de (visited on 11/17/2020).

[3] Unicode Consortium. *Unicode Overview*. 2020. URL: https://home.unicode.org/basic-info/overview (visited on 11/23/2020).

[4] World Wide Web Consortium. *Cascading Style Sheets*. 2020. URL: https://w3.org/Style/CSS (visited on 11/27/2020).

[5] World Wide Web Consortium. *W3C HTML*. 2020. URL: https://w3.org/html (visited on 11/27/2020).

[6] Microsoft Corporation. *Monaco Editor API Documentation*. 2020. URL: https://microsoft.github.io/monaco-editor/api/index.html (visited on 11/27/2020).

[7] Microsoft Corporation. *Monarch Documentation*. 2020. URL: https://microsoft.github.io/monaco-editor/monarch.html (visited on 11/27/2020).

[8] Microsoft Corporation. *TypeScript Language Documentation*. 2020. URL: https://typescriptlang.org (visited on 11/27/2020).

[9] Microsoft Corporation. *Visual Studio Code Editor*. 2020. URL: https://code.visualstudio.com (visited on 10/01/2020).

[10] Douglas Crockford. *Introducing JSON*. 2020. URL: https://json.org/json-en.html (visited on 11/27/2020).

[11] TUM: Lehrstuhl für Datenbanksysteme. *Umbra Database System*. 2020. URL: https://umbra.db.in.tum.de (visited on 11/09/2020).

[12] Ezoic. *Ascii Table*. 2020. URL: http://asciitable.com (visited on 11/25/2020).

[13] OpenJS Foundation. *Node.js Documentation*. 2020. URL: https://nodejs.org/en/docs (visited on 11/27/2020).

[14] Facebook Incorporated. *React Framework Documentation*. 2020. URL: https://reactjs.org (visited on 11/27/2020).

[15] npm Incorporated. *npm Documentation*. 2020. URL: https://docs.npmjs.com (visited on 11/27/2020).

[16] Timo Kersten, Viktor Leis, and Thomas Neumann. "Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra." In: 2020.

[17] André Kohn, Viktor Leis, and Thomas Neumann. "Adaptive Execution of Compiled Queries." In: *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 2018, pp. 197–208. DOI: 10.1109/ICDE.2018.00027.

[18] Chris Lattner. *LLVM Language Reference Manual*. 2020. URL: https://llvm.org/docs/LangRef.html (visited on 09/18/2020).

[19] Bram Moolenaar. *Vim Editor Documentation*. 2020. URL: https://vim.org/docs.php (visited on 09/17/2020).

[20] G. E. Moore. "Cramming More Components Onto Integrated Circuits." In: *Proceedings of the IEEE* 86.1 (Jan. 1998), pp. 82–85. ISSN: 1558-2256. DOI: 10.1109/JPROC.1998.658762.

[21] Thomas Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware." In: *Proc. VLDB Endow.* 4.9 (2011), pp. 539–550. DOI: 10.14778/2002938.2002940.

[22] Thomas Neumann and Michael J. Freitag. "Umbra: A Disk-Based System with In-Memory Performance." In: *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. cidrdb.org, 2020.

[23] TPC. *TPC-H Homepage*. 2020. URL: http://tpc.org/tpch (visited on 09/29/2020).