HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

Supporting Swift 4 Generics in Tifig

Master Thesis

University of Applied Sciences Rapperswil

Fall Semester 2017/18

Author:	Mario Meili
Advisor:	Prof. Dr. Farhad D. Mehta
Industry Partner:	Institute for Software (IFS) HSR

Abstract

Tifig is a Swift Integrated Development Environment (IDE) based on Eclipse. Since Swift has been open-sourced, a compiler for Linux systems became part of the Swift project, allowing the development of Swift applications on platforms other than macOS. Because Xcode is only available on macOS, Tifig aims to be the preferred alternative. With its newest release, many language features and syntax changes were introduced to Swift. This resulted in inconsistencies between the behaviour of the Swift compiler and Tifig. The aim of this thesis was to improve the existing Tifig IDE by:

- *Supporting the new enhancements in Swift's generics system.*
- *Consolidating the existing documentation in a form that promotes continual development.*

To achieve this, Tifig's parser and indexer were extended to support the newly introduced syntax and to ensure correct indexing order and index correctness. Tifig's type checker was modified to enable correct type resolution. To improve the current documentation, a study of multiple large and successful open source projects was conducted.

The results of this thesis encompass:

- *The newest alpha release of Tifig*
- *An extensive documentation for Tifig contributors*
- *Introductory tutorials for first-time Tifig users*
- *A short theoretical study on how to improve the performance and resolve current issues of the Swift type checker*

Acknowledgements

Firstly, I would like to thank my thesis advisor Prof. Dr. Farhad D. Mehta, who was always willing to lend a helping hand when I ran into problems.

Secondly, I would like to thank my colleague and friend Toni Suter, who laid the groundwork for this thesis by developing Tifig from scratch. In addition, he took over the role of a technical advisor, providing me with all the necessary knowledge about his plug-in infrastructure and the Swift programming language in general.

In addition, I would like to thank the users of the Swift users group mailing list for helping me to verify my findings regarding Swift compiler bugs in record time.

Finally, I would like to express my gratitude to my girlfriend for providing me with unfailing support and continuous encouragement.

Contents

List of Abbreviations

HSR Hochschule für Technik Rapperswil

IDE Integrated Development Environment

IFS Institute for Software

AST Abstract Syntax Tree

CI Continuous Integration

1. Introduction

Swift is a multi-paradigm programming language developed by Apple Inc. It was released in 2014. Only one year later, Swift became an open-source project. This opened up the possibility for the community to actively influence the further development of the language. That is the reason why Swift still experiences many changes between major releases. In Swift 4, which was released during the period of this thesis, namely in September 2017, around 30 proposals from the community were implemented [27].
Swift is not the first open-source project of Apple Inc. In 2007, the clang compiler for C, C++ and Objective-C (and other languages) was open-sourced as well [2]. However, the contributors had no influence on the programming languages clang compiles. Open-sourcing Swift can therefore be seen as a first, in that the community can actively influence the development of the language itself. Another first was the release of a Swift compiler for Linux systems. This opened the door for developers of language tooling to even cross the boundaries of operating systems. One such attempt to allow Swift application development cross-platform is Tifig. Tifig is a Swift IDE based on Eclipse [5]. Its development was started as a master project thesis by Toni Suter [30].

1.1. Problems

Tifig currently supports Swift version 3.1. The changes introduced to the language make Tifig unusable for the development of Swift 4 applications. This in turn makes Tifig incapable of competing with other products. Therefore, Tifig needs to be developed further.
So far, Tifig has been a single person project. This changed with the beginning of this thesis. It is possible, that even more contributors will work on Tifig in the future. The documentation of the project is currently almost completely done in [30]. This is a problem, because a thesis report is not a suitable format of documentation for promoting continual development.

1.2. Goals

The aim of this thesis is to develop the Tifig IDE further, in order to support Swift 4 generics. Additionally, tasks that promote the practical use and further development of Tifig will be undertaken as per the resources available. This shall be achieved by

- modifying Tifig's parser to support the newly introduced syntax,
- adapting Tifig's indexer to guarantee a correct indexing order and indexing correctness,
- extending Tifig's type checker by the necessary constraints to enable correct type resolution for Swift 4 and
- consolidating existing documentation and documentation relevant to the tasks executed as part of this project in a form that promotes continual development of Tifig.

The following proposals shall be implemented in order to support Swift 4 generics in Tifig:

1. SE-0142: Permit where clauses to constrain associated types [26]
2. SE-0148: Generic Subscripts [22]
3. SE-0156: Class and Subtype existentials [25]

1.3. Document Structure

The remainder of this document is organized as follows. Chapter 2 summarizes the architecture and functionality of Tifig as described in [30]. A short section describes an important change in Tifig's infrastructure, that was implemented after the original master thesis. In addition, a brief overview of related work is given. The analysis of Tifig's infrastructure and the implementation of the above proposals is documented in Chapter 3. In Chapter 4, the Swift type checker is studied more closely and improvements concerning performance are proposed. Chapter 5 describes the process of finding the best possible documentation format for the Tifig project. The results of this thesis are gathered and discussed in Chapter 6 and finally, Chapter 7 provides a conclusion and proposes future work to further improve the Tifig IDE.

2. Background & Related Work

As already mentioned in Chapter 1, the work conducted during the period of this thesis builds on the already existing Tifig project. Because the architecture and especially the functionality of Tifig are not trivial, but necessary to understand the proceedings in this report, a minimal overview of the most important components and processes is given in Section 2.1. Section 2.2 describes an important change that occured between the end of the original master thesis and the start of this thesis. Related projects are listed and briefly discussed in Section 2.3.

2.1. Tifig Overview

Tifig is a collection of plug-ins that are built on top of the Eclipse IDE infrastructure. It makes use of the Eclipse Plug-in Development Environment [7] to bring the plug-ins together. How this works is not easily explained and would go beyond the scope of this thesis. Interested readers can consult [6], which is a short tutorial about extending Eclipse. Further, it is worth mentioning that the entire source code of Tifig is written in Java.
In this overview section, the components of Tifig are discussed briefly, pointing out the ones relevant for this thesis. These are then looked at separately in more detail. A closing example demonstrates the mechanisms described.

2.1.1. Plug-ins

The following are the plug-ins Tifig is built of:

- `ch.hsr.ifs.tifig.branding`
- `ch.hsr.ifs.tifig.branding.feature`
- **`ch.hsr.ifs.tifig.core`**
- `ch.hsr.ifs.tifig.feature`
- `ch.hsr.ifs.tifig.product`
- `ch.hsr.ifs.tifig.target`

- **ch.hsr.ifs.tifig.ui**
- ch.hsr.ifs.tifig.updatesite

The plug-ins `branding`, `branding.feature`, `feature`, `product`, `target` and also `updatesite` are all important for a functioning plug-in environment. They are needed for building the final product Tifig. However, the business logic of the Tifig IDE is not part of these plug-ins and therefore, only the plug-ins `core` and `ui` are relevant for this thesis. The two additional plug-ins shown below have the sole purpose of testing their functionality:

- **ch.hsr.ifs.tifig.core.tests**
- **ch.hsr.ifs.tifig.ui.tests**

Everything related to implementations in Tifig will relate to one of the plug-ins above highlighted in bold.

2.1.2. Components

Figure 2.1 shows an overview of the components relevant for writing and maintaining source code. As can be seen, all components are either affiliated with the `core` plug-in or the `ui` plug-in. The original figure has been taken from [30] and was adjusted to highlight the remarks of this section.
Note, that not all the processes shown in the figure are relevant in the scope of this thesis. For example, building an executable out of the source code and also launching said executable are not of interest, because this is done using the Swift compiler from Apple Inc. That is why the *Builder* and *Launcher/Debugger* component are shown in a faded manner.
Of interest is what happens when a *User* enters or changes source code. As is shown, the *Editor* component notifies the *Reconciler* about such changes, which in turn triggers reconciliation. In other words, the *Reconciler* component notifies Tifig's core components, that the changed source code has to be processed. The *Lexer* component takes the source code as its input and generates tokens. These tokens are then processed by the *Parser* component, resulting in an Abstract Syntax Tree (AST) representation of the source code. The AST is used by the *Indexer* component for semantic analysis. Both the *Parser* and *Indexer* components can generate and set markers for the *Editor*. This closes the circle.
Why are the *Editor* and *Reconciler* faded? The reason behind this is simple. The mechanisms provided by these components are not affected by changes to the syntax and semantics of Swift. The *Reconciler* solely needs to know the locations and corresponding offsets of changes that occurred. It does not even need to know

what kind of file has been edited. The *Editor*'s responsibility is to trigger the *Reconciler* and to display the markers generated by the *Parser* and the *Indexer*. These markers in turn are nothing else than source code locations, corresponding offsets and a message. Again, the editor does not need to know the underlying file format.

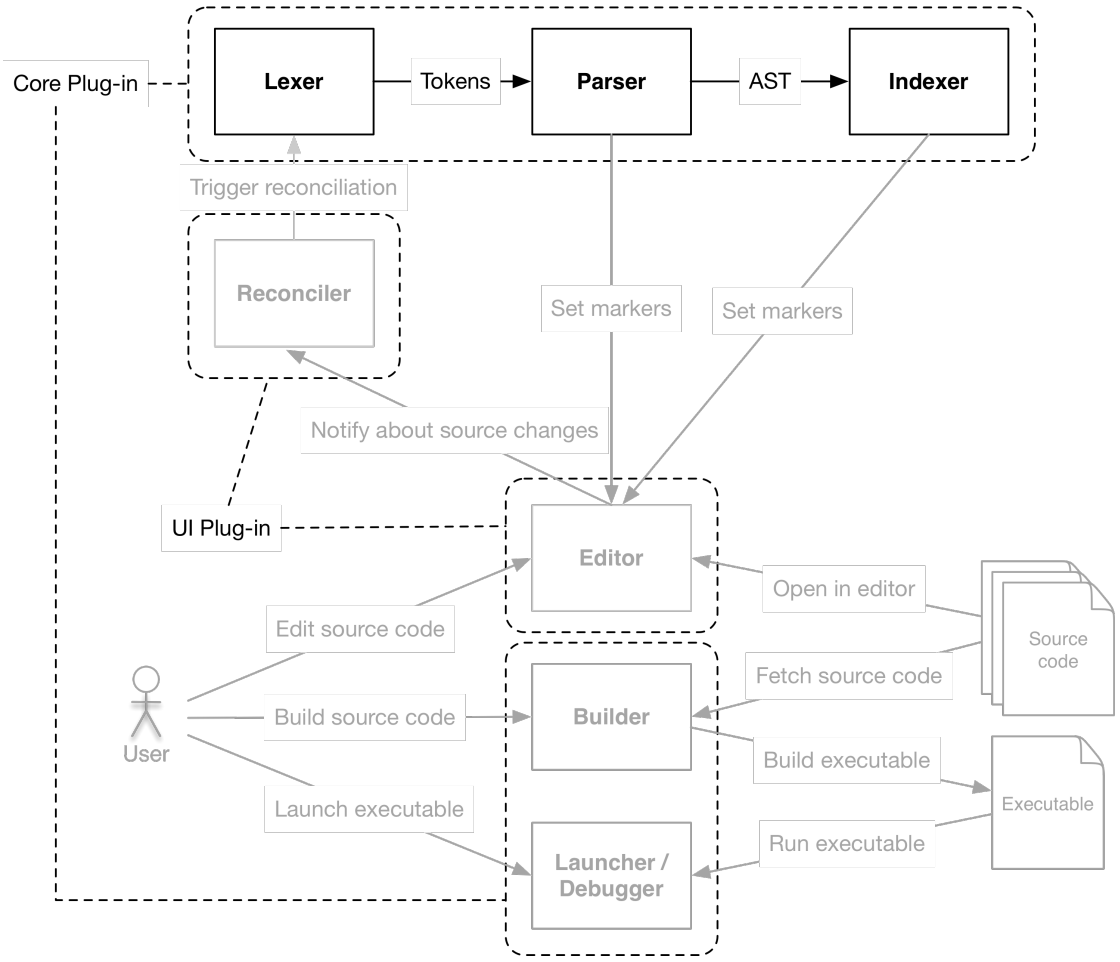

Figure 2.1.: *Tifig components*

Concluding the component overview, the plug-ins relevant for this thesis are reduced to the following:

- `ch.hsr.ifs.tifig.core`
- `ch.hsr.ifs.tifig.core.tests`

The *Lexer*, *Parser* and *Indexer* components will be described more closely in the following sections.

2.1.3. Lexer

The *Lexer* takes a stream of characters and generates tokens for the *Parser*. The `core` plug-in of Tifig provides a package `lexer`, which contains all the necessary functionality to achieve that. The contents of the package are shown in Figure 2.2.

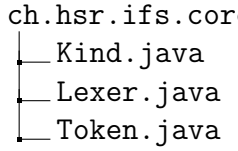

Figure 2.2.: *Lexer package*

The class `Kind` is an enum that contains all the different kinds a token can represent. An instance of class `Token` has such a `Kind` property, an offset, a length and a property `text`, that contains the `String` of the source code it represents. The main functionality is implemented in the `Lexer` class, as can be seen in Listing 2.1.

```
1  public class Lexer {
2
3    // ...
4
5    public Lexer(char[] source, String filePath) {
6      this.source = source;
7      this.filePath = filePath;
8      // ...
9    }
10
11   // ...
12
13   public Token nextToken() {
14     lastToken = lex();
15     return lastToken;
16   }
17
18   public List<Token> allTokensExceptComments() {
19     final List<Token> allTokens = new ArrayList<>();
20     Token nextToken = null;
21     do {
22       nextToken = nextToken();
23       if(nextToken.is(Kind.COMMENT)) {
24         continue;
25       }
26       allTokens.add(nextToken);
27     } while(nextToken.isNot(Kind.EOF));
28     return Collections.unmodifiableList(allTokens);
29   }
30
31   // ...
32
33 }
```

Listing 2.1: *Lexer*

It is possible to either request the next token not consumed so far, or alternatively, to request the entire list of tokens using the method `allTokensExceptComments()`. Note, that this is preexisting code and not part of the implementations done during this thesis.

2.1.4. Parser

As was the case for the *Lexer*, the *Parser* code is located in a separate `parser` package within the `core` plug-in. The contents of the package are shown in Figure 2.3.

```
ch.hsr.ifs.core.parser
├── AttributeParser.java
├── DeclParser.java
├── ExprParser.java
├── MismatchedTokenException.java
├── NoViableAltException.java
├── Parser.java
├── ParserModule.java
├── PatternParser.java
├── RecognitionException.java
├── StmtParser.java
└── TypeParser.java
```

Figure 2.3.: *Parser package*

The classes shown above can be grouped into three categories:

1. There are three classes for specific exceptions that can occur during the parse process. Because they are not relevant for the implementations later in this document, they will not be further discussed.

2. Parser modules all extend the `ParserModule` class. They can be identified by their names conforming to `*Parser.java`. Each of these modules is responsible for parsing specific Swift language constructs. For example, the `DeclParser` module is responsible for parsing declarations and so on.

3. The `Parser` class represents the entry point of the parse logic. Listing 2.2 shows the relevant methods.

```
public class Parser {
  private final DeclParser declParser = new DeclParser(this);
  private final StmtParser stmtParser = new StmtParser(this);
  // the same for all other parser modules

  // ...

  public Parser(String filePath, List<Token> tokens, int tokenIndex) {
    this.filePath = filePath;
    this.tokens = tokens;
    this.tokenIndex = tokenIndex;
    declParser.setParserModules(declParser, stmtParser, exprParser,
        typeParser, patternParser, attributeParser);
    stmtParser.setParserModules(declParser, stmtParser, exprParser,
        typeParser, patternParser, attributeParser);
    // the same for all other parser modules
  }

  public Parser(Lexer source) {
    this(source.getFilePath(), source.allTokensExceptComments(), 0);
  }

  public static SourceFile parse(String source, String filePath) {
    final Lexer lexer = new Lexer(source.toCharArray(), filePath);
    final Parser parser = new Parser(lexer);
    return parser.parse();
  }

  public SourceFile parse() {
    // ...
    final List<IDecl> decls = new ArrayList<>();
    // ...
    while(la(1).isNot(Kind.EOF)) {
      // ...
      try {
        IDecl decl = null;
        if(declParser.isStartOfDecl(1)) {
          decl = parse(declParser::decl);
        } else if(StmtParser.isStartOfStmt(la(1)) ||
            ExprParser.isStartOfExpr(la(1))) {
          final IStmt stmt = parse(stmtParser::stmt);
          decl = new TopLevelStmtDecl(stmt);
        } // ...
        decls.add(addTokens(decl));
      } // ...
    }

    // link all nodes to their parent
    final SourceFile ast = addTokens(new SourceFile(decls.toArray(
        new IDecl[decls.size()]), filePath));
    ast.accept(new ASTGenericVisitor() {
      // ...
    });
    return ast;
  }

  // ...

}
```

Listing 2.2: *Parser*

The `Parser` class has a property for each of the parser modules. In the constructor, each of these properties has the other modules set as their own parser modules (lines 12 to 16). The static `parse()` method (lines 23 to 27) instantiates a new `Lexer` and then instantiates a new `Parser`, providing the `Lexer` instance as the argument of the constructor. Finally, it calls the `parse()` method (lines 29 to 55) on the newly created parser, returning the AST represented by an instance of class `SourceFile`. In this top-level parse method, it is checked whether the first token consumed is the start of a declaration or a statement. This must be the case when starting to parse a new file. Depending on the result of this check, the corresponding module calls its own parse method (here `decl()` or `stmt()`).
This describes a recursive descent parser. The tokens are dispatched to the appropriate modules, where they are parsed. After that, the so created `ASTNode` instances are linked to their parents (lines 49 to 53), which completes the AST.
The `SourceFile` class representing the AST and the classes for all specializations of `ASTNode` are not part of the `parser` package. Instead, they can be found in the `ast` package and the nested packages thereof. Figure 2.4 shows the nested package structure, leaving out the Java files.

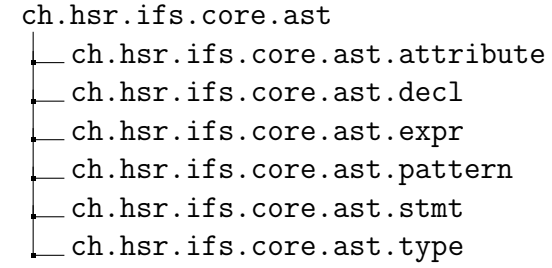

Figure 2.4.: *AST package*

Again, the code presented in this section was already part of Tifig's implementation and therefore not part of the implementations undertaken for this thesis.

2.1.5. Indexer

The *Indexer* is the largest and most complex component of Tifig. Because of that, the implementation of the *Indexer* ranges over multiple packages. Figure 2.5 shows the `indexer` package, which has several nested packages. Because of the large amount of code involved in the indexing process, only the most important classes are briefly discussed in this section. These classes are contained in the parent package `indexer` and are listed in the figure below. Next to the `Indexer` class, the classes `DefinitionPassVisitor`, `TypeAnnotationPassVisitor` and not to forget

the `TypeCheckPassVisitor` best describe what is going on during the indexing process.

```
ch.hsr.ifs.core.indexer
├── // ...
├── DefinitionPassVisitor.java
├── Indexer.java
├── TypeAnnotationPassVisitor.java
├── TypeCheckPassVisitor.java
├── // ...
├── ch.hsr.ifs.core.indexer.bindings
│   └── // contains classes for the various kinds of bindings
├── ch.hsr.ifs.core.indexer.cs
│   └── // contains everything necessary for constraint generation
│       and resolution
├── ch.hsr.ifs.core.indexer.scope
│   └── // contains classes for the different scopes
├── ch.hsr.ifs.core.indexer.swiftmodel
│   └── // contains everything related to the Swift model
└── ch.hsr.ifs.core.indexer.types
    └── // contains classes for various index types
```

Figure 2.5.: *Indexer package*

In order to perform the semantic analysis of the Swift source code, the *Indexer* makes extensive use of the visitor pattern [24]. In fact, performing the indexing process requires multiple traversals of the AST generated by the *Parser*. Figure 2.6 shows the inner architecture of the *Indexer*. This figure is taken from [30] in its original form.

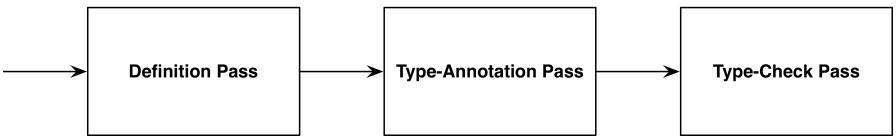

Figure 2.6.: *Indexer components*

Listing 2.3 shows the corresponding implementation of the `Indexer` class. The static method `indexFiles()` takes a list of `SwiftFile` objects and a `StdlibModule` as its arguments. The method then calls `definitionPass()` on all the files provided (line 6). What `definitionPass()` does is creating a new `DefinitionPasssVisitor` and calling `accept()` with it on the AST of the current file (lines 16 to 24). Lines

7 to 11 encompass preparations for the call of `typeAnnotationPass()` (line 12), again executed for each of the given files. Finally, the same structure is followed to visit the AST of each file with a `TypeCheckPassVisitor` (line 13).

```
1  public class Indexer {
2    // ...
3
4    public static void indexFiles(List<SwiftFile> files,
5        StdlibModule stdlibModule) {
6      files.forEach(file -> definitionPass(file, stdlibModule));
7      files.forEach(file -> {
8        file.addImports();
9        file.connectExtensions();
10     });
11     files.forEach(SwiftFile::prepareForTypeAnnotationPass);
12     files.forEach(Indexer::typeAnnotationPass);
13     files.forEach(Indexer::typeCheckPass);
14   }
15
16   private static void definitionPass(SwiftFile file,
17       SwiftModule stdlibModule) {
18     final DefinitionPassVisitor definitionPassVisitor =
19         new DefinitionPassVisitor(file);
20     file.getAST().accept(definitionPassVisitor);
21     if(stdlibModule != null) {
22       file.addImportedBinding(stdlibModule);
23     }
24   }
25
26   private static void typeAnnotationPass(SwiftFile file) {
27     final TypeAnnotationPassVisitor typeAnnotationPassVisitor =
28         new TypeAnnotationPassVisitor();
29     file.getAST().accept(typeAnnotationPassVisitor);
30   }
31
32   private static void typeCheckPass(SwiftFile file) {
33     final TypeCheckPassVisitor typeCheckPassVisitor =
34         new TypeCheckPassVisitor(file);
35     file.getAST().accept(typeCheckPassVisitor);
36   }
37
38   // ...
39 }
```

Listing 2.3: *Indexer*

In the following, each of the three passes is shortly explained. Note, that the bare description is possibly not enough to fully understand all the mechanics of the indexer. The intention is to give a rough idea of the processes. As usual, a more detailed explanation can be found in [30], the original documentation of Tifig.

Definition Pass

The `DefinitionPassVisitor` traverses the AST of each given file. For each declaration of a name it visits, a binding for that name and kind is created. In addition, the binding is stored in the proper lexical scope. Because each scope can have multiple child scopes, but can only have one parent, a so called scope tree emerges as the result of this pass. Note, that the name of each binding is set after this pass. However, the binding has no type assigned so far.

Type-Annotation Pass

The type-annotation pass looks for `ASTNode` occurrences that stand for explicit type annotations. It then transforms the AST types into index types. Finally, the type of the bindings that have such an explicit type annotation can be set. The result is the same scope tree from the definition pass enriched with some type information.

Type-Check Pass

During the type-check pass, the `TypeCheckPassVisitor` traverses all expressions of the AST. While doing so, the expressions are type-checked. To do this, Tifig uses a constraint-based type checker. If all types check out, each subexpression is assigned a type. Also, each binding that has no type set so far is assigned the inferred type. The type inference happens in a bi-directional manner, because Swift allows type information to flow from the bottom of the AST to the top and also from the root of the expression tree down to the leaves [30]. If all expressions are successfully type-checked, the scope tree is filled completely. Every binding has assigned a type.

2.1.6. A Practical Example

To visualize the processes described so far, a practical example shall be given in this section. The example encompasses a selection of all the steps undertaken by the core components described above, so that all mechanics are shown at least once and at least on an excerpt of the example code.
Listing 2.4 shows the code on which the steps will be demonstrated. The generic function `_min()` is declared, taking two arguments of the generic type `T` (lines 1 to 3). The generic parameter clause shows that type `T` must conform to the `Comparable` protocol. `Comparable` is part of the Swift standard library and will not be examined closer. The `_min()` function simply returns the smaller of the two arguments provided. The function is then called and the result is assigned to the constant `x` (line 5). Finally, `x` is printed to the console (line 6).

```
1  func _min<T: Comparable>(_ a: T, _ b: T) -> T {
2    return a < b ? a : b
3  }
4
5  let x = _min(5,10)
6  print(x)
```

Listing 2.4: *Generic function call*

Assuming that the last key stroke was made, finishing the `print()` statement on line 6 of the example code, the *Editor* notices this change and triggers the *Reconciler*. The *Reconciler* in turn triggers the *Lexer*. This is where this example starts. Figure 2.7 shows how the *Lexer* component transforms the character stream `let x = _min(5,10)` into tokens. Each of the tokens contains the original text, the offset and the length of the text. Of course, all the other lines of the example code are transformed as well.

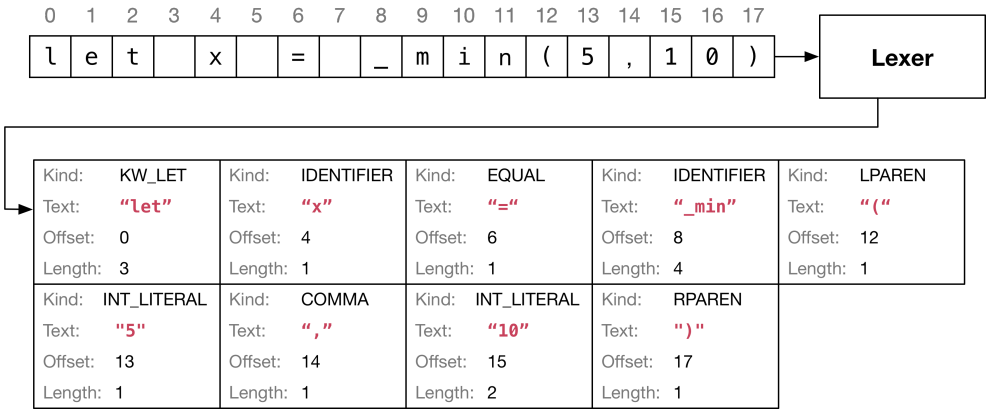

Figure 2.7.: *Tokens generated by the Lexer*

The *Parser* takes the list of tokens generated by the *Lexer* as its input. Figure 2.8 shows the AST output generated for the tokens of the statement under inspection before. Note, that this is only an excerpt of the AST.
Of special interest is the subtree starting with the `FunctionCallExpr` node. This expression is composed of an `IdentifierExpr` and a `ParenthesizedExpr`. The identifier holds the function's name `_min`. The parenthesized expression holds the two literal expressions `5` and `10`. Later in this section, the type-checking process will be demonstrated on this exact function call expression.

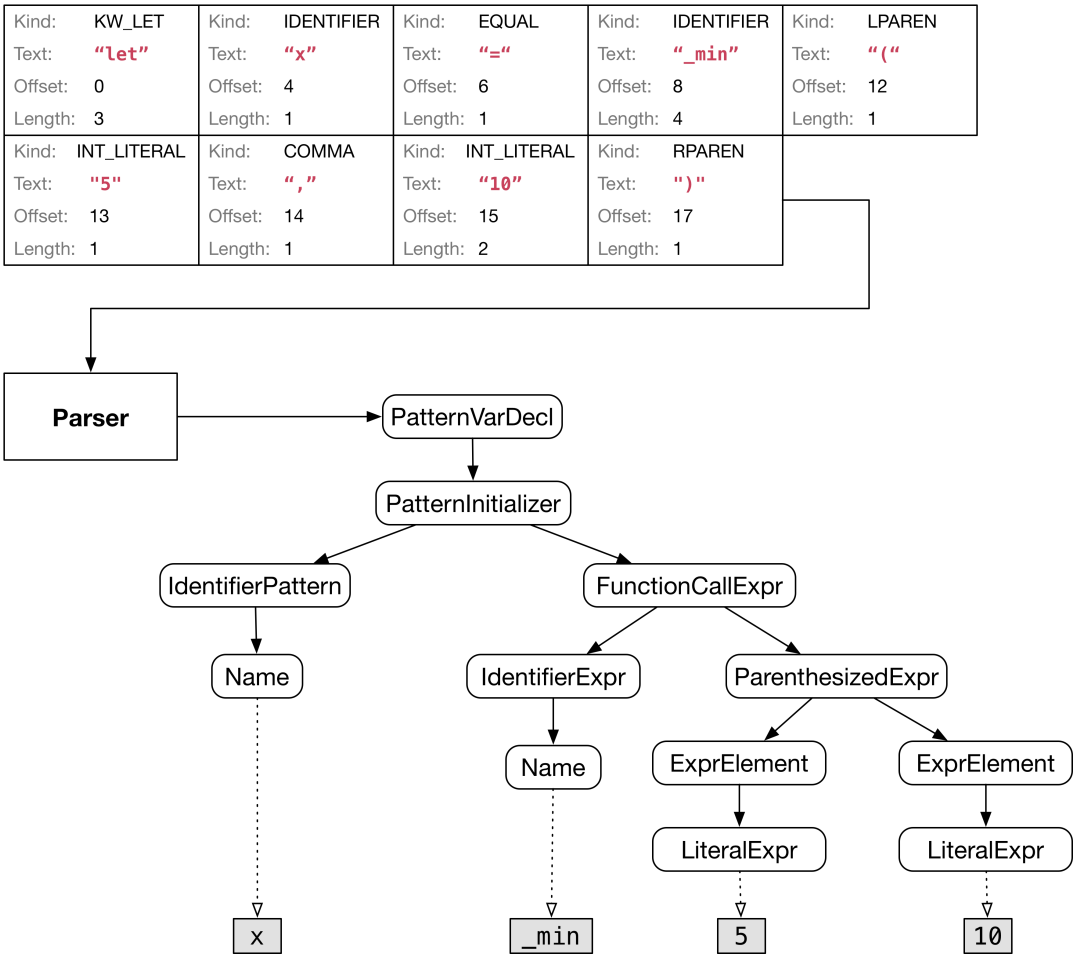

Figure 2.8.: *AST generated by the Parser*

The *Parser* has finished its job and the *Indexer* takes over. As discussed earlier, three steps have to be performed.
The first of these steps is the definition pass. The `DefinitionPassVisitor` traverses the AST and generates the scope tree shown in Figure 2.9. This time, the figure shows the result for the entire example code. Note, that each binding is enclosed by a scope and that none of the bindings have a type so far. One special binding is the `ArchetypeType` binding in the generic parameter scope. It represents the generic type parameter `T`. It has special fields worth discussing a little more:

(*) The same type field is set, if a constraint demands that `T` has to be equal to another type. This can be specified in a generic where clause.

(**) The fields for adopted protocols and the conformance class are pretty much

self-explanatory. If `T` has to conform to one or more protocols, the protocol binding is added to the list of adopted protocols. If `T` must be a subclass of another class, this class binding is set as the conformance class.

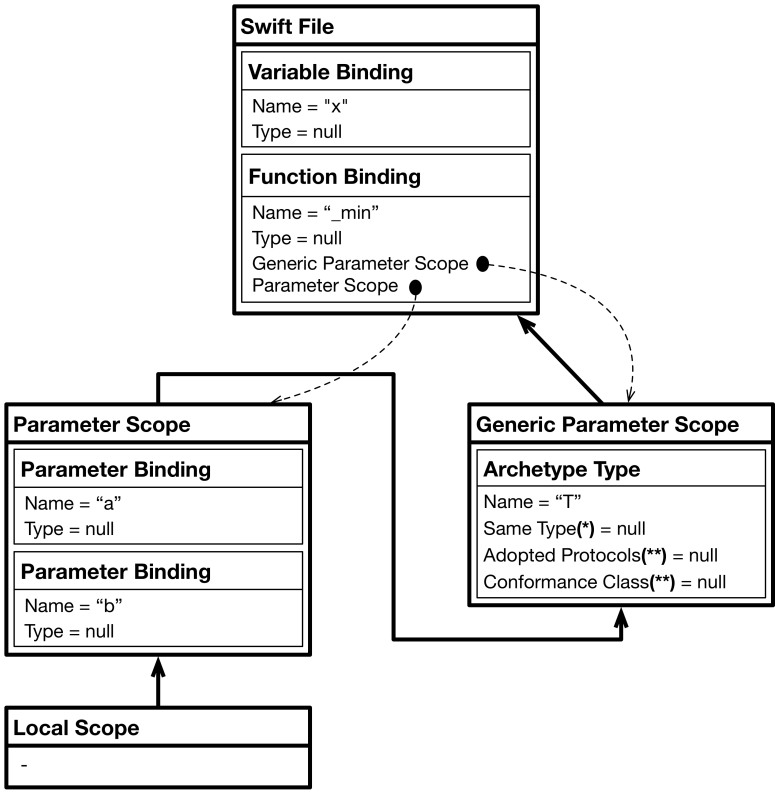

Figure 2.9.: *Scope tree generated during definition pass*

The second step is the type annotation pass. Again, the AST is traversed and wherever nodes are passed that represent a type annotation, an index type is created and assigned to the corresponding binding.
The updated scope tree can be seen in Figure 2.10. The changes are highlighted in bold. A first observation is that the types for the parameter bindings of `a` and `b` have been set to `T`. Since the type of the function parameters is now known and the return type of the function is annotated, the `TypeAnnotationPassVisitor` correctly set the type of the function binding to `(T, T) -> T`. It is also noteworthy, that the list of adopted protocols of the archetype binding `T` has been updated. It now contains the protocol binding for the protocol `Comparable`, that is defined in Swift's standard library. The final and most important observation is that the variable binding of the variable `x` has still no type assigned. This makes

perfect sense, because no type annotations are provided in the statement where `x` is declared.

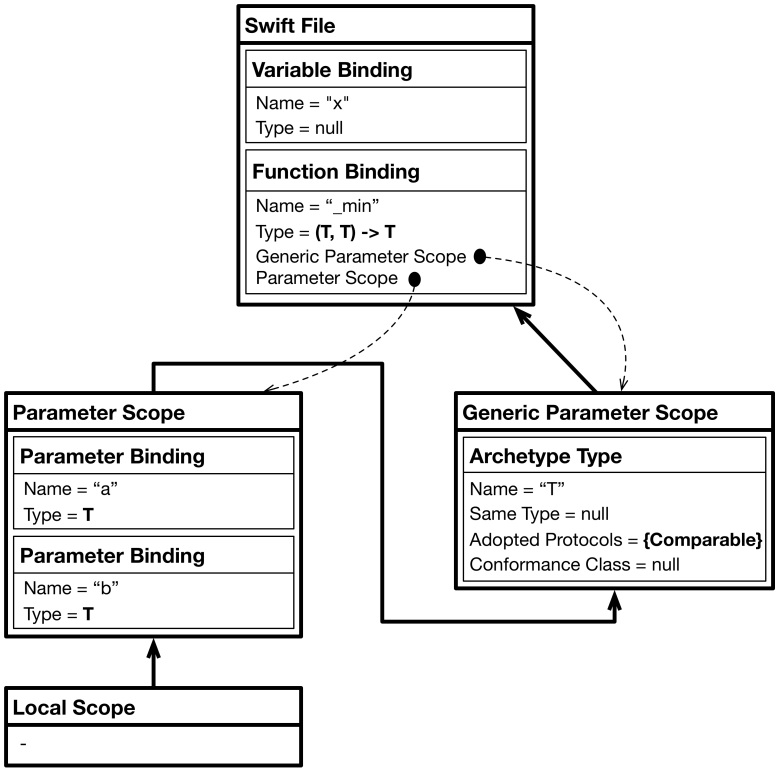

Figure 2.10.: *Scope tree updated after type definition pass*

To complete the entries of the scope tree, a third and last step is necessary. The `TypeCheckPassVisitor` traverses the AST and type-checks every expression it encounters. In this example, the type-checking of the function call expression `_min(5,10)` shall be demonstrated.
First, the so called constraint generator traverses the part of the AST representing the expression. While doing so, each subexpression is assigned a type. Figure 2.11 shows this excerpt of the AST with the assigned types. So far, no concrete types have emerged. Instead, the constraint generator used type variables `$T0` to `$T4` as an intermediate placeholder. But `$T0` does not occur in the AST. This is because it is immediately replaced with the type `($T1, $T1) -> $T1`, since `$T1` is the type variable for the generic type parameter `T`. `$T4` does not occur either. This is because it is the type variable for variable `x`.
This is not everything the constraint generator has done. It also created a set of constraints on the type variables. The constraints either constrain a single type variable or describe a relation between two of them.

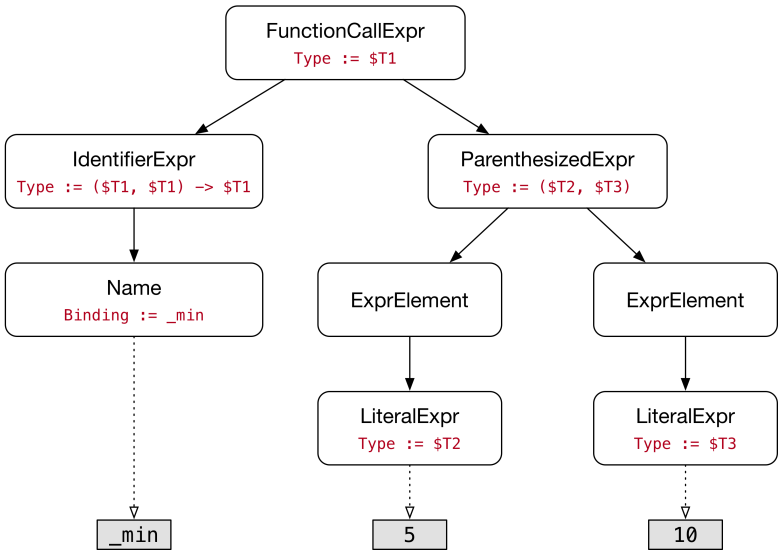

Figure 2.11.: *AST after constraint generation*

In the following, the constraints are listed:

- `$T1 ConformsTo Comparable`
- `$T2 LiteralConformsTo ExpressibleByIntegerLiteral`
- `$T3 LiteralConformsTo ExpressibleByIntegerLiteral`
- `$T2 ArgumentConversion $T1`
- `$T3 ArgumentConversion $T1`
- `$T1 Conversion $T4`

The constraint solver tries to solve this system of constraints by setting `$T2` and `$T3` to type `Int`, because this is the default type for variables constrained with `ExpressibleByIntegerLiteral`. Luckily, this also fulfils all the other constraints, by setting all of the other type variables to `Int` as well.
As mentioned above, the type variable `$T4` represents the type of variable `x`. Because it is now known that `$T4` is of type `Int`, the type of the variable binding for variable `x` is also `Int`.
Figure 2.12 shows the final scope tree. All the bindings have been assigned their type. This concludes the process of indexing the source code. In the editor, the index is used to jump to the definition of a name or to show the type of a name while hovering over it.

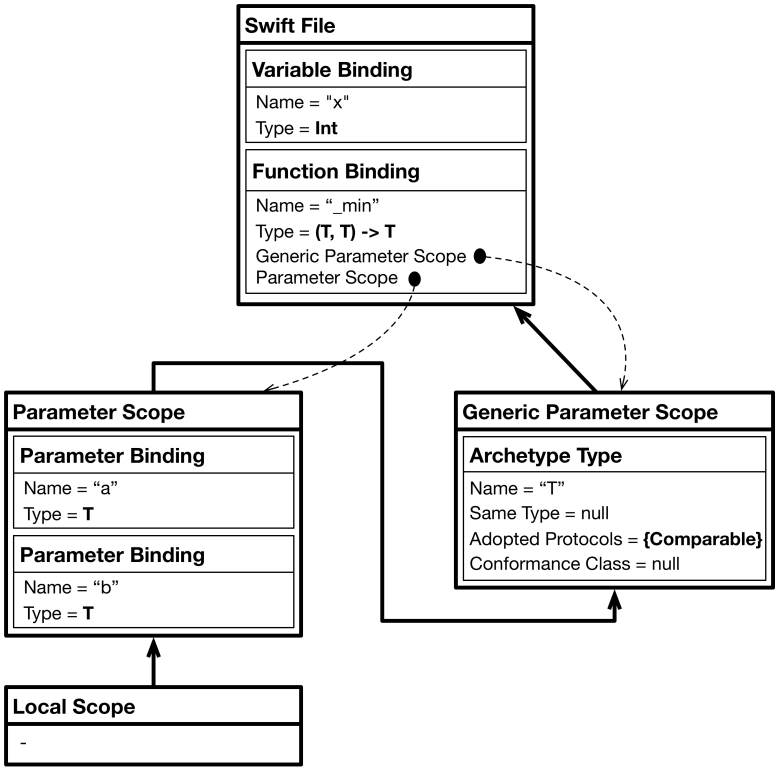

Figure 2.12.: *Scope tree updated after type check pass*

This concludes the short practical example. For a more detailed description of Tifig's infrastructure, the reader is once again advised to consult [30].

2.2. Representation of Generic Type Bindings

In the time between the hand-in of the original thesis and the start of this thesis, Tifig was further developed. The report of the original thesis was therefore already outdated. Important changes in the infrastructure regarding generic type variables and associated types were introduced, that have an important impact on the work conducted during this thesis. This section's intent is to close this gap and to provide the reader with the additional knowledge necessary to follow the documentation in the remainder of this document.

2.2.1. Archetypes

An archetype is a special kind of a type binding. It is used to represent bindings for generic type variables and associated types. Because Swift allows the developer

to constrain generic type variables and associated types, it is necessary for the indexer to collect these constraints. For that, the archetype binding is used. It has the properties `sameType`, `defaultType` and `protocolCompositionType`, the latter of which has the properties `adoptedProtocols` and `conformanceClass`. When a constraint on a generic type variable or associated type is encountered, it can be stored on the archetype representing it. This is best shown with an example.

```
1  protocol P {}
2
3  func f<T>(x: T) -> T where T: P {
4    return x
5  }
```

Listing 2.5: *Archetype example*

Listing 2.5 shows the declaration of the generic function `f()`. In the generic where clause of the function, the generic type parameter `T` is constrained to be comforming to protocol `P` (line 3). During the type definition pass, an archetype binding is created for the generic type parameter `T`. After the type annotation pass, the constraint `T: P` is stored on that archetype binding, in that the binding of protocol `P` is added to the list of adopted protocols. Other constraints are handled in a similar manner.
Archetype bindings can be virtual. This is the case, when an associated type is further constrained in a generic context, for example, in a generic function declaration. The constraint is then not stored on the archetype binding of the associated type itself, but a virtual copy is created that shadows the original archetype binding for this context. An example is given in Listing 2.6.

```
1  protocol P1 {
2    associatedtype A: P2
3  }
4
5  protocol P2 {
6    associatedtype B
7  }
8
9  func f<T: P1>(x: T) -> T where T.A.B == Int {
10   return x
11 }
```

Listing 2.6: *Virtual archetype example*

The generic type parameter `T` must conform to protocol `P1`. Therefore, it inherits the associated type `A`. The associated type `A` in turn inherits associated type `B` via the conformance to protocol `P2`. In the scope of the generic function, the type of

`T.A.B` is set to be equal to `Int`. It is intuitively clear, that this constraint should not influence code that is outside of the function's scope. So, a virtual copy of the archetype bindings representing associated types `A` and `B` are created. In addition, the `sameType` property of the virtual archetype binding of `B` is set to `Int`.

2.2.2. Associated Type Binding References

In Swift, the same nested associated type can be reached through different paths. Listing 2.7 shows an example, where both associated types `A` and `B` of protocol `P1` have an associated type themselves, namely `C` of protocol `P2`.

```
1  protocol P1 {
2    associatedtype A: P2
3    associatedtype B: P2
4  }
5
6  protocol P2 {
7    associatedtype C
8  }
9
10 func f<T: P1>(x: T) where T.A.C == Int {
11   let y: T.A.C
12   let z: T.B.C
13 }
```

Listing 2.7: *Associated type binding references example*

In the generic where clause of the function declaration for function `f()`, `T.A.C` is constrained to have type `Int`. Of course, the type of constant `y` is determined to be `Int` after resolution. However, the constant `z` is resolved to have type `T.B.C`. The conclusion to be drawn is that the constraint is only valid when the associated type `C` is addressed via `T.A.C`. Therefore, Tifig needed a construct that allowed to represent the paths to associated types. For that purpose, associated type binding references were introduced. They are bindings that hold all the names included in the path, ordered from left to right.

2.3. Related Work

Currently, there are only a handful of IDEs that allow the development of Swift applications. These alternatives to Tifig are Xcode [21], AppCode [1] and CLion [3]. Xcode is a proprietary IDE developed by Apple Inc., that allows to write applications in Objective-C, C++ and Swift. AppCode and CLion are both developed by JetBrains. While AppCode is intended for the development of iOS

and macOS applications, CLion is a cross-platform IDE for developing applications in C and C++. A plug-in allows the development of Swift code in CLion.
In the following, the above IDEs are compared by the criteria cost, operating system compatibilities and iOS and macOS development. An overview is given in Table 2.1.

Criteria	**Tifig**	**Xcode**	**AppCode**	**CLion**
Cost	free	free	paid	paid
Compatible to macOS	✓	✓	✓	✓
Compatible to Linux	✓			✓
Compatible to Windows	(✓)			(✓)
iOS/macOS development		✓	✓	

Table 2.1.: *Swift IDE comparison*

The goal of Tifig is to allow cross-platform development of Swift applications. Therefore, CLion with its Swift plug-in is the closest competitor. The compatibility to Windows is put in parentheses, because the IDEs can be run on Windows, but there is no Swift compiler implementation for Windows yet. The advantage of Tifig compared to CLion is that it is freely available, whereas CLion has to be paid via a subscription. In contrast to Xcode and AppCode, both Tifig and CLion are not capable of building iOS and macOS applications. This is probably the biggest disadvantage of Tifig.

3. Enhancing Tifig's Generics System

With the release of Swift version 4, Apple Inc. introduced several changes to Swift's generics system. These changes allow developers of Swift applications to be even more expressive than before. For developers of language tooling like the Tifig IDE, the need for integrating the changes into the existing infrastructure arises. But also, the developers and contributors working on the Swift compiler, the standard and core libraries and the Swift package manager need to update their infrastructure, in order to remain compatible with the newest language version.
The realization of the changes introduced by the accepted proposals mentioned in section 1.2 are documented in this chapter. For each of the proposals, a brief introduction is given, pointing out the syntactic changes in Swift's grammar. Then, with the new grammar in mind, the given lexer and parser infrastructure of Tifig is analyzed. The analysis then is continued on the indexer infrastructure, along with the included type checker mechanisms. Having completed the analysis, the implementation specifics are documented. How the implementation is tested is discussed in section 3.4. To round off the chapter, the flaws still present in the current implementation of Tifig are described in section 3.5.
Note, that this chapter requires previous knowledge of the Swift programming language and the Tifig plug-in architecture. The reader is advised to at least study the short overview section 2.1. Furthermore, it is strongly recommended to read through [30] before going ahead.

3.1. Generic Subscripts

The use of subscripts allows Swift developers to define an access mechanism for the members of their custom types in a manner that is commonly known for arrays or collections in other programming languages.

3.1.1. Introduction

Subscripts can be declared in the scope of classes, structures, enumerations, extensions and protocols. It is possible to define an arbitrary number of subscripts for

each type, as long as they differ in the types, argument labels and number of their arguments or their return type. Listing 3.1 shows how a subscript can be called. In this example, getting an element of an array by using an index in square brackets right after the name of the constant holding the array (line 3), is illustrated.

```
1  let arr = [1,2,3,4,5] // assigning an Array<Int> to the constant arr
2
3  let thirdElem = arr[2] // accessing element at index 2 with subscript
```

Listing 3.1: *Subscript call example*

To better understand how subscripts can be declared for custom types, a short example is provided in Listing 3.2. As can be seen, the use cases for the subscript notation are not limited to using indices in the form of integer values. Here, a house is described by the kind and the number of each kind of rooms it has. The subscript is then declared to take a `String` describing a room and returning the count of the given room kind. The declaration of the subscript ranges from lines 10 to 15 and an additional example of a subscript call is visible on line 21.

```
1  struct House {
2    let rooms = [
3      (name: "bedroom", count: 2),
4      (name: "bathroom", count: 2),
5      (name: "kitchen", count: 1),
6      (name: "livingroom", count: 1)
7    ]
8
9    // subscript declaration
10   subscript(roomName: String) -> Int {
11     guard let room = rooms.first(where: { $0.name == roomName }) else {
12       return 0
13     }
14     return room.count
15   }
16
17 }
18
19 let myHouse = House()
20
21 print("My house has \(myHouse["bedroom"]) bedrooms.")
22 // prints: My house has 2 bedrooms.
```

Listing 3.2: *Subscript declaration example*

The general syntax of a subscript definition is shown in Listing 3.3. The definition and formatting is taken from the official Swift programming language reference [16] to avoid ambiguity. The subscript declaration starts with the keyword `subscript`, followed by the subscript's *parameters* in parentheses. Finally, the signature of

the subscript is completed by providing the subscript's *return type*. The body of the subscript declaration contains a getter and setter introduced by the keywords `get` and `set`. These can be omitted, if the subscript is not intended to change the type it operates on. The body then simply contains *statements* (as was the case in Listing 3.2). However, this has no impact on the implementation coming up.

```
1 subscript ( parameters ) -> return type {
2   get {
3     statements
4   }
5   set( setter name ) {
6     statements
7   }
8 }
```

Listing 3.3: *Subscript declaration syntax*

To complete the introduction of subscripts, the grammar of subscript declarations is shown in Figure 3.1. Note, that this excerpt shows the grammar for Swift 3.1, before subscripts were extended to make use of generics. Again this overview is taken from [16].

subscript-declaration → *subscript-head* *subscript-result* *code-block*
subscript-declaration → *subscript-head* *subscript-result* *getter-setter-block*
subscript-declaration → *subscript-head* *subscript-result* *getter-setter-keyword-block*
subscript-head → $attributes_{opt}$ $declaration\text{-}modifiers_{opt}$ `subscript` *parameter-clause*
subscript-result → `->` $attributes_{opt}$ *type*

Figure 3.1.: *Subscript declaration grammar Swift 3.1*

As was mentioned above, subscripts can also be declared within protocol declarations. In that case, the declaration is referred to as a subscript requirement (only the subscript signature can be declared). The syntax and grammar for these declarations differ from what was shown before. Listing 3.4 presents the special syntax for protocol subscript requirements.

```
1 subscript ( parameters ) -> return type { get set }
```

Listing 3.4: *Protocol subscript requirement syntax*

In Figure 3.2, the grammar for protocol subscript requirements is given. Note, that the grammar directly references the two last grammar entries shown in Figure 3.1.

| *protocol-subscript-declaration* → *subscript-head* *subscript-result* *getter-setter-keyword-block*

Figure 3.2.: *Protocol subscript requirement grammar Swift 3.1*

The following section points out the changes introduced to subscripts with the release of Swift 4.

3.1.2. Making Subscripts Generic

In his proposal [22], Eidhof suggested to extend subscripts with generics. The motivation behind this change was to reduce the number of specific subscripts with essentially the same behaviour and to translate some generic methods into subscripts to make their usage feel more natural. The extension does not only affect the developers writing Swift applications, but also benefits the implementation of the standard library.

```
1  extension Collection {
2
3    // subscript declaration
4    subscript<Indices: Sequence>(indices indices: Indices) -> [Iterator.Element]
5        where Indices.Iterator.Element == Index {
6      var result: [Iterator.Element] = []
7      for index in indices {
8        result.append(self[index])
9      }
10     return result
11   }
12
13 }
14
15 let arr = [1,2,3,4,5,6]
16 print(arr[indices: [0,2,4]]) // prints: [1, 3, 5]
17 print(arr[indices: 1...4]) // prints: [2, 3, 4, 5]
```

Listing 3.5: *Generic subscript example*

In Listing 3.5, a simple example of a generic subscript is shown. It is declared in an extension of the `Collection` protocol (lines 4 to 11), which is part of the standard library. Right after the `subscript` keyword, the generic parameter clause `<Indices: Sequence>` is provided. The generic type parameter `Indices`, which has to conform to `Sequence`, is then used as the type of the subscript's parameter. Another new syntactic element is the generic where clause, that optionally follows the return type (line 5). After the keyword `where`, an arbitrary number of constraints for the generic parameter and its associated types can be formulated. In this example, the constraint makes sure, that the elements of the sequence parameter match the type of the collections index type. Both the generic parameter clause and the generic

where clause are syntactically equivalent to the corresponding syntactic elements in function declarations.
The updated Swift 4 grammar for subscript declarations shown in Figure 3.3 contains no surprises. At the expected positions, the generic parameter clause and the generic where clause were inserted and marked as optional. The bold highlighting has no special meaning other than to make clear where the changes occurred.

subscript-declaration → *subscript-head* *subscript-result* ***generic-where-clause$_{opt}$*** *code-block*
subscript-declaration → *subscript-head* *subscript-result* ***generic-where-clause$_{opt}$*** *getter-setter-block*
subscript-declaration → *subscript-head* *subscript-result* ***generic-where-clause$_{opt}$*** *getter-setter-keyword-block*
subscript-head → *attributes$_{opt}$* *declaration-modifiers$_{opt}$* `subscript` ***generic-parameter-clause$_{opt}$*** *parameter-clause*
subscript-result → `->` *attributes$_{opt}$* *type*

Figure 3.3.: *Subscript declaration grammar Swift 4*

At this point, an interesting observation can be made. The syntax for subscript declarations looks remarkably similar to the syntax of function declarations. A quick look at the function declaration grammar in Listing 3.4 underlines the observation. The differences are the following:

- Subscript declarations have no name. The sequence of symbols `func` (*identifier* | *operator*) of the function declaration can therefore be mapped to the terminal symbol `subscript` of the subscript declaration.
- Subscripts are not (yet) capable of throwing or rethrowing. Therefore, the two production rules for *function-signature* can be combined into one, leaving out the terminal symbols `throw` and `rethrow`. To further close the gap between functions and subscripts, a proposal has already been formulated and is currently available as a draft [29].
- The non-terminal symbol *subscript-result* is not optional. The corresponding non-terminal symbol *function-result* is, however.

function-declaration → *function-head* *function-name* *generic-parameter-clause$_{opt}$* *function-signature* *generic-where-clause$_{opt}$* *function-body$_{opt}$*
function-head → *attributes$_{opt}$* *declaration-modifiers$_{opt}$* `func`
function-name → *identifier* | *operator*
function-signature → *parameter-clause* `throws`$_{opt}$ *function-result$_{opt}$*
function-signature → *parameter-clause* `rethrows`$_{opt}$ *function-result$_{opt}$*
function-result → `->` *attributes$_{opt}$* *type*

Figure 3.4.: *Function declaration grammar Swift 4*

The similarities between subscripts and functions have an important impact on how generics can be introduced to the Tifig infrastructure. This is further elaborated in section 3.1.3.
For the sake of completeness, the grammar for protocol subscript requirements is illustrated in Figure 3.5.

protocol-subscript-declaration → *subscript-head* *subscript-result* ***generic-where-clause***$_{opt}$ *getter-setter-keyword-block*

Figure 3.5.: *Protocol subscript requirements grammar Swift 4*

In this section, a brief (and incomplete) introduction to subscripts and the corresponding changes with the release of Swift 4 has been provided. For a more detailed documentation, the reader is advised to consult [16].

3.1.3. Implementation

When changing the infrastructure of Tifig, it is advisable to follow a certain process. First, the lexer generating the tokens of the Swift source code is updated, if necessary. Then, the parser is extended to be able to consume the newly created or reused tokens. This results in necessary changes to the AST. Finally, the indexer implementation is revisited. This also includes the integrated type checker.

Changes to the Lexer

In section 3.1.2, the similarities between generic subscript declarations and function declarations were discussed. For the lexer, the conclusion to be drawn was that the introduction of new tokens was not necessary. The additions to subscripts were built using the same tokens as have been used in function declarations all along.

Changes to the Parser and the AST

In order to build a correct AST, the recursive descent parser had to be adjusted to the new language syntax. However, the changes were straightforward. The `parser` package, which is part of the `ch.hsr.ifs.tifig.core` plugin, contains the `DeclParser` class. This class is responsible for the correct parsing of the various declarations of Swift. To support generic parameter clauses and generic where clauses, the already existing methods `subscriptDecl()` and `protocolSubscriptDecl()` needed to be extended. Fortunately, Tifig's infrastructure already provided classes `GenericParameterClause` and `GenericWhereClause`, as well as the method `parseIf` of class `ParserModule`, which parses the before

mentioned constructs, if present. Listing 3.6 shows the complete implementation of the changed `subscriptDecl()` method.

```
1  private SubscriptDecl subscriptDecl() throws RecognitionException {
2    final Name subscriptName = parseKeywordName(Kind.KW_SUBSCRIPT);
3
4    final GenericParameterClause genericParameterClause =
5        parseIf(Kind.LANGLEBRACKET, this::genericParameterClause);
6    final ParameterClause parameterClause = parse(this::parameterClause);
7    match(Kind.ARROW);
8    final IASTType returnType = parse(typeParser::type);
9    final GenericWhereClause genericWhereClause =
10       parseIf(Kind.KW_WHERE, this::genericWhereClause);
11
12   CodeBlock getterBlock = null;
13   VarDeclClauseList clauseList = null;
14   if(speculate(this::varDeclClauseList)) {
15     clauseList = parse(this::varDeclClauseList);
16     if(!isValidVarDeclClauseList(clauseList)) {
17       throw new RecognitionException("unknown variable declaration clause");
18     }
19   } else {
20     getterBlock = parse(stmtParser::codeBlock);
21   }
22   return new SubscriptDecl(subscriptName, genericParameterClause,
23       parameterClause, returnType, genericWhereClause, getterBlock,
24       clauseList);
25 }
```

Listing 3.6: *Parsing subscript declarations*

Lines 4 to 5 are responsible for the parsing of the generic parameter clause, whereas lines 9 to 10 parse the generic where clause, if present. As the result of the method call, a new instance of `SubscriptDecl` is returned. The constructor on lines 22 to 24 takes two new arguments, namely `genericParameterClause` and `genericWhereClause`. Note that this required changes to the `SubscriptDecl` class, which will be explained later in this section.
Listing 3.7 shows the corresponding implementation for parsing generic subscript requirements in protocols.

```
1  private ProtocolSubscriptDecl protocolSubscriptDecl()
2        throws RecognitionException {
3    final Name subscriptName = parseKeywordName(Kind.KW_SUBSCRIPT);
4    final GenericParameterClause genericParameterClause =
5        parseIf(Kind.LANGLEBRACKET, this::genericParameterClause);
6    final ParameterClause parameterClause = parse(this::parameterClause);
7    match(Kind.ARROW);
8    final IASTType returnType = parse(typeParser::type);
9    final GenericWhereClause genericWhereClause =
10       parseIf(Kind.KW_WHERE, this::genericWhereClause);
11   final VarDeclClauseList clauseList = parse(this::varDeclClauseList);
12   if(!isValidVarDeclClauseList(clauseList, ClauseType.get, ClauseType.set)) {
13     throw new RecognitionException("unknown var decl clause");
```

```
14    }
15    return new ProtocolSubscriptDecl(subscriptName, genericParameterClause,
16        parameterClause, returnType, genericWhereClause, clauseList);
17  }
```

Listing 3.7: *Parsing protocol subscript requirements*

Parsing the generic parameter clause is done on lines 4 to 5. Parsing the generic where clause takes place on lines 9 to 10. Again, the constructor of the return type, here, an instance of `ProtocolSubscriptDecl`, was extended to take the additional arguments `genericParameterClause` and `genericWhereClause`. This completes the necessary changes introduced to the parser infrastructure.
The modifications explained so far were not compatible with the AST implementation of Tifig.
The classes `SubscriptDecl` and `ProtocolSubscriptDecl` both inherit from class `Decl`, which in turn inherits from `ASTNode`. The modifications to class `SubscriptDecl` are shown in Listing 3.8.

```
1   public class SubscriptDecl extends Decl implements IFunctionLikeDecl,
2       IGeneralizable {
3     private final Name subscriptName;
4     private final GenericParameterClause genericParameterClause;
5     //...
6     private final GenericWhereClause genericWhereClause;
7     //...
8
9     public SubscriptDecl(Name subscriptName,
10        GenericParameterClause genericParameterClause,
11        ParameterClause parameterClause, IASTType returnType,
12        GenericWhereClause genericWhereClause, CodeBlock getterBlock,
13        VarDeclClauseList clauseList) {
14      this.subscriptName = subscriptName;
15      this.genericParameterClause = genericParameterClause;
16      this.parameterClause = parameterClause;
17      this.returnType = returnType;
18      this.genericWhereClause = genericWhereClause;
19      this.getterBlock = getterBlock;
20      this.clauseList = clauseList;
21    }
22
23    @Override
24    public GenericParameterClause getGenericParameterClause() {
25      return genericParameterClause;
26    }
27
28    @Override
29    public GenericWhereClause getGenericWhereClause() {
30      return genericWhereClause;
31    }
32
33    //...
34
35    @Override
```

```
36    public boolean accept(ASTVisitor visitor) {
37      return acceptVisitor(visitor, getAttributes(), subscriptName,
38          genericParameterClause, parameterClause, returnType,
39          genericWhereClause, getterBlock, clauseList);
40    }
41
42  }
```

Listing 3.8: *Subscript declaration node*

Essentially, the properties `genericParameterClause` and `genericWhereClause` on lines 4 and 6 complete the AST node for subscript declarations. The constructor was extended to take the new arguments `genericParameterClause` and `genericWhereClause` and upon instantiation, to assign the arguments to the corresponding properties. Lines 23 to 31 show the implementation of getter methods for the new properties. These methods were overridden in order to make `SubscriptDecl` implement the `IGeneralizable` interface. Finally, every AST node has to override an `accept()` method, taking an argument of type `ASTVistor`, to make the node traversable. Here, the child nodes `genericParameterClause` and `genericWhereClause` were added on line 38 and 39, respectively. Listing 3.9 shows the corresponding implementation for the protocol subscript requirement node.

```
1   public class ProtocolSubscriptDecl extends Decl implements
2       IProtocolMemberDecl, IFunctionLikeDecl, IGeneralizable {
3     private final Name subscriptName;
4     private final GenericParameterClause genericParameterClause;
5     //...
6     private final GenericWhereClause genericWhereClause;
7     //...
8
9     public ProtocolSubscriptDecl(Name subscriptName,
10        GenericParameterClause genericParameterClause,
11        ParameterClause parameterClause,
12        IASTType returnType, GenericWhereClause genericWhereClause,
13        VarDeclClauseList clauseList) {
14      this.subscriptName = subscriptName;
15      this.genericParameterClause = genericParameterClause;
16      this.parameterClause = parameterClause;
17      this.returnType = returnType;
18      this.genericWhereClause = genericWhereClause;
19      this.clauseList = clauseList;
20    }
21
22    @Override
23    public GenericParameterClause getGenericParameterClause() {
24      return genericParameterClause;
25    }
26
27    @Override
28    public GenericWhereClause getGenericWhereClause() {
29      return genericWhereClause;
```

```
30    }
31
32    //...
33
34    @Override
35    public boolean accept(ASTVisitor visitor) {
36      return acceptVisitor(visitor, getAttributes(), subscriptName,
37          genericParameterClause, parameterClause, returnType,
38          genericWhereClause, clauseList);
39    }
40
41    //...
42  }
```

Listing 3.9: *Protocol subscript requirement node*

Having introduced the above changes, Tifig's parser is now capable of building and traversing the AST for generic subscripts. To round off this section, a brief example shall be given. Figure 3.6 shows the AST generated for the example in Listing 3.5.

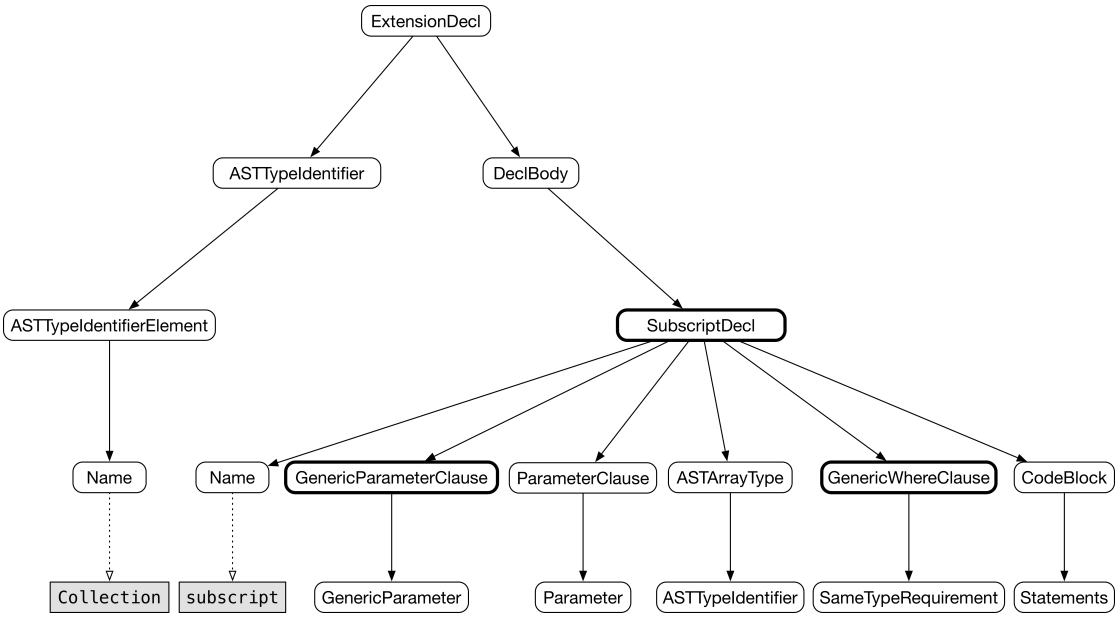

Figure 3.6.: *AST for generic subscript declaration*

The `SubscriptDecl` is a child node to the `DeclBody` of the extension declaration. Highlighted with thick borders are, next to the `SubscriptDecl` node, the newly available child nodes `GenericParameterClause` and `GenericWhereClause`.

Changes to the Indexer and the Type Checker

As with the AST nodes in the previous section, the bindings for subscripts already existed in the `ch.hsr.ifs.tifig.core.indexer.bindings` package. The goal was therefore to extend the infrastructure by the necessary functionality. First of all, the `ISubscriptBinding` interface needed to be enriched. This is shown in Listing 3.10.

```
1  public interface ISubscriptBinding extends IMemberBinding, IStorageBinding {
2    void setParameterScope(ParameterScope parameterScope);
3
4    void setGenericParameterScope(GenericParameterScope genericParameterScope);
5
6    GenericParameterScope getGenericParameterScope();
7
8    List<IParameterBinding> getParameters();
9
10   boolean isGeneric();
11
12   List<IArchetypeType> getGenericTypeParameters();
13 }
```

Listing 3.10: *Subscript binding interface*

In addition to the scope for subscript parameters, a generic parameter scope was needed. The getter and setter signatures were defined on lines 4 and 6. A helper method to get the generic type parameters of the subscript was declared on line 12. Finally, it is helpful to be able to determine, whether a subscript is in fact generic or not. Therefore, all implementations of `ISubscriptBinding` need to implement the `isGeneric()` method (line 10).
Because `SubscriptBinding` inherits from `ISubscriptBinding`, the above mentioned methods had to be implemented. The implementation was straightforward. However, a few interesting changes occurred that are worth discussing. The source code containing these changes is shown in Listing 3.11.

```
1  public class SubscriptBinding extends AbstractBaseBinding implements
2      ISubscriptBinding, IDeclaredBinding {
3    //...
4
5    @Override
6    public void _validate() {
7      final SubscriptDecl subscriptDecl =
8          (SubscriptDecl)getDeclarationName().getParent();
9      TypeUtils.resolveOwner(this);
10     TypeUtils.resolveGenericParameters(this,
11         subscriptDecl.getGenericParameterClause(),
12         subscriptDecl.getGenericWhereClause(), true);
13     final FunctionType type = TypeUtils.getFunctionType(getParameters(),
14         ThrowingBehaviour.None, subscriptDecl.getReturnType());
15     setType(type);
```

```
16      TypeUtils.inferAdditionalTypeParameterRequirements(type,
17          getGenericTypeParameters());
18      if(subscriptDecl.getGetterBlock() != null) {
19        resolveGetterBlock(subscriptDecl.getGetterBlock());
20      } else {
21        for(final VarDeclClause clause : subscriptDecl.getClauses()) {
22          resolveVarDeclClause(clause);
23        }
24      }
25    }
26
27    //...
28  }
```

Listing 3.11: *Subscript binding*

Every binding extends the `AbstractBaseBinding` class. This in turn means that every binding must provide an implementation of the `_validate()` method. Essentially, this method attempts to resolve all possibly unknown members of the binding it is called on. Note, that this happens as a side effect and no new binding is returned. To support the resolution of the newly introduced generic type parameters, the static call to `resolveGenericParameterClause()` was added on lines 11 to 13. This method was already defined in the utils class `TypeUtils`, because it is not only needed for subscripts, but also for functions. A call to `inferAdditionalTypeParameterRequirements()` was added on line 17, to infer additional requirements for the individual type parameters from their use in the rest of the subscript signature. Luckily, the implementations of these methods were already existing in `TypeUtils` and usable without additional changes.

```
1   public class ProtocolSubscriptBinding extends SubscriptBinding {
2     //...
3
4     @Override
5     public void _validate() {
6       final ProtocolSubscriptDecl protocolSubscriptDecl =
7           (ProtocolSubscriptDecl)getDeclarationName().getParent();
8       TypeUtils.resolveOwner(this);
9       TypeUtils.resolveGenericParameters(this,
10          protocolSubscriptDecl.getGenericParameterClause(),
11          protocolSubscriptDecl.getGenericWhereClause(), true);
12      final FunctionType type = TypeUtils.getFunctionType(getParameters(),
13          ThrowingBehaviour.None, protocolSubscriptDecl.getReturnType());
14      setType(type);
15      TypeUtils.inferAdditionalTypeParameterRequirements(type,
16          getGenericTypeParameters());
17    }
18  }
```

Listing 3.12: *Protocol subscript binding*

For class `ProtocolSubscriptBinding`, the adjustment was much simpler, because it does not directly inherit from `ISubscriptBinding`, but extends `SubscriptBinding`. The final implementation of its `_validate()` method is shown in Listing 3.12.
So far, it is possible to parse generic subscripts. Also, the corresponding bindings have been adjusted. What was left to do was to update the resolution logic in the type checker.

```
1  public class DefinitionPassVisitor extends ASTVisitor {
2    //...
3
4    private void createProtocolSubscriptBinding(
5        ProtocolSubscriptDecl protocolSubscriptDecl) {
6      final Name subscriptName = protocolSubscriptDecl.getDeclaredName();
7      final boolean isSettable = protocolSubscriptDecl.hasSetter();
8      final ProtocolSubscriptBinding protocolSubscriptBinding =
9          new ProtocolSubscriptBinding(subscriptName, isSettable);
10     currentScope.define(protocolSubscriptBinding);
11     if(protocolSubscriptDecl.getGenericParameterClause() != null) {
12       genericParameterScope = new GenericParameterScope();
13       protocolSubscriptBinding.setGenericParameterScope(
14           genericParameterScope);
15     }
16     parameterScope = new ParameterScope(null);
17     protocolSubscriptBinding.setParameterScope(parameterScope);
18   }
19
20   //...
21
22   private void createSubscriptBinding(SubscriptDecl subscriptDecl) {
23     final Name subscriptName = subscriptDecl.getDeclaredName();
24     final AccessLevel accessLevel =
25         getAccessLevelFromDeclAndCurrentScope(subscriptDecl);
26     final boolean isOverride =
27         subscriptDecl.hasDeclModifier(DeclModifier.Override);
28     final boolean isSettable = subscriptDecl.hasSetter();
29     final ISubscriptBinding subscriptBinding = new SubscriptBinding(
30         accessLevel, subscriptName, isOverride, isSettable);
31     currentScope.define(subscriptBinding);
32     if(subscriptDecl.getGenericParameterClause() != null) {
33       genericParameterScope = new GenericParameterScope();
34       subscriptBinding.setGenericParameterScope(genericParameterScope);
35     }
36     parameterScope = new ParameterScope(null);
37     subscriptBinding.setParameterScope(parameterScope);
38   }
39
40   //...
41 }
```

Listing 3.13: *Definition Pass Visitor*

Listing 3.13 shows an excerpt of the `DefinitionPassVisitor` class. Its purpose is to visit AST nodes and, while doing so, to create bindings for all names occurring in the AST. This also happens for subscript declarations and protocol

subscript requirements. The two methods in the listing create new bindings for `SubscriptDecl` and `ProtocolSubscriptDecl` nodes. New are the lines 11 to 15 and 32 to 35, respectively. Basically, it is now necessary to check for a generic parameter clause and if present, a new generic parameter scope has to be created and assigned to the binding.
Finally, a minimal change to the constraint system was necessary. The change occured in the `getGenericTypeParametersForBinding` method, as can be seen in Listing 3.14.

```
1  public class ConstraintSystem {
2    //...
3
4    private List<IArchetypeType> getGenericTypeParametersForBinding(
5        IBinding binding) {
6      if(binding instanceof IFunctionBinding) {
7        final IFunctionBinding functionBinding = (IFunctionBinding)binding;
8        if(functionBinding.isGeneric()) {
9          return functionBinding.getGenericTypeParameters();
10       }
11     } else if(binding instanceof ISubscriptBinding) {
12       final ISubscriptBinding subscriptBinding = (ISubscriptBinding)binding;
13       if(subscriptBinding.isGeneric()) {
14         return subscriptBinding.getGenericTypeParameters();
15       }
16     }
17     return new ArrayList<>();
18   }
19
20   //...
21 }
```

Listing 3.14: *Constraint System*

This helper method is used by the constraint system to get the type of member references. So far, the method had to deal with function bindings only. The handling of the generic type parameters of subscripts was added on lines 11 to 16.

3.1.4. Summary

In this section, Tifig's infrastructure was extended to support generic subscripts. It was shown in detail how such an extension can be achieved following the path of the compiler.

3.2. Class & Subtype Existentials

Existential types appear in many popular programming languages. In Scala, existential types can be defined using the `forSome` keyword. This is demonstrated

in Listing 3.15. Without going into much detail, it can be observed that the existential type `Existential` is defined (line 8) and later used as argument type in the `printAll` method (line 10).

```
1  class MyClass[T](val value: T) {
2    def print(): Unit = {
3      println(value)
4    }
5  }
6
7  object Main {
8    type Existential = MyClass[A] forSome { type A }
9
10   def printAll(list: List[Existential]): Unit = {
11     list.foreach(x => x.print)
12   }
13
14   def main(args: Array[String]): Unit = {
15     val x = new MyClass("Scala")
16     val y = new MyClass(1: Int)
17     val z = new MyClass(0.1: Double)
18     printAll(List(x,y,z))
19   }
20 }
21 // prints:
22 // Scala
23 // 1
24 // 0.1
```

Listing 3.15: *Existential types in Scala*

In Java, bounded existential types can be expressed as a parameterized type and instantiated by a wildcard `?` together with an upper or lower bound. For example, `List<? extends Animal>` or `List<? super Animal>` [4].
In this section, the introduction of class and subtype existentials to Swift is discussed.

3.2.1. Introduction

As of Swift 3.1, the only existentials representable were conformances to a set of protocols. This also includes the set containing a single protocol or zero protocols. The latter is represented by the `Any` type [14]. To express conformances to two or more protocols, Swift comes with a special syntax for so called protocol composition types. Protocol composition types can be used when specifying a type in type annotations, in generic parameter clauses, and in generic where clauses [16]. Listing 3.16 shows an example of how protocol composition types are used in a generic parameter clause.

```
1  protocol Runner {
2    func run()
3  }
4
5  protocol Bicyclist {
6    func ride()
7  }
8
9  protocol Swimmer {
10   func swim()
11 }
12
13 class Triathlete: Runner, Bicyclist, Swimmer {
14   func run() { print("I'm running!") }
15   func ride() { print("I'm riding my bicycle!") }
16   func swim() { print("I'm swimming!") }
17 }
18
19 func participate<T: Runner & Bicyclist & Swimmer>(_ participant: T) {
20   participant.run()
21   participant.ride()
22   participant.swim()
23 }
24
25 let participant = Triathlete()
26 participate(participant)
27 // prints:
28 // I'm running!
29 // I'm riding my bicycle!
30 // I'm swimming!
```

Listing 3.16: *Protocol composition type example*

In the example, three protocols are declared, each of which contains a method requirement (lines 1 to 11). To conform to protocol `Runner`, a type must implement the `run()` method. The same goes for `Bicyclist` and `Swimmer`, where a conforming type must implement the method `ride()` or `swim()`, respectively. After that, the class `Triathlete` is declared (lines 13 to 17). The class must conform to all of the above protocols, which is stated in the class's type inheritance clause (line 13). Going on, a generic function `participate()` is declared (lines 19 to 23). The generic parameter `T` is constrained by the protocol composition type `Runner & Bicyclist & Swimmer` (line 19). Because it is known that `T` conforms to all the three protocols, all three methods declared in the protocols can be called on `participant`, the parameter of the function with type `T`. Finally, `participate()` is called with an instance of class `Triathlete` (line 26).
The example already reveals the syntax of protocol composition types. For the sake of completeness, the syntax is shown again in Listing 3.17. As was the case in section 3.1.1, the definition and formatting is taken from [16].

```
1  Protocol 1 & Protocol 2
```

Listing 3.17: *Protocol composition syntax*

To make the developers' lives a bit easier, it is allowed to chain multiple protocols in one protocol composition type. This can be better observed in Figure 3.7, which shows the grammar for protocol composition types.

```
protocol-composition-type         → type-identifier & protocol-composition-continuation
protocol-composition-continuation → type-identifier | protocol-composition-type
```

Figure 3.7.: *Protocol composition type grammar*

The non-terminal symbol *protocol-composition-type* is defined recursively via *protocol-composition-continuation*. This way, it is ensured that the composition is composed of at least two *type-identifer* symbols. Note, that the grammar does not specify the kind of type identifiers that can be part of the protocol composition type. This is defined in text form in [16].
The following section points out the changes introduced to protocol composition types with the release of Swift 4.

3.2.2. Adding Class & Subtype Existentials

Hart and Zheng wrote a proposal to enrich Swift with class and subtype existentials that conform to protocols, to bring more expressive power to Swift's type system [25]. The main motivation behind the proposal was to bring Swift closer to the expressiveness of Objective-C. This is not a matter of taste, but because Swift must be capable of bridging types from Objective-C.
Listing 3.18 shows an example of how the new feature can be used. For that, the example of Listing 3.16 was extended with class `TriathlonWinner`, which inherits from `Triathlete` (lines 13 to 15). The class also conforms to the newly introduced protocol `Winner` (lines 3 to 5). It therefore has to implement the `win()` method. A generic function `win()` was also added, taking an argument of type `T` (lines 23 to 26). This time, the generic type parameter `T` is not only constrained by protocol `Winner`, but must also be a subclass of `Triathlete`. Note, that the syntax of the protocol composition type remains the same, but contains a class type (line 23). This was not possible in Swift 3.1.

```
1  // ...
2
3  protocol Winner {
4    func win()
```

```
5  }
6
7  class Triathlete: Runner, Bicyclist, Swimmer {
8    func run() { print("I'm running!") }
9    func ride() { print("I'm riding my bicycle!") }
10   func swim() { print("I'm swimming!") }
11 }
12
13 class TriathlonWinner: Triathlete, Winner {
14   func win() { print("I won!") }
15 }
16
17 func participate<T: Runner & Bicyclist & Swimmer>(_ participant: T) {
18   participant.run()
19   participant.ride()
20   participant.swim()
21 }
22
23 func win<T: Triathlete & Winner>(_ winner: T) {
24   participate(winner)
25   winner.win()
26 }
27
28 let winner = TriathlonWinner()
29 win(winner)
30 // prints:
31 // I'm running!
32 // I'm riding my bicycle!
33 // I'm swimming!
34 // I won!
```

Listing 3.18: *Subtype existential example*

The syntax and grammar for protocol composition types remains the same, although a wider variety of non-terminal symbols *type-identifier* is allowed. Because this is a semantic constraint, it is written down in text form, accompanying the grammar definition.
Each item in a protocol composition list is either the name of a class, the name of a protocol or a type alias whose underlying type is a protocol composition type, a protocol, or a class. When a protocol composition type contains type aliases, it's possible for the same protocol to appear more than once. Duplicates are then ignored. Further, the list of items can contain at most one class [16].
Before heading into implementation details, it is important to point out discrepancies between the proposal text [25], the official specification [16] and the actual implementation of the Swift compiler:

- The proposal stipulated, that multiple class types can occur in a protocol composition type, as long as they come from the same type hierarchy. The idea was, that in this case, the most specialized type would remain part of the protocol composition type, while the others are discarded as duplicates.

Although the proposal is marked as implemented in Swift 4, testing the code in Listing 3.19 results in a compile error.

```
1  protocol P {}
2  class C {}
3  class D : C { }
4  class F : D, P { }
5  let t: C & D & P = F() // Should compile according to proposal.
6  // Compile error line 5: protocol-constrained type cannot contain class
7  // 'D' because it already contains class 'C'
```

Listing 3.19: *Feature not implemented in Swift 4*

The code provided in the listing is taken directly from the proposal text. After a brief exchange over the Swift users mailing list [10], it became clear that only a partial implementation of the proposal found its way into Swift 4. The discrepancy was then reported on the Swift bug tracker by the author [11].

- The specification described above states, that duplicates of protocols are allowed in protocol composition types. The Swift compiler allows the same for class types. Although the number of class types in the composition is specified to be one, it is possible to list the same class type multiple times. In addition, the duplicates can be hidden behind type aliases. The code shown in Listing 3.20 therefore compiles.

```
1  protocol P {}
2  protocol Q {}
3  protocol R {}
4  class C {}
5
6  typealias PQC = P & Q & C
7  typealias PQRC = C & PQC & Q & R & C
8
9  let x: PQC & PQRC & R
10 // Type of x: C & P & Q & R
```

Listing 3.20: *Duplicates of class types*

To prevent confusion of Tifig users, it was decided to implement class and subtype existentials according to the implementation of the Swift compiler.

3.2.3. Implementation

In this section, the implementation details for class and subtype existentials are documented. The arrangement of Section 3.1.3 is maintained, meaning that the introduced changes are listed incrementally from the lexer via the parser to the indexer.

Changes to the Lexer

The introduction of class and subtype existentials did not require new keywords or other lexical complements. Therefore, changes to the lexer were not necessary.

Changes to the Parser and the AST

In Section 3.2.2, it was already mentioned that the grammar for protocol composition types did not undergo any changes between the Swift versions 3.1 and 4. Although it is now allowed to have class types within protocol composition types, the AST remains the same. Listing 3.21 shows the declaration of a constant `x`, annotated with a protocol composition type (line 5). It can be seen that class `C` is part of the protocol composition type.

```
1 protocol P {}
2 protocol Q {}
3 class C {}
4
5 let x: C & P & Q
```

Listing 3.21: *Subtype existential in protocol composition*

An excerpt of the AST generated from this code example is shown in Figure 3.8. The `ASTTypeAnnotation` node is part of a `TypedPattern` with the name `x`. Its child node is an `ASTProtocolCompositionType` node, which represents `C & P & Q`. As can be seen, the three children of the `ASTProtocolCompositionType` are all `ASTTypeIdentifier` nodes. Each is parent to one `ASTTypeIdentiferElement`, which in turn are parents to the `Name` nodes `C`, `P` and `Q`.
Obviously, the AST does not differentiate between classes and protocols. On this level of abstraction, where only the syntactical correctness matters, all type identifiers are considered the same.

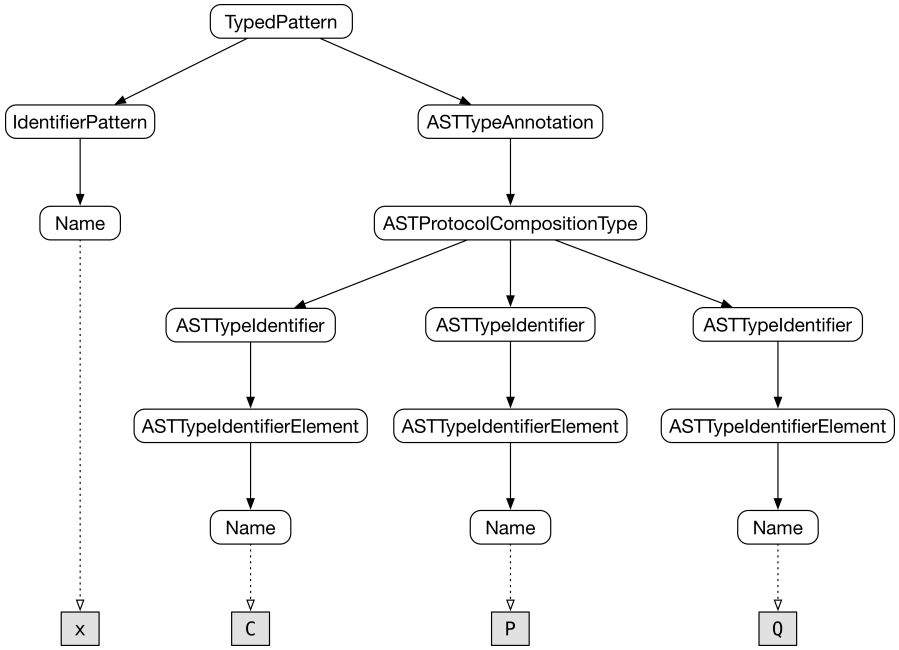

Figure 3.8.: *AST for subtype existential in protocol composition*

Not needing to change the AST means not needing to change the parser, because the parser's purpose is to build the AST. Tifig's parser infrastructure therefore remained untouched.

Changes to the Indexer and the Type Checker

The previous section showed, that the AST node for protocol composition types did not change. This is not true for `ProtocolCompositionType`, the corresponding index type. Listing 3.22 shows the changed bits of class `ProtocolCompositionType`.

```
1  public class ProtocolCompositionType implements IMemberOwner,
2      IProtocolAdopter {
3    private Set<ProtocolTypeBinding> adoptedProtocols = new LinkedHashSet<>();
4    private ITypeBinding conformanceClass = null;
5
6    // ...
7
8    public void removeAnyObjectConformance() {
9      adoptedProtocols = new LinkedHashSet<>(adoptedProtocols.stream()
10         .filter(p -> !(p.getDeclarationName().getIdentifier()
11         .equals("AnyObject")))
12         .collect(Collectors.toList()));
13   }
14
15   // ...
16
17   public void setConformanceClass(ITypeBinding conformance) {
```

```
18      assert(TypeUtils.isClassType(conformance));
19      conformanceClass = conformance;
20    }
21
22    public ITypeBinding getConformanceClass() {
23      return conformanceClass;
24    }
25
26    // ...
27
28    private OverloadSet getOverloadSetFromProtocolExtensions(Name name) {
29      final OverloadSet set = new OverloadSet();
30      if(conformanceClass instanceof ClassTypeBinding) {
31        final ClassTypeBinding classType =
32            (ClassTypeBinding)conformanceClass;
33        set.merge(classType.getMemberScope().getLocalOverloadSet(name));
34      } else if(conformanceClass instanceof GenericTypeInstance) {
35        final GenericTypeInstance genericTypeInstance =
36            (GenericTypeInstance)conformanceClass;
37        set.merge(((ClassTypeBinding)genericTypeInstance
38            .getGenericType())
39            .getMemberScope()
40            .getLocalOverloadSet(name));
41      }
42
43      // ...
44      return set;
45    }
46
47    // ...
48
49    @Override
50    public IType transform(Function<IType, IType> fn) {
51      final IType transformed = fn.apply(this);
52      if(transformed == null || transformed != this) {
53        return transformed;
54      }
55
56      final ProtocolCompositionType protocolCompositionType =
57          new ProtocolCompositionType();
58      // ...
59      if(conformanceClass != null) {
60        protocolCompositionType.setConformanceClass(
61            (ITypeBinding)conformanceClass.transform(fn));
62      }
63      return protocolCompositionType;
64    }
65
66    // ...
67
68    @Override
69    public ITypeBinding getExistentialTypeConformanceClass() {
70      return conformanceClass;
71    }
72
73    // ...
74  }
```

Listing 3.22: *Protocol composition type*

Next to the set `adoptedProtocols`, a new property `conformanceClass` of type `ITypeBinding` was added (line 4). The corresponding getter and setter methods were defined on lines 17 to 24. The remaining methods are listed below:

- `AnyObject` is a protocol, that all classes implicitly conform to. If `AnyObject` occurs in a protocol composition type, the annotated variable must be assigned a class type. This is what is called a class existential. If a class is also part of the same protocol composition type, the `AnyObject` protocol becomes irrelevant. The method `removeAnyObjectConformance()` was implemented to remove all occurrences of `AnyObject` from a protocol composition type (lines 8 to 13).
- The `getOverloadSetFromProtocolExtensions()` method is used to gather all members of adopted protocols listed in the protocol composition type (lines 28 to 45). It was updated to also collect the members of the `conformanceClass`. Note, that `conformanceClass` can be a `ClassTypeBinding`, but can also be a `GenericTypeInstace`.
- `transform()` is a method that takes a function as its argument, and applies this function to adopted protocols. Because the protocol composition type has a new property `conformanceClass`, `transform()` with the given function has to be called on this as well (lines 59 to 62). One can think of `transform()` as a `map()` function on trees. It can for example be used to simplify types or to remove sugar, depending on the function provided.
- Finally, the method `getExistentialTypeConformanceClass()` was added, returning the property `conformanceClass`. Note the `@Override` annotation. A default implementation of this newly introduced method was added to the `IType` interface, which is inherited by the `ProtocolCompositionType` class.

Section 2.2 already gave a quick introduction into archetypes. Archetypes are used to represent generic parameters. It is therefore not surprising, that archetypes are affected by the changes to the protocol composition type, because generic parameters can have a type annotation.

```
1  public abstract class AbstractBaseArchetypeType extends AbstractBaseBinding
2      implements IArchetypeType {
3    // ...
4    private final ProtocolCompositionType protocolCompositionType =
5        new ProtocolCompositionType();
6    // ...
7
8    @Override
9    public void setConformanceClass(ITypeBinding conformance) {
10     protocolCompositionType.setConformanceClass(conformance);
```

```
11    }
12
13    @Override
14    public ITypeBinding getConformanceClass() {
15      validate();
16      return protocolCompositionType.getConformanceClass();
17    }
18
19    // ...
20  }
```

Listing 3.23: *Abstract base archetype type*

Listing 3.23 shows an excerpt of class `AbstractBaseArchetypeType`. The class already has a property of type `ProtocolCompositionType` (line 3). A getter and a setter method for the `conformanceClass` had to be added (lines 9 to 17), that reroute to the original methods defined on the protocol composition type itself. Again, both methods are overridden. The original declaration was added in the `IArchetypeType` interface. This concludes the changes on the bindings.
The indexer has to perform the transformation of AST types into index types. The `TypeUtils` class holds the necessary methods. It is shown in Listing 3.24.

```
1   final public class TypeUtils {
2     // ...
3
4     private static ProtocolCompositionType transformProtocolCompositionType(
5         ASTProtocolCompositionType astProtocolCompositionType) {
6       final ProtocolCompositionType protocolCompositionType =
7           new ProtocolCompositionType();
8       boolean isCompositionInvalid = false;
9
10      for(final ASTTypeIdentifier astType : astProtocolCompositionType
11          .getProtocolIdentifiers()) {
12        IType type = getIndexType(astType);
13        type = removeSugar(type);
14
15        if(type instanceof ProtocolCompositionType) {
16          final ProtocolCompositionType protocolComposition =
17              (ProtocolCompositionType)type;
18          for(final ProtocolTypeBinding protocol : protocolComposition
19              .getAdoptedProtocols()) {
20            protocolCompositionType.addAdoptedProtocol(protocol);
21          }
22          if(protocolComposition.getConformanceClass() != null) {
23            if(protocolCompositionType.getConformanceClass() == null) {
24              protocolCompositionType.setConformanceClass(
25                  protocolComposition.getConformanceClass());
26            } else if(!protocolCompositionType.getConformanceClass()
27                .equals(protocolComposition.getConformanceClass())) {
28              isCompositionInvalid = true;
29            }
30          }
31        } else if(type instanceof ProtocolTypeBinding) {
32          final ProtocolTypeBinding protocol = (ProtocolTypeBinding)type;
```

```
33          protocolCompositionType.addAdoptedProtocol(protocol);
34        } else if(TypeUtils.isClassType(type)) {
35          final ITypeBinding conformanceClass = (ITypeBinding)type;
36          if(protocolCompositionType.getConformanceClass() == null) {
37            protocolCompositionType.setConformanceClass(conformanceClass);
38          } else if(!protocolCompositionType.getConformanceClass()
39              .equals(conformanceClass)) {
40            isCompositionInvalid = true;
41          }
42        } else {
43          isCompositionInvalid = true;
44        }
45      }
46
47      if(protocolCompositionType.getConformanceClass() != null) {
48        protocolCompositionType.removeAnyObjectConformance();
49      }
50
51      if(isCompositionInvalid) {
52        return null;
53      }
54
55      return protocolCompositionType;
56    }
57
58    // ...
59
60    public static void addAdoptedProtocolsToArchetype(IArchetypeType archetype,
61        IASTType astType) {
62      final IType type = resolveAndCleanType(astType);
63      if(type instanceof ProtocolTypeBinding) {
64        final ProtocolTypeBinding protocol = (ProtocolTypeBinding)type;
65        archetype.addAdoptedProtocol(protocol);
66      } else if(type instanceof ProtocolCompositionType) {
67        final ProtocolCompositionType protocolCompositionType =
68            (ProtocolCompositionType)type;
69        for(final ProtocolTypeBinding protocol : protocolCompositionType
70            .getAdoptedProtocols()) {
71          archetype.addAdoptedProtocol(protocol);
72        }
73        if(protocolCompositionType.getConformanceClass() != null) {
74          archetype.setConformanceClass(protocolCompositionType
75              .getConformanceClass());
76        }
77      } else if(TypeUtils.isClassType(type)) {
78        final ITypeBinding classType = (ITypeBinding)type;
79        archetype.setConformanceClass(classType);
80      }
81    }
82
83    // ...
84  }
```

Listing 3.24: *Type utils*

- The static method `transformProtocolCompositionType()` (lines 4 to 56) takes an AST type `ASTProtocolCompositionType` as its argument. The

method is responsible for the transformation of this AST type into a corresponding index type `ProtocolCompositionType`. In addition, the method checks, if the protocol composition type is semantically valid. In the previous version of Tifig, as soon as a non-protocol type identifier occurred, `null` was returned, signaling an invalid transformation. This was no longer a possible approach. Instead, a distinct behavior was implemented depending on whether the type identifier is an instance of `ProtocolCompositionType` (lines 15 to 30), a `ProtocolTypeBinding` (lines 31 to 33) or a class type (lines 34 to 41). Finally, if the property `conformanceClass` of `ProtocolCompositionType` is set, all occurrences of the `AnyObject` protocol can be removed (lines 47 to 49). If the semantic check failed at some point, the `isCompositionInvalid` flag is set to `true`. In that case, `null` is returned. Otherwise, the index type of the protocol composition type is returned as the result.

- After a new archetype is created, the protocols it adopts must be added to it. The method `addAdoptedProtocols()` is used to achieve this. Here, the lines 73 to 80 were added, for the case where the protocol composition of the archetype is constrained by a class conformance.

```
1  public class ConstraintSystem {
2    // ...
3
4    private void addConstraintsForArchetype(IArchetypeType archetype,
5        IType baseType, Map<IArchetypeType, IType> replacements) {
6      // ...
7
8      if(archetype.getConformanceClass() != null) {
9        simplifier().addConstraint(ConstraintKind.Subtype, baseType,
10           archetype.getConformanceClass());
11     }
12
13     // ...
14   }
15
16   // ...
17 }
```

Listing 3.25: *Constraint system*

A class conformance constraint on a type must be checked by the type checker. In Section 2.1.6, it was shown how the type checker builds a constraint system for every expression where type-checking is necessary. Listing 3.25 shows part of the `addConstraintsForArchetype()` method of class `ConstraintSystem`. The method was extended to generate an additional constraint for archetypes, namely a subtype constraint on the base type to the `conformanceClass` of the underlying protocol composition type.

```
1  public class CSSimplify {
2    // ...
3    private SolutionKind matchConcreteTypes(IType type1, IType type2,
4        IType desugar1, IType desugar2, ConstraintKind kind,
5        int flags, List<ConversionRestrictionKind> conversions) {
6      // ...
7
8      if(kind.ordinal() >= ConstraintKind.Subtype.ordinal()) {
9        // ...
10
11       if(type1.getExistentialTypeConformanceClass() != null &&
12           TypeUtils.isClassType(type2)) {
13         conversions.add(ConversionRestrictionKind.Superclass);
14       }
15
16       // ...
17     }
18
19     // ...
20   }
21
22   // ...
23
24   private SolutionKind matchSuperclassTypes(IType type1, IType type2,
25       int flags) {
26     final int subflags = getDefaultDecompositionOptions(flags);
27     final IType classType1 = type1;
28     final IType classType2 = type2;
29
30     for(ITypeBinding super1 = TypeUtils.getSuperClass(classType1);
31         super1 != null; super1 = TypeUtils.getSuperClass(super1)) {
32       if(matchTypes(super1, classType2, ConstraintKind.Equal,
33           subflags) == SolutionKind.Solved) {
34         return SolutionKind.Solved;
35       }
36     }
37
38     return SolutionKind.Error;
39   }
40
41   private SolutionKind matchExistentialTypes(IType type1, IType type2,
42       ConstraintKind kind, int flags) {
43     // ...
44
45     if(type2.getExistentialTypeConformanceClass() != null) {
46
47       if(type1.isExistentialType()) {
48         if(type1.getExistentialTypeConformanceClass() != null) {
49           return matchSuperclassTypes(
50               type1.getExistentialTypeConformanceClass(),
51               type2.getExistentialTypeConformanceClass(),
52               subflags);
53         }
54       } else if(TypeUtils.isClassType(type1)) {
55         return matchSuperclassTypes(type1,
56             type2.getExistentialTypeConformanceClass(),
57             subflags);
58       }
59
60       return SolutionKind.Error;
61
62     }
```

```
63
64      // ...
65    }
66
67    // ...
68  }
```

Listing 3.26: *Constraint system simplifier*

The final adjustment that was undertaken took place in the simplifier that is part of the constraint system. Listing 3.26 shows an excerpt of the methods of class `CSSimplify` described below:

- The method `matchConcreteTypes()` checks whether two types provided as arguments fulfil a given constraint. While doing so, it often happens that new constraints have to be generated. The method had to be adjusted in order to create a new subtype conversion restriction, whenever an existential type with a class conformance encounters a class type (lines 11 to 13).
- `matchExistentialTypes()` (lines 41 to 65) is triggered by adding the superclass conversion restriction. The method assumes that the argument `type2` is an existential type and then tries to verify the constraint represented by the argument `kind` between `type1` and `type2`. If `type1` too is an existential type, the `matchSuperClassTypes()` method is called with the `conformanceClass` properties of `type1` and `type2`. If `type1` is already a class type, `matchSuperClassTypes()` is called with `type1` and the conformance class of `type2`.
- `matchSuperClassTypes()` checks, if the argument `type2` is a super class of `type1`. The method had to be adjusted to work for all kinds of types and not only `ITypeBinding` instances (lines 24 to 39).

If the types provided match, `SolutionKind.Solved` is returned, indicating that the constraint under inspection is fulfilled and must no longer be kept active in the constraint system.

3.2.4. Summary

This section explained how class and subtype existentials were implemented in Tifig. It was highlighted, that semantic changes can have a large influence on the implementation of the indexer, while the underlying syntax does not change. Finally, it was shown that a thorough study of specifications, proposals and existing implementations is necessary to avoid unintended behavior of the IDE.

3.3. Where Clauses on Associated Types

Associated types are used in Swift as a way to define placeholder types in protocol definitions.

3.3.1. Introduction

The associated type stands for a type, that can be used to define members in the protocol. When the protocol is adopted, the associated type must be replaced with an actual type. This is best shown with an example. Listing 3.27 illustrates the use of associated types in a protocol declaration.

```
1  protocol Container {
2    associatedtype Element
3    mutating func append(_ element: Element)
4    var count: Int { get }
5    subscript(i: Int) -> Element { get }
6  }
7
8  extension Container {
9    func fold<Result>(_ initialValue: Result, _ binaryOperation:
10       (Result, Element) -> Result) -> Result {
11     guard count != 0 else { return initialValue }
12     var intermediate = initialValue
13     for index in 0..<count {
14       intermediate = binaryOperation(intermediate, self[index])
15     }
16     return intermediate
17   }
18 }
19
20 struct NumberContainer: Container {
21   private var elements = [Int]()
22   typealias Element = Int
23   mutating func append(_ element: Element) {
24     elements.append(element)
25   }
26   var count: Int { return elements.count }
27   subscript(i: Int) -> Element { return elements[i] }
28 }
29
30 var nc = NumberContainer()
31 nc.append(1); nc.append(2); nc.append(3)
32 print(nc.fold(0, +)) // prints: 6
```

Listing 3.27: *Associated type example*

The example shows the definition of a protocol `Container`. It has an associated type `Element`, that is a placeholder for the type of elements the container has (line 2). The associated type can then be used to declare the type signatures of methods within the protocol. Here, `append()` is declared, taking one argument of

type `Element` (line 3). Also, a subscript is declared taking an `Int` as its argument and returning an `Element` (line 5). An extension is then used to define the method `fold()` (lines 8 to 19). The associated type `Element` is still only a placeholder. Note, that the extension makes use of `Element` without redeclaring it as associated type. Extensions can make use of all associated types of the original protocol.
The struct `NumberContainer` conforms to the protocol `Container`. It implements the getter for the `count` property (line 27), the method `append()` (lines 24 to 26) and the subscript (line 28). It does so by defining an underlying property `elements` of type `Array<Int>` (line 22) on which the corresponding methods are performed. But most importantly, a type alias `Element` is defined for type `Int` (line 23). This replaces the associated type `Element` with `Int` for all instances of `NumberContainer`.
Finally, a new `NumberContainer` is instantiated and assigned to variable `nc` (line 31). The container is then filled with the numbers 1, 2 and 3 (line 32). Then, the fold method is called on `nc`, and the result is printed to the console (line 33).
Figure 3.9 shows the grammar for associated type declarations in Swift 3.1. Once again, it is taken from [16].

protocol-associated-type-declaration → *attributes*$_{opt}$ *access-level-modifier*$_{opt}$ `associatedtype` *typealias-name* *type-inheritance-clause*$_{opt}$ *typealias-assignment*$_{opt}$

Figure 3.9.: *Associated type declaration grammar Swift 3.1*

The name of the non-terminal symbol for associated type declarations already gives away, that associated types can only be declared within protocols. The terminal symbol `associatedtype` is followed by a mandatory non-terminal symbol *typealias-name*. The symbol *type-inheritance-clause* represents the types the associated type inherits from. `associatedtype A: SomeClass, Equatable` is an example for an associated type declaration with a type inheritance clause. The *typealias-assignment* symbol represents the assignment of a (default) type to the associated type declared, for example `associatedtype A = Int`.

3.3.2. Constraining Associated Types with Where Clauses

Protocols cannot make use of generic type paramjeters. To implement generic behavior in protocols, associated types have to be used. However, because associated types could only be constrained with inheritance constraints, associated types did not have the same expressive power as generic types parameters [26]. To change this, Bandes-Storch, Gregor and Hart wrote [26], a proposal for allowing where clauses in associated type declarations. The idea was to make associated types as

expressive as generic type parameters and in addition, to improve some designs in the standard library. Listing 3.28 reveals the core issue.

```
1  extension MutableCollection where Indices.Iterator.Element == Index {
2    mutating func mapInPlace(_ transform: (Iterator.Element) throws ->
3        Iterator.Element) rethrows {
4      for index in indices {
5        self[index] = try transform(self[index])
6      }
7    }
8  }
9
10 var arr = [1,2,3]
11 arr.mapInPlace({ x in x * 2 })
12 print(arr) // prints: [2, 4, 6]
```

Listing 3.28: *Limitations of associated types in Swift 3.1*

The Swift standard library defines some functionality of its types in extensions. The example shows such an extension of the protocol `MutableCollection`. In the extension, the function `mapInPlace()` is defined, that transforms all elements of the mutable collection with the provided function `transform()`. The generic where clause of the protocol extension states, that `Indices.Iterator.Element == Index`. This condition has to be fulfilled, or else `mapInPlace()` would not work. In Swift 3.1, this requirement had to be stated in this and every other extension of `MutableCollection`, because the constraints could not be defined on the associated types of protocol `MutableCollection` itself. Listing 3.29 shows the same example including the improvements possible with Swift 4.

```
1  extension MutableCollection {
2    mutating func mapInPlace(_ transform: (Element) throws ->
3        Element) rethrows {
4      for index in indices {
5        self[index] = try transform(self[index])
6      }
7    }
8  }
9
10 var arr = [1,2,3]
11 arr.mapInPlace({ x in x * 2 })
12 print(arr) // prints: [2, 4, 6]
```

Listing 3.29: *Where clause on associated type*

The generic where clause of the previous example is gone. It was moved to the generic where clause of the associated type declaration of `Indices` in protocol `Collection`, which is inherited by `MutableCollection`. The excerpt of the standard library containing this declaration is shown in Listing 3.30 (lines 5 to 8).

```
1  /* Collection.swift */
2  public protocol Collection: Sequence where SubSequence: Collection {
3    // ...
4
5    associatedtype Indices : Collection = DefaultIndices<Self>
6        where Indices.Element == Index,
7              Indices.Index == Index,
8              Indices.SubSequence == Indices
9
10   // ...
11 }
12
13 /* Sequence.swift */
14 public protocol Sequence {
15   // ...
16   associatedtype Element
17
18   // ...
19   associatedtype Iterator : IteratorProtocol
20       where Iterator.Element == Element
21
22   // ...
23 }
```

Listing 3.30: *Container and sequence protocol*

The listing contains a second excerpt of file `Sequence.swift`. In the `Sequence` protocol declaration, an associated type `Iterator` is declared. In its generic where clause the requirement `Iterator.Element == Element` is stated (line 20). The introduction of this where clause has the effect, that the type annotations `Iterator.Element` could be replaced by `Element`, which was done in the second code example above (Listing 3.29, lines 2 and 3).
The introduction of generic where clauses on associated types removed a lot of duplicate code from the Swift standard library. Associated types can now be generally constrained in the protocol they are declared. If an extension wants to introduce new constraints on the inherited associated types, this is still possible, using the where clause of the extension or redeclaring (shadowing) the associated type with new constraints. Note, that not only protocol extensions were affected by the aforementioned issue. Also, every other generic context, for example generic functions, had to provide the constraints on associated types they made use of.
Because where clauses on associated types did not exist in Swift 3.1, the grammar for associated type declarations needed to be updated for Swift 4. The updated grammar is shown in Figure 3.10.

protocol-associated-type-declaration → *attributes*$_{opt}$ *access-level-modifier*$_{opt}$ `associatedtype`
typealias-name *type-inheritance-clause*$_{opt}$
typealias-assignment$_{opt}$ ***generic-where-clause*$_{opt}$**

Figure 3.10.: *Associated type declaration grammar Swift 4*

The optional non-terminal symbol *generic-where-clause* was introduced at the end of the *protocol-associated-type-declaration*. The change is highlighted in bold in the figure above.

3.3.3. Implementation

In this section, the implementation details for where clauses on associated types are documented. Again, the section is arranged following the usual structure.

Changes to the Lexer

The introduction of where clauses on associated types did not require new keywords or other lexical complements. The syntax of the generic where clause is the same as is used in class declarations, generic function declarations and so on. Therefore, the lexer did not need to be changed.

Changes to the Parser and the AST

The new syntax required to change the way of parsing associated type declarations. As was the case in Section 3.1.3, the class `DeclParser` needed to be extended. Listing 3.31 shows the updated method `associatedTypeDecl()`.

```
1  private AssociatedTypeDecl associatedTypeDecl() throws RecognitionException {
2    match(Kind.KW_ASSOCIATEDTYPE);
3    final Name name = parse(this::name);
4    final TypeInheritanceClause typeInheritanceClause =
5        parseIf(Kind.COLON, this::typeInheritanceClause);
6    IASTType type = null;
7    if(la(1).is(Kind.EQUAL)) {
8      match(Kind.EQUAL);
9      type = parse(typeParser::type);
10   }
11   final GenericWhereClause genericWhereClause = parseIf(Kind.KW_WHERE,
12       this::genericWhereClause);
13   return new AssociatedTypeDecl(name, typeInheritanceClause, type,
14       genericWhereClause);
15 }
```

Listing 3.31: *Parsing associated type declarations*

Lines 11 to 12 were added, in order to parse a generic where clause, if present. The method returns a new instance of class `AssociatedTypeDecl`. The constructor call was extended to take an additional argument of type `GenericWhereClause` (lines 13 to 14). Of course, this required to change the implementation of class `AssociatedTypeDecl`. The changed parts of this class are shown in Listing 3.32.

```
1  public class AssociatedTypeDecl extends Decl implements IProtocolMemberDecl,
2      ITypeInheritanceClauseOwner {
3    // ...
4    private final GenericWhereClause genericWhereClause;
5
6    public AssociatedTypeDecl(Name name,
7        TypeInheritanceClause typeInheritanceClause, IASTType type,
8        GenericWhereClause genericWhereClause) {
9      this.name = name;
10     this.typeInheritanceClause = typeInheritanceClause;
11     this.type = type;
12     this.genericWhereClause = genericWhereClause;
13   }
14
15   // ...
16
17   public GenericWhereClause getGenericWhereClause() {
18     return genericWhereClause;
19   }
20
21   // ...
22
23   @Override
24   public boolean accept(ASTVisitor visitor) {
25     return acceptVisitor(visitor, getAttributes(), name,
26         typeInheritanceClause, type, genericWhereClause);
27   }
28 }
```

Listing 3.32: *Associated type declaration*

First, a new property `genericWhereClause` of type `GenericWhereClause` was added (line 3), along with a corresponding getter method (lines 17 to 19). The constructor was extended to take an additional argument of the same class, and assigning it to the property (lines 6 to 13). Finally, the overridden `accept()` method had to be adjusted, so that the call to `acceptVisitor()` takes the `genericWhereClause` property as its final argument (lines 22 to 27). This completes the changes to the parser and AST infrastructure of Tifig.
To round off the section, the AST generated for the associated type declaration in Listing 3.29 is given in Figure 3.11. As usual, the important AST nodes are highlighted with a thick border. The `AssociatedTypeDecl` node has a `GenericWhereClause` node as its second child. A `SameTypeRequirement` node has the `GenericWhereClause` node as its parent. The requirement always has two `ASTTypeIdentifier` nodes as its children. Note, that the left `ASTTypeIdentifier` is parent to two `ASTTypeIdentifierElement` nodes. This represents the code `PartialSequence.Element`.

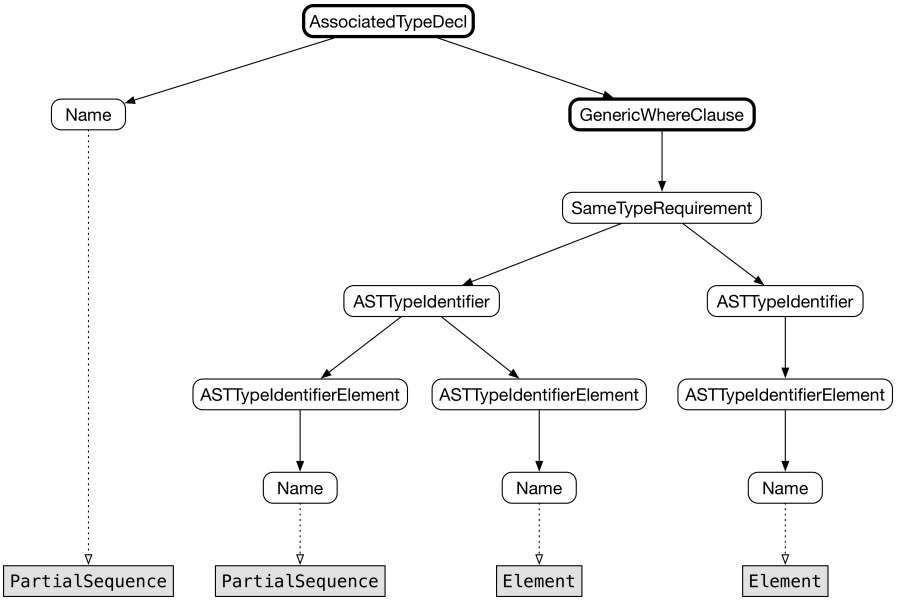

Figure 3.11.: *AST for associated type declaration*

Changes to the Indexer and the Type Checker

Associated types are closely related to generic type parameters. The main difference is, that protocols inheriting associated types of protocols they themselves conform to do not show this in their declarations. Instead, the inherited associated types can be used in the protocols body and can even be shadowed if the same identifier is redeclared. In addition, inherited associated types can be further constrained in the current scope.
When looking at the proposal text, the changes necessary to introduce where clauses on associated types seem to be straightforward and bound to a specific location in the code, namely to associated type declarations. However, this is not the case. In order to achieve the correct resolution of associated types, the general processing of generic where clauses (no matter the declaration they are attached to) had to be adjusted. This is best shown with an example. Listing 3.33 shows the example code.

```
1  protocol P {
2    associatedtype P1
3    associatedtype P2
4  }
5  protocol Q: P where P1: R {
6    associatedtype P2 where P2 == P1.R1
7  }
8  protocol R {
9    associatedtype R1
10 }
```

```
11 func f<T: Q>(x: T) -> T where T.P2: Equatable {
12   return x
13 }
```

Listing 3.33: *Associated type mechanics*

The listing shows some of the mechanisms regarding associated types. The protocol `P` has two associated types `P1` and `P2` (lines 1 to 4). Protocol `Q` conforms to `P` and therefore inherits its associated types. Already in the where clause of the protocol declaration, the associated type `P1` is constrained further, in that it has to conform to protocol `R` (line 5). Note, that this constraint is only concerning the inherited associated type in the context of `Q` and has no influence on `P1` of `P`. Line 6 shows another way to shadow an inherited associated type. In the where clause of the associated type declaration, `P2` is constrained to have the same type as `P1.R1`. Finally, a generic function `f()` takes an argument of the generic type parameter `T`, which has to conform to `Q`. In the generic where clause of the function, the nested associated type `P2` is constrained to conform to the protocol `Equatable` (line 11). Summarizing the example, it can be stated, that requirements on associated types can come from various locations in the code and can be constrained even more in every generic context they are used in.
Listing 3.34 shows an excerpt of the class `TypeUtils`, where the logic for resolving generic parameters is located.

```
1  final public class TypeUtils {
2    // ...
3
4    private static IRequirement[] sortRequirementsOfGenericWhereClause(
5        IRequirement[] requirements) { /*...*/ }
6
7    // ...
8
9    public static void resolveGenericParameters(IBinding binding,
10       GenericParameterClause genericParameterClause,
11       GenericWhereClause genericWhereClause,
12       boolean topLevelArchetypesShouldBeVirtual) {
13
14     if(genericParameterClause != null) {
15       validateGenericParameterClause(genericParameterClause);
16     }
17
18     if(genericWhereClause != null) {
19       processWhereClauseRequirements(binding, genericParameterClause,
20       genericWhereClause, topLevelArchetypesShouldBeVirtual);
21     }
22
23   }
24
25   // ...
26
27   private static void processWhereClauseRequirements(
```

```
28        IBinding binding, GenericParameterClause genericParameterClause,
29        GenericWhereClause genericWhereClause,
30        boolean topLevelArchetypesShouldBeVirtual) { /*...*/ }
31
32    // ...
33
34    private static void processRequirement(IArchetypeType archetype,
35        TypeIdentifierElement[] elements, boolean isConformanceRequirement,
36        IASTType otherType) { /*...*/ }
37
38    // ...
39  }
```

Listing 3.34: *Type utils overview*

The listing shows the signatures of the most important methods that have been adjusted for better processing of associated types. The public and static method `resolveGenericParameters()` is shown completely (lines 9 to 23), because it is the entry point for validating declaration bindings that have a generic context. Of special importance regarding associated types is line 19, where the private method `processWhereClauseRequirements()` is called, if a generic where clause is present.
Parts of the implementation of `processWhereClauseRequirements()` are given in Listing 3.35.

```
1   private static void processWhereClauseRequirements(IBinding binding,
2       GenericParameterClause genericParameterClause,
3       GenericWhereClause genericWhereClause,
4       boolean topLevelArchetypesShouldBeVirtual) {
5
6     final IRequirement[] requirements =
7         sortRequirementsOfGenericWhereClause(
8             genericWhereClause.getRequirements());
9     final ITypeBinding enclosingSelfType =
10        getSelfTypeBinding(genericWhereClause);
11    final ITypeBinding selfType;
12
13    // Reset the enclosingSelfType and the selfType depending on the kind
14    // of binding provided.
15
16    for(final IRequirement requirement : newRequirements) {
17      final ASTTypeIdentifier lhs = requirement.getTypeIdentifier();
18      final IASTType rhs = requirement.getType();
19      ASTTypeIdentifier constrainedType = null;
20      IASTType otherType = null;
21
22      // Generate virtual archetypes if necessary and determine, if lhs
23      // or rhs is the constrained type of the requirement.
24
25      if(constrainedType != null && otherType != null) {
26        final TypeIdentifierElement[] elements =
27            getTypeIdentifierElements(constrainedType);
28        final IArchetypeType archetype =
29            (IArchetypeType)elements[0].getName().getBinding();
```

```
30          final boolean isConformanceRequirement =
31              requirement instanceof ConformanceRequirement;
32          processRequirement(archetype, elements,
33              isConformanceRequirement, otherType);
34        }
35
36      }
37
38  }
```

Listing 3.35: *Process where clause requirements*

The complete implementation ranges over a hundred lines of code handling different types of bindings that are presented as the method's first argument. In the listing, these binding specific details are replaced by comments, describing the process in general (lines 13 to 14 and 22 to 23). Essentially, what the method does is to solve three problems. These problems are listed in the following:

- **Order of Requirements:** All requirements of the provided where clause are iterated to be processed. Analyzing the previous implementation of the processing revealed, that the order in which requirements are processed has an impact on the resolution. To see that, an example is given in Listing 3.36.

```
1  protocol P {
2    associatedtype P1
3  }
4  protocol Q {
5    associatedtype Q1
6    func f() -> Q1
7  }
8  func g<T>(x: T) where T.P1 == Int, T.P1 == T.Q1, T: P, T: Q {
9    let z = x.f()
10 }
```

Listing 3.36: *Requirement order example*

The requirements in the function's generic where clause are iterated from left to right. Without reordering the requirements, Tifig would not be able to resolve the requirements. To resolve `T.P1 == T.Q1`, it must be known first that `T` has inherited associated types `P1` and `Q1`. The sorting of requirements makes sure, that conformance requirements are evaluated first. The new iteration order is `T: P, T: Q, T.P1 == Int, T.P1 == T.Q1`. In addition, the deeper an associated type is nested, the later its conformances have to be evaluated, for example `T.P1: Equatable` has to be evaluated after `T: P`. The sorting of the requirements is implemented in method `sortRequirementsOfGenericWhereClause()`, which is called in the

method `processWhereClauseRequirements()` above (Listing 3.35, lines 6 to 8).

- **Virtual Archetypes:** Associated types are represented by archetypes, as is the case with generic type parameters. While iterating over the requirements, a lot of validation is done on demand. Virtual archetypes have to be created, if an inherited associated type is shadowed and more constrained. This virtual archetype does not replace previous archetypes representing the same associated type. It is created in addition and is valid in the context where it was created. A simple example is shown in Listing 3.37.

```
1 protocol P {
2   associatedtype A
3 }
4 extension P where A == Int {}
```

Listing 3.37: *Virtual archetype example*

An extension of protocol `P` is declared, constraining associated type `A` to be type `Int`. This constraint is only valid in the scope of the extension. To map this into bindings usable by the indexer, an archetype is created for associated type `A`. For the extension, an additional archetype that is marked virtual is created for `A`. On this virtual archetype, the `sameType` property is set to `Int`. The method `processWhereClauseRequirements()` is responsible for creating virtual archetypes for the requirements, if necessary.

- **`lhs` vs. `rhs`:** For each requirement, it must be determined, which of the types in the requirement is the constrained type. This is important, because Swift does not define on which side of the requirement the constrained type must occur. For example, one could write `A == Int` or `Int == A`. Note, that `Int` is defined as a struct in the Swift standard library. Therefore, a semantic analysis is always necessary for this determination. The distinction between the constrained and the other type is necessary, because it must be known to which binding the requirement shall be attached.

Once all of the above problems are solved and the constrained type is identified, the method `processRequirement()` is called (lines 32 to 33). The implementation of `processRequirement()` is shown in Listing 3.38.

```
1 private static void processRequirement(IArchetypeType archetype,
2     TypeIdentifierElement[] elements, boolean isConformanceRequirement,
3     IASTType otherType) {
4   final IArchetypeType nestedArchetype = archetype.getNestedArchetype(
```

```
5         Arrays.copyOfRange(elements, 1, elements.length));
6     if(nestedArchetype == null) {
7       return;
8     }
9
10    if(isConformanceRequirement) {
11      addAdoptedProtocolsToArchetype(nestedArchetype, otherType);
12    } else {
13      nestedArchetype.setSameType(resolveAndCleanType(otherType));
14    }
15  }
```

Listing 3.38: *Process requirements*

As can be seen, the requirement is not directly translated into a constraint for the constraint system, but added as either an adopted protocol or a same type to the nested archetype, depending on the type of requirement. The type checker builds the constraints for the requirements on archetypes when it encounters an expression where the type represented by the archetype is used.

3.3.4. Summary

The documentation provided in this section shows only the most essential changes made in the process of adding where clauses for associated types. Overall, a multitude of 35 class files had to be adjusted to support the adapted infrastructure, including a large refactoring of existing source code. However, the current infrastructure is still not able to achieve all resolutions that are supposed to work in Swift 4. The known limitations are listed in Section 3.5.

3.4. Testing

To make sure that Tifig's lexer, parser and indexer behave as intended, a large JUnit test suite of automated tests is maintained. The tests are located in the plug-in `ch.hsr.ifs.tifig.core.tests`. During the course of this thesis, the test cases for the parser and indexer were extended. Whenever possible, the introduction of new features took place in a test driven manner, meaning that the tests were written before the implementation started.
In the following sections, a short overview of the files containing the newly introduced tests is given.

3.4.1. Parser Tests

Below is the list of parser test files that were extended during the implementation of all three proposals.

- `SubscriptDeclTests.java` (2 tests added)
- `ProtocolDeclTests.java` (5 tests added)

A total of seven new tests were added to test the new parser behavior.

3.4.2. Indexer Tests

Below is the list of indexer test files that were extended during the implementation of all three proposals.

- `GenericSubscriptTests.java` (65 tests added, new file)
- `AssociatedTypeBindingTests.java` (10 tests added)
- `ExtensionBindingTests.java` (2 tests added)
- `GenericFunctionTests.java` (10 tests added)
- `ProtocolTypeBindingTests.java` (2 tests added)
- `VirtualArchetypeCreationTests.java` (18 tests added, new file)
- `ClassSubtypeExistentialTests.java` (29 tests added, new file)

A total of 136 new tests were added to test the new indexer behavior.

3.5. Known Limitations

During the course of this thesis, it was discovered that the existing infrastructure of Tifig is not capable of resolving bindings in all situations where associated types are involved. Because this discovery was made in a late stage, a refactoring of the infrastructure was not possible anymore. Instead, the current infrastructure was adjusted to its limits and made working in most real-world use cases. This means, that the situations in which resolution is not possible often appear constructed and seem unlikely to appear in code of business applications. In this section, the known limitations of Tifig's infrastructure and their causes are discussed.

3.5.1. Multiple Same Type Requirements

In Section 3.3.3, it was shown how requirements can be reordered in order to achieve a correct resolution of bindings. However, processing conformance requirements first is not always enough to ensure a complete resolution. Listing 3.39 shows the same example that was used in Section 3.3.3, extended with an additional function definition for function `g()`, where the order of the same type requirements is reversed.

```
1  protocol P {
2    associatedtype P1
3  }
4  protocol Q {
5    associatedtype Q1
6    func f() -> Q1
7  }
8  func f<T>(x: T) where T.P1 == Int, T.P1 == T.Q1, T: P, T: Q {
9    let z = x.f()
10 }
11 func g<T>(x: T) where T.P1 == T.Q1, T.P1 == Int, T: P, T: Q {
12   let z = x.f()
13 }
```

Listing 3.39: *Multiple same type requirements*

When hovering over the name of constant `z` in function `f()`, Tifig shows the constant's type as `Int` as expected. When hovering over the name of constant `z` in function `g()`, however, Tifig shows the constant's type as `T.Q1`. Why is the resolution not working in function `g()`, when the requirements defined in the where clauses of both functions are the same.
The answer lies in the implementation of class `AbstractBaseArchetypeType`. Listing 3.40 shows the excerpt of this class, which is responsible for the outcome.

```
1  public abstract class AbstractBaseArchetypeType extends AbstractBaseBinding
2      implements IArchetypeType {
3    // ...
4    private IType sameType = null;
5    // ...
6
7    @Override
8    public void setSameType(IType sameType) {
9      if(sameType instanceof AssociatedTypeBindingReference) {
10       return;
11     }
12     this.sameType = sameType;
13   }
14
15   // ...
```

```
16 }
```

Listing 3.40: *Same type property of base archetype*

Archetypes described by this class have a property called `sameType` (line 4). Once a same type requirement is encountered and the constrained type has been determined, the method `processRequirement()` of class `TypeUtils` is called, which in turn calls `setSameType()`. The implementation of this method states, that if the argument `sameType` is not an instance of `AssociatedTypeBindingReference`, the `sameType` property shall be assigned the argument `sameType` (lines 8 to 13).
In the above example, the requirement `T.P1 == T.Q1` is the first same type requirement encountered. Both the types in this requirement are represented as an `AssociatedTypeBindingReference`. Because both bindings are determined to be constrained, and both binding's base type is part of the function's generic parameter clause, an educated guess is made and the left side is chosen to be the constrained type (this is not correct in all cases, but usually, people tend to write constraints this way). This means, that `setSameType()` is called on the archetype representing `P1` with argument `T.Q1`. But because `T.Q1` cannot be further evaluated and is still an instance of `AssociatedTypeBindingReference`, the `sameType` property on `P1` is not set.
Because each archetype can have at most one same type, this makes some sense. If a specific type was already assigned to the `sameType` property, it would be overwritten by a less specific one. This as well is an educated guess which works in most cases. In this example however, the same type requirement `T.P1 == T.Q1` has no effect at all in the resolution in function `g()`. This problem could not be solved so far.

3.5.2. Multiple Paths to the Same Associated Type

In Swift, it is possible to address an associated type via different paths. In this section, a limitation is discussed that results from exactly this. Listing 3.41 shows an example that cannot be properly resolved.

```
1  protocol Package {
2    associatedtype Content
3    associatedtype InternalPackage
4  }
5
6  protocol GiftBox: Package {
7    associatedtype InternalPackage: Package
8      where InternalPackage.InternalPackage == InternalPackage,
9        Content == InternalPackage.Content
10 }
11
```

```
12 func f<T: GiftBox>(_ c: T) where T.InternalPackage.Content == Int {
13   let x: T.InternalPackage.Content
14   let y: T.Content
15 }
```

Listing 3.41: *Same type requirement on associated type via different paths*

The goal in this example is to find the most specific type for the constants `x` and `y` in the function declaration of function `f()`. When looking at the code closer, it becomes clear quickly that `x` and `y` should both be of type `Int`. This is true for the constant `x`. Tifig is able to resolve the type of `x` to be `Int`. However, the type of `y` was resolved to be `T.Content`. What happened here?
The problem is, that `Content == InternalPackage.Content`, the same type requirement on the associated type `InternalPackage` (line 9), leads to the conclusion, that the associated type `Content` declared in protocol `GitfBox` is represented by the same archetype as the associated type `Content` that is accessed via `InternalPackage`. But this is not true in the context of function `f()`.
Because of the same type requirement in the where clause of function `f()`, a virtual copy was created for the associated type `Content` referenced via path `T.InternalPackage.Content`. This has to be done, because the requirement of this associated type to be `Int` is only valid in the generic scope of the function. Outside the function, the requirement has no influence.
When the type for `T.Content` is resolved, no same type requirement for `Int` is defined on the corresponding archetype. Tifig is currently not capable of collecting and merging together same types of such distributed archetypes.

3.5.3. Summary

To solve the limitations described in this section, a large scale refactoring of Tifig's infrastructure is necessary. Because the limitations were discovered too late and the Swift generics will be changed even more in the near future, it was decided not to attempt the refactoring. A potential solution to the problem described in Section 3.5.2 is presented in a theoretical manner in Section 4.3.2.

4. Study of the Swift Type Checker

Swift is a modern multi-paradigm language, incorporating object-oriented, protocol-oriented, functional and imperative characteristics. It does not come as a surprise, that the semantic analysis of Swift code is a difficult task. During the course of this thesis, the Swift type system and the according type inference mechanisms implemented in the Swift type checker were analyzed and discussed frequently. This chapter's intent is to provide insight into the type-checking process of Swift, to ease the access for future developers of Tifig. First, the characteristics of Swift's type system are described in Section 4.1. Then, an overview of the type checker is given in Section 4.2. Section 4.3 then describes the bottlenecks of the type checker's performance and existing optimization strategies and proposes two additional optimization mechanics.

4.1. Swift's Type System

In this section, Swift's type system is described by means of its characteristics. The goal is to categorize it, so that it can be better compared to other known type systems. An initial set of the most general characteristics is listed in the following:

- **Nominal:** Swift has a nominal or nominative type system. In nominal type systems, compatibility and equivalence of data types is determined by explicit declarations and the name of its types. The necessity of having a nominal type system arises, when types have to be checked for equivalence and also subtype relations [12].
- **Statically typed:** Being statically typed means that all types of variables, constants and functions must be set at compile-time. If types are not explicitly declared, the Swift compiler must be able to infer them, or else the compilation will fail, yielding a type error [28].
- **Strongly typed:** When a variable or constant is used in expressions, the Swift compiler checks if its type is fitting. For example, an argument provided in a function call expression must be of the type that is declared in the function's declaration or a type that is convertible to that type.

Although Swift's type system is nominal, it is often described as ML-like (ML's type system is structural). This comes from the fact, that type information flows in two directions during type inference. Listing 4.1 shows an example that illustrates this. The example is taken from [20].

```
1  func round(_ x: Double) -> Int { /* ... */ }
2  var pi: Double = 3.14159
3  var three = round(pi) // variable three has type Int
4
5  func identity<T>(_ x: T) -> T { return x }
6  var eFloat: Float = -identity(2.71828)  // numeric literal gets type Float
```

Listing 4.1: *Bi-directional type inference*

The variable `three` has type `Int`, after type inference is completed. The type information came from the subexpression `pi` (line 3). On the other hand, the numeric literal 2.71828 is inferred to have type `Float`. This information came from the type of the variable `eFloat` (line 6). The bi-directional type inference is achieved with a constraint-based type checker, that is explained in Section 4.2.
Because a formal definition of Swift's type system does not exist or is at least not publicly available, this section shall be concluded by listing the features of Swift that heavily influence the type system:

- Protocol-oriented programming
- Object-oriented programming via classes
- Function overloading
- Operator overloading
- Subtyping
- Constrained parametric polymorphism

Note, that Swift's type system does not support rank-2 polymorphism. Instead, generic function types are reserved for the types of named declarations only, which is analogous to the *let polymorphism* restriction in ML [20].

4.2. Type Checker Overview

Because type inference has to be done bi-directionally, it makes sense to use a type checker that follows the nature of Algorithm W, the classical Hindley-Milner type inference algorithm. However, constrained polymorphic types and

function overloading are features that are not part of the Hindley-Milner (ML) type system, but surely are part of Swift [20]. That this has a significant impact on the performance of the type checker is shown in Section 4.3.
As was mentioned in Section 4.1, Swift uses a constrained-based type checker. What this means becomes more clear when looking at the steps performed by the type checker:

- **Constraint generation:** Type checking starts with generating constraints for the expression under inspection. A type is assigned to each subexpression. If the type of a subexpression is unknown, a type variable is created. Programmatically, the constraint generator component walks an expression from the leaves up to the root of the expression's AST [20]. In addition to assigning types, the constraint generator defines constraints based on the restrictions given for the types and type variables. The result is a closed constraint system per expression.

- **Constraint solving:** The constraint solver component is responsible for finding all possible solutions of the before generated constraint system. It starts by assigning a fixed type to one of the type variables [30]. All constraints that describe a relation on this type variable are then simplified. This process goes on until a fixed type for each type variable is found or if the path leads to no valid solution. The solver can make use of backtracking if a dead end is encountered. But, in the end, all possible assignments of fixed types have to be explored. If the constraints are not solvable, the expression is ill-typed and the type-checking ends with an error. If one solution is found, the expression is well-typed and the type checker can move on to the solution application step. If multiple solutions are found, a ranking of the solutions has to take place. The ranking criteria can be found in [20]. If one solution is better than the others, the type checker goes to the next step. If not, the compilation stops because the expression is ambiguous.

- **Solution application:** In this final step, the type variables defined by the constraint generator component are replaced by the fixed types that were found in the solution step. This produces a fully type-checked expression that makes all implicit conversions and resolved overloads explicit [20].

The resemblance to the Hindley-Milner type inference algorithm is quite obvious. The naming of the steps changed slightly from *constraint generation*, *unification* and *annotation* to the ones described above. This completes the short overview of the type checker's components. A more detailed documentation can be found in [20].

4.3. Performance Improvements

The type checker's performance is of course dependent on the implementation of each of its components. However, the main problem is the size of the solution spaces of each constraint system. Because it is not enough to find one suitable solution, the complete solution space has to be explored. In the worst case, this solution space is exponential. In [20], Apple Inc. writes:

> *Solving the constraint systems generated by the Swift language can, in the worst case, require exponential time. Even the classic Hindley-Milner type inference algorithm requires exponential time, and the Swift type system introduces additional complications, especially overload resolution.*

To antagonize the impact of this on the type checker's performance, the scope of the type inference is limited to single expressions at a time. Solving the constraint systems locally helps to reduce the algorithmic complexity of the problem [28].

4.3.1. Existing Optimizations

Optimizing the performance of the type checker can roughly be translated to optimizing the techniques used to explore the solution space. In the documentation of the type checker [20], Apple Inc. describes their optimization techniques. In this section, these techniques are listed and explained superficially.

Constraint Graph

For each constraint system generated, the Swift type checker builds a constraint graph that illustrates the relations between type variables. Vertices represent the type variables, whereas edges represent the constraints of the constraint system. Because a non-simplified constraint can concern more than two type variables, an edge can connect more than two vertices, which makes the constraint graph a hypergraph. The use of the constraint graph can be demonstrated on an example. Listing 4.2 shows the code, for which the expression `(1, 2)` has to be type-checked.

```
1 let tuple: (Int, Double) = (1, 2)
```

Listing 4.2: *Constraint graph example code*

The generated constraints for expression `(1, 2)` are listed in following. Note, that `$T0` is the type variable for subexpression `1` and `$T1` for subexpression `2`, respectively.

- `$T0 LiteralConformsTo ExpressibleByIntegerLiteral`
- `$T1 LiteralConformsTo ExpressibleByIntegerLiteral`
- `$T0 Conversion Int`
- `$T1 Conversion Double`

Figure 4.1 shows the corresponding constraint graph that is generated along with the constraints. The notation and formatting is taken from [30].

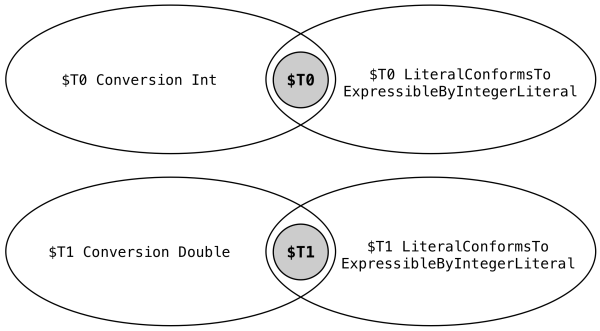

Figure 4.1.: *Constraint graph*

The important observation to be made in this example is that the constraint graph is not necessarily a connected graph. For the type checker, this means that each connected component of the graph can be processed separately. This makes the solution space smaller and divides it into separate solution spaces.

Simplification Worklist

After making an assumption on the fixed type of a type variable, the type checker tries to simplify the remaining constraints of the system based on that assumption. To prevent attempted simplifications on constraints unrelated to the type variable, the type checker keeps a worklist of constraints to be checked. To keep the worklist up to date, the constraint graph can be used to find related constraints. The simplification process is finished, when no more constraints are listed in the worklist.

Solver Scopes

Whenever the type checker makes an assumption, a new solver scope is created which captures the results of said assumption [20]. The solver scopes build a stack

that contains all assumptions made so far. If a solution is found, the solution can be built by popping all solver scopes from the stack and on the way, gathering the assumptions that led to the solution. If a solver scope leads to no solution, it can be popped from the stack and a new assumption can be made. This makes backtracking very easy.

Online Scoring

Solving constraints can lead to multiple valid solutions. As was mentioned in Section 4.2, the solutions must then be ranked in order to determine the best one. For that, a scoring system is part of the type checker's implementation that can determine a score for each (partial) solution. This score can be updated for each partial solution and with every assumption that is made. If the partial solution under inspection has a score that is worse than the best score of an already existing solution, the type checker does not need to go on. Instead, it can backtrack to the point, where the score of the partial solution was better. This mechanism is called online scoring.

Conclusion

The type checker documentation was last updated in October of 2016. It is possible, that the described optimizations were improved or that new optimization measures were added to the type checker's implementation. However, for some expressions, the type-checking can still take up to a few seconds. This means, that there is still room for improvement. This will be shown in Sections 4.3.2 and 4.3.3, respectively.

4.3.2. Associated Type Graphs

The optimization in this section is discussed using an example that makes use of existing code of the Swift standard library. The approach solves one of the hardest resolution challenges faced during the course of the thesis. Listing 4.3 shows combined excerpts of standard library implementations, along with a generic function that was defined in a custom Swift project.

```
1  /* Excerpts of different standard library files */
2
3  protocol Sequence {
4    associatedtype Element
5    associatedtype Iterator: IteratorProtocol
6        where Iterator.Element == Element
7    associatedtype SubSequence
8    func filter(_ isIncluded: (Element) throws -> Bool) rethrows -> [Element]
9  }
10 protocol _IndexableBase {
11   associatedtype Element
```

```
12    associatedtype SubSequence
13  }
14  protocol _Indexable: _IndexableBase {}
15  protocol Collection: _Indexable, Sequence {
16    associatedtype Iterator
17    associatedtype SubSequence: _IndexableBase, Sequence
18      where SubSequence.SubSequence == SubSequence,
19            Element == SubSequence.Element
20  }
21  protocol _IteratorProtocol {
22    associatedtype Element
23  }
24
25  /* Custom generic function */
26
27  func f<T: Collection>(_ c: T) where T.Iterator.Element == Int {
28    let x = c.filter { a in true }
29    let y: T.SubSequence.Element
30    let z: T.Iterator.Element
31    let v: T.Element
32  }
```

Listing 4.3: *Same type requirements on distributed associated types*

The underlying problem is the same as was described in Section 3.5.2. The goal is to find the most specific type for the constants `x`, `y`, `z` and `v`. When looking at the code closer, it becomes clear quickly that `x` is of type `Array<Int>` and all the other constants have type `Int`. However, with the exception of constant `z` (where the same type requirement is directly defined in the function's generic where clause), all other constants cannot be completely resolved by Tifig. Constant `x` is of type `Array<T.Element>`, `y` of type `T.SubSequence.Element` and `v` of type `T.Element`. The problem is the existence of multiple virtual archetypes for associated types `Element`, which are considered the same due to the same type constraints.
The idea is to solve this problem by building a graph that models the relations of associated types (declared and inherited) for each protocol. Because the generic function's generic type parameter `T` must conform to protocol `Collection`, the creation and usage of the associated type graph of `Collection` shall be demonstrated.

Graph Generation Procedure

The associated type graph can be built following the steps described below:

1. **Create self type node:** Create a node that represents the protocol's self type.
2. **Connect declared associated types:** Create a node for each of the protocols associated types. Provide an edge from the self type node to the associated types.

3. **Collect inherited associated types:** For each of the inherited protocols, go to step 1. Connect the self type node of the recursively generated graph with the current protocol's self type node with an edge that represents conformance. Connect the associated types that are required to be the same (by same type requirements) with an edge representing type equality.

4. **Resolve inheritance on associated types:** Step 3 collects the associated types of protocols. In this step, the same procedure has to be done for protocols inherited on associated types themselves, with the associated type acting as self type node.

5. **Normalize the graph:** Merge conformances into the self type node. Merge nodes connected by same type edges, if they have the same name.

Building the Graph for `Collection`

The line number references in this section correspond to the line numbers of Listing 4.3. The `Collection` protocol is declared having the two associated types `Iterator` and `SubSequence` (lines 15 to 20). Figure 4.2 shows the associated type graph after the first two steps of the generation procedure are done.

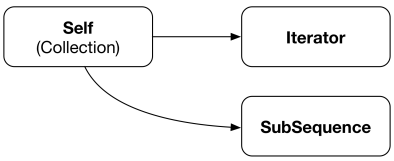

Figure 4.2.: *Graph for collection protocol after steps 1 and 2*

In step 3, the procedure starts recursively for the inherited protocol `_Indexable`. A self type node is created. Because `_Indexable` declares no associated types, the procedure skips step 2 and starts processing the only inherited protocol `_IndexableBase`. Figure 4.3 shows the associated type graph of protocol `_Indexable` after step 3 is complete.

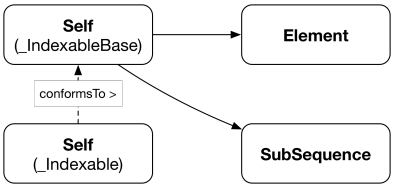

Figure 4.3.: *Graph for indexable protocol after steps 1 to 3*

Step 4 can be skipped for this graph, because the associated types of `_IndexableBase` do not inherit from other protocols directly. The graph is then normalized in step 5. The procedure resumes the processing of the associated type graph for `Collection` in step 3. The newly created graph of `_Indexable` is attached to the self type node of the current graph. Step 3 then continues with the processing of the inherited protocol `Sequence`.

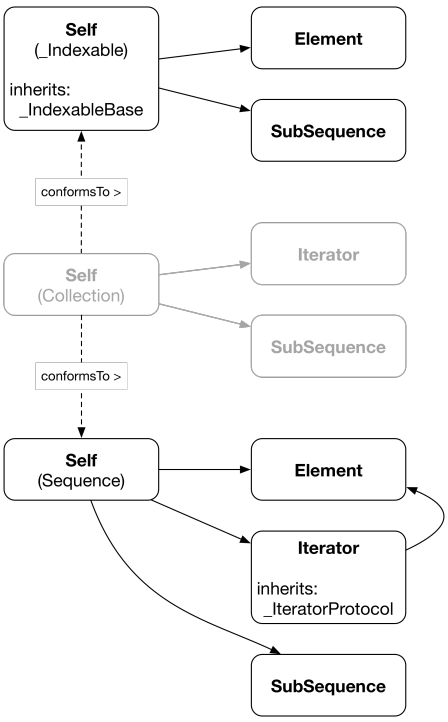

Figure 4.4.: *Graph for collection protocol after step 3*

Figure 4.4 shows, how the associated type graph of `Collection` looks after step 3. The graph components that existed after steps 1 and 2 are faded, to better

highlight the changes introduced in step 3. Figure 4.5 shows how the graph looks like after step 4 of the procedure is completed.

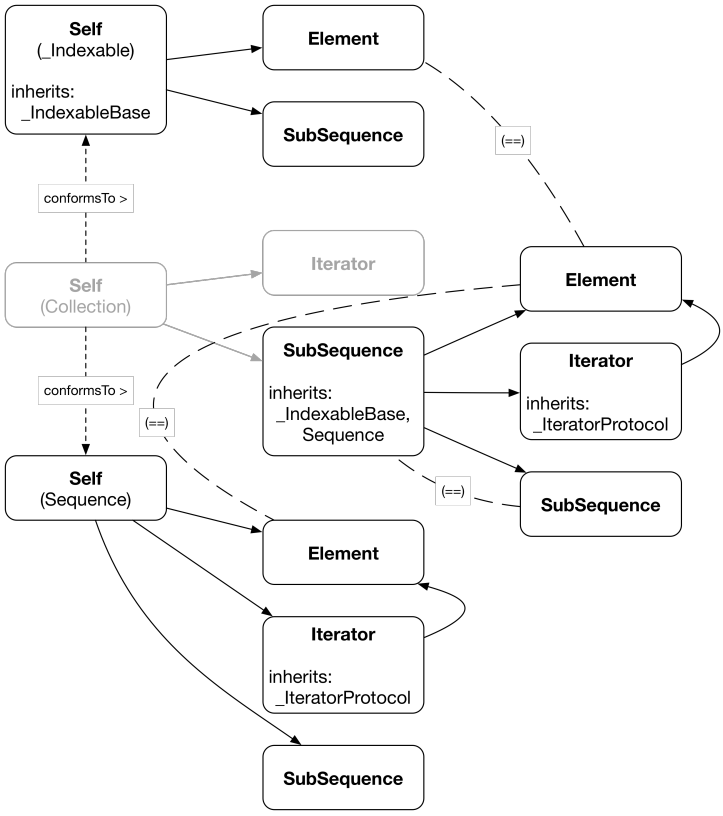

Figure 4.5.: *Graph for collection protocol after step 4*

Finally, the graph is normalized. The same type relation edges are removed and the nodes that they connected are merged. Conformances are integrated in the corresponding self type nodes. The normalized associated type graph for the protocol `Collection` is shown in Figure 4.6.

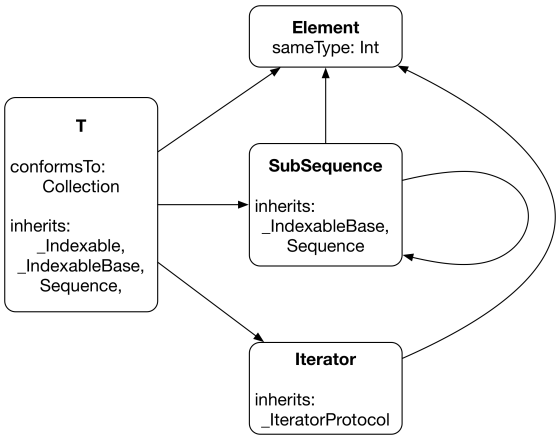

Figure 4.6.: *Normalized associated type graph*

Usage of the Associated Type Graph

Looking back to the declaration of the generic function `f()`, the goal is still to find the most specific type for the constants `x`, `y`, `z` and `v`. Because it is known that the generic type parameter `T` conforms to `Collection`, `T` can be set as the self type in this generic context. This means, that in the associated type graph of protocol, `T` can be set in the self type node. In addition, type `Int` can be set as the same type on the node `Element`, via the path `T.Iterator.Element`. The updated graph is shown in Figure 4.7.

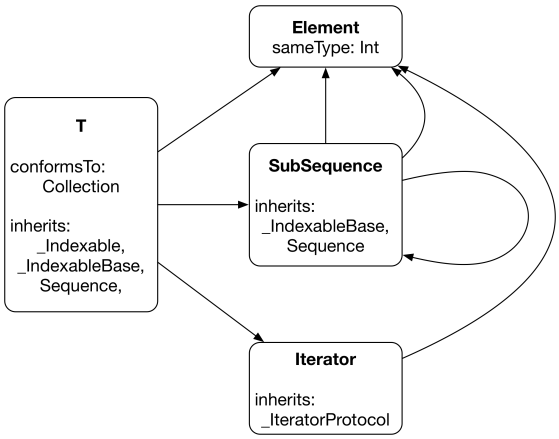

Figure 4.7.: *Associated type graph in function context*

To resolve the types of the variables of the example code, one must only follow the paths in the graph described in the variables' type annotation. For example, variable `y` is annotated with type `T.SubSequence.Element`. Following the path in the associated type graph leads to node `Element` and reveals the same type `Int`. The other variables can be resolved analogously.

Benefits

The example given above showed the ease of using an associated type graph. In addition, the associated type graph could be constructed during the type annotation pass of the indexer. Once a graph is completed, it has to be adjusted only when the described protocol or a protocol inherited by the described protocol changes or when Tifig is closed, which results in the discarding of the index. If Tifig should be able to persist the index in the future, this approach would be especially useful for protocols defined in the Swift standard library, because they change only with releases of Swift itself.

4.3.3. Preliminary Constraint Simplification

A problem that is sometimes encountered when writing large functional-style expressions or mathematical expressions using many operators, is the following compiler error:

```
Error:  expression was too complex to be solved in reasonable time;
consider breaking up the expression into distinct sub-expressions
```

Listing 4.4 shows a code example that results in the above error message. The example is taken from [23].

```
1  let a: Double = -(1 + 2) + -(3 + 4) + -(5)
```

Listing 4.4: *Error-generating code example*

On the first glance, this expression does not seem to be complicated to resolve. The problem is, that all the operators used in this example have a large amount of overloads. The unary minus operator has 15 overloads, while the binary plus operator has 45. For the example in the expression, this means that there are a combined $15 * 45 * 45 * 15 * 45 * 45 * 15 = 922'640'640$ possible overload choices. In this calculation, all constraints other than the disjunctions were neglected. The solution space would therefore be even larger.
The simplification mechanics in the Swift type checker are implemented to simplify constraints after each assumption made for a type variable. In his blog post [23],

Gallagher proposes an improvement of the type checker implementation that starts even earlier with the simplification of constraints, namely before any assumptions are made. Because type information can flow in two directions in Swift, propagating information through constraints must be done in two directions as well. Gallagher writes:

> *Since Swift constraints are bidirectional, the constraint path for each node starts at all of the leaves of the expression, traverses via the trunk and then traverses back along a linear path to the node.* [23]

By propagating information of other constraints, it is possible that certain overloads can be discarded. To demonstrate the effect, Gallagher gives and example, where he solves the constraint system of the code `let a: Double = 1 + -(2)` without making any assumptions. The propagation of information was enough to solve the constraint system in linear time.

Benefits

At this point, it is important to mention that Gallagher does not claim his solution to improve the type checker's performance. The approach is not proved formally or otherwise. However, his reasoning indicates potential in this optimization. There would still be cases, where the propagated information would not better the exponential time factor. However, in some cases, the approach should reduce the problem to logarithmic or even linear complexity. The detailed reasoning can be found in [23].

4.4. Summary

This chapter gave a brief insight into the characteristics of Swift's type system and the type checker. The type checker's performance was then briefly discussed, followed by an overview of currently implemented optimizations. Finally, two possible optimization mechanisms were presented, that could potentially be beneficial to the performance of the type checker.

5. Improving the Documentation

Starting out as a project thesis, Tifig was built from the ground up. The project was then continued as part of a master thesis. It grew rapidly, because the goal was to support all features of Swift 3.1. But because Tifig was a single person project, the documentation of the project was somewhat neglected. The technical report of the master thesis [30] was the only reference to the implementation. That is the reason why one of the goals of this thesis was to consolidate existing documentation and documentation relevant to the tasks executed as part of this project in a form that promotes continual development of Tifig, as was stated in Section 1.2.
This chapter describes the approach taken to better the overall documentation of the Tifig project. In Section 5.1, the selection process and study of large and successful open-source projects is documented. Sections 5.2 and 5.3 describe the implementation of the documentation improvements. Finally, Section 5.4 describes the continuous integration of the documentation in the development process.

5.1. Study of Open-Source Projects

Having a good documentation is a must in large projects, especially, if more than one person is involved. For open-source projects, providing the necessary documentation is substantial, because everyone can contribute, no matter their background and previous knowledge of the project. That is why large and successful open-source projects were regarded as fitting candidates for studying documentation practices.

5.1.1. Project Selection

The goal of the project selection was to find recent and successful open-source projects. For that, a series of blog posts, which list a ranking of best open-source projects over the course of a year, was consulted. In [18], the top ten projects of 2016 are credited. Projects of the year 2017 are ranked and listed in [18], [17] and [19]. Finally, [15] presents a ranking of best open-source software in 2018.
Adding up all the projects listed in the above posts, a total number of 37 open-source projects were selected for the study. In Section 5.1.3, the projects are listed by name.

5.1.2. Defining the Criteria

The projects under inspection were not rated. Also, the contents of the documentations were not read or reviewed. Instead, several ways for documentation and interaction with contributors were previously defined. Then, it was investigated, which of the predefined ways each open-source project makes use of. The forms of documentation used by the most projects would so be identified as promising. In the following, these forms of documentation are listed:

- Website
- Source Documentation
- Installation Guide
- Tutorial(s)
- Blog
- Forum
- Link to Source
- Wiki
- FAQ

Note, that source documentation means either some kind of source code reference, a description of how contributions can be done (developer tutorials) or both. Under the term tutorials falls everything that is related to instructions on how to use the end product, excluding installation guides, which are a criterion themselves.

5.1.3. Evaluation

Tables 5.1 and 5.2 show the findings for each of the 37 open-source projects. The bottom row of both tables shows the total number of occurences of each form of documentation over all projects.
Table 5.1 shows, that 35 out of 37 projects have a corresponding website. The only exceptions are the *Create React App* and *DSSTNE* projects, whose presence is limited to their GitHub repositories. Surprising to the author, only twelve projects provided documentation of their source code. The only documentation technique used by all 37 projects was the installation guide. User tutorials were part of 32 projects and can therefore be seen as important as well. The final result shown

in this first table is the amount of blogs. About two thirds of the projects have integrated a blog with news postings on their website.

Project	Website	Source Doc.	Install Guide	Tutorial	Blog
Atom	✓	✓	✓	✓	✓
Eclipse Che	✓		✓	✓	✓
FreeCAD	✓		✓	✓	
GnuCash	✓		✓	✓	✓
Kodi	✓		✓	✓	✓
MyCollab	✓		✓	✓	✓
OpenAPS	✓		✓	✓	✓
OpenHAB	✓		✓	✓	
OpenToonz	✓		✓	✓	
Roundcube	✓		✓	✓	✓
Yarn	✓		✓	✓	✓
Create React App			✓	✓	
Hyper	✓		✓	✓	
Parse	✓	✓	✓	✓	✓
Bulma	✓		✓	✓	✓
anime.js	✓		✓	✓	
Weex	✓	✓	✓	✓	✓
Sawtooth's Ledger	✓	✓	✓	✓	✓
CarbonData	✓		✓	✓	
DSSTNE			✓	✓	
Chord	✓		✓	✓	✓
Poseidon	✓		✓		
Trireme	✓	✓	✓		✓
Ansible	✓		✓	✓	✓
Kolibri	✓		✓	✓	✓
TensorFlow	✓	✓	✓	✓	✓
React Native	✓	✓	✓	✓	✓
Kubernetes	✓	✓	✓	✓	✓
Vault	✓		✓	✓	✓
LibreOffice	✓		✓	✓	
Gimp	✓	✓	✓	✓	✓
VLC	✓	✓	✓	✓	✓
Shotcut	✓		✓	✓	✓
Audacity	✓		✓	✓	
Firefox	✓	✓	✓		
Thunderbird	✓	✓	✓		
KeePass	✓		✓		✓
Total	**35**	**12**	**37**	**32**	**24**

Table 5.1.: *Open-source project overview part 1*

Table 5.2 reveals, that twelve projects make use of a forum. 30 out of 37 projects

have a link on their project website that navigates to the source code repository. 13 projects use a wiki for documentation and 16 projects answer frequently asked questions.

Project	**Forum**	**Link to Source**	**Wiki**	**FAQ**
Atom	✓	✓		✓
Eclipse Che		✓		
FreeCAD	✓		✓	
GnuCash		✓	✓	✓
Kodi	✓	✓	✓	
MyCollab	✓	✓		✓
OpenAPS		✓		✓
OpenHAB	✓	✓		
OpenToonz	✓	✓		✓
Roundcube	✓		✓	
Yarn		✓		
Create React App				
Hyper		✓		
Parse		✓	✓	
Bulma		✓		
anime.js		✓		
Weex		✓	✓	✓
Sawtooth's Ledger		✓		✓
CarbonData		✓		✓
DSSTNE				✓
Chord		✓	✓	
Poseidon	✓			
Trireme		✓		
Ansible		✓		
Kolibri	✓	✓		
TensorFlow		✓		
React Native		✓		
Kubernetes		✓		✓
Vault		✓		
LibreOffice		✓	✓	✓
Gimp		✓	✓	✓
VLC	✓	✓	✓	✓
Shotcut	✓			✓
Audacity		✓	✓	✓
Firefox		✓	✓	
Thunderbird		✓	✓	
KeePass	✓			✓
Total	**12**	**30**	**13**	**16**

Table 5.2.: *Open-source project overview part 2*

According to the study, the three most common practices for documentation are websites, installation guides and user tutorials. Again, the one outcome that was not expected by the author was the scarce use of source code documentation throughout all the projects.
The Tifig project already has a website including an installation guide, a blog and frequently asked questions. Therefore, it was decided to improve the current documentation with the following components:

- Source Documentation
- Tutorial(s)

The remainder of this chapter provides a more detailed reasoning for each component and documents their implementation.

5.2. Developer Documentation

According to the study conducted and described in Section 5.1, providing source code documentation or tutorials for developers is not common. Only a third of the open-source projects under inspection made use of one or both approaches.
Despite this outcome, it was decided that a reference guide and developer tutorials shall be implemented for the Tifig project. The reasons for this decision are listed in the following:

- **Project complexity:** Excluding tests, the Tifig IDE project consists of over 30'000 lines of code. Without a proper documentation, getting an initial overview as a new project member is therefore very time consuming.
- **Development environment setup:** Tifig is developed under the roof of the Institute for Software (IFS). The further development of Tifig in the form of future theses is not only likely, but already happened twice since the original master thesis was finished. The correct setup of the development environment is important, in order for contributions to be possible. The setup not only includes the installation of the Swift compiler and Eclipse, but also requires the uniform configuration of the code formatter settings.
- **Previous learnings:** During the course of this thesis, previous learnings were passed on to the author. A developer documentation lends itself as a platform where previous learnings can be documented for future project members.

It was further decided that the reference guide of Tifig's source code shall be part of a developer tutorial website that is hosted separately from the Tifig.net website. Tifig users should not be confused with information about Tifig's internals, which makes a physical separation of the developer documentation and user tutorials inevitable.

5.2.1. Implementation

During the evaluation of possible technologies, the focus was laid on the automatic generation of a reference guide using Javadoc annotations in the source code. A widely used and easy to use tool to achieve this is Sphinx [13]. Originally intended for the generation of reference guides for Python, extensions for other languages have been implemented. For Java, the Javasphinx extension [8] can be used. Sphinx also allows to extend the reference guide by custom documentation in the form of reStructuredText (a markup language) files.
After the initial setup, the documentation can be built using the `make html` command in the terminal. An excerpt of the contents of the `Makefile` is shown in Listing 5.1.

```
1  # Minimal makefile for Sphinx documentation
2  #
3
4  # You can set these variables from the command line.
5  SPHINXOPTS    =
6  SPHINXBUILD   = python -msphinx
7  SPHINXPROJ    = tifig_developer_doc
8  SOURCEDIR     = source
9  BUILDDIR      = build
10
11 # ...
12
13 source/javadoc:
14   @echo Javadoc
15   @javasphinx-apidoc -u -o source/javadoc/branding --title='Branding Plugin'
16       ../ch.hsr.ifs.tifig.branding/src/
17   @javasphinx-apidoc -u -o source/javadoc/core --title='Core Plugin'
18       ../ch.hsr.ifs.tifig.core/src/
19   # ...
20   @javasphinx-apidoc -u -o source/javadoc/ui --title='UI Plugin'
21       ../ch.hsr.ifs.tifig.ui/src/
22
23 clean:
24   rm -rf source/javadoc
25   @$(SPHINXBUILD) -M clean "$(SOURCEDIR)" "$(BUILDDIR)" $(SPHINXOPTS) $(O)
26
27 # ...
```

Listing 5.1: *Makefile of developer documentation*

Usually, to build the reference guide from the Javadoc annotations in the source code, the `javasphinx-apidoc` command needs to be called for every project directory. To prevent this, the calls were integrated into the `Makefile` (lines 15 to 21). Using the `make html` command now triggers the rebuild of the reference guide for all listed plug-in projects. In addition, a build goal for cleaning up previously generated documentation was defined (lines 23 to 25).
The customization of the page content can be done in the `index.rst` file, which is also generated during the initial setup. Listing 5.2 shows a part of the file.

```
1  Tifig Developer Documentation
2  ****************************
3
4
5  This documentation page contains various tutorials that help you as a
6  developer working on the Tifig Swift IDE, the tifig.net website
7  (`link <https://www.tifig.net/>`_) or even this developer documentation.
8
9  |
10
11 .. toctree::
12    :maxdepth: 2
13    :caption: Developer Tutorials
14
15    tutorials/tifig
16    tutorials/swift
17    tutorials/tifig_net
18    tutorials/developer_documentation
19
20 |
21
22 .. toctree::
23    :maxdepth: 1
24    :caption: Reference Guide
25
26    javadoc/branding/packages
27    javadoc/core/packages
28    javadoc/pasta_core/packages
29    javadoc/pasta_tree/packages
30    javadoc/ui/packages
```

Listing 5.2: *Index file of developer documentation*

The file defines two tables of content. The first represents the custom developer tutorials written in the form of `.rst` files in the subdirectory `tutorials/` (lines 11 to 18). The second links the previously generated source code documentation (lines 22 to 30).
Because the documentation needs to be adjusted along with the source code of Tifig, the complete resources were added to the Tifig IDE project. The content of the developer tutorials can be found in Appendix A.

5.3. User Tutorials

Most of the studied projects support first-time users with short tutorials on how to get started using their software. This was not the case for the Tifig project. To embed the tutorials directly on the projects' websites, next to the installation guide, was the most common approach. Therefore, it was decided to extend the content of Tifig's website.
The tutorials should provide the user with information on how to start working with Tifig. For that, the topics shown below were selected to become part of a newly introduced *Getting Started* tab:

- Initial configuration of Tifig
- Setting up a first Swift project
- Managing dependencies using the Swift package manager

5.3.1. Implementation

The Tifig.net website is built using Jekyll, a static website generator for blogs [9]. Unfortunately, Jekyll is not built to generate multiple blogs for the same site. However, the blog like structure seemed fitting for user tutorials, because as with posts, it should be easy to introduce new tutorials and reorder them at will. Also, the blog posts for a Jekyll generated site are written in Markdown (a markup language) and simply put into the `_posts/` folder, a subdirectory of the website's root directory.
After a closer study of the Jekyll documentation it became clear, that hosting a second blog was not possible. However, one can define own collections of content, that can then be iterated over and rendered one by one. Listing 5.3 shows part of the `_config.yml` file, which contains the configuration of the Jekyll site.

```
1  # ...
2
3  # Build settings
4  markdown: kramdown
5  permalink: pretty
6  # ...
7  plugins_dir:
8    - octopress-autoprefixer
9
10 # ...
11
12 collections:
13   tutorials:
14     output: true
```

Listing 5.3: *Jekyll configuration*

The new collection was defined with the name `tutorials` (lines 12 to 14). The option `output: true` makes sure that for each item of the collection a static HTML file is created, so that it can later be linked to.
Each page follows the layout `default.html`, which includes `footer.html`, `head.html` and `header.html`. The latter of the three needed to be updated, because it contains the navigation element of the website. The final version of the header file is shown in Listing 5.4.

```
1  <header class="site-header">
2
3    <div class="wrapper">
4
5      <a class="site-title" href="{{ site.baseurl }}/">
6        <img src="/images/logo.png" class="site-logo">{{ site.title }}
7      </a>
8
9      <nav class="site-nav">
10       <div class="trigger">
11         <a class="page-link{% if page.nr == 0 %}
12             active{% endif %}" href="/">Blog</a>
13         <a class="rss-link" href="{{ "/feed.xml" | prepend: site.baseurl
14             | prepend: site.url }}"><i class="fa fa-rss-square fa-lg"
15             aria-hidden="true"></i></a>
16         <a class="page-link{% if page.nr == 1 %} active{% endif %}"
17             href="/download">Download</a>
18         <a class="page-link{% if page.nr == 2 %} active{% endif %}"
19             href="/tutorials">Getting Started</a>
20         <a class="page-link{% if page.nr == 3 %} active{% endif %}"
21             href="/faq">FAQ</a>
22       </div>
23     </nav>
24
25   </div>
26
27 </header>
```

Listing 5.4: *Header layout*

As can be observed (lines 18 to 19), a `page-link` was added to the component `nav` with the title *Getting Started*. This created a permalink to `/tutorials/`, but so far, no resources were defined. To change that, a new file `tutorials.html` was added to the site project. The content of this file is shown in Listing 5.5.

```
1  ---
2  layout: default
3  title: Getting Started
4  permalink: /tutorials/
5  nr: 2
6  sidebar: tutorials
7  ---
8
9  <div class="tutorial-nav">
```

```
10    <h2 class="post-title" itemprop="name headline">
11      <a class="post-link" href="/tutorials/">Overview</a>
12    </h2>
13    <ul>
14        {% for item in site.data.navigation.tutorials %}
15          <li><a href="{{ item.url }}">{{ item.title }}</a></li>
16        {% endfor %}
17    </ul>
18  </div>
19  {% assign tutorials = site.tutorials | sort: 'tutorial_index' %}
20  {% for post in tutorials %}
21    <article class="post" itemscope itemtype="http://schema.org/BlogPosting">
22      <header class="post-header">
23        <h2 class="post-title" itemprop="name headline">
24          <a class="post-link"
25              href="{{ post.url | prepend: site.baseurl }}">
26                {{ post.title }}
27          </a>
28        </h2>
29      </header>
30      <div class="post-content" itemprop="articleBody">
31        {{ post.content }}
32      </div>
33    </article>
34  {% endfor %}
```

Listing 5.5: *Tutorials subsite*

Note, that the code shown in the listing is not pure HTML. For example, the beginning of the file contains a so called front matter (lines 1 to 7), that has to be valid YAML. If a front matter is present, Jekyll treats the page as a special file. In the front matter, variables can be defined, which can then be used in the HTML code. In addition, the Liquid template language is used to loop over collections of files (here the collection tutorials) and access their contents for rendering (lines 19 to 32). When building the website, Jekyll translates these statements into static HTML.
The code presented in the above listing iterates over the files in the subdirectory `_tutorials/`, which contain the Markdown annotated text for the user tutorials. The content of the user tutorials can be found in Appendix B.

5.3.2. RSS Feed

During the study of Jekyll's documentation, it became clear that an RSS feed can be added to any Jekyll website with the insertion of one single line of code. In fact, Listing 5.4 in Section 5.3.1 already contained the code that adds an RSS feed link to the navigation of the Tifig.net website (lines 13 to 15). The link references the file `feed.xml`, which was automatically created during the site's setup procedure. Although the distribution of RSS feeds in open-source projects was not studied, it was decided to add it to the Tifig.net website.

5.4. Continuous Integration

A source code reference is only useful when it is maintained properly. Being able to automatically generate the reference guide, it made sense to integrate the generation into the build process of Tifig. For that, the Tifig repositories were moved to GitLab, which has it's own Continuous Integration (CI) environment. To make use of GitLab CI, a YAML file with the name `.gitlab-ci.yml` had to be added to the existing repository. Listing 5.6 shows how the build was configured for the Tifig project.

```
1  variables:
2    # ...
3
4  stages:
5  - build
6  - test
7  - build-sphinx
8  - build-sphinx-docker-image
9  - deploy-sphinx-docker-image
10
11 build:
12   # ...
13
14 test:
15   # ...
16
17 build-sphinx:
18   image: # ...
19   stage: build-sphinx
20   when: always
21   only:
22     - swift4@tonisuter/tifig
23   script:
24     - apt-get update
25     - apt-get -y upgrade
26     - apt-get install -y python-pip
27     - pip install git+https://github.com/bronto/javasphinx.git
28     - pip install git+https://github.com/rtfd/sphinx_rtd_theme.git
29     - pip install git+https://github.com/djungelorm/sphinx-tabs.git
30     - make -C ./doc html
31   artifacts:
32     paths:
33       - ./doc/build/html
34
35 build-sphinx-docker-image:
36   image: docker:latest
37   stage: build-sphinx-docker-image
38   when: always
39   only:
40     - swift4@tonisuter/tifig
41   script:
42     # ...
43
44 deploy-sphinx-docker-image:
45   image: # ...
46   stage: deploy-sphinx-docker-image
```

```
47    when: always
48    only:
49      - swift4@tonisuter/tifig
50    script:
51      # ...
52    environment:
53      # ...
```

Listing 5.6: *GitLab CI configuration*

The build can be divided into stages, each of which is executed if the previous one succeeds. The stages are first listed (lines 4 to 9) and then defined one by one. The build and test stages are used only for the Tifig IDE. The stage `build-sphinx` generates a new and updated version of the developer documentation website (lines 17 to 33). The next stage creates a docker image that is then pushed into GitLab's own docker image registry (lines 35 to 42). Finally, the docker image is deployed to the server, where the documentation is hosted and a restart of the container is triggered (lines 44 to 53). By using this approach, the documentation is not only up to date at any time, but checking out a previous commit of the project allows the local build of any state of the documentation.
The deployment of the Tifig.net website was configured for CI in a similar manner, because the deployment of a new version of the website had to be done manually prior to this thesis.

6. Results

This chapter summarizes and evaluates the results of this thesis according to the goals stated in Section 1.2. Results regarding the implementation of the proposals are gathered in Section 6.1. In Section 6.2, the results of the consolidation of Tifig's documentation are presented. Finally, the results of the study of Swift's type checker are discussed in Section 6.3.

6.1. Implementation of Proposals in Tifig

The implementation of the three proposals listed in Section 1.2 was documented in Chapter 3. The proposals for generic subscripts and class and subtype existentials were implemented completely and were thoroughly tested. The proposal for allowing generic where clauses to constrain associated types was only implemented partly, because of the limited time period of the thesis. The known limitations of this implementation were therefore listed in Section 3.5.
The majority of the implementations was integrated in a new alpha release of Tifig. The remaining changes will be integrated into the following release.

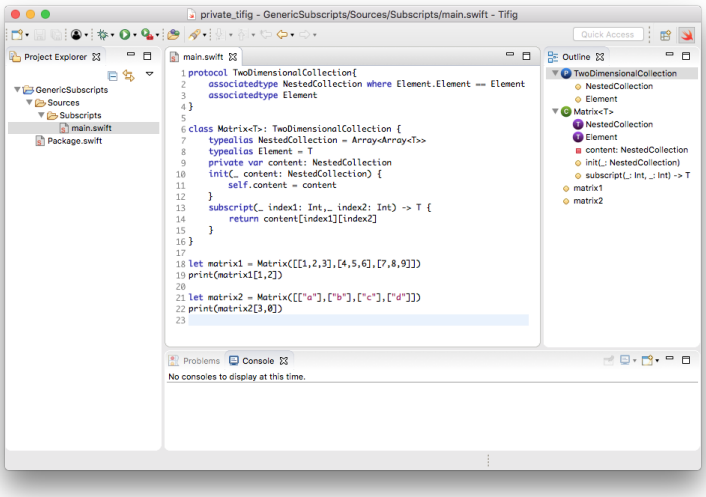

Figure 6.1.: *Tifig IDE Alpha Release 0.8.0*

6.2. Consolidation of the Documentation

The measures taken to improve the documentation of Tifig were described in Chapter 5. User tutorials for first-time Tifig users were integrated into the existing website `https://www.tifig.net`.

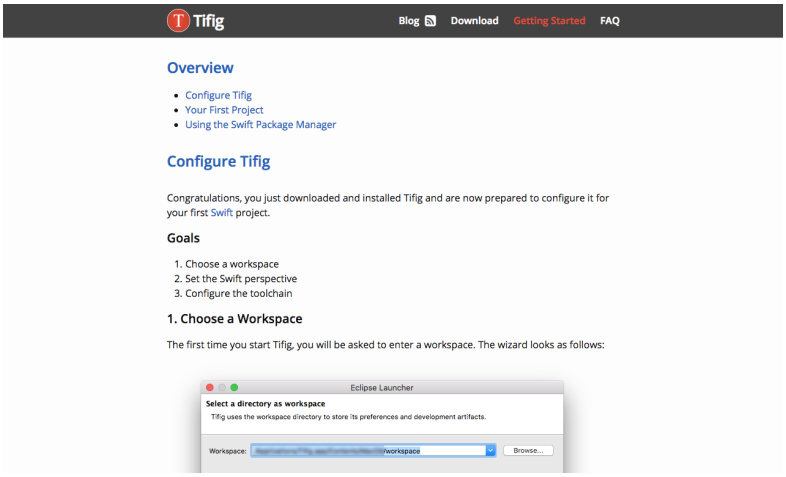

Figure 6.2.: *User tutorials on Tifig's website*

For contributors of the Tifig project, a separate documentation was created, describing the development environment and contribution mechanisms. An automatically generated reference guide for the source code is included. To avoid confusion, this documentation is hosted separately on `https://tifig.dev.ifs.hsr.ch`.

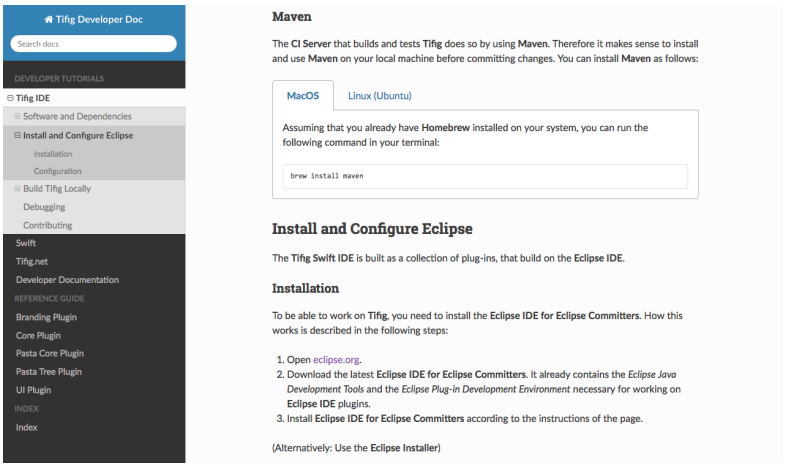

Figure 6.3.: *Developer tutorials with reference guide*

These forms of documentation were selected according to a study of open-source

projects described in Section 5.1. Finally, the setup for continuous integration of Tifig and the accompanying documentations was described in 5.4.

6.3. Study of Swift's Type Checker

The limitations described in Section 3.5 were intensively discussed during the course of this thesis and a theoretical solution for the underlying problem was developed. The approach was documented along with a brief discussion of Swift's type system and type checker in Chapter 4. The aim was to preserve the knowledge gained for the future development of Tifig.

7. Conclusion & Future Work

The adoption of Tifig's parser and indexer for supporting generic subscripts, class and subtype existentials and where clauses on associated types improved the usability of Tifig for developing Swift 4 projects. The few limitations of the final result were documented and a theoretical solution approach was presented. A study of Swift's type checker was done as an additional task in order to ease the access of future developers to the project.
The existing documentation of Tifig was consolidated in a form, that was first evaluated with a study on the documentation of multiple large and successful open-source projects. User tutorials were designed and implemented for first-time users of Tifig and integrated into the existing project's website. Developer tutorials including a reference guide of the source code were set up, in order to promote the continual development of Tifig.

7.1. Future Work

The development of an IDE is a continuous task. This is no different for Tifig. In the following, possible future tasks to extend and improve Tifig are listed:

- The limitations of the type inference of associated types are known. Therefore, improving the corresponding indexer implementation is the most pressing future task. This could likely be achieved by implementing associated type graphs.
- There are still features introduced in Swift 4 that have not been integrated into Tifig. The implementation of these features is essential for the overall usability of Tifig.
- Apple Inc. already provides a beta version of the Swift 4.1 compiler. New features introduced in Swift 4.1 can already be implemented for Tifig before the official release. The most impactful change will be the support of recursive constraints on associated types. Integrating this into the current Tifig infrastructure will require a large scale refactoring of the current indexer implementation.

- The recursive descent parser of Tifig has to make speculations when encountering certain sequences of tokens. Currently, backtracking results in parsing the same code multiple times. To improve the performance of the parser, transforming it into a memoizing parser is necessary.
- Because the semantic analysis of Swift code is still not complete, the implementation of refactorings was postponed so far. The addition of simple refactorings, such as automatically adding a closing brace when typing an opening one, would improve to overall user experience of Tifig.

Bibliography

[1] AppCode. https://www.jetbrains.com/objc/. Accessed: 2018-02-08.

[2] Clang Wikipedia. https://en.wikipedia.org/wiki/Clang. Accessed: 2018-01-27.

[3] CLion. https://www.jetbrains.com/clion/. Accessed: 2018-02-08.

[4] Covariance and contravariance Wikipedia. https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science). Accessed: 2018-01-28.

[5] Eclipse. http://www.eclipse.org/. Accessed: 2018-01-23.

[6] Eclipse IDE Plug-in Development: Plug-ins, Features, Update Sites and IDE Extensions. http://www.vogella.com/tutorials/EclipsePlugin/article.html. Accessed: 2018-01-24.

[7] Eclipse PDE. http://www.eclipse.org/pde/. Accessed: 2018-01-24.

[8] Javasphinx. https://github.com/bronto/javasphinx. Accessed: 2018-01-31.

[9] Jekyll. https://jekyllrb.com/. Accessed: 2018-01-31.

[10] Multiple Class Types in Protocol Composition. https://forums.swift.org/t/multiple-class-types-in-protocol-composition/7224. Accessed: 2018-01-28.

[11] Multiple Class Types in Protocol Composition Bug Report. https://bugs.swift.org/browse/SR-6561. Accessed: 2018-01-28.

[12] Nominal type system Wikipedia. https://en.wikipedia.org/wiki/Nominal_type_system. Accessed: 2018-02-06.

[13] Sphinx Python Documentation Generator. http://www.sphinx-doc.org/en/stable/. Accessed: 2018-01-31.

[14] Swift Lexicon. https://github.com/apple/swift/blob/master/docs/Lexicon.rst. Accessed: 2018-01-28.

[15] The best open source software 2018. http://www.techradar.com/news/the-best-open-source-software. Accessed: 2018-01-30.

[16] The Swift Programming Language. https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/. Accessed: 2018-01-07.

[17] The top 8 new open source projects. https://www.infoworld.com/article/3174151/open-source-tools/the-top-8-new-open-source-projects.html. Accessed: 2018-01-30.

[18] Top 10 open source projects of 2016. https://opensource.com/article/16/12/yearbook-top-10-open-source-projects. Accessed: 2018-01-30.

[19] Top 6 Open Source Projects In 2017. https://hackernoon.com/top-6-open-source-projects-in-2017-db34b9d034a2. Accessed: 2018-01-30.

[20] Type Checker Design and Implementation. https://github.com/apple/swift/blob/master/docs/TypeChecker.rst. Accessed: 2018-02-06.

[21] What's New in Xcode 9. https://developer.apple.com/xcode/. Accessed: 2018-02-08.

[22] C. Eidhof. Generic Subscripts. https://github.com/apple/swift-evolution/blob/master/proposals/0148-generic-subscripts.md.

[23] M. Gallagher. Exponential time complexity in the Swift type checker. https://www.cocoawithlove.com/blog/2016/07/12/type-checker-issues.html.

[24] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[25] D. Hart. Class and Subtype existentials. https://github.com/apple/swift-evolution/blob/master/proposals/0156-subclass-existentials.md.

[26] D. Hart. Permit where clauses to constrain associated types. https://github.com/apple/swift-evolution/blob/master/proposals/0142-associated-types-constraints.md.

[27] T. Kremenek. Swift 4.0 Released! https://swift.org/blog/swift-4-0-released/.

[28] S. Pestov. The secret life of types in Swift. https://medium.com/@slavapestov/the-secret-life-of-types-in-swift-ff83c3c000a5.

[29] B. Royal-Gordon. Throwing Properties and Subscripts. https://github.com/brentdax/swift-evolution/blob/throwing-properties/proposals/0000-throwing-properties.md.

[30] T. Suter. Tifig. 2017.

Appendix

A. Developer Documentation Content

This chapter contains the developer tutorials written for the `https://tifig.dev.ifs.hsr.ch` website. The content represents the state of the finished thesis and may be outdated quickly. Each section in this chapter corresponds to one developer tutorial. No further comments are provided. Note, that the website is not accessible outside the Hochschule für Technik Rapperswil (HSR) domain.

Tifig IDE

This section explains the necessary steps to make you ready for working on the **Tifig Swift IDE**. It leads you through the installation and configuration of your **Eclipse IDE**, which in turn depends on **Java**. If you have already installed some of the following software, you can go ahead and skip the installation step.

Software and Dependencies

Java

Java is required to run **Eclipse**. Also, the **Tifig Swift IDE** is programmed using Java. The minimum version required is **Java 8**.
To get **Java** for your system, follow the steps below:

- MacOS:
  1. Open `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.
  2. Download the latest **JDK** installer for your operating system.
  3. Install **JDK** according to the instructions of the page.
- Linux (Ubuntu):
  1. Run the following command in your terminal:

```
1       apt-get install default-jdk
2
```

Listing A.1: *Install Java*

2. Verify the installed **JDK** version by running:

```
1       java -version
2
```

Listing A.2: *Verify Java version*

Swift

Although **Tifig** parses and indexes the **Swift** code by itself, compilation is done by using the **Apple Swift Compiler**. To be able to compile and run code in your **Tifig Swift IDE**, you have to install the compiler.

- MacOS: There are two alternative ways to get the **Swift Compiler**.
  1. *Either*: Install the newest version of **Xcode** from the **App Store**. Swift will be installed along with Xcode.
  2. *Or*: Download and install from `https://swift.org/download/`. The installation instructions can be found on the page.
- Linux (Ubuntu):
  1. Download and install from `https://swift.org/download/`. The installation instructions can be found on the page.
  2. Add *swift* to your *PATH* variable in your *.profile* file, to make the compiler available from every directory.

Maven

The **CI Server** that builds and tests **Tifig** does so by using **Maven**. Therefore it makes sense to install and use **Maven** on your local machine before committing changes. You can install **Maven** as follows:

- MacOS: Assuming that you already have **Homebrew** installed on your system, you can run the following command in your terminal:

```
brew install maven
```

Listing A.3: *Install Maven MacOS*

- Linux (Ubuntu): Run the following command in your terminal:

```
apt-get install maven
```

Listing A.4: *Install Maven Linux*

Install and Configure Eclipse

The **Tifig Swift IDE** is built as a collection of plug-ins, that build on the **Eclipse IDE**.

Installation

To be able to work on **Tifig**, you need to install the **Eclipse IDE for Eclipse Committers**. How this works is described in the following steps:

1. Open `http://www.eclipse.org/downloads/eclipse-packages/`.
2. Download the latest **Eclipse IDE for Eclipse Committers**. It already contains the *Eclipse Java Development Tools* and the *Eclipse Plug-in Development Environment* necessary for working on **Eclipse IDE** plugins.
3. Install **Eclipse IDE for Eclipse Committers** according to the instructions of the page.

(Alternatively: Use the **Eclipse Installer**)

Configuration

To prevent different coding styles in the **Tifig IDE** project, the developers have agreed on certain save actions for automatic code formatting.

- In your **Eclipse IDE** preferences, choose *Java* > *Editor* > *Save Actions* and configure as follows:

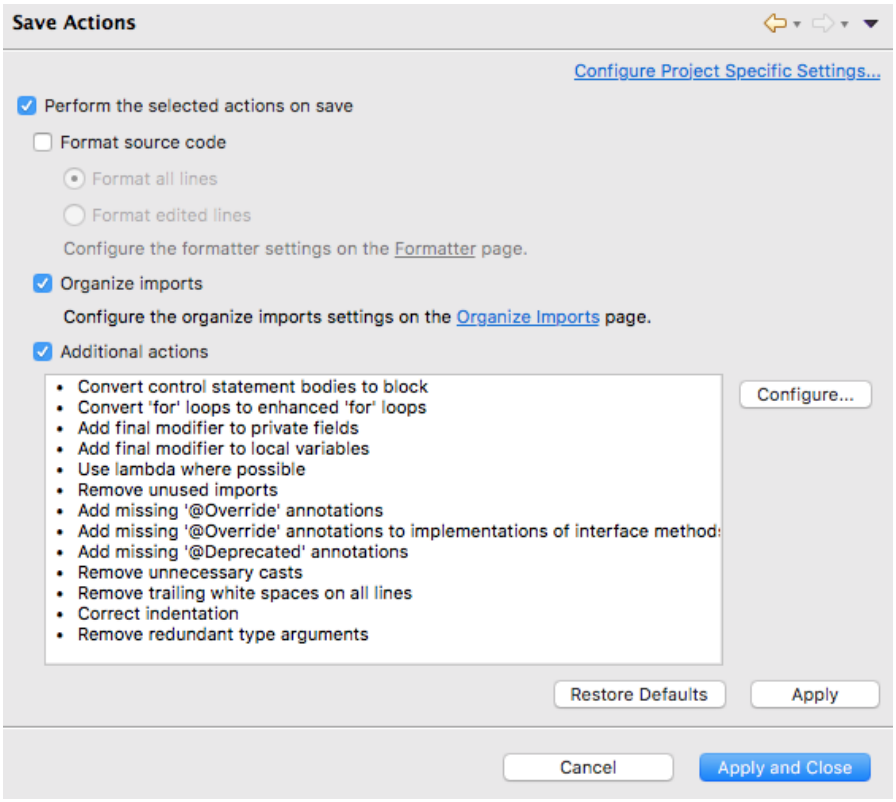

Figure A.1.: *Save actions*

Your additional actions window will most likely look different than the one on this screenshot. To configure the additional actions, press the *Configure...* button on the right of the window.

- In the tab *Code Organizing*, configure the following:

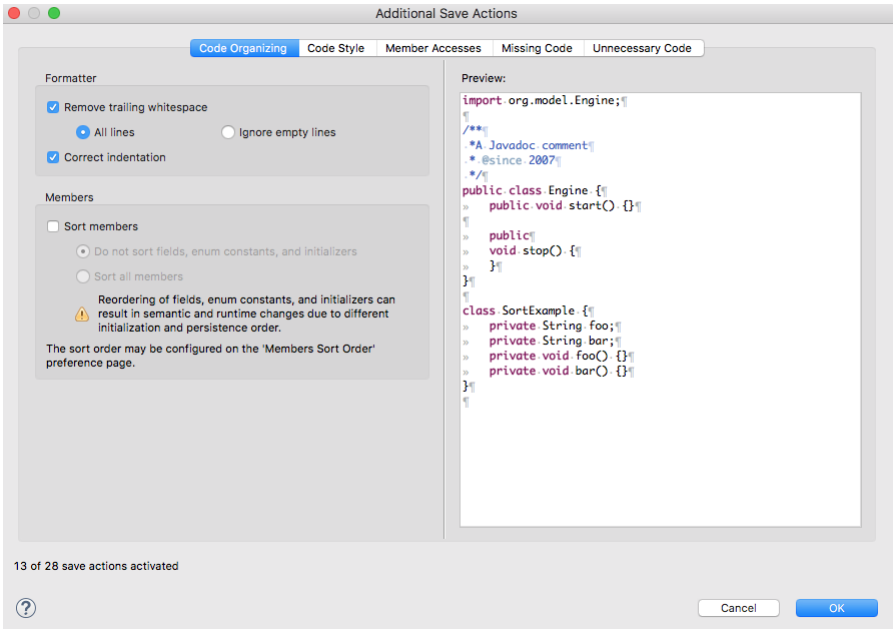

Figure A.2.: *Code organizing*

- In the tab *Code Style*, configure the following:

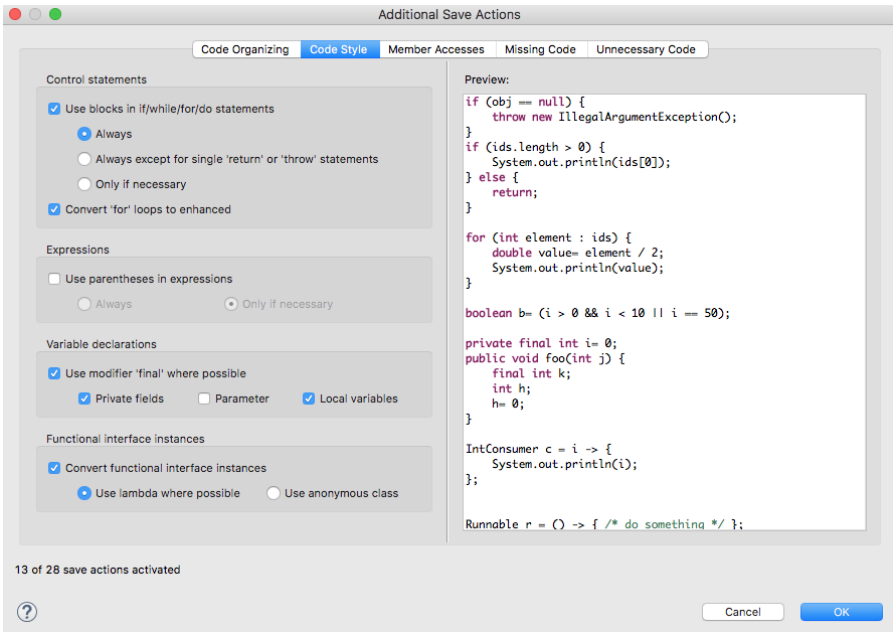

Figure A.3.: *Code style*

- In the tab *Member Accesses*, configure the following:

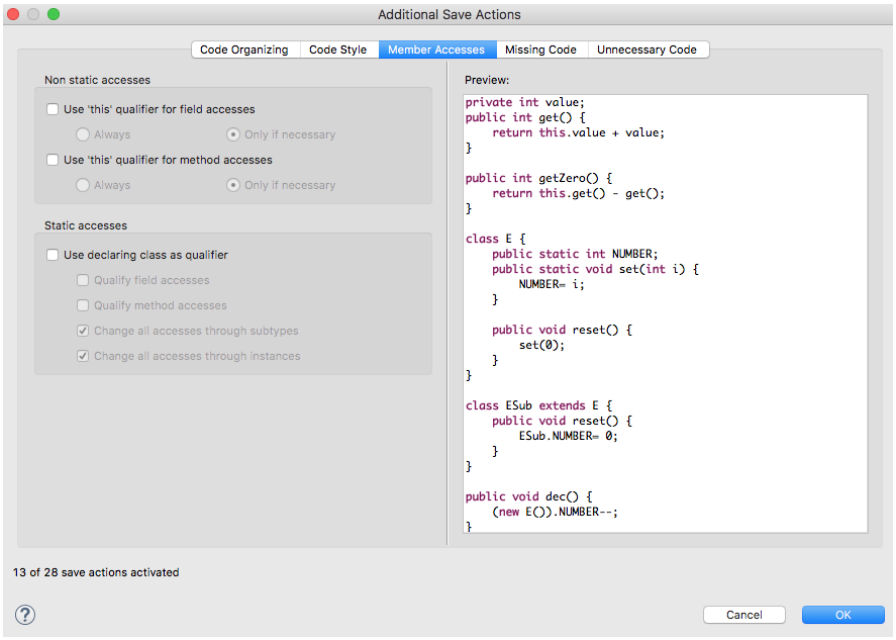

Figure A.4.: *Member accesses*

- In the tab *Missing Code*, configure the following:

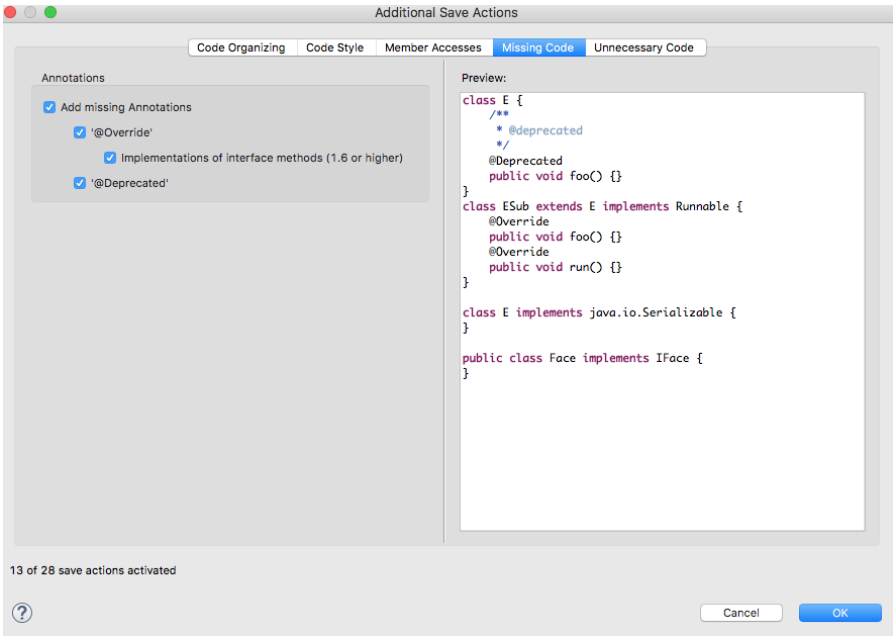

Figure A.5.: *Missing code*

- In the tab *Unnecessary Code*, configure the following:

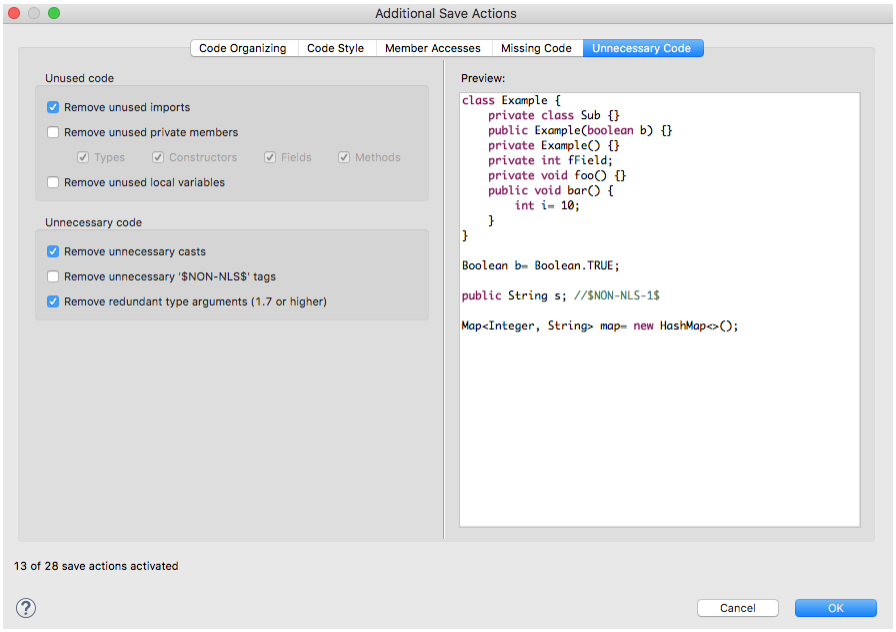

Figure A.6.: *Unnecessary code*

- After you have configured everything successfully, press the *OK* button on the bottom right to close the additional save actions.
- Back on the save actions window, press the *Apply and Close* button.

You are now ready to build the **Tifig Swift IDE** locally.

Build Tifig Locally

To build the **Tifig Swift IDE** locally, follow the steps below:

1. Fork the **Tifig** repository `https://gitlab.dev.ifs.hsr.ch/tonisuter/tifig`.
2. Clone your forked repository.
3. Launch your **Eclipse IDE for Eclipse Committers** and create a new workspace.
4. Go to *File* > *Import* > *General* > *Existing Projects into Workspace* and set the root directory to the location of your clone of the **Tifig** repository.

5. Make sure that all projects are selected and click *Finish* to import the projects into your workspace.

6. Open the file *ch.hsr.tifig.target/ch.hsr.tifig.target.target* and click on *Set as Target Platform*. This might take a while, because **Eclipse** will download the dependencies specified in the target file.

From here, you have two alternative ways of building **Tifig**.

Using a Run Cofiguration

1. Create a new *Eclipse Application* run configuration.

2. Under *Program to Run > Run a product:* choose *ch.hsr.ifs.tifig.branding.product*.

3. Press *Apply*.

4. Run the newly created run configuration to launch your local copy of the **Tifig Swift IDE**.

Using the Terminal

1. In the terminal, change your directory to the root of your cloned repository.

2. Run the following command:

```
1    mvn clean verify
2
```

Listing A.5: *Build with Maven*

3. If the build succeeded, the application bundles can be found in the folder *./ch.hsr.tifig.product/target/products*.

Debugging

The steps above finish the setup and configuration of your **Tifig Swift IDE** development environment.
When debugging, there is one helpful trick, to avoid you a lot of unecessary pain.

- To avoid triggering indexer breakpoints for the manifest file, it is helpful to define a trigger breakpoint on the following line in *Indexer.java*:

```
1    final List<SwiftFile> files =
2        pkg.streamFiles().collect(Collectors.toList());
3
```

Listing A.6: *Build with Maven*

Contributing

- Make your changes.
- Create a merge request on the **Tifig** repository `https://gitlab.dev.ifs.hsr.ch/tonisuter/tifig`.
- When the merge request is completed, **Tifig** will be built and tested automatically via **GitLab CI**.

That's all! Enjoy working on **Tifig**!

Swift

Some parts of the **Tifig Swift IDE** are built closely to the **Swift** parser and type checker of Apple's corresponding implementation. This is possible, because Apple open sourced these implementations.
For debugging purposes, it can therefore sometimes be helpful to debug certain code snippets using the actual implementations to compare with the behaviour of **Tifig**.

> *Note*: Unfortunately, this only works if you are developing on a **Mac**!

If you do, you can follow the instructions in this section to prepare your **Mac** for debugging the original **Swift** implementations.

Software and Dependencies

CMake

- **CMake** is needed to build the source code of **Swift**.
- To install **CMake** on your system, you can run the following command in your terminal (**Homebrew** needed):

```
brew install cmake
```

Listing A.7: *Install CMake*

Source Preparation

Follow the steps below to prepare the source code of **Swift**:

1. Go to the **Swift** repository `https://github.com/apple/swift`.
2. Scroll to the *README.md*.
3. Make sure your system fulfils the listed system requirements.
4. Complete the following commands in the terminal:

```
mkdir swift-source

cd swift-source

git clone https://github.com/apple/swift.git

./swift/utils/update-checkout --clone

cd swift

utils/build-script --release-debuginfo --debug-swift --xcode
# Swift frontend built in debug

```

Listing A.8: *Install Swift*

Install and Configure Xcode

Installation

Install the newest **Xcode** version from the **App Store**.

Configuration

To be able to debug **Swift** in Xcode, follow the steps below:

1. Open *Swift.xcodeproj*.
2. Choose *Automatically Create Schemes*.

3. Wait until the indexer is done (this takes some time).
4. Choose *swift* as the build scheme and *My Mac* as the target.
5. In the *Scheme* dropdown menu, scroll to the bottom and choose *Edit Scheme*.
6. In section *Run*, switch to the tab *Arguments*.
7. Add the following argument to *Arguments Passed On Launch*: */<somePatch>/<someFile>.swift*.

That's all! Enjoy debugging **Swift** in **Xcode**!

Tifig.net

Another essential part of developing the **Tifig Swift IDE** is to maintain and enrich the **Tifig.net** website `https://www.tifig.net/`. This section describes how you can run your local copy of the site in order to make contributions.
Before you can start, you will have to install some dependencies.

Software and Dependencies

Ruby

- **Ruby** is needed to build **Tifig.net**. More specifically, you need several **Ruby Gems**.
- To install **Ruby** and **Gems**, follow the instructions below:
- MacOS: Your **MacOS** comes with **Ruby** already installed. So you don't have to do anything in this step.
- Linux (Ubuntu): Run the following command in your terminal:

```
1    apt-get install ruby-full
2
```

Listing A.9: *Install Ruby*

JavaScript

- To install **JavaScript** on your system, do the following:
- MacOS: Run the following command in your terminal (**Homebrew** needed):

```
1    brew install node
2
```

Listing A.10: *Install JavaScript Mac*

- Linux (Ubuntu): Run the following command in your terminal:

```
1    apt-get install nodejs
2
```

Listing A.11: *Install JavaScript Linux*

Jekyll

- **Jekyll** is a static site generator used to build the **Tifig.net** website. Therefore, it has to be installed on your system to run a local copy thereof.
- To install **Jekyll**, run the following command in your terminal:

```
1    gem install jekyll
2
```

Listing A.12: *Install Jekyll*

Octopress Autoprefixer

- To automatically add CSS vendor prefixes to the generated **Jekyll** site, **Tifig.net** uses the **Octopress Autprefixer** gem.
- To install the **Octopress Autoprefixer**, run the following command in your terminal:

```
1    gem install octopress-autoprefixer
```

```
2
```

Listing A.13: *Install Octopress Autoprefixer*

Build Tifig.net Locally

To build and run the website locally on your device, follow the steps below:

1. Fork the **Tifig.net** repository `https://gitlab.dev.ifs.hsr.ch/tonisuter/tifig.net`.
2. Clone your forked repository to your device.
3. To start hosting the site locally, run the following bash command in the repository's root directory:

```
1    jekyll serve
2
```

Listing A.14: *Start hosting site locally*

4. Access the now running site under *http://localhost:4000/*.
5. To stop the server, press *Ctrl-C* when your terminal is in focus.

Contributing

- Make your changes.
- Create a merge request on the **Tifig.net** repository `https://gitlab.dev.ifs.hsr.ch/tonisuter/tifig.net`.
- When the merge request is completed, the website will automatically be updated on the production server via **GitLab CI**.

That's all! Enjoy working on **Tifig.net**!

Developer Documentation

Over time, it will become necessary to make changes to this very documentation. This section describes how you can do that. In addition, if you would like to create a documentation page for your plug-in, you can find a tutorial marked as optional. As usual, we will start with the installation of the necessary software dependencies.

Software and Dependencies

Python (+pip)

- The documentation is built using scripts written in **Python**.
- Also, some python packages are needed, which is why you have to install **pip**.
- Here is how you install **Python** and **pip**:
- MacOS: Run the following commands in your terminal (**Homebrew** needed):

```
1    brew install python
2
3    easy_install pip
4
```

Listing A.15: *Install Python Mac*

- Linux (Ubuntu): Run the following command in your terminal:

```
1    apt-get install python-pip python-dev build-essential
2
```

Listing A.16: *Install Python Linux*

Javasphinx

- To automatically create a reference guide from **Java** source code, **javasphinx** is required.
- To install the newest version of **javasphinx**, run the following command in your terminal:

```
pip install git+https://github.com/bronto/javasphinx.git
```

Listing A.17: *Install Javasphinx*

Read the Docs Theme

- To be able to contribute to this documentation, the **Read the Docs Theme** must be installed on your system.
- To install the newest version of the **Read the Docs Theme**, run the following command in your terminal:

```
pip install git+https://github.com/rtfd/sphinx_rtd_theme.git
```

Listing A.18: *Install Read the Docs Theme*

Sphinx Tabs

- As you can see below, the **Tifig Developer Documentation** uses beautiful tabs to separate instructions for different operating systems.
- To install the **Sphinx Tabs**, run the following command in your terminal:

```
pip install git+https://github.com/djungelorm/sphinx-tabs.git
```

Listing A.19: *Install Sphinx Tabs*

CMake

- **CMake** is needed to build the documentation website.
- To install **CMake** on your system, do the following:
- MacOS: Run the following command in your terminal (**Homebrew** needed):

```
brew install cmake
```

```
2
```

Listing A.20: *Install CMake Mac*

- Linux (Ubuntu): Run the following command in your terminal:

```
1    apt-get install cmake
2
```

Listing A.21: *Install CMake Linux*

Build the Documentation Locally

To build the **Tifig Developer Documentation** locally, follow the steps below:

1. Fork the **Tifig** repository `https://gitlab.dev.ifs.hsr.ch/tonisuter/tifig`.
2. Clone your forked repository.
3. Change directory in the terminal to the *doc/* folder in your cloned repository.
4. Run the following command in your terminal:

```
1    make html
2
```

Listing A.22: *Build the documentation*

5. The static doc pages are generated in the *build/html/* folder. The root page is called *index.html*.

Contributing

- Make your changes.
- Create a merge request on the **Tifig** repository `https://gitlab.dev.ifs.hsr.ch/tonisuter/tifig`.

- When the merge request is completed, **Tifig** will be built and tested automatically via **GitLab CI**. This includes a stage where the documentation is built and published on the developer doc page `https://tifig.dev.ifs.hsr.ch/`.

That's all! Enjoy working on the **Tifig Developer Documentation**!

(Optional) Create A New Sphinx Documentation

The setup of the documentation took some time because the technology was never used before by the developer team. Hoping that this form of documenting the software developed within the institute, it seemed logical to write down how the initial version of the developer documentation was set up. This assures a reproducible process.
Before you start with the following steps, make sure you have set up the dependencies necessary according to section A.

Doc Creation

Here is how you set up and configure a **Javasphinx** documentation page:

1. Run the following command in the terminal from the directory you want to create the documentation:

```
1    sphinx-quickstart
2
```

Listing A.23: *Create empty Sphinx documentation*

2. The installation process will start and you will be asked to enter configuration parameters one by one. Here is a list of the configuration chosen to build this documentation (<Enter> takes the default value shown in the square brackets):

Parameter Prompt	Chosen Option
Root path for the documentation [.]	<Enter>
Separate source and build directories (y/N) [n]	y
Name prefix for templates and static dir [_]	<Enter>
Project name	"PROJECT"
Author name(s)	"AUTHOR"
Project version	"X.X"
Project release [0.1]	"X.X.X"
Project language [en]	<Enter>
Source file suffix [.rst]	<Enter>
Name of your master document (without suffix) [index]	<Enter>
Do you want to use the epub builder (y/n) [n]	y
autodoc: automatically insert docstrings from modules (y/N) [n]	y
doctest: automatically test code snippets in doctest blocks (y/N) [n]	y
intersphinx: link between Sphinx documentation of different projects (y/N) [n]	<Enter>
todo: write "todo" entries that can be shown or hidden on build (y/N) [n]	<Enter>
coverage: checks for documentation coverage (y/N) [n]	<Enter>
imgmath: include math, rendered as PNG or SVG images (y/n) [n]	y
mathjax: include math, rendered in the browser by MathJax (y/n) [n]	y
ifconfig: conditional inclusion of content based on config values (y/N) [n]	<Enter>
viewcode: include links to the source code of documented Python objects (y/n) [n]	<Enter>
githubpages: create .nojekyll file to publish the document on GitHub pages (y/n) [n]	y
Create Makefile? (Y/n) [y]	<Enter>
Create Windows command file? (Y/n) [y]	n

Table A.1.: *Initial configuration*

After this initial configuration, a *Makefile* and two directories (*source/* and *build/*) have been created. The *source/* folder contains the files *conf.py* and *index.rts*.

Configuration

1. In the file *conf.py*, add **Javasphinx** and **Sphinx Tabs** to the list of extensions. The result should look like this:

```
extensions = ['sphinx.ext.autodoc',
  'sphinx.ext.doctest',
  'sphinx.ext.mathjax',
  'sphinx.ext.githubpages',
  'javasphinx',
  'sphinx_tabs.tabs']

```

Listing A.24: *Adding extensions to the conf.py file*

2. If you want references to be created to external docsets, you will have to add the *javadoc_url_map* option in *conf.py*. Note, that the following is only an example and has to be extended for every docset you want to have references to. Here is an example:

```
javadoc_url_map = {
  'org.osgi.framework' :
    ('https://osgi.org/javadoc/r6/core/', 'javadoc'),
  'org.eclipse.ui.plugin' :
    ('http://help.eclipse.org/oxygen/topic/org.eclipse.platform.doc
      .isv/reference/api/', 'javadoc8'),
  'org.eclipse.core.runtime' :
    ('http://help.eclipse.org/oxygen/topic/org.eclipse.platform.doc
      .isv/reference/api/', 'javadoc8')
}

```

Listing A.25: *Adding options to the conf.py file*

3. This documentation uses the **Read the Docs Sphinx Theme**. If you want to use it as well, you have to adjust your *conf.py* file by adding the following:

```
import sphinx_rtd_theme
html_theme = "sphinx_rtd_theme"
html_theme_path = [sphinx_rtd_theme.get_html_theme_path()]

html_theme_options = {
  'collapse_navigation': True,
  'display_version': False,
  'navigation_depth': 3,
}
```

```
10
```

Listing A.26: *Adding themes to the conf.py file*

4. The documentation will not build now. You have to remove the theme related settings for the (old) alabaster theme.

Automatic Doc Generation

Your documentation is now able to generate *.rst* files from your source code and use this to create a reference guide. But to achieve that, you have to run several commands every time you want to build the documentation. To prevent this inconvenience, you can adjust the *Makefile*:

```
1   # Minimal makefile for Sphinx documentation
2   #
3
4   # You can set these variables from the command line.
5   SPHINXOPTS    =
6   SPHINXBUILD   = python -msphinx
7   SPHINXPROJ    = "PROJECT"
8   SOURCEDIR     = source
9   BUILDDIR      = build
10
11  # Put it first so that "make" without argument is like "make help".
12  help:
13    @$(SPHINXBUILD) -M help "$(SOURCEDIR)" "$(BUILDDIR)" $(SPHINXOPTS) $(O)
14
15  source/javadoc:
16    @echo Javadoc
17    @javasphinx-apidoc -u -o source/javadoc/branding
18        --title='Branding Plugin' <SOURCE_PATH>.branding/src/
19    @javasphinx-apidoc -u -o source/javadoc/core
20        --title='Core Plugin' <SOURCE_PATH>.core/src/
21    @javasphinx-apidoc -u -o source/javadoc/core_tests
22        --title='Core Tests Plugin' <SOURCE_PATH>.core.tests/src/
23    @javasphinx-apidoc -u -o source/javadoc/ui
24        --title='UI Plugin' <SOURCE_PATH>.ui/src/
25    @javasphinx-apidoc -u -o source/javadoc/ui_tests
26        --title='UI Tests Plugin' <SOURCE_PATH>.ui.tests/src/
27
28  clean:
29    rm -rf source/javadoc
30    @$(SPHINXBUILD) -M clean "$(SOURCEDIR)" "$(BUILDDIR)" $(SPHINXOPTS) $(O)
31
32  .PHONY: help Makefile source/javadoc
33
34  # Catch-all target: route all unknown targets to Sphinx using the new
35  # "make mode" option.  $(O) is meant as a shortcut for $(SPHINXOPTS).
36  %: Makefile source/javadoc
37    @$(SPHINXBUILD) -M $@ "$(SOURCEDIR)" "$(BUILDDIR)" $(SPHINXOPTS) $(O)
```

Listing A.27: *Extending the Makefile*

You can now build your documentation as is described in section A.

(Optional) Helpful Resources

The following links are helpful regarding the different topics discussed in this section:

- First Steps with Sphinx (`http://www.sphinx-doc.org/en/stable/tutorial.html`)
- Invocation of apidoc (`http://www.sphinx-doc.org/en/stable/invocation.html#invocation-apidoc`)
- javasphinx User Guide (`http://bronto-javasphinx.readthedocs.io/en/latest/#javasphinx-apidoc`)
- Parsing javadoc with Sphinx (`https://stackoverflow.com/questions/14254527/python-parsing-javadoc-with-python-sphinx`)
- Sphinx RTD Theme Repository (`https://github.com/rtfd/sphinx_rtd_theme`)
- reStructuredText Primer (`http://www.sphinx-doc.org/en/stable/rest.html#rst-primer`)

If you find more helpful links, please feel free to contribute them to this documentation.

(Optional) Known Issues

Method References

The **javalang** python package (needed to parse the Java sources) cannot handle certain types of method references, e.g.:

```
1    @unittest.expectedFailure
2    def test_method_reference_explicit_type_arguments_for_generic_type(self):
3            """ currently there is no support for method references
4                for an explicit type.
5            """
6            self.assert_contains_method_reference_expression_in_m(
7                parse.parse(setup_java_class("List<String>::size;")))
8
9    @unittest.expectedFailure
10   def test_method_reference_from_array_type(self):
11           """ currently there is no support for method references
12               from a primary type.
```

```
13              """
14              self.assert_contains_method_reference_expression_in_m(
15                  parse.parse(setup_java_class("int[]::new;")))
```

Listing A.28: *Method reference tests of javalang*

See the complete issue on the **javalang** issue page `https://github.com/c2nes/javalang/blob/master/javalang/test/test_java_8_syntax.py#L198`.
There are two possible ways to solve this kind of issue:

1. Update the your installed **javalang** version using the following command in your terminal:

```
1    pip install --ignore-installed git+https://github.com/c2nes/javalang.git
2
```

Listing A.29: *Update javalang*

2. If this does not resolve the issue, change the source code to a Java construct that is supported by the current **javalang** version.

Outdated six Package

As with the **javalang** python package, some errors occur when the **six** python package is outdated. To install a newer version of **six**, run the following command in your terminal:

```
1    pip install --ignore-installed six
```

Listing A.30: *Update six*

B. User Tutorials Content

This chapter contains the user tutorials written for the `https://tifig.net` website. The content represents the state of the finished thesis and may be outdated quickly. Each section in this chapter corresponds to one user tutorial. No further comments are provided.

Configure Tifig

Congratulations, you just downloaded and installed Tifig and are now prepared to configure it for your first Swift project.

Goals

1. Choose a workspace
2. Set the Swift perspective
3. Configure the toolchain

1. Choose a Workspace

The first time you start Tifig, you will be asked to enter a workspace. The wizard looks as follows:

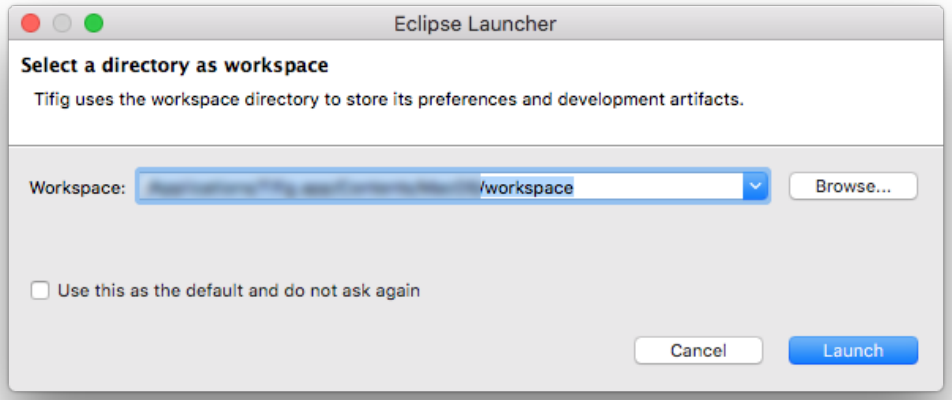

Figure B.1.: *Workspace selection*

You can either type or paste a path directly into the dropdown menu or search for a directory using the *Browse* button on the right.
If you don't want this wizard to show every time you start up Tifig, you can check the box at the bottom left of the wizard to set your new workspace as the default.
When you are done, click on the *Launch* button.

2. Set the Swift Perspective

After successfully setting the workspace, the splash screen with the Tifig logo will show. You can see in the progress bar (below the logo) how the various plug-ins of the IDE are loaded.
Because it is the first time you started Tifig, you will be greeted by the welcome screen:

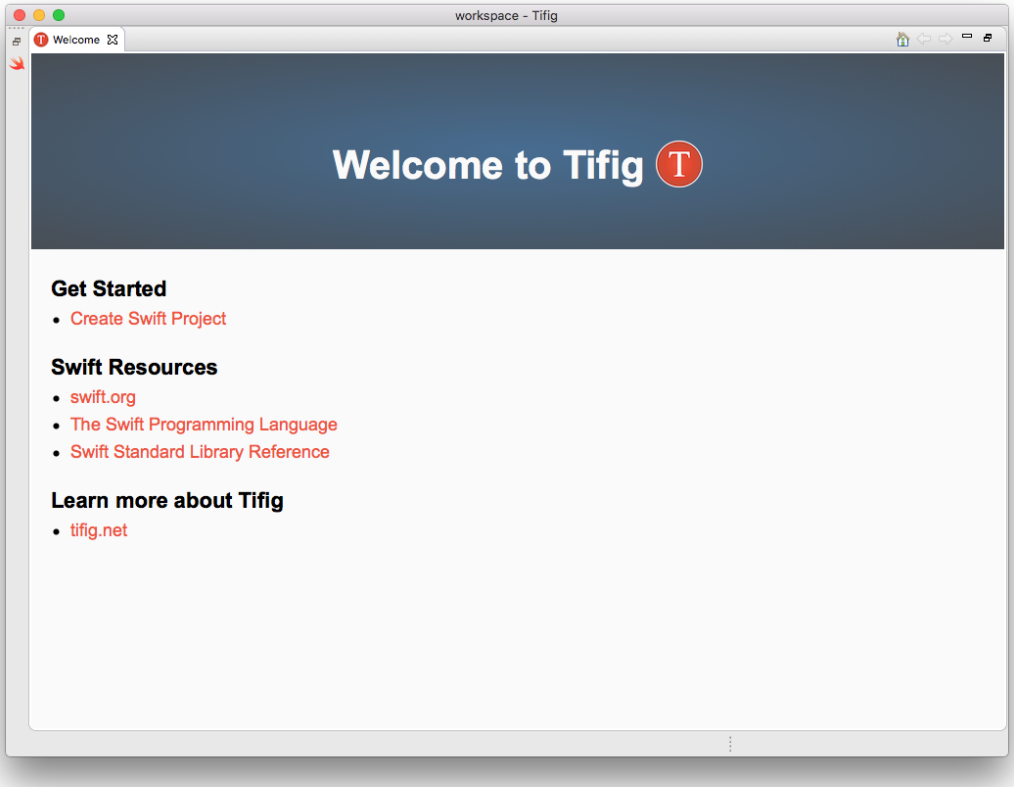

Figure B.2.: *Welcome screen*

From here you can directly create a new Swift project or learn more about Swift from various resources. In this tutorial, we will first make sure that your IDE is correctly configured and therefore, we recommend that you close the welcome screen tab by clicking the close symbol on the top left.
After doing so, the blank view of your newly created workspace will appear:

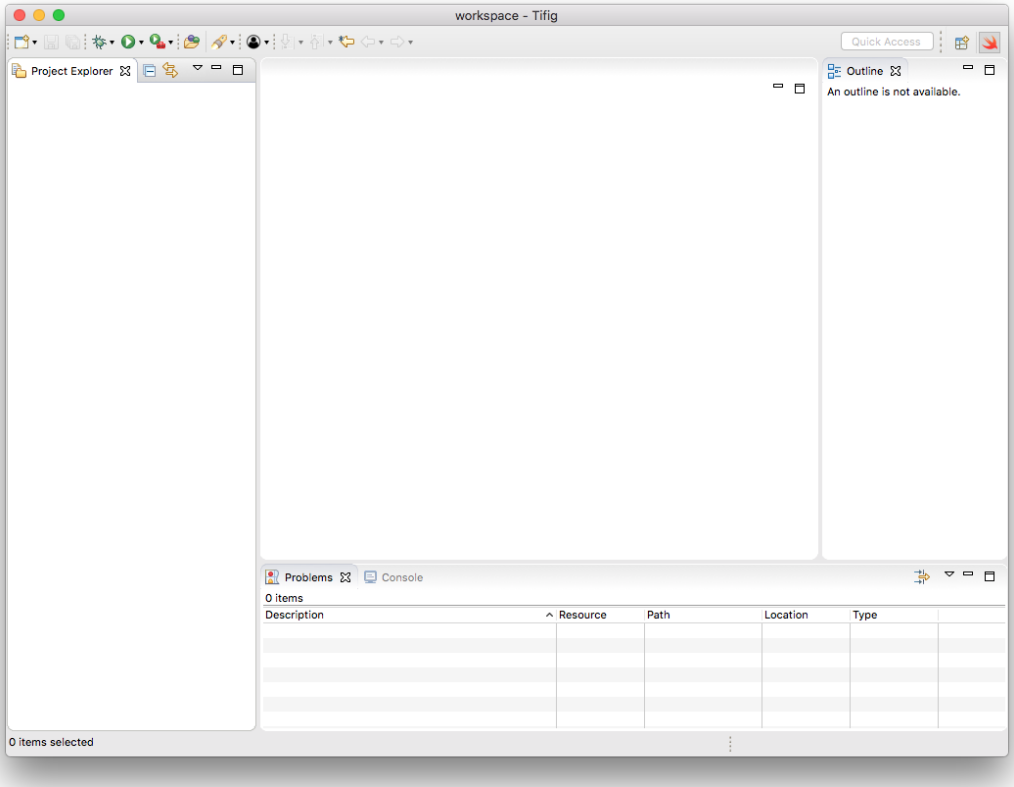

Figure B.3.: *Empty workspace*

Notice the Swift logo on the top right of the window? Clicking on it enables the Swift perspective in your IDE. Normally, the Swift perspective is set as the default and should be enabled already. If this is not the case, click on the logo to switch to the Swift perspective.

3. Configure the Toolchain

Tifig does not come with a built-in Swift installation. However, if you followed the installation instructions in the download section, Tifig should have set your Swift toolchain already. To verify this, open the *Preferences*.

> **Note:** The *Preferences* are not located in the same menus on a Mac and on a Linux system, respectively. On a Mac, the path is *Tifig/Preferences...*. On a Linux system, the path is *Window/Preferences*.

In the *Preferences*, choose *Swift Toolchains*. The wizard looks as follows:

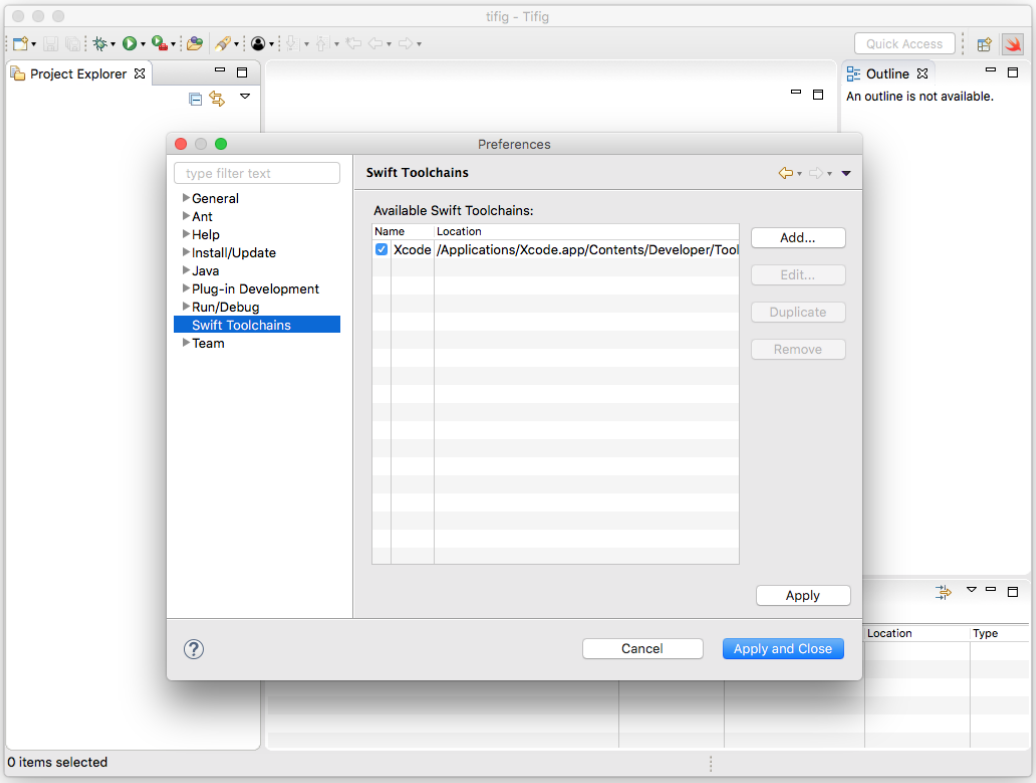

Figure B.4.: *Swift toolchains*

If you had a Swift installation before opening Tifig, an initial entry should be present and selected in the toolchain list. If this is not the case, you can add your preferred toolchain by using the *Add...* button on the right of the list. You can also edit, duplicate and remove existing entries.

Summary

You have successfully prepared Tifig for your first Swift project by choosing the Swift perspective and setting your Swift toolchain. To find out how to create, change and run your first Swift project, go to our next tutorial.

Your First Project

In this short tutorial, you will create and run your first Swift project using Tifig.

Goals

1. Create a project
2. Make changes
3. Run the project

1. Create a Project

There are several ways to open the *New Swift Project* wizard. Choose one of the following:

- Right-click in the *Project Explorer* window. Select *New/Swift Project*.
- Choose *File/New/Swift Project* from the main menu.
- On the welcome screen (see Configure Tifig), click on *Create Swift Project*.

After doing so, the wizard shown below opens:

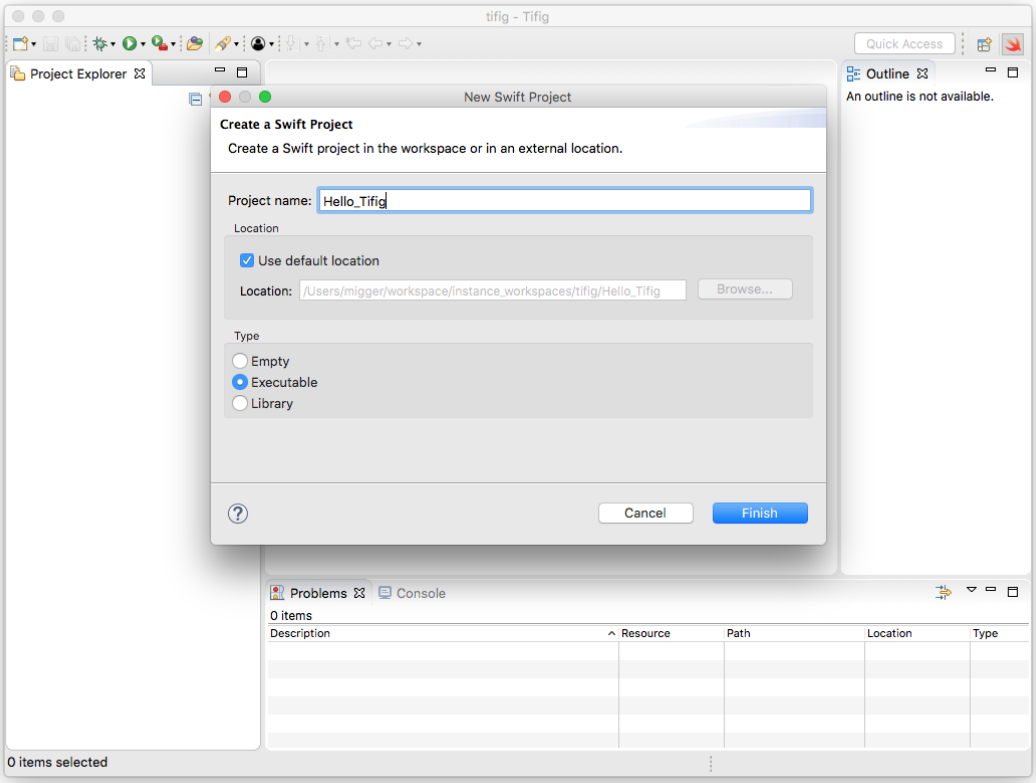

Figure B.5.: *New project wizard*

Here, you have to choose a name for your project. We chose *Hello_Tifig*, but you can choose whatever name you like.
The wizard has checked the box for using the default location, which creates a folder with your project name in your current workspace. If you want, you can change this by unchecking the box and providing an alternative location.
Tifig differentiates between three project types:

- Empty
- Executable
- Library

For our example, we will stick with the executable project.
Click *Finish* to return to the main window, which now looks as follows:

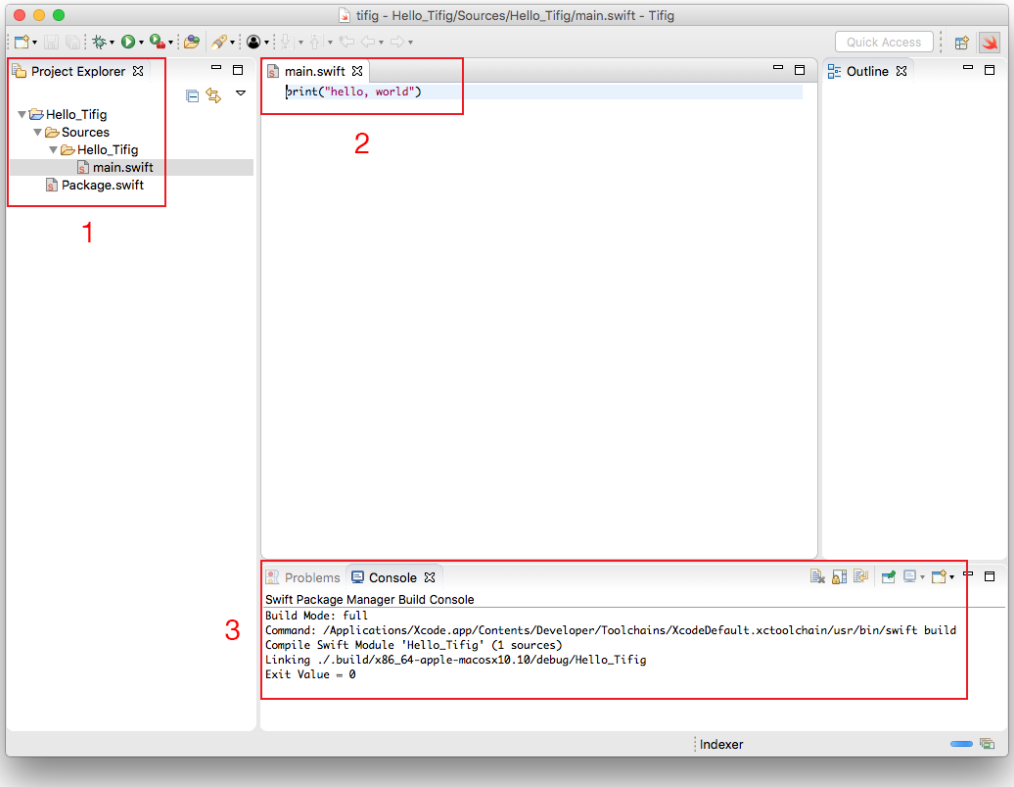

Figure B.6.: *Initial project*

As you can see, the parts that have changed are highlighted in red rectangles and numbered from 1 to 3. We will discuss each of the changes separately below.

1. Project File Structure

If you have a closer look at the *Project Explorer* window, you can see that Tifig has created some files and folders for you.

```
.
└── Hello_Tifig
    ├── Package.swift
    └── Sources
        └── Hello_Tifig
            └── main.swift
```

Figure B.7.: *File structure*

The main folder has the name of your project. In our case, this is *Hello_Tifig*. In it, a *Sources/Hello_Tifig* folder has been created, containing the *main.swift* file. This file will be our main focus in this tutorial.
In addition, your project folder contains a *Package.swift* file. This file will be further discussed in the Using the Swift Package Manager tutorial.

2. The main.swift File

The name of this file is not set by accident. The *main.swift* file is used as the entry point of your entire application. By default, the newly created file has the following content:

```
1 print("hello, world")
```

Listing B.1: *main.swift*

As you probably would have guessed, this application prints *hello, world* to your console when being executed.

3. Console Output

Having clicked on *Finish* in the *New Swift Project* wizard, Tifig has already started working for you. On the bottom, you can see the output of the Swift compiler. If you have configured your toolchain correctly, you should get something similar to the following:

```
1 Build Mode: auto
2 Command: /Applications/Xcode.app/Contents/Developer/Toolchains/
3     XcodeDefault.xctoolchain/usr/bin/swift build
4 Compile Swift Module 'Hello_Tifig' (1 sources)
5 Linking ./.build/x86_64-apple-macosx10.10/debug/Hello_Tifig
6 Exit Value = 0
```

Listing B.2: *Compiler output*

If your console does not show this, it is possible that the indexer is still running. You can check that by looking at the bottom right corner of Tifig. If the indexer is still running, you will see the word *Indexer* and a blue progress indicator (see screenshot above).

2. Make Changes

You can make changes to your files by just typing them in the *main.swift* window. Saving your changes will trigger the Swift compiler to start a new build, which you can see in your console output. In our example, we replaced `print("hello, world")` with `print("Hello Tifig")`, matching our project name. Tifig now looks as follows:

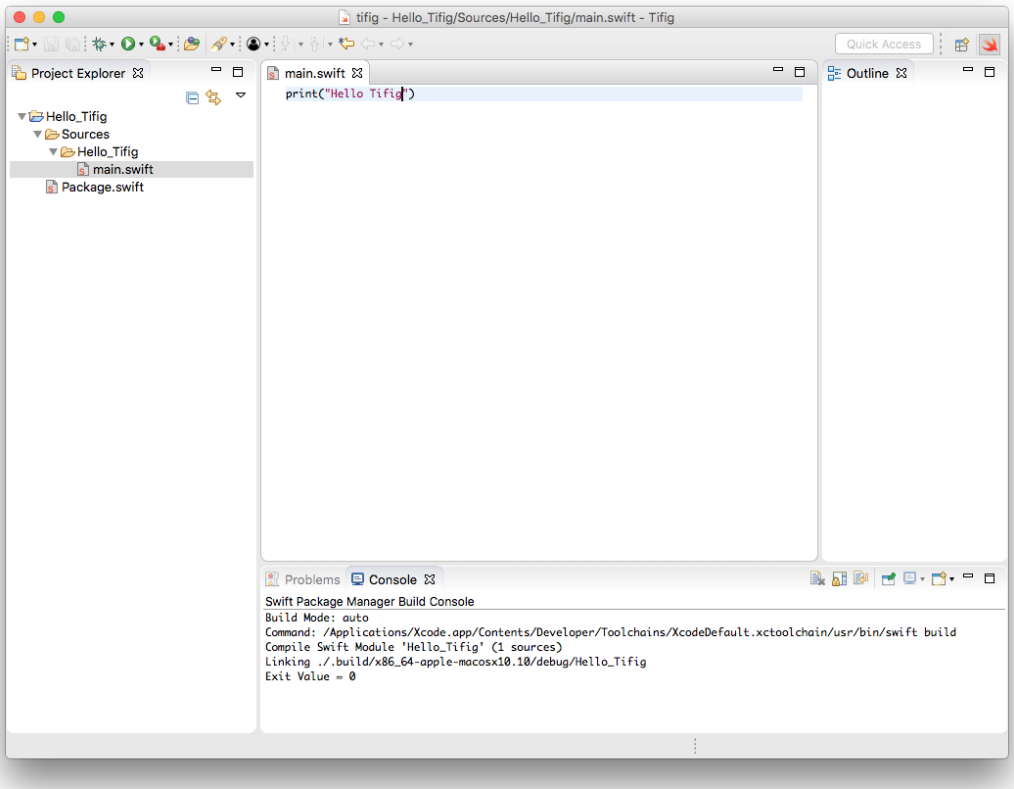

Figure B.8.: *Changed project*

3. Run the Project

To run your project, you can do one of the following:

- Click on the green *Run* Button in the toolbar on the top.
- Choose *Run/Run* from the main menu.

If everything works out, the console output should display *Hello Tifig*, as in the screenshot below:

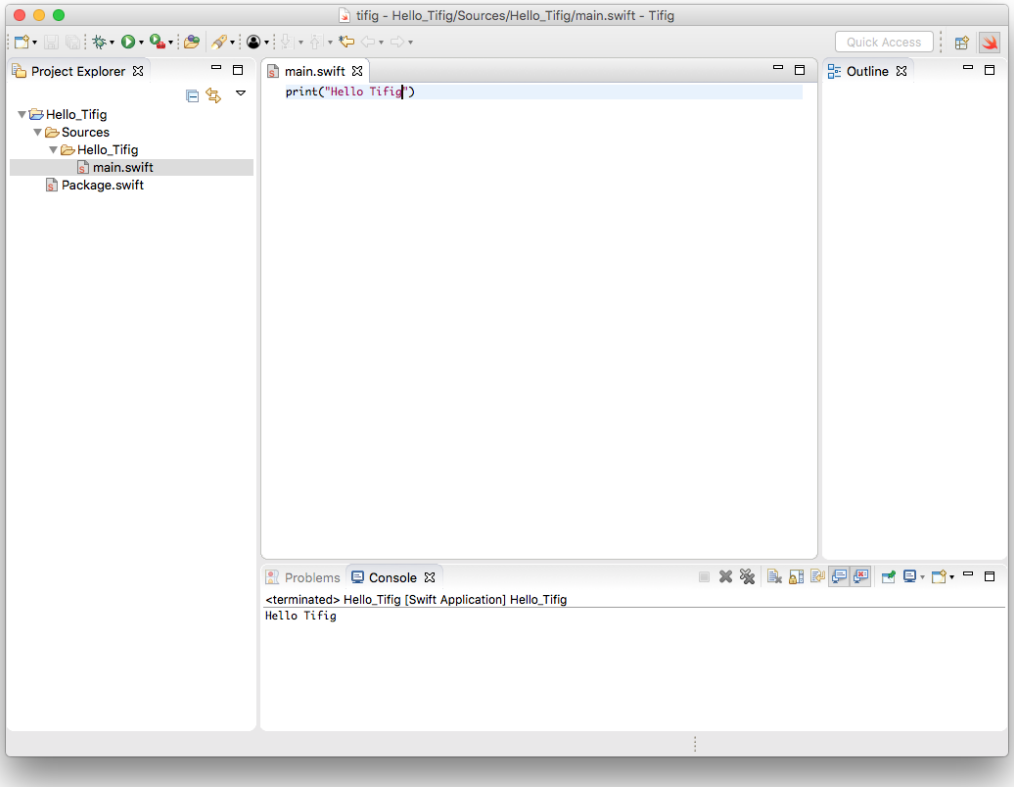

Figure B.9.: *Running project*

Summary

Congratulations, you have just created, changed and run your first Swift project in Tifig. Of course, this example shows the most simple project, having no other dependencies than the Swift standard library and only consisting of a single file. To find out how you can manage dependencies using the Swift *Package Manager*, go to our next tutorial.

Using the Swift Package Manager

In this tutorial, you will learn how to manage dependencies using the Swift *Package Manager*. For this, we will create a small application that uses a predefined package. If you want to look at a more challenging example, you can visit the swift.org documentation.

Goals

1. Add a dependency to your project
2. Create a small application that uses the dependency

1. Add a Dependency to Your Project

Tifig already prepared your project for the inclusion of packages. Open the *Package.swift* file from your existing *Hello_Tifig* project by double-clicking it in the *Project Explorer* window. The following was already created for you:

```
// swift-tools-version:4.0
// The swift-tools-version declares the minimum version of Swift required to
// build this package.

import PackageDescription

let package = Package(
    name: "Hello_Tifig",
    dependencies: [
        // Dependencies declare other packages that this package depends on.
        // .package(url: /* package url */, from: "1.0.0"),
    ],
    targets: [
        // Targets are the basic building blocks of a package. A target can
        // define a module or a test suite.
        // Targets can depend on other targets in this package, and on
        // products in packages which this package depends on.
        .target(
            name: "Hello_Tifig",
            dependencies: []
        )
    ]
)
```

Listing B.3: *Package.swift*

The library you are going to use is located on GitHub. It defines the following two public functions:

```
public func sayHello() {
    print("Hello Tifig")
}

public func sayGoodBye() {
    print("Goodbye Tifig")
}
```

Listing B.4: *Library functions*

To use these two functions, you have to make some adjustments to your *Package.swift* file:

- In the dependencies section, add the following line:

```
.package(url: "https://github.com/pipeaesac/Hello_Tifig_Library.git",
    from: "1.0.0"),
```

Listing B.5: *Library dependency*

  This tells the package manager where the dependency is located and which versions are allowed to be used with your application.

- In the targets section, add `"Hello_Tifig_Library"` to the already existing but so far empty target dependency list.

  Your resulting *Package.swift* file should look as follows:

```
// swift-tools-version:4.0
// The swift-tools-version declares the minimum version of Swift
// required to build this package.

import PackageDescription

let package = Package(
    name: "Hello_Tifig",
    dependencies: [
        // Dependencies declare other packages that this package
        // depends on.
        // .package(url: /* package url */, from: "1.0.0"),
        .package(url: "https://github.com/pipeaesac/
            Hello_Tifig_Library.git", from: "1.0.0"),
    ],
    targets: [
        // Targets are the basic building blocks of a package.
        // A target can define a module or a test suite.
        // Targets can depend on other targets in this package,
        // and on products in packages which this package depends on.
        .target(
            name: "Hello_Tifig",
            dependencies: ["Hello_Tifig_Library"]
        )
    ]
)
```

Listing B.6: *Changed Package.swift file*

That is all it takes. The *Hello_Tifig_Library* is now ready to be used in your application.

2. Create a Small Application That Uses The Dependency

To use the newly included library, replace the existing code in your *main.swift* file with the following:

```
1  import Hello_Tifig_Library
2
3  sayHello()
4  sayGoodBye()
```

Listing B.7: *Changed main.swift file*

When saving, the dependency will be downloaded and a file called *Package.resolved* will be created. It contains a description of your resolved dependency:

```
1  {
2    "object": {
3      "pins": [
4        {
5          "package": "Hello_Tifig_Library",
6          "repositoryURL":
7              "https://github.com/pipeaesac/Hello_Tifig_Library.git",
8          "state": {
9            "branch": null,
10           "revision": "980d2afceb985a5598d7bc6116557b75469857f2",
11           "version": "1.0.0"
12         }
13       }
14     ]
15   },
16   "version": 1
17 }
```

Listing B.8: *Package.resolved*

Now you can go ahead and run your application. After completing all of the above steps, Tifig should look like this:

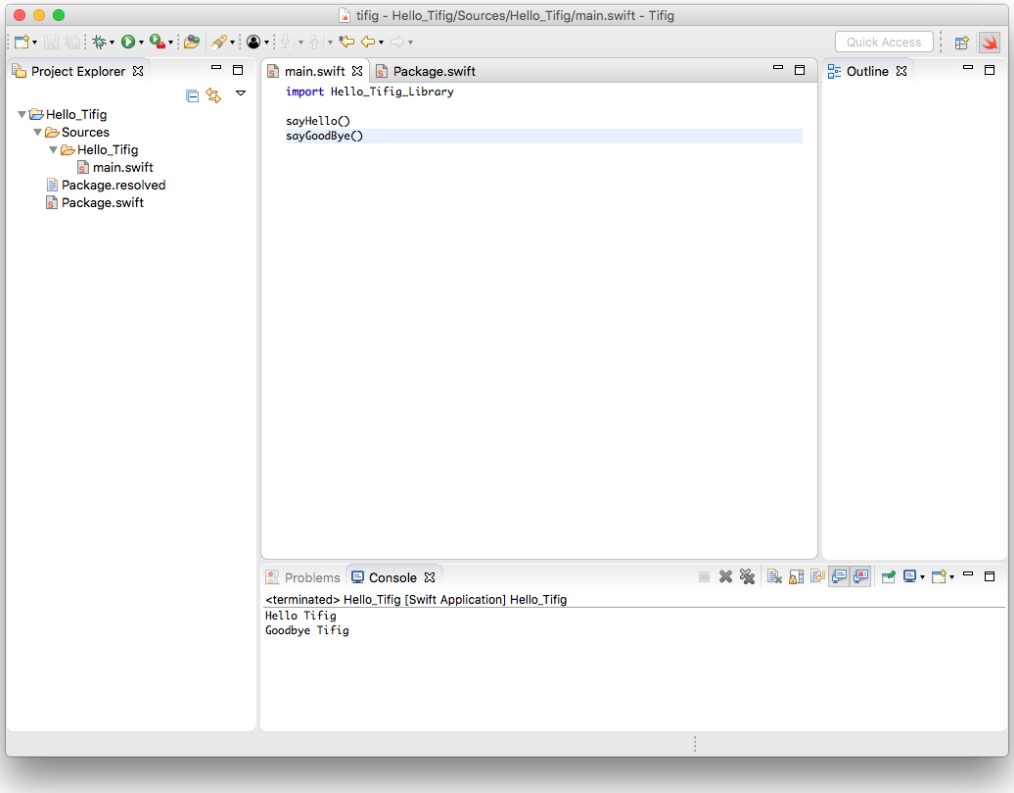

Figure B.10.: *Finished project*

Summary

You have successfully finished all the tutorials and are now ready to start using Tifig. We hope you will enjoy working on your Swift projects with it.
If you have ideas or wishes for further tutorials, please let us know.

C. Agreement for Master Thesis

Student: Mario Meili
Semester: HS 2017/18
Advisor: Prof. Dr. Farhad Mehta
Project Partner: Institute for Software (IFS) HSR
Project Start Date: 18.08.2017
Project End Date: 09.02.2018
ECTS-Credits: 27 ECTS Credits

Project Title

Supporting Swift 4 Generics in Tifig

Goals and Project Description

Swift is a multi-paradigm programming language developed by Apple Inc., released in 2014 and open-sourced in 2015. Being only three years old, Swift still experiences many changes between major releases. Swift 4, which will be released in Q3 of 2017, introduces a set of changes regarding generics, which are outlined below.

Tifig is a Swift IDE based on Eclipse. Its development was started as a Masters project by Toni Suter[1]. Tifig currently supports Swift version 3.1.

In order to make development using Tifig in Swift 4 possible, the Tifig IDE needs to be developed further.

Aim:
The main aim of this master thesis is to develop the Tifig IDE further in order to support Swift 4 Generics. Additionally, tasks that promote the practical use and further development of the Tifig IDE will be undertaken as per the resources available.

Task proposed currently:

[1] `https://eprints.hsr.ch/575/1/HS16-EP-MA-ToniSuter-Tifig.pdf`

1. To modify the Tifig Swift parser to support the newly introduced syntax.
2. To extend the Tifig type checker by the necessary constraints to enable correct type resolution for Swift 4.
3. To adapt the Tifig indexer to guarantee a correct indexing order and indexing correctness.
4. To consolidate existing documentation and documentation relevant to the tasks executed as part of this project in a form that promotes continual development of the Tifig IDE.
5. To complete current tasks with high priority according to the issue page of the Tifig repository[2].

The Swift evolution website[3] summarises the current changes under review for upcoming releases. The following proposals shall be implemented in order to support Swift 4 Generics in Tifig:

1. SE-0142: Permit where clauses to constrain associated types
2. SE-0148: Generic Subscripts
3. SE-0156: Class and Subtype existentials

Deliverables

- A technical report in English, describing the work done as part of this project.
- Consolidated developer source code and documentation in English and in a form that is easily usable for further development of the Tifig IDE (Replaces the formal requirement of a scientific paper).
- A poster in A0 format summarising the results of this project.
- A critical personal reflection on the project and a statement of originality.
- A DVD containing all artefacts produced as part of this project, including source code.
- A final oral presentation of results with discussion.

[2] `https://github.com/tonisuter/tifig/issues`
[3] `https://apple.github.io/swift-evolution`

Competencies to Be Gained (Professional, Methodological and Self-Competence)

- The ability to understand, reflect on, and present scientific results.
- The ability to postulate, develop and evaluate hypotheses systematically, using the scientific method.
- Contribute to the state of the art in the application of programming language theory and technology in the industry.

Assessment Criteria

Per the module description SWSY_MT:

1. Overall assessment
   Criteria: Originality, innovativeness and applicability of the project results. Achievement of all project goals.
2. Organisation and Execution
   Criteria: Formulation of the task description, project planning, planned and systematic execution of the project, independent thought, dedication and collaboration skills.
3. Report
   Criteria: Content, structure, presentation and language.
4. Presentation
   Criteria: Consideration of the target audience, language and content.
5. Content
   Criteria: Preliminary study, requirement analysis, design, complexity, and scope. Quality of the artefacts produced.