Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

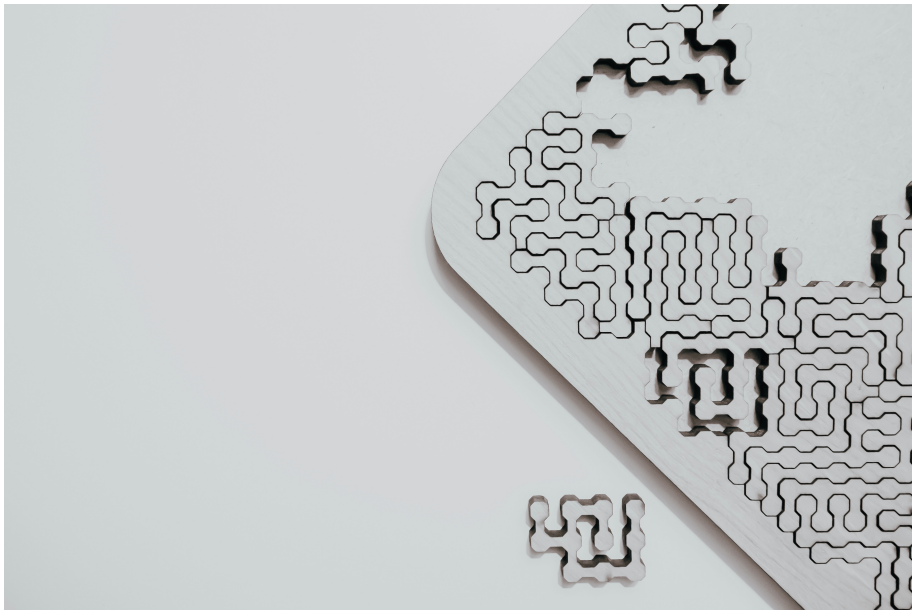
Lehrstuhl Prof. Dr.-Ing. Snelting

Typinferenz und Vollständigkeitsüberprüfung für Structural Pattern Matching in Python

Bachelorarbeit von

Adrian Freund

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M. Sc. Johannes Bechberger
	M. Sc. Simon Bischof
Abgabedatum:	28. Oktober 2021

Zusammenfassung

In Python 3.10 wurde mit dem Match-Statement Structural Pattern Matching eingeführt. Hierbei handelt es sich sowohl um ein Kontrollfluss-, als auch um ein Variablenbindungskonstrukt, welches z.B. dazu verwendet werden kann, Daten aus komplexen Datenstrukturen zu extrahieren.

Für Python gibt es verschiedene Typchecker, welche benutzt werden können, um Typfehler noch vor der Laufzeit zu finden und somit Typfehler zur Laufzeit zu verhindern oder zu verringern. Diese Arbeit beschreibt eine Möglichkeit, solch eine Typüberprüfung auch auf das neue Match-Statement anzuwenden.

In manchen Fällen ist es von Programmierern gewollt, dass eine Fallunterscheidung vollständig ist, also für jede mögliche Eingabe ein Pfad ausgewählt wird. Nicht vollständige Fallunterscheidungen an Stellen, an denen eine vollständige Fallunterscheidung erwartet wird, können zu Fehlern zur Laufzeit führen. Daher wird in dieser Arbeit auch eine Möglichkeit zur Vollständigkeitsüberprüfung von Match-Statements beschrieben.

Weiterhin beschreibt diese Arbeit eine Implementierung dieser Typ- und Vollständigkeitsüberprüfung im Typchecker *mypy*.

Inhaltsverzeichnis

,

1 Einführung

Python ist eine dynamisch typisierte Programmiersprache[1]. Die Referenzimplementierung CPython wird konstant weiterentwickelt und mit neuer Funktionalität ergänzt. Am 4. Oktober 2021 ist Python 3.10 erschienen und hat Unterstützung für Structural Pattern Matching zur Sprache hinzugefügt[2].

1.1 Structural Pattern Matching

Da es in Python kein Switch-Case-Statement gibt werden für Fallunterscheidungen oft lange If-Elif-Ketten verwendet[3]. Um die Lesbarkeit von Code, welcher solche If-Elif-Ketten verwendet, zu verbessern, wurde mit dem Match-Statement Structural Pattern Matching eingeführt[4].

Pattern Matching ist eine Art der Fallunterscheidung, welche vor allem in funktionalen, statisch typisierten Programmiersprachen zur Anwendung kommt, jedoch auch auf objektorientierte und dynamische typisierte Programmiersprachen angepasst wurde. Bei Pattern Matching können im Gegensatz zu If-Elif-Ketten auch neue Variablen gebunden werden und so auch Daten aus komplexen Datenstrukturen extrahiert werden[1].

So wird z.B. in Code 1.1 zwischen fünf verschiedenen Fällen unterschieden. Dabei werden in Fall 1 die Variablen `x` und `y` und in Fall 5 die Variable `other_event` gebunden. In beiden Fällen können die neu gebundenen Variablen im zum Fall gehörenden Block verwendet werden. Aufgrund der Scoping-Regeln in Python können von einem Match-Statement gebundene Variablen auch nach dem Match-Statement verwendet werden[4]. Je nachdem welcher und ob ein Fall ausgeführt wurde können die Variablen hier jedoch auch ungebunden sein.

In Python beginnt ein Match-Statement immer mit einem `match`-Schlüsselwort, gefolgt von einer Expression, dem Match-Subjekt. Anschließend folgen ein oder mehrere Case-Blöcke, welche jeweils aus einem `case`-Schlüsselwort gefolgt von einem Pattern, optional einer durch ein `if`-Schlüsselwort eingeleiteten Guard-Expression und einem Codeblock bestehend aus ein oder mehreren Statements. Wenn das Match-

```
match event.get():
    case Click(position=(x, y)):
        handle_click_at(x, y)
    case KeyPress(key_name="Q") | Quit():
        game.quit()
    case KeyPress(key_name="up arrow"):
        game.go_north()
    case KeyPress(key_name=k) if k.isdigit():
        game.select_slot(int(k))
    ...
    case KeyPress():
        pass # Ignore other keystrokes
    case other_event:
        raise ValueError(
            f"Unrecognized event: {other_event}")
```

Code 1.1: Leicht abgewandeltes Beispiel eines Match-Statements aus PEP 636 [5]

Subjekt mit einem Pattern übereinstimmt und entweder keine Guard-Expression vorhanden ist oder die Guard-Expression zu `True` evaluiert, wird der zum Case-Block gehörenden Codeblock ausgeführt und alle weiteren Case-Blöcke nicht überprüft[4].

Hierbei ist zu beachten, dass manche Pattern weitere Pattern enthalten und so beliebig weit verschachtelt sein können.

1.2 Statische Analyse

Bei einer statischen Analyse wird Quellcode durch ein anderes Programm überprüft. Hierdurch können Fehlern mancher Fehlerkategorien frühzeitig gefunden und anschließend von Programmierer behoben werden. Im Gegensatz zu der dynamischen Analyse wird bei der statischen Analyse das zu analysierende Programm nicht ausgeführt, sondern nur dessen Quellcode überprüft[6].

1.2.1 Typüberprüfung

Ein wichtiger Bereich der statischen Analyse ist die Typüberprüfung. In kompilierten Programmiersprachen wird typischerweise vom Compiler eine Typüberprüfung durchgeführt. Es gibt für manche Sprachen jedoch auch externe Programme zur Typüberprüfung.

CPython, die Standartimplementierung der Python Programmiersprache, nutzt einen Bytecode-Compiler und eine einfache Virtuelle Maschine um den Python-Bytecode auszuführen[7]. Der Python Bytecode-Compiler nimmt keine Typüberprüfung vor[8], weshalb es verschiedene externe Typchecker für Python gibt. Dazu gehören zum Beispiel mypy[1], pytype[2], Pyright[3] und Pyre[4].

1.2.2 Vollständigkeitsüberprüfung

Mittels statischer Analyse kann auch die Vollständigkeit einer Fallunterscheidung, wie z.B. eines Match-Statements oder einer If-Elif-Kette überprüft werden. Eine Fallunterscheidung ist hierbei vollständig, wenn für jede mögliche Eingabe einer der Fälle der Fallunterscheidung zutrifft.

```
from enum import Enum
class Color(Enum):
    RED = 0
    GREEN = 1
    BLUE = 2

def get_color_name(color: Color) -> str:
    match color:
        case Color.RED:
            return "Red"
        case Color.GREEN:
            return "Green"
        case Color.BLUE:
            return "Blue"
```

Code 1.2: Beispiel eines Match-Statements zur unterscheidung von `Enum`-Elementen

So ist das Match-Statement in Code 1.2 vollständig, da `color` den Typ `Color` hat. Wenn dem `Color`-Enum ein weiterer Wert hinzugefügt wird, dem Match-Statement jedoch nicht, so könnte dies zu unerwartetem Verhalten führen. Dies wird in Code 1.3 verdeutlicht. Hier werden zunächst ein Enum `Color` und eine Funktion `get_color_name` definiert. Anschließend wird die Funktion `get_color_name` mit dem Parameter `Color.PURPLE` aufgerufen. Dort trifft im Match-Statement keiner der Fälle zu, weshalb kein Return-Statement ausgeführt wird. Aus diesem Grund gibt

[1] `http://mypy-lang.org/`
[2] `https://google.github.io/pytype/`
[3] `https://github.com/Microsoft/pyright/`
[4] `https://pyre-check.org/`

die Funktion den Standard-Rückgabewert `None` zurück, welcher nicht mit der Typannotation `str` kompatibel ist. Ein Typchecker sollte diesen Typfehler erkennen.

```
from enum import Enum
class Color(Enum):
    RED = 0
    GREEN = 1
    BLUE = 2
    PURPLE = 3

def get_color_name(color: Color) -> str:
    match color:
        case Color.RED:
            return "Red"
        case Color.GREEN:
            return "Green"
        case Color.BLUE:
            return "Blue"

print(get_color_name(Color.PURPLE))
```

Code 1.3: Beispiel eines Python-Programms mit einem Fehler welcher durch Vollständigkeitüberprüfung entdeckt werden könnte

2 Verwandte Arbeiten

Diese Arbeit baut auf das Paper „Dynamic Pattern Matching with Python“ [1] auf, welches beschreibt wie Pattern Matching in einer dynamischen Sprache wie Python umgesetzt werden kann. Hierbei werden Typchecker jedoch nur beiläufig erwähnt. Es bildet die Basis für PEP (Python Enhancement Proposal) 622 [9]. PEP 622 spezifiziert wie Structural Pattern Matching sich in Python verhalten könnte und stellt auch Anforderungen an mögliche Typchecker. Dieser PEP wurde jedoch bevor er in Kraft treten konnte von PEP 634 [10], 635 [4] und 636 [5] abgelöst. Diese PEPs beschreiben eine leicht anderes Verhalten für Structural Pattern Matching in Python und bilden die Basis der Implementierung im CPython 3.10. Da keiner dieser drei PEPs Vorgaben macht, wie ein Typchecker mit einem Match-Statement umgehen sollte, wurde sich für diese Arbeit teilweise an den Anforderungen aus PEP 622 orientiert. Teilweise wurden aber auch gewollt andere Lösungen gewählt.

„Lower your guards: a compositional pattern-match coverage checker“ [11] stellt eine Möglichkeit zur Vollständigkeitsüberprüfung von Patten Matching am Beispiel von Haskell dar. Da es sich bei Python um eine dynamisch typisierte Sprache handelt, sind nicht alle Erkenntnisse daraus direkt auf das Python Match-Statement anwendbar.

„Lessons from Building Static Analysis Tools at Google“ [12] behandelt größtenteils die Integration von statischer Analyse in Arbeitsabläufe, enthält aber auch Informationen dazu, wie Programmier mit Statische-Analyse-Tools umgehen und wie solche Tools designt sein sollten um Programmierer davon abzuhalten, Fehler und Warnungen zu ignorieren.

3 Grundlagen

Im Folgenden wird auf die Details der Typinferenz und Vollständigkeitsüberprüfung für Match-Statements eingegangen. Die Behandlung von Guards wird hier nicht explizit beschrieben. Da Guards beliebige boolsche Ausdrücke sein können, können sie genau wie If-Statements behandelt werden.

3.1 Typinferenz

Bei einem Python Match-Statement können zwei verschiedene Informationen inferiert werden: Die Typen von neu gebundenen Variablen und der genarrowte, also eingeschränkte, Typ des Match-Subjekts. In Python gibt es 10 Arten von Pattern. Zwei davon, das Capture-Pattern und das AS-Pattern binden neue Variablen[4]. Da Pattern jedoch in anderen Pattern verschachtelt vorkommen können müssen für beide Informationen alle Arten von Pattern behandelt werden. So kann z.B. auch ein Class-Pattern neue Variablen binden, indem es Capture-Pattern als Subpattern enthält. Diese neu inferierten Informationen werden der vor der Ausführung des Match-Statements gültigen Typumgebung hinzugefügt und bilden so die Typumgebung für den Codeblock des Cases, zu dem das entsprechende Pattern gehört.

In Code 3.1 hat `s` in der Typumgebung vor dem Match-Statement den Typ `object`. In der Typumgebung des Codeblocks des Cases hat `s` den Typ `Sequence[int]`. Zusätzlich enthält diese Typumgebung die Variable `x` mit dem Typ `int`.

`reveal_type` ist hierbei eine Hilfsfunktion, welche nur zur während dem Typchecken existiert. Sie lässt den Typchecker den statischen Typ der übergebenen Variable ausgeben. Diese Typchecker-Ausgaben sind hier durch Codekommentare dargestellt.

```
s: object
reveal_type(s)  # Revealed type is "builtins.object"

match s:
    case [1, 2, int(x)]:
        reveal_type(s)
        # Revealed type is "typing.Sequence[builtins.int]"
        reveal_type(x)  # Revealed type is "builtins.int"
```

Code 3.1: Beispiel der Typinferenz bei einem Match-Statement

3.2 Vollständigkeitsüberprüfung

Die Vollständigkeitsüberprüfung baut direkt auf der Typinferenz auf. Hier wird ebenfalls rekursiv jedes Pattern behandelt. Zusätzlich zu dem Typ auf den das Subjekts im Codeblock genarrowed werden kann und den Typen der neu gebundenen Variablen muss hier noch ein Rest-Typ inferiert werden. Der Rest-Typ ist der Typ auf den das Subjekt genarrowed werden kann, falls das Pattern nicht zutrifft. Wenn ein Pattern immer zutrifft, so ist der Rest-Typ dieses Patterns der leere Typ $\bot$.

```
s: Union[str, int]

match s:
    case int():
        print("s was an int")
    case x:
        print("s was not an int")
```

Code 3.2: Beispiel der Vollständigkeitsüberprüfung bei einem Match-Statement

Das Pattern des ersten Cases wird gegen den Typ des Match-Subjekts gematched. Anschließend werden alle weiteren Pattern jeweils gegen den Rest-Typ des vorherigen Case gematched. Ist der Rest-Typ eines Patterns der leere Typ, so werden alle folgenden Cases als nicht erreichbar markiert. Ein Match-Statement ist genau dann vollständig, wenn der Rest-Typ des letzten Cases der leere Typ ist.

In Code 3.2 ist der Rest-Typ des Patterns des ersten Cases `str`. Da das Pattern des zweiten Cases immer zutrifft ist der Rest-Typ hier $\bot$. Das Match-Statement ist damit vollständig.

3.3 Arten von Pattern

Die Herleitung der Typen kann durch Herleitungsbäume beschrieben werden. Hierbei beschreibt $case_p$ die bei Pattern p anzuwendende Regel. Γ ist die Typumgebung vor dem Match-Statement, Γ_+ die Typumgebung bei Zutreffen des Patterns und Γ_- die Typumgebung bei nicht zutreffen des Patterns. s ist das Match-Subjekt. Es sei $cond_+(x, y)$ der neue Typ, wenn der Typ x auf den Typ y genarrowed wird und $cond_-(x, y)$ der entsprechende Rest-Typ.

Hierbei seien

$$cond_+(x,y) = \begin{cases} x, & \text{falls } x \leq y \\ y, & \text{falls } x > y \\ intersect(x,y), & \text{sonst} \end{cases}$$

$$cond_-(x,y) = \begin{cases} \bot, & \text{falls } x \leq y \\ x \setminus y, & \text{falls } x \not\leq y \wedge x \text{ ist ein Union-Typ} \\ x, & \text{sonst} \end{cases}$$

Außerdem sei $intersect(x, y)$ ein neuer Typ, der nur während der Typüberprüfung existiert und mit allen Typen kompatibel ist, die sowohl von x, als auch von y erben. $intersect(x, y)$ sei $\bot$, falls ein Typ nicht sowohl von x als auch von y erben kann. In Python ist Mehrfachvererbung bis auf wenige Ausnahmen erlaubt.

$UnionType(x_1 \dots x_n)$ sei ein statischer Typ, für eine Variable, die zur Laufzeit einen der Typen x_1 bis x_n oder einen Subtyp einer dieser Typen hat. Weiterhin sei

$$UnionType(x_1 \dots z \dots x_n) \setminus y = \begin{cases} UnionType(x_1 \dots x_n), & \text{falls } z \leq y \\ UnionType(x_1 \dots z \dots x_n), & \text{sonst} \end{cases}$$

Bei Pattern mit Subpattern sei s_{p_x} das Match-Subjekt, dass gegen Pattern p_x gematched wird.

Wenn für ein Pattern mit den gegebenen Herleitungsregeln kein Γ_+ oder Γ_- hergeleitet werden kann ist der entsprechende Case zur Laufzeit nicht erreichbar und es sei $\Gamma_- = \Gamma$.

3.3.1 Literal-Pattern

Das Literal-Pattern (Code 3.3) matched genau dann, wenn der Wert des Match-Subjektes mit dem Wert des Literals übereinstimmt. Die Ausnahme bilden die Literale

`None`, `True` und `False`, welche nicht nach Wert, sondern nach Identität verglichen werden[4]. Durch ein Literal-Pattern werden keine neuen Variablen gebunden. Der Typ des Match-Subjekts kann jedoch mit Hilfe der Regel $case_{literal}$ genarrowed werden. c ist hierbei das Literal, gegen welches gematched wird.

3.3.2 Capture-Pattern

Das Capture-Pattern (Code 3.4) matched jedes Match-Subjekt und kann daher den Typ des Match-Subjekts nicht narrowen. Der Rest-Typ ist daher der leere Typ. Zusätzlich bindet es eine neue Variable, welche den Wert und daher auch den Typ des Match-Subjekts bekommt[4], was durch Regel $case_{capture}$ ausgedrückt wird.

3.3.3 Wildcard-Pattern

Das Wildcard-Pattern (Code 3.5) matched, genau wie das Capture-Pattern (Abschnitt 3.3.2), jedes Match-Subjekt, bindet aber keine Variable[4]. Hier kann daher das Subjekt nicht genarrowed werden. Der Rest-Typ ist der leere Typ. Das Wildcard-Pattern wird durch Regel $case_{wildcard}$ beschrieben.

3.3.4 Value-Patterns

Das Value-Pattern (Code 3.6) verhält sich ähnlich zu dem Literal-Pattern (siehe Abschnitt 3.3.1). Anstatt eines Literals erwartet es jedoch einen Ausdruck, der einen Punkt enthält, wie z.B. einen Attributszugriff. Ausdrücke welche keinen Punkt enthalten werden als Capture-Pattern (Abschnitt 3.3.2) interpretiert. Ein Value-Pattern matched genau dann, wenn der Wert des Ausdruckt mit dem Wert des Match-Subjektes übereinstimmt[4]. Die Typinferenz wird durch Regel $case_{value}$ beschrieben.

3.3.5 Sequence-Patterns

Das Sequence-Pattern (Code 3.7) matched genau dann, wenn das Match-Subjekt eine Sequenz, also z.B. eine Liste oder ein Tupel, ist und alle Subpattern matchen[4]. Bei einem Sequence-Pattern entspricht der Rest-Typ dem Typ des Match-Subjektes, kann also nicht weiter genarrowed werden, da nur anhand der Typen nicht bestimmt werden kann, ob das Pattern zur Laufzeit matched. Dies hängt von zusätzlichen

```
match s:
    case 1:
        print('s matches literal 1')
    case "Hello world"
        print('s matches literal "Hello World"')
```

Code 3.3: Literal-Pattern

$$case_{literal} \frac{\Gamma \vdash c : \tau_c \qquad \Gamma \vdash s : \tau_s}{\Gamma_+ = \Gamma, s : cond_+(\tau_s, \tau_c) \qquad \Gamma_- = \Gamma, s : cond_-(\tau_s, \tau_c)} \tag{3.1}$$

```
match s:
    case x:
        print("Match subject was " + x)
```

Code 3.4: Capture-Pattern

$$case_{capture} \frac{\Gamma \vdash s : \tau_s}{\Gamma_+ = \Gamma, x : \tau_s \qquad \Gamma_- = \Gamma, s : \bot} \tag{3.2}$$

```
match s:
    case _:
        print("This always matches")
```

Code 3.5: Wildcard-Pattern

$$case_{wildcard} \frac{}{\Gamma_+ = \Gamma \qquad \Gamma_- = \Gamma, s : \bot} \tag{3.3}$$

```
match s:
    case HTTPStatus.OK:
        print("Match subject matches value of
            HTTPStatus.OK")
```

Code 3.6: Value-Pattern

$$case_{value} \frac{\Gamma \vdash v : \tau_v \qquad \Gamma \vdash s : \tau_s}{\Gamma_+ = \Gamma, s : cond_+(\tau_s, \tau_v) \qquad \Gamma_- = \Gamma, s : cont_-(\tau_s, \tau_v)} \tag{3.4}$$

Informationen, welche nicht im Typsystem kodiert sind, wie z.B. der Länge der Sequenz, ab.

Die neuen Typumgebungen, sowie die Match-Subjekte der Subpattern lassen sich hierbei mit den Regeln $case_{sequence_1}$, $case_{sequence_2}$ und $case_{sequence_3}$ herleiten. *Sequence* ist hierbei der abstrakte Typ `typing.Sequence` und $\top$ der universelle Supertyp, welcher in Python dem Typ `object` entspricht. $join(x_1 \dots x_n)$ ist der spezifischste gemeinsame Supertyp der Typen x_1 bis x_n. $A[B]$ steht wie in Python für einen generischen Typen A mit Typparameter B. n sei die Anzahl der Subpattern.

3.3.6 Mapping-Pattern

Das Mapping-Pattern (Code 3.8) enthält eine Menge von Schlüssel-Wert-Paaren, wobei die Schlüssel Ausdrücke und die Werte Subpattern sind. Es matched wenn das Match-Subjekt eine `get`-Methode hat und die Subpattern die Ergebnisse dieser `get`-Methode bei Eingabe des entsprechenden Schlüssels matchen[4]. Ein Mapping-Pattern kann das Match-Subjekt nicht narrowen, da es nicht vom Typ des Match-Subjektes, sondern nur von dem Vorhandensein einer Methode auf dem Match-Subjekt abhängig ist.

Es lässt sich durch die Regeln $case_{mapping_1}$ und $case_{mapping_2}$ beschreiben. Hierbei sei k_i der i-te Schlüssel und n die Anzahl der Schlüssel-Wert-Paare.

3.3.7 Class-Pattern

Ein Class-Pattern (Code 3.9 matched genau dann, wenn das Match-Subjekt vom als Klasse angegebenen Typ oder einem Subtyp ist und alle Subpattern matched. Hierbei werden die Subpattern in Positional-Pattern und Keyword-Pattern eingeteilt. Keyword-Pattern haben die Form `attr=pattern`, wobei das Pattern `pattern` gegen das Attribut `attr` des Match-Subjektes matched wird. Um Positional-Pattern zu matchen muss zunächst das `__match_args__`-Attribut der angegebenen Klasse gelesen werden. Dieses enthält die Namen der Attribute, gegen die die positional Pattern gematched werden[4].

Einige in Python integrierte Klassen wie z.B. `int` und `str` matchen positionale Subpattern nicht gegen ein Attribut des Objektes, sondern gegen das Objekt selbst[4]. Diese Sonderfälle werden der Übersichtlichkeit halber in den folgenden Regeln nicht behandelt.

Die inferierten Typen lassen sich mit den Regeln $case_{class_1}$, $case_{class_2}$ und $case_{class_3}$

```
match s:
    case [p1, p2]:
        print("s matches Sequence [p1, p2]")
```

Code 3.7: Sequence-Pattern

$$case_{sequence_1} \frac{\Gamma \vdash s : Sequence[x]}{\Gamma \vdash s_{p_1} : x \wedge \cdots \wedge \Gamma \vdash s_{p_n} : x} \tag{3.5}$$

$$case_{sequence_2} \frac{\Gamma \nvdash s : Sequence[x]}{\Gamma \vdash s_{p_1} : \top \wedge \cdots \wedge \Gamma \vdash s_{p_n} : \top} \tag{3.6}$$

$$case_{sequence_3} \frac{\Gamma \vdash s : \tau_s \qquad \Gamma_{p_1+} \vdash p_1 : \tau_{p_1} \ \cdots \ \Gamma_{p_n+} \vdash p_n : \tau_{p_n}}{\Gamma_+ = (\bigcup_{x=1}^{n} \Gamma_{p_n+}), s : cond_+(\tau_s, Sequence[join(\tau_{p_1} \cdots \tau_{p_n})]) \qquad \Gamma_- = \Gamma} \tag{3.7}$$

```
match s:
    case {1: p1, "foo": p2}:
        print("s matches Sequence [p1, p2]")
```

Code 3.8: Mapping-Pattern

$$case_{mapping_1} \frac{\Gamma \vdash k_i : \tau_k \qquad \Gamma \vdash s.get : x \to y \qquad \tau_k \leq x}{\Gamma \vdash s_{p_n} : y} \tag{3.8}$$

$$case_{mapping_2} \frac{}{\Gamma_+ = \bigcup_{x=1}^{n} \Gamma_{p_n+} \qquad \Gamma_- = \Gamma} \tag{3.9}$$

herleiten. t sei hierbei die zu matchende Klasse und a_i sei der Attributsname des Attributs, gegen das das i-te Pattern gematched wird.

3.3.8 AS-Pattern

Ein AS-Pattern (Code 3.10) enthält genau ein Subpattern und matched genau dann, wenn dieses Subpattern matched. Außerdem enthält ein AS-Pattern einen Namen und bindet das Match-Subjekt an diesen Namen[4]. Es lässt sich mit Regel $case_{as}$ beschreiben.

3.3.9 OR-Pattern

Ein OR-Pattern matched genau dann, wenn mindestens eines der Subpattern matched. In einem OR-Pattern müssen alle Subpattern die gleichen Namen binden. Wenn mindestens eines der Subpattern den Namen x bindet müssen auch alle anderen Subpattern den Namen x binden[4]. Hier wird nur das erste Subpattern gegen das den Typen des Match-Subjekts gematched. Alle weiteren Subpattern werden gegen den Rest-Typ des vorherigen Patterns gematched. Dies wird durch die Regeln $case_{or_1}$ und $case_{or_2}$ ausgedrückt. Die Regeln $case_{or_3}$ und $case_{or_4}$ beschreiben den Rest der Typinferenz.

3.3.10 Group-Pattern

Ein Group-Pattern (Code 3.12) besteht aus einem Subpattern umgeben von runden Klammern. Es matched genau dann, wenn das Subpattern matched[4] und wird von Regel $case_{group}$ beschrieben.

```
match s:
    case A(p1, attr=p2):
        print("s is of class A with attributes
               matching p1 and p2")
```

Code 3.9: Class-Pattern

$$case_{class_1} \frac{\Gamma \vdash s.a_i : x}{\Gamma \vdash s_{p_n} : x} \tag{3.10}$$

$$case_{class_2} \frac{\Gamma, s : \tau_s \qquad \Gamma_{p_1-} \vdash s : \bot \ \ldots \ \Gamma_{p_n-} \vdash s : \bot}{\Gamma_+ = \Gamma, s : cond_+(\tau_s, t) \qquad \Gamma_- = \Gamma, cond_-(\tau_s, t)} \tag{3.11}$$

$$case_{class_3} \frac{\Gamma, s : \tau_s \qquad \Gamma_{p_i-} \nvdash s : \bot}{\Gamma_+ = \Gamma, s : cond_+(\tau_s, t) \qquad \Gamma_- = \Gamma} \tag{3.12}$$

```
match s:
    case p1 as x:
        print("Match-Subject was " + x)
```

Code 3.10: AS-Pattern

$$case_{as} \frac{\Gamma \vdash s : \tau_s}{\Gamma_+ = \Gamma_{p_1+}, x : \tau_s \qquad \Gamma_- = \Gamma_{p_1-} \qquad \Gamma \vdash s_{p_1} : \tau_s} \tag{3.13}$$

```
match s:
    case p1 | p2:
        print("p1 or p2 matches")
```

Code 3.11: OR-Pattern

$$case_{or_1} \frac{\Gamma \vdash s : \tau_s}{\Gamma \vdash s_{p_1} : \tau_s} \tag{3.14}$$

$$case_{or_2} \frac{\Gamma_{p_i-} \vdash s : \tau_s}{\Gamma \vdash s_{p_{i+1}} : \tau_s} \tag{3.15}$$

$$case_{or_3} \frac{\Gamma_{p_1+} \vdash s : \tau_{p_1} \ \ldots \ \Gamma_{p_n+} \vdash s : \tau_{p_n}}{\Gamma_+ = (\bigcup_{i=1}^{n} \Gamma_{p_i+}), s : UnionType(\tau_{p_1} \ldots \tau_{p_n})} \tag{3.16}$$

$$case_{or_4} \frac{\Gamma_{p_n 1-} \vdash s : \tau_{p_n}}{\Gamma_- = \Gamma, s : \tau_{p_n}} \tag{3.17}$$

```
match s:
    case (p1):
        print("s matches subpattern p1")
```

Code 3.12: Group-Pattern

$$case_{group} \frac{}{\Gamma_{+} = \Gamma_{p_1+} \qquad \Gamma_{-} = \Gamma_{p_1-}} \tag{3.18}$$

4 Implementierung

Die in Kapitel 3 beschriebene Typ- und Vollständigkeitsüberprüfung wurde in dem Open-Source Typchecker mypy[1] implementiert. Mypy hatte bei Beginn dieser Arbeit noch keine Unterstützung für das Python 3.10 Match-Statement und Python 3.10 im Allgemeinen, weshalb zusätzlich zur Unterstützung von Match-Statements auch noch mehrere Fehler in mypy und dessen Dependencies behoben wurden, um es unter Python 3.10 lauffähig zu machen.

4.1 Typinferenz

Die Typinferenz für Match-Statements wurde mithilfe eines Visitors implementiert. Der bereits existierende Visitor `TypeChecker` ruft hierbei für das Pattern jedes Cases den neu implementierten `PatternChecker` Visitor auf, welcher für Subpattern wiederum sich selbst aufrufen kann. Der `PatternChecker` liefert für jedes Pattern den Typ, auf den das Subjekt genarrowed werden kann, falls das Pattern zutrifft, den Typ, auf den das Subjekt genarrowed werden kann, falls das Pattern nicht zutrifft und eine Menge von Name-Typ-Paaren welche die innerhalb des Patterns gebundenen Namen enthält. Der `TypeChecker` behandelt zuerst die Pattern aller Cases und fügt anschließend die neu gebundenen Variablen der aktuellen Typumgebung hinzu. Erst danach werden die Codeblöcke der Cases überprüft. Dadurch können die neu gebundenen Variablen wie in Code 4.1 als Union-Typen inferiert werden. Innerhalb der jeweiligen Cases werden die Union-Typen dann auf einen spezifischeren Typen genarrowed.

4.2 Vollständigkeitsüberprüfung

Python bietet keine offizielle Möglichkeit um zu markieren, dass ein Match-Statement vollständig sein sollte. Unvollständige Match-Statements sind erlaubt und haben auch sinnvolle Anwendungen. Es gibt jedoch einen in der Python-Community verbreiteten

[1] `https://github.com/python/mypy/`

```
m: object

match m:
    case int(x):
        reveal_type(x)
        # Revealed type is "builtins.int"
    case str(x):
        reveal_type(x)
        # Revealed type is "builtins.str"

reveal_type(x)
# Revealed type is "Union[builtins.int, builtins.str]"
```

Code 4.1: Wenn mehrere Cases den gleichen Namen binden wird ein Union-Typ inferiert

Trick die gewünschte Vollständigkeit von If-Elif-Ketten zu markieren[13]. Dieser funktioniert mit dieser Implementierung auch mit Match-Statements. Zunächst wird eine Funktion `assert_never` wie in Code 4.2 definiert. `NoReturn` steht hierbei für den Python-Typ `typing.NoReturn`, welcher mit keinem anderen Typ kompatibel ist. Anschließend wird diese Funktion wie in Code 4.3 am Ende es Match-Statements in einem Case mit Wildcard-Pattern aufgerufen. Bei der Überprüfung des Funktionsaufrufs gibt mypy einen Typfehler aus, da `NoReturn` mit keinem Typen kompatibel ist. Wenn der Case jedoch als unerreichbar inferiert wurde, so wird er nicht überprüft und es wird kein Fehler ausgegeben. Sollte der Case zur Laufzeit doch erreicht werden, z.B. weil durch einen Bug im Typchecker er fälschlicherweise als unerreichbar inferiert wurde, so wird durch das `assert False` zur Laufzeit ein Fehler ausgegeben.

```
def assert_never(x: NoReturn) -> NoReturn
    assert False, f"Unhandled type: {type(x).__name__}"
```

Code 4.2: Definition der `assert_never` Funktion

```
match m:
    ...
    case _:
        assert_never(m)
```

Code 4.3: Als vollständig markiertes Match-Statement

4.3 Guards

Zur Überprüfung der Guards konnten bereits bestehende Mypy-Funkionen für die Überprüfung von If-Statements benutzt werden. Diese erwarten jedoch, dass bei allen im Guard verwendeten Variablen bereits der Typ bekannt ist. Da wie in Abschnitt 4.1 beschrieben zuerst alle Patten überprüft werden und erst anschließend die neuen Variablen gebunden werden können die Guards auch erst überprüft werden, nachdem die Pattern aller Cases überprüft wurden. Da Guards jedoch die Typinferenz beeinflussen können müssten sie eigentlich direkt nach jedem Pattern, anstatt nach allen Pattern behandelt zu werden. In Code 4.4 hat `r` den Typ `Union[int, str]`. Wenn der Guard erst nach Behandlung beider Pattern behandelt werden würde, würde `r` fälschlicherweise als `str` inferiert werden, da davon ausgegangen wird, dass der erste Case für alle `int` zutrifft.

Um dieses Problem zu lösen werden in der Implementierung alle Pattern zweimal behandelt. Bei der ersten Überprüfung werden Guards und Vollständigkeitsüberprüfung ignoriert. Es wird also jedes Pattern gegen das Match-Subjekt gematched. Mit den Ergebnissen dieser ersten Überprüfung werden die Typen der neu gebundenen Variablen gesetzt. Anschließend wird jedes Pattern ein zweites Mal überprüft. Dieses Mal werden Guards und Vollständigkeitsüberprüfung mit einbezogen. Es wird also wie in Abschnitt 4.2 beschrieben das Pattern des ersten Cases gegen den Typ des Match-Statements gematched und jedes weitere Pattern gegen den Rest-Typ nach Behandlung des vorherigen Cases.

```
m: Union[int, str]

match m:
    case int(x) if False:
        pass
    case r:
        reveal_type(r)
        # N: Revealed type is Union[int, str]
```

Code 4.4: Guards können die Inferenz von Typen beeinflussen

5 Evaluation

Da Python 3.10 erst am 4. Oktober 2021 veröffentlicht wurde gibt es bisher nur wenige Verwendungen des Match-Statements, welche für eine Evaluation genutzt werden können. Daher wurde die im vorherigen Kapitel vorgestellte Implementierung mit folgenden Daten getestet:

- Selbst erstellte Pattern-Matching-Tests in der mypy Test-Suite
- Pattern-Matching Tests der CPython Test-Suite
- Beispiele aus den PEPs 634–636

Außerdem wurden Teile der mypy Test-Suite sowohl mit Pyright getestet, als auch nach Scala übersetzt und mit dem Scala Compiler getestet. Die anderen Python-Typechecker, pytype und Pyre, haben noch keine Unterstützung für das Match-Statement.

Es wurde auch versucht, die Beispiele des „Lower your Guards“-Papers[11] von Haskell nach Python zu übersetzen und für weitere Tests zu nutzen. Dies war allerdings bei einem Großteil der Tests aufgrund von Limitierungen des Python Match-Statements nicht möglich, weshalb diese Tests nicht in die Evaluierung eingeflossen sind.

Im Folgenden bezieht sich *mypy* auf mypy mit der im vorherigen Kapitel vorgestellte Implementierung von Typinferenz und Vollständigkeitsüberprüfung für Match-Statements.

5.1 Mypy Test-Suite

Die mypy Test-Suite wurde während der Entwicklung kontinuierlich um Pattern-Matching-Tests erweitert. Insgesamt wurden 12 Parsing-Tests, 9 Semantische-Analyse-Tests und 97 Typchecking-Tests hinzugefügt. Diese Tests sind alle erfolgreich.

5.2 CPython Test-Suite

Die CPython enthält in der Datei `Lib/test/test_patma.py` eine Menge an Tests ohne explizite Typannotationen. Beim Versuch diese zu mithilfe der Implementierung zu überprüfen sind drei Probleme aufgefallen:

5.2.1 Subpattern von Class-Pattern

Beim Matchen von Class-Pattern wurden die Subpattern gegen den falschen Typ gematched, wenn bei der entsprechenden Klasse Subpattern nicht gegen Attribute, sondern gegen das gesamte Match-Subjekt gematched werden. Anstatt gegen den tatsächlichen Typ des Subjekts zu matchen wurde gegen die angegebene Klasse gematched, welche z.B. bei generischen Klassen ungenauer sein könnte. Aufgefallen ist dieser Fehler bei Code 5.1. Code 5.2 ist ein einfacher zu verstehendes Beispiel, welches denselben Fehler auslöst. `m` hat hierbei den Typ `List[int]`. Dieser wird aufgrund des `list` Class-Patterns zunächst gegen `list[Any]` gematched, was zutrifft. Anschließend sollte der gesamte Typ `List[int]` gegen das Subpattern `[x]` gematched werden. Stattdessen wurde jedoch der Typ `list[Any]` dagegen gematched und daher für das innere Capture-Pattern ein falscher Typ inferiert.

Dieser Fehler wurde behoben und ein entsprechender Regression-Test zur mypy Test-Suite hinzugefügt.

5.2.2 Programmabsturz bei nicht matchendem Sequence-Pattern

Außerdem wurde entdeckt, dass die Implementierung abstürzt, wenn ein Sequence-Pattern das Match-Subjekt nicht matchen kann überprüft wird, da in diesem Fall eine Variable nicht zugewiesen wurde und die Implementierung später versucht diese Variable zu lesen.

Dies wurde durch Konvertierung des Programmcodes der Implementierung mit dem Tool `com2ann`[1] verursacht.

Python unterstützt seit der Version 3.6 zusätzlich zu Funktionsannotationen auch Variablenannotationen. In früheren Python-Versionen mussten Variablentypen daher als Code-Kommentare annotiert werden. Zu Beginn der Implementierung wurde

[1] `https://github.com/ilevkivskyi/com2ann`

```
x = [[{0: 0}]]
match x:
    case list([({-0-0j: int(real=0+0j, imag=0-0j)  |
            (1) as z},)]):
        y = 0
self.assertEqual(x, [[{0: 0}]])
self.assertEqual(y, 0)
self.assertEqual(z, 0)
```

Code 5.1: Der fehlschlagende Testfall aus der CPython Test-Suite

```
from typing import List
m: List[int]

match m:
    case list([x]):
        reveal_type(x)
        # Revealed type should be int, but is Any
```

Code 5.2: Einfache Reproduktion des Fehlers, der von Code 5.1 ausgelöst wird

Python 3.5 noch vom mypy unterstützt. Später wurde die minimale Python Version auf 3.6 angehoben und mithilfe vom `com2ann` alle Typ-Kommentare in Typannotationen umgewandelt. Dabei ist ein `None`-Standartwert verloren gegangen. Der Fehler wurde manuell behoben und ein Regression-Test hinzugefügt. Zusätzlich wurde der Code nach ähnlichen Fehlern durchsucht und es wurden keine weiteren Vorkommen gefunden.

5.2.3 Anpassung der Match-Argumente bei Vererbung

Bei Class-Pattern mit Positional-Pattern werden die entsprechenden Attributnamen im Attribut `__match_args__` nachgesehen. Um zu verhindern, dass dieses sich während der Laufzeit ändert und somit falsche Typen inferiert werden gibt die Implementierung eine Fehlermeldung aus, wenn `__match_args__` nicht als `Final`, also konstant, markiert ist. In manchen der CPython Tests wird `__match_args__` jedoch in Unterklassen überschrieben. Dies ist nicht möglich, wenn das Attribut als `Final` markiert ist. Da diese Einschränkung kann nicht behoben werden, ohne durch Nicht-erzwingen der `Final`-Annotation eventuell falsche Typinferenzen zuzulassen, wurde dieses Verhalten beibehalten.

```
if not can_match:
    new_type = None  # type: Optional[Type]
elif isinstance(current_type, TupleType):
    ...
```

Code 5.3: Code vor der Konvertierung

```
if not can_match:
    new_type: Optional[Type]
elif isinstance(current_type, TupleType):
    ...
```

Code 5.4: Code nach der Konvertierung

```
new_type: Optional[Type]
if not can_match:
    new_type = None
elif isinstance(current_type, TupleType):
    ...
```

Code 5.5: Manuell korrigierter Code

5.3 Pyright

Die Match-Statement Test der Mypy-Testsuite wurden auch mit Pyright getestet. Verwendet wurde hierzu Pyright 1.1.178. Hierbei sind 16 Unterschiede zwischen den Implementierungen aufgefallen. In drei Fällen sind diese Unterschiede auf verschiedene Designentscheidungen zurückzuführen, wobei keine der Optionen eindeutig besser ist.

In sieben Fällen wurden von Pyright falsche Informationen, also Informationen, welche nicht dem Verhalten von CPython entsprechen, inferiert, von mypy jedoch Korrekte. In weiteren vier Fällen wurden zwar von beiden Implementierungen korrekte Informationen inferiert, mypy lieferte jedoch genauere Ergebnisse. In einem Fall inferierte Pyright einen genaueren Typ.

Ein letzter Fall wurde sowohl von Pyright, als auch von mypy falsch inferiert.

Nach Rücksprache mit einem dem Pyright Entwickler Eric Traut wurde mir bestätigt, dass es sich bei sechs der sieben Fehler tatsächlich um Bugs handelt. Der siebte Fehler wurde absichtlich nicht behandelt, da es sich um einen seltenen Randfall handelt, bei welchem nach Meinung vom Herrn Traut die Nachteile der Behandlung überwiegen.

Im Folgenden werden nur die Fälle, in welchen diese Implementierung Informationen falsch oder ungenauer als Pyright inferiert.

5.3.1 Sequence-Pattern narrowed keine Union-Typen

Wenn mypy ein Subjekt mit einem Union-Typ auf ein Sequence-Pattern matched, so wurde der Typ des Subjektes anschließend nicht genarrowed. Der Grund hierfür war, dass zwar ein neuer Typ berechnet wurde, anschliessend aufgrund eines Programmierfehlers aber vergessen wurde, die Narrowing-Funktion mit diesem Typen aufzurufen. Dieser Fehler wurde behoben. In Code 5.6 ist das Verhalten vor der Korrektur zu sehen.

```
from typing import List, Union
m: Union[List[List[str]], str]

match m:
    case [list(['str'])]:
        reveal_type(m)
        # mypy infers Union[List[List[str]], str]
        # pyright infers List[List[str]]
```

Code 5.6: Beispiel eines Programms, bei welchem Pyright einen genaueren Typen inferiert

5.3.2 Narrowing bei Literal-Pattern

Bei Literal-Pattern inferieren sowohl mypy als auch Pyright Typen, welche nicht korrekt sind. In Python kann mittels der `__eq__`-Methode bestimmt werden, wie Typen sich sowohl bei Vergleichen mit `==`, als auch bei Vergleichen in Match-Statements verhalten. [14]. Aufgrund dessen kann mit Hilfe von Literal-Pattern im Allgemeinen keine Aussage über den Typ des Match-Subjektes getroffen werden. Die Ausnahme bilden dabei Literal-Pattern mit den Werten `True`, `False` und `None`, welche in Match-Statements nicht mit `__eq__`, sondern nach Identität verglichen werden[4].

Da es sich hierbei aber um einen seltenen Randfall handelt und der Standartfall, bei welchem das Narrowing korrekt ist, viele Anwendungen hat, ergibt es Sinn zugunsten der Nutzbarkeit auf etwas Korrektheit zu verzichten. Das Problem tritt nur auf, wenn eine Klasse ihre `__eq__` Methode so definiert, dass sie bei Eingabe von `int`- oder `str`-Werten `True` zurückliefern kann.

```
class A:
    def __eq__(self, other: object):
        return True

m: object = A()

match m:
    case 1:
        reveal_type(m)
        # Both mypy and pyright infer "Literal[1]",
        # correct would be "object"
```

Code 5.7: Beispiel eines Literal-Pattern mit überschriebener __eq__-Methode

5.4 Scala

Zur Evaluation wurden Teile der Mypy-Testsuite nach Scala übersetzt und mit dem Scala Compiler getestet. Aufgrund von Limitierungen von Scala konnten nicht alle Match-Statement-Tests übersetzt werden.

Scala Pattern Matching unterstützt keine Sequence- und Mapping-Pattern. Außerdem erlaubt Scala in Or-Pattern (Pattern-Alternatives) keine Variablenbindung. In Class-Pattern unterstützt Scala nur spezielle Case-Classes, welche mit Python Dataclasses vergleichbar sind. Außerdem werden in Class-Pattern keine Keyword-Argumente unterstützt. Daher können Tests, welche diese Funktionen nutzen, nicht nach Scala übersetzt werden. Andererseits hat Scala Unterstützung für Sealed Classes, welche von Python nicht unterstützt werden. Sealed Classes wurden in PEP 622[9] vorgeschlagen, wurden jedoch bei der Aufteilung auf die PEPs 634–635 entfernt. Sealed Classes sind Klassen bei welchen alle Kindklassen zu Compilezeit bekannt sind[15]. Dadurch können sie ähnlich wie Enums verwendet werden.

Der Scala Compiler bietet keine Möglichkeit den inferierten Typ eines Ausdrucks zu Debugging-Zwecken ausgeben zu lassen. Es kann jedoch durch Zuweisen an eine Variable mit Typ `Nothing` ein Fehler erzeugt werden, in dessen Fehlermeldung auch der Typ des zugewiesenen Ausdrucks ausgegeben wird.

Sämtliche Test wurden mit Scala Version 2.13.6 durchgeführt.

Hierbei sind mehrere Unterschiede in den Implementierungen aufgefallen.

```
m: A = new A
m match {
    a => val x: Nothing = a
}
```

Code 5.8: Zuweisung an eine Variable mit Typ `Nothing`

5.4.1 Match-Subjekt Narrowing

Scala narrowed das Match-Subjekt nicht. Stattdessen muss das Pattern auf einen neuen Namen gebunden werden, um eventuelles Narrowing nutzten zu können. In Code 5.9 behält `m` den Typ `Parent`. Mypy würde bei dem entsprechenden Python-Code den Typ von `m` auf `A` narrowen.

```
class Parent
class A extends Parent
val m: Parent = new A

m match {
  case a: A => {} // m has type Parent, a has type A
}
```

Code 5.9: Scala narrowed keien Match-Subjekte

Als Folge daraus generiert Scala, im Gegensatz zu mypy (Code 5.10), auch keine Intersection-Typen.

```
class A: ...
class B: ...

m: B

match m:
    case A():
        reveal_type(m)  # N: Revealed type is
        # "__main__.<subclass of "B" and "A">"
```

Code 5.10: Mypy generiert Intersection-Typen

5.4.2 Inkompatible Typen

Werden in Scala durch ein Match-Statement inkompatible Typen verglichen, so wird dies direkt zur Compilezeit als Fehler markiert. Mypy meldet in diesem Fall standardmäßig keinen Fehler, sondern markiert nur den zum Case gehörenden Block als unreachable.

Durch die Kommandozeilenoption `--warn-unreachable` kann jedoch auch mypy dies direkt als Fehler melden.

```
m: int

match m:
    case "str":
        reveal_type(m)
```

Code 5.11: Ein nicht ausführbarer Case in Python

```
test.py:4: error: Subclass of "int" and "str" cannot exist:
  would have incompatible method signatures
test.py:5: error: Statement is unreachable
```

Code 5.12: Ausgabe von Code 5.11 mit `--warn-unreachable`

6 Fazit und Ausblick

Diese Arbeit hat gezeigt, dass Typinferenz und Vollständigkeitsüberprüfung von Pattern-Matching-Konstrukten auch in dynamisch typisierten Programmiersprachen möglich ist. In manchen Fällen muss ein Kompromiss zwischen Korrektheit und Nutzbarkeit gefunden werden, da ein vollständig korrekter Typchecker aufgrund von Eigenschaften von Python in manchen Fällen keine nützlichen Informationen inferieren kann.

Es wurde eine Implementierung von Typinferenz und Vollständigkeitsüberprüfung für Python 3.10 Match-Statements in mypy erstellt, welche in vielen Fällen nützliche und genaue Ergebnisse liefert und mit anderen Typcheckern, sowohl für Python, als auch für andere Programmiersprachen vergleichbar, in manchen Fällen sogar genauer, ist.

Für den ersten Teil dieser Implementierung wurde bereits ein Pull Request auf dem mypy GitHub-Repository erstellt[1]. Dieser befindet sich zum Zeitpunkt der Abgabe dieser Arbeit gerade in der Code-Review-Phase. Der zweite Teil der Implementierung wird anschließend folgen. Mehrere kleinere Verbesserungen wurden bereits von den Mypy-Entwicklern angenommen und sind jetzt offiziell Teil von mypy.

[1] `https://github.com/python/mypy/pull/10191`

Literaturverzeichnis

[1] T. Kohn, G. van Rossum, G. B. Bucher II, Talin, und I. Levkivskyi, „Dynamic Pattern Matching with Python," in *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, Serie DLS 2020. New York, NY, USA: Association for Computing Machinery, 2020, S. 85–98. [Online]. Verfügbar: https://doi.org/10.1145/3426422.3426983

[2] P. G. Salgado, „PEP 619: Python 3.10 Release Schedule," Python Software Foundation, PEP 619, Mai 2020. [Online]. Verfügbar: https://www.python.org/dev/peps/pep-0619/

[3] G. van Rossum, „PEP 3103: A Switch/Case Statement," Python Software Foundation, PEP 3103, Juni 2006. [Online]. Verfügbar: https://www.python.org/dev/peps/pep-3103/

[4] T. Kohn und G. van Rossum, „PEP 635: Structural Pattern Matching: Motivation and Rationale," Python Software Foundation, PEP 635, Sep. 2020. [Online]. Verfügbar: https://www.python.org/dev/peps/pep-0635/

[5] D. F. Moisset, „PEP 636: Structural Pattern Matching: Tutorial," Python Software Foundation, PEP 636, Sep. 2020. [Online]. Verfügbar: https://www.python.org/dev/peps/pep-0636/

[6] M. Beller, R. Bholanath, S. McIntosh, und A. Zaidman, „Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1, 2016, S. 470–481.

[7] „Python 3.10 Documentation — Glossary." [Online]. Verfügbar: https://docs.python.org/3.10/glossary.html

[8] G. van Rossum, J. Lehtosalo, und Łukasz Langa, „PEP 484: Type Hints," Python Software Foundation, PEP 484, Sep. 2014. [Online]. Verfügbar: https://www.python.org/dev/peps/pep-0484/

[9] B. Bucher, D. F. Moisset, T. Kohn, I. Levkivskyi, G. van Rossum, und Talin,

„PEP 622: Structural Pattern Matching," Python Software Foundation, PEP 622, Juni 2020. [Online]. Verfügbar: https://www.python.org/dev/peps/pep-0622/

[10] B. Bucher und G. van Rossum, „PEP 634: Structural Pattern Matching: Specification," Python Software Foundation, PEP 634, Sep. 2020. [Online]. Verfügbar: https://www.python.org/dev/peps/pep-0634/

[11] S. Graf, S. Peyton Jones, und R. G. Scott, „Lower Your Guards: A Compositional Pattern-Match Coverage Checker," *Proc. ACM Program. Lang.*, Vol. 4, Nr. ICFP, Aug. 2020. [Online]. Verfügbar: https://doi.org/10.1145/3408989

[12] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, und C. Jaspan, „Lessons from Building Static Analysis Tools at Google," *Commun. ACM*, Vol. 61, Nr. 4, S. 58–66, Mar. 2018. [Online]. Verfügbar: https://doi.org/10.1145/3188720

[13] „Mypy issue #5818 — Preferred idiom for exhaustiveness checking of unions." [Online]. Verfügbar: https://github.com/python/mypy/issues/5818

[14] „Python 3.10 Documentation — The Python Language Reference — Data Model." [Online]. Verfügbar: https://docs.python.org/3.10/reference/datamodel.html

[15] „Tour of Scala — Pattern Matching." [Online]. Verfügbar: https://docs.scala-lang.org/tour/pattern-matching.html

Erklärung

Hiermit erkläre ich, Adrian Freund, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

Danke

Ich danke dem mypy-Team, insbesondere Jukka Lehtosalo für die Entwicklung und Pflege von mypy, sowie das Review meines Pull-Requests.

Weiterhin danke ich allen Entwicklern und Designern des Python Match-Statements, ohne dessen Arbeit diese Arbeit gar nicht möglich gewesen wäre und insbesondere Brandt Buchner und Guido van Rossum, welche schon früh Feedback zu meinen Designentscheidungen in der Implementierung gegeben haben.

Ich danke auch Pyright Entwickler Eric Traut, für die Überprüfung meiner Evaluationsergebnisse beim Vergleich von mypy und Pyright.