Automatisierung der Dokumentation für die Sprache Polarity

Bachelorarbeit

Philip-Daniel Ebsworth
Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
13. April 2025

Autor: Philip-Daniel Ebsworth
Referent: Prof. Dr. Klaus Ostermann
Eingereicht: 13. April 2025

Inhaltsverzeichnis

1 Einleitung

Listing 1.1: Polarity Beispielcode

```
-- | Expressions of the object language
data Exp {
    -- | Variables using a deBruijn representation
    Var(x: Nat),
    -- | Lambda abstractions
    Lam(body: Exp),
    -- | Function applications
    App(lhs rhs: Exp)
}
```

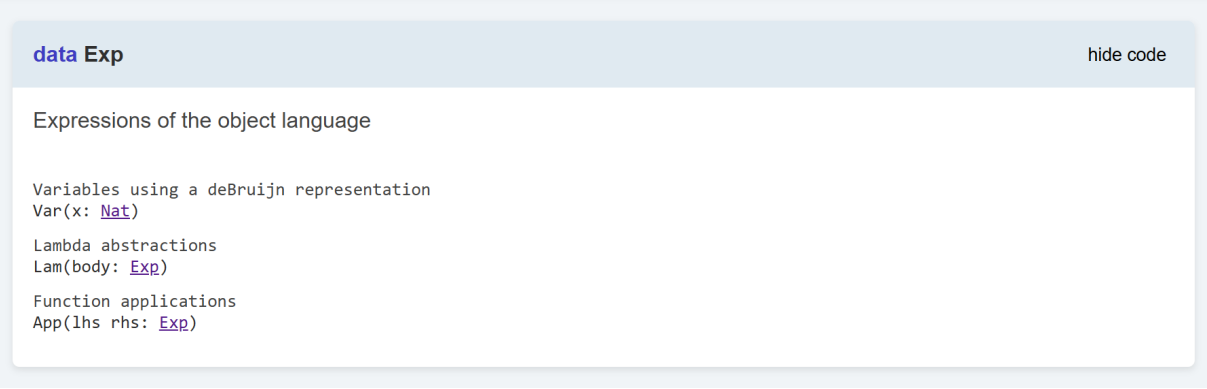

Abbildung 1.1: Beispiel einer automatisch generierten HTML-Dokumentation

1.1 Motivation

Dokumentation ist ein grundlegender Bestandteil der Softwareentwicklung und gewinnt besonders in Bezug auf stark typisierte Sprachen wie *Polarity* weiter an Bedeutung. Um Entwicklerinnen und Entwicklern die Arbeit mit der Sprache zu ermöglichen, ist eine konsistente, strukturierte und aktuelle Dokumentation unerlässlich. Für eine wissenschaftliche Programmiersprache wie *Polarity*, die sich auf fortgeschrittene Konzepte wie abhängige Typen stützt und sich zudem in einem frühen Entwicklungsstadium befindet, ist eine übersichtliche und aktuelle Dokumentation von besonderer Bedeutung. Manuelle Dokumentation bietet dabei nicht nur den Nachteil, dass die Erstellung und Wartung zeitaufwendig und fehleranfällig sind, sondern auch, dass sie oft nicht den aktuellen Stand des Codes widerspiegelt. Daher bietet sich für die Dokumentation von *Polarity* ein automatisierter Prozess an, um diese konsistent, aktuell und übersichtlich zu halten.

1.2 Zielsetzung

Mit dieser Motivation ist es das Ziel dieser Arbeit, einen vollautomatischen Prozess zu entwickeln, der die bestehende Polarity-Dokumentation nahtlos in eine ansprechende HTML-Dokumentation überführt. Der Fokus liegt dabei nicht nur auf der inhaltlichen Korrektheit der Dokumentation, sondern auch auf einer übersichtlichen und nutzerfreundlichen Darstellung, die den Anforderungen moderner Web-Standards entspricht.

Auf diese Weise soll sichergestellt werden, dass Änderungen in *Polarity* schnell und zuverlässig in der Dokumentation abgebildet werden und ein konsistentes Werkzeug für Nutzerinnen und Nutzer entsteht.

1.3 Aufbau der Arbeit

Diese Arbeit ist wie folgt strukturiert: Zunächst werden in *Kapitel 2* die theoretischen Konzepte aus *Deriving Dependently-Typed OOP from First Principles* vorgestellt, auf denen Polarity maßgeblich basiert. Zudem werden Grundlagen und Beispiele automatisierter Dokumentation erläutert.
Um den grundlegenden Aufbau der Polarity-Sprache sowie des hier vorgestellten Projekts zu erklären, wird in *Kapitel 3* die eigentliche Struktur beschrieben.
Der Kern dieser Arbeit liegt in *Kapitel 4*, welches den Prozess der automatischen Dokumentation unter Verwendung von *Rust* detailliert beschreibt.
Kapitel 5 soll einen Einblick in die Performance und Vollständigkeit der Dokumentation geben.
Abschließend werden in *Kapitel 6* die wichtigsten Ergebnisse zusammengefasst und ein Ausblick auf mögliche Erweiterungen für zukünftige Arbeiten gegeben.

2 Grundlagen

Im Folgenden wird, um einen Einblick in die spezifischen Anforderungen für Polarity zu ermöglichen, sowohl die Arbeit von Binder u. a. 2024 „Deriving Dependently-Typed OOP from First Principles“ als auch die Unterscheidung zwischen Daten und Codaten sowie Definitionen und Codefinitionen erläutert. Auf diesen Grundlagen wird die Sprache Polarity vorgestellt und ihre Besonderheiten eingeordnet. Anschließend werden grundlegende Begriffe und Anforderungen für eine automatisierte Dokumentation untersucht und am Beispiel bestehender Systeme für Haskell und Rust erläutert. Abschließend werden anhand dieser Bausteine die Anforderungen für eine Dokumentation für Polarity abgeleitet.

2.1 Deriving Dependently-Typed OOP from First Principles

Das Expression Problem ist der Ausgangspunkt für die Arbeit „Deriving Dependently-Typed OOP from First Principles“ von Binder, Skupin, Süberkrüb und Ostermann. Das Expression Problem beschreibt laut Binder u. a. 2024, dass die meisten Typen entweder dahingehend erweiterbar sind, neue Wege zur Produktion des Typs zu eröffnen oder neue Wege für den Konsum des Typs zu unterstützen, jedoch nicht beides gleichzeitig. Dabei stellt genau dieser Unterschied eine Möglichkeit dar, zwischen funktionaler und objektorientierter Programmierung zu differenzieren. Funktionale Programmierung ermöglicht es, leicht neue Konsumenten zu erweitern, erschwert jedoch die Produktion neuer Typen. Objektorientierte Programmierung hingegen erleichtert die Produktion neuer Typen, macht jedoch die Erweiterung um neue Konsumenten schwieriger, wie bei Binder u. a. 2024 erklärt.

Binder u. a. 2024 zufolge gilt diese Unterscheidung auch für abhängig getypte Sprachen, die beinahe ausschließlich der funktionalen Programmierung folgen. Ein Grund dafür ist, dass die meisten Theorien zu abhängig getypten Sprachen besser für die funktionale Programmierung erforscht sind. Der neue Ansatz von Binder u. a. 2024 besteht daher darin, einen abhängig typisierten, objektorientierten Kalkül zu entwickeln, der als Grundlage für eine objektorientierte, abhängig getypte Programmiersprache dienen kann.

Um eine solche Sprache zu konstruieren, stützen sich Binder u. a. 2024 auf Defunktionalisierung und Refunktionalisierung, um eine objektorientierte Sprache direkt aus einer funktionalen Sprache abzuleiten. Dabei definieren Binder u. a. 2024 die Defunktionalisierung wie Danvy und Nielsen 2001 als eine Programmtransformation welche auf dem gesamten Programm operiert, die Funktionen höherer Ordnung eliminiert, indem sie Lambda-Abstraktionen durch Konstruktoren eines Datentyps ersetzt. Ergänzend dazu wird eine top-level-Apply-Funktion eingeführt. Die Refunktionalisierung nach Danvy und Millikin 2009 stellt das partielle Inverse der Defunktionalisierung dar. Sie führt Funktionen höherer Ordnung wieder ein, indem sie Konstruktoren durch Lambda-Abstraktionen ersetzt.

Auf dieser Grundlage zeigten Rendel, Trieflinger und Ostermann 2015, dass sich Defunktionalisierung und Refunktionalisierung auf beliebige Daten- und Codaten-Typen verallgemeinern lassen. Dadurch werden diese Transformationen sowohl mächtiger als auch symmetrischer, da die Refunktionalisierung nun nicht mehr nur ein partielles, sondern ein vollständiges Inverses der Defunktionalisierung darstellt.

Im Weiteren gehen Binder u. a. 2024 von einer Dualität zwischen funktionaler und objektorientierter Programmierung aus. Um eine solche Dualität überhaupt erst zu erklären, ist eine

klare Abgrenzung zwischen funktionaler und objektorientierter Programmierung erforderlich. Sie definieren die Essenz der funktionalen Programmierung als algebraische Datentypen sowie Pattern Matching, während sie die Essenz der objektorientierten Programmierung als das Programmieren gegen Interfaces beschreiben. Zudem ist laut Binder u. a. 2024 eine klare Abgrenzung von Daten sowie Definitionen die der funktionalen Programmierung zugeordnet werden und Codaten sowie Codefinitionen die der objektorientierten Programmierung zugeordnet werden erforderlich. Eine genauere Einordnung von Daten und Codaten sowie Definitionen und Codefinitionen wird in Abschnitt 2.2 Beispielhaft erklärt.

Im weiteren Verlauf der Arbeit leiten Binder u. a. 2024. auf dieser Grundlage eine abhängig getypte Programmiersprache ab, in der Daten und Codaten symmetrisch und gleichberechtigt behandelt werden. Damit demonstrieren sie, dass es möglich ist, eine abhängig getypte, objektorientierte Programmiersprache zu entwickeln, die auf den Prinzipien der funktionalen Programmierung basiert. Polarity ist die Verwirklichung dieser Ideen und wird in Abschnitt 2.3 vorgestellt.

2.2 Daten, Codaten, Definitionen und Codedefinitionen

Um eine genauere Einordnung von Daten und Codaten sowie Definitionen und Codefinitionen zu ermöglichen, werden die Begriffe im Folgenden erläutert. Dabei werden Beispiele direkt aus der Programmiersprache Polarity verwendet, um die Konzepte zu verdeutlichen.

Daten wie nach dem Konzept von Binder u. a. 2024 für Funktionale Programmierung definiert, sind algebraische Strukturen, die durch Konstruktoren definiert werden. Ein übliches Beispiel hierfür ist die Definition einer Liste:

Listing 2.1: Definition einer Liste in Polarity

```
data List(a: Type) {
  Nil(a: Type): List(a),
  Cons(a: Type, x: a, xs: List(a)): List(a)
}
```

Hierbei werden Listen entweder durch das leere Element `Nil` oder durch die Konstruktion eines neuen Listenelements mit `Cons` gebildet.

Codaten stehen im direkten gegensatz zu Daten und werden durch Beobachtungen definiert. Ein Beispiel ist ein unendlicher Stream von Werten:

Listing 2.2: Definition eines Streams in Polarity

```
codata Stream(a: Type) {
  -- | The head observation which yields the first element.
  Stream(a).hd(implicit a: Type): a,
  -- | The tail observation which yields the remainder of the stream.
  Stream(a).tl(implicit a: Type): Stream(a)
}
```

Hier wird ein Stream nicht durch eine Liste von Elementen beschrieben, sondern durch die Möglichkeit, das erste Element (`head`) und den Rest des Streams (`tail`) auszulesen. Dieses Prinzip erlaubt es, auch unendliche Datenstrukturen zu definieren.

Definitionen sind Funktionen oder Operationen, die für Datenstrukturen definiert werden. Sie verwenden typischerweise pattern matching, um ihre Argumente zu verarbeiten. Ein Beispiel für die Definition einer einfachen Negationsfunktion für Booleans ist:

Listing 2.3: Definition einer Negation in Polarity

```
def Bool.not: Bool {
  T => F,
  F => T
}
```

Dabei wird die Negation eines Booleans durch Abfrage auf die möglichen Werte `T` und `F` definiert.

Im Gegensatz dazu stehen Codefinitionen, die für Codatenstrukturen verwendet werden. Sie spezifizieren, wie Codatenstruktur auf Beobachtungen reagieren. Ein Beispiel für eine unendliche Sequenz ist:

Listing 2.4: Definition einer Repeat-Funktion in Polarity

```
codef Repeat(a: Type, elem: a): Stream(a) {
  .hd(_) => elem,
  .tl(_) => Repeat(a, elem)
}
```

Dieses Code-Beispiel definiert einen unendlichen Stream, bei dem der Kopf(head) (`hd`) immer den Wert `elem` liefert. Der Schweif(tail) (`tl`) verweist rekursiv wieder auf `Repeat(a, elem)`, wodurch eine endlose Sequenz identischer Elemente entsteht.

2.3 Polarity

Polarity ist die Umsetzung der von Binder u. a. 2024 in ihrer Arbeit „Deriving Dependently-Typed OOP from First Principles“ vorgestellten Konzepte. Polarity ist eine funktionale Programmiersprache, die Daten und Codaten gleichberechtigt behandelt. Wie auf der Website der Sprache vom Polarity lang team 2024 erklärt wird, verzichtet Polarity auf fest im Compiler verankerte Typen und erlaubt es dem Nutzer, alle Typen als benutzerdefinierte Codaten zu implementieren. Dabei befindet sich die in Rust geschriebene Sprache noch in der Entwicklung und dient vornehmlich der Erforschung der von Binder u. a. 2024 vorgestellten Konzepte. Momentan befindet sich die Sprache noch in der Entwicklung, erfreut sich aber bereits einer wachsenden Community.

2.4 Dokumentation von Programmiersprachen

Die Dokumentation stellt in jeder Programmiersprache die zentrale Informationsquelle dar, um Entwicklerinnen und Entwickler den Einstieg in die Sprache zu erleichtern und eine effiziente Nutzung zu ermöglichen. Dabei werden in der Regel grundlegende Konzepte sowie die Syntax der Sprache erläutert, sodass Nutzende schnell den Zugang zur Programmiersprache finden. Auch bei der Verwendung von Bibliotheken oder Frameworks ist eine gute Dokumentation unerlässlich, da sie hilft, die Funktionen und Möglichkeiten der jeweiligen Module zu verstehen und effektiv anzuwenden.

Damit eine Dokumentation effektiv nutzbar ist, sollte sie klar und konsistent strukturiert sein. Zudem ist es wichtig, stets aktuelle Inhalte zu präsentieren, da veraltete Informationen zu Missverständnissen führen können. Moderne Dokumentationen werden häufig automatisiert aus dem Quellcode generiert, sodass Änderungen im Code unmittelbar auch in der Dokumentation berücksichtigt werden. Um eine übersichtliche Struktur und eine optimale Durchsuchbarkeit zu gewährleisten, sollte die Dokumentation in verschiedene Abschnitte unterteilt werden. Moderne Programmiersprachen setzen daher vermehrt auf automatisierte Dokumentationswerkzeuge, einige davon werden in Abschnitt 2.5 näher beschrieben, die es ermöglichen, die Dokumentation direkt

aus dem Quellcode zu generieren. Zudem bieten manche Dokumentationen interaktive Elemente wie Codebeispiele oder Tutorials, die das Verständnis und die Anwendung der Sprache weiter erleichtern.

2.5 Bestehende Dokumentationswerkzeugen

Viele Programmiersprachen setzen auf Werkzeuge zur Automatisierung ihrer Dokumentation. Im Folgenden werden zwei dieser Werkzeuge untersucht, um Anforderungen für eine Dokumentation für Polarity abzuleiten. Bei den untersuchten Werkzeugen handelt es sich um das Dokumentationstool Haddock für Haskell sowie Rustdoc für Rust. Beide Werkzeuge setzen auf die automatisierte Generierung von Dokumentation, haben sich in der Praxis bewährt und sollten als Vorbild für die Dokumentation von Polarity dienen können.

2.5.1 Haddock in Haskell

Haskell setzt mit *Haddock* auf ein Werkzeug, das speziell für die Dokumentation von Haskell-Code entwickelt wurde. Wie in Marlow 2002 beschrieben, nimmt Haddock eine Sammlung von Haskell-Quellmodulen und erzeugt Dokumentation in einem oder mehreren Ausgabeformaten. Dabei ist derzeit nur HTML das einzig vollständig unterstützte Format. Vom Entwickler im Quellcode verfasste Kommentare, die in einer bestimmten Annotation enthalten sind, werden von Haddock interpretiert und in die Dokumentation übernommen. Wie in Simon Marlow 2025 erläutert, versteht Haddock außerdem das Haskell-Modulsystem und kann Querverweise selbstständig und abhängig vom Quellcode generieren. Dank dieser Verbindung zwischen Quellcode und Kommentaren wirken sich Änderungen direkt auf die Dokumentation aus. Sowohl Verweise innerhalb von Modulen als auch die durch die Annotationen vorgegebenen Informationen werden automatisch erkannt und in die fertige Dokumentation übernommen. Die Informationen bleiben dabei auch bei Änderungen im Quellcode stets aktuell. Nicht zuletzt wird durch die einheitliche Struktur das Arbeiten mit der Dokumentation maßgeblich erleichtert.

2.5.2 Rustdoc in Rust

Da Polarity in Rust geschrieben ist, bietet sich das Rust-Dokumentationstool *Rustdoc* als Vorbild für die Dokumentation von Polarity an. Vom Prinzip her ähnelt Rustdoc Haddock. Auch hier werden Kommentare im Quellcode verwendet, um die Dokumentation automatisch zu generieren. Anders als in Haddock darf in Rustdoc, wie vom Rust-Team beschrieben, entweder ein Wurzelverzeichnis des Projekts oder alternativ eine Markdown-Datei als Eingabe verwendet werden. In beiden Fällen generiert Rustdoc HTML sowie CSS- und JavaScript-Dateien, die für die Dokumentation benötigt werden. Auch hier liegt der Fokus auf der Kombination von Quellcode und Kommentaren, sodass Änderungen direkt in die Dokumentation übernommen werden können. Durch die einheitliche Form wird eine konsistente und aktuelle Dokumentation gewährleistet.

2.6 Anforderungen an eine HTML-basierte Dokumentation für Polarity

Aus den Untersuchten lassen sich einige Anforderungen für Polarity ableiten. Wie bei Haddock und Rustdoc sollte die Dokumentation für Polarity automatisch aus dem Quellcode generiert werden. Da Polarity sich derzeit in der Entwicklung befindet, wird so sichergestellt, dass die Dokumentation aktuell bleibt.

Ein weiterer Aspekt ist die Struktur der Dokumentation. Bei der Erstellung sollten gegebenenfalls Abhängigkeiten direkt aus den Modulen hergeleitet werden; eine solche Herleitung ermöglicht den Aufbau von Querverweisen und erleichtert die Navigation in der Dokumentation. In Verbindung mit einer HTML-basierten Dokumentation wird dadurch eine schnelle und übersichtliche Darstellung gewährleistet.

Um eine gut übersichtliche Dokumentation zu generieren, ist ein automatisiertes Syntax-Highlighting unerlässlich. Durch die farbliche Hervorhebung von Schlüsselwörtern und Syntax-Elementen wird die Lesbarkeit deutlich verbessert. Um ein besseres Verständnis zu ermöglichen, sollten Codebeispiele direkt in die Dokumentation eingebunden werden. Dadurch können Entwicklerinnen und Entwickler die Sprache direkt anhand von Beispielen erlernen und anwenden.

Wie in vielen Fällen wären auch interaktive Elemente wünschenswert, die die Dokumentation aufwerten würden. Da ein solches Feature jedoch den Rahmen dieser Arbeit sprengen würde, wird darauf im Folgenden verzichtet.

3 Projektbeschreibung

Im Folgenden wird die Struktur der Programmiersprache Polarity erläutert, wobei die für die Erstellung der Dokumentation wesentlichen Komponenten hervorgehoben werden. Anschließend wird der Ablauf der Dokumentationserstellung sowie die einzelnen Schritte im Detail beschrieben. Zum Schluss erfolgt eine Betrachtung der ausgewählten Werkzeuge, die in den Dokumentationsprozess integriert wurden.

3.1 Projektstruktur von Polarity

3.1.1 Hauptverzeichnisse der Sprache Polarity

- **app** Enthält alle Anwendungen für die Kommandozeile, darunter auch eine Anwendung zur Generierung der Dokumentation.
- **bench** Beinhaltet eine Suite für Leistungs-Benchmarking der Sprache.
- **examples** Sammlung von Beispielprogrammen, die für die bei der Automatisierung in HTML-code überführt werden sollen.
- **std** Die Standardbibliothek von Polarity, welche wichtige Funktionen und Module bereitstellt.
- **test** Integrationstests für die Sprache, unterteilt in:
  - **suites** Enthält verschiedene Testfälle.
  - **test-runner** Implementiert den Testausführungsmechanismus.
- **web** Eine Web-Demo-Anwendung zur Demonstration der Sprache direkt im Browser.
- **target_pol** Enthält die generierte Dokumentation der Sprache.

3.1.2 Implementierung der Sprache

Das Verzeichnis **lang** enthält die eigentliche Implementierung von Polarity:

- **ast** Enthält die Definition für den abstrakten Syntaxbaums AST.
- **docs** Hauptverzeichnis für die automatische HTML-Dokumentation der Sprache.
- **driver** enthält die Implementierung des Compiler-Drivers, der für die Steuerung der Kompilierung zuständig ist.
- **elaborator** Umwandlung eines ungetypten Syntaxbaums in einen getypten Syntaxbaum.
- **lowering** Transformation des konkreten Syntaxbaums in den abstrakten Syntaxbaum.
- **lsp** Implementierung eines Language Servers LSP zur Unterstützung von Entwicklungsumgebungen.

- **miette_util** Verwaltung von Quellcode-Spans für Fehlermeldungen und Debugging.
- **parser** Beinhaltet den Lexer und Parser zur syntaktischen Analyse des Quellcodes.
- **printer** Übersetzung des abstrakten Syntaxbaums in menschenlesbaren Quellcode.
- **transformations** Lifting oder Refunktionalisierung.

3.1.3 Für die Dokumentation relevante Komponenten

Für die Erstellung der Dokumentation sind einige Komponenten und damit Verzeichnisse von Bedeutung. Der eigentliche Befehl zum Erstellen der Dokumentation wird in der Kommandozeilenanwendung **app** implementiert, welche die Funktionen aus dem Verzeichnis **docs** verwendet, um die Quelldateien zu lesen und in eine strukturierte Form zu überführen. Das Verzeichnis **docs** bildet dabei das zentrale Element, das den eigentlichen Prozess zur Automatisierung der Dokumentation enthalten soll. Alle grundlegenden Implementierungen werden in diesem Modul abgelegt, um einen modularen Aufbau zu gewährleisten.

Aus diesem Modul heraus wird Bezug auf die Implementierung der Sprache Polarity genommen. Für das Auslesen der Quelldateien und das Erfassen der modularen Struktur werden Funktionen aus den Verzeichnissen **lang/driver** sowie **lang/ast** herangezogen, wobei auch rekursiv Funktionen aus **lang/lowering** verwendet werden. Zur Überführung der Quelldateien in eine strukturierte Form wird auf die Funktionen aus dem Verzeichnis **lang/printer** zurückgegriffen. Für Fehlerbehandlung und Debugging werden die Funktionen aus dem Verzeichnis **miette_util** eingesetzt.

3.2 Ablauf der Dokumentation

Die Generierung der Dokumentation besteht aus mehreren Schritten, die es ermöglichen, Polarity-Code zu analysieren, zu Annotieren, zu formatieren und schließlich zu strukturieren.

3.2.1 Parsen der Quelldatei

Zunächst wird die zu verarbeitende Polarity-Quelldatei geladen und ihr Pfad aufgelöst. Der Inhalt der Datei wird anschließend durch eine Parser-Funktion in eine interne Datenstruktur überführt. Wie in allen Bereichen des Prozesses wird darauf geachtet, dass die Daten konsistent und fehlerfrei verarbeitet werden. Sollten während dieses Prozesses Fehler auftreten, werden diese durch eine Fehlerbehandlungsroutine formatiert und entsprechend ausgegeben.

3.2.2 Hinzufügen von Annotationen

Ein zentraler Bestandteil der Dokumentationsgenerierung ist die erneute Durchmusterung der Datenstruktur, um die einzelnen Bestandteile des Codes mit Annotationen zu versehen. Die spezifische Annotationen, die beispielsweise Schlüsselwörter, Typen oder Kommentare hervorheben liefern die Informationen für den Renderer, um die Daten in die passende Struktur zu überführen. So kann beispielsweise ein Codeblock automatisch mit Syntaxhervorhebungen versehen werden.

3.2.3 Erzeugung einer formatierten Textrepräsentation

Sobald die Daten in einer geeigneten Repräsentation vorliegen, erfolgt die Überführung in eine formatierte HTML-Syntax. Hierbei wird die Datenstruktur durchlaufen und für jedes Element eine zur Annotation passende Ausgabe generiert.

3.2.4 Integration in die finale Dokumentation

Abschließend wird der generierte Inhalt in ein vordefiniertes Template eingebunden. Dabei wird darauf geachtet, dass die Struktur der Dokumentation erhalten bleibt und alle Informationen korrekt eingebettet werden. Die finale HTML-Datei wird an einem vom Quellpfad abhängigen Ort gespeichert, um eine logische Organisation der generierten Dokumentation sicherzustellen.

3.3 Auswahl der Werkzeuge

3.3.1 Askama: Template-Rendering für Rust

Für die Darstellung des aus den Quelldaten erstellten HTML-Dokuments ist mehr als nur die reine Textausgabe notwendig. Zum einen benötigt eine HTML-Datei standardisierte Strukturen wie Header und Footer, zum anderen müssen die Daten in eine für den Nutzer ansprechende Form gebracht werden. Hierfür kommt die Rust-Bibliothek `Askama` zum Einsatz. Wie in der Dokumentation beschrieben, handelt es sich bei AskamaOchtman 2025 um eine Template-Engine, die sich an das Jinja-Template-System Projects 2025 aus der Python-Welt anlehnt und eine effiziente sowie typsichere Lösung für das Erstellen von HTML-Templates in Rust bietet.

Vorteile von Askama

Die Nutzung eines Template-Systems für die Dokumentationserstellung bringt mehrere Vorteile mit sich. Einerseits ermöglicht es die zentrale Verwaltung der HTML-Struktur, sodass Änderungen an der Darstellung an einer einzigen Stelle vorgenommen werden können. Andererseits erlaubt die Modularität von Askama eine sinnvolle Zerlegung der Templates in kleinere, wiederverwendbare Bestandteile, was die Wartbarkeit und Erweiterbarkeit der Dokumentationsgenerierung verbessert. Ein weiterer entscheidender Vorteil ist die Überprüfung der Templates bereits zur Kompilierzeit. Dadurch können Fehler frühzeitig erkannt und vermieden werden, was die Zuverlässigkeit des Systems erhöht.

Integration von Askama in den Dokumentationsgenerator

Im Rahmen dieser Arbeit wird Askama genutzt, um die aus dem Polarity-Code gewonnenen Informationen in strukturierte HTML-Seiten zu überführen. Der Prozess erfolgt in mehreren Schritten: Zunächst werden die Polarity-Quelldateien in einer strukturierten Datenrepräsentation erfasst. Anschließend wird ein Datenmodell erstellt, das diese Informationen verwaltet und an das entsprechende Askama-Template übergibt. Die Templates definieren die Struktur der HTML-Seiten, indem sie Platzhalter für Inhalte und Formatierungen vorsehen. Abschließend wird die finale HTML-Ausgabe erzeugt und in einer Zieldatei gespeichert.

Askama-Beispielanwendung

Listing 3.1: Integration von Askama in den Dokumentationsgenerator

```
#[derive(Template)]
#[template(path = "documentation.html")]
struct DocumentationTemplate<'a> {
    title: &'a str,
    content: &'a str,
}
```

```
fn generate_html(title: &str, content: &str) -> String {
    let template = DocumentationTemplate { title, content };
    template.render().unwrap()
}

fn main() {
    let title = "Polarity Dokumentation";
    let content = "<p>Text der Seite</p>";

    let html_output = generate_html(title, content);
    std::fs::write("output.html",
                   html_output).expect("Failed to write file");
}
```

Hier wird eine Struktur `DocumentationTemplate` definiert, die die Variablen `title` und `content` enthält. In der Funktion `generate_html` wird ein Objekt dieser Struktur erstellt und in ein Template eingefügt. Der generierte HTML-Code wird anschließend in einer Datei gespeichert, die als Dokumentation verwendet werden kann.

3.3.2 Einsatz der Bibliothek `pretty` für die Dokumentation

Ein weiteres Werkzeug, das für die Überführung der Codedateien strukturierte Form genutzt wird, ist die Bibliothek `pretty`. Diese Bibliothek bietet umfangreiche Funktionen zur Formatierung von Texten und ermöglicht die Erstellung strukturiert formatierter Ausgaben wie auf Löbel 2024 erklärt.

Vorteile von `pretty`

Durch die Annotation der Codeblöcke besteht die Möglichkeit, verschiedene Varianten von Codeelementen unterschiedlich zu formatieren und den generierten HTML-Code entsprechend anzupassen. Mit den Funktionen von `pretty` kann die Struktur der Ausgabe präzise gesteuert werden. Code-Fragmente lassen sich gezielt gruppieren, einrücken und mit zusätzlichen Annotationen versehen, um eine optimale Lesbarkeit zu gewährleisten. Dies ist für die automatische Generierung einer strukturierten HTML-Dokumentation von Vorteil.

Integration von `pretty` in den Dokumentationsgenerator

Im Rahmen dieser Arbeit wird `pretty` eingesetzt, um den Code aus den Polarity-Quelldateien zu annotieren und anschließend in eine passende HTML-Struktur zu überführen. Der Prozess erfolgt in mehreren Schritten: Zunächst werden die Codeblöcke analysiert, ihre Bestandteile anhand ihrer Funktion annotiert und als verschachtelte Datenstruktur gespeichert. Anschließend werden diese annotierten Strukturen mithilfe einer render Funktion in das gewünschte HTML-Format konvertiert. Der so generierte HTMLCode wird schließlich als Askama-Variable übergeben und in das finale Template eingebunden. Diese Vorgehensweise gewährleistet, dass der Code nicht nur korrekt dargestellt, sondern auch inhaltlich strukturiert und verständlich präsentiert wird.

`pretty`-Beispielanwendung

Listing 3.2: Integration von pretty zur Code-Formatierung wie auf Löbel 2024

```
impl SExp {
```

```
    pub fn to_doc(&self) -> RcDoc<()> {
        match *self {
            Atom(ref x) => RcDoc::as_string(x),
            List(ref xs) =>
                RcDoc::text("(")
                    .append(RcDoc::intersperse(
                        xs.iter().map(|x| x.to_doc()),
                        RcDoc::line()
                    ).nest(1).group())
                    .append(RcDoc::text(")"))
        }
    }
}
```

In diesem Beispiel wird eine Implementierung für eine S-Expression-Repräsentation definiert. Die Methode `to_doc()` formatiert die Struktur mithilfe der `pretty`-Bibliothek. Einzelne Atome werden als String ausgegeben, während Listen eingerückt und gruppiert werden, um eine klar erkennbare hierarchische Struktur zu erzeugen.

4 Implementierung

In diesem Kapitel wird die technische Umsetzung der automatischen Dokumentation für Polarity vorgestellt. Ausgangspunkt ist eine in Rust geschriebene Anwendung, welche mehrere Arbeitsschritte zur Erstellung der HTML-Dokumente automatisiert. Dazu zählen das Einlesen und Parsen der Quelltexte, die Erzeugung eines untyped syntax tree (UST), sowie die Transformation dieses Baumes in eine HTML-Seite.

4.1 Automatischer Schreibvorgang der HTML-Dokumentation

Listing 4.1: Auszug aus `/docs/src/doc.rs`

```rust
pub async fn write_html() {
    if !Path::new(CSS_PATH).exists() {
        fs::create_dir_all(Path::new(CSS_PATH).parent().unwrap())
            .expect("Failed to create CSS directory");
        fs::write(CSS_PATH, CSS_TEMPLATE_PATH)
            .expect("Failed to create CSS file");
    }
    write_modules().await;
}

async fn write_modules() {
    let css_path = get_absolut_css_path();
    let folders: Vec<&Path> = vec![Path::new("examples/"), Path::new("std")];
    let path_list = get_files(folders.clone());
    let list = generate_html_from_paths(folders);
    for (source_path, target_path) in path_list {
        let mut db = Database::from_path(&source_path);
        let uri = db.resolve_path(&source_path)
                    .expect("Failed to resolve path");
        let prg = db.ust(&uri).await.expect("Failed to get UST");

        let title = source_path.file_stem().unwrap().to_str().unwrap();
        let code = prg.generate_docs();
        let content = generate_module_docs(title, &code);
        let html_file = generate_html(title, &list, &content, &css_path);

        fs::write(target_path, html_file.as_bytes())
            .expect("Failed to write to file");
    }
}

fn generate_module_docs(title: &str, content: &str) -> String {
    let template = ModuleTemplate { title, content };
```

```
    template.render().unwrap()
}
#[derive(Template)]
#[template(path = "module.html", escape = "none")]
struct ModuleTemplate<'a> {
    title: &'a str,
    content: &'a str,
}

fn generate_html(title: &str, list: &str, code: &str, css: &str) -> String {
    let template = IndexTemplate { title, list, code, css };
    template.render().unwrap()
}
#[derive(Template)]
#[template(path = "index.html", escape = "none")]
struct IndexTemplate<'a> {
    title: &'a str,
    list: &'a str,
    code: &'a str,
    css: &'a str,
}
```

Die Funktion `write_html()` ist der Haupteinstiegspunkt in die automatische Dokumentation.

Zunächst wird sichergestellt, dass die benötigte CSS-Datei angelegt wird, falls sie noch nicht vorhanden ist. Anschließend wird `write_modules()` aufgerufen, welche alle wesentlichen Schritte zur Erstellung der HTML-Dokumentation enthält.

Im ersten Schritt werden in

```
let folders: Vec<&Path> = vec![Path::new("examples/"), Path::new("std")];
```

die Pfade zu den Quelltexten festgelegt, die in der Dokumentation berücksichtigt werden sollen. Aus diesen werden mit `let path_list = get_files(folders.clone());` alle `.pol`-Dateien ermittelt und in einer Liste von Tupeln zusammengefasst, wobei jedes Tupel den Quell- und Zielort enthält.

Im nächsten Schritt wird durch `generate_html_from_paths(folders)` eine Navigationsleiste erzeugt, die die gefundenen Module und Ordner auflistet und direkt in eine mit Hyperlinks versehene HTML-Struktur überführt.

Die eigentliche Dokumentation wird anschließend in einer Schleife über die `path_list` für jede Quelldatei erzeugt. Der dazugehörige Ablauf gliedert sich in drei Schritte, die in den folgenden Abschnitten genauer erläutert werden:

- Parsen der Quelltexte in 4.2.
- Annotieren sowie Rendern der einzelnen Deklarationen in 4.3.
- Generierung des eigentlichen HTML-Codes über Template-Funktionen in 4.4.

4.2 Parsen zur Analyse der Quelldateien

Um eine automatische generation der Dokumentation zu ermöglichen, müssen die Quelldateien zunächst eingelesen und analysiert werden. Dazu wird ein *Parser* benötigt, der die syntaktischen Strukturen der Polarity-Dateien erkennt und in eine interne Repräsentation überführt. In Polarity

kommen verschiedene sprachspezifische Konstrukte zum Einsatz darunter *Data-* und *Codata-*Deklarationen, *Def-* und *Codef*-Blöcke sowie *Let*-Anweisungen. wodurch der Parser mehrere syntaktische Elemente erkennen und in geeigneten Strukturen ablegen muss. Dazu gehören:

- Kommentare,
- Typdeklarationen und Importe,
- Konstruktoren und Destruktoren.

Eine solche Struktur ist in Polarity bereits enthalten und wird innerhalb von `write_html()` über die Methode `db.ust(&uri)` aufgerufen, wie der folgende Codeabschnitt veranschaulicht:

```
let mut db = Database::from_path(&source_path);
let uri = db.resolve_path(&source_path).expect("Failed to resolve path");
let prg = db.ust(&uri).await.expect("Failed to get UST");
```

Hier wird in `db` eine Datenbankinstanz erzeugt, die den Quelltext aus `source_path` einliest und analysiert. `resolve_path` ermittelt den konkreten Pfad der Quelldatei, während `ust` den ungetypten Syntaxbaum (UST) zurückgibt. Der Aufruf `db.ust(&uri)` ist essentiell, da er den *untyped syntax tree* (UST) zurückliefert.

4.2.1 Aufbau des untyped syntax tree (UST)

Der Aufbau des *untyped syntax tree (UST)* erfolgt in mehreren Schritten:

1. **Einlesen der Quelldatei und CST-Erzeugung**: `db.cst(&uri)`.
2. **Ermittlung aller Modul-Abhängigkeiten** via `db.deps(&uri)`.
3. **Aufbau bzw. Abruf einer Symboltabelle** für das aktuelle sowie abhängige Module.
4. **Erzeugen des UST** aus dem *CST* und den Symboltabellen über ein *Lowering*.

Der Eigentliche Aufruf des UST geschieht in der Funktion `ust`:

Listing 4.2: Funktionsdefinition aus `/driver/src/database.rs`

```
pub async fn ust(&mut self, uri: &Url) -> Result<Arc<ast::Module>, Error> {
    match self.ust.get_unless_stale(uri) {
        Some(ust) => {
            log::debug!("Found ust in cache: {}", uri);
            ust.clone()
        }
        None => self.recompute_ust(uri).await,
    }
}
```

Dabei wir zunächst geprüft, ob der UST bereits im Cache vorhanden ist und gegebene Falls zurück gegeben. Sollte dies nicht der Fall sein, wird der UST mit `recompute_ust` neu berechnet.

Listing 4.3: Funktionsdefinition aus `/driver/src/database.rs`

```
pub async fn recompute_ust(&mut self, uri: &Url) -> Result<Arc<ast::Module>,
    Error> {
    log::debug!("Recomputing ust for: {}", uri);
```

```
    let cst = self.cst(uri).await?;
    let deps = self.deps(uri).await?;

    // Compute the SymbolTable from the module itself
    // and all direct dependencies.
    let mut symbol_table = SymbolTable::default();
    let module_symbol_table = self.symbol_table(uri).await?;
    symbol_table.insert(uri.clone(), module_symbol_table);
    for dep in deps {
        let module_symbol_table = self.symbol_table(&dep).await?;
        symbol_table.insert(dep.clone(), module_symbol_table);
    }

    let ust = lowering::lower_module_with_symbol_table(&cst, &symbol_table)
        .map_err(Error::Lowering)
        .map(Arc::new);

    self.ust.insert(uri.clone(), ust.clone());
    ust
}
```

Die Funktion lädt zunächst den Concrete Syntax Tree des angegebenen Moduls und dessen direkte Abhängigkeiten.

```
let cst = self.cst(uri).await?;
let deps = self.deps(uri).await?;
```

Anschließend wird eine Symboltabelle aufgebaut, in die zuerst die Symboltabelle des Hauptmoduls eingefügt wird, gefolgt von denen aller Abhängigkeiten. Mit dieser Symboltabelle wird dann das Modul mittels einer Lowering-Funktion in einen untyped syntax tree überführt. Abschließend wird das neu berechnete UST in einer internen Map gespeichert und zurückgegeben.

4.3 Erzeugung des HTML-Codes

Nachdem der UST erzeugt wurde, erfolgt die Umwandlung in HTML-Code. Der Einstieg erfolgt über die Methode `prg.generate_docs()`. Die dabei aufgerufene Funktion:

Listing 4.4: Auszug aus `/docs/src/generate_docs.rs`

```
impl GenerateDocs for Module {
fn generate_docs(&self) -> String {
    self.decls.iter().map(|decl| decl.generate_docs()).collect::<Vec<_>>().
    join("<br>")
}
}
```

Durchläuft alle Deklarationen im Modul und ruft für jede Deklaration die Methode `generate_docs()` auf. Die entstehenden HTML-Fragmente werden dann in einem String zusammengefügt und durch `<br>` getrennt.

Je nach konkretem Deklarationstyp `Data`, `Codata`, `Def`, `Codef` oder `Let` wird an eine spezifische `generate_docs()`-Methode übergeben:

Listing 4.5: Auszug aus `/docs/src/generate_docs.rs`

```
impl GenerateDocs for Decl {
    fn generate_docs(&self) -> String {
        match self {
            Decl::Data(data) => data.generate_docs(),
            Decl::Codata(codata) => codata.generate_docs(),
            Decl::Def(def) => def.generate_docs(),
            Decl::Codef(codef) => codef.generate_docs(),
            Decl::Let(l) => l.generate_docs(),
        }
    }
}
```

4.3.1 Deklaration

Im Folgenden wird der Ablauf anhand der `Data`-Deklaration beispielhaft erläutert. Dabei wird die Deklaration erneut in für die Darstellung relevante Informationen zerlegt:

Listing 4.6: Auszug aus `/docs/src/generate_docs.rs`

```
impl GenerateDocs for Data {
    fn generate_docs(&self) -> String {
        let Data { span: _, doc, name, attr, typ, ctors } = self;
        let doc = if doc.is_none() { "".to_string() } else { format!("{}<br>",
    doc.generate()) };
        let name = &name.id;
        let attr: String = print_html_to_string(attr, Some(&PrintCfg::default
   ()));
        let typ: String = print_html_to_string(typ, Some(&PrintCfg::default()
   ));

        let body = if ctors.is_empty() {
            "".to_string()
        } else {
            format!("<ul>{}</ul>", ctors.generate())
        };

        let data = DataTemplate { doc: &doc, name, attr: &attr, typ: &typ,
   body: &body };
        data.render().unwrap()
    }
}
```

Wichtig sind hierbei `doc`, `name`, `attr`, `typ` und schließlich `body`, die in einem `DataTemplate` zusammengeführt werden. Das `DataTemplate` wird mit den erzeugten Strings für `doc`, `name`, `attr`, `typ` und `body` befüllt, wobei `data.render()` anschließend die Template-Variablen in den entsprechenden HTML-Code umwandelt. Dadurch entsteht eine formatierte HTML-Struktur, die die Polarity-spezifischen Deklarationen übersichtlich abbildet. Der Ablauf des `DataTemplate` wird in 4.4 genauer erläutert.

4.3.2 Annotation

Im obigen Beispiel (`Data::generate_docs`) wird die Funktion `print_html_to_string` aufgerufen, um etwa das `attr`-Feld in einen HTML-String zu überführen. Diese Hilfsfunktion greift auf eine Methode `print_html` zurück, die ihre Ausgabe in ein `io::Write`-Interface schreibt:

Listing 4.7: Auszug aus `/docs/src/printer.rs` sowie `/ast/src/decls.rs`

```
pub fn print_html<W: io::Write, P: Print>(pr: P, cfg: &PrintCfg, out: &mut W)
     -> io::Result<()> {
    let alloc = Alloc::new();
    let doc_builder = pr.print(cfg, &alloc);
    doc_builder.render_raw(cfg.width, &mut render::RenderHtml::new(out))
}

pub fn print_html_to_string<P: Print>(pr: P, cfg: Option<&PrintCfg>) ->
    String {
    let mut buf = Vec::new();
    let def = PrintCfg::default();
    let cfg = cfg.unwrap_or(&def);
    print_html(pr, cfg, &mut buf).expect("Failed to print to string");
    String::from_utf8(buf).expect("Failed to convert Vec<u8> to String")
}

impl Print for Attributes {
    fn print<'a>(&'a self, cfg: &PrintCfg, alloc: &'a Alloc<'a>) -> Builder<'
    a> {
        if self.attrs.is_empty() {
            alloc.nil()
        } else {
            let p = print_comma_separated(&self.attrs, cfg, alloc);
            alloc.text("#").append(p.brackets()).append(alloc.hardline())
        }
    }
}
```

- `print_html` ruft über das `Print`-Trait eine `print(cfg, &alloc)`-Methode auf, in der das Objekt (z. B. `Attributes`) in interne *Doc-Bausteine* (Text, Zeilenumbrüche etc.) überführt wird.

- `doc_builder.render_raw` arbeitet dann mit einem `RenderHtml`, das die *Doc-Bausteine* inklusive möglicher Annotationen in HTML umwandelt.

- `print_html_to_string` kapselt `print_html`, indem es die Ausgabe in einen Puffer (`Vec<u8>`) schreibt, den es anschließend in einen `String` konvertiert.

Auf diese Weise werden sämtliche Polarity-Strukturen von Attributen bis hin zu komplexen Deklarationen automatisiert in HTML übersetzt und profitieren vom integrierten Syntax-Highlighting und Annotationen.

4.3.3 Render

Der eigentliche *Renderer* für die HTML-Ausgabe ist in der folgenden Struktur `RenderHtml` definiert. Dieser nutzt die im vorherigen Schritt erzeugten Annotationen, um das Dokument Schritt für Schritt in HTML zu transformieren:

Listing 4.8: Auszug aus `/docs/src/render/html.rs`

```
impl<W> pretty::RenderAnnotated<'_, Anno> for RenderHtml<W>
where
    W: io::Write,
{
    fn push_annotation(&mut self, anno: &Anno) -> Result<(), Self::Error> {
        self.anno_stack.push(anno.clone());
        let out = match anno {
            Anno::Keyword => "<span class=\"keyword\">",
            Anno::Ctor => "<span class=\"ctor\">",
            Anno::Dtor => "<span class=\"dtor\">",
            Anno::Type => "<span class=\"type\">",
            Anno::Comment => "<span class=\"comment\">",
            Anno::Backslash => "",
            Anno::BraceOpen => "",
            Anno::BraceClose => "",
            Anno::Error => "<span class=\"error\">",
            Anno::Reference { module_uri, name } => &format!(
                "<a href=\"{}#{}\">",
                get_target_path(Path::new(module_uri.as_str())).
   to_string_lossy(),
                name
            ),
        };
        self.upstream.write_all(out.as_bytes())
    }

    fn pop_annotation(&mut self) -> Result<(), Self::Error> {
        let res = match self.anno_stack.last() {
            Some(Anno::Backslash)
            | Some(Anno::BraceOpen)
            | Some(Anno::BraceClose)
            | Some(Anno::Type) => Ok(()),
            Some(Anno::Reference { module_uri: _, name: _ }) => {
                self.upstream.write_all("</a>".as_bytes())
            }
            _ => self.upstream.write_all("</span>".as_bytes()),
        };
        self.anno_stack.pop();
        res
    }
}
```

Die `RenderHtml`–Struktur arbeitet mit dem generischen Parameter `W: io::Write`, sodass beliebige Ausgaben (z. B. Dateien oder Puffer) möglich sind. **Wichtig** ist die *Annotationen*-Verarbeitung mittels `push_annotation` und `pop_annotation`:

- *push_annotation* schreibt passend zur erkannten `Anno`-Variante ein HTML-Tag wie `<span class="keyword">` oder `<a href="...">`.
- *pop_annotation* wird aufgerufen, sobald diese Markierung endet; es wird also `</span>` oder `</a>` in den Ausgabestrom geschrieben. Einige Annotationstypen (`Backslash`, `BraceOpen`, `BraceClose`, `Type`) erzeugen hierbei gegebenenfalls keinen konkreten HTML-Abschluss.

Der so entstehende Code ist im Ergebnis bereits *escaping-sicher* (via `askama_escape`). Auf diese Weise wird Code, der Schlüsselwörter oder Typreferenzen enthält, bei der Ausgabe in farblich markierten bzw. verlinkten Abschnitten dargestellt, was zu einer übersichtlichen und gut lesbaren HTML-Dokumentation führt.

4.4 Integration von Templates und Styling

Die generierten HTML-Fragmente werden mithilfe der *Askama*-Bibliothek in zuvor definierte Templates eingebettet. Dadurch lassen sich Inhalt und Layout klar trennen, was insbesondere bei Wartung und zukünftiger Erweiterung von Vorteil ist.

4.4.1 Askama-Templates

Ein Beispiel ist das folgende `data.html`-Template, das für die Darstellung einer `Data`-Deklaration verwendet wird:

Listing 4.9: Beispiel für ein Askama-Template aus `/docs/templates/data.html`

```
<span class="anchor" id="{{name}}"></span>
<div class="card">
    <div class="card-header">
        <div>
            <span>
                <span class="keyword">data</span> {{name}}{{attr}}{{typ}}
            </span>
        </div>
        <div>
            <button class="button" onclick="toggleCard(this)">
                hide code
            </button>
        </div>
    </div>
    <div class="doc" style="display: block;">
        <span>{{doc}}</span>
    </div>
    <div class="card-content" style="display: block;">
        <code>
            {{body}}
        </code>
    </div>
</div>
```

Dabei werden Platzhalter wie `{{name}}` oder `{{doc}}` dynamisch durch die im entsprechenden `DataTemplate` hinterlegten Werte ersetzt. Diese Aufteilung von Layout und dynamischen Inhalten erleichtert sowohl das gestalterische Feintuning (z. B. Farben und Abstände im *CSS*) als auch die Programmierung.

4.4.2 Beispiel: `index.html`

Nachdem die einzelnen *Daten*-Templates generiert wurden, erfolgt die Integration in eine Hauptseite die Funktionalen Komponenten der Html enthält. es werden dabei auch eine erstellte Navigationbar sowie die css eingebunden. Dabei fungieren `{{title}}`, `{{css}}`, `{{list}}` und `{{code}}` als austauschbare Platzhalter, die über das zugehörige `IndexTemplate` im Rust-Code befüllt werden:

Listing 4.10: Beispiel für ein Askama-Template aus `/docs/templates/index.html`

```
<!DOCTYPE html>
<html lang="de">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{{title}}</title>
    <link rel="stylesheet" href="{{css}}">
</head>
<body>
    <nav>
        <ul>
            {{list}}
        </ul>
    </nav>
    <main>{{code}}</main>

    <script>
        const folders = document.querySelectorAll('.label');

        folders.forEach(folder => {
            folder.addEventListener('click', function(e) {
                e.stopPropagation();
                this.parentElement.classList.toggle('collapsed');
            });
        });

        function toggleCard(button) {
            const content = button.
                            parentElement.
                            parentElement.
                            nextElementSibling.
                            nextElementSibling;
            if (content.style.display === "none" ||
                content.style.display === "") {
                    content.style.display = "block";
                    button.textContent = "hide code";
```

```
            } else {
                content.style.display = "none";
                button.textContent = "show code";
            }
        }
    </script>
</body>
</html>
```

Der `<nav>`-Bereich enthält per generierte Navigationslinks zu den vorhandenen Modulen oder Ordnerstrukturen, die in `{{list}}` eingefügt werden. Im `<main>`-Bereich wird schließlich der eigentliche Dokumentationscode (`{{code}}`) platziert. Über einfache JavaScript-Funktionen können Ordner (`.label`) aufgeklappt werden und Code-Blöcke (`toggleCard`) dynamisch ein- oder ausgeblendet werden.

4.4.3 Einbindung der CSS

Das Layout der generierten Seiten wird durch eine gesonderte CSS-Datei festgelegt.

Listing 4.11: Auszug aus `/docs/templates/style.css`

```
:root {
    --background-1: #f0f4f7;
    --background-2: #dfe6ea;
    --background-3: #e2e8ec;
    --background-4: #e0eaf1;

    --text-color-1: #2f2f2f;
    --text-color-2: #444444;

    --highlight-1: #3f3dc2;

    --shadow: rgba(0, 0, 0, 0.1);
}

html,
body {
    margin: 0;
    padding: 0;
    background-color: var(--background-1);
    color: var(--text-color-1);
    font-family: sans-serif;
    box-sizing: border-box;
}
//
.keyword {
    color: var(--highlight-1);
    font-weight: bold;
}
```

Durch die Definition von `:root`-Variablen lassen sich Farben und Layout konsistent über alle Seiten hinweg steuern. So wird etwa `var(--highlight-1)` für Schlüsselwörter (`.keyword`) eingesetzt,

während `var(--background-1)` oder `var(--shadow)` grundlegende Flächen und Schatten definieren. Damit werden alle generierten HTML-Seiten einheitlich formatiert und optisch konsistent dargestellt und Änderungen am Farbschema oder an der Anordnung von Elementen lassen sich bequem in einer einzigen CSS-Datei vornehmen.

4.5 Zusammenfassung

Abschließend wurde ein System vorgestellt, das Polarity-Quelltexte automatisiert in HTML-Dokumentationen überführt. Ausgangspunkt ist die Erstellung eines *untyped syntax tree (UST)*, der über einen mehrstufigen Parsing-Prozess (inklusive Symboltabellen und Abhängigkeitsauflösung) ermittelt wird. Mithilfe von `generate_docs()`-Methoden werden aus diesem UST strukturierte HTML-Fragmente erzeugt, wobei ein *Renderer* für die korrekte Umwandlung und Hervorhebung verantwortlich ist. Eine klare Trennung von Inhalt und Layout, realisiert durch *Askama*-Templates und ein zentrales CSS-Stylesheet, ermöglicht eine ansprechende und einheitliche Präsentation der Dokumentation.

Der modulare Aufbau gestattet die problemlose Erweiterung durch weitere Sprachfeatures, Dokumentationsformate und zukünftige Transformations Mechanismen. In den nächsten Kapiteln wird dieses Konzept evaluiert und ein Ausblick auf mögliche künftige Weiterentwicklungen gegeben.

5 Evaluation

In diesem Kapitel wird die im Rahmen dieser Arbeit entwickelte automatischen HTML-Dokumentation der Programmiersprache *Polarity* hinsichtlich Effektivität und Anwendbarkeit bewertet. Dabei wird vor allem die Korrektheit des generierten HTML-Codes, die Skalierbarkeit des Dokumentationsprozesses sowie die Erweiterbarkeit bei Zukünftigen Erweiterungen überprüft. Abschließend erfolgt eine kritische Betrachtung, inwieweit die Lösung den zuvor definierten Anforderungen gerecht wird und welche Grenzen sich im Zuge der Entwicklung offenbart haben.

5.1 Versuchsaufbau und Testkonzept

Um den entwickelten Dokumentationsgenerator zu überprüfen wurde in einem ersten Schritt die Richtige Darstellung so wie die korrekte Verlinkung der Module innerhalb der Navigation getestet. Zu diesem Zweck wurden reihen von Ordnerstrukturen erstellt, die unterschiedlich tiefe sowie Dateimengen enthielten.

Um den entwickelten Dokumentationsgenerator systematisch zu überprüfen, wurde er auf verschiedenen Quelldateien und Beispielverzeichnissen ausgeführt, die unterschiedliche Varianten von Polarity-Quelldateien abdecken. Dabei wurde sowohl kleine, einfache Module als auch umfangreiche Beispielprogramme aus der `std`-Bibliothek und dem `examples`-Verzeichnisse getestet. Das Ziel war, sowohl grundlegende Funktionen wie das Erkennen von *data*- und *codata*-Konstrukten als auch komplexere Strukturen, wie verschachtelte *Definitionen* und *Codefinitionen*, zu testen.

Entscheidend hierfür war vor allem die Korrektheit der Erzeugten HTML-Dateien. Dafür wurde überprüft, ob ihre Struktur konsistent ist und ob alle Hyperlinks korrekt auf andere Polarity-Definitionen verweisen. Zudem wurde getestet, ob das Einbinden externer Bibliotheken wie *pretty* oder *Askama* unerwartete Seiteneffekte verursacht.

Ein weiterer wichtiger Aspekt waren Leistungstests, bei denen die Laufzeit des Dokumentationsgenerators sowie sein Ressourcenverbrauch analysiert wurden. Vor allem bei wachsenden Projektgrößen sollte sichergestellt sein, dass der Prozess sowohl effizient als auch stabil bleibt.

5.2 Korrektheit und Vollständigkeit der Dokumentation

Eine zentrale Anforderung bestand darin, den gesamten Polarity-Quellcode einschließlich Typkonstrukte möglichst vollständig abzubilden. Dabei sollte im Sinne einer übersichtlichen Dokumentation nur solche Informationen dargestellt werden, die für eine Nutzer relevant sind. Um auf konsistente und korrekte Darstellung zu prüfen, wurden existente Beispiel Dateien genutzt sowie Testdateien erstellt, in denen *data*, *codata*, *def* und *codef* gezielt variiert wurden. Auch bei den komplexesten Beispielen, die rekursive Typen und verschachtelte Definitionen enthielten, konnte die Dokumentation korrekt generiert werden.

Darüber hinaus wurde geprüft, ob Kommentare und Metainformationen, die in Polarity zur Beschreibung von Funktionen oder Typen verwendet werden, an den richtigen Positionen im HTML-Dokument erscheinen. Die Ergebnisse bestätigten, dass alle relevanten Dokumentationselemente an der erwarteten Stelle im generierten Output integriert wurden.

In einem letzten Schritt wurde die korrekte Verlinkung der einzelnen Module und Typen überprüft. Dabei wurde sichergestellt, dass alle Referenzen innerhalb der Dokumentation korrekt aufgelöst und verlinkt wurden.

5.3 Performance und Skalierbarkeit

Um die Geschwindigkeit des Dokumentationsgenerators zu bewerten, wurden Laufzeitmessungen mit Testprojekten unterschiedlicher Größe durchgeführt. Diese unterschieden sich vor allem in der Anzahl der `.pol`-Dateien sowie deren Verschachtelungstiefe. Bei den kleinsten Beispielprojekten, die nur wenige *data*- oder *def*-Deklarationen enthielten, konnten sämtliche Dokumentationsdateien in unter einer Sekunde generiert werden. Bei mittelgroßen Projekten mit bis zu 50 `.pol`-Dateien, die eine Vielzahl von Typen und Funktionen umfassten, erhöhte sich die Laufzeit auf wenige Sekunden. Der größte Anteil der Verarbeitungszeit entfiel dabei auf das Einlesen und Parsen der Quelldateien.

Der Speicherbedarf blieb während der Generierung im Normalfall stabil, da jeweils nur einzelne Dateien samt ihrer Abhängigkeiten verarbeitet wurden. Dadurch konnte sichergestellt werden, dass auch größere Projekte effizient dokumentiert werden können.

5.4 Grenzen und Limitationen

Trotz der soliden Abdeckung und guten Leistungsfähigkeit des Ansatzes wurden während der Tests auch einige Schwachstellen deutlich. Eine der größten Schwächen ist, dass die generierte HTML-Dokumentation zwar Quelltext-Hervorhebung und eine benutzerfreundliche Navigation bietet, jedoch keine direkte „Live-Ausführung“ des Polarity-Codes möglich ist. Hier wäre es vorteilhaft die Dokumentation, eine Art interaktiven „Playground“ zu erweitern, der es erlaubt code auszuprobieren.

Ein weiterer Punkt, der künftig eine Rolle spielen könnte, sind Erweiterungen der Sprache selbst. Polarity beruht auf einem mächtigen Typkonzept, das im Laufe der Forschung wahrscheinlich noch erweitert wird. Sollte es also neue Sprachkonstrukte geben, müssten auch die Parser- und Render-Komponenten angepasst werden, um diese Neuerungen nahtlos in die Dokumentation zu integrieren.

Auch wenn diese Aspekte den aktuellen Funktionsumfang etwas einschränken, beeinträchtigen sie nicht die Nutzbarkeit des Dokumentationsgenerators für umfangreiche Polarity-Projekte. Die Lösung bleibt ein leistungsfähiges Werkzeug für die automatische Dokumentation mit Potenzial für weitere Erweiterungen.

6 Zusammenfassung und Ausblick

Die vorliegende Arbeit verfolgte das Ziel, einen vollautomatischen Prozess zur Erstellung einer HTML-Dokumentation für die Programmiersprache *Polarity* zu entwickeln und in den bestehenden Entwicklungsablauf zu integrieren. Dabei ergaben sich folgende wesentliche Erkenntnisse und Resultate:

6.1 Wichtigste Ergebnisse

6.1.1 Automatisierung der Dokumentation

Die Implementierung ermöglicht es, Polarity-Quelldateien (`.pol`) einzulesen, zu parsen und direkt in übersichtliche HTML-Strukturen zu überführen. Insbesondere konnten *data*-, *codata*-Deklarationen und *def*-, *codef*-Blöcke mitsamt Kommentaren erfolgreich extrahiert und in einer einheitlichen HTML-Dokumentation dargestellt werden.

6.1.2 Verbesserte Wartbarkeit:

Durch die direkte Einbindung in Polarity wird das Risiko veralteter oder widersprüchlicher Dokumentation erheblich reduziert. Änderungen im Quellcode werden unmittelbar übernommen, wodurch Redundanzen zwischen Code und Dokumentation minimiert werden.

6.1.3 Erweiterbare Struktur:

Dank des modularen Aufbaus können neue Sprachkonstrukte und zusätzliche Template-Layouts problemlos integriert werden. Dies bietet Flexibilität für zukünftige Anpassungen und ermöglicht eine schnelle Reaktion auf Weiterentwicklungen der Polarity-Sprache.

6.2 Ausblick: Weiterentwicklungsmöglichkeiten

Obwohl der Kern der Automatisierung funktionsfähig ist, gibt es verschiedene Ansatzpunkte, um die Lösung weiter zu verbessern:

- **Interaktive Dokumentation:** Eine mögliche Erweiterung wäre die Einbindung eines *Playground*-ähnlichen Moduls. Dies würde es Anwendern erlauben, direkt in der Dokumentation Polarity-Code auszuführen, zu testen und so Konzepte besser zu verstehen.
- **Feingranulare Codeanalyse:** Die Implementierung ließe sich um detailliertere Typ- und Abhängigkeitsanalysen erweitern, um in der HTML-Dokumentation Verweise auf benötigte Bibliotheken oder konkrete Beispielszenarien anzuzeigen.

6.3 Fazit

Mit der in dieser Arbeit vorgestellten Lösung zur automatisierten HTML-Dokumentation konnte eine erweiterbare Grundlage für die Dokumentation von Polarity geschaffen werden. Ausgangspunkt bildete die Idee, sowohl Daten- als auch Codatenstrukturen gemäß der in *Deriving*

Dependently-Typed OOP from First Principles beschriebenen Dualität zu handhaben. Die im Zuge dessen entwickelte Softwarearchitektur zeichnet sich durch eine klare Trennung zwischen Dokumentationsinhalten, Template-Design und Quellcodeanalyse aus. Dadurch lassen sich neue Sprachfeatures oder Veränderungen an Polarity leicht integrieren, ohne das Grundsystem stark anzupassen.

In der Evaluation konnte gezeigt werden, dass das Dokumentationswerkzeug sowohl in kleineren Beispielprojekten als auch bei größeren Codebasen verlässlich arbeitet. Insbesondere die automatische Verknüpfung von Abhängigkeiten und das interaktive Navigationskonzept in der HTML-Ausgabe erleichtern das Verständnis für die verschiedenen Konstrukte der Sprache. Da alle relevanten Informationen direkt aus dem Quellcode abgeleitet werden, reduziert sich der Pflegeaufwand erheblich und Inkonsistenzen zwischen Code und Dokumentation werden zuverlässig vermieden. Somit steht ein System zur Verfügung, das die Transparenz und Wartbarkeit in der Softwareentwicklung mit Polarity entscheidend fördert.

Dennoch bietet das Projekt Potenzial für künftige Weiterentwicklungen: So wäre etwa die Integration einer interaktiven Spielumgebung (*Playground*) oder die zusätzliche Einbindung weiterer Codeanalysen denkbar, um Nutzerinnen und Nutzern noch tiefere Einblicke in Polarity zu gewähren.

Insgesamt zeigt das Ergebnis der Arbeit, dass die praktische Implementierung eine anwenderfreundliche Dokumentation hervorbringt, die die Weiterentwicklung von Polarity langfristig unterstützen wird.

Literatur

Binder, David, Ingo Skupin, Tim Süberkrüb und Klaus Ostermann (2024). „Deriving Dependently-Typed OOP from First Principles“. In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA1, S. 1–27. DOI: 10.1145/3649846. URL: https://doi.org/10.1145/3649846.

Danvy, Olivier und Kevin Millikin (2009). „Refunctionalization at Work“. In: *Science of Computer Programming* 74.8, S. 534–549. DOI: 10.1016/j.scico.2007.10.007.

Danvy, Olivier und Lasse R. Nielsen (2001). „Defunctionalization at Work“. In: *Proceedings of the Conference on Principles and Practice of Declarative Programming*. Florence, S. 162–174. DOI: 10.1145/773184.773202.

Löbel, Marvin (2024). *pretty - A Wadler/Leijen Pretty-Printing Library for Rust*. https://docs.rs/pretty/latest/pretty/. Zugriff am 18. Februar 2025.

Marlow, Simon (2002). „Haddock: A Haskell Documentation Tool“. In: *Proceedings of Haskell'02*. Microsoft Research Ltd., Cambridge, U.K. Pittsburgh, PA, USA: ACM.

Ochtman, Dirkjan (2025). *Askama - Type-safe, compiled Jinja-like templates for Rust*. https://docs.rs/askama/latest/askama/. Zugriff am 18. Februar 2025.

Polarity lang team (2024). *polarity-lang*. Accessed: 2025-03-05. URL: https://polarity-lang.github.io/faq/.

Projects, Pallets (2025). *Jinja - A modern and designer-friendly templating language for Python*. https://jinja.palletsprojects.com/en/stable/. Zugriff am 18. Februar 2025.

Rendel, Tillmann, Julia Trieflinger und Klaus Ostermann (2015). „Automatic Refunctionalization to a Language with Copattern Matching: With Applications to the Expression Problem“. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. Vancouver, BC, Canada: Association for Computing Machinery, S. 269–279. DOI: 10.1145/2784731.2784763.

Rust-Team, The (o. D.). *Was ist Rustdoc?* https://doc.rust-lang.org/rustdoc/what-is-rustdoc.html. Zugriff am 03. März 2025.

Simon Marlow, David Waern (2025). *haddock: A documentation-generation tool for Haskell libraries*. https://hackage.haskell.org/package/haddock. Zugriff am 03. März 2025.

AI-Nutzung

AI Name	Version	Use Case
ChatGPT	GPT-4o	Verbesserung von Grammatik und Rechtschreibung
ChatGPT	GPT-4o	Automatisches Vervollständigen von kleineren Codeabschnitten
ChatGPT	OpenAI o1 and o1-mini	Verbesserung von Grammatik und Rechtschreibung
ChatGPT	OpenAI o1 and o1-mini	Automatisches Vervollständigen von kleineren Codeabschnitten

Tabelle 1: Genutzte AI Modelle

Erklärung

Laut Beschlüssen der Prüfungsausschüsse Bioinformatik, Informatik, Informatik Lehramt, Kognitionswissenschaft, Machine Learning, Medieninformatik und Medizininformatik der Universität Tübingen vom 05.02.2025. Gültig für Abschlussarbeiten (B.Sc./M.Sc./B.Ed./M.Ed.) in den zugehörigen Fächern. Bei Studienarbeiten und Hausarbeiten bitte nach Maßgabe des/der jeweiligen Prüfers/Prüferin.

1. Allgemeine Erklärungen

Hiermit erkläre ich:

- Ich habe die vorgelegte Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.
- Ich habe alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet.
- Die Arbeit war weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens.
- Falls ich ein elektronisches Exemplar und eines oder mehrere gedruckte und gebundene Exemplare eingereicht habe (z.B., weil der/die Prüfer/in(nen) dies wünschen): Das elektronisch eingereichte Exemplar stimmt exakt mit dem bzw. den von mir eingereichten gedruckten und gebundenen Exemplar(en) überein.

2. Erklärung bezüglich Veröffentlichungen

Eine Veröffentlichung ist häufig ein Qualitätsmerkmal (z.B. bei Veröffentlichung in Fachzeitschrift, Konferenz, Preprint, etc.). Sie muss aber korrekt angegeben werden. Bitte kreuzen Sie die für Ihre Arbeit zutreffende Variante an:

☒ Die Arbeit wurde bisher weder vollständig noch in Teilen veröffentlicht.

☐ Die Arbeit wurde in Teilen oder vollständig schon veröffentlicht. Hierfür findet sich im Anhang eine vollständige Tabelle mit bibliographischen Angaben.

3. Nutzung von Methoden der künstlichen Intelligenz (KI, z.B. chatGPT, DeepL, etc.)

Die Nutzung von KI kann sinnvoll sein. Sie muss aber korrekt angegeben werden und kann die Schwerpunkte bei der Bewertung der Arbeit beeinflussen. Bitte kreuzen Sie alle für Ihre Arbeit zutreffenden Varianten an und beachten Sie, dass die Varianten 3.4 - 3.6 eine vorherige Absprache mit dem/der Betreuer/in voraussetzen:

☐ 3.1. Keine Nutzung: Ich habe zur Erstellung meiner Arbeit keine KI benutzt.

☒ 3.2. Korrektur Rechtschreibung & Grammatik: Ich habe KI für Korrekturen der Rechtschreibung und Grammatik genutzt, ohne dass es dabei zu inhaltlich relevanter Textgeneration oder Übersetzungen kam. Das heißt, ich habe von mir verfasste Texte in derselben Sprache korrigieren lassen. Es handelt sich um rein sprachliche Korrekturen, sodass die von mir ursprünglich intendierte Bedeutung nicht wesentlich verändert oder erweitert wurde. Im Zweifelsfall habe ich mich mit meinem/r Betreuer/in besprochen. Alle genutzten Programme mit Versionsnummer sind im Anhang meiner Arbeit in einer Tabelle aufgelistet.

☒ 3.3. Unterstützung bei der Softwareentwicklung: Ich habe KI als Unterstützung beim Schreiben von Code in der Softwareentwicklung genutzt. Es handelt sich hierbei lediglich um Unterstützung und nicht um die automatische Generierung von größeren Programm-Teilen. Im Zweifelsfall habe ich mich mit meinem/r Betreuer/in besprochen. Alle genutzten Programme mit Versionsnummer sind im Anhang meiner Arbeit in einer Tabelle aufgelistet.

☐ 3.4. Übersetzung: Ich habe *nach vorheriger Absprache und mit Erlaubnis meines/r Betreuer/in* KI zur Übersetzung von mir in einer anderen Sprache geschriebenen Texte genutzt. Jede derartige Übersetzung ist im laufenden Text gekennzeichnet und der Anhang meiner Arbeit enthält eine Tabelle mit einem vollständigen Nachweis aller übersetzten Textstellen und der verwendeten Programme mit Versionsnummer.

☐ 3.5. Code-Generierung: Ich habe *nach vorheriger Absprache und mit Erlaubnis meines/r Betreuer/in* KI zur Erzeugung von Code in der Softwareentwicklung genutzt. Der Anhang meiner Arbeit enthält eine Tabelle mit einem vollständigen Nachweis aller derartigen Nutzungen, der verwendeten Programme mit Versionsnummer und der verwendeten Prompts.

☐ 3.6. Text-Generierung: Ich habe *nach vorheriger Absprache und mit Erlaubnis meines/r Betreuer/in* KI zur Erzeugung von Text in meiner Arbeit genutzt. Jede derartige Verwendung von KI ist im laufenden Text gekennzeichnet und der Anhang meiner Arbeit enthält eine Tabelle mit einem vollständigen Nachweis aller derartigen Nutzungen, der verwendeten Programme mit Versionsnummer und der verwendeten Prompts.

Falls ich in irgendeiner Form KI genutzt haben (siehe oben), dann erkläre ich:

Mir ist bewusst, dass ich die Verantwortung trage, falls es durch die Verwendung von KI zu fehlerhaften Inhalten, zu Verstößen gegen das Datenschutzrecht, Urheberrecht oder zu wissenschaftlichem Fehlverhalten (z.B. Plagiaten) kommt.

4. Abschluss und Unterschrift(en)

Mir ist bekannt, dass ein Verstoß gegen diese Erklärung prüfungsrechtliche Konsequenzen haben und insbesondere dazu führen kann, dass die Prüfungsleistung mit „nicht ausreichend“ bzw. die Studienleistung mit „nicht bestanden“ bewertet wird und bei mehrfachem oder schwerwiegendem Täuschungsversuch eine Exmatrikulation erfolgen bzw. ein Verfahren zur Entziehung eines eventuell verliehenen akademischen Titels eingeleitet werden kann.

Philip-Daniel Ebsworth	Tübingen, 13.04.2025	
Vorname, Nachname Student/in	Ort, Datum	Unterschrift

Die Punkte 3.4 - 3.6 erfordern eine Zustimmung des/r Betreuer/in. Sollten Sie einen dieser Punkte angekreuzt haben, dann sollte der/die Betreuer/in bitte hier unterschreiben:

Ich habe der oben genannten Nutzung von KI zur Erstellung der Arbeit zugestimmt.

Vorname, Nachname Betreuer/in	Ort, Datum	Unterschrift