technische universität
dortmund

Masterarbeit

Typebasierte Taint-Analyse im Lambda-Kalkül und die Anwendung auf C

Richard Stewing
1. August 2020

Betreuer:
Prof. Dr. Falk Howar
Malte Mues, M. Sc.

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 14 für Software Engineering
Arbeitsgruppe AQUA
https://ls14-www.cs.tu-dortmund.de/

Zusammenfassung

Diese Arbeit übersetzt die dynamische Taint-Analyse in eine statische Analyse, die mithilfe der Typtheorie realisiert wird. Hierzu wird der Aufbau des einfach getypten Lambda-Kalkül erläutert. Anschließend werden die Regeln für die Typisierung der Taint-Analyse dem einfach getypten Lambda-Kalkül hinzugefügt. Für diese Erweiterung werden die Subjektreduktion und die Vollständigkeit der Analyse gezeigt.

Nachdem die theoretischen Überlegungen abgeschlossen sind, werden die Überlegungen auf C übertragen. Es werden die Hindernisse der Übersetzungen beschrieben und daraus die Einschränkungen der hier beschriebenen Übersetzung abgeleitet.

Die so entstehende Übersetzung wird zur Evaluation auf das Kommandozeilenprogramm *wc* angewendet. Diese Anwendung identifiziert einen Integer-Overflow. Anschließend wird eine Veränderung des Codes vorgenommen, um diesen Fehler zu beheben, sodass die Analyse die Korrektheit des Programms belegen kann. Die so gewonnenen Erkenntnisse führen zu der abschließenden Evaluation, in der die Eigenschaften der Analyse behandelt werden.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Aufgrund der stetig steigenden Menge an Software, die kritische Aufgaben übernimmt, steigen auch die Qualitätsanforderungen an diese Software. In der Industrie steuert Software viele Fertigungsprozesse, führt Qualitätssicherung durch oder entscheidet wie viel Dünger auf die Felder der Landwirte gebracht werden muss [Lia+18]. Diese Entwicklung beschränkt sich aber nicht nur auf die Industrie, auch im Privatleben vieler Menschen übernimmt Technologie, und im speziellen Software, eine immer wichtigere Rolle. Zum Beispiel werden die meisten Überweisungen nicht mehr in einer Filiale, sondern auf Internetportalen, getätigt [BV18].

Ein Nachteil dieser Entwicklung ist, dass mehr Systeme für die Fehler in der Software anfällig sind. Einige dieser Fehler führen zu Sicherheitslücken, über die bekannten Sicherheitslücken wird mit *Common Vulnerabilities and Exposures* (CVE) eine öffentliche Liste von *bekannten* Sicherheitslücken geführt. Aktuell zählt diese Liste 133005 Einträge [Cor20]. Diese Zahl bietet nur eine Größenordnung, da es sich zum einen ausschließlich um bekannte Sicherheitslücken handelt und zum anderen lediglich Sicherheitslücken betrachtet werden. Logische Fehler werden also nicht untersucht, auch wenn diese später zu einer Sicherheitslücke werden können. Die große Anzahl an Sicherheitslücken zeigt bereits, dass das Finden von Sicherheitslücken zunehmend an Wichtigkeit gewinnen muss, um die Software, und damit die Privatsphäre der Menschen, zu schützen.

Sicherheitslücken folgen bestimmten Mustern und können daran in unterschiedliche Gruppen unterteilt werden. Eine solche Unterteilung wird in der *Common Weakness Enumeration* (CWE) präsentiert. Beispielhaft unterscheidet CWE zwischen CWE-378, unsicheren Zugangsberechtigungen[1], und CWE-824, dem Zugriff auf einen nicht initialisierten Zeiger[2]. CWE unterscheidet nicht nur Programmierfehler, wie CWE-378 und -824, sondern auch Verwendungs- und Konfigurationsfehler. Diese unterscheiden sich grundsätzlich von den Fehlern, die in dieser Arbeit behandelt

[1] `https://cwe.mitre.org/data/definitions/378.html`
[2] `https://cwe.mitre.org/data/definitions/824.html`

werden, da sie nicht zur Entwicklungszeit, sondern zur Verwendungszeit der Software auftreten.

Die Schwachstellen, die hier betrachtet werden sollen, folgen dem Muster, dass Eingaben, die aus einer nicht vertrauenswürdigen Quelle stammen, ohne Verifizierung verwendet werden. Beispiele für solche Schwachstellen sind unter anderem CWE-426[3] und CWE-349[4]. Beide beschreiben die Verwendung von externen Daten. Beim CWE-426 handelt es sich um einen Datei-Pfad, der ohne Verifizierung verwendet wird und beim CWE-349 werden verifizierte und nicht verifizierte Daten nicht separat betrachtet. Beide Schwachstellen erlauben es, ein System ohne vorherige Prüfung des Systems zu steuern und bieten damit mögliche Angriffspunkte.

Eine weitere Schwachstelle, die diesem Muster folgt, ist die SQL Injection[5]. Auch hier wird eine externe Eingabe verwendet, um durch Stringkonkatenation eine SQL-Anfrage zu erzeugen. Die Anfrage wird dann, ohne dass die Anfrage oder die Eingabe vorher verifiziert wird, ausgeführt. Im Folgenden wird ein Angriff, der durch eine SQL Injection durchgeführt wird, beispielhaft erläutert, um diese Art von Schwachstelle zu veranschaulichen.

1.1 Ein Beispiel

Die Verwendung von nicht verifizierten Eingaben soll an dieser Stelle mit einem Beispiel präzisiert werden. Dazu wird eine SQL Injection betrachtet, die auf einer Internetseite durchgeführt wird. Bei dieser Internetseite könnte es sich zum Beispiel um einen Shop handeln, der eine Benutzereingabe in Form eines Suchfelds zur Verfügung stellt. Hierfür wird davon ausgegangen, dass der Shop eine SQL-Datenbank mit drei Tabellen verwendet:

- `products`: Beinhaltet alle Produkte mit Preis, Namen und zugehörigen Schlüsselwörtern.
- `user`: Beinhaltet alle registrierten Benutzer mit Namen, Email-Adresse und einer Referenz auf die zugehörige Kreditkarte.
- `creditcards`: Beinhaltet alle Kreditkarten mit dem Namen des Kartenbesitzers, der Kartennummer, Sicherheitscode und Ablaufdatum.

Dieses Suchfeld erlaubt die Eingabe eines Suchwortes, welches dann zum Erstellen einer SQL-Anfrage verwendet wird. Dieses kann durch Pseudocode (Listing 1.1) dargestellt werden.

[3] `https://cwe.mitre.org/data/definitions/426.html`
[4] `https://cwe.mitre.org/data/definitions/349.html`
[5] CWE-89: `https://cwe.mitre.org/data/definitions/89.html`

Input: String *i* mit der Eingabe aus dem Suchfeld
Output: Resultat *r* der Anfrage, die *i* verwendet
1 *q* := SELECT * FROM products WHERE keywords LIKE '%${i}%';
2 *r* := executeQuery(*q*);
3 **return** *r*;

Listing 1.1.: Erzeugung und Auswertung einer SQL-Anfrage aus der Eingabe eines Suchfeldes

Der Pseudocode fügt die Eingabe *i* lediglich in eine SQL-Anfrage ein. Für vorgesehene Suchanfragen entsteht so eine vorgesehene Anfrage. Sucht ein Besucher der Seite zum Beispiel nach „Schuhe", entsteht die Anfrage:

```
SELECT * FROM products WHERE keywords LIKE '%Schuhe%'
```

Es scheint als würden genau die Anfragen erzeugt, die von dem System erwartet werden, daher wird entschieden, dass `executeQuery` keine Verifizierung der Eingabe vornehmen muss. Durch diese Unvorsichtigkeit ist es nun möglich, beliebige SQL-Anfragen an das System zu stellen.

Betrachtet wird hierzu die Eingabe „';SELECT * FROM creditcards --". Diese Eingabe sieht nicht aus wie die Eingaben, die in einem Suchfeld erwartet werden würden. Gleichzeitig ist es aber eine vollkommen korrekte Eingabe.[6] Als Suchanfrage hat diese Eingabe keine Bedeutung, allerdings ist sie aber dahingehend interessant, was passiert, wenn sie in die bereits existierende Anfrage eingebettet wird, wie es vorher mit dem Wort „Schuhe" passiert ist:

```
SELECT * FROM products WHERE keywords LIKE '%';SELECT * FROM creditcards --%'
```

Aus der einen Anfrage gegen die `products`-Tabelle wurden zwei Anfragen. Eine weiterhin gegen die `products`- und eine gegen die `creditcards`-Tabelle. Durch das Ergebnis der zweiten Anfrage erhält der Anfragensteller Zugriff auf alle Kreditkarteninformationen, die bei dem Shop hinterlegt sind.

Die nicht verifizierte Verwendung von Eingaben eines Benutzers, stellt also ein erhebliches Sicherheitsrisiko dar. Diese Eingabe kann also als „verschmutzt" (*eng.* tainted) markiert werden. Diese Verschmutzung überträgt sich dann auf alle Berechnungen, in denen ein verschmutzter Wert verwendet wird. Durch eine explizite Prüfung von `executeQuery` auf diese Markierung kann dann die Sicherheitslücke geschlossen werden. Diese Verfolgung des Taint-Status ist die Grundidee der Taint-Analyse.

[6] Die Suchfeldeingaben wurden *technisch* nicht eingeschränkt, damit ist jede Zeichenkette eine zulässige Eingabe.

1.2 Zielsetzung der Arbeit

Ziel der Arbeit ist es die dynamische Taint-Analyse auf Typen-Ebene statisch darzustellen. Hierzu wird nicht auf die bereits existierenden, statischen Ansätze zur Taint-Analyse aufgebaut, sondern es wird die von Schwartz, Avgerinos und Brumley [SAB10] beschriebene Aufteilung in verschmutzte und nicht verschmutzte Werte auf Typen übertragen. Es wird also von der Datenflussanalyse für die statische Taint-Analyse abgewichen. Der typbasierte Ansatz zur Taint-Analyse wird durch eine Erweiterung des einfach getypten Lambda-Kalküls dargestellt. Ziel ist hierbei, dass die Analyse, die durch den Kalkül beschrieben wird, *vollständig* ist, das heißt, dass alle Fehler bezüglich der Taint-Analyse erkannt werden. *Korrektheit* wird nicht garantiert.

Außerdem soll dieser Kalkül im Rahmen der Programmiersprache C interpretiert und modifiziert werden, um eine Anwendung auf das Programm *wc* zu ermöglichen. Ziel ist hierbei, die Modifikationen, die am Programmcode vorgenommen werden müssen, möglichst gering zu halten. Um dieses Ziel zu erreichen, wird die Sprache C eingeschränkt. Zum Beispiel werden keine Zeiger oder Arrays betrachtet und es wird als Schleife lediglich `while` verwendet.

1.3 Verwandte Arbeiten

Die statische Taint-Analyse wird in der Literatur oft durch eine Datenflussanalyse behandelt [Wan+08; Tri+09; Arz+14]. Eine solche Datenflussanalyse baut einen Graphen von möglichen Pfaden innerhalb eines Programms auf [AC76]. Diese Arbeit versucht den Aufbau dieses Graphen zu vermeiden, indem der relevante Teil des Graphen durch Typen kodiert wird. Der relevante Teil ist dabei, ob sich der Wert aus einer Quelle ergeben hat. Dadurch genügt es den in den Typen kodierten Taint-Status zu betrachten ohne den vollständigen Graphen aufzubauen. Ein Nachteil dieser Konstruktion ist, dass dadurch die Analyse weniger präzise wird, was zu einer Überapproximation der Verschmutzung führen kann. Die Markierungen der Typen entsprechen dabei den Markierungen, die bei der dynamischen Taint-Analyse verwendet werden [SAB10].

Damit orientiert sich dieser Ansatz an der Arbeit von Palsberg und Ørbæk. In [PØ95] nutzen Palsberg und Ørbæk Typen, um damit Vertrauensaussagen über die Werte treffen zu können. Der hier präsentierte Ansatz unterscheidet sich dabei in zwei Aspekten. Zum einen initialisieren Palsberg und Ørbæk den Taint-Status durch spezielle Konstruktoren, die den Taint-Status des jeweiligen Ausdruckes überschreiben. Ein solcher Ansatz wurde hier nicht gewählt, da das Überschreiben des Taint-Status

das Ergebnis einer Analyse erheblich verfälschen kann. Anstelle der dort verwendeten Konstruktoren wurden Konstruktoren für Quellen und Senken eingefügt, die dies nicht erlauben. Allerdings wird auch in dem hier entwickelten Ansatz die Möglichkeit gegeben, den Taint-Status eines Ausdrucks durch Bereinigung zu beeinflussen. Ihr Aufbau macht sie allerdings weniger anfällig für einen Missbrauch, der die Analyse beeinflusst. Zum anderen wird hier der Typansatz weiterentwickelt, in Form von Typ-Vereinfachung. Dies gestattet die komplexen Typen, die während der Analyse entstehen zu vereinfachen, was die Ableitungsregeln verkleinert und die Analyse erleichtert. Auf der Arbeit von Palsberg und Ørbæk bauen weitere Arbeiten der *Information Flow Type Theory* (*dt.* Informationsfluss Typtheorie) auf. Ziel dieser Arbeiten ist es Zugriffsrechte auf verschiedene Datenobjekte in einem System zu kodieren [TZ07; SM03; SV98].

Der hier beschriebene Ansatz wird auf die Programmiersprache C angewendet. Andere Ansätze versuchen ebenfalls das Typsystem von C zu erweitern, um die Sicherheit zu erhöhen [Con+07; And07; Tar+18]. Bei einem dieser Ansätze handelt es sich um *Checked C*[7]. Das Forschungsprojekt von Microsoft™ erweitert dabei C, so dass überprüft wird, ob Zugriffe auf ein Array gültig sind. Außerdem wird ein Typ für sichere Zeiger eingeführt. Diese verifizieren, dass sie valide sind, wenn sie dereferenziert werden. Zusätzlich erlauben sie keine arithmetischen Berechnungen, um umliegende Speicherbereiche zu erreichen. Checked C hat also zum Ziel die Speicherzugriffe zu verifizieren und dadurch die Sicherheit von Programmen zu erhöhen. Der hier vorgestellte Ansatz betrachtet die Daten, die hinter diesen Speicherzugriffen liegen. Durch die Verfolgung des Datenflusses durch die Typen, soll sichergestellt werden, dass diese vor Verwendung in kritischen Bereichen überprüft werden.

[7] `https://github.com/Microsoft/checkedc/wiki`

1.4 Gliederung

Der Aufbau der Arbeit wird im Weiteren beschrieben. Kapitel 2 legt den Grundstein für alle darauf folgenden Betrachtungen und ist hierbei in drei Teile unterteilt. Abschnitt 2.1 erklärt die dynamische Taint-Analyse auf Basis von [SAB10]. Anschließend führen Abschnitt 2.2 und Abschnitt 2.3 in Berechnungsmodelle beziehungsweise Typsysteme im Lambda-Kalkül ein. Kapitel 3 beschreibt ein Typsystem nach den Regeln der Taint-Analyse auf Basis des Lambda-Kalküls. Kapitel 4 übersetzt die in Kapitel 3 entwickelten Regeln nach C (Abschnitt 4.2) und wendet diese dann zum finden einer „unbekannten" Schwachstelle (*eng.* Unknown Vulnerability Detection) auf Unix-Programm *wc* (Abschnitt 4.3) an. Hierfür wird in Abschnitt 4.1 in die in *wc* verwendeten Teile der Programmiersprache C eingeführt. Auf Basis dieser Anwendung werden in Kapitel 5 Rückschlüsse auf die praktische Anwendbarkeit eines solchen Verfahrens gezogen und Schwachstellen und zukünftige Verbesserungsmöglichkeiten identifiziert.

Kapitel 2

Grundlagen

In Abschnitt 1.1 wurde gezeigt, wie die Verwendung der Eingabe eines Nutzers zu einer Sicherheitslücke führen kann. Voraussetzung hierfür ist, dass die Eingabe nicht verifiziert wird. Das in dem Abschnitt erläuterte Beispiel verwendetet die Eingabe, um eine SQL-Anfrage zu erzeugen und somit die Umsetzung einer SQL Injection zu gestatten.

Eine Möglichkeit solche Schwachstellen zu erkennen, ist die Taint-Analyse. Hierfür stehen sich zwei Ansätze gegenüber, die dynamische [SAB10] und die statische Taint-Analyse [Wan+08; Tri+09]. Die statische Taint-Analyse wird dabei häufig in speziellen Domänen umgesetzt. So wenden zum Beispiel Tripp et al. die statische Taint-Analyse auf Webapplikationen an.

In dieser Arbeit soll die statische Taint-Analyse nicht in einer speziellen Domäne behandelt werden. Um die dafür notwendigen Konzepte zu erläutern, werden diese durch die dynamische Taint-Analyse verdeutlicht. Im Anschluss wird in Abschnitt 2.2 das Lambda-Kalkül als Berechnungsmodell, in dem die Taint-Analyse verwendet wird, vorgestellt. Zur Darstellung des Taint-Status, welcher angibt, ob eine Variable durch eine Eingabe „verschmutzt" ist, werden Typen verwendet. Für ein besseres Verständnis der nachfolgenden Ausführungen, wird, durch eine Beschreibung des einfach getypten Lambda-Kalküls in Abschnitt 2.3, in die Typtheorie eingeführt. Zunächst wird die dynamische Taint-Analyse betrachtet.

2.1 Dynamische Taint-Analyse

Die dynamische Taint-Analyse überwacht den Informationsfluss eines Programms zur Laufzeit. Wie in Abschnitt 1.1 beschrieben, können, zum Beispiel durch Nutzereingaben, Variablen „verschmutzt" (*eng.* tainted) werden. Der Ursprung solch einer Verschmutzung wird im Folgenden als *Quelle* bezeichnet. Des Weiteren können

kritische Bereiche identifiziert werden, in denen verschmutzte Variablen Sicherheitslücken bilden, da sie potentiell manipulierte Daten beinhalten. Aus diesem Grund ist in vielen Szenarien ein Abbruch des Programms der Ausführung eines so manipulierten Programms vorzuziehen. In Abschnitt 1.1 wäre `executeQuery` ein solch kritischer Bereich. Im Folgenden werden kritische Bereiche als *Senken* bezeichnet. Die Überwachung zur Laufzeit hat also zum Ziel, den Informationsfluss von Quellen zu Senken zu kontrollieren und sicher zu stellen, dass keine verschmutzte Variable eine Senke erreicht oder das Programm abzubrechen, sollte dies doch geschehen.

Die dynamische Taint-Analyse zeichnet sich insbesondere dadurch aus, dass sie zur Laufzeit arbeitet. Um dies zu erläutern, wird eine einfache Sprache ARIT für arithmetische Ausdrücke entwickelt. Die Sprache wird dann mit zwei unterschiedlichen Semantiken versehen. Bei der ersten handelt es sich um die übliche Interpretation einer solchen Sprache. Die zweite erweitert diese Semantik durch das Nachverfolgen des Taint-Status (*dt.* Verschmutzungszustand).

2.1.1 ARIT

ARIT ist eine einfache Sprache für arithmetische Ausdrücke. Sie beschränkt sich auf Addition und Multiplikation, da diese über den natürlichen Zahlen abgeschlossen sind [Ban90]. Zusätzlich verfügt sie über Primitive für die Ein- und Ausgabe, die später als Quellen und Senken verwendet werden. Außerdem wird das Setzen von Variablen gestattet. Dadurch wird das Aufeinanderfolgen von Aussagen (*eng.* Statements) bedingt.

Es können also drei Komponenten identifiziert werden.

1. Ausdrücke beschreiben arithmetische Berechnungen.
2. Aussagen sind Zuweisungen von Variablen und Ein- und Ausgaben.
3. Sequenzen sind Folgen von Aussagen und Ausdrücken. Die Reihenfolge der Aussagen und Ausdrücke ist für die Auswertung relevant.

Diese drei Komponenten können nun in Form von kontext-freien Grammatiken [Coh86] formuliert werden. Definition 2 baut auf Definition 1 auf und Definition 3 baut auf Definition 2 auf. Alle Definitionen setzen eine Variablenmenge $\mathbf{V}$ voraus.

Definition 1 (Exp) *Exp ist die Grammatik der arithmetischen Berechnungen.*

$$\begin{aligned} exp ::= {} & x \in \mathbf{V} \quad | \\ & n \in \mathbb{N} \quad | \\ & exp \oplus exp \, | \\ & exp \otimes exp \end{aligned}$$

Definition 2 (Statement) *Statement definiert die Sprache der Zuweisungen und von Ein- und Ausgaben.*

$$\begin{aligned} stmt ::= x \in \mathbf{V} := exp \quad & | \\ x \in \mathbf{V} := get\_input() \quad & | \\ set\_output(exp) & \end{aligned}$$

Definition 3 (Sequence) *Sequence definiert eine Folge von Zuweisungen und Ausdrücken.*

$$\begin{aligned} seq ::= & stmt; \quad | \\ & stmt; seq \end{aligned}$$

Durch die so definierte Sprache, können nun Folgen von Berechnungen formuliert werden.

Beispiel 1 ($2 * 3 + 4$) *Die Berechnung von $2*3+4$ kann in eine schrittweise Berechnung gegliedert werden. Sei $\mathbf{V} = \{x, y\}$.*

$$\begin{aligned} x &:= 2 \otimes 3; \\ y &:= x \oplus 4; \\ &set\_output(y); \end{aligned}$$

Außerdem kann die Eingabeanweisung genutzt werden um eine Variable zu belegen.

Beispiel 2 ($2 * a + 4$) *Die Berechnung von $2*a+4$ kann in eine schrittweise Berechnung gegliedert werden. Sei $\mathbf{V} = \{a, x, y\}$.*

$$\begin{aligned} a &:= get\_input(); \\ x &:= 2 \otimes a; \\ y &:= x \oplus 4; \\ &set\_output(y); \end{aligned}$$

Eine einfache Semantik An dieser Stelle soll die Semantik, die durch die übliche Interpretation der verwendeten Symbole definiert wird, formalisiert werden. Hierzu wird eine algebraische Modellierung nach [Pad19] verwendet, die die Syntax von der Semantik trennt.

Die Syntax wird durch eine Signatur dargestellt. Sie beschreibt, wie Terme der Signatur gebildet werden. Sie entspricht dabei der bereits definierten Grammatik. Anstelle der einzelnen Sprachen, werden in dieser Modellierung Symbole, genannt Sorten, verwendet.

Definition 4 (ARIT(V)) *Sei* $\mathbf{V}$ *die Variablenmenge und S Menge der Sorten mit* $S = \{exp, statement, seq\}$*. Die Menge F von Funktionssymbolen entspricht dann der Menge:*

$$\begin{aligned}\{v &: \mathbf{V} \to exp,\\ n &: \mathbb{N} \to exp,\\ \oplus, \otimes &: exp \times exp \to exp,\\ get\_input &: \mathbf{V} \to statement,\\ set\_output &: exp \to statement,\\ := &: \mathbf{V} \times exp \to statement,\\ embed &: statement \to seq,\\ cons &: statement \times seq \to seq,\\ \}.&\end{aligned}$$

Dann gilt $ARIT(\mathbf{V}) = (S, F)$.

Anhand von Definition 4 werden nun Beispiel 1 und Beispiel 2 in die entsprechende Schreibweise übertragen. Sämtliche Operationen werden in Präfixnotation geschrieben, zum Beispiel wird an Stelle von $x := 1$ nun $:= (x, 1)$ geschrieben. Dies ist eine rein syntaktische Veränderung, die nur aus Gründen der Konsistenz beibehalten wird.

Beispiel 3 ($2 * 3 + 4$**)** *Beispiel 1 wie es nach Definition 4 dargestellt werden würde:*

$$\begin{aligned}cons(\ &:= (v(x), \otimes(n(2), n(3))),\\ &cons(\ := (v(y), \oplus(v(x), n(4))),\\ &\quad embed(set\_output(v(y))))).\end{aligned}$$

Die Übersetzung stellt sich so dar, dass die Semikolons, die vorher zur Verkettung von Zuweisungen und Ausdrücken genutzt wurden, durch die Konstruktoren *cons* und

embed ersetzt wurden. Die anderen Operatoren wurden in Präfixnotation übernommen. Zusätzlich wurden die Konstruktoren n und v eingefügt, um natürliche Zahlen und Variablen aus $\mathbf{V}$ in Ausdrücke einzubetten. Durch die gleichen Anpassungen kann Beispiel 2 ebenfalls übersetzt werden.

Beispiel 4 ($2 * a + 4$) *Beispiel 2 wie es nach Definition 4 dargestellt werden würde:*

$$\begin{aligned}&cons(get\_input(a),\\ &\quad cons(:=(v(x), \otimes(n(2), v(a))),\\ &\quad\quad cons(:=(v(y), \oplus(v(x), n(4))),\\ &\quad\quad\quad embed(set\_output(v(y))))))).\end{aligned}$$

Die in Definition 4 formulierte Signatur, kann nun um eine Semantik erweitert werden. Hierbei ist das Vorgehen wie folgt: Zuerst muss für jede Sorte eine Menge definiert werden, für die das Symbol semantisch steht. Diese Menge wird auch Trägermenge genannt. Im zweiten Schritt werden alle Funktionssymbole durch Funktionen interpretiert. Eine Funktion muss dabei so gewählt werden, dass sie mit dem Typen des Funktionssymbols übereinstimmt, wenn die Sorten durch die ihnen zugeordneten Mengen ersetzt werden.

Für die hier vorgestellte Sprache bietet es sich an für alle Sorten die gleiche Trägermenge zu wählen. Für eine Trägermenge ist entscheidend, welche Operationen durch sie ermöglicht werden müssen. In diesem Fall werden folgende Dinge benötigt:

- Es muss eine Variablenbelegung beschrieben werden.
- Es muss Ein- und Ausgabe möglich sein.
- Es muss die Rückgabe von Ergebnissen möglich sein.
- Eine Berechnung muss abgebrochen werden können, falls eine Variable nicht belegt ist.

Im Falle der Variablenbelegung bietet sich eine Funktion von der Variablenmenge $\mathbf{V}$ zu der Vereinigung von $\mathbb{N}$ (inklusive 0) und $* = \{\star\}$ an. Dies lässt zu, dass Variablen keine Belegung haben genau dann, wenn sie auf $\star$ abgebildet werden. Diese Menge wird im Folgenden durch $(\mathbb{N} \cup *)^{\mathbf{V}}$ bezeichnet.

Ein- und Ausgaben lassen sich auf viele Weisen modellieren. An dieser Stelle wird ein Kanal mit den Operationen

$$\text{push} : \mathbf{IO} \times \mathbb{N} \to \mathbf{IO}$$

und

$$\text{pop} : \mathbf{IO} \to \mathbb{N} \times \mathbf{IO}$$

```
1    v(x)(c) = if c = *
2              then *
3              else let (i, f, io) = c in
4                   if f(x) = *
5                   then *
6                   else (f(x), f, io)
```

Listing 2.1.: Definition des v Konstruktors

verwendet. push schreibt hierbei Ausgaben, während pop Eingaben zurück gibt. Die Menge der Kanäle wird hier mit **IO** bezeichnet.

Die Rückgabe von Ergebnissen kann durch eine natürliche Zahl erfolgen. Eine gescheiterte Berechnung kann durch das Element $\star$ bezeichnet werden.

Um diese Eigenschaften zu kombinieren, können diese Mengen, $\mathbb{N}, (\mathbb{N} \cup *)^{\mathbf{V}}, \mathbf{IO}$, durch das kartesische Produkt zusammengebracht werden. Die einzige Ausnahme bildet $*$ zum Abbrechen einer Berechnung. Da sie an Stelle von allen anderen Werten verwendet werden soll, wird sie mit dem Produkt zu $(\mathbb{N} \times (\mathbb{N} \cup *)^{\mathbf{V}} \times \mathbf{IO}) \cup *$ vereinigt. Diese Menge wird als Kontext einer Berechnung interpretiert. Um einen Schritt zu vollziehen, wird eine Abbildung von einem Kontext zu einem anderen Kontext benötigt. Als Trägermenge ergibt sich also die Menge $(\mathbb{N} \times (\mathbb{N} \cup *)^{\mathbf{V}} \times \mathbf{IO}) \cup * \to (\mathbb{N} \times (\mathbb{N} \cup *)^{\mathbf{V}} \times \mathbf{IO}) \cup *$, die $\mathbf{T}$ genannt wird. Im Folgenden wird nun die Semantik der einzelnen Konstruktoren besprochen.

Der Konstruktor v bettet lediglich eine Variable in einen Ausdruck ein. Der Wert einer Variable wird durch ihre Belegung bestimmt. Ist sie unbelegt, führt dies zu einem Abbruch der Berechnung. Der Typ von v ergibt sich zu $\mathbf{V} \to \mathbf{T}$, wenn exp durch die Trägermenge ersetzt wird. Eine mögliche Interpretation für diese Funktion betrachtet den Eingabe-Kontext c und verifiziert, dass kein Abbruch vorliegt. Liegt ein Abbruch vor, wird dieser weitergegeben (vergleiche Zeile 2, Listing 2.1), liegt keiner vor, wird der Wert der Variable mit der Variablenbelegung ermittelt (vergleiche Zeilen 4-6, Listing 2.1). Wenn die Variable nicht belegt ist, führt dies zu einem Abbruch (vergleiche Zeilen 4 und 5, Listing 2.1). Eine Definition könnte zum Beispiel wie Listing 2.1 aussehen. Hier wird der Isomorphismus zwischen C^{B^A} und $C^{(A,B)}$ verwendet. Dies gestattet bereits auf die Parameter (B) der Funktion zuzugreifen, die zurückgeben wird (C^B). In dem hier betrachteten Fall kann auf den zu transformierenden Kontext zugegriffen werden, um den neuen Kontext zu erzeugen. Dieses Prinzip wird für die Semantik aller Konstruktoren verwendet.

Der n Konstruktor arbeitet auf eine ähnliche Weise, allerdings fällt der Fehlerfall der Variablen weg. Dadurch ergibt sich die einfachere Funktion in Listing 2.2.

```
1    n(j)(c) = if c = *
2              then *
3              else let (i, f, io) = c in
4                   (j, f, io)
```

Listing 2.2.: Definition des *n* Konstruktors

```
1    plus(e,e')(c) = if c = *
2                    then *
3                    else let (i, f, io) = c in
4                         if e(c) = * OR e'(c) = *
5                         then *
6                         else let (j, _, _) = e(c)
7                                  (j', _, _) = e'(c) in
8                              (j+j', f, io)
```

Listing 2.3.: Definition von $\oplus$

Die Operatoren $\oplus$ und $\otimes$ arbeiten, bis auf die zugrunde liegende Operation, identisch. Daher wird hier nur der $\oplus$ Operator detailliert betrachtet. Der Typ der beiden Operatoren ist $\mathbf{T} \times \mathbf{T} \to \mathbf{T}$. Es handelt sich dabei also um zwei Programme, die im gleichen Kontext ausgewertet werden müssen (vergleiche Zeilen 6 und 7, Listing 2.3). Zuvor muss überprüft werden, ob eines der beiden Programme bereits die Berechnung abgebrochen hat (vergleiche Zeile 4, Listing 2.3). Ist dies nicht geschehen, werden sie im gleichen Kontext ausgewertet und die Rückgabewerte in `j` und `j'` gespeichert (vergleiche Zeilen 6 und 7, Listing 2.3). Anschließend werden diese durch die Operation, in diesem Fall durch die Addition, kombiniert und mit ihnen ein neuer Kontext erzeugt (vergleiche Zeile 8, Listing 2.3). Dies geschieht in Listing 2.3. Die übrigen Operatoren arbeiten nach den gleichen Prinzipien. Ein vollständige Definition kann Listing 2.4 entnommen werden.

Die hier beschriebene Sprache formalisiert die intuitive Interpretation der arithmetischen Ausdrücke. Der Term aus Beispiel 3

$$\begin{aligned}&\text{cons}(\ :=(v(x), \otimes(n(2), n(3))),\\ &\quad \text{cons}(\ :=(v(y), \oplus(v(x), n(4))),\\ &\quad\quad \text{embed}(\text{set\_output}(v(y)))))\end{aligned}$$

kann durch die oben beschriebene Semantik interpretiert werden. Diese Interpretation entspricht einer Funktion vom Typ der Trägermenge $\mathbf{T}$. Die Interpretation erlaubt also einen Kontext, in einen neuen Kontext zu transformieren. Diese

```
1   get_input(x)(c) = if c = *
2                     then *
3                     else let (i, f, io) = c
4                              (j, io')   = pop(io) in
5                          (j, f[x->j], io')
6
7   set_output(e)(c) = if c = *
8                      then *
9                      else if e(c) = *
10                          then *
11                          else let (i, f, io) = e(c)
12                                   io'       = push(io, i) in
13                               (i, f, io')
14
15  :=(x,e)(c) = if c = *
16               then *
17               else if e(c) = *
18                    then *
19                    else let (i, f, io) = e(c) in
20                         (i, f[x->i], io)
21
22  embed(a)(c) = a(c)
23  cons(a,s)(c) = s(a(c))
```

Listing 2.4.: Definition von den restlichen Operatoren. Es wird davon ausgegangen, dass push und pop immer erfolgreich sind.

Transformation entspricht der Berechnung. So kann der Kontext, in dem keine Variablen belegt werden, der aktuelle Rückgabewert beliebig und der IO-Kanal leer ist, ebenfalls transformiert werden. Dieser Kontext wird dann durch Beispiel 3 zu $(10, [x \to 6, y \to 10], (\epsilon, 10))$ ausgewertet. Der IO-Kanal wurde hier in einen Kanal für die Eingaben und einen für die Ausgaben geteilt. Die Eingaben sind auch am Ende der Berechnung leer, während die Ausgaben durch das Rechenergebnis erweitert wurden.

Was diese Sprache aber für die Taint-Analyse interessant macht, ist die Möglichkeit durch Eingaben das Programm zu beeinflussen. Zum Beispiel würde der Term aus Beispiel 4 in dem Kontext $(0, [], (2, \epsilon))$ zu $(8, [x \to 4, y \to 8], (\epsilon, 8))$ ausgewertet werden.[1] Die Eingabe wurde hier durch die Kanäle simuliert. Das bedeutet aber für die Taint-Analyse, dass alle Ergebnisse, die sich aus der Eingabe ergeben, verschmutzt sind. Die Taint-Analyse versucht nun den Taint-Status nach zu verfolgen. Hierzu soll nun die Semantik so angepasst werden, dass der Taint-Status verfolgt wird und das Programm abbricht, wenn ein verschmutzter Wert in einer Senke verwendet werden soll.

Eine Semantik zur Taint-Analyse In der einfachen Sprache ARIT besteht die Möglichkeit Eingaben zu simulieren. Für die beispielhafte Analyse werden Eingaben als Quellen bezeichnet. Als Senke fungiert die Ausgabe. Die Beschreibung welche Funktionen Quellen und welche Senken sind wird in der Taint-Analyse als Verschmutzungsregeln (*eng.* Taint Policy) bezeichnet. Die Taint Policy kann auch festlegen in welcher Art Verschmutzungen in Berechnungen propagiert werden. In dieser Arbeit wird davon ausgegangen, dass alle Berechnungen jede Verschmutzung weiter propagieren. Die Werte werden zu Paaren erweitert. In [SAB10] wird diese Erweiterung durch eine Abbildung von Variablen auf den Taint-Status realisiert. Die erste Komponente spiegelt weiterhin den Wert wieder, während die zweite Komponente den Taint-Status darstellt. Für den Taint-Status wird der boolesche Wert *T*, wenn der Wert nicht verschmutzt ist, und der boolesche Wert *F*, falls der Wert verschmutzt ist, verwendet.

Daraus ergibt sich beispielsweise die Semantik für den $\oplus$ Operator aus Listing 2.5. Die Semantik verändert sich nur an wenigen Stellen. Zum Beispiel muss der Rückgabewert der Teilausdrücke um den Taint-Status erweitert werden. Bei der Berechnung des neuen Ergebnisses, werden dann die beiden Status so miteinander kombiniert, dass das Ergebnis verschmutzt ist, sobald eines der Zwischenergebnisse verschmutzt ist (vergleiche Zeile 8 Listing 2.5).

Quellen fügen den Taint-Status erstmals hinzu. Dies spiegelt sich dann in der Semantik von *get_input* wieder, wie in Listing 2.6 gezeigt.

[1] [] bezeichnet hier die leere Variablenbelegung

```
plus(e,e')(c) = if c = *
                then *
                else let (i, f, io) = c in
                     if e(c) = * OR e'(c) = *
                     then *
                     else let ((j,t), _, _) = e(c)
                              ((j',t'), _, _) = e'(c) in
                          ((j+j', t AND t'), f, io)
```

Listing 2.5.: Definition von $\oplus$

```
get_input(c) = if c = *
               then *
               else let (i, f, io) = c
                        (j, io')   = pop(io) in
                    ((j, F), f, io')
```

Listing 2.6.: Definition von get_input mit dynamischer Taint-Analyse

Die Semantik einer Senke muss dementsprechend verifizieren, dass sie keine verschmutzte Eingabe verwendet. Dies entspricht einer Überprüfung zur Laufzeit. Für die hier verwendete Sprache wird *set_output* als eine Senke definiert. Die Interpretation von *set_output* muss also zu Listing 2.7 verändert werden. Wird versucht einen verschmutzen Wert auszugeben, bricht das Programm auf diese Weise ab.

Bei der dynamischen Taint-Analyse wird also die Semantik erweitert, um für jeden Wert einen Taint-Status zu verwalten. Die bisher beschriebene Semantik führt ausschließlich zu einem Verteilen der Verschmutzung (*eng.* taint spread). Es müssen aber auch Verschmutzungen bereinigt werden. Der Vorteil der dynamischen Taint-Analyse liegt darin, dass der Taint-Status so auch einfach bereinigt werden kann (*eng.*

```
set_output(e)(c) = if c = *
                   then *
                   else if e(c) = *
                        then *
                        else let ((i, t), f, io) = e(c) in
                             if t
                             then (i, f, push(i,io))
                             else *
```

Listing 2.7.: Definition von set_output mit dynamischer Taint-Analyse

```
set_output(e)(c) = if c = *
                   then *
                   else if e(c) = *
                        then *
                        else let ((i, t), f, io) = e(c) in
                             if t OR i < 100
                             then (i, f, push(i,io))
                             else *
```

Listing 2.8.: Definition von set_output mit dynamischer Taint-Analyse und Sanitization

Sanitization). So kann zum Beispiel eine einfache Überprüfung eingeführt werden, die kontrolliert, ob eine Eingabe verwendet werden kann, ohne das Programm an sich zu verändern. Durch eine solche Überprüfung sind dann Variablen bereinigt und können in kritischen Bereichen verwendet werden.

Hierzu wird der Term

$$\begin{aligned}&\text{cons}(\text{get\_input}(a),\\ &\quad \text{cons}(:=(v(x), \otimes(n(2), v(a))),\\ &\quad\quad \text{cons}(:=(v(y), \oplus(v(x), n(4))),\\ &\quad\quad\quad \text{embed}(\text{set\_output}(v(y))))))\end{aligned}$$

aus Beispiel 4 ein weiteres Mal aufgegriffen. Mit der Definition aus Listing 2.7 führt dieses Programm bei jeder Eingabe zum Absturz. Wird die Definition aber, wie in Listing 2.8 beschrieben, verändert, werden manche Eingaben weiter zugelassen. So wird zum Beispiel $((0, T), [], (2, \epsilon))$ weiterhin ausgewertet und zwar zu $((8, F), [a \to (2, F), x \to (4, F), y \to (8, F)], (\epsilon, 8))$, wohingegen $((0, T), [], (50, \epsilon))$ zu einem Abbruch führt. Weitere Ansätze zur Sanitization bauen auf Invarianten von bestimmten Operatoren auf [Bit20; NS05]. Zu entscheiden wann die Sanitization durchgeführt wird (*eng.* taint sanitization problem) ist eines der schwierigsten Probleme der Taint-Analyse.

Damit wird die Beschreibung der dynamischen Taint-Analyse abgeschlossen. Sie arbeitet zur Laufzeit, um möglichst präzise den Taint-Status zu verfolgen und nutzt Techniken zur Sanitization und bereinigt auf diese Weise den Taint-Status. Diese Bereinigung verhindert, dass sich die Verschmutzung zu weit ausbreitet (*eng.* overtainting).

Diese Arbeit beschreibt eine Taint-Analyse auf der Basis von Typtheorie. Im Folgenden wird etwas Abstand zu der Taint-Analyse genommen. Üblicherweise werden Typsysteme im Lambda-Kalkül betrachtet, was auch der Vorgehensweise dieser

Arbeit entspricht. Dies macht es notwendig, den zugrundeliegenden Kalkül ausführlich zu beschreiben. Hierzu wird in Abschnitt 2.2 in den Lambda-Kalkül eingeführt, während Abschnitt 2.3 auf dieser Basis die Typtheorie betrachtet.

2.2 Lambda-Kalkül

In Abschnitt 2.1 wurden die Grundlagen der dynamischen Taint-Analyse beschrieben. Hierzu wurde eine einfache Sprache entwickelt, die im Anschluss mit einer Semantik versehen wurde, die in der Lage ist, den Taint-Status zu verwalten. Die statische Taint-Analyse kommt ohne eine Veränderung der Semantik aus. Der Ansatz, der in dieser Arbeit vorgestellt wird, versucht den Taint-Status auf Basis der Typen des Programms zu beschreiben. Trotz der Trennung, die auf diese Weise zwischen der Semantik und der Taint-Analyse geschaffen wird, ist ein Berechnungsmodell von Nöten, um die Analyse zu beschreiben. Hierfür wurde sich aus drei Gründen für den Lambda-Kalkül entschieden. Zum einen bietet der Lambda-Kalkül ein einfaches, Turing-vollständiges Berechnungsmodell [Sør06]. Zum anderen arbeitet der Lambda-Kalkül auf die gleiche Weise wie Programmiersprachen. Ein Beispiel für eine solche Sprache ist Haskell[2] [HF92]. Zuletzt ist es in der Literatur üblich Typsysteme mittels unterschiedlicher Varianten des Lambda-Kalküls zu untersuchen [NG14; Pie02; HS86]. Ein Typsystem auf Basis des Lambda-Kalküls wird mit dem einfach-getypten Lambda-Kalkül in Abschnitt 2.3 betrachtet.

In diesem Abschnitt werden die Grundlagen des Lambda-Kalküls entwickelt. Zu Beginn wird dafür die Syntax durch Beispiele erläutert. Die Beispiele werden im weiteren Verlauf auch dazu dienen, die unterschiedlichen Konzepte, die im Anschluss vorgestellt werden, weiter zu verdeutlichen. Diese kleine Einführung in den Lambda-Kalkül baut auf der Arbeit von Sørensen [Sør06] auf.

Der Lambda-Kalkül untersucht Objekte, die λ-Terme genannt werden. Zuvor werden die *Pre*terme betrachtet, welche sich von den gewöhnlichen λ-Termen in der Art unterscheiden, dass Gleichheit zwischen ihnen unterschiedlich betrachtet wird.

Definition 5 (Preterm) *Sei* $\mathbf{V} = \{v_0, \dots\}$ *eine unendliche Variablenmenge. Die Grammatik der Preterme ergibt sich dann durch:*

$$\Lambda^- = x \in \mathbf{V} \mid (\Lambda^-\Lambda^-) \mid (\lambda x \in V.\Lambda^-).$$

Durch diese Grammatik können unterschiedliche Terme gebildet werden.

[2] `https://www.haskell.org`

Beispiel 5 (Preterme) *Sei* $a, b, c, x, y, z \in \mathbf{V}$

$$(\lambda x.(\lambda y.x)) \tag{2.1}$$
$$(\lambda x.(\lambda y.(\lambda z.(x(zy))))) \tag{2.2}$$
$$(\lambda a.(\lambda b.a)) \tag{2.3}$$
$$(\lambda a.(\lambda b.(\lambda c.(a(cb))))) \tag{2.4}$$

Auf einer intuitiven Ebene soll $\lambda x.M$, wenn M ein Preterm ist, eine Funktion darstellen, die x als Eingabeparameter besitzt und deren Rumpf durch M gebildet wird. Im Gegensatz dazu steht (MN), dies soll für die Anwendung einer Funktion stehen. M wäre in diesem Beispiel eine Funktion, deren Eingabeparameter durch N ersetzt werden soll. Es ist aber anzumerken, dass durch diesen Formalismus nicht vorgegeben wird, dass M eine Funktion sein muss. Die Ausführungsrelation, die später definiert wird, führt einen Ausdruck, bei dem dies nicht der Fall ist, nicht weiter aus, ein Preterm dieser Form wird durch den Formalismus aber nicht verhindert.

An diesem Beispiel zeigt sich allerdings bereits eine Schwäche der Preterme. Die Preterme $(\lambda x.(\lambda y.x))$ und $(\lambda a.(\lambda b.a))$ werden als unterschiedlich betrachtet, obwohl sie aus der Sicht einer Berechnung gleich sind. Bevor dies betrachtet wird, sollen aber einige Bezeichnungen erläutert werden.

Wie in Definition 5 veranschaulicht, werden Preterme auf drei unterschiedliche Weisen konstruiert. Ein Preterm, der nur aus einem Element aus $\mathbf{V}$ besteht, wird als Variable bezeichnet. Ein Preterm der Form $\lambda x.M$ wird Abstraktion über x genannt. Der dritte Fall, (MN), wird als Applikation von M auf N bezeichnet. M und N benennen hier beliebige Preterme.

Im Folgenden werden einige vereinfachte Schreibweisen verwendet:

- (KLM) an Stelle von $((KL)M)$
- $(\lambda x.\lambda y.M)$ an Stelle von $(\lambda x.(\lambda y.M))$
- $(\lambda x.MN)$ an Stelle von $(\lambda x.(MN))$
- $(M\lambda x.N)$ an Stelle von $(M(\lambda x.N))$

In den meisten Fällen wird außerdem die äußerste Klammer entfernt.

Nun soll die Anmerkung zu Beispiel 5 betrachtet werden. Sie bezog sich auf die Differenzierung zwischen den Pretermen $(\lambda x.(\lambda y.x))$ und $(\lambda a.(\lambda b.a))$. Die Unterscheidung liegt lediglich in der Benennung der Variablen. Vergleicht man dies mit der Programmierung, entspricht das der Differenzierung, wenn sich der Name eines Eingabeparameters einer Funktion unterscheidet. Aus der Sicht eines Programmierers verhalten sich die beiden Funktionen gleich. Daher wird nun eine Relation definiert,

die alle Preterme in Beziehung setzt, wenn es sich lediglich um eine Variablenumbenennung handelt.

Definition 6 (α-Äquivalenz) *Bezeichne $P[x := Q]$ die Termersetzung nach [Sør06] und $FV(P)$ die Menge freier Variablen in P. Sei die α-Äquivalenz, $=_\alpha$, die kleinste Relation über Λ^-, so dass*

$$\begin{aligned} &P =_\alpha P && \forall P \\ &\lambda x.P =_\alpha \lambda y.P[x := y] && \text{falls } y \notin FV(P) \end{aligned}$$

gilt und $=_\alpha$ unter

$$\begin{aligned} &P =_\alpha P' && \Rightarrow \forall x \in V : \lambda x.P =_\alpha \lambda x.P' \\ &P =_\alpha P' && \Rightarrow \forall Z \in \Lambda^- : PZ =_\alpha P'Z \\ &P =_\alpha P' && \Rightarrow \forall Z \in \Lambda^- : ZP =_\alpha ZP' \\ &P =_\alpha P' && \Rightarrow P' =_\alpha P \\ &P =_\alpha P' \wedge P' =_\alpha P'' && \Rightarrow P =_\alpha P'' \end{aligned}$$

geschlossen ist.

Durch diese Definition ist es nun möglich, „intuitiv gleiche" Preterme als gleich zu betrachten. Beispielsweise kann nun $\lambda x.x =_\alpha \lambda y.y$ geschrieben werden. Da man gewöhnlicherweise $\lambda x.x$ schreibt, wenn man von allen Termen spricht, die unter $=_\alpha$ gleich sind, werden λ-Terme durch ihre Äquivalenzklasse bezüglich $=_\alpha$ interpretiert. Formal bedeutet dies, dass $\lambda x.x$ nun die Menge aller, zu diesem Term α-äquivalenten Terme, bezeichnet. Daraus ergibt sich Definition 7 für λ-Terme.

Definition 7 (λ-Terme) *Sei für $M \in \Lambda^-$ die Äquivalenzklasse $[M]_\alpha$ definiert durch*

$$[M]_\alpha = \{N \in \Lambda^- | M =_\alpha N\}.$$

Die Menge Λ der λ-Terme ergibt sich dann aus

$$\Lambda = \{[M]_\alpha | M \in \Lambda^-\}.$$

Bis zu diesem Zeitpunkt wurde nur die Syntax des Lambda-Kalküls betrachtet. An diesem Punkt soll die Berechnungsfähigkeit dieses Modells betrachtet werden. Berechnungen im Lambda-Kalkül erfolgen durch die Ersetzung von Variablen. Hierzu soll ein Beispiel betrachtet werden.

Beispiel 6 (Variablenersetzung) *Eine Berechnung der Form $(\lambda x.x)N$ sollte dazu führen, dass alle Vorkommen der Variable x im Rumpf der Funktion, in diesem Fall x, ersetzt werden. Man würde also erwarten, dass die Ausführung dieser Berechnung zu dem Ergebnis N führt.*

Die einfache Ersetzung aus diesem Beispiel, soll auf die gleiche Art auch auf komplexere Terme anwendbar sein. Zum Beispiel sollte $(\lambda x.xx)N$ zu (NN) evaluiert werden. Eine solche Variablenersetzung wird nun definiert.

Definition 8 (Variablenersetzung) *Für alle $M, N \in \Lambda$ und $x \in \mathbf{V}$ wird die Ersetzung von x durch N in M geschrieben als $M\{x := N\}$*

$$
\begin{aligned}
x\{x := N\} &= N \\
x\{y := N\} &= x && \text{falls } x \neq y \\
(PQ)\{x := N\} &= P\{x := N\}Q\{x := N\} \\
(\lambda y.P)\{x := N\} &= \lambda y.P\{x := N\} && \text{falls } x \neq y, \text{ mit } y \notin FV(N).
\end{aligned}
$$

Durch die Operation auf Termen, anstelle von Pretermen, ist es möglich, die Abstraktionsvariable so zu wählen, dass die Bedingung erfüllt ist. Dies vereinfacht die Ersetzung im Vergleich zu der von Sørensen [Sør06] beschriebenen Ersetzung auf Pretermen, da weniger Fallunterscheidungen nötig sind.

Um von dieser Variablenersetzung nun zur Ausführung von λ-Termen zu gelangen, ist eine kleine Erweiterung nötig. Diese sollte zwei Dinge zur Folge haben. Zum einen wird die Variablenersetzung angestoßen, das heißt eine Applikation wird mit dem Term in Beziehung gesetzt, der ihrer Ausführung entspricht. In dem Beispiel entspricht dies den Termen $(\lambda x.x)N$ und N, die in eine Beziehung gebracht werden. Zum anderen wird dies an beliebiger Stelle in einem Term erlaubt. Das heißt, wenn an zwei Stellen die Möglichkeit zur Ersetzung gegeben ist, werden beide Ersetzungen gestattet. Zum Beispiel kann $(\lambda x.(\lambda y.y)x)N$ sowohl zu $(\lambda y.y)N$, als auch zu $(\lambda x.x)N$ ausgewertet werden. Auf dem einen Weg wird das x durch N ersetzt, während auf dem anderen Weg y durch x ersetzt wird. Dieses Beispiel dient nur der Veranschaulichung. In der Praxis führen beide Ersetzungen zum gleichen λ-Term, da $(\lambda y.y)N =_\alpha (\lambda x.x)N$. Dieses Verhalten wird durch die β-Reduktion beschrieben.

```
1    flip f a b = f b a
```

Listing 2.9.: Die Haskell-Implementierung der Funktion `flip` aus der Haskell-Standardbibliothek

Definition 9 (β-Reduktion) *Sei die β-Reduktion, $\rightarrow_\beta$, die kleinste Relation über Λ, so dass*

$$(\lambda x.P)Q \rightarrow_\beta P\{x := Q\}$$

gilt und $\rightarrow_\beta$ unter

$$\begin{aligned} P \rightarrow_\beta P' &\Rightarrow \forall x \in \mathbf{V} : \lambda x.P \rightarrow_\beta \lambda x.P' \\ P \rightarrow_\beta P' &\Rightarrow \forall Z \in \Lambda : PZ \rightarrow_\beta P'Z \\ P \rightarrow_\beta P' &\Rightarrow \forall Z \in \Lambda : ZP \rightarrow_\beta ZP'. \end{aligned}$$

geschlossen ist.

Ein Term der Form $(\lambda x.P)Q$ wird β-Redex und ein Term der Form $P\{x := Q\}$ wird β-Kontraktum genannt. Ein Term M ist in β-Normalform, falls kein Term N existiert, so dass $M \rightarrow_\beta N$ gilt. Die mehrschrittige β-Reduktion, $\twoheadrightarrow_\beta$, wird durch den transitiven und reflexiven Abschluss der hier beschriebenen β-Reduktion gebildet. Der Begriff der β-Gleichheit $=_\beta$ bezeichnet Terme als gleich, falls sie zu dem gleichen Term reduzieren. $=_\beta$ wird durch den transitiven, reflexiven und symmetrischen Abschluss von $\rightarrow_\beta$ gebildet.

Zu Beginn wurde auf die starke Beziehung zwischen funktionalen Programmiersprachen und dem Lambda-Kalkül hingewiesen. Mit den hier beschriebenen Grundlagen soll dies nun etwas weiter vertieft werden. Es wird die Implementierung einer einfachen Funktion der Haskell-Standardbibliothek, auch Prelude genannt, betrachtet. Dadurch wird deutlich, dass die beiden Implementierungen sich gleichen. Anschließend wird im Fall von Haskell eine Evaluierung und im Fall des Lambda-Kalküls eine Reduktion durchgeführt, um zu zeigen, dass die beiden Implementierungen zu dem gleichen Ergebnis führen.

Zunächst soll die Haskell-Funktion betrachtet werden. Listing 2.9 und Listing 2.10 zeigen eine Funktion, die drei Parameter nimmt und den ersten Parameter auf den dritten und zweiten anwendet. Eine Funktion `f`, die die Eingabe also in umgekehrter Reihenfolge erwartet, kann so die Parameter in der für sie richtigen Reihenfolge erhalten.

```
1 flip = \f -> \a -> \b -> f b a
```

Listing 2.10.: Die Haskell-Implementierung der Funktion `flip` aus der Haskell-Standardbibliothek

Die Lambda-Kalkül-Implementierung dieser Funktion ähnelt der Haskell-Variante stark. Der entsprechende Lambda-Term lautet $\lambda f.\lambda a.\lambda b.fba$. Einzig die Syntax für das Übergeben von Parametern hat sich verändert. Tatsächlich ist die von Haskell verwendete Syntax ein syntaktischer Zucker, der von Haskell bereitgestellt wird. Haskell formt diesen zu der in Listing 2.10 gezeigten Implementierung um. Der Unterschied, der bleibt, ist die variierende Notation für das λ.[3]

Nicht überraschend können beide Funktionen ausgewertet werden, sodass das Ergebnis in beiden Fällen übereinstimmt. Hierzu soll der Term $(\lambda f.\lambda a.\lambda b.fba)FAB$ betrachtet werden. Durch schrittweise Ersetzung gelangt man zu

$$\begin{aligned}(\lambda f.\lambda a.\lambda b.fba)FAB &\rightarrow_\beta (\lambda a.\lambda b.Fba)AB \\ &\rightarrow_\beta (\lambda b.FbA)B \\ &\rightarrow_\beta FBA.\end{aligned}$$

Im Fall von Haskell wird `flip F A B` ebenfalls zu `F B A` ausgewertet.

Dies schließt die Einführung in den Lambda-Kalkül ab. Der Lambda-Kalkül ist ein Berechnungsmodell, welches sich insbesondere anbietet, wenn es um Konzepte von Programmiersprachen geht. Die starke Entsprechung von Programmiersprachen zu λ-Termen erlaubt das Experimentieren mit diesen Konzepten wie in Programmiersprachen. Dies ist auch der Grund, warum der Lambda-Kalkül der Taint-Analyse, die dieser Arbeit zugrunde liegt, als Berechnungsmodell dient. Die Taint-Analyse dieser Arbeit bedient sich noch einem weiteren Konzept, dem der Typsysteme. Dieses Konzept wird in Abschnitt 2.3 im Weiteren an einem Beispiel, dem einfach getypten Lambda-Kalkül, betrachtet.

2.3 Der einfach getypte Lambda-Kalkül

Die statische Variante der Taint-Analyse versucht den Datenfluss des Programms zur Kompilierzeit zu untersuchen. Hierbei können unterschiedliche Techniken zum Einsatz kommen. Die vorliegende Arbeit adaptiert Techniken aus der Typtheorie, um die Taint-Analyse zu betreiben. Um das nötige Verständnis für diese Techniken zu vermitteln, wird in diesem Teil der Arbeit, eine Einführung in die Typtheorie

[3] Es existieren Spracherweiterungen, die diesen Unterschied ebenfalls beseitigen.

präsentiert. Hierbei wird der in Abschnitt 2.2 betrachtete Lambda-Kalkül durch Regeln zur Typisierung erweitert, die Bedeutung dieser Regeln erläutert und mit Beispielen veranschaulicht. Dieser Abschnitt basiert ebenfalls auf der Arbeit von Sørensen [Sør06].

Durch den Lambda-Kalkül werden Funktionen dargestellt. Tatsächlich sind Funktionen die Konstruktion, die zur Erzeugung aller anderer Datentypen verwendet werden kann. Die Mächtigkeit dieses Modells beschreibt Abschnitt 2.2 bereits als Turing-vollständig. Das bedeutet, dass sämtliche, durch eine Turingmaschine, konstruierbaren Datentypen, auch durch den Lambda-Kalkül dargestellt werden können. Diesen Datentypen werden aber keine Typen zugeordnet. Das bedeutet, dass die Konstruktion von natürlichen Zahlen, Paaren und Summen möglich ist, ihre Verwendung als solche aber erst zur Laufzeit feststeht. Daraus folgt, dass falsch verwendete Konstruktionen zu unerwarteten Ergebnissen führen können. Hierzu ein Beispiel:

Beispiel 7 (Typenlose Konstruktion) *Sei $K = \lambda x.\lambda y.x$, K bezeichnet also intuitiv die Funktion, die immer den ersten Eingabeparameter zurück gibt. Die natürlichen Zahlen können im Lambda-Kalkül dadurch dargestellt werden, dass man zur Darstellung der Zahl n eine Funktion s n-mal auf eine Eingabe z anwendet. Daraus ergibt sich also:*

- *0:* $\lambda s.\lambda z.z$
- *1:* $\lambda s.\lambda z.(sz)$
- *2:* $\lambda s.\lambda z.(s(sz))$
- *n:* $\lambda s.\lambda z.(s^n z)$

mit

$$(s^0 z) = z(s^n z) = (s(s^{n-1} z))$$

Die so kodierten natürlichen Zahlen heißen Church-kodiert. Eine einfache Funktion wie „+1" kann dann durch den Term $\lambda n.\lambda s.\lambda z.s(nsz)$ dargestellt werden. Eine Anwendung dieser Funktion auf den Term K führt aber zu einem unerwarteten Ergebnis:

$$(+1\ K) \rightarrow_\beta \lambda s.\lambda z.ss.$$

Dies stellt wahrscheinlich eine unbeabsichtigte Anwendung dar, die durch den Formalismus nicht verhindert wird. Eine Möglichkeit, solche Anwendungen und ähnliches zu verhindern, sind Typsysteme. Hierbei handelt es sich um eine Menge von Regeln, die die gültige Kombination von Termen beschreiben. Die Typen, die durch diese Regeln entstehen, beschreiben hierbei Eigenschaften des Terms. Durch unterschiedliche Regelsysteme werden unterschiedliche Eigenschaften beschrieben.

An dieser Stelle wird der einfach getypte Lambda-Kalkül beschrieben. Die Eigenschaft, die durch dieses Regelsystem beschrieben wird, ist die Funktionsverkettung. Das heißt, ob die Ausgabe einer Funktion, als Eingabe einer anderen Funktion verwendet werden kann.

Um dies darzustellen werden zwei Konstruktionen benötigt. Typen für Eingabeparameter und Typen für Funktionen. Hier ist zu beachten, dass auch Funktionen selbst Eingaben von Funktionen sein können. Es werden zwar auch Ausgaben von Funktionen modelliert, aber da diese Eingaben von anderen Funktionen sind, werden diese wie Eingabeparameter behandelt. Typen von Funktionen werden aus Paaren gebildet. Daraus wird folgende Notation konstruiert.

Definition 10 (Einfache Typen) *Sei U ein unendlich, abzählbares Alphabet, dessen Element* Typvariablen *genannt werden. Die Menge der einfachen Typen Π wird dann gebildet durch:*

$$\Pi ::= U \mid (\Pi \to \Pi).$$

Im Allgemeinen wird $\to$ als rechts-assoziativ betrachtet.

Des Weiteren wird nun der Begriff eines Kontexts eingeführt. Ein Kontext verwaltet die Typinformationen von Variablen. Er ordnet also einer Variable einen Typen zu.

Definition 11 (Kontext) *Ein Kontext Γ ist eine Menge von Paaren aus $\mathbf{V} \times \Pi$, geschrieben als*

$$\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$$

mit $x_1, \ldots, x_n \in \mathbf{V}$, $x_i \neq x_j$ falls $i \neq j$ und $\tau_1, \ldots, \tau_n \in \Pi$.

Die Domain eines Kontexts, $dom(\Gamma)$ bezeichnet alle Variablen des Kontexts:

$$dom(\Gamma) = \{x_1, \ldots, x_n\}.$$

Der Wertebereich $|\Gamma|$ bezeichnet alle Typen des Kontexts:

$$|\Gamma| = \{\tau_1, \ldots, \tau_n\}.$$

$\{x : \tau\}$ wird durch $x : \tau$ abgekürzt, außerdem ist es erlaubt „Γ, Γ'" für „$\Gamma \cup \Gamma'$" zu schreiben, falls $dom(\Gamma) \cap dom(\Gamma') = \emptyset$ gilt.

Durch diese Definitionen kann nun diskutiert werden, wann ein Term typisierbar ist. Hierbei wird sich auf die oben beschriebene Eigenschaft der Funktionsverkettung fokussiert. Als Bedingung muss hierfür erfüllt sein, dass eine Funktion nur auf ein Argument angewendet werden kann, wenn dieses dem Typen des Eingabeparameters entspricht. Dies entspricht dem, was man in Programmiersprachen wie C erwartet. Ein Parameter x kann einer Funktion f nur übergeben werden, wenn x vom Typen mit dem Eingabetypen von f übereinstimmt.

Zur Typisierung von Termen müssen zwei Dinge beachtet werden. Zum einen müssen Funktionen durch Abstraktionen, wie in Abschnitt 2.2 beschrieben, konstruiert werden. Und zum anderen stellt sich die Frage, welchen Typ eine Variable hat. Die Regeln, die die Typisierung der Terme entscheiden, sollen im Folgenden entwickelt werden. Bevor diese Regeln diskutiert werden, soll aber die verwendete Notation beschrieben und auf unterschiedliche Sichtweisen eingegangen werden.

Eine Regel besteht aus zwei Teilen, die durch einen waagerechten Strich getrennt werden. Oberhalb des Strichs steht die Prämisse, unterhalb dessen die Konklusion [Plo81]. Diese Regeln können auf zwei Arten gelesen werden. Liest man sie von oben nach unten, sagen sie aus, dass aus der Prämisse die Konklusion gefolgert werden kann. Von unten nach oben drücken sie aus, dass es um die Konlusion zu zeigen reicht, die Prämisse zu zeigen. Veranschaulicht wird dies durch folgendes Beispiel:

Beispiel 8

$$\frac{\alpha \wedge \beta}{\alpha} \ (\wedge E)$$

Wie oben beschrieben, kann diese Regel auf zwei Weisen verstanden werden:

1. *Um α zu zeigen, reicht es $\alpha \wedge \beta$ zu zeigen (unten nach oben).*
2. *Wenn $\alpha \wedge \beta$ bekannt ist, dann kann daraus α gefolgert werden (oben nach unten).*

Üblicherweise wird außerdem der Name der Regeln neben den Strich geschrieben.

Eine weitere Notation, die erläutert werden soll, ist die „typt" Relation. Sie setzt einen Kontext in Beziehung zu einem Term und einem Typ. Geschrieben wird sie als $\Gamma \vdash M : \tau$. Die Konstruktion dieser Relation entspricht genau den Ableitungsregeln, die nun entwickelt werden.

Eine Form von Termen, die mit einem Typen versehen werden müssen, sind Variablen. Hierbei kann eine Variable nur mit einem Typen versehen werden, wenn sie bereits im Kontext einen Typen besitzt. Die einfachste Veranschaulichung kann durch Listing 2.11 erfolgen. Der Typ von x kann ab dem Zeitpunkt ermittelt werden, wenn x

```
1    int var_example(int a) {
2      int x = a * 2;
3      return y;
4    }
```

Listing 2.11.: Eine einfache C-Funktion. Sie veranschaulicht, dass eine Variable nur mit einem Typen versehen werden kann, wenn sie sich im Kontext befindet.

sich im Kontext befindet. x wird durch `int x = a * 2` in den Kontext aufgenommen. Dadurch, dass y nie in den Kontext aufgenommen wurde, kann für diese Variable auch kein Typ ermittelt werden. Daraus ergibt sich also, dass man einen Typen τ für ein x ableiten kann, geschrieben als $x : \tau$, genau dann, wenn x den Typen τ in dem aktuellen Kontext Γ besitzt. Für eine Ableitungsregel ergibt sich also Definition 12.

Definition 12 (Ableitungsregel Ax)

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \ (Ax)$$

Eine weitere Art von Term, die untersucht werden muss, ist die Applikation. Diese Art Term kombiniert zwei Terme in der Art, dass eine Funktion auf einen anderen Term angewendet wird. Daraus ergeben sich zwei Anforderungen. Sei der Term, der untersucht wird, MN. Erstens muss der Term M eine Funktion sein, das heißt M muss einen Funktionstypen besitzen. In der vorher beschriebenen Notation bedeutet das, dass M einen Typen $\sigma \to \tau$ haben muss. Des Weiteren muss N den Typen der Eingabe von M haben, ansonsten soll diese Eingabe nicht zulässig sein. N hat also den Typen σ und die Anwendung muss zu dem Rückgabetypen von M, τ, führen. All diese Bedingungen arbeiten immer in einem Kontext und da keine Operation den Kontext manipuliert, wird erwartet, dass die Typen im gleichen Kontext gelten. Dieser Sachverhalt kann zu der Ableitungsregel App kombiniert werden:

Definition 13 (Ableitungsregel App)

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \ (App)$$

Bei der letzten Art von Term handelt es sich um die Abstraktion. Sie erzeugt die Funktionstypen und führt Veränderungen am Kontext durch. Hierzu soll wieder Listing 2.11 betrachtet werden. Die Funktion bekommt den Parameter a durch `int example_var(int a)` übergeben. In dem Rumpf der Funktion befindet sich nun

also ein `a` vom Typen `int` im Kontext. Der Lambda-Kalkül arbeitet auf eine ähnliche Weise, wenn $\lambda x.M$ ein Typ zugeordnet werden soll, wird M in einem Kontext betrachtet, der ein x einschließt. Der Typ von x wird dabei so gewählt, dass er mit dem Eingabeparameter der Funktion übereinstimmt. Eine Regel, die diese Bedingung erfüllt, kann wie in Definition 14 definiert werden.

Definition 14 (Ableitungsregel `Abs`)

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \ (Abs)$$

Die hier beschriebenen Regeln sind ein System, das von Curry [Cur34] entwickelt wurde. Es existiert ein ähnliches System, entwickelt von Church [Chu40]. Ein Unterschied liegt in der Art der Regelformulierung. Church formulierte seine Regeln durch die Konstruktion seiner Terme. Die Konstruktion, die Curry vorschlug, wählt aus der Menge der λ-Terme, die λ-Terme aus, für die ein Typ abgeleitet werden kann. Die Ableitung dieser Typen wird im Folgenden betrachtet.

Die Ableitungen werden durch das wiederholte Anwenden der Regeln gebildet. In dem vorliegenden Kalkül wird der so entstehende Ableitungsbaum vollständig durch den Term bestimmt. Formal wird diese Eigenschaft das Generationenlemma (*eng.* Generation Lemma, Theorem 5) genannt [Sør06]. Anhand des hier präsentierten Ableitungsbaums wird außerdem auf die Unifikation, die der Typ-Rekonstruktion zugrunde liegt, eingegangen.

Beispiel 9 (Ableitungsbaum) *Der Term $\lambda x.\lambda y.\lambda z.xz(yz)$ soll im leerem Kontext betrachtet werden. Es soll also $\emptyset \vdash \lambda x.\lambda y.\lambda z.xz(yz) : \tau$ gezeigt werden. Für eine bessere Übersicht wird der Ableitungsbaum in drei Ableitungsbäume unterteilt.*
Beweisbaum 1:

$$\dfrac{\dfrac{}{x : \alpha, y : \beta, z : \gamma \vdash z : \gamma}\ (Ax) \qquad \dfrac{}{x : \alpha, y : \beta, z : \gamma \vdash xz : \alpha}\ (Ax)\ mit\ \alpha = \gamma \to \tau_4 \to \tau_3}{x : \alpha, y : \beta, z : \gamma \vdash xz : \tau_4 \to \tau_3}\ (App)$$

Beweisbaum 2:

$$\dfrac{\dfrac{}{x : \alpha, y : \beta, z : \gamma \vdash z : \gamma}\ (Ax) \qquad \dfrac{}{x : \alpha, y : \beta, z : \gamma \vdash y : \beta}\ (Ax)\ mit\ \beta = \gamma \to \tau_4}{x : \alpha, y : \beta, z : \gamma \vdash (yz) : \tau_4}\ (App)$$

Beweisbaum 3:

$$\dfrac{\dfrac{\dfrac{\dfrac{\textit{Beweisbaum 1} \qquad \textit{Beweisbaum 2}}{x : \alpha, y : \beta, z : \gamma \vdash xz(yz) : \tau_3}\ (Abs)\ mit\ \tau_2 = \gamma \to \tau_3}{x : \alpha, y : \beta \vdash \lambda z.xz(yz) : \tau_2}\ (Abs)\ mit\ \tau_1 = \beta \to \tau_2}{x : \alpha \vdash \lambda y.\lambda z.xz(yz) : \tau_1}\ (Abs)\ mit\ \tau = \alpha \to \tau_1}{\emptyset \vdash \lambda x.\lambda y.\lambda z.xz(yz) : \tau}\ (Abs)$$

Es zeigt sich also, dass eine Ableitung für den Term existiert. Hierbei gibt es aber eine weitere Voraussetzung. An den Ableitungsregeln findet man eine Reihe von Gleichungen, die sicherstellen, dass die Regeln anwendbar sind.

$$\begin{aligned} \tau &= \alpha \to \tau_1 \\ \tau_1 &= \beta \to \tau_2 \\ \tau_2 &= \gamma \to \tau_3 \\ \beta &= \gamma \to \tau_4 \\ \alpha &= \gamma \to \tau_4 \to \tau_3 \end{aligned}$$

Die Unifikation beschäftigt sich mit dem Lösen dieses Gleichungssystems. Das Lösen dieses Gleichungssystems geschieht durch Einsetzen in die unterschiedlichen Gleichungen. Ein Gleichungssystem ist hierbei unlösbar, bzw. enthält einen Widerspruch, wenn eine Gleichung der Form

$$\alpha = \alpha \to \beta \qquad \textit{oder}$$
$$\alpha = \beta \to \alpha$$

entsteht. In dem vorliegenden Gleichungssystem erhält man durch wiederholtes Einsetzen:

$$\tau = (\gamma \to \tau_4 \to \tau_3) \to (\gamma \to \tau_4) \to \gamma \to \tau_3.$$

Ein vollständiger Algorithmus zur Typ-Rekonstruktion, wie die hier beschriebene Herleitung eines Typen auch genannt wird, kann ebenfalls [Sør06] entnommen werden.

Bevor diese Einführung in die Typtheorie mittels des einfach getypten Lambda-Kalküls abgeschlossen wird, soll noch eine wichtige Eigenschaft dieses Kalküls hervorgehoben werden. Die Eigenschaft beschäftigt sich mit dem Zusammenhang zwischen Typen und Reduktion eines Terms. Stellt man sich die Terme als die Programme des Lambda-Kalküls vor, dann entspricht die Ausführung der Reduktion. Eine intuitive Erwartung an ein Programm ist, dass sich der Typ eines Programms durch Ausführung nicht ändert. Dies soll an dieser Stelle verifiziert werden. Hierzu wird zuallererst, die entsprechende Eigenschaft formal beschrieben.

Theorem 1 (Subjektreduktion) *Geben* $\Gamma \vdash M : \tau$ *und* $M \to_\beta N$ *kann gefolgert werden, dass* $\Gamma \vdash N : \tau$.

Das Theorem 1 beschreibt, dass wenn M einen Typ hat und M zu N reduziert wird, N den gleichen Typ hat, wie M ihn hatte. Um dieses Theorem zu beweisen, wird das Prinzip der Induktion verwendet. Die Induktion läuft über die Konstruktion von $\to_\beta$.

Beweis 1 (Subjektreduktion)
Induktionsanfang: Sei $M = (\lambda x.P)Q$ *und* $N = P\{x := Q\}$.

$$\frac{\dfrac{\Gamma, x : \sigma \vdash P : \tau}{\Gamma \vdash \lambda x.P : \sigma \to \tau}\ (Abs) \qquad \Gamma \vdash Q : \sigma}{\Gamma \vdash (\lambda x.P)Q : \tau}\ (App)$$

Es ist also zu zeigen, dass $\Gamma \vdash N : \tau$ *unter der Annahme, dass* $\Gamma, x : \sigma \vdash P : \tau$ *und* $\Gamma \vdash Q : \sigma$. *Dies entspricht genau dem Theorem 7 [Sør06].*

Induktionsschritt 1 : Sei $P \to_\beta P'$ *mit* $\Gamma, x : \sigma \vdash P : \tau$ *und* $\Gamma, x : \sigma \vdash P' : \tau$, $M = \lambda x.P$ *und* $N = \lambda x.P'$ *für beliebige* $x \in \mathbf{V}$.

$$\frac{\Gamma, x : \sigma \vdash P : \tau}{\Gamma \vdash \lambda x.P : \sigma \to \tau}\ (Abs)$$

In dem gleichen Kontext Γ *ist ebenfalls zu zeigen, dass* $\lambda x.P'$ *den Typ* $\sigma \to \tau$ *hat:*

$$\frac{\Gamma, x : \sigma \vdash P' : \tau}{\Gamma \vdash \lambda x.P' : \sigma \to \tau}\ (Abs)$$

Induktionsschritt 2: Sei $P \to_\beta P'$ *mit* $\Gamma \vdash P : \sigma \to \tau$ *und* $\Gamma \vdash P' : \sigma \to \tau$, $M = PZ$ *und* $N = P'Z$ *für beliebige* $Z \in \Lambda$ *mit* $\Gamma \vdash Z : \sigma$. *In beiden Fällen kann der Typ* τ *abgeleitet werden. Für* PZ:

$$\frac{\Gamma \vdash P : \sigma \to \tau \qquad \Gamma \vdash Z : \sigma}{\Gamma \vdash PZ : \tau}\ (App)$$

Für $P'Z$:

$$\frac{\Gamma \vdash P' : \sigma \to \tau \qquad \Gamma \vdash Z : \sigma}{\Gamma \vdash P'Z : \tau}\ (App)$$

Induktionsschritt 3: Sei $P \rightarrow_\beta P'$ *mit* $\Gamma \vdash P : \sigma$ *und* $\Gamma \vdash P' : \sigma$, $M = ZP$ *und* $N = ZP'$ *für beliebige* $Z \in \Lambda$ *mit* $\Gamma \vdash Z : \sigma \rightarrow \tau$. *In beiden Fällen kann der Typ* τ *abgeleitet werden. Für* ZP*:*

$$\frac{\Gamma \vdash Z : \sigma \rightarrow \tau \qquad \Gamma \vdash P : \sigma}{\Gamma \vdash ZP : \tau} \ (App)$$

Für ZP'*:*

$$\frac{\Gamma \vdash Z : \sigma \rightarrow \tau \qquad \Gamma \vdash P' : \sigma}{\Gamma \vdash ZP' : \tau} \ (App)$$

Damit ist gezeigt, dass die Ausführung eines Programms im Lambda-Kalkül, den Typen nicht verändert und erlaubt Typen als Invarianten eines Programms zu interpretieren. Das heißt, Typen sind Eigenschaften eines Programms, die sich nicht durch Ausführung verändern.

Mit diesem Teil der Arbeit, liegt das theoretische Fundament auf dem alle folgenden Überlegungen aufbauen. Damit wäre es an dieser Stelle möglich, die Taint-Analyse auf Basis von Typen zu beschreiben. Dabei baut das System auf ähnlichen Konzepten auf wie die dynamische Taint-Analyse. Zur Verfolgung des Taint-Status werden, anstelle semantischer Veränderungen der Werte, aber Markierungen der Typen verwendet. Das so aufgebaute System wird im folgenden Kapitel 3 beschrieben.

Kapitel 3

Typisierung einer Taint-Analyse

Die Taint-Analyse, die in Abschnitt 2.1 beschrieben wurde, basiert auf einer Analyse zur Laufzeit. Hierzu wird die Semantik des Programms verändert. Die Kernidee dabei ist die Werte zu verändern, indem eine Komponente, welche den Taint-Status repräsentiert, hinzugefügt wird. Der Taint-Status wird dabei bei Berechnungen weitergegeben. Eine verschmutzte Variable hat zur Folge, dass die gesamte Berechnung verschmutzt ist. Zusätzlich wird bei der Verwendung eines Ergebnisses in einem kritischen Bereich, dessen Status überprüft. Dies bedingt zusätzliche Berechnungen zur Laufzeit. Durch eine statische Analyse auf Basis von Typen, können diese Berechnungen zur Übersetzungszeit vorgenommen werden. Dadurch kann die Berechnung im Vergleich zur dynamischen Taint-Analyse effizienter durchgeführt werden, während trotzdem sichergestellt wird, dass keine verschmutzten Berechnungen in einem kritischen Bereich verwendet werden.

Im Folgenden wird eine Taint-Analyse im Lambda-Kalkül beschrieben. Hierzu wird in Abschnitt 3.1 eine Syntax entwickelt, welche sich an dem Lambda-Kalkül orientiert. Zudem wird die entsprechende β-Relation definiert. Auf Basis dieser Syntax werden dann Ableitungsregeln in Abschnitt 3.2 beschrieben, welche anhand von Beispielen in der Programmiersprache C veranschaulicht werden. Der so entstehende Kalkül wird $\tilde{\lambda}$-Kalkül genannt. Analog zum einfach getypten Lambda-Kalkül wird darüber hinaus eine Adaption der Subjektreduktion gezeigt.

3.1 Erweiterung des Lambda-Kalküls zur Taint-Analyse

In diesem Teil der Arbeit wird das Berechnungsmodell aus Abschnitt 2.2 um Aspekte der Taint-Analyse erweitert. Zu diesem Zweck, wird wie folgt vorgegangen: Analog zu Abschnitt 2.2 werden zuerst die Preterme entwickelt, die dann durch eine Äquivalenzrelation identifiziert werden. Auf den so entstehenden Termen wird eine

Substitutionsfunktion definiert, die auf die gleiche Weise arbeitet wie im Abschnitt 2.2. Abgeschlossen wird dieser Abschnitt durch eine Reihe von Beispielen, die in Abschnitt 3.2 durch die statische Taint-Analyse untersucht werden.

Die Hauptaufgabe der Taint-Analyse sieht das Verfolgen des Datenflusses von den Quellen zu den Senken vor. Es sollte also durch die Terme markiert werden, wo Quellen und wo Senken zu finden sind. Hierzu werden zwei Konstruktoren für die Preterme hinzugefügt. Sie repräsentieren wie Abstraktionen durch λ, Funktionen, allerdings erwarten diese einen bestimmten Taint-Status als Eingabe oder geben einen Wert mit einem bestimmten Taint-Status zurück. Diese Konstruktoren werden mit I und Z bezeichnet. I erwartet, dass die Eingabe als nicht verschmutzt erkannt wurde und Z gibt immer einen verschmutzten Wert zurück. I bezeichnet also Senken und Z Quellen.

Eine weitere Aufgabe der Taint-Analyse ist die Bereinigung von Variablen und Berechnungen. Die dynamische Taint-Analyse ist in der Lage, dies durch Inspektion der Werte zu erreichen. Das gleiche Vorgehen ist zur Übersetzungszeit nicht möglich. Es wird also ein entsprechendes Primitiv eingeführt, der sogenannte Δ-*Check-Point*. An einem solchen Punkt kann eine Berechnung auf ihre Reinheit untersucht werden. Abhängig davon wird dann einer von zwei Termen gewählt, der als Funktionsrumpf verwendet wird. Die Struktur ähnelt einer `if`-Verzweigung, da eine von zwei Möglichkeiten zur Ausführung gewählt wird. Aus diesen Vorüberlegungen ergibt sich dann Definition 15.

Definition 15 ($\tilde{\lambda}$-Preterm) *Sei* $\mathbf{V} = \{v_0, \dots\}$ *eine unendliche Variablenmenge. Die Grammatik der* $\tilde{\lambda}$*-Preterme ergibt sich dann durch:*

$$\tilde{\Lambda}^- = x \in \mathbf{V} \mid (\tilde{\Lambda}^- \tilde{\Lambda}^-) \mid (\lambda x \in V.\tilde{\Lambda}^-) \mid (\mathrm{I}x \in V.\tilde{\Lambda}^-) \mid (\mathrm{Z}x \in V.\tilde{\Lambda}^-) \mid (\Delta x \in \mathbf{V}.\tilde{\Lambda}^- \tilde{\Lambda}^-).$$

Die Bezeichnungen der Terme für den Lambda-Kalkül werden an dieser Stelle erweitert. Die Abstraktion über x wird je nach verwendetem Konstruktor λ-, I- oder Z-Abstraktion genannt. Ein Term der Form $\Delta x.PQ$ wird Δ-Check-Point genannt. Es werden außerdem die gleichen abkürzenden Schreibweisen wie für Lambda-Terme verwendet.

Die $\tilde{\lambda}$-Preterme haben die gleiche Eigenschaft wie die Preterme des üblichen Lambda-Kalküls. Im Folgenden wird daher gezeigt, wie die α-Äquivalenz für $\tilde{\lambda}$-Preterme adaptiert werden kann. Hierzu muss zuerst die Variablenersetzung auf Pretermen zu $\tilde{\lambda}$-Pretermen übersetzt werden [Sør06].

Definition 16 (Variablenersetzung auf $\tilde{\lambda}$-Pretermen) *Seien $M, N \in \tilde{\Lambda}^{-}$, $x, y \in \mathbf{V}$ mit $x \neq y$ und $FV(M)$ die freien Variablen des Terms M. Die Substitution von x in M durch N geschrieben $M[x := N]$ wird dann definiert durch:*

$$
\begin{array}{lll}
x[x := N] & = N & \\
y[x := N] & = y & \\
 & & \\
(PQ)[x := N] & = P[x := N]Q[x := N] & \\
 & & \\
(\lambda x.P)[x := N] & = \lambda x.P & \\
(\lambda y.P)[x := N] & = \lambda y.P[x := N], & \textit{falls } y \notin FV(N) \vee x \notin FV(P) \\
(\lambda y.P)[x := N] & = \lambda z.P[y := z][x := N], & \textit{falls } y \in FV(N) \wedge x \in FV(P) \\
 & & \\
(\mathrm{I}x.P)[x := N] & = \mathrm{I}x.P & \\
(\mathrm{I}y.P)[x := N] & = \mathrm{I}y.P[x := N], & \textit{falls } y \notin FV(N) \vee x \notin FV(P) \\
(\mathrm{I}y.P)[x := N] & = \mathrm{I}z.P[y := z][x := N], & \textit{falls } y \in FV(N) \wedge x \in FV(P) \\
 & & \\
(\mathrm{Z}x.P)[x := N] & = \mathrm{Z}x.P & \\
(\mathrm{Z}y.P)[x := N] & = \mathrm{Z}y.P[x := N], & \textit{falls } y \notin FV(N) \vee x \notin FV(P) \\
(\mathrm{Z}y.P)[x := N] & = \mathrm{Z}z.P[y := z][x := N], & \textit{falls } y \in FV(N) \wedge x \in FV(P) \\
 & & \\
(\Delta x.QR)[x := N] & = \Delta x.QR & \\
(\Delta y.QR)[x := N] & = \Delta x.Q'R', & \textit{falls } y \notin FV(N) \vee \\
 & & (x \notin FV(Q) \wedge x \notin FV(R)) \\
 & & \textit{mit } Q' = Q[x := N] \\
 & & \textit{und } R' = R[x := N] \\
(\Delta y.QR)[x := N] & = \Delta z.Q'R', & \textit{falls } y \in FV(N) \wedge \\
 & & (x \in FV(Q) \vee x \in FV(R)) \\
 & & \textit{mit } Q' = Q[y := z][x := N] \\
 & & \textit{und } R' = R[y := z][x := N].
\end{array}
$$

Da sich I- und Z-Abstraktionen von der üblichen λ-Abstraktion zur Ausführungszeit nicht unterscheiden, verläuft die Substitution auf ihnen ganzheitlich parallel. Die Substitution auf einem Δ-Check-Point hingegen unterscheidet sich geringfügig. Die Substitution muss auf zwei Funktionsrümpfe angewendet werden und entsprechend muss ebenfalls die logische Formel zur Fallunterscheidung angepasst werden.

Die Übersetzung der α-Äquivalenz zur $\tilde{\alpha}$-Äquivalenz ergibt sich dann auf eine ähnliche Weise:

Definition 17 ($\tilde{\alpha}$-Äquivalenz) *Bezeichne $FV(P)$ die Menge freier Variablen in P. Sei die $\tilde{\alpha}$-Äquivalenz, $=_{\tilde{\alpha}}$, die kleinste Relation über $\tilde{\Lambda}^-$, so dass*

$$
\begin{aligned}
&P =_{\tilde{\alpha}} P && \forall P \\
&\lambda x.P =_{\tilde{\alpha}} \lambda y.P[x := y] && \text{falls } y \notin FV(P) \\
&\mathrm{I}x.P =_{\tilde{\alpha}} \mathrm{I}y.P[x := y] && \text{falls } y \notin FV(P) \\
&\mathrm{Z}x.P =_{\tilde{\alpha}} \mathrm{Z}y.P[x := y] && \text{falls } y \notin FV(P) \\
&\Delta x.MN =_{\tilde{\alpha}} \Delta y.M[x := y]N[x := y] && \text{falls } y \notin FV(M) \wedge y \notin FV(N)
\end{aligned}
$$

gilt und $=_{\tilde{\alpha}}$ unter

$$
\begin{aligned}
&P =_{\tilde{\alpha}} P' && \Rightarrow \forall x \in V : \lambda x.P =_{\tilde{\alpha}} \lambda x.P' \\
&P =_{\tilde{\alpha}} P' && \Rightarrow \forall x \in V : \mathrm{I}x.P =_{\tilde{\alpha}} \mathrm{I}x.P' \\
&P =_{\tilde{\alpha}} P' && \Rightarrow \forall x \in V : \mathrm{Z}x.P =_{\tilde{\alpha}} \mathrm{Z}x.P' \\
\\
&P =_{\tilde{\alpha}} P' && \Rightarrow \forall x \in V, \forall Z \in \tilde{\Lambda}^- : \Delta x.PZ =_{\tilde{\alpha}} \Delta x.P'Z \\
&P =_{\tilde{\alpha}} P' && \Rightarrow \forall x \in V, \forall Z \in \tilde{\Lambda}^- : \Delta x.ZP =_{\tilde{\alpha}} \Delta x.ZP' \\
\\
&P =_{\tilde{\alpha}} P' && \Rightarrow \forall Z \in \tilde{\Lambda}^- : PZ =_{\tilde{\alpha}} P'Z \\
&P =_{\tilde{\alpha}} P' && \Rightarrow \forall Z \in \tilde{\Lambda}^- : ZP =_{\tilde{\alpha}} ZP' \\
\\
&P =_{\tilde{\alpha}} P' && \Rightarrow P' =_{\tilde{\alpha}} P \\
&P =_{\tilde{\alpha}} P' \wedge P' =_{\tilde{\alpha}} P'' && \Rightarrow P =_{\tilde{\alpha}} P''
\end{aligned}
$$

geschlossen ist.

Eine intuitive Interpretation dieser Definition bedeutet, dass durch Abstraktionen gebundene Variablen beliebig umbenannt werden können. Dies schließt auch die Abstraktion, die in einem Δ-Check-Point versteckt ist, mit ein. Die Übersetzung von λ-Termen ergibt sich dann analog:

Definition 18 ($\tilde{\lambda}$-Terme) *Sei für $M \in \tilde{\Lambda}^-$ die Äquivalenzklasse $[M]_{\tilde{\alpha}}$ definiert durch*

$$[M]_{\tilde{\alpha}} = \{N \in \tilde{\Lambda}^- | M =_{\tilde{\alpha}} N\}.$$

Die Menge $\tilde{\Lambda}$ der $\tilde{\lambda}$-Terme ergibt sich dann aus

$$\tilde{\Lambda} = \{[M]_{\tilde{\alpha}} | M \in \tilde{\Lambda}^-\}.$$

Von nun an werden die Überlegungen auf die Menge der $\tilde{\lambda}$-Terme fokussiert. Zur Definition einer Variante der β-Reduktion wird nun eine Übersetzung der Variablensubstitution auf Termen benötigt. Hierbei werden die Abstraktionen wieder auf die gleiche Weise behandelt. Die Ersetzung auf einem Δ-Check-Point wird hierbei wie die auf einer Abstraktion behandelt, allerdings mit dem Unterschied, dass in beiden Rümpfen ersetzt wird.

Definition 19 (Variablenersetzung auf $\tilde{\lambda}$-Termen) *Für alle $M, N \in \tilde{\Lambda}$ und $x \in \mathbf{V}$ wird die Ersetzung von x durch N in M geschrieben als $M\{x := N\}$*

$$
\begin{array}{lll}
x\{x := N\} & = N & \\
x\{y := N\} & = x & \textit{falls } x \neq y \\
(PQ)\{x := N\} & = P\{x := N\}Q\{x := N\} & \\
(\lambda y.P)\{x := N\} & = \lambda y.P\{x := N\} & \textit{falls } x \neq y, \textit{mit } y \notin FV(N) \\
(\mathrm{I}y.P)\{x := N\} & = \mathrm{I}y.P\{x := N\} & \textit{falls } x \neq y, \textit{mit } y \notin FV(N) \\
(\mathrm{Z}y.P)\{x := N\} & = \mathrm{Z}y.P\{x := N\} & \textit{falls } x \neq y, \textit{mit } y \notin FV(N) \\
(\Delta y.QR)\{x := N\} & = \Delta y.Q'R' & \textit{falls } x \neq y, \textit{mit } y \notin FV(N) \\
 & & \textit{und } Q' = Q\{x := N\} \\
 & & \textit{und } R' = R\{x := N\}.
\end{array}
$$

Durch Definition 19 ist es nun auch möglich die β-Reduktion zu übersetzen. Die Konstruktion dieser Relation wirft allerdings ein Problem auf. Die Reduktion eines Δ-Check-Points hängt von der Semantik ab, also was es für einen Wert bedeutet „rein" zu sein. An dieser Stelle kann das aber nicht bestimmt werden, da dies domänenspezifisch ist. Aus dem Grund werden immer beide „Wege" als mögliche Reduktion bestimmt. Es wird aber davon ausgegangen, dass immer der „richtige Weg" gewählt wird. Dies entspricht einer Überapproximation und führt für die Analyse dazu, dass auch Programme als fehlerbehaftet erkannt werden, die keinen Fehler haben. Das Problem liegt darin, dass auch ein Pfad überprüft wird, der nicht ausgeführt wird. Tritt dort ein Fehler auf, ist dieser zwar praktisch nicht relevant, wird aber trotzdem als solcher erkannt und wird somit als „false positive" von der Analyse zurückgegeben. Daraus ergibt sich:

Definition 20 ($\tilde{\beta}$-Reduktion) *Sei die $\tilde{\beta}$-Reduktion, $\to_{\tilde{\beta}}$, die kleinste Relation über $\tilde{\Lambda}$, so dass*

$$
\begin{aligned}
(\lambda x.P)Q &\to_{\tilde{\beta}} P\{x := Q\} \\
(\mathrm{I}x.P)Q &\to_{\tilde{\beta}} P\{x := Q\} \\
(\mathrm{Z}x.p)Q &\to_{\tilde{\beta}} P\{x := Q\} \\
(\Delta x.QR)P &\to_{\tilde{\beta}} Q\{x := P\} \\
(\Delta x.QR)P &\to_{\tilde{\beta}} R\{x := P\}
\end{aligned}
$$

gilt und $\to_{\tilde{\beta}}$ unter

$$
\begin{aligned}
P \to_{\tilde{\beta}} P' &\Rightarrow \forall x \in \mathbf{V} : \lambda x.P \to_{\tilde{\beta}} \lambda x.P' \\
P \to_{\tilde{\beta}} P' &\Rightarrow \forall x \in \mathbf{V} : \mathrm{I}x.P \to_{\tilde{\beta}} \mathrm{I}x.P' \\
P \to_{\tilde{\beta}} P' &\Rightarrow \forall x \in \mathbf{V} : \mathrm{Z}x.P \to_{\tilde{\beta}} \mathrm{Z}x.P' \\
P \to_{\tilde{\beta}} P' &\Rightarrow \forall x \in \mathbf{V}, \forall Z \in \tilde{\Lambda} : \Delta x P Z \to_{\tilde{\beta}} \Delta x P' Z \\
P \to_{\tilde{\beta}} P' &\Rightarrow \forall x \in \mathbf{V}, \forall Z \in \tilde{\Lambda} : \Delta x Z P \to_{\tilde{\beta}} \Delta x Z P' \\
P \to_{\tilde{\beta}} P' &\Rightarrow \forall Z \in \tilde{\Lambda} : PZ \to_{\tilde{\beta}} P'Z \\
P \to_{\tilde{\beta}} P' &\Rightarrow \forall Z \in \tilde{\Lambda} : ZP \to_{\tilde{\beta}} ZP'
\end{aligned}
$$

geschlossen ist.

Die $\tilde{\beta}$-Reduktion erlaubt es uns nun $\tilde{\lambda}$-Terme auszuwerten beziehungsweise zu reduzieren. Im Folgenden werden einige Beispiele für $\tilde{\lambda}$-Terme besprochen und reduziert. Hierbei wird auf zwei Dinge aufmerksam gemacht. Zum einen werden auf dieser Ebene alle Abstraktionen gleich behandelt, das heißt die Semantik der I- und Z-Abstraktion als Senke beziehungsweise Quelle wird hier nicht beachtet. Das hat zur Folge, dass auch Terme reduziert werden, die die Ziele der Taint-Analyse verletzen. Das bedeutet auch Terme, die verschmutzte Werte in kritischen Bereichen verwenden, werden reduziert. Außerdem ist in diesem Kalkül die Wirkung der Sanitization nicht gegeben, da der Δ-Check-Point in beliebige Richtung ausgewertet werden kann.

Zuerst soll ein Beispiel aus dem bekannten Lambda-Kalkül betrachtet werden. Der Term für die `flip` Funktion ist in beiden Kalkülen identisch, das heißt er ist sowohl Element von Λ als auch von $\tilde{\Lambda}$. Es kann also geschrieben werden als:

$$\begin{aligned}(\lambda f.\lambda a.\lambda b.fba)FAB &\to_{\tilde{\beta}} (\lambda a.\lambda b.Fba)AB \\ &\to_{\tilde{\beta}} (\lambda b.FbA)B \\ &\to_{\tilde{\beta}} FBA\end{aligned}$$

Das nächste Beispiel verwendet die aus Beispiel 7 bekannte Kodierung für natürliche Zahlen. Zusätzlich wird die Multiplikation definiert, um eine Berechnung zu simulieren. Durch anschließendes Verwenden von I- und Z-Abstraktion wird ein Term konstruiert, der gegen die Ziele der Taint-Analyse verstößt, aber trotzdem korrekt reduziert. Dadurch wird dargestellt, dass die Semantik des Kalküls solche Berechnungen nicht verhindert. Diese werden erst durch die in Abschnitt 3.2 beschriebenen Regeln ausgeschlossen.

Beispiel 10 ($2 * a$) *Sei die Multiplikation auf Church-kodierten natürlichen Zahlen (Beispiel 7) definiert durch [Sør06]:*

$$A_* = \lambda m.\lambda n.\lambda s.\lambda z.m(ns)z.$$

Nun soll folgender Term betrachtet werden:

$$(\mathrm{I}x.A_*x2)((\mathrm{Z}y.3)1).$$

Eine mögliche $\tilde{\beta}$-Reduktion ergibt sich dann durch:

$$\begin{aligned}(\mathrm{I}x.A_*x2)((\mathrm{Z}y.3)1) &\to_{\tilde{\beta}} (\mathrm{I}x.A_*x2)3 \\ &\to_{\tilde{\beta}} A_*3\ 2 \\ &= (\lambda m.\lambda n.\lambda s.\lambda z.m(ns)z)3\ 2 \\ &\to_{\tilde{\beta}} \lambda s.\lambda z.3(2s)z \\ &= \lambda s.\lambda z.3((\lambda s.\lambda z.s(sz))s)z \\ &\to_{\tilde{\beta}} \lambda s.\lambda z.3(\lambda z.s(sz))z \\ &= \lambda s.\lambda z.(\lambda s.\lambda z.s(s(sz)))(\lambda z.s(sz))z \\ &\to_{\tilde{\beta}} \lambda s.\lambda z.(\lambda z.(\lambda z.s(sz))((\lambda z.s(sz))((\lambda z.s(sz))z)))z \\ &\to_{\tilde{\beta}} \lambda s.\lambda z.(\lambda z.s(sz))((\lambda z.s(sz))((\lambda z.s(sz))z)) \\ &\to_{\tilde{\beta}} \lambda s.\lambda z.(\lambda z.s(sz))(\lambda z.s(sz))(s(sz)) \\ &\to_{\tilde{\beta}} \lambda s.\lambda z.(\lambda z.s(sz))(s(s(s(sz)))) \\ &\to_{\tilde{\beta}} \lambda s.\lambda z.s(s(s(s(sz)))) \\ &= 6\end{aligned}$$

Die Konstante 3 *kann in diesem Beispiel wie die Simulation einer Eingabe interpretiert werden. Der hier dargestellte Term entspricht also einem Programm, das eine Eingabe eines Benutzers entgegennimmt, diese mit der Konstante* 2 *multipliziert und dieses Ergebnis zurückgibt. Um einen kritischen Bereich zu simulieren, wurde hier die Multiplikation mit* 2 *mit einer* I*-Abstraktion markiert. Das Beispiel sollte in dieser Form also von einer statischen Analyse zurückgewiesen werden. Ein verschmutzter Wert, der durch die simulierte Eingabe entsteht, sollte nicht in dem kritischen Bereich verwendet werden.*

In dem folgenden Beispiel wird ein sehr ähnliches Programm dargestellt. Der Unterschied zwischen den beiden Programmen liegt darin, dass hier die Eingabe durch einen Δ-Check-Point bereinigt wird. Dadurch sollte dieses Programm durch eine statische Analyse akzeptiert werden.

Beispiel 11 ($2 * a$**)** *Sei die Multiplikation auf Church-kodierten natürlichen Zahlen (Beispiel 7) definiert durch [Sør06]:*

$$A_* = \lambda m.\lambda n.\lambda s.\lambda z.m(ns)z.$$

Nun soll folgender Term betrachtet werden:

$$(\mathrm{I}x.A_* x2)((\Delta x.x1)((\mathrm{Z}y.3)1)).$$

Eine mögliche $\tilde{\beta}$*-Reduktion ergibt sich dann durch:*

$$\begin{aligned}
(\mathrm{I}x.A_* x2)((\Delta x.x1)((\mathrm{Z}y.3)1)) &\to_{\tilde{\beta}} (\mathrm{I}x.A_* x2)((\Delta x.x1)3) \\
&\to_{\tilde{\beta}} (\mathrm{I}x.A_* x2)3 \\
&\to_{\tilde{\beta}} A_* 3\ 2 \\
&= (\lambda m.\lambda n.\lambda s.\lambda z.m(ns)z)3\ 2 \\
&\to_{\tilde{\beta}} \lambda s.\lambda z.3(2s)z \\
&= \lambda s.\lambda z.3((\lambda s.\lambda z.s(sz))s)z \\
&\to_{\tilde{\beta}} \lambda s.\lambda z.3(\lambda z.s(sz))z \\
&= \lambda s.\lambda z.(\lambda s.\lambda z.s(s(sz)))(\lambda z.s(sz))z \\
&\to_{\tilde{\beta}} \lambda s.\lambda z.(\lambda z.(\lambda z.s(sz))((\lambda z.s(sz))((\lambda z.s(sz))z)))z \\
&\to_{\tilde{\beta}} \lambda s.\lambda z.(\lambda z.s(sz))((\lambda z.s(sz))((\lambda z.s(sz))z)) \\
&\to_{\tilde{\beta}} \lambda s.\lambda z.(\lambda z.s(sz))(\lambda z.s(sz))(s(sz)) \\
&\to_{\tilde{\beta}} \lambda s.\lambda z.(\lambda z.s(sz))(s(s(s(sz)))) \\
&\to_{\tilde{\beta}} \lambda s.\lambda z.s(s(s(s(sz)))) \\
&= 6
\end{aligned}$$

Hierbei ist zu beachten, dass das Programm, abhängig von der semantischen Bedeutung des Δ-Check-Points, auch einen anderen Pfad gehen kann. Ein alternativer, aber genauso gültiger Reduktionspfad wäre:

$$
\begin{aligned}
(\mathrm{I}x.A_*x2)((\Delta x.x1)((\mathrm{Z}y.3)1)) &\to_{\tilde{\beta}} (\mathrm{I}x.A_*x2)((\Delta x.x1)3) \\
&\to_{\tilde{\beta}} (\mathrm{I}x.A_*x2)1 \\
&\to_{\tilde{\beta}} A_*1\ 2 \\
&= (\lambda m.\lambda n.\lambda s.\lambda z.m(ns)z)1\ 2 \\
&\to_{\tilde{\beta}} \lambda s.\lambda z.1(2s)z \\
&= \lambda s.\lambda z.1((\lambda s.\lambda z.s(sz))s)z \\
&\to_{\tilde{\beta}} \lambda s.\lambda z.1(\lambda z.s(sz))z \\
&= \lambda s.\lambda z.(\lambda s.\lambda z.sz)(\lambda z.s(sz))z \\
&\to_{\tilde{\beta}} \lambda s.\lambda z.(\lambda z.(\lambda z.s(sz))z)z \\
&\to_{\tilde{\beta}} \lambda s.\lambda z.(\lambda z.s(sz))z \\
&\to_{\tilde{\beta}} \lambda s.\lambda z.s(sz) \\
&= 2
\end{aligned}
$$

Dieses Kapitel hat ein Berechnungsmodell entwickelt, das sich an dem Lambda-Kalkül orientiert. Der Unterschied besteht in einigen Primitiven, die auf die Taint-Analyse ausgelegt sind. So wurden separate Konstruktoren für Quellen und Senken hinzugefügt, sowie ein Konstruktor, der das Bereinigen von Werten erlaubt. Es wurde außerdem demonstriert, dass dieser Kalkül bisher nicht in der Lage ist, eine statische Taint-Analyse durchzuführen. Hierzu werden in Abschnitt 3.2 Ableitungsregeln entwickelt, die den Taint-Status verfolgen und auf diese Weise sicherstellen, dass verschmutzte Werte nicht in kritischen Bereichen verwendet werden.

3.2 Typisierung der Taint-Analyse

Das in Abschnitt 3.1 vorgestellte Berechnungsmodell ist eine Variante des λ-Kalküls. Die vorgenommenen Veränderungen erlauben es, spezielle Aspekte der Taint-Analyse zu betrachten. Hierzu wurden drei weitere Konstruktoren eingeführt:

- Z-Abstraktion zur Markierung von Quellen
- I-Abstraktion zur Markierung von Senken
- und Δ-Check-Points über die Δ-Konstruktion.

Bis zu diesem Zeitpunkt wurden diese Konstruktoren, im Fall der Abstraktionen, wie λ-Abstraktion und, im Fall der Δ-Check-Points, wie Abstraktion mit zwei möglichen Rümpfen behandelt. Die semantische Bedeutung der Konstruktoren für die Taint-Analyse wurde also noch nicht umgesetzt. Dies kann dadurch erklärt werden, dass die Taint-Analyse in diesem Modell durch ein Typsystem umgesetzt werden soll, also getrennt von der Semantik des Berechnungsmodells.

Eine mögliche Sichtweise für Typsysteme ist die, dass sie die Menge der möglichen Terme, in unserem Fall $\tilde{\Lambda}$, auf eine Menge von gültigen Termen einschränken [Sør06]. Bei Anwendung auf die Domäne der Taint-Analyse ergibt sich folgendes Bild: Das Typsystem schränkt die Menge aller Programme auf die ein, die durch Sanitization den Fluss von Informationen von Quellen zu Senken bereinigt halten. Umgekehrt macht das Typsystem auf unsichere Programme aufmerksam. Durch geschickte Implementierung des Typsystems ist es außerdem möglich, Programmpunkte, an denen verschmutzte Werte in kritische Bereiche eindringen, zu identifizieren. Dadurch können diese Stellen ausgebessert werden.

Im Folgenden werden die Regeln entwickelt, die dies ermöglichen sollen. Die Regeln werden durch Beispiele in der Programmiersprache C entwickelt und veranschaulicht. Im Anschluss werden diese Regeln auf zwei bereits bekannte Beispiele, Beispiel 10 und Beispiel 11, angewendet. Zusätzlich wird noch die Subjektreduktion bewiesen, die die Eigenschaft der Typen als Invarianten des Programms belegt.

Bevor Ableitungsregeln für die Typen entwickelt werden können, müssen zunächst die Typen beschrieben werden. Einfache Typen nach Definition 10 werden durch zwei Konstruktoren gebildet. Zum einen Typvariablen und zum anderen durch eine rekursive Konstruktion über Funktionstypen. Jede Typvariable muss in zwei Gruppen unterteilt werden, eine verschmutzte und eine unverschmutzte. Diese beiden Gruppen werden anhand der Markierungen ! und ? unterschieden. Eine Typvariable σ wird also unterschieden in $\sigma!$ und $\sigma?$. Der Typ $\sigma?$ symbolisiert, dass der zugehörige Wert vom einfachen Typ σ ist und verschmutzt ist. Der Typ $\sigma!$ hingegen beschreibt einen Wert vom Typen σ, der definitiv nicht verschmutzt ist. Da Typvariablen auch für Funktionen stehen können, beziehungsweise mit ihnen unifiziert werden können, muss die gleiche Notation ebenfalls für Funktionstypen gelten.

Eine Funktion $\sigma \to \tau$ wird also ebenfalls in zwei Gruppen unterschieden, $(\sigma \to \tau)!$ und $(\sigma \to \tau)?$. Hierbei ist die Verwendung von, zum Beispiel $\sigma = \sigma'!$ zu beachten. Generell gilt, dass eine Typvariable σ sowohl mit einem $\tau?$ als auch mit einem $\tau!$ unifiziert werden kann. Wohingegen der Typ $\sigma?$ nur mit einem $\tau?$ und $\sigma!$ nur mit einem $\tau!$ unifiziert werden kann.

Die Begründung liegt darin, dass der Typ $\sigma!?$ keine Semantik besitzt und daher nicht als gültiger Typ akzeptiert wird. Die Semantik eines Funktionstypen $(\sigma \to \tau!)?$ kann allerdings durch den Typen $(\sigma \to \tau?)!$ beschrieben werden. Intuitiv sagt diese

Transformation aus, dass eine verschmutzte Funktion die Verschmutzung an ihre Ausgabe weitergibt. Die Semantik von $(\sigma \to \tau?)?$ kann ebenfalls durch den Typen $(\sigma \to \tau?)!$ beschrieben werden.

Alles in allem werden zwei Dinge durch diese Überlegungen deutlich. Erstens muss Definition 10 erweitert werden und zweitens muss eine Reduktion auf den Typen definiert werden, die komplexe Funktionstypen vereinfacht. Für die Definition der Verschmutzungstypen (*eng.* Taint-Types) ergibt sich:

Definition 21 (Taint-Types) *Sei U ein unendliches, abzählbares Alphabet, dessen Elemente* Typvariablen *genannt werden. Die Menge der Taint-Types $\tilde{\Pi}$ wird dann gebildet durch:*

$$\tilde{\Pi} ::= U! \mid U? \mid (\tilde{\Pi} \to \tilde{\Pi})! \mid (\tilde{\Pi} \to \tilde{\Pi})?$$

Im Allgemeinen wird $\to$ als rechts-assoziativ betrachtet. Für $(\sigma \to \tau)!$ wird im Folgenden $\sigma \to \tau$ geschrieben.

Die Reduktion hat zum Ziel die Typen, die durch die Verschmutzung einer Funktion entstehen, zu vereinfachen. Die Regel dabei ist, dass eine Funktion die Verschmutzung an ihre Ausgabe weitergibt. Des Weiteren soll die Vereinfachung auch im Inneren der Typen erfolgen. Die Struktur ähnelt daher der β-Reduktion.

Definition 22 ($\tilde{\pi}$-Reduktion) *Sei die $\tilde{\pi}$-Reduktion, $\rightarrowtail_{\tilde{\pi}}$, die kleinste Relation über $\tilde{\Pi}$, so dass*

$$(\sigma \to \tau!)? \rightarrowtail_{\tilde{\pi}} \sigma \to \tau?$$
$$(\sigma \to \tau?)? \rightarrowtail_{\tilde{\pi}} \sigma \to \tau?$$

gilt und $\rightarrowtail_{\tilde{\pi}}$ unter

$$T \rightarrowtail_{\tilde{\pi}} T' \Rightarrow \forall S \in \tilde{\Pi} : T \to S \rightarrowtail_{\tilde{\pi}} T' \to S$$
$$T \rightarrowtail_{\tilde{\pi}} T' \Rightarrow \forall S \in \tilde{\Pi} : S \to T \rightarrowtail_{\tilde{\pi}} S \to T'$$

geschlossen ist.

Ein Typ τ ist in $\tilde{\pi}$-Normalform, falls kein Typ σ existiert, sodass $\tau \rightarrowtail_{\tilde{\pi}} \sigma$ gilt. Außerdem ist ein Typ τ eine $\tilde{\pi}$-Normalform von σ, falls ein Reduktionspfad existiert, so dass $\sigma \rightarrowtail_{\tilde{\pi}} \ldots \rightarrowtail_{\tilde{\pi}} \tau$ gilt und τ in $\tilde{\pi}$-Normalform ist. Die $\tilde{\pi}$-Reduktion erlaubt es Typen in einer einfacheren Form zu betrachten. In dieser einfacheren Form steht die Markierung ? nur noch an Typvariablen. Dies entspricht einer Bereinigung der Funktionstypen, indem die Verschmutzung auf die Ausgabe übertragen wird. Um

eine solche Transformation anwenden zu dürfen, muss gezeigt werden, dass jeder Typ, eine $\tilde{\pi}$-Normalform besitzt. Hierfür werden einige Hilfslemma entwickelt und bewiesen.

Als eine erste Überlegung werden bereits bereinigte Funktionen betrachtet. Sind die Bestandteile der Funktion, das heißt, der Ein- und Ausgabetyp, bereits in Normalform, so sollte der vollständige Typ ebenfalls in Normalform sein.

Theorem 2 ($\tilde{\pi}$-Normalform von unverschmutzten Funktionen) *Falls $\sigma, \tau \in \tilde{\Pi}$ in $\tilde{\pi}$-Normalform sind, kann gefolgert werden, dass $\sigma \to \tau$ in $\tilde{\pi}$-Normalform ist.*

Bewiesen werden kann dies durch die Betrachtung der Reduktionsrelation. Der Aufbau eines Funktionstypen, wie oben beschrieben, macht es nicht möglich einen weiteren Reduktionsschritt wahrzunehmen. Daher befindet sich ein solcher Typ in $\tilde{\pi}$-Normalform.

Beweis 2 ($\tilde{\pi}$-Normalform von unverschmutzten Funktionen) *Seien $\sigma, \tau \in \tilde{\Pi}$ in $\tilde{\pi}$-Normalform.*

- *Da kein τ' existiert, so dass $(\sigma \to \tau'!)? = \sigma \to \tau$, $\sigma \to \tau \not\rightarrowtail_{\tilde{\pi}} \sigma \to \tau'?$*
- *Da kein τ' existiert, so dass $(\sigma \to \tau'?)? = \sigma \to \tau$, $\sigma \to \tau \not\rightarrowtail_{\tilde{\pi}} \sigma \to \tau'?$*
- *Da kein σ' existiert, so dass $\sigma \rightarrowtail_{\tilde{\pi}} \sigma'$, $\sigma \to \tau \not\rightarrowtail_{\tilde{\pi}} \sigma' \to \tau$*
- *Da kein τ' existiert, so dass $\tau \rightarrowtail_{\tilde{\pi}} \tau'$, $\sigma \to \tau \not\rightarrowtail_{\tilde{\pi}} \sigma \to \tau'$*

Also existiert kein Typ ρ, so dass $\sigma \to \tau \rightarrowtail_{\tilde{\pi}} \rho$ und damit ist $\sigma \to \tau$ in $\tilde{\pi}$-Normalform.

Auf die gleiche Weise kann auch gezeigt werden, dass mit ? oder ! markierte Typvariablen ebenfalls in $\tilde{\pi}$-Normalform sind.

Theorem 3 ($\tilde{\pi}$-Normalform von Typvariablen) *Für alle $\sigma \in U$ sind $\sigma!$ und $\sigma?$ in $\tilde{\pi}$-Normalform.*

Zuletzt können diese Überlegungen genutzt werden, um per Induktion zu zeigen, dass jeder Typ eine $\tilde{\pi}$-Normalform besitzt. Die Idee des Induktionsschrittes ist dabei folgende: Die Bestandteile eines Funktionstypen besitzen über die Induktionshypothese selbst Normalformen. Der so entstehende Reduktionspfad kann auf den Funktionstypen erweitert werden und durch Theorem 2 kann gezeigt werden, dass dieser Pfad dann zu einer Normalform führt.

Theorem 4 ($\tilde{\pi}$-Normalform von Typen) *Für jeden Typ σ existiert ein Typ τ, so dass τ die $\tilde{\pi}$-Normalform von σ ist.*

Beweis 3 ($\tilde{\pi}$-Normalform von Typen) *Sei $\sigma \in \tilde{\Pi}$. Es wird durch Induktion gezeigt, dass ein τ existiert, so dass τ die $\tilde{\pi}$-Normalform von σ ist.*

Induktionsanfang: Sei $\sigma = u!$ für ein $u \in U$ oder $\sigma = u?$ für ein $u \in U$, durch Theorem 3 ist σ in $\tilde{\pi}$-Normalform.

Induktionsschritt 1: Sei $\sigma = \tau_1 \to \tau_2$ mit $\tau_1, \tau_1', \tau_2, \tau_2' \in \tilde{\Pi}$, τ_1' die $\tilde{\pi}$-Normalform von τ_1 und τ_2' die $\tilde{\pi}$-Normalform von τ_2. Über die Definition der Normalform existiert ein Reduktionspfad zwischen τ_1, τ_1 und τ_2, τ_2'. Es gilt also $\sigma = \tau_1 \to \tau_2 \rightarrowtail_{\tilde{\pi}} \ldots \rightarrowtail_{\tilde{\pi}} \tau_1' \to \tau_2 \rightarrowtail_{\tilde{\pi}} \ldots \rightarrowtail_{\tilde{\pi}} \tau_1' \to \tau_2'$. Über Theorem 2 ist $\tau_1' \to \tau_2' = \tau$ in $\tilde{\pi}$-Normalform.

Induktionsschritt 2: Sei $\sigma = (\tau_1 \to \tau_2!)?$ mit $\tau_1, \tau_1', \tau_2, \tau_2' \in \tilde{\Pi}$, τ_1' die $\tilde{\pi}$-Normalform von τ_1 und τ_2' die $\tilde{\pi}$-Normalform von $\tau_2?$. Über die Definition der Normalform existiert ein Reduktionspfad zwischen τ_1, τ_1 und $\tau_2?, \tau_2'$. Es gilt also $\sigma = (\tau_1 \to \tau_2!)? \rightarrowtail_{\tilde{\pi}} \tau_1 \to \tau_2? \rightarrowtail_{\tilde{\pi}} \ldots \rightarrowtail_{\tilde{\pi}} \tau_1' \to \tau_2? \rightarrowtail_{\tilde{\pi}} \ldots \rightarrowtail_{\tilde{\pi}} \tau_1' \to \tau_2'$. Über Theorem 2 ist $\tau_1' \to \tau_2' = \tau$ in $\tilde{\pi}$-Normalform.

Induktionsschritt 3: Sei $\sigma = (\tau_1 \to \tau_2?)?$ mit $\tau_1, \tau_1', \tau_2, \tau_2' \in \tilde{\Pi}$, τ_1' die $\tilde{\pi}$-Normalform von τ_1 und τ_2' die $\tilde{\pi}$-Normalform von $\tau_2?$. Über die Definition der Normalform existiert ein Reduktionspfad zwischen τ_1, τ_1 und $\tau_2?, \tau_2'$. Es gilt also $\sigma = (\tau_1 \to \tau_2?)? \rightarrowtail_{\tilde{\pi}} \tau_1 \to \tau_2? \rightarrowtail_{\tilde{\pi}} \ldots \rightarrowtail_{\tilde{\pi}} \tau_1' \to \tau_2? \rightarrowtail_{\tilde{\pi}} \ldots \rightarrowtail_{\tilde{\pi}} \tau_1' \to \tau_2'$. Über Theorem 2 ist $\tau_1' \to \tau_2' = \tau$ in $\tilde{\pi}$-Normalform.

Jeder Typ besitzt nun eine $\tilde{\pi}$-Normalform und im Folgenden wird davon ausgegangen, dass sämtliche Typen in Normalform vorliegen. Diese sind in ihrer Struktur einfacher und besitzen keine verschmutzten Funktionstypen, was eine intuitivere Verwendung ermöglicht, da man meistens eine Trennung von Funktion und Daten verwendet, auch wenn diese Trennung zunehmend durch funktionale Sprachen wie Haskell aufgebrochen wird.

In dieser Arbeit werden zwei Markierungen für Typen verwendet. Ein weiterer Ansatz könnte drei Markierungen verwenden. Die zusätzliche Markierung wird verwendet, um den Taint-Status als „egal" zu markieren. Dieser Ansatz wurde hier verworfen, da er einen entscheidenden Nachteil hat. Eine dritte Markierung, im Folgenden durch $\sigma.$ beschrieben, führt dazu, dass die Unifikation erheblich erschwert wird. Der Algorithmus, der von Robinson [Rob71] dargestellt wird, arbeitet durch das Umschreiben von Gleichungen. Zum Beispiel könnte er das Gleichungssystem von

$$\sigma? = \sigma.$$
$$\sigma? = \sigma!$$

zu

$$\sigma? = \sigma.$$
$$\sigma. = \sigma!$$

transformieren. $\sigma.$ soll sowohl gleich zu $\sigma?$ als auch gleich zu $\sigma!$ sein. Im ersten Gleichungssystem ist *lokal* erkennbar, dass das System nicht gelöst werden kann, da $\sigma? \neq \sigma!$ gilt. Im zweiten ist dies nicht mehr *lokal* zu erkennen. Das Erkennen der Lösbarkeit wird *global*. Für den Algorithmus bedeutet dies, dass nur bestimmte Umschreibungen zu einem Zeitpunkt möglich sind. Dadurch soll verhindert werden, dass ein System, das bereits *lokal* als nicht lösbar zu erkennen ist, diese Eigenschaft nicht verliert. Daher wurde sich hier für eine Modellierung mit lediglich zwei Markierungen entschieden, da dies erlaubt, den Algorithmus nach [Rob71] direkt zu verwenden.

Im Folgenden sollen nun die Ableitungsregeln entwickelt werden. Diese orientieren sich an Beispielen der Programmiersprache C. Zuerst sollen Variablen betrachtet werden bevor Regeln eingeführt werden, die Funktionen verwenden.

Variablen Variablen werden in den meisten Fällen durch eine Deklaration in ein Programm eingeführt. An dieser Stelle wird auch der Typ einer Variable bestimmt und in den Kontext aufgenommen. Alle weiteren Verwendungen dieser Variable machen die Annahme, dass der Typ in dem Kontext existiert. Wird diese Annahme verletzt, führt das zu einem Fehler, da der so definierte Term nicht abgeleitet werden kann. Hier wird vorausgesetzt, dass eine Variable in dem Kontext existiert, in dem die Variablen verwendet werden. Entsprechend wird auch der Typ durch den Kontext vorgegeben. Es entsteht also eine Regel, welche der Regel aus dem einfachen getypten Lambda-Kalkül ähnelt. Es ist aber zu beachtet, dass das σ hier für $\sigma'!$ oder $\sigma'?$ stehen kann:

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \text{ (Ax)}$$

Applikation Die Applikation entspricht der Funktionsanwendung. Hierzu wird nun ein Beispiel (Listing 3.1) betrachtet, in dem eine einfache Funktion a definiert wird, die eine Eingabe entgegennehmen soll. Typen in der oben beschriebenen Form wurden in den Kommentaren eingefügt, um die Aussagekraft zu erhöhen. Es stellt sich nun die Frage, ob a nur auf y, nur auf z oder sowohl auf y als auch auf z anwendbar sein sollte. Die Anwendung von a auf y sollte erhalten bleiben, da sie vollständig unifizierbar ist. Der Typ von y entspricht genau dem Argumenttyp von a. Es bleibt offen, ob a auch auf z anwendbar sein sollte. Die Überlegung begründet sich in der Semantik, die den Markierungen zugedacht wird. Ein mit ! markierter

```
1   // a : int? -> int?
2   int a(int x){
3     return x + 1;
4   }
5   /*
6   ....
7   */
8   // y : int?
9   int y = 10;
10  // z : int!
11  int z = 10;
```

Listing 3.1.: Eine Funktionsanwendung in C mit annotierten Taint-Types

Typ wird als nicht verschmutzt interpretiert, wohingegen ? für verschmutzt steht. Nun sollte es ja möglich sein, jeden nicht verschmutzten Wert auch als verschmutzt zu interpretieren. Damit würden die Annahmen über den Wert lediglich geschwächt. Wenn die Annahmen über die Variable z nun abgeschwächt würden, wären die Typen von a und z ebenfalls kompatibel. Es wird also der Typ von z überapproximiert. Es entstehen also zwei Ableitungsregeln, eine bezieht sich direkt auf die Applikation und die andere bezieht sich auf ein allgemeineres Konzept, das aber hauptsächlich durch die Applikation seine Anwendung findet. Um diese Regeln zu verdeutlichen, wird ein weiteres Beispiel in Form von C-Code betrachtet.

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau} \text{ (App)}$$

$$\frac{\Gamma \vdash M : \sigma!}{\Gamma \vdash M : \sigma?} \text{ (weak)}$$

Beispiel 12 (Fib) *`fib` (Listing 3.2) berechnet die* n*-te Fibonacci-Zahl in der Sequenz. Die Funktion kann auf drei unterschiedliche Weisen aufgerufen werden. Im ersten, zweiten und vierten Fall sind die Typen kompatibel und können daher nach der üblichen Applikation angewendet werden. Im dritten Beispiel gilt dies nicht, dort muss der Typ von x geschwächt werden, um danach kompatibel zu sein.*

Abstraktion Um die üblichen Konstruktoren des Lambda-Kalküls abzudecken, fehlt an dieser Stelle noch die Abstraktion. In dem hier vorgestellten Kalkül existiert nicht nur die übliche λ-Abstraktion, sondern noch zwei weitere. Die weiteren Abstraktionen dienen der Markierung von Quellen und Senken. Die Semantik, die

```c
// fib : int! -> int!
// fib : int! -> int?
// fib : int? -> int?
int fib(int n) {
  if(n == 0) {
    return 1;
  } else if(n == 1) {
    return 1;
  } else {
    return fib(n-1) + fib(n-2);
  }
}

// Bsp 1: x     : int!
//        fib   : int! -> int!
//        fib x :         int!
// Bsp 2: x     : int?
//        fib   : int? -> int?
//        fib x :         int?
// Bsp 3: x     : int!
//        fib   : int? -> int?
//        fib x :         int?
// Bsp 4: x     : int!
//        fib   : int! -> int?
//        fib x :         int?
```

Listing 3.2.: Die C-Funktion `fib`.

mit diesen Konstruktoren verbunden wird, wurde bisher nicht durch einen Formalismus beschrieben. Dies geschieht an dieser Stelle. Zuerst soll aber die λ-Abstraktion betrachtet werden. Sie soll für übliche Funktionen stehen. Betrachtet werden soll hierzu ein weiteres Mal die Funktion `fib` (Listing 3.2).

Um zu entscheiden wie die Typen abgeleitet werden sollen, können zwei Fälle unterschieden werden. Zum einen die Situation, wenn `n` verschmutzt ist, und zum anderen, wenn `n` nicht verschmutzt ist. Zunächst wird davon ausgegangen, dass `n` verschmutzt ist. In diesem Fall sind sämtliche Werte, die sich aus `n` ergeben, ebenfalls verschmutzt. In Listing 3.2 bedeutete das, dass durch die Verwendung von `n` in der `if`-Verzweigung, alle Rückgaben ebenfalls verschmutzt sind. Das bedeutet, dass eine verschmutzte Eingabe `n` auch zu einem verschmutzten Ergebnis führen muss. Für eine nicht verschmutzte Eingabe dreht sich das Bild. Die `if`-Verzweigung führt zu keiner Verschmutzung, Konstanten können als unverschmutzt interpretiert werden, und da `fib` dann rekursiv aufgerufen wird, kann durch Induktion gezeigt werden, dass das Ergebnis ebenfalls nicht verschmutzt ist. Die Eingabe bestimmt also den Taint-Status des Ergebnisses, wobei der Taint-Status weitergegeben wird. Diesen Sachverhalt kann man nun in zwei entsprechende Ableitungsregeln fassen:

$$\frac{\Gamma, x : \sigma! \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma! \to \tau} \ (\texttt{AbsI}\lambda)$$

$$\frac{\Gamma, x : \sigma? \vdash M : \tau?}{\Gamma \vdash \lambda x.M : \sigma? \to \tau?} \ (\texttt{AbsS}\lambda)$$

Nun sollen die anderen Abstraktionen betrachtet werden. Da diese jeweils eine Teilmenge der λ-Abstraktion abdecken sollen, lassen sich die Ableitungsregeln ähnlich wie oben begründen. Die Z-Abstraktion beschreibt Quellen. Dadurch muss die Ausgabe einen Typ der Form $\sigma?$ besitzen. Bei einer Betrachtung der Ableitungsregeln der λ-Abstraktion fällt auf, dass nur eine Regel in Betracht kommt, $\texttt{AbsS}\lambda$. Umgekehrt gilt für die I-Abstraktion, dass nur die $\texttt{AbsI}\lambda$-Regel als Parallele in Betracht gezogen werden kann. Daraus ergeben sich also zwei Regeln:

$$\frac{\Gamma, x : \sigma! \vdash M : \tau}{\Gamma \vdash \mathrm{I}x.M : \sigma! \to \tau} \ (\texttt{AbsI})$$

$$\frac{\Gamma, x : \sigma? \vdash M : \tau?}{\Gamma \vdash \mathrm{Z}x.M : \sigma? \to \tau?} \ (\texttt{AbsZ})$$

```
1    // wert : int?
2    if(rein(wert)) {
3      verwende_wert_1(wert);
4    } else {
5      verwende_wert_2(wert);
6    }
```

Listing 3.3.: `if` zur Bereinigung von Werten

Δ-Check-Points Der letzte Konstruktor dieses Kalküls ist der Δ-Check-Point. Semantisch dient dieser Konstruktor der Bereinigung von Werten. Die Struktur ähnelt dabei einer `if`-Verzweigung. Diese wird an dieser Stelle auch als Beispiel herangezogen (Listing 3.3). Bei diesem Vorgehen wird davon ausgegangen, dass der Wert `wert` verschmutzt ist, ansonsten wäre die Überprüfung nicht nötig. Nun soll getestet werden, ob es sich um einen gültigen oder ungültigen Wert handelt. Im positiven Fall kann dann `wert` als rein betrachtet werden, im negativen Fall bleibt er auf Typebene verschmutzt, auch wenn nun bekannt ist, dass dieser Wert weiterhin nicht verwendet werden sollte. Da es aber möglich ist, dass eine Bereinigung des Wertes erreicht werden kann, wird dieser auch im `else`-Zweig verwendet.

Die beiden Verzweigungen arbeiten auf eine Weise wie Abstraktionen, die je nach Taint-Status des Wertes ausgewählt und mit dem Wert versorgt werden. Durch diesen Charakter als Abstraktion ist es möglich, dass sie den Wert je als einen anderen Typ interpretieren. Insgesamt kann dies zu folgender Regel zusammengefasst werden:

$$\frac{\Gamma, x : \sigma! \vdash M : \tau \qquad \Gamma, x : \sigma? \vdash N : \tau}{\Gamma \vdash \Delta x.MN : \sigma? \to \tau} \ (\Delta\text{-check})$$

Beispiele Die hier beschriebenen Regeln wirken zunächst etwas abstrakt, daher werden sie nun durch Anwendung auf Beispiel 10 und Beispiel 11 etwas anschaulicher betrachtet. Um die Beispiele einfach zu halten, wird von primitiven Typen für natürliche Zahlen zusammen mit einer entsprechenden Funktion zur Multiplikation, $\otimes$, ausgegangen. Eine Ableitung für den Typ der natürlichen Zahlen in Church-Kodierung kann Anhang C entnommen werden. Der Typ der natürlichen Zahlen lautet nun $\mathbb{N}$ mit den entsprechenden Markierungen. Die Multiplikation hat dabei folgende mögliche Typen:

- $\mathbb{N}! \to \mathbb{N}! \to \mathbb{N}!$
- $\mathbb{N}! \to \mathbb{N}! \to \mathbb{N}?$
- $\mathbb{N}? \to \mathbb{N}! \to \mathbb{N}?$

- $\mathbb{N}! \to \mathbb{N}? \to \mathbb{N}?$
- $\mathbb{N}? \to \mathbb{N}? \to \mathbb{N}?$

Nun soll der Term aus Beispiel 10 betrachtet werden.

Beispiel 13 *Der adaptierte Term aus Beispiel 10 lautete* $(\mathrm{I}x.\otimes x2)((\mathrm{Z}y.3)1)$. *Nun soll versucht werden einen Typ für diesen Term zu finden:*
Ableitungsbaum 1:

$$\dfrac{\dfrac{}{x : \mathbb{N}! \vdash \otimes : \mathbb{N}! \to \mathbb{N}! \to \mathbb{N}!}\,(Ax) \qquad \dfrac{}{x : \mathbb{N}! \vdash x : \mathbb{N}!}\,(Ax)}{x : \mathbb{N}! \vdash \otimes x : \mathbb{N}! \to \mathbb{N}!}\,(App)$$

Ableitungsbaum 2:

$$\dfrac{\dfrac{\textit{Ableitungsbaum 1} \qquad \dfrac{}{x : \mathbb{N}! \vdash 2 : \mathbb{N}!}\,\textit{Primitiver Typ}}{x : \tau_2! \vdash \otimes x2 : \tau_3!}\,(App)\ \textit{mit}\ \tau_2 = \tau_3 = \mathbb{N}}{\emptyset \vdash \mathrm{I}x.\otimes x2 : \tau_1 \to \tau}\,(Abs\mathrm{I})\ \textit{mit}\ \tau_1 = \tau_2! \wedge \tau = \tau_3!$$

Ableitungsbaum 3:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y : \tau_4? \vdash 3 : \mathbb{N}!}\,\textit{Primitiver Typ}}{y : \tau_4? \vdash 3 : \tau_5?}\,(weak)\ \textit{mit}\ \tau_5 = \mathbb{N}}{\emptyset \vdash \mathrm{Z}y.3 : \tau_4? \to \tau_5?}\,(Abs\mathrm{Z}) \qquad \dfrac{\dfrac{}{\emptyset \vdash 1 : \mathbb{N}!}\,\textit{Primitiver Typ}}{\emptyset \vdash 1 : \tau_4?}\,(weak)\ \textit{mit}\ \tau_4 = \mathbb{N}}{\emptyset \vdash (\mathrm{Z}y.3)1 : \tau_1}\,(App)\ \textit{mit}\ \tau_1 = \tau_5?$$

Ableitungsbaum 4:

$$\dfrac{\textit{Ableitungsbaum 2} \qquad \textit{Ableitungsbaum 3}}{\emptyset \vdash (\mathrm{I}x.\otimes x2)((\mathrm{Z}y.3)1) : \tau}\,(App)$$

Durch diese Bäume entsteht dann folgendes Gleichungssystem:

$$\begin{aligned}
\tau_2 &= \mathbb{N} \\
\tau_3 &= \mathbb{N} \\
\tau_1 &= \tau_2! \\
\tau &= \tau_3! \\
\tau_5 &= \mathbb{N} \\
\tau_1 &= \tau_5? \\
\tau_4 &= \mathbb{N}
\end{aligned}$$

Dieses Gleichungssystem ist allerdings inkonsistent, da durch wiederholtes Einsetzen die Gleichung

$$\mathbb{N}! = \mathbb{N}?$$

entsteht. Es liegt also ein Fehler vor, der durch dieses Typsystem erkannt wird.

Als nächstes wird das Beispiel 11 betrachtet. Semantisch arbeiten die beiden Beispiele ähnlich, der Unterschied liegt darin, dass in diesem Beispiel ein Δ-Check-Point verwendet wird, um den Wert zu bereinigen. Dadurch sollte für dieses Beispiel ein Typ abzuleiten sein.

Beispiel 14 *Der adaptierte Term aus Beispiel 11 lautete* $(\mathrm{I}x.\otimes x2)((\Delta x.x1)((\mathrm{Z}y.3)1))$. *Nun soll versucht werden, einen Typ für diesen Term zu finden:*
Ableitungsbaum 1:

$$\dfrac{\dfrac{}{x : \mathbb{N}! \vdash \otimes : \mathbb{N}! \to \mathbb{N}! \to \mathbb{N}!}\,(Ax) \qquad \dfrac{}{x : \mathbb{N}! \vdash x : \mathbb{N}!}\,(Ax)}{x : \mathbb{N}! \vdash \otimes x : \mathbb{N}! \to \mathbb{N}!}\,(App)$$

Ableitungsbaum 2:

$$\dfrac{\dfrac{\textit{Ableitungsbaum 1} \qquad \dfrac{}{x : \mathbb{N}! \vdash 2 : \mathbb{N}!}\,\textit{Primitiver Typ}}{x : \tau_2! \vdash \otimes x2 : \tau_3!}\,(App)\ \textit{mit}\ \tau_2 = \tau_3 = \mathbb{N}}{\emptyset \vdash \mathrm{I}x.\otimes x2 : \tau_1 \to \tau}\,(Abs\mathrm{I})\ \textit{mit}\ \tau_1 = \tau_2! \wedge \tau = \tau_3!$$

Ableitungsbaum 3:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y : \tau_4? \vdash 3 : \mathbb{N}!}\,\textit{Primitiver Typ}}{y : \tau_4? \vdash 3 : \tau_5?}\,(weak)\ \textit{mit}\ \tau_5 = \mathbb{N}}{\emptyset \vdash \mathrm{Z}y.3 : \tau_4? \to \tau_5?}\,(Abs\mathrm{Z}) \qquad \dfrac{\dfrac{}{\emptyset \vdash 1 : \mathbb{N}!}\,\textit{Primitiver Typ}}{\emptyset \vdash 1 : \tau_4?}\,(weak)\ \textit{mit}\ \tau_4 = \mathbb{N}}{\emptyset \vdash (\mathrm{Z}y.3)1 : \tau_5?}\,(App)$$

Ableitungsbaum 4:

$$\dfrac{\dfrac{}{x : \tau_5! \vdash x : \tau_5!}\,(Ax)\ \textit{mit}\ \tau_1 = \tau_5! \qquad \dfrac{}{x : \tau_5! \vdash 1 : \mathbb{N}!}\,\textit{Primitiver Typ mit}\ \tau_1 = \mathbb{N}!}{\emptyset \vdash \Delta x.x1 : \tau_5? \to \tau_1}\,(\Delta\text{-}check)$$

Ableitungsbaum 5:

$$\frac{\textit{Ableitungsbaum 3} \qquad \textit{Ableitungsbaum 4}}{\emptyset \vdash (\Delta x.x1)(\mathrm{Z}y.3)1 \,:\, \tau_1} \; \textit{(App) mit } \tau_1 = \tau_5?$$

Ableitungsbaum 6:

$$\frac{\textit{Ableitungsbaum 2} \qquad \textit{Ableitungsbaum 5}}{\emptyset \vdash (\mathrm{I}x. \otimes x2)((\mathrm{Z}y.3)1) \,:\, \tau} \; \textit{(App)}$$

Durch diese Bäume entsteht dann folgendes Gleichungssystem:

$$\begin{aligned} \tau_2 &= \mathbb{N} \\ \tau_3 &= \mathbb{N} \\ \tau_1 &= \tau_2! \\ \tau &= \tau_3! \\ \tau_5 &= \mathbb{N} \\ \tau_1 &= \tau_5! \\ \tau_4 &= \mathbb{N} \end{aligned}$$

Tatsächlich ist das so entstandene Gleichungssystem konsistent, was bedeutet, dass kein Fehler von dem Typsystem erkannt wurde.

Durch diese Beispiele wurde veranschaulicht, dass das Typsystem erkennt, wenn eine Eingabe in einem kritischen Bereich verwendet werden soll. Eingaben werden durch Z-Abstraktionen und kritische Bereiche durch I-Abstraktionen simuliert. Soll die Eingabe trotzdem verwendet werden, kann dies durch einen Δ-Check-Point geschehen. Ein solcher zwingt einen Anwender den Fall einer verschmutzten Eingabe zu beachten und schützt auf diese Weise vor Fehlern.

Im Folgenden soll gezeigt werden, dass die Subjektreduktion, die im Lambda-Kalkül existiert, durch die vorgestellte Erweiterung nicht verletzt ist. Diese Eigenschaft ist entscheidend, um Typen als Invarianten eines Programms zu betrachten. Das Vorgehen unterscheidet sich dabei nicht von dem Beweis im Lambda-Kalkül. Zuerst wird das Substitutionslemma gezeigt, dann wird per Induktion gezeigt, dass die Subjektreduktion erhalten bleibt.

Die Subjektreduktion setzt die Typisierung eines Terms in Beziehung zu der Ausführung. Die Ausführung, die durch die $\tilde{\beta}$-Reduktion dargestellt wird, basiert auf der Substitution von Variablen. Aus diesem Grund wird zuerst das Substitutionslemma gezeigt, bevor dieses genutzt wird, um die Subjektreduktion zu zeigen.

```
1    int a = 10;
2    int b = 10*a+3;
```

Listing 3.4.: Auslagerung eines Wertes 10 in die Variable a.

```
1    int b = 10*10+3;
```

Listing 3.5.: Ersetzung der Variable a durch den Wert 10.

Das Substitutionslemma sagt aus, dass die Ersetzung einer Variable durch einen Term des gleichen Typs, den Typ des gesamten Terms nicht ändert. Intuitiv lässt sich dies durch ein Beispiel aus der Programmierung veranschaulichen. Die Erwartung in der Programmierung ist, dass eine Variable durch ihren Wert ersetzt werden kann und dies den Wert des Terms nicht verändert. Listing 3.4 und Listing 3.5 sollten also äquivalent sein. Die gleiche Erwartung wird auf Ebene der Typen durch das Substitutionslemma ausgedrückt. Im Folgenden wird das Substitutionslemma durch eine Induktion bewiesen.

Beweis 4 (Substitutionslemma) *Zu zeigen ist „falls* $\Gamma, x : \tau \vdash M : \sigma$ *und* $\Gamma \vdash N : \tau$ *gilt, kann* $\Gamma \vdash M\{x := N\} : \sigma$ *gefolgert werden."*
Induktion über $\Gamma, x : \tau \vdash M : \sigma$.

Induktionsanfang 1: Sei $M = y \in \mathbf{V}$ *und* $\Gamma, x : \tau \vdash y : \sigma$. $\Gamma \vdash y : \sigma$, *da* $y\{x := N\} = y$.

Induktionsanfang 2: Sei $M = x \in \mathbf{V}$ *und* $\Gamma, x : \tau \vdash x : \tau$. $\Gamma \vdash N : \tau$, *da* $x\{x := N\} = N$.

Induktionsschritt 1: Sei $M = PQ$, $\Gamma, x : \tau \vdash PQ : \sigma$, $\Gamma \vdash P\{x := N\} : \sigma' \to \sigma$ *und* $\Gamma \vdash Q\{x := N\} : \sigma'$. *Da* $(PQ)\{x := N\} = P\{x := N\}Q\{x := N\}$, *kann geschrieben werden:*

$$\frac{\Gamma \vdash Q\{x := N\} : \sigma' \qquad \Gamma \vdash P\{x := N\} : \sigma' \to \sigma}{\Gamma \vdash P\{x := N\}Q\{x := N\} : \sigma} \; (App)$$

Induktionsschritt 2: Sei $M = \lambda y.P$, $\Gamma, x : \tau \vdash \lambda y.P : \sigma \to \sigma'$, $y \neq x$, $y \notin FV(N)$ *und* $\Gamma, y : \sigma \vdash P\{x := N\} : \sigma'$. *Da* $(\lambda y.P)\{x := N\} = \lambda y.P\{n := N\}$, *kann geschrieben werden:*

$$\frac{\Gamma, y : \sigma \vdash P\{x := N\} : \sigma'}{\Gamma \vdash \lambda y.P\{x := N\} : \sigma \to \sigma'} \; (AbsI\lambda) \text{ oder } (AbsS\lambda)$$

Induktionsschritt 3: Sei $M = \mathrm{Z}y.P, \Gamma, x : \tau \vdash \mathrm{Z}y.P : \sigma? \to \sigma'?$, $y \neq x$, $y \notin FV(N)$ *und* $\Gamma, y : \sigma? \vdash P\{x := N\} : \sigma'?$. *Da* $(\mathrm{Z}y.P)\{x := N\} = \mathrm{Z}y.P\{n := N\}$, *kann geschrieben werden:*

$$\frac{\Gamma, y : \sigma? \vdash P\{x := N\} : \sigma'?}{\Gamma \vdash \mathrm{Z}y.P\{x := N\} : \sigma? \to \sigma'?} \; (Abs\mathrm{Z})$$

Induktionsschritt 4: Sei $M = \mathrm{I}y.P, \Gamma, x : \tau \vdash \mathrm{I}y.P : \sigma! \to \sigma'$, $y \neq x$, $y \notin FV(N)$ *und* $\Gamma, y : \sigma! \vdash P\{x := N\} : \sigma'$. *Da* $(\mathrm{I}y.P)\{x := N\} = \mathrm{I}y.P\{n := N\}$, *kann geschrieben werden:*

$$\frac{\Gamma, y : \sigma! \vdash P\{x := N\} : \sigma'}{\Gamma \vdash \mathrm{I}y.P\{x := N\} : \sigma! \to \sigma'} \; (Abs\mathrm{I})$$

Induktionsschritt 5: Sei $M = \Delta y.PQ, \Gamma, x : \tau \vdash \Delta y.PQ : \sigma? \to \sigma'$, $y \neq x$, $y \notin FV(N)$ $\Gamma, y : \sigma! \vdash P\{x := N\} : \sigma'$, *und* $\Gamma, y : \sigma? \vdash Q\{x := N\} : \sigma'$. *Da* $(\Delta y.PQ)\{x := N\} = \Delta y.P\{n := N\}, Q\{n := N\}$, *kann geschrieben werden:*

$$\frac{\Gamma, y : \sigma? \vdash Q\{x := N\} : \sigma' \qquad \Gamma, y : \sigma! \vdash P\{x := N\} : \sigma'}{\Gamma \vdash \Delta y.P\{x := N\}Q\{x := N\} : \sigma? \to \sigma'} \; (\Delta\text{-}check)$$

Induktionsschritt 6: Sei $\Gamma, x : \tau \vdash M : \sigma?, \Gamma \vdash M\{x := N\} : \sigma!$.

$$\frac{\Gamma \vdash M\{x := N\} : \sigma!}{\Gamma \vdash M\{x := N\} : \sigma?} \; (weak)$$

Nun soll die Subjektreduktion gezeigt werden. Hierzu wird analog zu Beweis 1 vorgegangen. Da gezeigt werden soll, dass aus $\Gamma \vdash M : \tau$ und $M \to_{\tilde{\beta}} N$ folgt, dass $\Gamma \vdash N : \tau$ gilt, wird eine Induktion über $\to_{\tilde{\beta}}$ durchgeführt.

Beweis 5 (Subjektreduktion)
Induktionsanfang 1: Sei $M = (\lambda x.P)Q$ *und* $N = P\{x := Q\}$.

$$\frac{\dfrac{\Gamma, x : \sigma \vdash P : \tau}{\Gamma \vdash \lambda x.P : \sigma \to \tau} \; (Abs\mathrm{I}\lambda) \text{ oder } (Abs\mathrm{S}\lambda) \qquad \Gamma \vdash Q : \sigma}{\Gamma \vdash (\lambda x.P)Q : \tau} \; (App)$$

Es ist also zu zeigen, dass $\Gamma \vdash N : \tau$ *unter der Annahme, dass* $\Gamma, x : \sigma \vdash P : \tau$ *und* $\Gamma \vdash Q : \sigma$. *Dies entspricht genau dem Theorem 7.*

Induktionsanfang 2: Sei $M = (\mathrm{Z}x.P)Q$ *und* $N = P\{x := Q\}$.

$$\dfrac{\dfrac{\Gamma, x : \sigma? \vdash P : \tau?}{\Gamma \vdash \mathrm{Z}x.P : \sigma? \to \tau?}\ (\mathit{Abs}\mathrm{Z}) \qquad \Gamma \vdash Q : \sigma?}{\Gamma \vdash (\mathrm{Z}x.P)Q : \tau?}\ (\mathit{App})$$

Es ist also zu zeigen, dass $\Gamma \vdash N : \tau?$ *unter der Annahme, dass* $\Gamma, x : \sigma? \vdash P : \tau?$ *und* $\Gamma \vdash Q : \sigma?$*. Dies entspricht genau dem Theorem 7.*

Induktionsanfang 3: Sei $M = (\mathrm{I}x.P)Q$ *und* $N = P\{x := Q\}$*.*

$$\dfrac{\dfrac{\Gamma, x : \sigma! \vdash P : \tau}{\Gamma \vdash \mathrm{I}x.P : \sigma! \to \tau}\ (\mathit{Abs}\mathrm{I}) \qquad \Gamma \vdash Q : \sigma!}{\Gamma \vdash (\mathrm{I}x.P)Q : \tau}\ (\mathit{App})$$

Es ist also zu zeigen, dass $\Gamma \vdash N : \tau$ *unter der Annahme, dass* $\Gamma, x : \sigma! \vdash P : \tau$ *und* $\Gamma \vdash Q : \sigma!$*. Dies entspricht genau dem Theorem 7.*

Induktionsanfang 4: Sei $M = (\Delta x.PP')Q$ *und* $N = P\{x := Q\}$*.*

$$\dfrac{\dfrac{\Gamma, x : \sigma? \vdash P : \tau \qquad \Gamma, x : \sigma? \vdash P' : \tau}{\Gamma \vdash \Delta x.PP' : \sigma? \to \tau}\ (\Delta\text{-}\mathit{check}) \qquad \Gamma \vdash Q : \sigma?}{\Gamma \vdash (\Delta x.PP')Q : \tau}\ (\mathit{App})$$

Es ist also zu zeigen, dass $\Gamma \vdash N : \tau$ *unter der Annahme, dass* $\Gamma, x : \sigma? \vdash P : \tau$ *und* $\Gamma \vdash Q : \sigma?$*. Dies entspricht genau dem Theorem 7.*

Induktionsanfang 5: Sei $M = (\Delta x.PP')Q$ *und* $N = P'\{x := Q\}$*.*

$$\dfrac{\dfrac{\Gamma, x : \sigma? \vdash P : \tau \qquad \Gamma, x : \sigma? \vdash P' : \tau}{\Gamma \vdash \Delta x.PP' : \sigma? \to \tau}\ (\Delta\text{-}\mathit{check}) \qquad \Gamma \vdash Q : \sigma?}{\Gamma \vdash (\Delta x.PP')Q : \tau}\ (\mathit{App})$$

Es ist also zu zeigen, dass $\Gamma \vdash N : \tau$ *unter der Annahme, dass* $\Gamma, x : \sigma? \vdash P' : \tau$ *und* $\Gamma \vdash Q : \sigma?$*. Dies entspricht genau dem Theorem 7.*

Induktionsschritt 1: Sei $P \to_\beta P'$ *mit* $\Gamma, x : \sigma \vdash P : \tau$ *und* $\Gamma, x : \sigma \vdash P' : \tau$*,* $M = \lambda x.P$ *und* $N = \lambda x.P'$ *für beliebige* $x \in \mathbf{V}$*.*

$$\dfrac{\Gamma, x : \sigma \vdash P : \tau}{\Gamma \vdash \lambda x.P : \sigma \to \tau}\ (\mathit{Abs}\mathrm{I}\lambda) \text{ oder } (\mathit{Abs}\mathrm{S}\lambda)$$

In dem gleichen Kontext Γ *ist ebenfalls zu zeigen, dass* $\lambda x.P'$ *den Typ* $\sigma \to \tau$ *hat:*

$$\frac{\Gamma, x : \sigma \vdash P' : \tau}{\Gamma \vdash \lambda x.P' : \sigma \to \tau} \text{ (AbsI}\lambda\text{) oder (AbsS}\lambda\text{)}$$

Induktionsschritt 2: Sei $P \to_\beta P'$ *mit* $\Gamma, x : \sigma? \vdash P : \tau?$ *und* $\Gamma, x : \sigma? \vdash P' : \tau?$, $M = \mathrm{Z}x.P$ *und* $N = \mathrm{Z}x.P'$ *für beliebige* $x \in \mathbf{V}$.

$$\frac{\Gamma, x : \sigma? \vdash P : \tau?}{\Gamma \vdash \mathrm{Z}x.P : \sigma? \to \tau?} \text{ (AbsZ)}$$

In dem gleichen Kontext Γ *ist ebenfalls zu zeigen, dass* $\mathrm{Z}x.P'$ *den Typ* $\sigma? \to \tau?$ *hat:*

$$\frac{\Gamma, x : \sigma? \vdash P' : \tau?}{\Gamma \vdash \mathrm{Z}x.P' : \sigma? \to \tau?} \text{ (AbsZ)}$$

Induktionsschritt 3: Sei $P \to_\beta P'$ *mit* $\Gamma, x : \sigma! \vdash P : \tau$ *und* $\Gamma, x : \sigma! \vdash P' : \tau$, $M = \mathrm{I}x.P$ *und* $N = \mathrm{I}x.P'$ *für beliebige* $x \in \mathbf{V}$.

$$\frac{\Gamma, x : \sigma! \vdash P : \tau}{\Gamma \vdash \mathrm{I}x.P : \sigma! \to \tau} \text{ (AbsI)}$$

In dem gleichen Kontext Γ *ist ebenfalls zu zeigen, dass* $\mathrm{I}x.P'$ *den Typ* $\sigma! \to \tau$ *hat:*

$$\frac{\Gamma, x : \sigma! \vdash P' : \tau}{\Gamma \vdash \mathrm{I}x.P' : \sigma! \to \tau} \text{ (AbsI)}$$

Induktionsschritt 4: Sei $P \to_\beta P'$ *mit* $\Gamma, x : \sigma! \vdash P : \tau$ *und* $\Gamma, x : \sigma! \vdash P' : \tau$, $M = \Delta x.PZ$ *und* $N = \Delta x.P'Z$ *für beliebige* $x \in \mathbf{V}$ *und* $Z \in \tilde{\Lambda}$ *mit* $\Gamma \vdash Z : \tau$.

$$\frac{\Gamma, x : \sigma! \vdash P : \tau \qquad \Gamma, x : \sigma? \vdash Z : \tau}{\Gamma \vdash \Delta x.PZ : \sigma? \to \tau} \text{ (}\Delta\text{-check)}$$

In dem gleichen Kontext Γ *ist ebenfalls zu zeigen, dass* $\Delta x.P'Z$ *den Typ* $\sigma? \to \tau$ *hat:*

$$\frac{\Gamma, x : \sigma! \vdash P' : \tau \qquad \Gamma, x : \sigma? \vdash Z : \tau}{\Gamma \vdash \Delta x.P'Z : \sigma? \to \tau} \text{ (}\Delta\text{-check)}$$

Induktionsschritt 5: Sei $P \to_\beta P'$ *mit* $\Gamma, x : \sigma? \vdash P : \tau$ *und* $\Gamma, x : \sigma? \vdash P' : \tau$, $M = \Delta x.ZP$ *und* $N = \Delta x.ZP'$ *für beliebige* $x \in \mathbf{V}$ *und* $Z \in \tilde{\Lambda}$ *mit* $\Gamma \vdash Z : \tau$.

$$\frac{\Gamma, x : \sigma? \vdash Z : \tau \qquad \Gamma, x : \sigma! \vdash P : \tau}{\Gamma \vdash \Delta x.ZP : \sigma? \to \tau} \text{ (}\Delta\text{-check)}$$

In dem gleichen Kontext Γ *ist ebenfalls zu zeigen, dass* $\Delta x.P'Z$ *den Typ* $\sigma? \to \tau$ *hat:*

$$\frac{\Gamma, x : \sigma? \vdash Z : \tau \qquad \Gamma, x : \sigma! \vdash P' : \tau}{\Gamma \vdash \Delta x.ZP : \sigma? \to \tau} \ (\Delta\text{-}check)$$

Induktionsschritt 6: Sei $P \to_\beta P'$ *mit* $\Gamma \vdash P : \sigma \to \tau$ *und* $\Gamma \vdash P' : \sigma \to \tau$, $M = PZ$ *und* $N = P'Z$ *für beliebige* $Z \in \Lambda$ *mit* $\Gamma \vdash Z : \sigma$. *In beiden Fällen kann der Typ* τ *abgeleitet werden. Für* PZ:

$$\frac{\Gamma \vdash P : \sigma \to \tau \qquad \Gamma \vdash Z : \sigma}{\Gamma \vdash PZ : \tau} \ (App)$$

Für $P'Z$:

$$\frac{\Gamma \vdash P' : \sigma \to \tau \qquad \Gamma \vdash Z : \sigma}{\Gamma \vdash P'Z : \tau} \ (App)$$

Induktionsschritt 7: Sei $P \to_\beta P'$ *mit* $\Gamma \vdash P : \sigma$ *und* $\Gamma \vdash P' : \sigma$, $M = ZP$ *und* $N = ZP'$ *für beliebige* $Z \in \Lambda$ *mit* $\Gamma \vdash Z : \sigma \to \tau$. *In beiden Fällen kann der Typ* τ *abgeleitet werden. Für* ZP:

$$\frac{\Gamma \vdash Z : \sigma \to \tau \qquad \Gamma \vdash P : \sigma}{\Gamma \vdash ZP : \tau} \ (App)$$

Für ZP':

$$\frac{\Gamma \vdash Z : \sigma \to \tau \qquad \Gamma \vdash P' : \sigma}{\Gamma \vdash ZP' : \tau} \ (App)$$

Damit ist nun gezeigt, dass die Typen in diesem Kalkül Invarianten der Programme, die durch Terme repräsentiert werden, darstellen. Dies ist für die Verwendung in der Programmierung essentiell, da es darstellt, dass die Typen, die statisch ermittelt werden, auch zur Laufzeit vorliegen. Das ist insbesondere für die hier vorgestellte Verwendung von Typen interessant. Typen markieren hier den Taint-Status. Würde dieser durch Auswertung der Terme von der statischen Analyse abweichen, wäre die Sicherheit, die durch die Analyse erzeugt werden soll, nicht gegeben.

In diesem Kapitel wurde der $\tilde{\lambda}$-Kalkül eingeführt. Es erweitert den Lambda-Kalkül, indem es weitere Konstruktoren für Quellen, Senken und Sanitization hinzufügt. Zusätzlich wurden wichtige Relationen des Lambda-Kalküls auf diese Erweiterung übertragen. Hierbei wurde gezeigt, dass das Hinzufügen von Konstruktoren nicht ausreicht, um eine statische Taint-Analyse durchzuführen. Daraufhin wurden Typen hinzugefügt, die in der Lage sind, den Taint-Status eines Wertes widerzuspiegeln.

Um diese Typen schließlich zur Analyse von Termen zu verwenden, wurden Ableitungsregeln entwickelt, die den Taint-Status berechnen können, ohne dass das Programm ausgeführt werden muss. Für die so entwickelten Regeln wurde im Anschluss gezeigt, dass sie Invarianten der Programme sind und daher die Aussagen über den Taint-Status, die durch den Typ getroffen werden, zur Laufzeit erhalten bleiben.

An dieser Stelle soll nun einmal auf die Vollständigkeit des Kalküls eingegangen werden. Die Vollständigkeit bezieht sich darauf, dass falls ein verschmutzter Wert in einer Senke verwendet werden soll, dies erkannt wird. Im $\tilde{\lambda}$-Kalkül entspricht dies dem Erkennen eines Typfehlers. Die Frage ist daher, ob ein Term SM getypt werden kann, wenn S eine Senke ist und M ein beliebiger, verschmutzter Term. Senken werden durch I-Abstraktionen dargestellt, es ist also bekannt, dass $S = \mathrm{I}x.S'$ gilt. Hierfür kann lediglich abgeleitet werden:

$$\frac{x : \sigma! \vdash S' : \tau}{\emptyset \vdash \mathrm{I}x.S' : \sigma! \to \tau} \ (\mathtt{AbsI})$$

Es ist also bekannt, dass S den Typen $\sigma! \to \tau$ hat unter der Annahme, dass $x : \sigma! \vdash S' : \tau$ abgeleitet werden kann. Ansonsten ist der Term S nicht typbar. Für M ist bekannt, dass M ein verschmutzter Wert ist, also gilt $M : \gamma?$. Die Frage die nun beantwortet werden muss ist: existiert unter diesen Annahmen ein Typ für SM. Hierzu muss folgende Ableitung möglich sein.

$$\frac{\emptyset \vdash S : \sigma! \to \tau \qquad \emptyset \vdash M : \gamma?}{\emptyset \vdash SM : \delta} \ (\mathtt{App}) \text{ mit } \sigma! = \gamma? \text{ und } \delta = \tau$$

Es muss also gezeigt werden, dass $\sigma! = \gamma?$ gilt. Es ist leicht zu sehen, dass es auf Grund des Taint-Status, nicht möglich ist, dass diese beiden Typen gleich sind. Damit ist gezeigt, dass der Kalkül vollständig ist.[1]

In Kapitel 4 werden die hier gelernten Erkenntnisse verwendet, um sie auf das Programm *wc* anzuwenden. Hierzu werden zuerst einige Regeln auf C übertragen. Die Überlegungen in Kapitel 4 stützen sich dabei auf die Erkenntnisse, die hier gesammelt wurden. Die so gefundenen Regeln können im Anschluss auf *wc* angewendet werden. Ziel ist es eine bekannte Schwachstelle, einen Integer-Overflow, zu finden.

[1] Eine Bereinigung eines verschmutzten Wertes ist ausschließlich durch einen Δ-Check-Point möglich. Ansonsten existiert keine Regel, die das Erzeugen des Typs $\sigma? \to \tau!$ ermöglicht. Die unkontrollierte Bereinigung eines Wertes ist also nicht möglich.

Kapitel 4

Anwendung auf C

Bis zu diesem Punkt der Arbeit wurde auf verschiedene Themen mit Bezug zur Taint-Analyse eingegangen. Das Konzept der dynamischen Taint-Analyse wurde in Abschnitt 2.1 beschrieben und durch ein Beispiel veranschaulicht. Anschließend wurden in Abschnitt 2.2 und Abschnitt 2.3 die Grundzüge des Lambda-Kalküls und der Typtheorie präsentiert. Diese Grundlagen wurden dann in Kapitel 3 zusammengeführt, um ein Kalkül zu entwickeln, das die Prinzipien der Taint-Analyse auf Ebene der Typtheorie bearbeitet.

Dieses Kalkül soll nun in einem praktischen Kontext betrachtet werden. Hierzu sollen die vorher entwickelten Regeln auf die Programmiersprache C übertragen (Abschnitt 4.2) und dann auf das Kommandozeilenprogramm *wc* angewendet werden (Abschnitt 4.3). Auf dieser Basis wird dann der Ansatz, der in dieser Arbeit vorgestellt wurde, in Kapitel 5 evaluiert.

4.1 Einführung in C

In Abschnitt 2.1 wurde die dynamische Taint-Analyse eingeführt und anhand der Beispielsprache ARIT beschrieben. Um die Taint-Analyse in Kapitel 3 vorzustellen, wird das Modell des Lambda-Kalküls um einige Komponenten erweitert. Dies bedingte es, dass Abschnitt 2.2 und Abschnitt 2.3 unterschiedliche Konzepte des Lambda-Kalküls erläutern. In Kapitel 4 soll das in Kapitel 3 präsentierte Modell auf das Programm *wc* angewendet werden. Um ein Verständnis für den dort vorgestellten Programmcode zu gewährleisten, wird an dieser Stelle auf einige Grundkonzepte der Programmiersprache C hingewiesen. Die Konzepte werden anhand von Beispielen dargestellt, an denen die Wirkung auf den Programmzustand beschrieben wird. Der Aufbau orientiert sich dabei an Kernighan und Ritchie [KR88]. Auf diese Arbeit wird auch für eine detailliertere Einführung in C verwiesen.

Ein Programm mit Ausgabe Um eine Ausgabe auf dem Bildschirm auszugeben, werden Informationen aus der Standardbibliothek benötigt. Diese unterscheiden sich zwischen den Plattformen und erlauben den gleichen Programmcode auf allen Plattformen zu verwenden. Um einen Teil der Standardbibliothek einzubinden, werden Kommandos der Form `#include<stdio.h>` verwendet. Diese können auch dazu genutzt werden, eigenen Programmcode aus einer anderen Datei einzubinden. Die generelle Ausführungsreihenfolge eines Programms ist von oben nach unten. Die Einbindung von Dateien erfolgt dadurch immer zu Beginn eines Programms.

Das Kommando zur Ausgabe einer Zeichenkette lautet `printf()`. Hierbei können zur Ausgabe von im Programm erzeugten Ergebnissen, eine Reihe von Sonderzeichen verwendet werden. Im Fall von *wc* werden `%d` und `%s` verwendet. `%d` ermöglicht die Ausgabe eines ganzzahligen Wertes und `%s` die Ausgabe einer berechneten Zeichenkette.

Jedes C-Programm beginnt mit dem Aufruf einer Funktion `main`. Diese Funktion hat den Rückgabetyp `int`, das heißt es wird erwartet, dass die Funktion einen `int`-Wert zurückgibt. `int` steht für integer (*dt.* ganze Zahl) und entspricht eben den ganzen Zahlen. Die Größe ist hierbei plattformabhängig, üblich sind vier Bytes (32 Bits). Das Zurückgeben eines Wertes wird durch das Kommando `return` erreicht, dies beendet außerdem die Funktion.

Abgesehen vom Rückgabetyp, sind auch die Eingabeparameter der Funktion festgelegt. Der erste Eingabeparameter ist hierbei ein `int`, das die Anzahl an Argumenten, die auf der Kommandozeile angegeben wurden, beschreibt. Hierbei wird der Programmname ebenfalls als Argument gezählt. Wird also das Programm *wc* aufgerufen durch „`wc tmp`", würde dem ersten Argument der Wert zwei übergeben werden. Der zweite Parameter der `main`-Funktion ist ein Argument, das Zugriff auf die oben beschriebenen Argumente erlaubt. Die Struktur dieses Parameters wird später weiter ausgeführt, der Typ ist allerdings mit `char**` gegeben. Die Eingabeparameter werden dann mit dem Rückgabetyp und dem Funktionsnamen zu der Signatur der Funktion kombiniert: `int main(int argc, char** argv)`.

Nach der Signatur der Funktion folgt ihr Rumpf. Der Rumpf wird hierbei mit „{" und „}" umschlossen. In diesem Rumpf befindet sich eine Folge von Kommandos, die durch das `return`-Kommando beendet wird. Jedes Kommando wird hierbei durch ein „`;`" beendet. Ein Programm, das die Zeichenkette „Hallo, Welt!1!" ausgibt, hat dann die Form von Listing 4.1.

Variablen und arithmetische Ausdrücke Die Definition einer Variable erfolgt durch das Festlegen von Typ und Name. Dies erfolgt dann durch ein Kommando der Form `typ name;`. Beispielhaft kann eine Variable `a` vom Typ `int` durch das Kommando `int a;` erzeugt werden. Da `int` der, in diesem Programm, dominierende Datentyp ist,

```c
#include<stdio.h>

int main(int argc, char** argv){
  printf("Hallo, %s!%d!", "Welt", 1);

  return 0;
}
```

Listing 4.1.: Ausgabe einer Zeichenkette durch ein C-Programm

```c
#include<stdio.h>

int main(int argc, char** argv){
  int a;
  a = 100;
  int b;
  b = 32;
  printf("100*21+32*2=%d",a*21+b*2);

  return 0;
}
```

Listing 4.2.: Verwendung eines arithmetischen Ausdrucks

wird sich auf diesen beschränkt. Die Zuweisung einer Variable wird durch `var = exp;` erreicht. `exp` kann hierbei ein einfacher arithmetischer Ausdruck, eine Konstante oder ein Funktionsaufruf sein.

Integer können durch die üblichen Rechenoperationen miteinander kombiniert werden. Hierbei ist zu beachten, dass die Division immer abrundet und die Zahlen beschränkt sind. Die genaue Größe ist plattformabhängig, üblich sind aber vier Bytes. Daraus ergibt sich, eine maximale Größe der Integer von $2^{32} - 1 = 4.294.967.295$, wenn nur der positive Zahlenraum betrachtet wird. Bei weiterer Addition führt dies zu einem sogenannten Integer-Overflow, der dem Integer den Wert null gibt.

Arithmetische Ausdrücke können ansonsten auf die übliche Weise konstruiert werden. Die Rechnung $100 * 21 + 32 * 2$ kann durch Listing 4.2 dargestellt werden.

```c
#include<stdio.h>

int f(int x){
  return x * x;
}

int main(int argc, char** argv){
  int a;
  a = 4;

  printf("f(%d)=%d", a, f(a));

  return 0;
}
```

Listing 4.3.: Verwendung einer Funktion

Funktionen Die Definition einer Funktion geschieht auf die gleiche Weise, wie für die `main`-Funktion beschrieben. Hierbei ist nur zu beachten, dass die Definition einer Funktion *vor* ihrer Verwendung erfolgen muss. Der Aufruf einer Funktion erfolgt durch den Namen der Funktion gefolgt von `()`. Sollte die Funktion Eingabeparameter erwarten, werden diese in der, bei der Definition der Funktion angegebenen Reihenfolge und durch Kommata separiert übergeben. Die Definition der Funktion $f(x) = x * x$ und deren Aufruf kann Listing 4.3 entnommen werden.

Kontrolloperatoren C besitzt eine Reihe an Kontrolloperatoren. Kontrolloperatoren bestimmen den Programmfluss z. B. durch Verzweigungen in Abhängigkeit von einer Variable. An dieser Stelle werden zwei Kontrolloperatoren betrachtet, `if` und `while`. Diese beiden Operatoren sind ausreichend, um die Programmabläufe in *wc* zu beschreiben.

Die Struktur des `if`-Operators erlaubt es, anhand eines Ausdrucks, einen von zwei Rümpfen auszuführen. Hierbei ist die Struktur wie in Listing 4.4 dargestellt. Zuerst wird `A` ausgewertet, ist `A` *wahr*, wird der Rumpf `B` ausgeführt, ansonsten `C`. Der `else` Rumpf ist optional.

Der `while`-Operator erlaubt das wiederholte Ausführen eines Rumpfes. Vor der Ausführung des Rumpfes wird hierbei überprüft, ob ein Ausdruck zu *wahr* ausgewertet wird, wenn dies nicht der Fall ist, wird der Rumpf nicht ausgewertet und das Programm setzt die Ausführung mit dem nächsten Kommando fort. Das `break` Kommando ist eine weitere Möglichkeit die wiederholte Ausführung zu beenden, indem die

```
if(A){
  B
} else {
  C
}
```

Listing 4.4.: Der `if`-Kontrolloperator

```
while(A){
  B
}
```

Listing 4.5.: Der `while`-Kontrolloperator

Schleife verlassen und das Programm fortgesetzt wird. Die Syntax des `while`-Operators ist in Listing 4.5 veranschaulicht.

Zeiger Zeiger in C sind eines der „kritischsten Features" [KR88]. Zeiger ermöglichen es den Programmzustand an variablen Stellen im Speicher zu ändern. Ein Zeiger (*eng.* Pointer) bezeichnet dabei eine Speicheradresse. Durch das Dereferenzieren dieser Speicheradresse, kann Zugriff auf den Speicher genommen werden. Der Typ eines Pointers wird mit einem `*` markiert, z. B. `int*`. `int*` bietet hierbei Zugriff auf einen Integer. Ein Zeiger kann durch `&` erzeugt und durch `*` dereferenziert werden. Ein Beispiel findet sich in Listing 4.6.

Zeiger werden in C auch zur Verwaltung von Arrays verwendet. Arrays sind Listen mit einer festen Länge von Werten gleichen Typs. Die Elemente werden hierbei durch ihre Position im Array identifiziert. C stellt dabei Arrays als Zeiger auf das erste Element dar. Durch eine spezielle Syntax wird erlaubt zu einem bestimmten Element in dem Array zu springen. Dies ist möglich, da Arrays in C immer in einem kontinuierlichen Speicherbereich verwaltet werden. Die spezielle Syntax wird durch `array[i]` gebildet, wobei `i` der Index des Elements ist. Die Indizierung in C beginnt mit null.

Das zweite Argument der `main`-Funktion ist ein ebenso beschriebenes Array. Der zweite `*` kommt durch die Schachtelung zustande, da die Elemente des Arrays ebenfalls Arrays sind. Die inneren Arrays sind dabei die Argumente, die auf die Kommandozeile in Form von Zeichenketten übergeben werden. Zeichenketten werden in C durch Arrays von `chars` (*dt.* Zeichen) repräsentiert. Das zweite Argument ist also ein Array von Zeichenketten.

```c
#include <stdio.h>

int main(){

  int a = 100;
  int* b = &a;
  *b = 200;


  printf("%d\n", *b);

  return 0;
}
```

Listing 4.6.: Verwendung eines Pointers zur Manipulation einer fremden Speicheradresse

In diesem Teil der Arbeit wurden die Grundkonzepte der Programmiersprache C beschrieben und mit Beispielen unterlegt. Die auf diese Weise beschriebenen Konzepte werden die Anwendung des in Kapitel 3 beschriebenen Systems auf *wc* erleichtern und ermöglichen es, den Fokus auf die Typen zu legen, die durch die Regeln beschrieben werden sollen.

4.2 Übersetzen einiger Regeln für C

An dieser Stelle werden einige Regeln des in Kapitel 3 beschriebenen Kalküls in den Kontext der Programmiersprache C eingebettet. Dies geschieht auf eine praxisorientierte Weise, die darauf ausgelegt ist die Unterschiede, die entstehen, herauszuarbeiten. Der Fokus liegt an dieser Stelle nicht auf der vollständigen formalen Ausarbeitung dieses Systems, sondern auf der Darstellung der entwickelten Konzepte.

Zunächst sollen aber einige „Besonderheiten" der Programmiersprache C diskutiert werden. Diese beeinflussen der Kalkül in einer Art, sodass dies zu Hindernissen führt. Das Speichermodell stellt hierbei den größten Unterschied zu dem in Kapitel 3 beschriebenen Kalkül da. Im Folgenden sollen zunächst die Folgen des Speichermodells und insbesondere der Zeiger, die von C zur Verfügung gestellt werden beschrieben werden. Dies stellt somit die Grenzen des hier beschriebenen Ansatzes dar.

```
1  !int x = 0;
2  !int *pt = &x;
3
4  ?int f(int a, int* b) {
5    *b = a;
6    return 0;
7  }
```

Listing 4.7.: Ein Zeiger erlaubt schreibenden Zugriff auf eine reine Speicherstelle.

Folgen des Speichermodells Die Programmiersprache C unterscheidet sich in einigen Bereichen von dem formalen Berechnungsmodell des Lambda-Kalküls. Der prägnanteste Unterschied ist das Speichermodell. C, im Kontrast zum Lambda-Kalkül, besitzt Variablen, die während des Programmablaufs verändert werden können. Die hier beschriebenen Folgen gelten für alle Sprachen, die wie C veränderliche Variablen unterstützen. Sprachen, die nur konstante Variablen unterstützen, erlauben den in Kapitel 3 vorgestellte Kalkül in einer direkteren Art zu anzuwenden.

Das Speichermodell bietet auch die Möglichkeit Zeiger auf Speicherstellen zu verwalten. Diese stellen als Argumente für Funktionen ein Problem dar. Dies soll nun näher erläutert werden. In C ist es üblich Zeiger als eine Möglichkeit zu nutzen mehr als einen Wert zurückzugeben, weil ein Zeiger erlaubt die Speicherstelle, auf die er zeigt, zu verändern. Dies erlaubt der aufgerufenen Funktion, Speicher, den sie nicht initialisiert hat, zu verändern. Eine einfache Version des Kalküls verwendet einen Taint-Status für einen Zeiger. Dieser Taint-Status beschreibt sowohl den Status der Adresse als auch den Status des Speichers mit dieser Adresse. Ein solches einfaches Modell ist aber nicht ausreichend, wie durch das folgende Beispiel dargestellt wird (vergleiche Listing 4.7).

Das Problem für diese Version des Kalküls liegt darin, dass die Veränderung von „externem Speicher" in einer Funktion dazu führen kann, dass eine reine Variable mit verschmutzten Daten beschrieben werden kann. Das Szenario wird durch Listing 4.7 verdeutlicht. Zunächst wird eine Speicherstelle initialisiert und als Senke markiert. Wird nun die Funktion `f` mit einer beliebigen Zahl `a` und dem Zeiger `pt` aufgerufen, wird durch die `weak`-Regel der Typ von `pt` zu int $*$? geändert und daher ist die Zuweisung in der Funktion zulässig. Dadurch kann dem Speicher von `x` ein verschmutzter Wert zugewiesen werden, ohne dass dies außerhalb der Funktion `f` am Typen zu erkennen ist. Das heißt manche Fehler, die durch Zeiger hervorgerufen werden, werden momentan nicht erkannt. Um diese zu erkennen, wären weitere Regeln für Zeiger nötig. Es werden daher nur Programme betrachtet, die Zeiger eingeschränkt nutzen.

```
1    int a = 100; // als !int
2    if(b){
3      a = a + 1;
4    }
```

Listing 4.8.: Eine `if`-Verzweigung

Regeln, die das Verwenden von Zeigern erlauben, müssten Information über den Taint-Status der Adresse, die im Zeiger abgelegt ist, und über die Speicherstelle an dieser Adresse beinhalten. Die Indirektion, die durch die Zeiger ausgedrückt wird, muss also auf Typebene ebenfalls abgebildet werden. Ein solcher Ansatz würde weiterhin keine Zeiger-Arithmetik betrachten, weil dies eine Modellierung des Speichermodells voraussetzt. Aus diesen Gründen wurden im Rahmen der Arbeit Zeiger nicht näher betrachtet. Arrays wurden in der Arbeit ebenfalls nicht beachtet, da sie in C den Umgang mit Zeigern voraussetzen. Im Folgenden werden die Regeln für eine Teilmenge von C beschrieben.

`if`-Verzweigung Dadurch, dass Variablen veränderlich sind, kann sich der Taint-Status des Wertes einer Variable von dem Status der Variable unterscheiden. Der Grund dafür liegt in der möglichen Verschmutzung eines Blocks durch eine externe Bedingung. Diese Verschmutzung entsteht durch die Abhängigkeit des Blocks von der Bedingung. Formal ist ein Block von einer Bedingung abhängig, wenn die Ausführung des Blocks von der Bedingung abhängt [FOW87]. Dies wird auch Kontrollflussverschmutzung (*eng.* Control-flow Taint) genannt und führt zu einem Problem für die dynamische Taint-Analyse [SAB10]. Da die dynamische Taint-Analyse in einer Ausführung nur einen Ausführungspfad betrachtet, können nicht alle Abhängigkeiten korrekt ermittelt werden. Hier wird dieses Problem umgangen, da der entscheidende Teil der Abhängigkeiten, der Taint-Status, durch die Typen kodiert ist. An dieser Stelle wird ein Beispiel in Form einer `if`-Verzweigung betrachtet (Listing 4.8).

Wenn die Variable `b` nun verschmutzt ist, dann bedeutet das, dass alle Berechnungen in dem Block auch verschmutzt sind. Gleichzeitig darf der Variable `a` aber kein verschmutzter Wert zugeordnet werden. Die Folge ist, dass der Wert von `a` verschmutzt ist, während der Typ von der Variable `a` als rein markiert ist.

Um diesen Sachverhalt zu beschreiben, werden zwei Konstrukte verwendet, die beide einen Kontext darstellen. Der eine Kontext stellt die Typen der Variablen dar, der andere repräsentiert die Typen der Werte der Variablen. Hierbei gilt, dass eine verschmutzte Variable keinem reinem Wert zugeordnet werden kann.

Eine `if`-Verzweigung kann also getypt werden, wenn `b` ein boolescher Wert ist und die Blöcke der `if`-Verzweigung in den angepassten Kontext getypt werden können. Der Kontext wird dabei entsprechend dem Taint-Status von `b` angepasst. Es ist aber zu beachteten, dass nur der Kontext der Werte angepasst wird.

`while`-Verzweigung Die gleichen Regeln gelten für `while`-Verzweigungen. Wie bei der `if`-Verzweigung wird der Kontext in Abhängigkeit von `b` angepasst. In beiden Fällen müssen die Blöcke getypt werden. Der Typ, der dabei für die Blöcke erwartet wird, ist ein Unit-Typ (*dt.* Einheitstyp). Es wird also erwartet, dass in den Blöcken lediglich Zustandsänderungen durchgeführt werden. Der gesamte Typ der `if`- und `while`-Verzweigungen ist dann auch der Unit-Typ. Der Einheitstyp repräsentiert Zustandsänderungen.

Variablendeklaration und -zuweisung Auf der niedrigsten Ebene werden Zustandsänderungen durch das Einführen und Verändern von Variablen durchgeführt. Die Einführung einer Variable ist immer durch das Angeben eines Typen und eines Namens möglich. Diese werden dann in den Kontext eingefügt. Die Veränderung einer Variable ist möglich, wenn die Variable in dem Kontext existiert und der Wert, der der Variable zugeordnet werden soll, den gleichen Typ hat wie die Variable in dem Kontext. Bei einer Variable, die nicht als Senke oder Quelle markiert wird, ist es möglich, dass die Variable als rein oder als verschmutzt markiert wird. Hierzu wird wieder ein Beispiel betrachtet (Listing 4.9). Dort wird eine Variable `!int x` als Senke markiert und eine Variable `?int z` als Quelle markiert. Eine Variable, die auf die bekannte Weise `int y` deklariert wird, kann sowohl als rein betrachtet werden als auch als verschmutzt. Es werden also anschließend zwei mögliche Kontexte für diese Variable betrachtet. Dies bildet einen großen Unterschied zur dynamischen Taint-Analyse, die einer Variable immer nur einen Taint-Status zur Laufzeit zuschreibt [SAB10]. Die hier beschriebene Notation ist dem Versuch entsprungen, die Anzahl an nötigen Veränderungen der Syntax zu minimieren. Es ist zu beachten, dass auf der Syntaxebene von C die Markierung *vor* dem Typen angeben wird. Dies erlaubt die effiziente Implementierung eines Parsers, unterscheidet sich aber ansonsten nicht von der bisherigen Betrachtung mit der Markierung dem Typen folgend. Aus diesem Grund wird, außer in Programmcode, weiterhin die Notation, in der die Markierung dem Typen folgt, verwendet.

```
!int x; // x : int! wird hier als Senke markiert
int  y; // y : int? und y : int! können beide abgeleitet werden
?int z; // z : int? wird hier als Quelle markiert
```

Listing 4.9.: Variablen mit Markierung und ohne Markierung

```
!int fak(int a) {
  int result = 1;
  if(a > 1) {
    result = a * fak(a-1);
  }
  return result;
}
?int fak(int a) {
  int result = 1;
  if(a > 1) {
    result = a * fak(a-1);
  }
  return result;
}
```

Listing 4.10.: Definition von Quellen und Senken

Funktionsdeklaration Aus einem ähnlichen Grund ist die Syntax für die Definition von Funktionen hervorgegangen. Die meist vorkommende Definitionsform entspricht genau der Schreibweise aus Abschnitt 4.1. Einer so definierten Funktion können zwei Typen zugeordnet werden. Der eine Typ entspricht der Verschmutzung aller Eingaben, sowie der Ausgabe. Wenn alle Eingaben rein sind, dann ist auch die Ausgabe rein.

Wie im $\tilde{\lambda}$-Kalkül werden zwei Formen von Funktionen markiert, die Quellen und Senken darstellen. Hierzu wird die Syntax von C durch Markierungen erweitert. Diese werden beispielhaft in Listing 4.10 dargestellt. Durch diese Erweiterung können Quellen und Senken als solche identifiziert werden. Die Typableitung arbeitet dabei wie bei gewöhnlichen Funktionen mit dem Unterschied, dass die Typen der Ein- und Ausgaben,wie im $\tilde{\lambda}$-Kalkül beschrieben, festgesetzt sind. Die Quelle in Listing 4.10 hat dann den Typ $int? \to int?$ und die Senke $int! \to int!$.

Die Ausdrücke in den Funktionsrümpfen müssen alle mit dem Einheitstypen getypt werden können. Eine Ausnahme bildet das Argument für das `return`-Kommando. Das Argument muss mit dem Rückgabetypen der Funktion übereinstimmen. In Listing 4.10 muss `result` also den Typ $int!$ beziehungsweise $int?$ besitzen.

Overtainting in C Hier fällt ein Unterschied zu dem Kalkül auf, das in Kapitel 3 beschrieben wurde. Funktionen sind in C „abgeschlossen". Dies bedeutet, dass eine Funktion nur einen einfachen Wert zurückgibt, der keine Funktion sein kann. Dadurch können einfache Verschmutzungen einer Eingabe nicht einfach an die Ausgabe weitergegeben werden. Es wird also davon ausgegangen, dass entweder alle Eingaben rein sind oder alle Eingaben verschmutzt. Dies entspricht einer Vereinfachung. Theoretisch können sämtliche Kombinationen der Taint-Status der Eingaben mit nur einer Verschmutzung, zu einer verschmutzten Ausgabe führen. Beispielsweise ist das Ergebnis der Multiplikation $a * b$ verschmutzt, wenn:

- a verschmutzt ist
- b verschmutzt ist
- a und b verschmutzt sind.

Im $\tilde{\lambda}$-Kalkül könnten für die Multiplikation dann folgende Typen abgeleitet werden:

- $\mathbb{N}! \to \mathbb{N}! \to \mathbb{N}!$
- $\mathbb{N}! \to \mathbb{N}? \to \mathbb{N}?$
- $\mathbb{N}? \to \mathbb{N}! \to \mathbb{N}?$
- $\mathbb{N}? \to \mathbb{N}? \to \mathbb{N}?$
- $\mathbb{N}! \to \mathbb{N}! \to \mathbb{N}?$

Das Kalkül für C leitet lediglich den ersten und vierten Typ ab. Dies ist relevant, da es das Overtainting in der Taint-Analyse etwas „versteckt".

Δ-Check-Point Ein noch nicht betrachtetes Konzept, ist das Δ-Check-Point-Konzept. Die so dargestellte Bereinigung von Werten wird hier, im Vergleich zur dynamischen Taint-Analyse, explizit durchgeführt, weil das statische Erkennen von Bereinigunen nicht immer möglich ist. In Kapitel 3 wurde die Verwandtschaft zu einer `if`-Verzweigung behandelt. Sie motivierte die Ableitungsregel für die Δ-Check-Point-Konstruktion. An dieser Stelle fällt auf, dass bereits eine Regel für `if`-Verzweigungen existiert, die bestimmt wie sie in C typisiert werden können. Es wäre also eine Erweiterung der Syntax von Nöten, um dieses Konzept auf C zu übertragen. Dabei sind zwei Interpretationen denkbar. Die eine übersetzt die motivierende `if`-Verzweigung *direkt* in dieses neue Konzept. Hierfür würde `if` zum Beispiel durch ein neues Schlüsselwort, wie `clean` ersetzt werden, dadurch würde markiert werden, dass diese Verzweigung eine besondere Semantik hat. Alternativ wäre es möglich dies näher an den Überlegungen aus Kapitel 3 zu konstruieren. Diese Verzweigung würde also in eine spezielle Art von Funktion ausgelagert werden, die, wie die Δ-Abstraktion, zwei Rümpfe besitzt. Beides sind mögliche Herangehensweisen.

```
1     clean x (x != 0) {
2       //x : int!
3     } else {
4       //x : int?
5     }
```

Listing 4.11.: Definition von Quellen und Senken

In dies Arbeit wurde sich für die erste Variante entschieden. Es wurde also die Syntax von C um eine Konstruktion erweitert. Dieses entspricht einer `if`-Verzweigung. Allerdings wurde diese modifiziert. Vor der Bedingung befindet sich der Variablenname, der durch diesen Check-Point bereinigt wird. Auf die Bedingung in Klammern folgen zwei Blöcke in geschweiften Klammern. Der erste beschreibt das Vorgehen in dem Fall, dass die Variable die Bedingung erfüllt, um als rein interpretiert zu werden, während der zweite sie nur verschmutzt verwenden kann. Ein Beispiel kann in Listing 4.11 betrachtet werden. Zur Ausführung verhält sich dieser Code exakt wie die `if`-Verzweigung mit der gleichen Bedingung.

Die in diesem Abschnitt beschriebenen Regeln lassen sich am besten in einer Anwendung nachvollziehen. Dies wird im Folgenden vorgestellt. Hierzu wird schrittweise das Programm *wc* betrachtet. *wc* wurde als Beispiel ausgewählt, da es sich zum einen um ein Programm handelt, das im täglichen Gebrauch realistisch verwendet wird. Zum anderen bietet *wc* eine gute Balance der Programmgröße. Es ist groß genug, sodass es nicht zu einfach ist, es ist aber auch klein genug, sodass der Programmcode leicht verständlich ist und es keiner großen Erläuterung des Programmablaufs bedarf.

4.3 Identifikation einer Schwachstelle in *wc*

In diesem Abschnitt sollen die erworbenen Kenntnisse der Taint-Analyse auf Basis von Typen auf das Tool *wc* angewendet werden. Hierzu wird zuerst der vollständige Programmablauf beschrieben. Der gesamte Programmcode kann Anhang D entnommen werden. Im Anschluss werden schrittweise die einzelnen Programmblöcke betrachtet und die unterschiedlichen Regeln angewendet.

Die Aufgabe von *wc* ist es die Anzahl von Zeilen, Wörtern und Bytes in einer Datei zu zählen. *wc* geht so vor, dass über alle als Argument übergebenen Dateinamen iteriert wird. Für jeden dieser Dateinamen gilt das gleiche Verfahren: Die Datei wird geöffnet und Variablen zum Zählen der Zeilen, Wörter und Bytes werden initialisiert. Im Anschluss wird über den Dateiinhalt iteriert. Hierbei wird jedes Wort einzeln betrachtet. Ein Wort wird durch die Funktion `isword()` definiert. Im Anschluss wird

```
1  /* Sample implementation of wc utility. */
2
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <stdarg.h>
6  #include <ctype.h>
7
8  typedef unsigned int count_t;  /* Counter type */
```

Listing 4.12.: Includes und Typedeclarationen für *wc*.

bis zum Ende der Zeile über mögliche Leerzeichen iteriert. Währenddessen werden die zu Beginn initialisierten Variablen hochgezählt. Zum Abschluss werden die entsprechenden Variablen ausgegeben.

Nun sollen die Programmblöcke im Detail bearbeitet werden. Zu Beginn einer Taint-Analyse müssen allerdings Quellen und Senken identifiziert werden. Als Quelle werden hier die Eingabeargumente betrachtet. Dies geschieht aus zwei Gründen. Zum einen sind diese Argumente eine Eingabemöglichkeit, es ist also legitim diese Variablen als verschmutzt zu betrachten. Zum anderen führt dies zu einer weiteren Indirektion im Porgrammfluss, was das Beispiel schwieriger macht. Der Datenfluss muss durch diese Maßnahme länger verfolgt werden. Eine zusätzliche Quelle liegt mit der Funktion `fopen()` vor. Diese Funktion öffnet eine Datei und bildet daher auch eine Form von Eingabe.

Die Senke in diesem Beispiel wird durch die zählenden Variablen gebildet. Der Grund hierfür ist, dass diese ohne weitere Kontrollen verändert werden, potentiell könnte also eine „ungünstige" Eingabe zu falschen Ergebnissen führt. Tatsächlich wird dieser Test zeigen, dass eine Schwachstelle in diesem Programm existiert. Weitere Nachforschungen können dann zu der Erkenntnis führen, dass ein Integer-Overflow vorliegt.

Listing 4.12 zeigt den Programmkopf von *wc*. An dieser Stelle werden nur die benötigten Bibliotheken importiert und der Typ der Zählvariablen umbenannt. Für die Taint-Analyse geschieht an dieser Stelle keine interessante Veränderung des Programmzustands.

Listing 4.13 deklariert die Variablen, die global von dem Programm verwendet werden. An dieser Stelle wurden die nötigen Veränderungen vorgenommen, um die Variablen als Senken des Programms zu identifizieren. Durch diese Programmzeilen werden die Variablen mit den entsprechenden Typen nun in den Kontext aufgenommen.

```
1     /* Current file counters: chars, words, lines */
2     !count_t ccount;
3     !count_t wcount;
4     !count_t lcount;
5
6     /* Totals counters: chars, words, lines */
7     !count_t total_ccount = 0;
8     !count_t total_wcount = 0;
9     !count_t total_lcount = 0;
```

Listing 4.13.: Initialisierung/Deklaration der Zählvariablen

```
1     static void errf(char *fmt, ...) {
2     va_list ap;
3
4     va_start(ap, fmt);
5     error_print(0, fmt, ap);
6     va_end(ap);
7   }
```

Listing 4.14.: Eine Hilfsfunktion.

Nun soll beispielhaft eine der Hilfsfunktionen betrachtet werden. Listing 4.14 zeigt eine solche Hilfsfunktion. Da sie weder als Quelle noch als Senke deklariert wurde, kommen im Prinzip zwei Typen in Frage. Einer erwartet reine Eingaben und wandelt diese in einen reinen Wert um und der andere erwartet verschmutzte Eingaben und gibt den Taint-Status an die Ausgabe weiter. In beiden Fällen muss der Körper dann entsprechend in einem neuen Kontext getypt werden. Da keine verschmutzten Variablen von außen verwendet werden, hängt der Taint-Status der Ausgabe ausschließlich von den Eingaben ab und die Typableitung ist möglich. Es werden also zwei Typen für diese Funktion dem Kontext hinzugefügt: `errf` : char $*$? $\rightarrow$ void? und `errf` : char$*$! $\rightarrow$ void!.

An dieser Stelle soll die Kernfunktion `getword()` betrachtet werden. Diese Funktion iteriert über jedes Wort und manipuliert die Zählvariablen entsprechend. Der Code kann Listing 4.16 entnommen werden. Da diese Funktion die Variablen, die in diesem Beispiel Senken darstellen, manipuliert, werden hier Restriktionen durch die Typen sichtbar. Auch diese Funktion hat keine weiteren Markierungen, es sind also zwei Typen grundsätzlich denkbar, einer mit verschmutzten Ein- und Ausgaben und einer mit reinen Ein- und Ausgaben. Zunächst werden die reinen Eingaben betrachtet. Es werden keine Quellen verwendet, was dazu führt, dass auch keine

```
1     c = getc(fp);
2     while(c != EOF){
3     if(isword(c)){
4         wcount = wcount + 1;
5         break;
6     }
7     ccount = ccount + 1;
8     if((c) == '\n'){
9       lcount= lcount + 1;
10    }
11
12    c = getc(fp);
13    }
```

Listing 4.15.: Ausschnitt aus der Funktion `getword()`.

Verunreinigung in das Programm eindringen kann. Dieser Typ ist ableitbar, es kann also `getword` : FILE∗! → *int*! zum Kontext hinzugefügt werden. Ein Problem tritt bei einer verschmutzten Eingabe auf. Hierzu soll der Ausschnitt aus Listing 4.15 genauer betrachtet werden. An dieser Stelle wird einer Variable `c` ein neuer Wert zugewiesen. Dieser entsteht aus `fp`, der Variable, die die verschmutzte Eingabe der Funktion hält. Der Typ von `c` muss entsprechend diesen Taint-Status widerspiegeln. Durch die Verwendung von `c` in dem Ausdruck `c != EOF` überträgt sich dieser Status auf die `while`-Verzweigung. Die oben beschriebene Regel sieht in diesem Fall vor, dass der Kontext der Werte angepasst werden muss. Das heißt eine Verwendung der Variable `wcount` ist an dieser Stelle verunreinigt, obwohl der Typ der Variable selbst Reinheit beschreibt. Wichtig ist hier die Unterscheidung zwischen der Variable und dem Wert. Dieser Umstand führt nämlich dazu, dass für die Zeile `wcount = wcount + 1` kein Typ abgeleitet werden kann, da `wcount + 1` durch die Manipulation des Kontextes nun verschmutzt ist, der Variable `wcount` aber weiterhin nur reine Werte zugewiesen werden können. Es entsteht also ein Typfehler. Dadurch kann dem Kontext kein Typ hinzugefügt werden, bei dem die Eingabe verschmutzt ist.

Nun soll die Funktion betrachtet werden, die den Zählprozess für einen Dateinamen beginnt. Hier gilt das Gleiche wie für die Funktion `getword()`. Grundsätzlich sind zwei Eingabetypen möglich, die sich dann auf die Ausgabe übertragen. Zusätzlich handelt es sich aber bei `fopen()` um eine weitere Quelle. Das bedeutet, dass `fp` immer den Typen FILE ∗ ? besitzt. Das Problem tritt dann in der Zeile `while(getword(fp))` auf. An dieser Stelle soll `fp` als Eingabe der Funktion `getword` verwendet werden. Der Typ von `getword` erwartet aber, auf Grund der vorigen Ausführungen einen Wert des Typen FILE∗!. Dies führt zu einem Typfehler und dazu, dass *kein* Typ für die Funktion

```
1     int getword (FILE *fp) {
2     int c;
3     int set_result = 0;
4     int result = 0;
5
6     if(feof(fp)){
7       set_result = 1;
8     }
9
10    c = getc(fp);
11    while(c != EOF){
12    if(isword(c)){
13        wcount = wcount + 1;
14        break;
15    }
16    ccount = ccount + 1;
17    if((c) == '\n'){
18      lcount= lcount + 1;
19    }
20
21    c = getc(fp);
22    }
23
24    while(c != EOF){
25      ccount= ccount + 1;
26      if((c) == '\n'){
27    lcount = lcount + 1;
28      }
29      if(!isword(c)){
30    break;
31      }
32      c = getc(fp);
33    }
34    if(!set_result){
35      result = c != EOF;
36    }
37    return result;
38  }
```

Listing 4.16.: Die Funktion `getword`.

```
1  void counter(char *file) {
2    FILE *fp = fopen(file, "r");
3
4    if(!fp){
5      perrf ("cannot open file `%s'", file);
6    }
7    lcount = 0;
8    wcount = 0;
9    ccount = 0;
10   while(getword(fp)){
11     ;
12   }
13   fclose(fp);
14
15   report(file, ccount, wcount, lcount);
16   total_ccount = total_ccount + ccount;
17   total_wcount = total_wcount + wcount;
18   total_lcount = total_lcount + lcount;
19 }
```

Listing 4.17.: Die Funktion `counter()`.

```
1  !void incc(int c) {
2    ccount = c + 1;
3  }
```

Listing 4.18.: Die Funktion `incc()`.

`counter()` ermittelt werden kann. Dadurch wiederum kann das ganze Programm nicht kompiliert werden und das Programm wird als fehlerhaft zurückgewiesen. Durch ein ausführlicheres Beschreiben des Fehlers kann aber auch die Fehlerursache, die Verwendung einer verschmutzten Variable als Eingabe der Funktion `getword()`, kommuniziert werden, was dem Entwickler helfen kann solche Fehler zu vermeiden.

Nachdem dieser Fehler identifiziert werden konnte, stellt sich die Frage, wie der Fehler behoben werden kann. Hierzu wird im Folgenden ein Check-Point verwendet. Zunächst werden aber einige Veränderungen vorgenommen. Zuerst werden die Inkrementierungen in `getword()` in eigene Funktionen ausgelagert. Dadurch ist es möglich die Zustandsänderungen in Senken zu verschieben. Eine solche Funktion kann zum Beispiel wie in Abschnitt 4.3 aussehen. Daraus ergibt sich dann eine neue

Variante der Funktion `getword()` wie in Listing 4.19 zu sehen. Diese Veränderung sorgt nicht dafür, dass nun die Funktion mit einem `FILE*?` aufgerufen werden kann. Da die Funktion `incc(), incw()` und `incl()` als Senken markiert sind, müssen die Eingaben rein sein. Dies ist durch die Verunreinigung der Blöcke weiterhin nicht gegeben. Allerdings existiert die Möglichkeit, dass die Variablen gesäubert werden. Hierzu kann ein Check-Point genutzt werden.

Dies könnte so aussehen, dass vor der Verwendung der Zählvariable Check-Points eingeführt werden, die sicherstellen, dass kein Integer-Overflow auftritt. Dies würde dann zu Code wie in Listing 4.20 führen. Damit würde verhindert werden, dass ein Programm entsteht, in dem ein Integer-Overflow auftritt. Für die modifizierte Variante von `getword()` können dann die Typen FILE$*$! $\rightarrow$ *int*! und FILE $*$? $\rightarrow$ *int*? abgeleitet werden.

An diesem Beispiel kann man also sehen, dass wenn keine Verschmutzung der Funktion übergeben wurde, richtig erkannt wird, dass keine Verschmutzung auftritt. Dies führte schon in der nicht veränderten Variante von `getword()` (Listing 4.16) dazu, dass nur der Typ abgeleitet werden konnte, der keine Verschmutzung in die Funktion eindringen lässt, was der Funktion erlaubt korrekt zu arbeiten. In der veränderten Variante konnten durch Kontrollen, die statisch verifiziert werden, sichergestellt werden, dass das Programm auch bei Verunreinigungen korrekt arbeitet.

Damit ein Entwickler in der Lage ist diesen Kalkül während der Arbeit zu verwenden, muss ein Werkzeug existieren, das den Kalkül umsetzt. Hierzu wurde im Rahmen der Arbeit ein Prototyp entwickelt, der in der Lage ist Programmcode mit Markierungen von Quellen und Senken zu lesen und in einen Syntaxbaum zu transformieren. Dieser Baum kann anschließend von einem Type-Checker, der die oben beschriebenen Regeln implementiert, analysiert werden. Dabei werden Typen hergeleitet und wenn kein Typ gefunden werden kann, wird dies als Fehler gemeldet.

Mit Hilfe dieses Checkers wurde außerdem das Kommandozeilen-Programm *head* überprüft. Dadurch wurde erkannt, dass die Eingabe, in diesem Fall die Zeilen der Datei, nicht überprüft werden. Das heißt zum Beispiel, wenn das Ziel ist zu verhindern, dass Steuerzeichen „geschrieben” werden, geschieht dies nicht.

Eine solche Überprüfung kann durch das automatische Einlesen des Programms schnell durchgeführt werden. Die größte Schwierigkeit liegt in der Auswahl der Quellen und Senken. Diese Auswahl muss von dem Entwickler getroffen werden.

```
1   int getword (FILE *fp) {
2   int c;
3   int set_result = 0;
4   int result = 0;
5
6   if(feof(fp)){
7     set_result = 1;
8   }
9
10  c = getc(fp);
11  while(c != EOF){
12  if(isword(c)){
13        incw(wcount);
14        break;
15  }
16  ccount = ccount + 1;
17  if((c) == '\n'){
18    incl(lcount);
19  }
20
21  c = getc(fp);
22  }
23
24  while(c != EOF){
25    incc(ccount);
26    if((c) == '\n'){
27      incl(lcount);
28    }
29    if(!isword(c)){
30  break;
31    }
32    c = getc(fp);
33  }
34  if(!set_result){
35    result = c != EOF;
36  }
37  return result;
38 }
```

Listing 4.19.: Die Funktion `getword`.

```
1     clean ccount (ccount < INTMAX) {
2       incc(ccount);
3     } else {
4     }
```

Listing 4.20.: Check-Point um `ccount` zu bereinigen.

Kapitel 4 hat das theoretisch erarbeitete Kalkül aus Kapitel 3 auf die Programmiersprache C übertragen und gezeigt, wie ein solcher Algorithmus auf einem Programm arbeiten würde und wie er genutzt werden kann, um Fehler in einem Programm zu identifizieren. In Kapitel 5 werden die Nützlichkeit, Einschränkungen und Verbesserungsmöglichkeiten dieses Ansatzes erläutert und evaluiert.

Kapitel 5

Fazit

In der vorliegenden Arbeit wurde die dynamische Taint-Analyse mit Hilfe von Typen statisch realisiert und auf C übertragen. Diese Taint-Analyse unterscheidet sich von den gängigen Ansätzen in der Literatur durch den Aufbau auf der Typtheorie. Um die Überlegungen verständlich zu erläutern, wurden zuerst die Grundkonzepte der dynamischen Taint-Analyse, des Lambda-Kalküls und der Typtheorie beschrieben.

Anschließend wurde auf dieser Basis das $\tilde{\lambda}$-Kalkül entwickelt. Dieses dient als formales Berechnungsmodell, in dem die Taint-Analyse entwickelt wurde und ermöglicht eine formale Betrachtung der Analyse. Die Taint-Analyse wurde dabei in Form von Ableitungsregeln der Typtheorie vorgestellt und mit Beispielen veranschaulicht. Zusätzlich wurde gezeigt, dass die Typen Invarianten des Programms darstellen. Um dies zu beweisen, ist es nötig Subjektreduktion zu zeigen. Um den Beweis verständlich zu gestalten wurde bereits in Abschnitt 2.3 das entsprechende Theorem für einfache Typen gezeigt.

Um diesen Ansatz zu evaluieren, wurden diese Überlegungen auf das Programm *wc* angewendet. Hierzu mussten einige Regeln auf Grund der C Semantik neu interpretiert werden. Die Anwendung hat gezeigt, dass dieser Ansatz die Taint-Analyse korrekt durchführt, aber immer noch einige Modifikationen an dem eigentlichen Programm erforderlich sind. Außerdem konnte eine Schwachstelle identifiziert werden, die sich bei näherer Betrachtung als ein Integer-Overflow herausstellt. Nun soll die typbasierte Taint-Analyse anhand der Anwendung auf *wc* evaluiert werden. Anschließend werden mögliche zukünftige Arbeiten identifiziert, die auf der Basis des hier vorgestellten Ansatzes bearbeitet werden könnten.

5.1 Evaluation

Zu Beginn lässt sich zunächst feststellen, dass die Schwachstelle in dem Programm *wc* gefunden wurde. Allerdings wurde nicht der Charakter der Schwachstelle als Integer-Overflow erkannt, da dies ein ausdrucksstärkeres Typsystem benötigt. Die Ausdrucksstärke des $\tilde{\lambda}$-Kalküls ist im Wesentlichen nicht größer als die Ausdrucksstärke der einfachen Typen aus Abschnitt 2.3. Die Eigenschaft, die durch die einfachen Typen beschrieben wird, ist lediglich die Funktionsverkettung, diese ist nicht ausdruckstark genug, um Integer-Overflows zu erkennen. Eine alternative Formulierung dieser Einschränkung ist, dass dieses Kalkül zwar die Verwendung von verschmutzten Variablen erkennt, aber nicht die Folgen der Verwendung beschreiben kann. Die Vermutung ist hierbei, dass diese Beschränkung durch das erweiterte Typsystem, in diesem Fall einfache Typen, gegeben wird.

Der Berechnungsaufwand wird ebenfalls in wesentlichen Teilen durch das erweiterte Typsystem bestimmt. Im Vergleich zu den einfachen Typen ist aber festzustellen, dass sich die Anzahl der Typen eines Ausdrucks verdoppelt. Zum einen die Variante für bereinigte Werte und zum anderen die Variante mit verschmutzten Werten. Ausnahmen bilden die Ausdrücke, die Quellen und Senken darstellen. Hierbei handelt es sich also um eine Abschätzung nach oben. Durch diese Verdopplung auf Typebene steigt aber auch der Berechnungsaufwand für das erweiterte Typsystem. Und zwar steigt dieser durch die Verdopplung auf jeder Ebene exponentiell im Vergleich zu dem ursprünglichen Typsystem. Einfache Typen erlauben eine Typ-Rekonstruktion in $\mathcal{O}(n)$ in der Größe des Terms [Sør06]. Daraus ergibt sich eine Typ-Rekonstruktion in $\mathcal{O}(2^n)$ für die hier vorgestellte Erweiterung.[1]

Diese Laufzeit könnte verbessert werden, wenn früher entschieden werden könnte, welchen Typ ein Ausdruck haben muss. Diese Entscheidung wird allerdings nicht vorgenommen. Es werden Typen erst ausgeschlossen, wenn diese dazu führen, dass die Typ-Rekonstruktion an anderer Stelle nicht fortgeführt werden kann. Dies geschieht genau dann, wenn keine Ableitungsregel mehr angewendet werden kann. Alle von dem Kalkül getroffenen Entscheidungen leiten sich also aus den Ableitungsregeln ab.

Durch diese Regeln entsteht außerdem eine Überapproximation. Diese Überapproximation führt dazu, dass Taint an Stellen identifiziert wird, an denen erkannt werden könnte, dass keine Verschmutzung vorhanden ist. Hierzu wird der Kalkül betrachtet, der in Kapitel 3 vorgestellt wurde. Die Beispiele, die hier betrachtet werden, übertragen sich auf die Übersetzung für C. In dem $\tilde{\lambda}$-Kalkül führt eine verschmutzte Eingabe in einer Funktion zu einer Verschmutzung in der Ausgabe. Bei einer Funktion, die also eine Eingabe bekommt, diese aber nie verwendet, führt das dazu, dass die Ausgabe als verschmutzt markiert wird, obwohl dies nicht zutrifft. Das einfachste Beispiel

[1] Dies wurde nicht formal gezeigt.

ist die Funktion $\lambda x.\lambda y.x$. Diese Funktion gibt immer die erste Eingabe zurück und ignoriert die zweite. Im $\tilde{\lambda}$-Kalkül können folgende Typen abgeleitet werden:

1. $\alpha! \to \beta! \to \alpha!$
2. $\alpha? \to \beta! \to \alpha?$
3. $\alpha! \to \beta? \to \alpha?$
4. $\alpha? \to \beta? \to \alpha?$
5. $\alpha! \to \beta! \to \alpha?$

Die Überapproximation tritt im dritten Fall auf. Hier ist die Ursache für die Verschmutzung die Verschmutzung des zweiten Arguments, da dieses aber nicht im Rumpf der Funktion verwendet wird, ist dieses unerheblich für den Typ der Funktion. Der Kalkül erkennt dies nicht und leitet den Typ ab. Eine Überapproximation tritt ebenfalls auf, wenn die `weak`-Regel verwendet wird (siehe fünfter Fall). Da aber weiterhin der „strengere" Rückgabetyp für die Kombination von Eingabetypen abgeleitet werden kann, handelt es sich dabei, um eine kleinere Einschränkung.

Um diese Überapproximation zu bewerten, stellt sich also die Frage, ob der skizzierte Fall, dass eine Eingabe in einer Funktion nicht verwendet wird, häufig auftritt. Hierzu kann natürlich der Standpunkt eingenommen werden, dass wenn eine Eingabe nicht verwendet wird, dann sollte sie auch gar nicht Eingabe der Funktion sein. Dies ist sicherlich nicht vollständig falsch, vergisst aber einen Aspekt, der häufiger auftritt und zwar, wenn die Eingaben nicht durch den Implementierer einer Funktion festgelegt werden. Dies tritt zum Beispiel im Fall von Interfaces auf. Ein Interface gibt die Eingaben einer Funktion vor, aber nicht die Implementierung. In diesem Szenario kann die Überapproximation häufig auftreten, während zu erwarten ist, dass bei den meisten anderen Funktionen die Überapproximation keine Rolle spielt.

Diese Eigenschaften sind ein inhärenter Teil des vorgestellten Kalküls, einige Einschränkungen treten aber durch den Charakter der Taint-Analyse auf. Die Taint-Analyse verfolgt den Datenfluss von einer Quelle zu einer Senke. Es wird also implizit erwartet, dass Quellen und Senken existieren. Es stellt sich die Frage, wie Quellen und Senken erkannt werden können. Die übliche Antwort ist, dass diese durch den Entwickler markiert werden. Der gleiche Ansatz wurde auch hier verwendet, was zur Folge hat, dass ein Programm, das untersucht werden soll, immer modifiziert werden muss, um diese Bereiche des Programms zu identifizieren. Diese Einschränkung wurde also von der dynamischen Taint-Analyse übernommen und ist keine Einschränkung des Kalküls an sich. Um dies zu quantifizieren wurde der Kalkül auf die Programme *wc* und *head* angewendet. Im Falle *wc* war die Modifikation von sechs aus 140 Zeilen Code, um Variablen als Senken zu markieren, notwendig. Zusätzlich mussten Funktionen als Quellen aus der Standardbibliothek gewählt werden. Die im Rahmen der Arbeit entwickelte Implementierung stellt Funktionen zur Verfügung,

um Funktionen aus der Standardbibliothek zu markieren. Im Fall des Beispiels war die Markierung von drei Funktionen notwendig. Dies wird als eine Modifikation im üblichen Umfang betrachtet, da erwartet wird, dass auch bei größeren Programmen nur eine begrenzte Zahl an Bereichen als Quellen oder Senken markiert wird. Die Mehrzahl der Funktionen wird den Taint-Status der Eingaben zu den Ausgaben weiter propagieren, dies entspricht dem Typ, der ohne Markierung einer Funktion für die Funktion abgeleitet wird und erfordert daher keine Modifikation des Programms. Für *head* war lediglich die Markierung von zwei Funktionen, einer Quelle und einer Senke, notwendig.

Das Problem, dass der Programmcode modifiziert werden muss, tritt auch bei der Bereinigung von Werten auf. Die dynamische Taint-Analyse umgeht dieses Problem durch eine Änderung der Semantik. Auf diese Weise wird die Sanitization vor dem Entwickler verborgen, der hier vorgestellte Ansatz macht die Bereinigung aber explizit und die Bereinigung muss daher auch als solche erkannt werden. Dies hat zur Folge, dass weitere Modifikationen an dem Programm vorgenommen werden müssen. Die explizite Bereinigung von Werten kann sowohl als Vor- als auch als Nachteil interpretiert werden, da nicht explizit zwischen dem Entwickler eines Programms und der ausführenden Maschine kommuniziert wird. Das Programm kann dann durch die Verwendung der ursprünglichen Semantik unsicher werden, ohne dass dies von einem Entwickler bemerkt wird.

Es lässt sich also zusammenfassen, dass der vorgestellte Kalkül die Taint-Analyse explizit in den Programmcode überträgt, während der dynamische Ansatz die Taint-Analyse separat von dem Programmcode in der Semantik durchführt. Die Veränderung der Semantik hat aber zusätzlichen Rechenaufwand zur Laufzeit zur Folge. Dieser Aufwand wird durch das vorgestellte Kalkül vor der Laufzeit durchgeführt. Das vorgestellte Kalkül ist außerdem vollständig, das heißt existierende Sicherheitslücken werden immer erkannt. Der Grund dafür liegt darin, dass Taint nur durch Sanitization entfernt werden kann, alle anderen Konstruktionen propagieren den Taint immer auf die nächste Ebene. Es lässt sich auch festhalten, dass der Kalkül überapproximiert und daher nicht korrekt ist.

5.2 Ausblick

Abschnitt 5.1 hat bereits einige Schwächen angesprochen, die untersucht werden sollten. Beispielsweise stellt sich die Frage, wie gut der vorgestellte Ansatz in anderen Typsystemen arbeitet und sie ergänzt. Zusätzlich sollte auch untersucht werden, wie Quellen und Senken erkannt werden können. Dies würde der dynamischen und der statischen Taint-Analyse helfen die Automatisierung zu erhöhen. Die gleichen Ansätze könnten auch genutzt werden, um die Bereinigung von Werten statisch zu erkennen, was die hier vorgestellte Analyse weiter verbessern würde.

Es stellen sich aber weitere Fragen, die nur indirekt mit den hier vorgestellten Themen zusammenhängen. In dieser Arbeit wurde ein Typsystem entwickelt, das auf eine spezielle Analyse ausgelegt wird, und es stellt sich die Frage, können weitere Analysen auf ähnliche Weise durch einfache Erweiterungen eines Typsystems realisiert werden? Dies würde erlauben ein Programmiersystem zu entwickeln, welches einfach um unterschiedliche Analysen erweitert werden kann. Typen würden in einem solchen System eine Art Framework zur Analyse bilden. Daraus ergibt sich auch die Frage, wie diese unterschiedlichen Analysen dann miteinander interagieren und ob sie miteinander verträglich sind, also ihre Analyseergebnisse gegenseitig nicht beeinflussen. Wäre dies der Fall, könnte ein solches Programmiersystem sowohl die Effizienz der Programmierung durch Hilfestellungen als auch die Sicherheit der Software durch die Analysen erhöhen.

Glossar

ARIT ARIT ist eine einfache Sprache für arithmetische Ausdrücke, die um Primitive für Ein- und Ausgaben erweitert wurde. In den Beispielen fungieren die Ein- und Ausgaben als Quellen beziehungsweise Senken. 8, 10, 15, 61, 87

C C ist eine von Dennies Ritchie bei Bell Labs entwickelte Programmiersprache. 4–6, 26, 27, 33, 42, 46, 47, 59, 61–72, 80–82, 87, 105

CVE CVE ist eine Liste von veröffentlichten Sicherheitsschwachtellen. CVE wird durch die The MITRE Corparation verwaltet und durch das U.S. Department of Homeland Security finanziert. 1, 87

CWE CWE ist eine Liste von Soft- und Hardwareschwachstellen-Typen. Diese werden durch einen Baum repräsentiert und bilden so eine Gruppierung der Schwachstellen. CWE wird durch die The MITRE Corparation verwaltet und durch das U.S. Department of Homeland Security finanziert. 1, 2, 87

head *head* ist ein Kommandozeilen-Programm, das die ersten *n* Zeilen einer Datei zurückgibt. *n* ist hierbei ein Parameter, welches dem Programm auf der Kommandozeile übergeben wird. 78, 83, 84, 87

Integer-Overflow Ein Integer-Overflow liegt bei einer Berechnung vor, die den Wertebereich eines Integers überschreitet. Üblicherweise korrumpiert dies den Wert des Integers. Bei einer Addition springt der Wert zu dem niedrigsten Wert, der durch den Datentyp dargestellt werden kann. CWE-Eintrag: `https://cwe.mitre.org/data/definitions/190.html` 59, 63, 73, 78, 81, 82, 87

Quelle Eine Quelle ist der Ursprung der Verschmutzung einer Variable. 4, 5, 7, 8, 15, 34, 38, 41, 42, 47, 49, 58, 69, 70, 73–75, 78, 82–85, 87

Senke Eine Senke ist ein kritischer Bereich. Das Eindringen einer verschmutzten (vgl. Verschmutzung) Variable führt zu einem Abbruch des Programms. 5, 8, 15, 16, 34, 38, 41, 42, 47, 58, 59, 67, 69, 70, 73, 74, 77, 78, 82–85, 87

SQL Injection SQL Injections beschreiben die Verwendung von speziellen Elementen in Eingaben, die in SQL-Anfragen verwendet werden, um die Semantik der SQL-Anfrage zu manipulieren. 2, 7, 88

The MITRE Corparation The MITRE Corparation ist eine Forschungs- und Entwicklungsabteilung der U. S. Regierung. 87, 88

U.S. Department of Homeland Security U.S. Department of Homeland Security ist eine Abteilung der U. S. Regierung und ist für die Aufrechterhaltung der inneren Sicherheit verantwortlich. 87, 88

Unix Unix ist ein Betriebsystem, das in den 1970zigern von Ken Thompson, Dennis Ritchie und anderen bei Bell Labs entwickelt wurde. 6, 88

Verschmutzung Was als eine Verschmutzung interpretiert wird kann zwischen Programmen variieren. Im Allgemeinen wird eine Eingabe allerdings als verschmutzt (*eng.* tainted) angesehen, da sie potentiell von außen manipuliert wurde. 7, 43, 87, 88, 99

wc *wc* ist ein Kommandozeilen-Programm, das die Wörter, Zeilen und Bytes einer Datei zählt. i, ii, 4, 6, 59, 61, 62, 64, 66, 72, 73, 75, 77, 79, 81–83, 88, 101–104, 106

Literatur

[AC76] F. Allen und J. Cocke. „A program data flow analysis procedure“. In: *Communications of the ACM* 19.3 (1976), S. 137 (siehe S. 4).

[And07] Z. Anderson. „Static Analysis of C for Hybrid Type Checking“. Magisterarb. EECS Department, University of California, Berkeley, Jan. 2007. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-1.html (siehe S. 5).

[Arz+14] S. Arzt et al. „Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps“. In: *Acm Sigplan Notices* 49.6 (2014), S. 259–269 (siehe S. 4).

[Ban90] G. Bancerek. „The fundamental properties of natural numbers“. In: *Formalized Mathematics* 1.1 (1990), S. 41–46 (siehe S. 8).

[Bit20] BitBlaze. *BitBlaze: Binary Analysis for Computer Security*. 2020. URL: http://bitblaze.cs.berkeley.edu (besucht am 27. 03. 2020) (siehe S. 17).

[BV18] Bankenverband. *Online-Banking in Deutschland*. Juni 2018. URL: https://bankenverband.de/media/files/2018_06_19_Charts_OLB-final.pdf (siehe S. 1).

[Chu40] Alonzo Church. „A formulation of the simple theory of types“. In: *Journal of Symbolic Logic* 5.2 (Juni 1940), S. 56–68. DOI: 10.2307/2266170 (siehe S. 28).

[Coh86] D. Cohen. *Introduction to Computer Theory*. New York: Wiley, 1986. ISBN: 978-0-471-80271-6 (siehe S. 8).

[Con+07] J. Condit et al. „Dependent Types for Low-Level Programming“. In: *Proceedings of the 16th European Symposium on Programming*. ESOP'07. Braga, Portugal: Springer-Verlag, 2007, S. 520–535. ISBN: 9783540713142 (siehe S. 5).

[Cor20] The MITRE Corparation. *CVE - Home*. 2020. URL: https://cve.mitre.org/cve/ (besucht am 25. 03. 2020) (siehe S. 1).

[Cur34] H. Curry. „Functionality in Combinatory Logic“. In: *Proceedings of the National Academy of Sciences* 20.11 (Nov. 1934), S. 584–590. DOI: 10.1073/pnas.20.11.584 (siehe S. 28).

[FOW87] J Ferrante, K. Ottenstein und J. Warren. „The program dependence graph and its use in optimization“. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9.3 (Juli 1987), S. 319–349. DOI: 10.1145/24039.24041 (siehe S. 68).

[FSF19] Free Software Foundation. *GNU Cflow Manual*. 2019. URL: https://www.gnu.org/software/cflow/manual/html_node/index.html (siehe S. 101–104, 106).

[HF92] P. Hudak und J. Fasel. „A gentle introduction to Haskell“. In: *ACM Sigplan Notices* 27.5 (1992), S. 1–52 (siehe S. 18).

[HS86] J. Hindley und J. Seldin. *Introduction to Combinators and (lambda) Calculus*. Bd. 1. CUP Archive, 1986 (siehe S. 18).

[KR88] B. Kernighan und D. Ritchie. *The C Programming Language, 2nd Edition*. Prentice Hall, Apr. 1988. ISBN: 0131103628 (siehe S. 61, 65).

[Lia+18] K. Liakos et al. „Machine learning in agriculture: A review“. In: *Sensors* 18.8 (2018), S. 2674 (siehe S. 1).

[NG14] R. Nederpelt und H. Geuvers. *Type theory and formal proof: an introduction*. Cambridge University Press, 2014 (siehe S. 18).

[NS05] J. Newsome und D. Song. „Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software.“ In: *NDSS*. Bd. 5. Citeseer. 2005, S. 3–4 (siehe S. 17).

[Pad19] P. Padawitz. *Fixpoints, Categories, and (Co)Algebraic Modeling*. 26. Juli 2019. URL: https://fldit-www.cs.uni-dortmund.de/~peter/DialgSlides.pdf (besucht am 27. 07. 2019) (siehe S. 10).

[Pie02] B. Pierce. *Types and programming languages*. MIT press, 2002 (siehe S. 18).

[Plo81] G. Plotkin. „A structural approach to operational semantics“. In: (1981) (siehe S. 26).

[PØ95] J. Palsberg und P. Ørbæk. „Trust in the lambda-calculus“. In: *BRICS Report Series* 2.31 (Juni 1995). DOI: 10.7146/brics.v2i31.19934 (siehe S. 4, 5).

[Rob71] J. Robinson. „Computational Logic: The Unification Computation“. In: 1971 (siehe S. 45, 46).

[SAB10] E. Schwartz, T. Avgerinos und D. Brumley. „All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)“. In: *2010 IEEE symposium on Security and privacy*. IEEE. 2010, S. 317–331 (siehe S. 4, 6, 7, 15, 68, 69).

[SM03] A. Sabelfeld und A. Myers. „Language-based information-flow security“. In: *IEEE Journal on selected areas in communications* 21.1 (2003), S. 5–19 (siehe S. 5).

[Sør06] M. Sørensen. „Lectures on the Curry-Howard Isomorphism“. In: *Studies in Logic and the Foundations of Mathematics*. Bd. 149. Elsevier Science, Sep. 2006. ISBN: 0444520775. URL: https://www.elsevier.com/books/lectures-on-the-curry-howard-isomorphism/sorensen/978-0-444-52077-7 (siehe S. 18, 20, 21, 24, 28–30, 34, 39, 40, 42, 82).

[SV98] G. Smith und D. Volpano. „Secure Information Flow in a Multi-Threaded Imperative Language“. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '98. San Diego, California, USA: Association for Computing Machinery, 1998, S. 355–364. ISBN: 0897919793. DOI: 10.1145/268946.268975 (siehe S. 5).

[Tar+18] D. Tarditi et al. „Checked C: Making C Safe by Extension“. In: *IEEE Cybersecurity Development Conference 2018 (SecDev)*. IEEE, Sep. 2018, S. 53–60. URL: https://www.microsoft.com/en-us/research/publication/checkedc-making-c-safe-by-extension/ (siehe S. 5).

[Tri+09] O. Tripp et al. „TAJ: Effective Taint Analysis of Web Applications“. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2009, S. 87–97. ISBN: 9781605583921. DOI: 10.1145/1542476.1542486 (siehe S. 4, 7).

[TZ07] S. Tse und S. Zdancewic. „Run-time principals in information-flow type systems“. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30.1 (2007), 6–es (siehe S. 5).

[Wan+08] X. Wang et al. „STILL: Exploit Code Detection via Static Taint and Initialization Analyses“. In: *2008 Annual Computer Security Applications Conference (ACSAC)*. Dez. 2008, S. 289–298. DOI: 10.1109/ACSAC.2008.37 (siehe S. 4, 7).

Anhang A

Zusammenfassung aller Regeln

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \text{ (Ax)}$$

$$\frac{\Gamma, x : \sigma! \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma! \to \tau} \text{ (AbsI}\lambda\text{)}$$

$$\frac{\Gamma, x : \sigma? \vdash M : \tau?}{\Gamma \vdash \lambda x.M : \sigma? \to \tau?} \text{ (AbsS}\lambda\text{)}$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau} \text{ (App)}$$

$$\frac{\Gamma \vdash M : \sigma!}{\Gamma \vdash M : \sigma?} \text{ (weak)}$$

$$\frac{\Gamma, x : \sigma! \vdash M : \tau}{\Gamma \vdash \mathrm{I}x.M : \sigma! \to \tau} \text{ (AbsI)}$$

$$\frac{\Gamma, x : \sigma? \vdash M : \tau?}{\Gamma \vdash \mathrm{Z}x.M : \sigma? \to \tau?} \text{ (AbsZ)}$$

$$\frac{\Gamma, x : \sigma! \vdash M : \tau \qquad \Gamma, x : \sigma? \vdash N : \tau}{\Gamma \vdash \Delta x.MN : \sigma? \to \tau} \ \Delta\text{-Check-Point}$$

Anhang B

Theoreme

Theorem 5 (Generationenlemma)

- $\Gamma \vdash x : \sigma$ *impliziert* $x : \sigma \in \Gamma$
- $\Gamma \vdash MN : \tau$ *impliziert, dass ein* σ *existiert, sodass* $\Gamma \vdash M : \sigma \to \tau$ *und* $\Gamma \vdash N : \sigma$
- $\Gamma \vdash \lambda x.M : \rho$ *impliziert, dass ein* τ *existiert, sodass* $\Gamma, x : \sigma \vdash M : \tau$ *und* $\rho = \sigma \to \tau$

Theorem 6 (Generationenlemma für $\tilde{\lambda}$)

- $\Gamma \vdash x : \sigma$ *impliziert* $x : \sigma \in \Gamma$
- $\Gamma \vdash MN : \tau$ *impliziert, dass ein* σ *existiert, sodass* $\Gamma \vdash M : \sigma \to \tau$ *und* $\Gamma \vdash N : \sigma$
- $\Gamma \vdash \lambda x.M : \rho$ *impliziert, dass entweder ein* τ *existiert, sodass* $\Gamma, x : \sigma! \vdash M : \tau$ *und* $\rho = \sigma! \to \tau$ *oder ein* $\tau?$ *exisitert, sodass* $\Gamma, x : \sigma? \vdash M : \tau?$ *und* $\rho = \sigma? \to \tau?$
- $\Gamma \vdash \mathrm{Z}x.M : \rho$ *impliziert, dass ein* τ *existiert, sodass* $\Gamma, x : \sigma? \vdash M : \tau?$ *und* $\rho = \sigma? \to \tau?$
- $\Gamma \vdash \mathrm{I}x.M : \rho$ *impliziert, dass ein* τ *existiert, sodass* $\Gamma, x : \sigma! \vdash M : \tau$ *und* $\rho = \sigma! \to \tau$
- $\Gamma \vdash \Delta x.MN : \rho$ *impliziert, dass ein* τ *existiert, sodass* $\Gamma, x : \sigma! \vdash M : \tau$, $\Gamma, x : \sigma? \vdash N : \tau$ *und* $\rho = \sigma? \to \tau$

Theorem 7 (Substitutionslemma) *Falls* $\Gamma, x : \tau \vdash M : \sigma$ *und* $\Gamma \vdash N : \tau$ *gilt, kann* $\Gamma \vdash M\{x := N\} : \sigma$ *gefolgert werden.*

Anhang C

Church-kodierte natürliche Zahlen im $\tilde{\lambda}$-Kalkül

Beispielhaft wird $2 = \lambda s.\lambda z.s(sz)$ betrachtet. Die Form des Typens ist $(\alpha\_ \to \alpha\_)\_ \to \alpha\_ \to \alpha\_$. Durch welche Kombination von ?und! können die _ ersetzt werden? Hierzu werden alle Kombinationen aufgezählt und abgeleitet oder, wenn dies nicht möglich ist, wird begründet aus welchem Grund:

- $(\alpha! \to \alpha!)! \to \alpha! \to \alpha! = (\alpha! \to \alpha!) \to \alpha! \to \alpha!$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash s : \alpha! \to \alpha!} \quad \overline{\Gamma \vdash z : \alpha!}}{\Gamma \vdash sz : \alpha!} \quad \overline{\Gamma \vdash s : \alpha! \to \alpha!}}{\Gamma = s : \alpha! \to \alpha!, z : \alpha! \vdash s(sz) : \alpha!}}{s : \alpha! \to \alpha! \vdash \lambda z.s(sz) : \alpha! \to \alpha!}}{\vdash \lambda sz.s(sz) : (\alpha! \to \alpha!) \to \alpha! \to \alpha!}$$

- $(\alpha! \to \alpha!)! \to \alpha! \to \alpha? = (\alpha! \to \alpha!) \to \alpha! \to \alpha?$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash s : \alpha! \to \alpha!} \quad \overline{\Gamma \vdash z : \alpha!}}{\Gamma \vdash sz : \alpha!} \quad \overline{\Gamma \vdash s : \alpha! \to \alpha!}}{\Gamma \vdash s(sz) : \alpha!}}{\Gamma = s : \alpha! \to \alpha!, z : \alpha! \vdash s(sz) : \alpha?}}{s : \alpha! \to \alpha! \vdash \lambda z.s(sz) : \alpha! \to \alpha?}}{\vdash \lambda sz.s(sz) : (\alpha! \to \alpha!) \to \alpha! \to \alpha?}$$

- $(\alpha! \to \alpha!)! \to \alpha? \to \alpha! = (\alpha! \to \alpha!) \to \alpha? \to \alpha!$.
  s kann nicht auf z appliziert werden, da $\alpha! \neq \alpha?$.
- $(\alpha! \to \alpha!)! \to \alpha? \to \alpha! = (\alpha! \to \alpha!) \to \alpha? \to \alpha?$.
  s kann nicht auf z appliziert werden, da $\alpha! \neq \alpha?$.
- $(\alpha! \to \alpha!)? \to \alpha! \to \alpha! = (\alpha! \to \alpha?) \to \alpha! \to \alpha!$.
  s kann nicht auf sz appliziert werden, da $\alpha! \neq \alpha?$.

- $(\alpha! \to \alpha!)? \to \alpha! \to \alpha? = (\alpha! \to \alpha?) \to \alpha! \to \alpha?$.
  s kann nicht auf sz appliziert werden, da $\alpha! \neq \alpha?$.

- $(\alpha? \to \alpha!)! \to \alpha! \to \alpha! = (\alpha? \to \alpha!) \to \alpha! \to \alpha!$.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash s : \alpha? \to \alpha!} \quad \dfrac{\dfrac{}{\Gamma \vdash z : \alpha!}}{\Gamma \vdash z : \alpha?}}{\dfrac{\Gamma \vdash sz : \alpha!}{\Gamma \vdash sz : \alpha?}} \qquad \dfrac{}{\Gamma \vdash s : \alpha? \to \alpha!}}{\dfrac{\dfrac{\Gamma = s : \alpha? \to \alpha!, z : \alpha! \vdash s(sz) : \alpha!}{s : \alpha? \to \alpha! \vdash \lambda z.s(sz) : \alpha! \to \alpha!}}{\vdash \lambda sz.s(sz) : (\alpha? \to \alpha!) \to \alpha! \to \alpha!}}$$

- $(\alpha? \to \alpha!)! \to \alpha! \to \alpha? = (\alpha? \to \alpha!) \to \alpha! \to \alpha?$.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash s : \alpha? \to \alpha!} \quad \dfrac{\dfrac{}{\Gamma \vdash z : \alpha!}}{\Gamma \vdash z : \alpha?}}{\dfrac{\Gamma \vdash sz : \alpha!}{\Gamma \vdash sz : \alpha?}} \qquad \dfrac{}{\Gamma \vdash s : \alpha? \to \alpha!}}{\dfrac{\dfrac{\dfrac{\Gamma \vdash s(sz) : \alpha!}{\Gamma = s : \alpha? \to \alpha!, z : \alpha! \vdash s(sz) : \alpha?}}{s : \alpha? \to \alpha! \vdash \lambda z.s(sz) : \alpha! \to \alpha?}}{\vdash \lambda sz.s(sz) : (\alpha? \to \alpha!) \to \alpha! \to \alpha?}}$$

- $(\alpha? \to \alpha!)! \to \alpha? \to \alpha! = (\alpha? \to \alpha!) \to \alpha? \to \alpha!$.
  Keine Ableitungsregel kann angewendet werden.

$$\dfrac{\dfrac{}{s : \alpha? \to \alpha! \vdash \lambda z.s(sz) : \alpha? \to \alpha!}}{\vdash \lambda sz.s(sz) : (\alpha? \to \alpha!) \to \alpha? \to \alpha!}$$

- $(\alpha? \to \alpha!)! \to \alpha? \to \alpha? = (\alpha? \to \alpha!) \to \alpha? \to \alpha?$.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash s : \alpha? \to \alpha!} \quad \dfrac{}{\Gamma \vdash z : \alpha?}}{\dfrac{\Gamma \vdash sz : \alpha!}{\Gamma \vdash sz : \alpha?}} \qquad \dfrac{}{\Gamma \vdash s : \alpha? \to \alpha!}}{\dfrac{\dfrac{\dfrac{\Gamma \vdash s(sz) : \alpha!}{\Gamma = s : \alpha? \to \alpha!, z : \alpha? \vdash s(sz) : \alpha?}}{s : \alpha? \to \alpha! \vdash \lambda z.s(sz) : \alpha? \to \alpha?}}{\vdash \lambda sz.s(sz) : (\alpha? \to \alpha!) \to \alpha? \to \alpha?}}$$

- $(\alpha? \to \alpha!)? \to \alpha! \to \alpha! = (\alpha? \to \alpha?) \to \alpha! \to \alpha!$.
  Der Typ von $s(sz)$ wäre $\alpha?$, dies stimmt nicht mit $\alpha!$ überein.

- $(\alpha? \to \alpha!)? \to \alpha! \to \alpha? = (\alpha? \to \alpha?) \to \alpha! \to \alpha?$.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash s : \alpha? \to \alpha?} \quad \dfrac{\overline{\Gamma \vdash z : \alpha!}}{\Gamma \vdash z : \alpha?}}{\Gamma \vdash sz : \alpha?} \quad \dfrac{}{\Gamma \vdash s : \alpha? \to \alpha?}}{\dfrac{\dfrac{\Gamma = s : \alpha? \to \alpha?, z : \alpha! \vdash s(sz) : \alpha?}{s : \alpha? \to \alpha? \vdash \lambda z.s(sz) : \alpha! \to \alpha?}}{\vdash \lambda sz.s(sz) : (\alpha? \to \alpha?) \to \alpha! \to \alpha?}}$$

- $(\alpha? \to \alpha!)? \to \alpha? \to \alpha! = (\alpha? \to \alpha?) \to \alpha? \to \alpha!$.
  Der Typ von $s(sz)$ wäre $\alpha?$, dies stimmt nicht mit $\alpha!$ überein.

- $(\alpha? \to \alpha!)? \to \alpha? \to \alpha? = (\alpha? \to \alpha?) \to \alpha? \to \alpha?$.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash s : \alpha? \to \alpha?} \quad \dfrac{}{\Gamma \vdash z : \alpha?}}{\Gamma \vdash sz : \alpha?} \quad \dfrac{}{\Gamma \vdash s : \alpha? \to \alpha?}}{\dfrac{\dfrac{\Gamma = s : \alpha? \to \alpha?, z : \alpha? \vdash s(sz) : \alpha?}{s : \alpha? \to \alpha? \vdash \lambda z.s(sz) : \alpha? \to \alpha?}}{\vdash \lambda sz.s(sz) : (\alpha? \to \alpha?) \to \alpha? \to \alpha?}}$$

Eine Church-kodierte natürliche Zahl besitzt also mehr als einen Typen, gemeinsam haben sie allerdings die Struktur $(\alpha \to \alpha) \to \alpha \to \alpha$. Darauf aufbauend wird folgende Notation verwendet, $(\alpha\_ \to \alpha\_) \to \alpha\_ \to \alpha?$ entspricht einem $\mathbb{N}?$ und $(\alpha\_ \to \alpha\_) \to \alpha\_ \to \alpha!$ einem $\mathbb{N}!$. Es ist zu beachten, dass _ durch eine beliebige Kombination von ! und ? belegt werden kann, es werden aber nur gültige Typen für die natürlichen Zahlen gewählt. Das Rational dahinter ist wie folgt, um zu zeigen, dass $\mathbb{N}!$ gilt, muss ein Typen abgeleitet werden, der nicht zu einer Verschmutzung führt. Da ansonsten die $\tilde{\pi}$-Reduktion die Verschmutzung zur Ausgabe der Funktion propagieren würde. Muss der Typ $\mathbb{N}?$ gezeigt werden, muss einer der letzteren Typen gezeigt werden, aus dem gleichen Grund.

Anhang D

Programmcode von *wc*

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include <ctype.h>
typedef unsigned int count_t;  /* Counter type */

count_t ccount; /* Current file counters: chars, words, lines */
count_t wcount;
count_t lcount;
count_t total_ccount = 0; /* Totals counters: chars, words, lines */
count_t total_wcount = 0;
count_t total_lcount = 0;

/* Print error message and exit with error status. If PERR is not 0,
   display current errno status. */
static void error_print(int perr, char *fmt, va_list ap) {
  vfprintf (stderr, fmt, ap);
  if(perr){
    perror (" ");
  }else{
    fprintf (stderr, "\n");
  }
  exit(1);
}
```

Listing D.1.: Das Programm *wc* nach [FSF19] (an einigen Stellen adaptiert). - 1

```
1     static void errf(char *fmt, ...) {
2     va_list ap;
3
4     va_start(ap, fmt);
5     error_print(0, fmt, ap);
6     va_end(ap);
7   }
8
9   /* Print error message followed by errno status and exit
10     with error code. */
11  static void perrf(char *fmt, ...) {
12    va_list ap;
13
14    va_start(ap, fmt);
15    error_print(1, fmt, ap);
16    va_end(ap);
17  }
18
19  /* Output counters for given file */
20  void report(char *file, count_t ccount, count_t wcount, count_t lcount) {
21    printf("%6d %6d %6d %s\n", lcount, wcount, ccount, file);
22  }
23
24  /* Return true if C is a valid word constituent */
25  static int isword(unsigned char c){
26    return isalpha(c);
27  }
```

Listing D.2.: Das Programm *wc* nach [FSF19] (an einigen Stellen adaptiert). - 2

```
1     /* Get next word from the input stream. Return 0 on end
2      of file or error condition. Return 1 otherwise. */
3   int getword (FILE *fp) {
4     int c;
5     int set_result = 0;
6     int result = 0;
7
8     if(feof(fp)){
9       set_result = 1;
10    }
11
12    c = getc(fp);
13    while(c != EOF){
14    if(isword(c)){
15        wcount = wcount + 1;
16        break;
17    }
18    ccount = ccount + 1;
19    if((c) == '\n'){
20      lcount= lcount + 1;
21    }
22
23    c = getc(fp);
24    }
25
26    while(c != EOF){
27      ccount= ccount + 1;
28      if((c) == '\n'){
29    lcount = lcount + 1;
30      }
31      if(!isword(c)){
32    break;
33      }
34      c = getc(fp);
35    }
36    if(!set_result){
37      result = c != EOF;
38    }
39    return result;
40  }
```

Listing D.3.: Das Programm *wc* nach [FSF19] (an einigen Stellen adaptiert). - 3

```
1     /* Process file FILE. */
2   void counter(char *file) {
3     FILE *fp = fopen(file, "r");
4
5     if(!fp){
6       perrf ("cannot open file `%s'", file);
7     }
8     lcount = 0;
9     wcount = 0;
10    ccount = 0;
11    while(getword(fp)){
12      ;
13    }
14    fclose(fp);
15
16    report(file, ccount, wcount, lcount);
17    total_ccount = total_ccount + ccount;
18    total_wcount = total_wcount + wcount;
19    total_lcount = total_lcount + lcount;
20  }
21
22  int main(int argc, char **argv) {
23    if(argc < 2){
24      errf ("usage: wc FILE [FILE...]");
25    }
26
27    int i = 1;
28    while(i < argc) {
29      counter(argv[i]);
30      i = i + 1;
31    }
32
33    if(argc > 2){
34      report ("total", total_ccount, total_wcount, total_lcount);
35    }
36
37    return 0;
38  }
```

Listing D.4.: Das Programm *wc* nach [FSF19] (an einigen Stellen adaptiert). - 4

Listingverzeichnis

Eidesstattliche Versicherung
(Affidavit)

Stewing, Richard — 191301

Name, Vorname
(Last name, first name)

Matrikelnr.
(Enrollment number)

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem folgenden Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present ~~Bachelor's~~/Master's* thesis with the following title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution.

Titel der ~~Bachelor~~-/Masterarbeit*:
(Title of the ~~Bachelor's~~/ Master's* thesis):

Typebasierte Taint-Analyse im Lambda-Kalkül und die Anwendung auf C

*Nichtzutreffendes bitte streichen
(Please choose the appropriate)

Dorsten, 30.07.2020

Ort, Datum
(Place, date)

Unterschrift
(Signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to €50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, section 63, subsection 5 of the North Rhine-Westphalia Higher Education Act (*Hochschulgesetz*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:**

Dorsten, 30.07.2020

Ort, Datum
(Place, date)

Unterschrift
(Signature)

**Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.