TU WIEN Informatics

Language Properties for Smart Contracts

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Aldin Rizvanović, BSc
Matrikelnummer 00756024

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ass.Prof. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Monika di Angelo
Mitwirkung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gernot Salzer

Wien, 31. Dezember 2020

Aldin Rizvanović	Monika di Angelo

TU WIEN Informatics

Language Properties for Smart Contracts

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Aldin Rizvanović, BSc
Registration Number 00756024

to the Faculty of Informatics

at the TU Wien

Advisor: Ass.Prof. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Monika di Angelo
Assistance: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gernot Salzer

Vienna, 31st December, 2020

Aldin Rizvanović	Monika di Angelo

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Aldin Rizvanović, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 31. Dezember 2020

Aldin Rizvanović

Kurzfassung

Blockchain ist mittlerweile schon quer durch die Gesellschaft fast jedem ein Begriff. Die Blockchain-Technologie, sprich was die Blockchain genau ist, wie sie aufgebaut ist und verwendet werden kann, ist hingegen für viele noch ein Mysterium. Smart Contracts sind Computerprogramme, die auf der Blockchain-Technologie basieren und diese Computerprogramme werden mit Programmiersprachen entwickelt. Stellt man sich die Frage, welche Programmiersprache man am Besten verwendet, so merkt man relativ schnell, dass die Auswahl an verfügbaren Programmiersprachen für Smart Contracts enorm groß ist und kontinuierlich größer wird.

Diese Arbeit versucht die Frage zu klären, welche spezifischen Eigenschaften eine Sprache für die Entwicklung von Smart Contracts besitzen muss. Zuerst werden allgemeine Anforderungen an eine Sprache für die Entwicklung von Smart Contracts evaluiert. Im zweiten Schritt werden die identifizierten Anforderungen mit Spracheigenschaften in Beziehung gesetzt, damit geprüft werden kann, ob und inwieweit eine gewählte Programmiersprache für die Entwicklung von Smart Contracts geeignet ist.

Mit den Erkenntnissen dieser Arbeit wird am Beispiel der Programmiersprache Solidity, welche als die de facto Standardsprache für Smart Contracts gilt, analysiert, wie geeignet die Sprache für die Entwicklung von Smart Contracts tatsächlich ist.

Abstract

Blockchain has become a household name throughout society. The Blockchain technology, meaning what the Blockchain exactly is, how it is structured and how it can be used, is however still a mystery for many. Smart Contracts are computer programs based on the Blockchain technology and these computer programs are developed with programming languages. If one asks oneself the question, which programming language is best to use, one will notice relatively quickly that the selection of available programming languages for Smart Contracts is enormous and continuously growing.

This thesis attempts to answer the question which specific characteristics a language must possess for the development of Smart Contracts. At first, general requirements for a language for the development of Smart Contracts are evaluated. In a second step, the indentified requirements are put in relation to language properties in order to check if and to what extent a chosen programming language is suitable for the development of Smart Contracts.

The findings of this work are used to analyse how suitable the programming language Solidity, which is considered the de facto standard language for Smart Contracts, actually is for the development of Smart Contracts.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

1.1 Problemstellung

Smart Contracts sind Computerprogramme in einem verteilten System, die auf der Bockchain-Technologie basieren. Die darin definierten Regelungen werden dabei nur unter gewissen Voraussetzungen ausgeführt – autonom ohne eine außenstehende Autoritätsinstanz.

Auch wenn der Begriff bzw. die Idee schon seit den 90er-Jahren existiert [Sza97], sieht man erst seit kurzem, bedingt durch die Verwendung in Zusammenhang mit Kryptowährungen, einen Anstieg an Interesse und Medienpräsenz. Auch das Forschungsumfeld zu Smart Contracts ist noch relativ jung. Mittlerweile existieren dennoch etliche Blockchain-Plattformen und Smart Contract Sprachen. Die wohl bekannteste Plattform ist Ethereum[1]. Ethereum bietet mit seiner Ethereum Virtual Machine (EVM) als Abstraktionsschicht die Möglichkeit der Entwicklung von neuen High-Level-Sprachen, die in EVM Bytecode übersetzt und auf der Ethereum Blockchain ausgeführt werden. Solidity ist eine solche High-Level-Sprache, die direkt von der Ethereum Foundation[2] dezidiert für Smart Contracts entwickelt wurde.

In der Vergangenheit wurden durch Angriffe und deren Folgen immer mehr Schwachstellen von Smart Contracts bekannt. Daher wird verstärkt daran gearbeitet die Sicherheit bereits bei der Entwicklung von Smart Contracts zu erhöhen – formale Spezifikation und Verifikation von Smart Contracts wird als eine Lösung gesehen. Zudem, oder auch genau aus diesem Grund, werden auch andere Sprachen und Sprachparadigmen für die Entwicklung von Smart Contracts betrachtet. Bei der Vielzahl an neuen Sprachen, die mittlerweile existieren, stellt sich die Frage, welche Eigenschaften eine Sprache für die

[1] https://www.ethereum.org
[2] https://www.ethereum.org/foundation

Entwicklung von Smart Contracts besitzen sollte und ob die de facto Standardsprache für Smart Contracts, Solidity, wirklich geeignet ist.

Forschungsfrage und abgeleitete Unterfragen:

Welche spezifischen Eigenschaften muss eine Sprache für die Entwicklung von Smart Contracts besitzen?

- Was sind die Anforderung an eine Sprache für die Entwicklung von Smart Contracts?
- In welcher Beziehung stehen diese Anforderungen mit den Eigenschaften der Sprache?
- Welche dieser Spracheigenschaften besitzt Solidity und inwieweit ist Solidity geeignet für die Entwicklung von Smart Contracts?

1.2 Erwartetes Resultat

Programmiersprachen haben unterschiedliche Eigenschaften und erlauben zudem die Entwicklung von Programmen durch verschiedene Programmierparadigmen. Diese Arbeit soll einen Überblick schaffen über die wichtigsten Anforderungen an eine Sprache in Bezug auf die Entwicklung von Smart Contracts. Zudem wird untersucht, ob Solidity, eine Sprache, die dezidiert für Smart Contracts entwickelt wurde, tatsächlich auch die beste Wahl ist, wenn es um Smart Contracts geht.

1.3 Methodisches Vorgehen

Zur Beantwortung der definierten Forschungsfrage wird wie folgt vorgegangen:

- Es werden zunächst Klassifizierungen von wichtigen Spracheigenschaften von Programmiersprachen ermittelt. Hierzu wird die aktuelle Fachliteratur durchsucht.
- Danach werden anhand aktueller Forschungsarbeiten Anforderungen an Smart Contracts evaluiert und untersucht, welche Spracheigenschaften diesen Anforderungen zugeordnet werden können und inwiefern sie diese beeinflussen.
- Mit Hilfe der gewonnenen Erkenntnisse wird am Beispiel der Programmiersprache Solidity analysiert, wie geeignet die Sprache für Smart Contracts tatsächlich ist. Für die Analyse von Solidity wird die vorhandene Dokumentation in Version v0.7.3[3] herangezogen.

[3] https://solidity.readthedocs.io/en/v0.7.3/

1.4 Aufbau der Arbeit

Die weitere Arbeit ist wie folgt aufgebaut:

Kapitel 2 zeigt den aktuellen Stand der Forschung zu Smart Contracts und den eingesetzten Sprachen. Die relevante Literatur lässt sich dabei in die Bereiche, Analysen zu Schwachstellen und Fehlern, formale Spezifikation und unterschiedliche Sprachen und Sprachparadigmen unterteilen.

Kapitel 3 gibt einen Überblick über den allgemeinen Aufbau von Programmiersprachen. Einerseits soll Hintergrundwissen für die drauf folgenden Kapitel vermittelt werden, andererseits soll aufgezeigt werden, wie Programmiersprachen klassifiziert und somit unterschieden werden können.

In Kapitel 4 werden zunächst Anforderungen an Smart-Contract-Sprachen erarbeitet, diese mit Spracheigenschaften in Bezug gestellt und zuletzt Solidity anhand der gefunden Anforderungen und Spracheigenschaften validiert.

Abschließend wird in Kapitel 5 die Arbeit und die gewonnen Erkenntnisse zusammengefasst und mögliche zukünftige Themen und Fragestellungen in Ausblick gestellt.

KAPITEL 2

State-of-the-art

Die relevante Literatur lässt sich in drei Bereiche gliedern. Eine Vielzahl an wissenschaftlichen Arbeiten behandelt aktuelle Fehler und ausgenutzte Schwachstellen in existierenden Smart Contracts. Sie zeigen einerseits, wie es zu diesen Schwachstellen gekommen ist, wie man sie am besten beheben und vermeiden kann, und versuchen andererseits die bekannten Probleme zu klassifizieren. Ein weiterer großer Teil der aktuellen Forschung behandelt die formale Spezifikation eingesetzter Sprachen, um Programme im Vorfeld bereits verifizieren zu können. Hier werden nicht nur vorhandene Sprachen formal spezifiziert, sondern auch neue Sprachen entwickelt oder bereits formal spezifizierte Sprachen für den Einsatz bei Smart Contracts analysiert. Der dritte erkennbare Bereich der Forschung ist die Evaluierung von unterschiedlichen Sprachen und Sprachparadigmen für den Einsatz in Smart Contracts.

2.1 Analyse von Schwachstellen und Fehlern

Atzei et al. [ABC17] klassifizieren erstmals Ursachen für bekannte sicherheitsrelevante Schwachstellen, um, wie sie sagen, einerseits Entwicklende davor zu bewahren, bekannte Fehler erneut zu begehen und andererseits, um die Forschung in die richtige Richtung zu lenken, damit solche Fehler in Zukunft vermieden werden können. Sie gliedern ihre zwölf Klassen in drei unterschiedliche Level (Solidity, EVM Bytecode und Blockchain), je nachdem, wo das Problem seinen Ursprung hat. Als Ursachen für Schwachstellen, die Solidity zu Grunde liegen, werden folgende Klassifizierungen identifiziert:

- **Call to the unknown**: Gewisse Aufrufe in Solidity, wie zum Beispiel `call` oder `send`, können zu Seiteneffekten wie beispielsweise das unbeabsichtigte Aufrufen der `fallback` Funktion führen.
- **Gasless send**: Der Aufruf von `send` kann zu einer unerwarteten *Out-of-Gas Exception* führen, da der Aufruf im Hintergrund wie ein `call` Aufruf mit einer minimalen

Menge von Gas (2300 Einheiten) gehandhabt wird und je nach Umsetzung der `fallback` Funktion auf der Empfängerseite diese 2300 Gas reichen oder eben zur besagten Exception führen.

- **Exception disorders**: Exceptions werden in Solidity nicht einheitlich behandelt. In einer Kette von verschachtelten Aufrufen führt bei direkten Aufrufen eine Exception zum Rückgängigmachen der gesamten Transaktion, wohingegen sobald zumindest ein Aufruf dieser Kette ein `call` Aufruf ist (`delegatecall` oder `send` verhalten sich ähnlich), eine Exception nur Effekte bis zu diesem `call` Aufruf bereinigt, der `call` Aufruf `false` zurück liefert und der Programmfluss nach diesem Aufruf wieder fortgesetzt wird. Fehlende Überprüfung der Rückgabewerte solcher Aufrufe bieten Angriffsmöglichkeiten.

- **Type Casts**: Bei direkten Aufrufen von anderen Contracts wird die Adresse umgewandelt in das definierte Interface des aufgerufenen Contracts. Hierbei wird aber weder überprüft, ob sich hinter der Adresse tatsächlich der angegebene Contract befindet, noch wird überprüft, ob das definierte Interface dem aufgerufenen Contract entspricht. Falsche Annahmen bei der Typumwandlung können als Schwachstelle ausgenutzt werden.

- **Reentrancy**: Die Annahme, dass eine nicht-rekursive Methode vor ihrer Beendigung nicht noch einmal betreten werden kann, ist in Solidity durch die `fallback` Methode nicht korrekt und ermöglicht Angriffe wie den DAO Hack [Dai16].

- **Keeping secrets**: Felder in Contracts können `public`oder `private` deklariert werden. Während `public` Felder von allen einsehbar sind, können `private` Felder nicht direkt eingesehen werden. Da Transaktionen auf der offen einsehbaren Ethereum-Blockchain veröffentlicht werden müssen, ist es möglich Variablen trotz `private` Deklaration herzuleiten.

Ursachen in Bezug auf die EVM:s

- **Immutable bugs**: Programme mit Fehlern sind nicht korrigierbar und es müssen aufwendige Workarounds durchgeführt werden, um diese Fehler zu entfernen.

- **Ether lost in transfer**: Ether, der an nicht (mehr) verwendete Adressen gesendet wird, ist für immer verloren und da es keine systemunterstützte Möglichkeit gibt solche Adressen zu erkennen oder den fälschlich verschickten Ether zurück zu bekommen, müssen Entwickelnde manuell prüfen und sicherstellen, dass die korrekte Adresse für einen Transfer verwendet wird.

- **Stack size limit**: Der Call Stack hat ein Limit von 1024 Frames. Bis zum Hard-Fork im Oktober 2016 [Swe16][But16] konnte dieses Limit noch erreicht und somit ausgenutzt werden. Aktuell wird über das Gas-Limit pro Block sichergestellt, dass das Stack Limit nie erreicht werden kann.

Ursachen in Bezug auf die Blockchain:

- **Unpredicable state**: Die Reihenfolge, in welcher Transaktionen von Minern aufgenommen werden, ist nicht definiert und kann dazu führen, dass sich der Smart Contract in einem anderen Zustand befindet, als man eigentlich beim Senden einer Transaktion angenommen hat. Forks, bei denen letztendlich die kürzere Kette verworfen wird, sind auch ein Grund für den nicht vorhersagbaren Zustand eines Smart Contracts in Bezug auf eine einzelne Transaktion.
- **Generating randomness**: Durch die deterministische Eigenschaft der EVM Bytecode Ausführung ist die Erzeugung von Pseudo-Zufälligkeit für gewisse Anwendungsfälle nötig, bildet aber zugleich auch einen weiteren Angriffsvektor. Initiale Seed Werte können entweder von Angreifern ausgelesen oder durch Miner ausgenutzt werden.
- **Time constraints**: Zeitlich beschränkte oder bedingte Programmteile werden meist durch den Block Timestamp definiert. Da dieser Timestamp in einem gewissen Maß von den Minern beeinflusst werden kann, sind auch in diesem Fall entsprechende Angriffe möglich.

Für alle klassifizierten Ursachen werden von Atzei et al. auch konkrete Attacken, die diese Schwachstellen ausnutzen, vorgestellt.

Luu et al. [LCO+16] untersuchen ebenfalls eine große Anzahl an existierender Smart Contracts und klassifizieren die gefundenen sicherheitsrelevanten Schwachstellen wie folgt:

- **Transaction Ordering Dependence**
- **Timestamp Dependence**
- **Mishandled Exceptions**
- **Reentrance Vulnerability**

Sie formalisieren zunächst eine, wie sie es nennen, 'lightweight' operationelle Semantik für Ethereum und zeigen dann, wie mit Hilfe von kleineren Verbesserungen diese Semantik die zuvor gefundenen Schwachstellen erkennen und somit verhindern könnte. Folgende Verbesserungsvorschläge werden präsentiert:

- **Guardian Transactions**: Neue Semantikregeln für die Ausführung von Transaktionen verhindern durch Definition von Bedingungen, die der aktuelle Zustand haben muss, das Problem der *Transaction Ordering Dependence*. Mit diesen Regeln sind die benötigten Eigenschaften des Zustandes definiert und Transaktionen werden nur unter den entsprechenden Bedingungen ausgeführt.

- **Deterministic Timestamp**: Statt dem leicht manipulierbaren Block Timestamp wird empfohlen den Block Index als globale Zeiteinheit zu verwenden. Dieser erhöht sich mit jedem Block, welcher etwa alle 12 Sekunden stattfindet, um eins und könnte mit wenig Aufwand in die aktuelle Implementierung eingebaut werden.
- **Better Exception Handling**: Automatisches Durchreichen der Exceptions auf dem EVM-Level wird als Verbesserungsvorschlag beschrieben. Zudem würden explizite `throw` und `catch` Instruktionen, wie sie in vielen anderen Sprachen zu finden sind, in der EVM ebenfalls hilfreich sein.

Da die von Luu et al. vorgeschlagenen Verbesserungen einen Hard Fork, also ein Update aller Clients, benötigen würden, haben sie in ihrem Paper ein selbst entwickeltes Tool *Oyente*[1] vorgestellt, welches die Verbesserungen als unabhängiges Plugin anbietet. Dieses in Python geschriebene Tool basiert auf symbolischer Ausführung und liefert mittels statischer Analyse der Programmpfade, anhand des EVM Bytecodes des Programmes und dem aktuellen globalen Ethereum Zustand als zweitem Inputparameter, den Anwendenden die problematischen Pfade im Programm als Ergebnis. Auf die konkrete Implementierung, den Aufbau und die letztendlich Evaluierung des Tools wird hier nicht näher eingegangen, kann aber im entsprechenden Paper ebenfalls gefunden werden.

2.2 Formale Spezifikation

Bhargavan et al. [BSZB+16] entwickeln fast zur selben Zeit wie Luu et at. ebenfalls ein Tool zur Analyse und Verifikation von Laufzeitsicherheit und funktionaler Korrektheit von Smart Contracts. Sie gehen hier aber einen anderen Weg, indem sie die zu analysierenden Programme in die Sprache F* [2] übersetzen, welche für Programmverifikation ausgelegt ist. Solidity Code kann von dem selbst in OCaml geschriebenen Übersetzer *Solidity** in F* Code überführt werden und EVM Bytecode kann durch den Übersetzer namens *EVM** in F* Code übersetzt werden. Um gewisse Eigenschaften von den analysierten Smart Contracts statisch verifizieren zu können, gibt es die Möglichkeit aus dem F* Code automatisiert Anfragen für Satisfiability Modulo Theory (SMT) Solver zu generieren.

Die Ansätze von Bhargavan et al. befinden sich aber noch im Anfangsstadium. Einerseits wird nur ein kleines Subset von Solidity betrachtet, eine Nutzung für jeglichen Solidity-Code ist nicht möglich. Andererseits sind die Ergebnisse noch eher als richtungsweisend zu verstehen und dafür geeignet mögliche sicherheitsrelevante Schwachstellen zu finden. Eine manuelle Spezifikation des Smart Contracts in F* und ein manueller Beweis können noch nicht ersetzt werden.

Grishchenko et al. [GMS18] stellen als erste eine vollständige strukturierte operationelle Semantik (auch Small-Step Semantik genannt) für EVM Bytecode vor, welche auch

[1] https://github.com/melonproject/oyente
[2] https://www.fstar-lang.org/

schon teilweise in F* formalisiert wurde . Zudem definieren sie semantisch folgende sicherheitsrelevante Eigenschaften für einen Teil der klassifizierten Fehler aus den Arbeiten von Luu et al. [LCO+16] und Atzei et al. [ABC17]:

- **Call Integrity**: Das zugrunde liegende Problem bei dieser Art von Fehlern liegt in der Abhängigkeit von nicht vertrauenswürdigem Code. Dabei werden drei Arten unterschieden, wie externe Contracts den eigenen Contract beeinflussen können. Der eigene Contract kann den Code des externen Contracts übernehmen (*Code Dependency*), von der Ausführung oder dem Rückgabewert des externen Contracts abhängen (*Effect Dependency*) oder der externe Contract manipuliert den globalen Zustand direkt bzw. indirekt durch erneuten Aufruf des eigenen Contracts (*Re-entrancy*). Aus diesen drei Arten leiten Grishschenko et al. dann die drei Bedingungen *Effect Independence*, *Code Independence* und *Single-entrancy* für *Call Integrity* ab.
- **Atomicity**: Hier wird das Problem betrachtet, das durch fehlendes oder fehlerhaftes Exception Handling in Verbindung mit der vorhandenen Gas Menge auftreten kann. Die Veränderung des globalen Zustandes durch die Ausführung von Contract Code sollte nicht von der Menge des verfügbaren Gas abhängen. Bei einer *Out-of-Gas Exception* sollte der globale Zustand durch die Ausführung nicht verändert werden.
- **Independence of Miner controlled Parameters**: *Independence of the Transaction Environment* definiert die Unabhängigkeit der von Minern direkt manipulierbaren Parametern wie den Block Timestamp. *Independence of Mutable Account State* definiert die Unabhängigkeit des von Minern durch entsprechende Anordnung von Transaktionen manipulierbaren Account State [3].

Tabelle 2.1 zeigt, welche Arten von klassifizierten Fehlern die von Grishchenko et al. definierten Sicherheitseigenschaften identifizieren und somit auch vermeiden können.

Sighn et al. [SPZ+20] haben mit einer systematischen Auswertung von über 6000 wissenschaftlichen Arbeiten, 35 Arbeiten selektiert, die die aktuellen Forschungsansätze und Probleme der formalen Spezifikation von Smart Contracts darstellen.

Folgende formale Methoden zur Verbesserung von Smart Contracts werden aktuell erforscht:

- Theorem Proving (31%)
- Model Checking (17%)
- Symbolic Execution (14%)

[3] Ein Ethereum Account State besteht aus *Nonce*, *Balance*, *CodeHash* und *StorageRoot*, dem Hash zum Root des zugehörigen globalen Speicher des Contracts [Woo14].

- Formal Modelling (14%)
- Specification language (9%)
- Finite State Machine (6%)
- Logic based approach (3%)
- Behaviorial Modelling (3%)
- Formal Reasoning (3%)

Diese formalen Methoden werden eingesetzt, um folgende Schwachstellen und Probleme von Smart Contracts zu beheben:

- Contract Functionality Verification (65%)
- Security (17%)
- Privacy (6%)
- Scalability (3%)
- Bug bounty (3%)
- Trustworthy Data Feeding (3%)
- Blockchain concensus protokol correctness (3%)

Sicherheitsrelevante Eigenschaft	**Bug**
Call Integrity	Reeentrancy [LCO+16] [ABC17] Call to the Unknown [ABC17]
Atomicity	Mishandeled Exception [LCO+16] [ABC17]
Independence of Mutable Account State	Transaction Order Dependency [LCO+16] Unpredictable State [ABC17]
Independence of Execution Environment	Timestamp Dependency [LCO+16] Time Constraints [ABC17] Generating Randomness [ABC17]

Tabelle 2.1: Zuordnung von Fehlern aus [ABC17] und [LCO+16] nach Grishchenko et al. [GMS18]

2.3 Unterschiedliche Sprachen und Sprachparadigmen

O'Conner [O'C17] stellt eine neue Low-level Sprache namens *Simplicity* vor, die analog zu Bitcoin Script oder der EVM zum Ausführen von Smart Contracts genutzt werden soll. Die typisierte, auf nur neun Kombinatoren basierende, funktionale Sprache ohne Schleifen oder Rekursion soll die bisherigen Probleme von Bitcoin Script, die fehlende Ausdruckskraft der Sprache und die bisherigen Probleme der EVM, die fehlende formale Spezifikation der Programmsemantik sowie der fehlenden Möglichkeit genaue Aussagen über die Laufzeiten und -kosten zu berechnen, lösen.

Simplicity basiert auf dem Sequentialkalkül, verzichtet damit bewusst im Vergleich zum Lambda-Kalkül auf Funktionen, und besteht aus nur drei Typen und neun Kombinatoren. Die denotationelle Semantik ist ebenfalls sehr kompakt rekursiv über diese neun Kombinatoren definiert. Die Sprache sowie die Semantik wurden in Coq[4], einem bekannten Beweisassistenten, definiert und erlauben somit eine formale Verifikation von Simplicity-Programmen in diesem Beweissystem. Neben der denotationellen Semantik wird auch die operationelle Semantik mit Hilfe einer eigenen abstrakten Maschine, der *Bit Machine*, definiert. Die *Bit Machine* basiert auf zwei Stacks, *read-only Frame* und *write-only Frame*, und entsprechenden Instruktionen, um die einzelnen Zellen im Stack zu manipulieren. Die operationelle Semantik von Simplicity ist definiert als rekursive Abbildung eines Ausdruckes in eine Sequenz von Instruktionen auf der *Bit Machine*. Dies ermöglicht statische Analysen zu benötigten Programmressourcen, wie die Anzahl an Instruktionen und die benötigte und maximale Anzahl an Zellen und Datenframes zur Programmausführung.

Ein sehr interessanter Aspekt in Bezug auf die Ausführung in der *Bit Machine* sind *Jets*. Simplicity-Programme werden intern als gerichteter azyklischer Graph (engl. Directed Acyclic Graph, DAG) abgebildet und die *Bit Machine* kann in diesem Graph Ausdrücke mit gleichen Unterausdrücken erkennen. Als *Jets* werden nun Codeabschnitte, z.B. in C geschrieben, bezeichnet, die formal beweisbar genau das umsetzen, was im Simplicity-Ausdruck definiert ist, und somit diesen ersetzen und direkt die Datenframes manipulieren können. Das erweitert die recht simple Sprache, die eigentlich zur reinen Berechnung gedacht ist, um alle nötigen Elemente, sodass sie auch in der Praxis einsetzbar wird. Für den Einsatz der Sprache auf der Blockchain werden weitere Kombinatoren beschrieben: Kombinatoren zur Berechnung und Validierung von Signaturen, zur Bildung von Merkle Root Hashes von Simplicity Ausdrücken, um *Pay To Script Hash* aus Bitcoin umsetzen zu können oder Kombinatoren für *Witness Values*, die als Parameter für Simplicity-Ausdrücke gesehen werden können, jedoch erst zur Ausführung bekannt geben werden müssen.

Ob Simplicity weitläufig zum Einsatz kommt, wird sich noch zeigen müssen. Zuerst müssen die zusätzlichen Kombinatoren zum Simplicity Core ebenfalls mit Coq bewiesen und entsprechende High-level Sprachen dafür entwickelt werden.

[4] https://coq.inria.fr/

Coblenz [Cob17] entwickelt aktuell eine neue Blockchain und Sprache für Smart Contract namens *Obsidian*[5]. Im Vergleich zu aktuell eingesetzten Sprachen, soll Obsidian eine sicherere Sprache werden, die einerseits bekannte Schwachstellen von vornherein nicht ermöglicht und andererseits Programmierende dabei unterstützt sicheren und korrekten Code zu schreiben, da Fehler per Sprachdesign vermieden werden. Wie genau das umgesetzt wird und ob es tatsächlich auch funktioniert, muss sich noch zeigen, denn aktuell wird am Obsidian-Prototyp noch gearbeitet. Die Sprache soll einerseits ein lineares Typsystem besitzen, um damit die Ressource Geld sowie Transaktionen von einem Compiler überprüfen lassen zu können, und andererseits soll die objektorientierte Sprache Zustände als First-Class-Objekte abbilden, um somit Zugriffe auf Objekte nur bei entsprechend validen Zuständen erlauben zu können. Als Ziel hat sich Coblenz gesetzt, die Sicherheit von Obsidian durch formale Beweise zu belegen und zugleich durch Umfragen und Studien zu zeigen, dass Entwickelnde mit Obsidian leichter korrekte und sichere Programme erstellen können, als mit anderen Sprachen für Smart Contract und Blockchain.

Mavridou und Laszka [ML17a] verfolgen wie Coblenz den zustandsbasierten Ansatz. Sie präsentieren *FSolidM*[6], ein Framework um Solidity Smart Contracts mittels endlicher Automaten (Finite State Machines, FSM) zu entwickeln. Ein Smart Contract wird als endlicher Automat abgebildet und ist definiert als Tupel $\langle S, s_0, C, I, O, \rightarrow \rangle$, bestehend aus Zuständen, einem Anfangszustand, Contract Variablen, Input Variablen, Output Variablen und einer Transition Relation für Zustandsübergänge.

Neben ihrem formalen Modell und dessen Implementierung stellen sie einen grafischen Editor zur Verfügung, mit dem die endlichen Automaten grafisch erstellt werden können, sowie einen eigenen Übersetzer, der ihre endlichen Automaten in Solidity Code umwandeln kann. Eine Sammlung an vorgefertigten Plugins für die endlichen Automaten, die gängige Abläufe und Sicherheitsfeatures leicht zu bestehender Funktionalität hinzufügen lassen, sind ebenfalls Teil des vorgestellten Frameworks.

Die vorgestellten Plugins *Locking* und *Transition Counter* verhindern durch Sperren der Methode die Reentrancy Schwachstelle und durch eindeutige Nummerierung der Zustandsübergänge das Problem des unbestimmbaren Zustandes durch unterschiedliche Transaktionsreihenfolgen. Die Plugins *Automatic Time Transitions* und *Access Control* ermöglichen automatische Überprüfung von zeitlichen Bedingungen und Ausführung von entsprechenden Zustandswechseln vor jedem anderen manuellen Zustandswechsel sowie das Verwalten von Zugriffsgruppen und Zugriffsberechtigungen bei Funktionen. Weitere Plugins zur Vermeidung von bekannten Schwachstellen sollen noch folgen, genauso wie eine Erweiterung des Frameworks um formale Verifikationsmöglichkeiten.

[5] https://mcoblenz.github.io/Obsidian/
[6] https://cps-vo.org/group/smartcontracts

Idelberger et al. [IGRS16] analysieren, im Vergleich zu den weit verbreiteten imperativen Ansätzen den Einsatz von deklarativen logikorientierten Sprachen. Sie zeigen, wie eine Menge an Regeln und ein zugehöriges Regelwerk, welches als Zustandsautomat gesehen werden kann, *Logic-based Smart Contracts* abbilden und wie diese im Vergleich zu normalen prozeduralen Smart Contracts auf der Blockchain eingesetzt werden können. Hierbei vergleichen sie sowohl den imperativen als auch den deklarativen Ansatz mit juristischen Verträgen - konkret mit der technischen Unterstützung oder sogar technischen Ablöse von juristischen Verträgen. Es werden folgende vier Phasen im Lebenszyklus eines Vertrages detailliert betrachtet:

- **Formation and Negotiation.** Logische Ausdrücke können als High-Level-Spezifikation verstanden werden, welche als Regeln definiert, direkt von *Logic-based Smart Contracts* bei der Ausführung verwendet werden können. Zudem können Logische Ausdrücke mittels auf Logik basierende Techniken, wie der formalen Verifikation, sehr leicht überprüft werden. Bei imperativen Ansätzen wird meist zuerst das Verhalten in natürlicher Sprache formuliert und dann in einen Smart Contract übersetzt. Da das Formulieren und Überprüfen von imperativen Code sehr mühsam werden kann, sind deklarative Ansätze laut Idelberger et al. in dieser Phase die bessere Wahl.
- **Contract Storage/Notarizing.** Die Aufbewahrung und notarielle Beurkundung von Verträgen kann mit beiden Ansätzen ohne viel Aufwand durch das Speichern auf einer Blockchain abgebildet werden. Da im Allgemeinen logische Ausdrücke im Vergleich zu imperativen Programmcode kompakter darstellbar sind, ist auch in diesem Punkt, bezogen auf die benötigte Speichermenge, laut Idelberger et al. der deklarative Ansatz die bessere Wahl.
- **Enforcement and Monitoring.** Die Vollstreckung und die Überprüfung der Einhaltung des Vertrages, kann als deployen und ausführen des Smart Contracts gesehen werden. Eine effiziente Ausführung im deklarativen Ansatz ist stark abhängig von dem zugrunde liegenden Inferenz-System und diese Ausführungskomplexität ist nicht immer so leicht zu kontrollieren, wie bei einem imperativen Programmstück. Dennoch wird von Idelberger et al. für diese Phase argumentiert, dass in Betrachtung der Gesamtheit der Ausführung, vor allem bei der Interaktion mit anderen Smart Contracts, der deklarative Ansatz die bessere Wahl ist, da hier unterschiedliche logische Regeln mehrerer Smart Contracts leichter vereint werden können, als unterschiedliche imperative Programmstücke.
- **Modification.** Zwar sind Smart Contracts auf einer Blockchain unveränderlich, doch die darin gespeicherten und verwendeten Daten können modifiziert werden. Im imperativen Ansatz kann ein Haupt-Contract Adressen zu weiteren Smart Contracts verwalten, die die eigentlichen Bedingungen und Funktionalitäten beinhalten. Analog dazu kann im deklarativen Ansatz die Wissensbasis und die zugehörigen logischen Regeln als `public`-Variablen abgebildet werden. Der Vorteil beim deklarativen Ansatz bei der Modifikation des Smart Contracts ist, dass die Reihenfolge

der Regeln keine Rolle spielt und diese somit ohne viel Aufwand Regeln gelöscht, ergänzt oder bearbeitet werden können. Beim imperativen Ansatz muss beim Modifizieren auf die Ausführungsreihenfolge geachtet werden.

- **Dispute Resolution.** Streitschlichtungen können grob in zwei Arten unterteilt werden:
  - *Rechtssprechende Beschlüsse*, bei denen eine rechtsprechende Person oder eine Jury ein Ergebnis bestimmt.
  - *Einvernehmliche Beschlüsse*, bei der die betroffenen Parteien, zum Beispiel durch Mediation oder Verhandlungen, eine Einigung finden.

  Beide dieser Hauptarten der Streitschlichtung sind sowohl mit imperativen als auch deklarativen Ansätzen umsetzbar. Idelberger et al. argumentieren aber, dass bedingt durch die grundlegende Idee hinter deklarativen Regeln - Vertragsklauseln widerspiegeln - es Juristen leichter fallen sollte, aus formalen Aussagen konstruierte Aussagen zu verstehen, bewerten und mit anderen zu vergleichen.

Beim Einsatz auf der Blockchain untersuchen sie *On-Chain-*, *Off-Chain-* und *On-Off-Chain-*Ansätze bezüglich wo die Regeln bzw. das Regelwerk positioniert werden.

Beim *Off-Chain-*Ansatz befindet sich das Inferenzsystem außerhalb der Blockchain, wodurch weniger Ressourcen der Blockchain benötigt werden, aber auch Vertrauen in externe Systeme verlangt. Beim *On-Chain-*Ansatz wird das Inferenzsystem auf der Blockchain abgebildet, was wiederum prüfbare und vertrauenswürdige Inferenzen bedeutet. Hier können aber, je nach Rechenkomplexität des gewählten Inferenzsystem-Algorithmus, schnell hohe Kosten durch hohe Ressourcenanforderungen an die Blockchain entstehen.

Der *On-Off-Chain-*Ansatz versucht die Vorteile der beiden anderen Ansätze zu nutzen, indem das Inferenzsystem in eine sozusagen abgespeckte Variante übersetzt wird und in dieser Form auf der Blockchain abgebildet wird. Diese simplere Darstellung soll kosteneffizienter als das allumfassende Inferenzsystem auf der Blockchain verwendet werden können. Die Limitierung bei diesem Ansatz liegt in der Wahl des Inferenz-Algorithmus, da dieser seine aktuelle Logik in vereinfachter Darstellung (zum Beispiel als Menge von Gleichungen) abbilden müssen kann.

Nach Idelberger et al. kann man den deklarativen Ansatz als Ergänzung sehen, der mit seinen logischen Regeln als höhere Spezifikationssprache auch formale Verifikation bestimmter Eigenschaften ermöglichen kann.

KAPITEL 3

Programmiersprachen

Dieses Kapitel basiert hauptsächlich auf dem Buch von Ghezzi und Jazayeri [GJ97] und soll einen Überblick über den allgemeinen Aufbau von Programmiersprachen geben. Dieses Hintergrundwissen wird in den weiteren Kapitel benötigt, um verstehen zu können, nach welchen Klassifizierungen Programmiersprachen unterschieden werden kann. Es sei hingewiesen, dass hier nur Themen behandelt werden, die für die weiteren Kapitel relevant sind und daher dieses Kapitel keine vollständige Aufzählung aller klassifizierbarer Unterschiede von Programmiersprachen darstellt.

Von der Struktur her orientiert sich dieses Kapitel an der Struktur des genannten Buches und ist gegliedert in Syntax und Semantik, Strukturierung der Daten, Strukturierung der Berechnung, Strukturierung des Programms und Überblick unterschiedlicher Sprachparadigmen. Anfänger

3.1 Syntax und Semantik

Jede Sprache besteht aus Syntax und Semantik. Die Syntax definiert einerseits mit lexikalischen Regeln aus welchen Zeichen Wörter bestehen können und mit syntaktischen Regeln, wie diese Wörter zu größeren Konstrukten kombiniert werden können. Die Semantik hingegen definiert die Bedeutung dieser aus Wörter gebildeten Konstrukte. Programme können syntaktisch korrekt sein ohne dabei semantisch korrekt zu sein und umgekehrt. Die Syntax kann also als Grammatik der Sprache gesehen werden, die durch ein Regelwerk klar definiert wird. Die Semantik einer Sprache kann entweder informal oder formal definiert werden. Operationelle Semantik wird zum Beispiel als informale Beschreibung gesehen, bei der die Programmausführung auf einer abstrakten Maschine informal beschrieben wird. Axiomatische Semantik oder denotationelle Semantik gehört zu den formalen Beschreibungen der Semantik, bei welcher das Programm als Zustandsautomat gesehen wird und Ausdrücke eindeutige Zustandswechsel auslösen und damit auch die Eigenschaften und Werte der definierten Programmvariablen verändern.

Konzept von Binding

Programme bestehen aus unterschiedlichen Bestandteilen, wie zum Beispiel Variablen oder Funktionen und diese Bestandteile haben gewisse Eigenschaften. Der Name, der Typ, eine gewisse Speicheradresse, der aktuelle Wert, all das kann als Eigenschaft einer Variable gesehen werden. Diese Eigenschaften werden zu bestimmten Zeiten, an bestimmten Positionen im Programm und auf bestimmte Art und Weise von einer Programmiersprache an eine Variable gebunden. Die unterschiedlichen Herangehensweisen beim Binden der Eigenschaften an die Bestandteile des Programms, stellen ein essentielles Konzept bei der Definition der Semantik einer Sprache dar und erzeugen somit auch signifikante Unterscheidungsmerkmale bei Programmiersprachen.

Variablen

Variablen sind eine Abstraktion eines Speicherbereiches in einem Computer. Formal ist eine Variable als ein 5-Tupel $\langle$ *Name, Scope, Type, L-Value, R-Value* $\rangle$ definiert.

- *Name*: Der Name einer Variable wird auf eine definierte Art und Weise in der Deklaration eingeführt. Programminstruktionen können die Variable durch ihren Namen manipulieren.
- *Scope*: Der Scope einer Variable wird meist bei der Deklaration der Variable definiert und bestimmt an welchen bzw. bis zu welchen Stellen im Programm die Variable bekannt ist.
- *Type*: Der Typ einer Variable kann als Regelwerk gesehen werden, welches definiert, welche Werte eine Variable annehmen kann und welche zugehörigen Operationen auf diese Variable angewendet werden können.
- *L-Value*: Der L-Value einer Variable entspricht dem Speicherbereich der Variable im Speicher.
- *R-Value*: Der R-Value einer Variable entspricht dem Wert der Variable kodiert nach dem entsprechenden Typ der Variable.

Routinen

Eine Routine ist eine allgemeine Abstraktion eines zu einer Einheit zusammengefassten Bereiches eines Programms. Routinen werden in unterschiedlichen Programmiersprachen auch Unterprogramme, Prozeduren oder Funktionen genannt. Routinen werden formal auch als ein 5-Tupel $\langle$ *Name, Scope, Type, L-Value, R-Value* $\rangle$ definiert. Der Name einer Routine wird, wie bei einer Variable, durch eine Deklaration eingeführt und der Scope definiert, wo dieser Name eine Bedeutung hat. Routinen werden durch einen Aufruf, der sich im Scope der Routine befinden muss, aktiviert, bei welchem der Name der Routine aufgerufen wird und die definierten Parameter übergeben werden. Name, Parameter und Rückgabewert der Routine werden im Header der Routine definiert, welcher den

Type der Routine bestimmt. Der L-Value einer Routine enthält eine Referenz auf den Speicherbereich, welcher den Body - die ausführbaren Elemente der Routine - beinhaltet. Der Wert, der nach Aktivierung durch Abarbeitung der Routine resultiert, wird an den R-Value der Routine gebunden.

Parameter

Routinen besitzen Eingangs- sowie Ausgangsparameter, wobei für die Ausgangsparameter der Begriff Rückgabewert gängig ist. Beim Aufruf der Routine und der Übergabe der zugehörigen Eingangsparameter, wird zwischen folgenden drei Arten unterschieden:

- **Call by Reference**: Der Aufrufer übergibt eine Referenz, also die Speicheradresse, als Parameter. Diese Variante wird auch *Call by Sharing* genannt, da die Variable zwischen dem Aufrufer und dem Aufgerufenen geteilt wird. Wird vom Aufgerufenen ein Wert an den formalen Parameter zugewiesen, ändert er damit auch den Wert des übergebenen Parameters und somit auch den Wert der dahinter liegenden Variable des Aufrufers.

- **Call by Copy**: Bei dieser Art der Parameterübergabe wird im Gegensatz zu *Call by Reference* verhindert, dass der Aufgerufene den Wert einer Variable beim Aufrufer direkt verändert. Der übergebene Parameter kann als Kopie gesehen werden und wird beim Aufgerufenen als lokale Variable gehandhabt. Bei *Call by Copy* wird zusätzlich unterschieden zwischen:

  - **Call by Value**: Der Informationsfluss erfolgt nur vom Aufrufer zum Aufgerufenen. Der Aufgerufene initialisiert seinen formalen Parameter mit dem übergeben Wert des Parameters.
  - **Call by Result**: Der Informationsfluss erfolgt nur vom Aufgerufenen zum Aufrufer. Der Wert der lokalen Variable wird am Ende des Aufrufes über den formales Parameter an die entsprechende Adresse des Aufrufer übergeben.
  - **Call by Value-Result**: Der Informationsfluss erfolgt sowohl vom Aufrufer zum Aufgerufenen beim Aufruf, als auch vom Aufgerufenen zum Aufrufer am Ende des Aufrufes und kombiniert somit die beiden zuvor genannten Methoden *Call by Value* und *Call by Result*.

- **Call by Name**: Anlog zu Call by Reference wird bei Call by Name eine Referenz bzw. Speicheradresse vom Aufrufer übergeben. Der Unterschied besteht aber darin, dass bei Call by Name bei jedem Zugriff auf den formalen Parameter durch den Aufgerufenen die Adresse zur übergebenen Referenz über den Namen neu evaluiert wird. Zu diesem Zeitpunkt kann sich diese Adresse schon wieder geändert haben und dies kann zu Problemen führen. Aus diesem Grund wird diese Art der Parameterübergabe in den heutzutage eingesetzten Programmiersprachen nicht mehr verwendet.

3.2 Strukturierung der Daten

Daten, samt ihren Eigenschaften, sowie Funktionen oder Operationen, die mit diesen Daten möglich sind, können als Basis jedes Computerprogramms gesehen werden. Ein Programm, als Blackbox gesehen, berechnet anhand von Eingabedaten und anhand definierter Regeln die Ergebnisdaten. Daten dienen unter anderem als Zwischenspeicher, Wertespeicher und über das Konzept des Typs, auch als Kategorisierung von im Programm gleich zu verwendenden Werten. Typen bzw. Typsysteme sind für Programmiersprachen sehr wichtig, denn damit wird ersichtlich, wie Daten zu interpretieren sind und welche Funktionen auf diesen Typ von Daten definiert sind. Heutzutage besitzen die meisten Sprachen ein Typsystem, auch wenn die Typen in manchen Fällen eher implizit und dynamisch berechnet werden und für Anwendende nicht immer offensichtlich sind. Zudem haben moderne Sprachen immer eine Menge an vorab definierten Datentypen, die von Anwendern direkt genutzt werden können oder auch zur Erstellung neuer Typen verwendet werden können.

Primitive und aggregierte Typen

Jede Programmiersprache wird mit einer Menge an vordefinierten, sozusagen eingebauten, Typen - daher auch im dem Englischen built-in types genannt - ausgeliefert, welche normalerweise das Verhalten der zugrundeliegenden Hardware widerspiegeln. Die Hardware hat hier unterschiedliche Sichten auf einen Bit-String. Der Bit-String "01010100"könnte zum Beispiel von einem Additionsbefehl als Zahl, aber von einem Ausgabebefehl als ASCII Zeichen interpretiert werden.

Folgende Beispiele sind in vielen Programmiersprachen als vordefinierte Typen zu finden:

- **Boolean**: Wahrheitswerte TRUE und FALSE
- **Character**: ASCII Zeichen
- **Integer**: Ganzzahlige Werte
- **Real**: Reelle Werte angenähert durch Darstellung über Mantisse und Exponent

Neben der Klassifizierung der Daten durch Typen dienen Typen auch dazu, eine falsche, fehlerhafte oder missbräuchliche Modifizierung der Daten aufzuzeigen und zu verhindern. Daten eines bestimmten Typen können nur auf die Art und Weise manipuliert werden, wie die für diesen Typ definierten Funktionen es zulassen. Der Einsatz von Typen bringt somit folgende Vorteile mit sich:

- **Abstraktion**: Moderne Programmiersprachen verbergen viele hardwarenahe Aspekte, um die Anwendenden bei ihrer Arbeit zu unterstützen und nicht mit technischen Routinen abzulenken. Solche Abstraktionen erhöhen die Lesbarkeit von Programmen und ermöglichen so das Erstellen von Programmen, welche sich auf das zugrunde

liegende Problem konzentrieren und nicht durch immer wiederkehrende Manipulationen von Bit-Strings für einfache Operationen durchzogen sind. Zudem ermöglicht die Abstraktion durch Typen, die Portierung von Programmen auf andere Systeme mit einer anderen internen Darstellung von Daten. Das Lesen und Schreiben von Daten wird in modernen Sprachen durch einfache Befehle ermöglicht, auch wenn dahinter für gegebenenfalls jede Ressource eine eigene komplizierte Abfolge von Low-Level-Befehlen verwendet wird.

- **Fehlererkennung zum Übersetzungszeitpunkt**: Sind dem Compiler die Typen der Variablen zur Übersetzungszeit bekannt, kann er bereits zu diesem Zeitpunkt einige Fehler wie fehlerhafte Zugriffe oder ungültige Operation mit dieser Variable erkennen.
- **Auflösung überladener Funktionen zum Übersetzungszeitpunkt**: Zusätzlich zur Fehlererkennung, kann der Compiler in statisch typisierten Programmiersprachen die Typen und somit auch die richtigen zugehörigen Maschinenbefehle für überladenen Funktionen auflösen. Die Abstraktionen in Programmiersprachen vermitteln das Gefühl, dass eine Addition von ganzzahligen Werten, Fließkommazahlen und Mischungen von beiden Arten alles das Gleiche ist und von dem Befehl, der hinter dem + Operator steckt, durchgeführt wird. Auf Maschinenebene können das durchaus sehr stark voneinander abweichende Befehle und Berechnungsarten sein. Das frühzeitige Wissen der richtigen Befehle erlaubt es, bei der Übersetzung entsprechende Optimierungen durchzuführen, die zum Beispiel bei dynamisch typisierten Programmiersprachen in der Art nicht möglich sind.
- **Präzision**: Der Einsatz von Typen ermöglicht den Anwendenden einer Programmiersprache, dem Compiler die genau für den Anwendungsfall nötige Menge an Speicher für die nötige Genauigkeit mitzuteilen. Dies ermöglicht es, dem Compiler wiederum spezifische Optimierungen durchzuführen.

Vordefinierte Typen werden auch primitive Datentypen oder elementare Datentypen bezeichnet. Hier muss aber berücksichtigt werden, dass es vordefinierte Datentypen geben kann, die nicht primitiv bzw. elementar sind. Der Typ String ist in vielen Sprachen bereits eingebaut, definiert wird er aber meist als Zeichenkette, gebildet aus einzelnen Zeichen vom Typ Character. Daher ist der Typ String, um genau zu sein, zwar ein vordefinierter, aber nicht primitiver Datentyp.

Wie am Beispiel String ersichtlich, können Programmiersprachen es den Anwendenden ermöglichen, eigene neue Datentypen zu definieren, die sich aus vordefinierten oder rekursiv, aus anderen zuvor definierten Datentypen, zusammensetzen. Solche Konstrukte werden aggregierte Datentypen genannt. Im Folgenden werden unterschiedliche Arten von aggregierten Datentypen näher betrachtet.

Kartesisches Produkt Das kartesische Produkt von n Mengen $A_1, A_2, ..., A_n$, ist definiert als die Menge von geordneten n-Tupel $(a_1, a_2, ..., a_n)$ Elementen, bei dem jedes

a_k zu A_k gehört. In Programmiersprachen werden die Elemente des kartesischen Produktes meist mit symbolischen Feldbezeichnung identifiziert. Über diese Feldbezeichnung kann jedes einzelne Element des n-Tupels, in der für die Programmiersprache definierten syntaktischen Notation angesprochen werden.

Finite Mapping Eine endliche Abbildung ist eine Funktion, die eine endliche Menge an Werten einer Definitionsmenge auf einen Bereichstypen als Zielmenge abbildet. In Programmiersprachen werden diese entweder intentional, über Mechanismen der Routinendefinition definiert, oder extensional, mittel eines Array-Konstruktors durch explizite Aufzählung definiert. Der Ausdruck *chardigits*[10]; definiert ein Mapping von Integer Werten im Bereich von 0 bis 9 auf eine Menge von Charactern. Zugegriffen wird auf die einzelnen Werte der Zielmenge über Indexierung mittels der Werte der Definitionsmenge, wie der Ausdruck *digits*[3] zeigt. Einige Programmiersprachen ermöglichen *slicing*, das Indexieren von mehr als einem Element des Bereiches. Durch den Ausdruck *digits*[3..5] wird ein ganzer Teilbereich angesprochen. Es gibt auch Programmiersprachen die nicht, wie sonst üblich, verlangen, dass die Definitionsmenge und/oder Zielmenge den gleichen Typ aufweisen muss. Somit ist folgender Ausdruck durchaus korrekt:

```
T = TABLE()
T<'RED'> = 'WAR'
T<6> = 20
T<4.6> = 'PEACE'
```

Listing 3.1: Assoziative Datenstruktur in SNOBOL4

Vereinigung Die Vereinigungsmenge von n Mengen, auch Disjunktion genannt, ist die Menge der Elemente, die in mindestens einer Elementenmenge der n Mengen enthalten sind. In Programmiersprachen ist dabei meist die Spezialform Kontravalenz bzw. ausschließende Disjunktion gemeint, die Elemente ausschließt, die in mehreren Mengen enthalten sind. Eine weitere Abwandlung der Vereinigung, die in Programmiersprachen oft gefunden werden kann, ist der Einsatz eines Hilfsfeldes oder sogenannten *tags* als Diskriminante, um in einer exklusiven Disjunktion kennzeichnen zu können, aus welcher Menge der Wert gewählt wurde. Diese Abwandlung wird auch bezeichnet als *discriminated union*, *tagged union* oder *variant record*. Eine Optimierung, die zwar auch die Gefahr von fehlerhaften Speicherzugriffen mit sich bringt, aber ebenfalls oft in Programmiersprachen gefunden wird, ist die Überlappung aller Varianten einer exklusiven Disjunktion auf die gleiche Speicheradresse. Die Interpretation und der Zugriff auf die Speicheradresse erfolgt dann über entsprechende Sichten anhand der Diskriminante.

Potenzmenge Die Potenzmenge ist die Menge aller Teilmengen einer gegebenen Grundmenge. Die Elemente einer Potenzmenge sind Mengen und somit sind Operation auf Potenzmengen immer Mengenoperationen, wie Disjunktion oder Konjunktion. In den meisten Programmiersprachen sind Potenzmengen nicht so stark vertreten und werden nicht von Haus aus so unterstützt, wie die bisher genannten Datentypen.

Sequenz Sequenzen oder auch Folgen genannt, sind eine Auflistung von endlich oder unendlich vielen Elementen eines bestimmten Typs. Die Unendlichkeit ist ein nicht so leicht in Programmiersprachen abbildbares Konzept. Je nachdem, ob die Sequenzen im Speicher abgelegt werden können oder nicht, kommen unterschiedliche Datenstrukturen wie Arrays oder rekursiv definierte Listen zum Einsatz.

Rekursion Das Konzept der Rekursion wird, wie bereits bei Sequenzen erwähnt, genutzt, um Konstrukte beliebiger Größe und Komplexität abzubilden. Ein rekursiver Datentyp T ist definiert als eine Struktur, die wiederum den Datentyp T beinhalten kann. Das gängigste Beispiel einer solchen rekursiven Struktur ist ein Binärbaum. Ein Binärbaum ist entweder leer, oder er besteht aus einer Wurzel mit einem linken und rechten Teilbaum, die wiederum Binärbäume sind. Rekursion wird in Programmiersprachen meist über Zeiger realisiert. Einzelne Elemente einer rekursiven Datenstruktur beinhalten immer auch Zeiger auf weitere zugehörige Datenelemente, beziehungsweise Speicheradressen. Das Konzept von Zeigern und die damit verbundene Arithmetik können schnell sehr komplex werden und sind somit auch sehr fehleranfällig.

Verbund Verbunddatentypen werden genutzt, um Strukturen zu definieren, die aus einer Komposition von mehreren anderen Typen erzeugt werden können. Die Definition kann als Liste von Feldern mit jeweils eigenständigen Typen gesehen werden. Im Speicher werden diese meist der Definitionsreihenfolge folgend hintereinander abgelegt und gegebenenfalls mit entsprechenden *Padding* zur Speicherplatzausrichtung ergänzt.

Abstrakte Datentypen Die bisher aufgezählten Datentypen geben durch den definierten Typ einer zusammengesetzten, selbst definierten Datenstruktur einen Namen. Dies erfüllt das Ziel der Klassifizierung durch Typen. Die Sicherstellung des richtigen Zugriffes wird erst mit abstrakten Datentypen ermöglicht. Abstrakte Datentypen erlauben es die Art und Weise, wie auf den Typen zugegriffen werden kann, zu definieren, während die interne Datenstruktur verborgen werden kann.

Typsysteme

Ein Typsystem kann als ein Regelwerk einer Sprache gesehen werden, welches definiert, wie die Verwendung der unterschiedlichen Typen der Sprache zu erfolgen hat. Vereinfacht dargestellt ist ein Typ eine Sammlung an Werten und Operationen, die auf diese Werte angewandt werden können.

Typfehler Bei dem Versuch ein Objekt mit einer unerlaubten, nicht im Typ des Objektes definierten Operation zu verändern, tritt ein Typfehler auf.

Typsicherheit Programme werden typsicher genannt, wenn garantiert werden kann, dass alle Operationen in diesem Programm nur auf Objekte ausgeführt werden, dessen

Typ diese Operation auch definiert. In einem typsicheren Programmen treten keine Typfehler auf.

Dynamische und statische Typprüfung Bei der Typprüfung wird versucht Fehler - syntaktische und semantische Fehler in der Nutzung der Programmiersprache - anhand des Typsystems zu erkennen. Dies kann dynamisch zur Laufzeit, durch Ausführung des Programms mit beispielhaftem Eingaben, oder statisch zum Übersetzungszeitpunkt, durch den Compiler erfolgen. Ein großer Nachteil der dynamischen Typprüfung ist, dass um einen Fehler zu finden, zuerst die richtigen, zu diesem Fehler führenden Eingabedaten gefunden und genutzt werden müssen. Statische Typprüfungen wiederum können nicht alle Fehler aufdecken, da gewisse Fehler erst zur Laufzeit erkannt werden können und nicht bereits im Vorfeld durch den Compiler allgemein ausgeschlossen werden können.

Stark und schwach typisiert Typsysteme, als Regelwerk gesehen, sollen die Erstellung von Programmen verhindern, die undefinierte Operationen auf Objekte eines Typs ausführen würden. Ein Typsystem wird als stark typisiert bezeichnet, wenn es Typsicherheit garantieren kann - also gartieren kann, dass Programme, die sich an das Typsystem halten, garantiert in keine Typfehler laufen. Eine stark typisierte Sprache besitzt ein stark typisiertes Typsystem und der Compiler dieser Sprache kann zum Übersetzungszeitpunkt Typsicherheit garantieren.

Typäquivalenz Bei der Übergabe von Parametern sind Programmiersprachen meist flexibler als es ein sehr striktes Typsystem verlangen würde. Wird zum Beispiel ein bestimmter Typ T als Operand erwartet, ist es auch möglich einen anderen Typen Q zu übergeben, ohne die Typsicherheit zu gefährden. In diesem Fall muss die Sprache definieren, dass der Typ Q äquivalent zum Typ T ist. Wird diese Kompatibilität im Typsystem definiert, so kann die Typprüfung jederzeit die korrekte Verwendung solcher Operationen überprüfen. Es wird unterschieden zwischen der *Namensäquivalenz*, die rein auf den gleichen Typnamen prüft, und der *Strukturäquivalenz*, die rekursiv die interne Struktur des Typen für die Prüfung in Betracht zieht.

Typumwandlung In dem Fall, dass die Typen nicht als äquivalent definiert sind, ist eine Typumwandlung nötig. In vielen Programmiersprachen wird eine implizite Typumwandlung durch den Compiler durchgeführt. Diese implizite Typumwandlung kann für jede Operation eigenständig definiert sein. Zum Beispiel kann in dem Ausdruck $x = x + z$; mit x als ganzzahliger Wert und z als Fließkommazahl, zuerst x implizit auf einen Fießkommatyp umgewandelt werden, um die arithmetische Operation + durchgeführt werden kann und das Ergebnis wiederum implizit auf einen ganzzahligen Typ umgewandelt werden, damit die Zuweisung = durchgeführt werden kann. Neben der impliziten Typumwandlung gibt es in den meisten Sprachen auch eine explizite Typumwandlung durch spezielle Schlüsselwörter, die die Sprache zur Verfügung stellt.

Untertyp In der Darstellung eines Typen als Menge an Werten mit zugehörigen erlaubten Operationen stellt vereinfacht beschrieben ein Untertyp eines Typen eine Teilmenge dieser Werte mit den gleichen zugehörigen erlaubten Operationen dar. Angepasste erlaubte Operationen für den Untertyp sind ebenfalls möglich und finden in objektorientierten Sprachen häufig ihre Anwendung. Untertypen können dort verwendet werden, wo ihre sogenannten *Supertypen* verlangt werden.

Generischer Typ Die Definition von abstrakten Datentypen kann in modernen Sprachen parametrisiert werden und wird somit auch als generische Definition von Typen angesehen. Auch bei generischen Typen kann Typsicherheit garantiert werden.

Polymorphie Moderne Programmiersprachen unterstützen alle eine Art der Polymorphie. Monomorphe Typsysteme, bei dem jedes Objekt und jede Operation genau nur einen Typ hat, sind in der Praxis schwer zu finden. Abbildung 3.1 zeigt die unterschiedlichen Arten der Polymorphie.

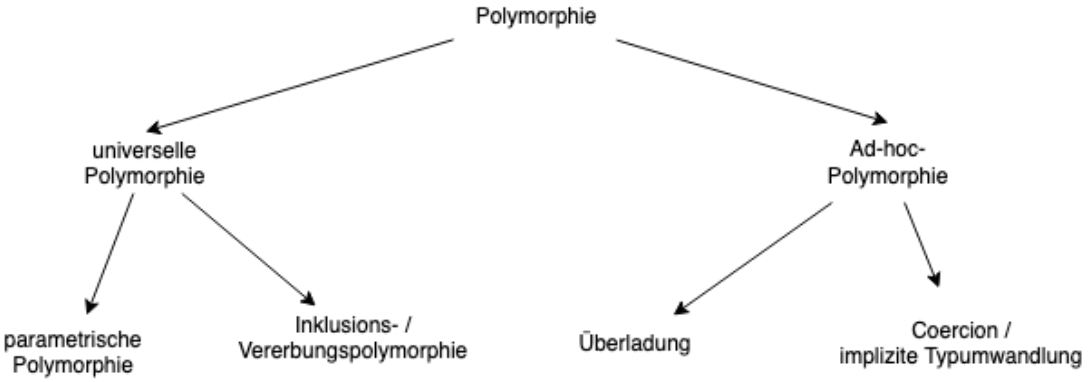

Abbildung 3.1: Arten von Polymorphie

3.3 Strukturierung der Berechnung

Nachdem im vorhergehenden Unterkapitel die Eigenschaften unterschiedlicher Datentypen genauer beleuchtet wurden, soll im folgenden Abschnitt ein Überblick zu den Kontroll- und Berechnungsstrukturen innerhalb eines Programms gegeben werden.

Ein Programm besteht im Grunde aus Ausdrücken (*expressions*) und Anweisungen (*statements*). Ein mathematischer Ausdruck mit mehreren Operatoren (welche wiederum unär, binär oder n-är sein können) kann in unterschiedlichen Notationen (prefix, postfix, infix) ausgeschrieben werden, die sich unter Umständen semantisch voneinander unterscheiden können (siehe den ++ Operator in Sprachen wie Java oder C).

Die Auflösung eines Ausdrucks folgt grundsätzlich mathematischen Regeln, benötigt jedoch genauere Definitionen bei Ausdrücken in denen Operatoren gleichen Ranges genutzt werden. Dementsprechend wird beispielsweise in Pascal vorgegeben, dass in einem solchen Fall die Operatoren des Ausdrucks von links nach rechts angewandt werden. Ähnliches gilt für Anweisungen; so wird der = Operator in C von rechts nach links assoziiert, wodurch folgende Statements äquivalent sind:

```
a = b = c = 0
a = (b = (c = 0))
```

Listing 3.2: Operatorassoziativität in C

Die Kommunikation unterschiedlicher Programmteile untereinander kann über Seiteneffekte geschehen, indem nicht-lokale (`nonlocal`) Variablen modifiziert werden. Problematisch hier ist die Unterscheidung zwischen gewollten und ungewollten Seiteneffekten, die durchaus schwierig nachzuvollziehen sein können. Ein solcher ungewollter Seiteneffekt tritt beispielsweise auf, sollte eine Programmeinheit $U1$ die Einheit $U2$ aufrufen und infolgedessen die nicht-lokale Variable `x` modifizieren, die jedoch zur Kommunikation zweier anderer Programmeinheiten gedacht ist.

Abhilfe kann hier geschaffen werden indem Parameter als einzige Möglichkeit der Kommunikation zwischen Programmeinheiten zur Verfügung gestellt werden. Die Einschränkung von Variablenzugriffen für genau bestimmte Programmeinheiten sowie die Möglichkeit solche Variablen `read-only` zu deklarieren sind weitere Mechanismen zur Reduktion ungewollter Seiteneffekte.

Die Kommunikation über Parameter kann jedoch aufgrund von *aliasing* dennoch zu ungewollten Seiteneffekten führen. Dies passiert wenn zwei Variablen so modifiziert werden, dass sie im Rahmen des Programmablaufs unerwarteterweise zu Aliasen werden, wodurch die Manipulation der einen Variable auch zu einer Änderung der anderen führt.

3.4 Strukturierung des Programms

Das Programmieren großer Systeme bringt einige neue Herausforderungen mit sich, die bei kleineren Programmen nicht auftreten. Begründet kann dies damit werden, dass Methoden, welche in kleinen Programmen adäquate Lösungen darstellen, nicht sonderlich gut skalieren. Die zwei fundamentalen Prinzipien, die das Programmieren großer Systeme (*programming in the large*) beeinflussen und von dem kleinerer Systeme unterscheiden, sind die **Abstraktion** und die **Modularität**.

Module sind kleinere, voneinander unabhängige Programmeinheiten, die zusammengefasst das Softwaresystem bilden. Abstraktionen werden entwickelt, um ein besseres Verständnis für die Problematik zu erlangen, die gelöst werden soll. Das Grundprinzip ist, dass je näher ein in Module unterteiltes Programm an die entwickelten Abstraktionen herankommt, desto einfacher ist es zu verstehen und zu handhaben. Ein weitgehend bekannter Ansatz, der diese Strategie zusammenfasst, ist das Prinzip von *divide and conquer*.

Der zugrundeliegende Mechanismus, welcher den Ansatz der Modularität unterstützt und von Programmiersprachen auf unterschiedliche Art und Weise umgesetzt wird, ist das Konzept der Datenkapselung (*Encapsulation*). Eine Programmeinheit, welche aus Prozeduren und Funktionen besteht, stellt für ein aufrufendes Programm (*Client*) Services zur Verfügung. In diesem Fall wird das aufgerufene Programm als *Server* oder *Service Provider* bezeichnet, welches sein Service „in gekapselter Form“ bereitstellt.

Aus diesem Grund werden Module über die Datenkapselung in zwei Teilen beschrieben: **Interface** und **Implementierung**. Während die Implementierung das Innere des Moduls beschreibt, bezieht sich das Interface auf den vom Modul bereitgestellten Service und wie auf diese zugegriffen werden kann. Die Methoden des Interfaces eines Moduls werden *exportiert* und wiederum bei Bedarf von Clients *importiert* um von diesen genutzt zu werden.

Die dadurch entstehenden Vorteile sind zum einen das Kombinieren von Programmkomponenten, die gesammelt einen Service darstellen und zum anderen die Möglichkeit, durch die Modifikation der Sichtbarkeit von Programmaspekten den Clients nur die für sie relevanten Aspekte der Programmeinheit offenzulegen. Der Vorteil einer solchen Implementierung ist ein übersichtlicherer Client, da dieser nicht wissen muss, wie ein Service funktioniert, um ihn zu nutzen sowie eine gewisse Autonomie des Services, da dieser unabhängig von seinen Clients modifiziert werden kann.

3.5 Sprachparadigmen

Eine weitere Möglichkeit Programmiersprachen zu klassifizieren sind Sprachparadigmen. Die Paradigmen einer Programmiersprache geben dem Programmcode ein grundlegendes Konzept bezüglich der Daten, Berechnung und Strukturierung. Auch wenn Programmiersprachen mehrere Sprachparadigmen unterstützen, gibt es häufig ein Sprachparadigma, welches fest im Grunddesign der Sprache verankert ist. Abbildung 3.2 zeigt eine vereinfachte Darstellung der unterschiedlichen Sprachparadigmen.

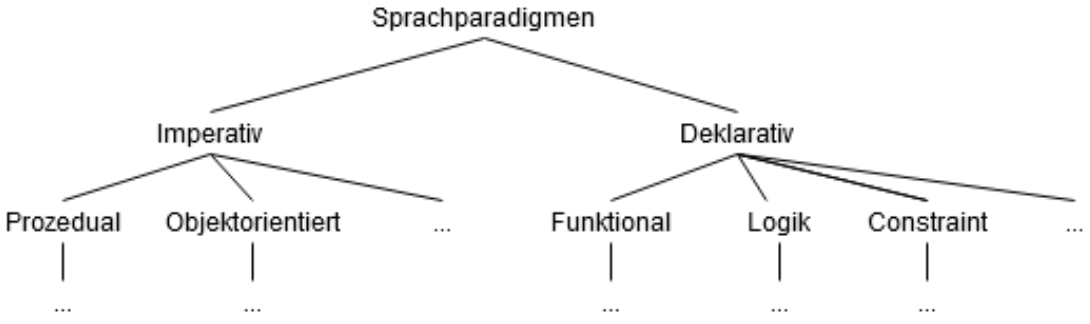

Abbildung 3.2: Sprachparadigmen

3.5.1 Objektorientierte Programmiersprachen

Sprachen, die als objektorientierte Programmiersprachen bezeichnet werden, wurden von Grund auf mit dem Ziel entwickelt die zwei Prinzipien der Modularität und Abstraktion zu erfüllen.

Allgemein werden objektorientierte Programmiersprachen durch vier besondere Eigenschaften charakterisiert:

Abstrakte Datentypen Eine objektorientierte Programmiersprache ist in der Lage abstrakte Datentypen zu definieren. Diese stellen die Einheiten der zuvor beschriebenen Modularität dar. Ein Beispiel der Umsetzung von abstrakten Datentypen ist das Klassenkonstrukt. **Objekte** sind in diesem Zusammenhang instanziierte Klassen, die wiederum **Instanzvariablen** und eine Reihe von Methoden besitzen.

Vererbung Eine Besonderheit, die mit diesen Datentypen einhergeht, ist das Konzept der Vererbung. Diese Besonderheit zeichnet sich dadurch aus, dass abstrakte Datentypen Eigenschaften anderer, zuvor definierter Datentypen erben und weiter verfeinern können. Dadurch können Eigenschaften, die in mehreren Datentypen vorkommen, wiederverwendet werden.

Inklusionspolymorphie Die durch Vererbung entstandene Beziehung der verschiedenen Klassen erlaubt es polymorphe Variablen zu nutzen, da alle Klassen, die von derselben Basisklasse entstanden sind, als besondere Formen dieser gesehen werden können.

Die Eigenschaft, Methoden sowohl auf Untertypen, als auch auf Basistypen anwenden zu können, wird als Inklusionspolymorphie bezeichnet.

Dynamisches Binden von Funktionsaufrufen Bei der Vererbung von abstrakten Datentypen gibt es in objektorientierten Programmiersprachen die Möglichkeit geerbte Funktionen abzuändern (*Override*). Durch diese Eigenschaft, in Kombination mit der Inklusionspolymorphie und der entstandenen Klassenhierarchie, ergeben sich nun bei einem Methodenaufruf unterschiedliche Möglichkeiten welche Methode tatsächlich ausgeführt wird: die der Basisklasse oder die neu definierte Methode der Unterklasse.

Mit dem dynamischen Binden wird der Mechanismus beschrieben, welcher zur Laufzeit, abhängig von dem Objekt entscheidet, ob die Methode der Basis- oder der Unterklasse aufgerufen wird. Von statischer Bindung wird gesprochen, wenn jederzeit die Methode der Basisklasse gewählt wird.

Typen und Subtypen Ein für die Vererbung wichtiges Konzept ist die Definition von Typen und Subtypen. Allgemein ist ein Typ eine Menge bestehend aus Werten und dazugehörigen Operationen. In diesem Kontext bezeichnet ein *Subtyp* eine Teilmenge dieser. Die übergeordnete Menge wird üblicherweise auch als *Supertyp* bezeichnet. Operationen, die auf einem Supertyp definiert werden, sind automatisch auch auf den

Subtyp anwendbar, was sich in der Objektorientierung durch die Inklusionspolymorphie ausdrückt.

Subtyping ist in der Objektorientierung entscheidend für die Beziehung von Objekten des Subtyps und jenen des Supertyps (auch *parent type* genannt), sodass Objekte des ersteren ebenfalls als Objekte des letzteren gelten.

Die Unterscheidung, ob es sich um einen Subtyp oder eine Unterklasse handelt, wird auf Basis der Kompatibilität gemacht. Wird beispielsweise die Funktion einer Basisklasse in einer Unterklasse so abgeändert, dass keine Kompatibilität mehr gegeben ist (zum Beispiel durch eine Signatur mit anderen Variablentypen) so spricht man von einer Unterklasse (*Subclass*) und nicht mehr von einem Subtyp.

3.5.2 Funktionale Programmiersprachen

Das Sprachparadigma der funktionalen Programmierung unterscheidet sich dahingehend von der objektorientierten, als dass sich diese auf die Theorie mathematischer Funktionen stützt. Werden Programmeinheiten in der Objektorientierung als Objekte zusammengefasst, um Seiteneffekte zu reduzieren, können diese in der funktionalen Programmierung, aufgrund der besonderen Eigenschaften mathematischer Funktionen, vollständig eliminiert werden.

Die Unterschiede zu imperativen Programmiersprachen lassen sich anhand dreier Grundkonzepte imperativer Sprachen illustrieren: *Variablen*, *Zuweisungen* sowie *Sequenzierung*.

In der imperativen Programmierung stellt eine Variable einen bestimmten Speicherbereich dar und kann durch eine Zuweisungsoperation im Laufe eines Programms modifiziert werden. Dadurch ist die Ausgabe eines solchen Programms zwangsweise von der Zuweisungsreihenfolge abhängig. Im Unterschied dazu spricht man in funktionalen Sprachen nicht von Variablen, sondern von Werten (eng. *Values*), die nicht von der Ausführungsreihenfolge abhängig, jedoch vielmehr wie in klassischen mathematischen Funktionen, an einen unveränderlichen Wert gebunden sind.

Die Verwendung von Schleifenaufrufen, um einen Wertbereich zu durchsuchen sowie eine Variable iterativ zu erhöhen, ist eine grundlegende Methode imperativer Programmiersprachen. In ihren funktionalen Pendants hingegen werden solche Berechnungen rekursiv unternommen, da das Konzept von Schleifenaufrufen nicht existiert.

Mathematische Programmierung Im mathematischen Kontext beschreibt eine Funktion die Beziehung zweier Mengen zueinander. Man spricht hier auch von einer Definitionsmenge und einer Zielmenge. Die Definition dieser beiden Mengen wird als *Signatur* bezeichnet, wohingegen die *Mapping Rule* (Rechenvorschrift) die Funktionsoperation selbst beschreibt.

Funktionale Programmiersprachen lassen sich somit durch zwei Grundprinzipien beschreiben: *Binding* und *Application*. Binding wird genutzt, um Daten an Namen zu binden,

```
square :: Int -> Int          -- Signatur
square x = x*x                -- Mapping rule
```

Listing 3.3: Funktionsdefinition in Haskell

wobei gilt, dass Werte und Funktionen gleichrangige Daten sind und somit beide als Parameter übergeben werden können. Application wird genutzt, um neue Werte zu berechnen.

Lambda-Kalkül Das Lambda-Kalkül entspricht einer einfachen formalen Sprache, welche die Berechnungen einer Funktion abbildet. Ausdrücke des Lambda-Kalküls können einzelne Identifier, Funktionsdefinitionen oder Funktionsanwendungen sein.

$$x \in Exp \Longleftarrow x \in Id; \qquad \text{(Identifier)}$$
$$(e_1 e_2) \in Exp \Longleftarrow e_1, e_2 \in Exp \qquad \text{(Funktionasanwendung)}$$
$$(\lambda x.e) \in Exp \Longleftarrow x \in Id, e \in Exp \qquad \text{(Funktionsdefinition)}$$

Die Funktionsdefinition $\lambda x.x * x$ beschreibt eine anonyme Funktion, die ein gegebenes x quadriert. Die entsprechende Funktionsanwendung auf die Zahl 2 wäre somit $((\lambda x.x * x)2)$. Bei der Auswertung solcher Ausdrücke herrscht Linksassoziativität.

Des Weiteren muss zwischen **gebundenen** und **freien** Variablen unterschieden werden. Gebundene Variablen sind jene, die durch ein λ eingeführt wurden. Alle übrigen Variablen werden als freie Variablen bezeichnet. Beispielsweise stellt das k im Ausdruck $\lambda x.x^k$ eine solche freie Variable dar.

Um die Berechnung der Funktion zu verstehen, benötigt man das Konzept der Substitution, die den Vorgang beschreibt, in welchem jeder formale Parameter durch einen aktuellen ersetzt wird. Ein Term t, in dem die Variable x durch s substituiert wird, wird formal als $[s/x]t$ beschrieben. Eine solche Substitution ist nur dann erlaubt, wenn die freie Variable s danach weiterhin ungebunden ist.

Die Auswertung einer Funktion erfolgt mittels dreier Umformungsregeln:

- *α-Konversion* (Umbenennung): Diese Regel besagt, dass alle gebundenen Variablen durch einen anderen Namen ersetzt werden können, ohne die Bedeutung des Ausdrucks zu verändern. Zwei Terme, die sich nur durch den Variablennamen unterscheiden, werden auch als α-äquivalent bezeichnet.

$$\lambda x.e \leftrightarrow \lambda y.[y/x]e \quad \text{(y nicht frei in e)}$$

- *β-Konversion* (Anwendung): Hier werden die gebundenen Variablen durch die Argumente ersetzt und die Funktion reduziert.

$$(\lambda x.e_1)e_2 \leftrightarrow [e_2/x]e_1$$

- *η-Konversion*: Im Sinne der Extensionalität, die besagt, dass zwei Funktionen, die für den gleichen Input den gleichen Output produzieren, äquivalent sind, werden in diesem Schritt redundante Funktionen eliminiert.

$$\lambda x.(ex) \leftrightarrow e \quad \text{(x nicht frei in e)}$$

Neue Funktionen können offensichtlich auch aus der Kombination mehrerer Funktionen bestehen (*Komposition*). Unter Benutzung dieser Regeln werden diese dann in Reduktionsschritten interpretiert, bis eine nicht weiter reduzierbare Form (*Normalform*) entsteht, die das Ergebnis darstellt.

KAPITEL 4

Hauptteil

In diesem Kapitel werden zuerst Anforderungen an Smart Contract Sprachen evaluiert. Hierzu werden Historische Anforderungen, Anforderungen anhand aktueller Anwendungen, Anforderungen anhand der allgemeinen Sprachentwicklung und Anforderungen anhand gängiger Smart-Contract-Sprachen betrachtet. Weiteres wird in diesem Kapitel analysiert, inwieweit die gefundenen Anforderungen spezifischen Spracheigenschaften zugeordnet werden können. Im letzten Teil dieses Kapitels wird Solidity anhand der Erkenntnisse dieses Kapitels hinsichtlich der Anforderungen und Spracheigenschaften validiert.

4.1 Anforderungen an Smart Contracts

4.1.1 Historische Anforderungen

Bereits im Jahr 1996 beschreibt Nick Szabo, der als Erfinder des Begriffes gilt, Smart Contracts wie folgt: *„A smart contract is a set of promises, specified in digital form, incuding protcols within which the parties perform on these promises“* [Sza96]. In dieser Arbeit geht Szabo auf die Grundprinzipien ein, die beim Aufbau eines Smart Contracts in Betracht gezogen werden sollten. Diese werden im weiteren Verlauf kurz beschrieben.

- **Observability** Mit Observability wird das Prinzip beschrieben, dass alle Vertragsteilnehmenden, die für die Vertragserfüllung relevanten Handlungen des bzw. der jeweils anderen Vertragsteilnehmenden zu einem gegebenen Zeitpunkt überprüfen können. Dadurch wird allen Parteien ermöglicht zu wissen, ob die Bedingungen des Vertrages eingehalten werden. Im Rahmen eines Smart Contracts wird dieses Prinzip mittels Code umgesetzt, der nur ausgeführt wird, wenn ein bestimmtes Set an Bedingungen erfüllt wurde.
- **Verifiability** Ein Vertrag soll den Teilnehmenden ermöglichen, einer unabhängigen Kontrollaufsicht beweisen zu können, ob eine der involvierten Parteien vertraglich

festgelegte Punkte missachtet hat oder im Zweifelsfall Vertragsbruch begangen wurde. Diese Verifizierbarkeit ist prinzipiell von zwei Faktoren abhängig:

- Die Vertragsteilnehmenden stellen die Information zur Verfügung und
- die Information muss für die Kontrollaufsicht zugänglich sein.

- **Privity** Im traditionellen Vertragswesen bedeutet Privity, dass lediglich die unterschreibenden Parteien an die Vertragsbedingungen gebunden sind. Bei Smart Contracts wird dieser Begriff breiter gefasst und dadurch ergänzt, dass Informationen und Inhalte eines Smart Contracts den beteiligten Parteien nur insoweit wie notwendig zugänglich gemacht werden sollen. Damit soll verhindert werden, dass der zugrundeliegende Code von Fremden eingesehen oder manipuliert wird.

- **Enforceability** Das vierte und letzte Grundprinzip bezieht sich auf die Vollstreckbarkeit eines Vertrages. Im Optimalfall kann durch gute Verifizierbarkeit die Notwendigkeit von Vollstreckbarkeit reduziert werden. Die Besonderheit des Smart Contracts ist hierbei die Automatisierung dieses Grundprinzips mittels `self-enforceability`.

4.1.2 Anforderungen anhand aktueller Anwendungen

Bartoletti und Pompianu [BP17] gehen in ihrem Paper auf die verschiedenen Anwendungsmöglichkeiten und übliche Designmuster beim Entwerfen von Smart Contracts ein. Die Autoren haben hierfür fünf Kategorien definiert, die sie als relevant sehen, sowie allgemein neun Designmuster erkannt.

4.1.2.1 Anwendungen

- **Finanziell**: Der klassische Anwendungsfall von Smart Contracts, in welchem die Steuerung des Geldflusses im Vordergrund steht.

  – *Crowdfunding und Initial Coin Offering (ICO)*
  Eine der Hauptanwendungsfälle für Smart Contracts ist die Durchführung von speziellen Crowdfundings, den sogenannten ICOs. Hierbei werden Projekte im Blockchain-Ökosystem vorgestellt, die durch den Verkauf von Tokens finanziert werden sollen. Laut [FMMT18] erreichte diese Art der Finanzierung einen ersten Hochpunkt im Jahr 2017, als in den letzten acht Monaten diesen Jahres der gesammelte Wert aller durchgeführten ICOs einen Wert von vier Milliarden US-Dollar überschritt. Im darauffolgenden Jahr wurde dieser Wert mit 7,8 Milliarden US-Dollar fast verdoppelt, wie in [Ico19] zu sehen ist. Der Großteil dieser wurde über Smart Contracts auf der Ethereum Blockchain durchgeführt, besonders mittels ERC-20 Token Standard Contracts.

Da diese Anwendung besonders beliebt ist, ist ein solcher Contract meistens durch Vererbung von bestehenden Contracts schnell erstellt. Unter anderem können folgende Funktionen in einem solchen Contract gefunden werden:

* Tokenname (bspw. `name = "Ethereum"`)
* Tokensymbol (bspw. `symbol = "ETH"`)
* Gesamtmenge der verfügbaren Tokens `uint256 public totalSupply`
* Durch `mapping` wird gespeichert, welcher Account welche Menge an Tokens besitzt
* `transfer` Funktion, die Nutzenden erlaubt Token zu versenden
* Eine `approve` Funkion, die einem anderen Account das Ausgeben von Tokens erlaubt

Werden nun Tokens an den Smart Contract überwiesen, bekommen die Teilnehmenden eine durch einen meist fest vorgegebenen Wechselkurs entsprechende Menge an neuen Tokens zurückgesendet. Durch die öffentliche Einsehbarkeit ist ein großes Level an Transparenz möglich, wodurch InvestorInnen jederzeit den aktuellen Stand der ICO sehen können.

– *Dezentrale Börsen*
Ein verwandter Anwendungsfall für Smart Contracts ist jener der sogenannten `decentralized exchanges`. Da viele Kryptowährungen vor allem als Spekulationsobjekte genutzt werden und dadurch auf eigens dafür vorgesehenen Börsen gehandelt werden, stellen diese durch ihre zentralisierte Eigenschaft ein großes Risiko dar. Sind solche Börsen nicht abgesichert und wird eine potentielle Schwachstelle im System entdeckt, so kann dies zu massiven finanziellen Schäden führen, die auch Börsen aus finanziellen Gründen zur Schließung zwingen kann.

Prominentestes Beispiel ist der Mt.Gox Hack [Led19] bei dem Bitcoin im (damaligen) Wert von ca $460 Millionen US-Dollar entwendet wurden. Doch sind auch bis heute Krypto-Börsen ein sehr lukratives Ziel für Angriffe. Wie in [Lar18] gezeigt, wurden im Jahr 2018 allein durch Angriffe auf Krypto-Börsen über $865 Millionen US-Dollar gestohlen.

Dezentrale Börsen versprechen eine erhöhte Sicherheit gegenüber traditionellen Börsen, indem Kauf, Verkauf und Auftragsbücher vollständig durch Smart Contracts verwaltet werden. Diese Smart Contracts beinhalten Funktionen, die den Austausch von Token zwischen den Teilnehmenden durchführen. Die Funktionsweise hier ist ähnlich zu klassischen Börsen:
Nachdem Angebote von Verkaufenden spezifiziert wurden (Token, Verkaufspreis und Dauer), können Interessierte ihre Angebote abgeben. Zum Ablauf der Dauer führt der Smart Contract die Transaktion durch. Ein wesentlicher Unterschied zu klassischen Krypto-Börsen, bei welchen es zunächst notwendig

ist die eigenen Token direkt an das Wallet der Börse zu senden, und dieser somit zu vertrauen, finden Transaktionen in dezentralen Börsen direkt von dem eigenen Wallet statt. Dadurch kann ein großer Teil potentieller Angriffsvektoren vermieden werden. Technisch bedingt sind dezentrale Börsen zum aktuellen Zeitpunkt auf eine gewisse Tokenart eingeschränkt, da noch keine Interoparabilität zwischen verschiedenen Blockchains möglich ist.

- *Kredite*
  Ein weiterer Anwendungsfall für Smart Contracts ist die Verwaltung von Krediten und Darlehen ohne die Beteiligung einer dritten Person. Unter dem Begriff `decentralized lending` werden verschiedene Plattformen zusammengefasst, die Darlehen mittels Smart Contracts verwalten und auszahlen. Nach ICOs ist die dezentrale Kreditvergabe der größte Markt für Anwendungen von Smart Contracts gemessen am Kapital, mit einem Volumen von um $250 Millionen, die von solchen Plattformen verarbeitet wurden. [Blo19] Der Großteil dieser Plattformen wird von Nutzenden für passives Einkommen genutzt, die gegen Zinsen ihre Token verleihen, womit die Kreditnehmer meistens Spekulationshandel betreiben.

- *Versicherungen*
  Einem ähnlichen Modell folgen auch durch Smart Contracts unterstützte Versicherungsangebote, die als `decentralized insurance` vermarktet werden. Primär werden hier parametrische Versicherungen abgeschlossen - also solche, die durch ein genaues Set an Daten potentielle Ereignisse definieren können und somit automatisiert Auszahlungen an die versicherte Person machen können. Als Beispiel kann hier eine Flugverspätung oder der Ernteausfall eines Bauern durch schlechte Witterung genannt werden.

- **Notariell**: Die unveränderliche Natur der Blockchain wird in Anwendungsfällen dieser Kategorie genutzt. Smart Contracts können dadurch als Besitzurkunden genutzt werden oder um die Echtheit eines Dokuments zu beweisen, indem dieses über einen Smart Contract auf der Blockchain abgebildet und dadurch permanent gesichert wird (bspw. Proof of Existance[1]). Darüber hinaus kann diese Eigenschaft auch dazu eingesetzt werden, um den Besitz von geistigem Eigentum zu beweisen, wovon vor allem Kunstschaffende profitieren können.

- **Spiele**: Die Anwendungsfälle von Smart Contracts für die Spieleindustrie sind zweierlei Natur. Zum einen sind Glücksspiele ein weiteres Gebiet in dem Potential für den Einsatz von Smart Contracts gesehen wird. Plattformen, die in der Glücksspielindustrie agieren und ihre Spiele durch Smart Contracts durchführen

[1] https://proofofexistence.com

lassen, werben mit überdurchschnittlicher Fairness, die klassische Casinos nicht bieten können.

Zum anderen entstanden Smart Contracts, um digitale Sammlerstücke zu kreieren, die bewiesenermaßen einzigartig sind und getauscht oder verkauft werden können (bspw. *CryptoKitties*[2]).

- **Wallets**: Smart Contracts dieser Kategorie verwalten Schlüssel und Güter, sowie das Senden von Transaktionen. Diese können einen oder mehrere Besitzer haben. Diese Contracts dienen oft als Abstraktion, um die Interaktion zwischen Blockchain und den Nutzenden zu vereinfachen. Die Smart Contracts können beispielsweise dazu genutzt werden, die klassischen hexadezimalen Adressen durch Usernamen zu ersetzen, wodurch der Einstieg vereinfacht werden kann. Smart Contract Wallets können auch mehrere EigentümerInnen haben.

- **Library**: Diese Contracts werden von anderen Smart Contracts dazu aufgerufen, um vordefinierte Operationen durchzuführen. *OpenZeppelin*[3] ist ein sehr bekanntes Beispiel für diese Art von Smart Contracts.

4.1.2.2 Design Patterns

- **Token**: Zentraler Aspekt des Blockchain-Ökosystems ist die Tokenisierung von Vermögenswerten oder Wirtschaftsgütern. Token können dabei verschiedenste digitale oder physische Güter bzw. eine Reihe von Berechtigungen (beispielsweise Stimmrechte oder Unternehmensanteile) für die besitzende Person repräsentieren.

  Sind Token für einen bestimmten Service gedacht und damit die einzige Möglichkeit mit einem Smart Contract zu interagieren, spricht man von *Utility Token*, wohingegen Token, deren Wert von externen Vermögenswerten abgeleitet wird, als *Security Token* bezeichnet werden. Durch die breite Einsetzbarkeit und Nutzung wurde die Standardisierung der Tokens auf der Ethereum Blockchain mit dem ERC-20 Standard [Eth19] durchgeführt.

- **Authorization**: Dieses Muster dient primär der Sicherstellung von korrekter Codeausführung durch Zugriffsrechte. In den meisten Fällen wird durch sogenannte `access restriction pattern` überprüft, ob die aufrufende Adresse die entsprechenden Berechtigungen hat, um einen Methodenaufruf durchzuführen. Mittels `ownership pattern` können Entwickelnde besonders kritische Funktionen mit Zugriffsbeschränkungen ausstatten, sodass diese nur noch von der eigenen Adresse aufgerufen werden können.

[2] https://www.cryptokitties.co
[3] `https://openzeppelin.com/`

- **Oracle**: Da in Smart Contracts manche Funktionen von Informationen außerhalb der Blockchain abhängig sind, braucht man geeignete Mechanismen, um diese zur Verfügung zu stellen. Da Smart Contracts innerhalb des Ethereum-Ökosystems Daten jedoch nicht direkt von externen Seiten anfordern können, werden Oracles als Schnittstelle dazwischen positioniert, wodurch externe Informationen in die Blockchain geladen werden. Oracles selbst sind spezielle Smart Contracts, die in direkter Verbindung mit der externen Datenquelle stehen und werden demnach auch als `data provider` bezeichnet. Durch dieses System wird garantiert, dass gleiche Anfragen von verschiedenen Nodes dasselbe Resultat zurückliefern, womit der Determinismus gewahrt wird.

- **Randomness**: Das Generieren von Zufallszahlen in einer deterministischen Turingmaschine, wie es Ethereum ist, stellt eine massive Herausforderung für Entwickelnde dar. Vor allem für jene Smart Contracts, in welchen Glücksspiele abgebildet werden, sind Zufallszahlen von größtmöglicher Bedeutung, um Fairness für die Spielenden zu garantieren. Das Problem liegt darin, ähnlich wie bei Orakeln, dass bei denselben Anfragen dasselbe Resultat garantiert werden muss. Darüber hinaus ist es durch die Offenheit der Blockchain schwierig eine geeignete Quelle für den Zufall zu finden, deren Wert trotz dieser Einsehbarkeit nicht vorherzusagen ist. Lösungsansätze hierfür sind speziell dafür konzipierte Orakel, welche die Zufallswerte off-chain generieren und diese dann für den Smart Contract bereitstellen. Eine Alternative wäre es, Hashes von noch nicht kreierten oder alten Blocks als Grundlage zu nehmen. Diese Methode wird jedoch in [Reu18] als unsicher eingestuft.

- **Poll**: Polls oder Umfragen sind meist Teil eines komplexeren Smart Contracts und werden dazu genutzt, um Nutzenden die Möglichkeit zu geben mittels Stimmenabgabe die Ausführung des Smart Contracts zu beeinflussen. Die Stimmabgabe erfolgt meistens durch spezielle Tokens und damit verbundene Adressen.

- **Time Constraint**: Manche Smart Contracts bedürfen einer genauen zeitlichen Einschränkung, die definiert, wann sie aufgerufen werden können. Dies ist vor allem notwendig, wenn der Zustand von externen Ereignissen abhängt, deren Zeitpunkt bekannt ist. Ein zeitlich beschränkter ICO-Smart Contract kann ab einem bestimmten Zeitpunkt dadurch keine weiteren Transaktionen mehr annehmen. Wird ein Smart Contract für notarielle Zwecke eingesetzt, kann durch dieses Design Pattern ein Zeitpunkt definiert werden, zu dem ein Dokument in den Besitz einer anderen Person übergeht.

- **Termination**: Ein auf der Blockchain eingesetzter Smart Contract muss gegebenenfalls zu einem gewissen Zeitpunkt terminieren. Da es aber durch die Bauweise

der Blockchain nicht möglich ist diesen einfach zu löschen, muss bereits beim Entwurf Code implementiert werden, der den Contract deaktiviert. Dies kann manuell durch ad-hoc Code oder automatisch mittels Aufruf der `selfdestruct` Funktion erreicht werden. Im Normalfall ist dies nur für die besitzende Person möglich.

- **Math**: Dieses Design Pattern wird genutzt, um die korrekte Ausführung kritischer Operationen wie bspw. die Veränderung des Guthabens zu bewahren. Häufig werden sie, wie in [Ria18] beschrieben, auch dazu eingesetzt, um Under-bzw. Overflows frühzeitig zu erkennen.

- **Fork-Check**: Aufgrund der Tatsache, dass die Blockchain Ethereum seit dem DAO-Hack schon mehrmals geforkt wurde, implementieren manche Contracts Code, um zu überprüfen, ob sie sich auf einer Fork Chain befinden. In manchen Fällen führen die Smart Contracts unterschiedliche Aktionen aus, abhängig davon auf welcher Ethereum Blockchain sie sich befinden.

4.1.3 Anforderungen anhand der Sprachentwicklung

In Ihrem Paper [CAMS18] argumentieren Coblenz et al. für eine interdisziplinäre Herangehensweise bei der Entwicklung von Programmiersprachen, welche sowohl traditionelle als auch anwenderorientierte Methoden vereint.

4.1.3.1 Traditionelle Ziele

- **Korrektheit**: Hat ein bestimmtes Programm bestimmte wünschenswerte Eigenschaften? Sprachen unterstützen die Korrektheit meist durch formale Ansätze, wie Typsysteme und Beweissysteme, um diese Frage beantworten zu können.

- **Performance**: Wie performant ist der ausgeführte Code? Die Ausführung von Code lässt sich auf der entsprechend eingesetzten Hardware gut messen und vergleichen.

- **Ausdrucksstärke**: Wie gut oder leicht lässt sich die Absicht des Entwickelnden explizit in der Sprache ausdrücken? Die Ausdrucksstärke einer Sprache hat auch Auswirkungen auf die Benutzerfreundlichkeit einer Sprache, da Anwendende gezwungen werden gewisse Formalismen, wie Typdefinitionen, zu einem bestimmten Zeitpunkt festzulegen, obwohl das zu diesem Zeitpunkt noch gar nicht gewünscht wird.

- **Übersetzungsgeschwindigkeit**: Wie lange dauert das Übersetzten meines Programms in ausführbaren Code? Auch wenn diese Eigenschaft aus den Zeiten von nicht so leistungsfähiger Computern stammt, ist sie heutzutage noch immer wichtig, da Programme immer größer und komplexer werden.

4.1.3.2 Anwenderorientierte Ziele

- **Verständlichkeit**: Wie verständlich ist der Programmcode? Es ist wichtig, dass Entwicklende, zu einem gelesenen Programmcode, Fragen schnell und korrekt beantworten können.
- **Einfache Schlussfolgerungen**: Wie einfach können Entwickelnde Fragen zu bestimmten Eigenschaften des Programms beantworten? Die anwenderorientierte Analogie zur Korrektheit ist eine wichtige Eigenschaft in Sprachen, vor allem wenn diese die traditionelle maschinenunterstützte Eigenschaft der Korrektheit nicht vollständig besitzen.
- **Modifizierbarkeit**: Wie leicht können bestimmte Änderungen an Programmen durchgeführt werden? Da viel Zeit in die Wartung und Weiterentwicklung von bestehenden Programmen investiert wird, ist es wichtig, dass Änderungen einerseits leicht möglich sind und andererseits die Auswirkungen dieser Änderungen für Entwicklende nachvollziehbar bleiben.
- **Erlernbarkeit**: Wie einfach können Entwickelnde die Sprache erlernen? Diese Eigenschaft ist ist der Praxis wichtig, damit die Sprache überhaupt angenommen und angewendet wird.

4.1.4 Anforderungen anhand gängiger Smart-Contract-Sprachen

In diesem Kapitel werden aktuelle Sprachen, die für Smart-Contracts eingesetzt werden, betrachtet und aus den Features und Eigenschaften, die diese Sprachen besitzen, werden scheinbar notwendige Anforderungen an Smart-Contract-Sprachen extrahiert. Als Einstiegspunkt wurde die Sammlung von Sergei Tikhomirov [st19] benutzt und primär Sprachen aus dem Ethereum Ökosystem betrachtet.

4.1.4.1 Ethereum Bytecode

Ethereum Bytecode[4] ist die Sprache, in die Ethereum Smart Contracts normalerweise übersetzt werden, um auf der EVM ausgeführt zu werden. EVM Bytecode ist eine Assembler-ähnliche Sprache, die gegeben durch den stack-basierten Aufbau der EVM, zu großen Teilen aus üblichen Stackinstruktionen [Woo14] aufgebaut ist:

- Stopp und arithmetische Operationen: `STOP`, `ADD`, ..., `SIGNEXTEND`
- Vergleich und Bitweise logische Operationen: `LT`, `GT`, ..., `BYTE`
- Stack-, Speicher-, Storage- und Flow-Operationen: `POP`, `MLOAD`, ..., `JUMPDEST`
- Push-Operationen: `PUSH1`, `PUSH2`, ..., `PUSH32`

[4] https://ethervm.io/

Blockchain	Smart-Contract-Sprache
Ethereum	Ethereum bytecode
	Solidity
	Vyper
	eWASM
	Idirs
	Flint
	Formality
	Huff
	Lira
Tezos	Michelson
	Liquidity
	fi
	LIGO
Libra	Move
Waves	RIDE
Kadena	Obsidian
	Pact

Tabelle 4.1: Übersicht der betrachteten Smart-Contract-Sprachen

- Duplikations-Operationen: `DUP1`, `DUP2`, ..., `DUP16`
- Austausch-Operationen: `SWAP1`, `SWAP2`, ..., `SWAP16`
- Logging-Operationen: `LOG0`, `LOG1`, ..., `LOG4`

Diese werden erweitert um spezielle Instruktionen, die unerlässlich sind für die korrekte Ausführung von Smart Contracts auf der EVM:

- SHA3: `SHA3` zur Berechnung des Keccak-256 Hash
- Contract Umgebungsinformationen: `ADDRESS`, `BALANCE`, ..., `RETURNDATACOPY`
- Block Umgebungsinformationen: `BKOCKHASH`, `COOINBASE`, ..., `GASLIMIT`
- System-Operationen: `CREATE`, `CALL`, ..., `SELFDESTRUCT`

4.1.4.2 Solidity

Solidity [Sol19] ist eine objektorientierte, statisch typisierte Programmiersprache, die sich syntaktisch an die ECMAScript Standards orientiert. Sie wurde speziell zur Entwicklung

von Smart Contracts auf der Ethereum Blockchain entwickelt, besitzt jedoch auch klassische Eigenschaften objektorientierter Sprachen.

Kompiliert wird die Sprache in Ethereum Bytecode, der dann wiederum auf die Ethereum Blockchain geladen und dort ausgeführt wird.

Da die Sprache von ehemaligen Mitarbeitern des Ethereum Projektes speziell für diese Blockchain entwickelt wurde, ist sie die meistgenutzte Sprache für Smart Contracts auf der EVM, kommt jedoch auch in vielen anderen Blockchainprojekten (wie zum Beispiel Cardano[5] oder Æternity[6] [Tok19]) zum Einsatz.

Besondere Features:

- **Speziell für Smart Contracts entwickelt**: Die gesamte Logik sowie die Datentypen wurden explizit für Smart Contracts entworfen, womit die Sprache stark spezialisiert ist.
- **Function Modifiers**: Modifier können verwendet werden, um das Verhalten von Funktionen auf deklarative Weise zu verändern (siehe Listing 4.1).
- **Vererbung von Smart Contracts**: Solidity unterstützt Mehrfachvererbung einschließlich Polymorphismus. Mittels `virtual` und `override` werden zum Beispiel vererbte Funktionen überladen.
- **Keine undefinierten Variablen**: In Solidity wurde auf undefinierte oder Null-Werte verzichtet, weshalb jeder Variable, abhängig von ihrem Typ, bei der Deklaration ein Standardwert zugewiesen wird.
- **Komplexe benutzerdefinierte Typen**: Anwendende können neue komplexe Typen wie z.B. Vertragstypen, Enums, Mappings, Structs oder Array-Typen erzeugen.
- **Libraries**: Bibliotheken werden einmal deployed und stellen damit Funktionen bereit, die andere Contracts aufrufen können. Sie werden mittel `library` erzeugt, sind hauptsächlich zur Wiederverwendung von Funktionen gedacht und können im Vergleich zu `contract`-Objekten keine Zustandvariablen besitzen, erben oder vererbt werden, Ether empfangen oder zerstört werden. *SafeMath*[7] ist zum Beispiel eine weit verbreitete Bibliothek für sichere mathematische Operation.

[5] `https://www.cardano.org/`
[6] `https://aeternity.com/`
[7] `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol`

```
pragma solidity ^0.5.0;

contract A {
    address private _owner;

    modifier onlyOwner() {
        require(msg.sender == _owner);
        _;
    }

    function doSomethingOnlyAsOwner() public onlyOwner {
        ...
    }
}
```

Listing 4.1: Function Modifier in Solidity

4.1.4.3 Vyper

Vyper [Vyp19] ist eine stark typisierte Universalsprache, die, wie Solidity, auf EVM Bytecode übersetzt wird. Das Ziel dieser Programmiersprache ist eine möglichst weitgehende Simplifizierung des Prozesses, um schneller Smart Contracts schreiben zu können, die aufgrund vereinfachter Lesbarkeit zu größerer Transparenz beitragen sollen. Dadurch sollen besonders wenig Angriffsvektoren ermöglicht werden.

Die Sprache fällt dadurch auf, dass sie syntaktisch stark an Python und logisch an Solidity angelehnt ist. Die Ziele der Sprache lassen sich auf drei Grundprinzipien zusammenfassen:

- **Maximale Sicherheit**: Mit dem Wegfall von einigen Prinzipien der Objektorientierten Programmierung, wie beispielsweise Klassenvererbung, soll ein erhöhtes Level an Sicherheit erreicht werden.
- **Maximale Lesbarkeit**: Die Sprache soll möglichst geringe Barrieren für Entwickelnde oder Interessierte mit weniger Programmiererfahrung darstellen. Zudem soll die Sprache möglichst wenig Spielraum bieten, um falschen Code zu schreiben. Die Einfachheit für Lesende steht hierbei über der Einfachheit für Entwickelnde.
- **Größtmögliche Einfachheit für Sprache und Compiler**: Aufbauend auf die maximale Lesbarkeit wurde auch darauf geachtet, dass Sprache und Compiler einfach bleiben um die Entwicklung in Vyper zu vereinfachen.

Besondere Features:

- **Bounds & Overflow Checking**: Sowohl bei Array-Zugriffen als auch arithmetischen Operationen werden Grenzen und Überläufe geprüft.
- **Signed Integers & Decimal Fixed Point Numbers**: Während dezimale Festkommazahlen in Solidity nicht vorgesehen sind, werden diese in Vyper ermöglicht.

- **Decidability**: In Vyper soll es möglich sein, eine genaue Obergrenze für den *Gas*-Verbrauch eines beliebigen Funktionsaufrufes zu berechnen.
- **Strong Typing**: Unterstützung von Einheiten (zum Beispiel `seconds`, `wei`, `wei per second`, `meters per second squared` oder `timestamp`).
- **Small and understandable compiler code**
- **Limited support for pure functions**: Alles, was als `constant` markiert wird, darf den Zustand nicht verändern.
- **Keine Modifier oder Klassenvererbung**: Modifier und Klassenvererbung wurden zum Zweck der vereinfachten Lesbarkeit gänzlich weggelassen. Hiermit soll verhindert werden, dass bei der Analyse des Codes, ein mehrfaches hin und herspringen innerhalb verschiedener Klassen notwendig ist, um den Code zu verstehen.
- **Kein Inline-Assembly**: Um mittels Suche nach dem Variablennamen das Auffinden aller Variablen-Zugriffe und -Modifikationen zu gewährleisten, wird in Vyper, anders als in Solidity, Inline-Assembly nicht unterstützt.
- **Kein Überladen von Funktionen und Operatoren**: Das Überladen von Funktionen und Operatoren erschwert das Lesen von Code, da immer geprüft werden muss, welche konkrete Funktion zu welchem Zeitpunkt ausgeführt wird. Da somit leicht irreführender Code geschrieben werden kann, wir auch darauf in Vyper verzichtet.
- **Keine Rekursion oder unendliche Schleifen**: Da es aufgrund von rekursiven Aufrufen und unendlichen Schleifen nicht möglich ist eine obere Schranke für Gas-Limits zu setzen, eröffnet dies die Möglichkeit von potentiellen Gas-Limit-Attacken. Basierend auf dem *Entscheidbarkeits-Prinzip* soll es zu jedem Zeitpunkt möglich sein, die obere Schranke eines jeden Funktionsaufrufs präzise zu berechnen.
- **Kein Binary Fix Point**: Da bei binäre Festkommawerte oft Annäherungen benötigt werden, werden diese zu Gunsten von dezimalen Festkommazahlen in Vyper nicht mehr unterstützt.

4.1.4.4 Ethereum WebAssembly

Ethereum WebAssembly (ewasm) befindet sich aktuell in aktiver Entwicklung und kann als deterministische Untermenge von WebAssembly (Wasm)[8] gesehen werden. Wasm ist ein binäres Befehlsformat für eine stack-basierte virtuelle Maschine.

Die Verwendung von WebAssembly als Format für Smart Contracts soll folgende Vorteile bringen:

[8] `https://webassembly.org/`

- Nahezu native Ausführungsgeschwindigkeit für Smart Contracts
- Die Möglichkeit Smart Contracts in vielen traditionellen Programmiersprachen, wie C, C++ und Rust, zu entwickeln
- Zugang zu einer großen Community für Entwickelnde und den Toolchains rund um WebAssembly

4.1.4.5 Idris

Idris [Il19] ist eine von Haskell und ML inspirierte rein funktionale General Purpose Language (GPL) mit *dependent types*. Idris kombiniert Features von bekannten funktionalen Sprachen mit denen der maschinengestützten Beweise, ist aber als GPL konzipiert. Aufgrund dieser und weiterer Eigenschaften wurde die Sprache als mögliche Smart Contract Sprache von Patterson und Edström [PE16] vorgeschlagen um präzise Ausführung von Code in Smart Contracts zu garantieren.

Besondere Features:

- **Polymorphic und Dependent Types**: Ein *abhängiger Typ* kann als eine Funktion, die Typen liefert, verstanden werden [PH06]. Ist das Argument der Funktion ein Typ, spricht man von polymorphen (abhängigen) Typen und ist das Argument ein Wert, so spricht man von abhängigen Typen. Durch abhängige Typen ist es möglich, einzelnen Werten einen Typ zuzuordnen, was genutzt werden kann, um das Verhalten von Programmen genauer zu spezifizieren und wodurch wiederum eine große Menge von Fehlern zum Übersetzungszeitpunkt abfangen kann.
- **Foreign Function Interface (FFI)**: Idris ermöglicht den leichten Zugriff auf Routinen von anderen Programmiersprachen, zumindest für C.
- **Compiler-unterstütztes interaktives editieren**: Der Compiler unterstützt einen beim Schreiben von Code durch Vorschlagen von Typen.
- **Eager Evaluation**: Eager oder auch strikte Evaluierung von Ausdrücken, direkt zu dem Zeitpunkt, wenn der Ausdruck an eine Variable gebunden wird, soll die Nachvollziehbarkeit von Ausführungsabläufen erleichtern.
- **Typen als First-Class-Citizen**: Typen können somit analog zu Werten und Funktionen verwendet werden. In Idirs können damit Funktionen geschrieben werden, die Typen als Argument und Rückgabewert enthalten können.
- **Do-Notation und Idiom Brackets**: Um Seiteneffekte beschreiben zu können, nutzt Idirs do-Notationen, um im Stil von imperativen Sprachen mehrere aufeinander folgende Operationen zusammenfassen zu können. Idiom Klammern $[\![f\, u_1 ... u_n]\!]$, welche von McBride und Patersin eingeführt wurden [MP08], ermöglichen eine

simplere Schreibform und somit leserlicheren und verständlicheren Code, für die Anwendung einer reinen Funktion auf Argumente mit Seiteneffekten (auch *Applicative Funktor* genannt).

- **State Machines**: Idris nutzt die `Control.ST` Bibliothek um Zustände zu verwalten und Zustandsänderungen in Funktionstypen mitzuverfolgen. Die `ST` Bibliothek ermöglicht es zudem Programme mit mehreren gegebenenfalls auch zusammenhängenden Zustandsübergangssystemen (engl. *state transition system*) zu schreiben. Dieser zustandsorientierte Ansatz wird auch von Brady in [Bra16] näher beschrieben.

- **Effects**: Idirs nutzt die von Brady [Bra13] eingeführte Embedded Domain Specific Language (EDSL) *Effect*, um als Verbesserung gegenüber von Monaden viele zusammenhängende kleine Seiteneffekte, auch *algebraischen Effekten* genannt, beschreiben und verwalteten zu können.

4.1.4.6 Flint

Flint [Fli19] ist eine statisch typisierte, Contract-orientierte Programmiersprache, die speziell für das Schreiben von robusten Smart Contracts auf der Ethereum Blockchain ausgelegt ist und die sich syntaktisch von Swift[9] inspiriert wurde. Von seinem Macher, Franklin Schrans, im Rahmen einer Diplomarbeit erstellt [Sch18], wurde Flint mit dem Ziel konzipiert, die Schwächen von Solidity durch besondere Sicherheits-Features auszugleichen und somit sicheren und vorhersagbaren Programmcode herzustellen.

Gemäß Schrans et al. [SED18] erlaubt Solidity eine Reihe unsicherer Programmiermuster, die zu schwer analysierbarem Code und im weiteren Verlauf aufgrund der Unveränderlichkeit von Smart Contracts zu finanziellen Schäden führen können. Bei der Entwicklung von Flint wurde genau bei diesen bekannten Schwachstellen in Solidity angesetzt und zudem namhafte Hacks der letzten Jahre als weitere Inspiration für die Entwicklung von Features herangezogen.

Besondere Features:

- **Caller Protections**: Fordern die Entwicklenden auf zu bestimmen, wer in der Lage sein darf gewisse kritische Funktionen innerhalb eines Contracts aufzurufen. Diese Schutzmaßnahmen werden für interne Aufrufe statisch und für externe zur Laufzeit geprüft.

- **Type States**: Type States eines Smart Contract sind alle möglichen Zustände, die durch diesen eingenommen werden können. Dadurch wird beispielsweise ermöglicht, dass bestimmte Funktionen nur in einem bestimmten Zustand aufgerufen werden können.

[9] `https://developer.apple.com/swift/`

- **Immutability by default**: Wird ein Zustand in einem Contract verändert, muss die entsprechende Funktion mit dem Keyword `mutates (...)` und einer Liste von den betroffenen Variablen versehen werden.

- **Asset Types**: Um sicherzustellen, dass ein Contract Zustand zu jedem Zeitpunkt den korrekten Wert an Ether darstellt, wurde der besondere *Asset* Typ eingeführt. Dieser erlaubt ein stark eingeschränktes Set an Operationen, um Angriffe wie Double-Spending und Re-Entrancy zu verhindern.

- **Protection Blocks**: Aufbauend auf Asset Types, durch welche Funktionsaufrufe eingeschränkt werden, kann durch die Definition von sogenannten `caller groups` von vornherein definiert werden, wer eine Funktion aufrufen darf. Der Protection Block kann ebenfalls vom Type State abhängig sein.

- **Finite Loops**: Die einzigen Schleifen, die Flint enthält, sind `for-in`-Schleifen zur Iteration von Arrays, Dictionaries und Ranges.

- **Initialisers**: Alle Contracts und Structs müssen public Initialisierer definiert haben, die auch alle zugehörigen Zustände definieren.

- **Limited Fallback Functions**: Fallbackfunktionen können den Zustand nicht verändern und die standardmäßige Fallbackfunktion führen einen Rollback auf den Contract aus.

- **Safe Arithmetic**: Integer Overflows führen zu einer Exception und die Ausführung des Contract wird abgebrochen.

4.1.4.7 Formality

Formality [moo19] ist eine für die Ethereum Blockchain, aktuell noch in Entwicklung befindliche funktionale Smart-Contract-Sprache und Beweissprache, die somit formale Beweise unterstützen soll und zugleich einfach, benutzerfreundlich und effizient sein soll.

Besondere Features:

- **Accessible Syntax**: Im Vergleich zu anderen Beweissprachen, wie Coq oder Agda, die eine sehr komplexe Syntax besitzen, hat es sich Formality zum Ziel gemacht eine einfache und vertraute Syntax einzusetzen. Die Syntax erinnert stark an Python oder JavaScript.

- **Fast and portable "by design"**: Formality soll so schnell wie theoretisch möglich werden. Die Sprache kommt ohne Garbage-Collection aus und ist dafür ausgelegt in enormen parallelen Architekturen evaluiert zu werden. Zudem wird die Sprache

Lazy evaluiert, besitzt ein klares Kostenmodell für die Blockchain, hat eine kompakte Laufzeitumgebung von etwa 600 Zeilen Code[10] und lässt sich sehr leicht auf unterschiedlichste Plattformen portieren.

- **Elegant underlying Type Theory**: Anstelle von komplexen eingebauten Datentypen, verwendet Formality sogenannte *Self Types*[FS14], die den Einsatz von induktiven Typen mit nativen Lambdas ermöglichen. Das ist einer der Gründe, die eine viel einfachere und kompaktere Implementierung der zugrunde liegenden Typtheorie ermöglichen.
- **Optimal high-order evaluator**: Der Substitutions-Algorithmus von Formality ist *asymptotisch performanter* als viele andere Closure-Funktion-Implementierungen. Dies erlaubt eine sehr schnelle Evaluierung von Programmen höherer Ordnung.
- **Guaranteed termination**: Alle in Formality geschriebenen Programme terminieren. Turing-Vollständigkeit kann dennoch explizit durch Korekursion, also einer Funktion mit iterativen Befehlen, die unendlich oft ausgeführt wird, abgebildet werden.

4.1.4.8 Huff

Huff [Azt19] ist eine Domain Specific Language (DSL), die für das Schreiben von stark optimierten EVM Programmcode entwickelt wurde und somit auch für die Entwicklung von Ethereum Smart Contrats eingesetzt werden kann. Huff ermöglicht den direkten Einsatz von EVM Bytecode-Blöcken ohne weitere Abstraktionen.

Besondere Features:

- **Macros**: Makros werden mittels zwei optionalen Parametern definiert. `takes`, die Anzahl an erwarteten Elementen am Stack und `returns`, die Anzahl an Elementen, die das Makro am Stack hinterlässt.
- **Jump Tables**: Huff ermöglicht Definitionen von Tabellen mit Sprungadressen direkt im Contract Bytecode. Dies erlaubt effiziente Ausführungsabläufe von Programmen durch Sprünge, anstatt durch bedingte Verzweigungen (If-Else-Then-Ausdrücke).
- **Syntactic Sugar**: Für die drei der am häufigsten verwendeten Instruktionen (die Größe eines Makros bzw. einer Tabelle auf den Stack pushen und den Offset einer Tabelle in Bezug zum Anfang des Contract Bytecodes auf den Stack pushen), gibt es vordefinierte Kurzschreibweisen, die verwendet werden können.
- **No Push Opcodes**: Literale können dezimal oder hexadezimal direkt in Huff genutzt werden und werden vom Compiler implizit durch die kleinst mögliche Push-Instruktion ersetzt.

[10] `https://github.com/moonad/Formality/blob/master/src/fm-net.ts`

4.1.4.9 Lira

Lira [eTo20] ist eine intuitive, leicht zu lernende, deklarative DSL, die für das Definieren von hoch komplexen Smart Contracts auf der Ethereum Blockchain von eToroX [11] für den Finanzbereich entwickelt wurde.

Besondere Features:

- **Secure Foundation**: Lira ist eine formal verifizierte Contract-Sprache, die auf der vorgestellten Lösung von Egelund-Müller et al. [EMEHR17] basiert.
- **Non-Turing complete**: Keine Turing-Vollständigkeit ist eine bewusste Designentscheidung, die die Sprache zwar in einem sehr kleinen Funktionsumfang einschränkt, die Semantik dieses Funktionsumfangs aber formal verifizieren kann.
- **Kompakte Sprachdefinition**: Contracts sind hauptsächlich zusammengesetzt durch die Kombination der `transfer`-Funktion, zum Transferieren von einer Tokeneinheit zwischen zwei Adressen, der `scale`-Funktion, zum Skalieren auf beliebige Tokenmengen und der `taranslate`-Funktion, zur zeitverzögerten Ausführung eines Contracts. Zudem gehören Ausdrücke zur bedingten Verzweigung, Zeiteinheiten sowie Operatoren zu den Bestandteilen der Sprache.

4.1.4.10 Michelson

Die Tezos Plattform[12] ist eine Blockchain, die sich als direkter Konkurrent zu Ethereum positioniert. Während Tezos der Meinung ist, dass Ethereum versucht ein Weltcomputer zu sein [Ml17b], hat sich Tezos bei der Konzeptionierung der eigenen Smart Contracts auf einige wenige Use-Cases, meist finanzieller Natur, beschränkt.

Zu diesem Zweck wurde Michelson[13] [Tez19] entwickelt, eine stark typisierte, stackbasierte DSL, die genutzt wird, um Smart Contracts auf der Tezos Blockchain zu erstellen und ohne Compiler direkt auf dieser auszuführen. Im Gegensatz zum Bytecode der EVM wurde bei Michelson, welches als Pendant dazu gesehen werden kann, auf Lesbarkeit gesetzt und dadurch weniger Ähnlichkeit zu Assembler-Syntax gewählt. Daher erinnert Michelson syntaktisch mehr an Sprachen wie Forth oder Lisp.

Das erklärte Ziel bei der Entwicklung war, dass selbst der von High-Level-Sprachen nach Michelson übersetzte Output eines Compilers lesbar und verständlich sein soll. Dadurch soll ein sicheres Arbeiten ermöglicht werden, da der Code nicht mehr übersetzt werden muss und somit dessen Ausführung genauer abschätzbar wird. Die Sprache wurde zudem durch einige unübliche Features wie Funktionen höherer Ordnung und Datenstrukturen wie `List` und `Map` ergänzt.

[11] `https://www.etorox.com/`
[12] `https://tezos.com/`
[13] `https://www.michelson.org/`

Michelson Befehlssatz kann wie folgt grob zusammen gefasst werden:

- Kontrollstrukturen: `FAILWITH` für den expliziten Abbruch, `{ I ; C }` Sequenzen, `IF bt bf` bedingte Verzweigungen, `LOOP body` und andere Arten von Schleifen, `EXEC` zum Ausführen von Funktionen vom Stack, ...
- Stack Operationen: `DROP`, `DUP`, `SWAP`, `PUSH`, ...
- Vergleichsinstruktionen: `EQ`, `NEQ`, `LT`, `GT`, `LE` oder `GE`
- Boolesche Operationen: `OR`, `AND`, `XOR` oder `NOT`
- Arithmetische Operationen: `NEG`, `ABS`, `ADD`, `DIV`, ...
- String Operationen: `CONCAT`, `SIZE`, `SLICE`, `COMPARE`
- Operationen für die speziellen Datentypen wie Sets, Maps oder Listen.
- Domänenspezifische Operationen
  - Operationen für domänenspezifische Datentypen wie `timestamp`, `address`, `signature` oder `chain_id`.
  - Operationen für den Umgang mit Contracts: `CREATE_CONTRACT`, `BALANCE`, `TRANSFER_TOKENS`, ...
  - Spezielle Operationen: `NOW` um den Timestamp des Block auf den Stack zu legen und `CHAIN_ID` um die Blockchain-Id am Stack abzulegen
  - Kryptographische Operationen: `HASH_KEY`, `BLAKE2B`, `KECCAK`, `SHA256`, `SHA512`, `SHA3`, `CHECK_SIGNATATURE` oder `COMPARE`

Besondere Features:

- **Primitive und High-Level-Datentypen**: Neben primitiven Datentypen wie `string`, `int` oder `bytes`, besitzt Michelson auch komplexere Datentypen wie `list (t)` für Listen, `pair (l) (r)` für Paare, `optional (t)` für optionale Werte, `set (t)` für Sets von Werten oder `map (k) (v)` Maps von Werten. Diese Datentypen sollen das Schreiben und Lesen von Michelson Programmen oder Programmen, die auf Michelson übersetzt werden, erleichtern.
- **Typsystem**: Das Typsystem von Michelson ist *korrekt*, sprich fehlerhafte Programme werden vom Typsystem erkannt. Ein wohltypisiertes Michelson Programm mit wohltypisierten Eingaben wird vom Michelson Interpreter nie in einem unerwarteten oder endlosen Zustand überführt und verhält sich immer wie erwartet. Im Gegenzug ist das Typsystem aber auch unvollständig, es kann also durchaus ein fehlerfreies Programm vom Typsystem abgelehnt werden, da die Typprüfung in gewissen komplexen Situationen Typfehler nicht ausschließen kann.

- **Bytecode der selbst geschrieben werden kann**: Die Sprache zielt darauf ab, trotz ihrer Eigenschaft nicht weiter übersetzt werden zu müssen, leicht erlernbar zu sein. Daher wurden Opcodes durch benannte Operationen wie `ADD` oder `NOT` ersetzt. Trotzdem gibt es bereits High-Level-Programmiersprachen, die nach Michelson kompiliert werden.
- **Formale Verifikation**: Es existieren bereits Formalisierungen der Michelson Syntax und Semantik in Coq [BCH+19], die für die formale Verifikation von Michelson Smart Contracts genutzt werden kann.
- **Monomorphie**: Das Typsystem von Michelson enthält und erlaubt keine Art von Polymorphismus, da der Algorithmus zur Typprüfung in einem einmaligen Durchlauf erfolgt und nicht mit Polymorphie umgehen kann.

4.1.4.11 Liquidity

Liquidity [OCa19] ist eine für die Tezos Blockchain und das Dune Network[14] entwickelte, statisch typisierte, rein funktionale High-Level-Sprache, die die Syntax von *OCaml*[15] und *ReasonML*[16] nutzt und auf Michelson übersetzt wird. Sie wurde speziell für Michelson entwickelt und enthält dementsprechend ähnliche Sicherheitsfeatures. Das Ziel bei der Entwicklung von Liquidity war es eine High-Level-Sprache zu kreieren, welche die von Michelson vorgegebenen Sicherheitsrichtlinien erfüllen kann.

Während in Michelson noch direkte Stack-Operationen ausgeführt werden müssen, wird dies in Liquidity durch lokale Variablen ersetzt, um das Schreiben von Smart Contracts zu vereinfachen. Durch die besonders nahe, an Michelson orientierte Entwicklung der Sprache ist es möglich sowohl in Michelson-Code zu kompilieren als auch durch den mitgelieferten Decompiler bestehenden Michelson-Code zu dekompilieren.

Besondere Features:

- **Abdeckung der Michelson Programmiersprache**: Alles was in Liquidity geschrieben werden kann, kann auch in Michelson geschrieben werden.
- **Verschiedene Syntax-Systeme**: Sowohl OCaml Syntax als auch JavaScript-ähnliche ReasonML Syntax werden von Liquidity unterstützt.
- **High-Level-Typen**: Beispielsweise `sum`-oder `record`-Typen können selbst definiert und genutzt werden.
- **Lokale Variablen anstatt Stackmanipulation**: Werte können in lokalen Variablen gespeichert werden.

[14] `https://dune.network/`
[15] `https://ocaml.org/`
[16] `https://reasonml.github.io/`

- **Polymorphimus**: Der eingesetzte Typinfernez-Algorithmus ermöglicht auch Polymorphismus. Im Allgemeinen wird die monomorphe Eigeschaft für Werte aus Michelson übernommen, doch es können polymorphe Funktionen geschrieben werden, welche bei der Übersetzung nach Michelson in mehrere monomorphe Funktionen tranfsormiert werden. Zudem erlaubt Liquidity parametrisierte benutzerdefinierte Typen.
- **Michelson Dekompiler**: Alle Michelson Programme können in leserliche Liquidity Programme übersetzt werden.
- **Liquidity Complier**: Der eigene Compiler soll effizient arbeiten und gute Optimierungen durchführen können.
- **Modularisierung**: Ein Modul- und Contract-System ermöglicht das Schreiben von wiederverwendbaren Code und Bibliotheken

4.1.4.12 fi

Eine weitere Smart-Contract-Programmiersprache für die Tezos Platform ist *fi*[17]. *fi* [fC20] ist eine statisch typisierte, High-Level-Sprache, deren Syntax an ECMAScript, JavaScript und Solidity erinnert. Zudem erlaubt fi eine objektorientierte Programmierung und der eigene Compiler übersetzt den Code direkt in wohltypisierten und formal verifizierbaren Michelson Code.

Besondere Features:

- **Einfache Basisstruktur**: Contract Definitionen bestehen aus *Constants*, *Storage Variables*, *Structs* und *Contract Entry Points*. Konstanten können, nachdem sie definiert wurden, im gesamten Contract verwendet werden. Speichervariablen werden permanent auf der Blockchain abgelegt und entsprechen dem Zustand des Contracts. Mit Structs können eigene komplexe Datentypen erzeugt werden und Entry Points sind öffentliche Methoden, die der Smart Contract zum Aufrufen anbietet.
- **Lokale Variablen**: Neben primitiven und komplexen Datentypen besitzt fi auch lokale, beziehungsweise temporäre Variablen. Diese Variablen können definiert werden mittelt `let <type|struct> <variable name> = <value>;` und ab da an über ihren Namen verwendet werden.
- **Strikte Funktionen**: Variablen dürfen nur über Funktionen modifiziert werden, es muss zum Beispiel eine `add`-Funktion verwendet werden, anstatt zwei Werte einfach per +-Operator zu addieren. Zudem handelt es sich bei fi Funktionen nur um strikte Funktionen, sobald ein Argument der Funktion undefiniert ist, ist auch das Ergebnis undefiniert.

[17] `https://fi-code.com/`

- **Funktion für Typumwandlung**: fi stellt Funktionen zur Verfügung, um die gängigsten Typen in andere umzuwandeln. Zum Beispiel kann `to_address` genutzt werden, um gewisse Typen, wie einen Contract, in eine Adresse umzuwandeln.

- **Variablen Modifier**: Um die Lesbarkeit von Code zu verbessern, wurden Variablen Modifier `<variable>.<modifier>(<arguments>);` eingeführt. Diese ermöglichen, über direkte Zugriffe, eine schnellere Kodierung von Variablenmodifikationen.

- **Globale Konstanten**: Einige globale Konstante, wie zum Beispiel `BALANCE` oder `SENDER` stehen in jedem Smart Contract zur Verfügung, um Blockchain-spezifische Informationen bereitzustellen.

4.1.4.13 LIGO

LIGO [Lig20] ist eine einfache, imperative Smart-Contract-Sprache, die durch den eigenen Compiler in Michelson-Code übersetz werden kann und somit ebenfalls auf der Tezos Plattform eingesetzt werden kann. Die statisch typisierte Sprache wurde primär dazu entworfen größere und komplexere Smart Contracts zu schreiben, was in Michelson potentiell größeren Aufwand verursachen würde. Darüber hinaus unterstützt sie verschiedene Syntax-Systeme, die als PascaLIGO (Pascal-ähnliche Syntax), CameLIGO (Caml-ähnliche Syntax) und ReasonLIGO (Reason-ähnliche Syntax) bezeichnet werden.

Besondere Features:

- **Mehrsprachigkeit**: LIGO kann bereits jetzt in drei syntaktischen Variationen geschrieben werden, die direkt in Michelson-Code kompiliert werden. Zukünftig sind weitere Sprachen geplant, um ein breiteres Feld an Entwickelnden anzusprechen. So ist explizit das bereits erwähnte Yul in Planung, um auch in Solidity geschriebene Smart Contracts auf der Tezos Blockchain ausführen zu können [Alf19].

- **Starke Typisierung**: Indem zuerst Typen definiert werden, dann Code geschrieben wird, können durch dieses Typsystem ein Großteil der Fehler im Code abgefangen werden.

- **Leichte Integration**: LIGO kann als NodeJS Bibliothek eingebunden und genutzt werden.

- **Modularität**: Sehr komplexte und große Smart Contracts können auf mehrere Files verteilt werden und mittels `#include` eingebunden werden.

4.1.4.14 Move

Move [Lib19] ist eine von der Facebook Inc.[18] entwickelte Programmiersprache, die auf der Libra Blockchain[19] eingesetzt werden soll. Innerhalb des Libra-Ökosystems wurden Smart Contracts durch sogenannte Modules ersetzt, die auf ähnliche Art und Weise funktionieren. Die statisch typisierte Programmiersprache Move ist als Bytecode mit High-Level-Syntax konzipiert und geht über das reine Schreiben von Smart Contracts hinaus, da sie auch zur Implementierung von sogenannten `custom transactions` eingesetzt werden kann.

Verbesserungen, die von Move versucht wurden umzusetzen, beziehen sich im Whitepaper [BCD+19] vor allem auf Solidity. Hier wird die indirekte Repräsentation von Gütern erwähnt. Im Gegenteil zu Solidity welches Ether nur als `int` Werte darstellt, ist es in Move möglich die gewählte Währung mittels Ressourcentypen direkt darzustellen. Dank linearer Logik ist es nicht möglich ein vom Typ `resource` deklariertes Objekt zu kopieren oder implizit zu verwerfen, sondern lediglich zwischen Speicherorten zu verschieben.

Besondere Features:

- **First-Class-Resourcen**: Move erlaubt es Entwickelnden eigene `resource` Typen zu definieren. Libra Coin[20], die Währung der Libra Blockchain, ist ein solcher Typ. Die bereits erwähnten Sicherheitseigenschaften, die durch die auf lineare Logik basierende Semantik entstehen, werden durch das Typsystem der Sprache statisch geprüft und sichergestellt. Zugriffe und vor allem das Erzeugen und Löschen von solchen First-Class-Ressourcen erfolgt in Move durch *Module*.

  Module sind eine Art von Smart Contracts in Move, die zur Datenabstraktion verwendet werden. Sie definieren Ressourcentypen und Operationen, um diese zu Erzeugen, Löschen und Bearbeiten. Module können auch Funktionen und Typen anderer Module aufrufen.

- **Flexibilität**: Alle Transaktionen auf der Libra Blockchain beinhalten ein *move transaction script* welches die Logik (beispielsweise eine Transaktion von Libra zwischen zwei Teilnehmenden) der Transaktion beschreibt. Dieses `transaction script` kann exakt ein Mal genutzt werden und kann nicht von anderen `transaction scripts` aufgerufen werden.

- **Sicherheit**: Move verfolgt einen hybriden Ansatz bei der Überprüfung von Programmen bezüglich Korrektheit und Sicherheit von Ressourcen, Typen und Speicher. Anstatt einer typisierten High-Level-Sprache, deren Compiler diese Eigenschaften prüft, oder einer untypisierten Low-Level-Assemblersprache, wo diese Prüfung zur Laufzeit erfolgt, nutzt Move typisierten Bytecode. Dieser Bytecode wird auf

[18] `https://about.fb.com/`
[19] `https://libra.org/`
[20] `https://libra-coin.cc/`

der Blockchain durch einen *Bytecode Verifier* zuerst geprüft und dann durch den *Bytpecode Interpreter* ausgeführt.

- **Verifizierbarkeit**: In Hinsicht auf die Berechungskomplexität ist es nicht praktikabel alle Sicherheitseigenschaften von Programmen auf der Blockchain durch den Bytecode Verifier prüfen zu lassen. Move versucht natürlich soviel wie möglich direkt auf der Blockchain zu prüfen, unterstützt aber durch explizite Designentscheidungen die statische Codeanalyse abseits der Blockchain. Da kein dynamisches Binden von Funktionen unterstützt wird, das Verändern von Variablenwerten eingeschränkt wird und die Datenabstraktion durch Module ermöglicht wird, ist Move zugänglicher für statische Codeanalysen als manch andere Sprachen.

4.1.4.15 RIDE

RIDE [Pla20] ist eine case-sensitive funktionale Programmiersprache, die auf der WAVES[21] Plattform eingesetzt wird, um Scripts und Smart Contracts zu schreiben. Die Sprache ist statisch typisiert und besitzt einige Features, die speziell für die Funktionalitäten der WAVES Plattform ausgelegt sind. Die WAVES Plattform steht in direkter Konkurrenz zu Ethereum und bietet dementsprechend ähnliche Features an. Erwähnenswert ist die Möglichkeit Accounts und Assets mit RIDE Scripten zu versehen, wodurch beispielsweise vor einer Transaktion eines sogenannten *Smart Assets*, dieses erst durch das Script validiert werden muss [Pla19].

Dementsprechend lassen sich RIDE Scripts in drei verschiedene Untertypen aufteilen:

- dApp Script - um dezentralisierte Applikationen zu schreiben
- Account Script: Wird an einen Account angehängt, der in Folge als `smart account` bezeichnet wird.
- Asset Script: Wird an ein Asset (tokenisiertes Gut) angehängt, welches in Folge als `smart asset` bezeichnet wird.

Besondere Features:

- **Unveränderbare Variablen**: Ein einer Variable zugewiesener Wert in RIDE kann zu keinem Zeitpunkt mehr verändert werden.
- **Kein *gas***: Die Gebühren sind pauschal definiert, wodurch die Berechnungskosten bereits bei der Entwicklung einfach abschätzbar sind.
- **Complexity**: Die Complexity ist eine dimensionslose Zahl, die nach oben mit 4000 beschränkt ist und für die Entwickelnden als Richtwert dienen soll wie viel Rechenkapazität für die Ausführung des Smart Contracts nötig sein wird.

[21] https://wavesplatform.com

- **Nicht Turing-Vollständig**: RIDE unterstützt keine Schleifen, Sprünge oder Rekursionen und ist somit nicht Turing-Vollständig.

4.1.4.16 Obsidian

Obsidian [Cob20] ist die bereits in Kapitel 2 erwähnte, von Michael Coblenz vorgeschlagene, objektorientierte und stark typisierte Sprache, um von Grund auf sicherere Smart Contracts zu schreiben. Besonders erwähnenswert hierbei ist der Ansatz eine Sprache zu entwickeln, die basierend auf Studien und Umfragen besonders entwickelndenfreundlich sein soll. Die Sprache bildet Zustände als First-Class-Objekte ab, wodurch Zugriffe auf diese nur bei validen Zuständen ermöglicht werden.

Besondere Features:

- **Ownership**: Referenzentypen in Obsidian bestehen immer aus einem *Contract* und einer *Permission*. Die Berechtigungen `Owned`, `Unowned` oder `Shared` definieren, wem dieses Objekt gehört. Es kann, neben den Unowned-Referenzen, immer nur zeitgleich eine Owned-Referenz geben oder mehrere Shared-Referenzen, falls es keine Owned-Referenz gibt.
- **Zustandsorientiertes Programmieren**: Wie üblich für `typestate`-orientierte Sprachen sind aufgerufene Methoden vom aktuellen Zustand des Objekts abhängig. Zudem ist die Zustandsänderung mit dem Ownership-Paradigma verbunden, da nur die Owned-Referenz den Zustand verändern kann.
- **Lineares Typsystem**: Um sicherzustellen, dass keine Ressourcen wegen Programmfehlern verloren gehen, wird ein lineares Typsystem vorgeschlagen. Contracts können mit dem Schlüsselwort `asset` gekennzeichnet werden und der Compiler warnt bei Fehlverwendung dieser Typen.

4.1.4.17 Pact

Pact [Pop17] [Kad19] ist eine für die Kadena[22] Blockchain entwickelte, speziell auf Smart Contracts ausgelegte, funktionale Programmiersprache. Um größtmögliche Sicherheit zu erreichen wurde hier, ähnlich wie bei Michelson, eine lesbare Sprache entwickelt, welche direkt auf der Blockchain ausgeführt werden kann. In der Herangehensweise und Syntax orientiert sich Pact jedoch zum einen stark an funktionale Sprachen wie LISP, sowie zum anderen an Datenbanksprachen wie SQL.

Besondere Features:

- **Safe-oriented Design**:

[22] https://kadena.io/en/

  - **Unveränderbare Variablen**: Variablen werden gebunden und nicht zugewiesen. Diese können im Laufe der Codeausführung nicht mehr ihren Wert verändern.
  - **Turing-incomplete**: Endlosschleifen und Rekursion wird nicht unterstützt. Wird eine Rekursion im Programmcode erkannt, wird bereits beim Laden des *Modules* eine Fehlermeldung angezeigt und die Ausführung abgebrochen. Darüber hinaus können Schleifen nur für endliche Listenstrukturen angewendet werden. Auch auf ein Kostenberechnungen oder *Gas* kann dadurch verzichtet werden.
  - **Besondere Conditionals**: Entwickelnde sollen bei `if` Statements, um Invarianten zu verifizieren, auf `enforce` zurückgreifen.
  - **Typsicherheit**: Durch Typinferenz benötigt die Sprache bei Variablendeklarationen keine Typinformationen, ist aber dennoch stark typisiert.
  - **Keine Null-Werte**: Das definierte `totality` Prinzip verlangt es, dass Entwickelnde alle Szenarios im Code abdecken müssen. Ein `null` Wert führt unter Umständen dazu gewisse Edge-Cases unbehandelt zu lassen.
  - **Keine Lambdas, Makros oder Eval-Ausdrücke**: Im Unterschied zu LISP sind in Pact keine anonymen Funktionen vorgesehen, sondern nur Funktionen auf Modul-Ebene. Makros und Evaluierungen von beliebigen Ausdrücken werden ebenfalls nicht unterstützt. Diese Designentscheidungen wurden primär zugunsten der Perfomance getroffen, können aber durch die Simplifizierung der Sprache auch der Fehlervermeindung und somit Sicherheit dienen.

- **Human-readable, on-ledger code**: Pact Code wird unverändert und in leserlicher LISP-Syntax direkt auf der Blockchain gespeichert.

- **Atomic execution (transactions)**: Contract Operationen entsprechen Modul Funktionsaufrufen und können bei Fehlern, analog Datendanktransaktionen, wieder rückgängig gemacht werden.

- **Module, Schemas und Tabellen**: Für jeden Smart Contract, der in Pact geschrieben werden soll, muss zu Beginn das `Schema` definiert werden. Mit `Schema` werden in Pact die Definitionen für Spalten in einer Tabelle bezeichnet, in der die gesamten Daten des Smart Contracts zu finden sind. Das Schema beinhaltet sowohl die Feldnamen als auch deren Typen. Die dadurch entstehende `Table` speichert die Daten, ähnlich wie bei SQL, in einem relationalen Datenbankmodell ab. Die gesamte Geschäftslogik des Smart Contracts sowie die zuvor erwähnten Table-Schemadeklarationen werden schlussendlich aus sogenannten `Modules` geladen und ausgeführt. Jegliche von Modules entstandenen Daten werden in Tables abgespeichert.

- **Module-guarded tables**: Tabellen können durch Module und die darin definierten Zugriffsmöglichkeiten vor direkten Zugriffen beschützt werden.

- **Single-sig and multi-sig public-key authorization**: Um die Zugriffsrechte auf verschiedene Teile des Smart Contracts möglichst genau zu regeln, arbeitet Pact mit *Keysets*. Beim Entwerfen eines Smart Contracts muss zuvor mittels der Funktion `define-keyset` deklariert werden, welches Keyset Transaktionen authorisieren oder Funktionen aufrufen darf. Hierzu werden entsprechende Keys generiert und zusätzlich spezifiziert, wie viele dieser Keys notwendig sind, um die gewünschte Operation durchzuführen. Sind beispielsweise drei Keys generiert worden und wurde als Regel `keys-any` gewählt, so reicht einer der generierten Keys bereits aus, um eine Transaktion zu zeichnen und damit zu authorisieren.
- **Multi-step "pacts" for confidential execution**: Pacts sind Funktionen in Pact, die Schritte zwischen mehreren Entitäten, im Sinne von Koroutinen, als sequenzielle Transaktionen auf der Blockchain abbilden lassen.
- **Key rotation support**: Keysets können im Nachhinein auch angepasst werden, um etwa neue Keys hinzuzufügen oder welche zu entfernen. Die Transaktion zur Änderung des Keysets muss aber durch das bestehende alte Keyset authorisiert werden.
- **Designed for direct integration with industrial databases**: Das Design von Pact ermöglicht die direkte Integration in gängige relationale Datenbanksysteme, um effizient mit historischen Daten umgehen zu können.

4.1.4.18 GPLs

Betrachtet man Tabelle 4.2 genauer, erkennt man, dass auch Sprachen wie Java, Javascript oder C++ bei der Entwicklung von Smart Contracts verwendet werden. Einerseits gibt es Plattformen wie Corda [23], die selbst in Kotlin geschrieben wurden, somit Kotlin und Java als Smart-Contract-Sprache anbieten und diese Smart Contracts direkt auf der Java Virtual Machine (JVM) ausführen [HGB19]. Andererseits gibt es auch Plattformen wie NEO[24], die Sprachen per Software Development Kit (SDK) einbinden.

Sprachen wie Java oder C++ wurden zwar nicht dezidiert für Smart Contracts entwickelt, existieren aber schon länger, sind durch die kontinuierliche Weiterentwicklung und Verwendung sehr stabil und werden von vielen Entwickelnden bereits sehr gut beherrscht. Den Ansätzen, die am Anfang dieses Kapitels erwähnt wurden, folgend, sind dies durchaus wichtige und wünschenswerte Eigenschaften von Sprachen im Allgemeinen und daher auch für Smart-Contract-Sprachen.

4.1.5 Zusammenfassung

In Kapitel 4.1.1 wurden historische Anforderungen betrachtet, die bei der Grundidee von Smart Contract angedacht wurden. Da zu diesem Zeitpunkt Smart Contracts sehr

[23] `https://www.corda.net/`
[24] https://neo.org

Blockchain-Name	Blockchain-Sprache	Smart-Contract-Sprache
Aerum	Go	Solidity
Aeternity	Elixir Rust	Sophia Varna Solidity
Ardor	Java	Java
Ark	Javascript	Javascript
Bitcoin	C++	Ivy-Lang
Boscoin		Web Ontology Language
ByteBall	Javascript	Javascript
Cardano	Haskell	Solidity Plutus
Corda	Kotlin	Java Kotlin
Counterparty		Solidity Serpent
Dfinity	Haskell	Solidity
EOS	Cpp	Cpp
Ethereum		Solidity
Ethereum classic	Go C++ Rust	Solidity
Exonum		Rust Java
HyperLedger Fabric		Golang
ICON	Python	Python
Komodo	C++	C++
Lisk	Javascript	Javascript
Monax		Solidity
Neblio		Cpp Go Ruby Java
Nem		JSON
Neo	C#	Javascript C++ .Net Java Kotlin Go
Nxt	Java	Java
Polkadot	Rust	
Qtum		Solidity
Quorum		
RChain		RHOLang
Rootstock (RSK)	Java	Solidity
Stellar	C++	
Stratis	C# .Net	C# .NET
Tomochain		
Viacoin		Java
Wanchain	Go	Solidity
Waves	Scala	Ride
Zen		FSharp
nuls	Java	Java

Tabelle 4.2: Blockchain- und Smart-Contract-Sprachen von 37 verglichenen Blockchain Plattformen [Tok19]

stark auf traditionelles Vertragswesen bezogen waren, können die Punkte Observability, Verifiability, Privity und Enforceability nicht so leicht für aktuelle Smart Contracts übersetzt werden. Von diesen vier Anforderungen, ist die Verifizierbarkeit, auf den aktuellen Kontext bezogen, noch immer ein wünschenswertes und verfolgtes Ziel. Eine weitere Anforderung, die zwar nicht direkt genannt wird, aber dennoch herausgelesen werden kann, ist Ownership, die Abbildung von Eigentum. Diese Anforderung lässt sich durch die Ziele des traditionellen Vertragswesen verstehen, geistiges Eigentum sowie Eigentum an körperlichen Gegenständen festzulegen und den Umgang damit abzusichern.

Ownership ist auch in Kapitel 4.1.2 wiederzufinden, welches die Anforderungen anhand aktueller Anwendungen und zugehöriger Entwurfsmuster betrachtet. Betrachtet man zum Beispiel finanzielle Anwendungen und Wallets, erkennt man das Prinzip von Ownership im Besitz, Tausch, Handel oder der Versicherung von dem Eigentum Geld. Auch notarielle Anwendungen nutzen Ownership, um Smart Contracts als Besitzurkunden abzubilden. Beim Anwendungsfall Spiele befindet man sich entweder beim Glückspiel wieder, bei einer finanziellen Anwendung oder, im Fall der CryptoKitties, beim Sammeln und Handeln von Eigentum. Beim letzten nennenswerten Anwendungsfall von Smart Contracts, Libraries, ist kein direkter Bezug zur Ownership Anforderung ersichtlich. Da aber die restlichen Anwendungen alle einen Bezug zeigen, reicht dies aus, um Ownership als wichtige Smart Contract Anforderung festzuhalten. In den in Kapitel 4.1.2 genannten Entwicklungsmuster ist Ownership, zum Beispiel im Token oder Authorization, ebenfalls zu finden, wovon aber auszugehen war, da diese Entwurfsmuster in den Anwendungen eingesetzt werden.

Kapitel 4.1.3 betrachtet allgemeine Anforderungen anhand aktueller Sprachentwicklungen und nennt neben den traditionellen Anforderungen Correctness, Performance, Expressiveness und Speed of Compiling auch die anwenderorientierte Anforderungen Understandability, Ease of reasoning, Modifiability und Learnability. Wohingegen die Performance und Übersetzungsgeschwindigkeit in aktuellen Sprachen aufgrund von immer besser werdender Computer im Allgemeinen keine allzu große Rolle mehr spielt, sind die Korrektheit und Ausdrucksstärke noch immer eine Herausforderung. Auch anhand der in Kapitel 4.1.4 betrachteten Sprachen für Smart Contracts ist zu sehen, dass Korrektheit eine wichtige Anforderung darstellt. Abbildung 4.1 zeigt, wie oft welches Feature bei den gängigen Smart-Contract-Sprachen angeführt wird. Da die jeweiligen Sprachen keine einheitliche Nomenklatur bei der Nennung ihrer Feature nutzen, zeigt Tabelle 4.3, als Basis für Abbildung 4.1, die Gruppierung bzw. Zuordnung der Feature zu Anforderungen.

Es sei angemerkt, dass es zu Kapitel 4.1.4 viele weitere Sprachen gibt und auch kontinuierlich neue Sprachen hinzukommen. All diese zu behandeln, würde den Umfang dieser Arbeit sprengen. Zudem sind zu vielen dieser Sprachen zum aktuellen Zeitpunkt noch sehr wenige Informationen verfügbar.

Tabelle 4.3: Vereinheitlichung und Zuordnung der beworbenen Sprachfeature gängiger Smart-Contract-Sprachen zu Sprachanfoderungen

Sprache	Feature	Zuordnung
Ethereum Bytecode	SHA3	DSL
	Contract Umgebungsinformationen	DSL
	Block Umgebungsinformationen	DSL
	System-Operationen	DSL
Solidity	Speziell für Smart Contracts entwickelt	DSL
	Function Modifiers	deklarative Programmierung
	Vererbung von Smart Contracts	Polymorphismus
	Keine undefinierten Variablen	Keine Null-Werte
	Komplexe benutzerdefinierte Typen	Benutzerdefinierte Typen
	Libraries	Libraries
Vyper	Maximale Sicherheit	Sicherheit
	Maximale Lesbarkeit	Lesbarkeit
	Größtmögliche Einfachheit für Sprache und Compiler	Einfachheit
	Bounds & Overflow Checking	Sicherheit
	Signed Integers & Decimal Fixed Point Numbers	Genauigkeit
	Decidability	Gas-Limit
	Strong Typing	Korrektheit
	Small and understandable compiler code	Understandability
	Limited support for pure functions	-
	Keine Modifier oder Klassenvererbung	Keine Vererbung
	Kein Inline-Assembly	-
	Kein Überladen von Funktionen und Operatoren	Eingeschränkter Polymorphismus
	Keine Rekursion oder unendliche Schleifen	Keine Rekursion
	Kein Binary Fix Point	-
Ethereum Web-Assembly	Nahezu native Ausführungsgeschwindigkeit	Ausführgeschwindigkeit
	Smart Contracts in vielen traditionellen Programmiersprachen	-
	Große Community und Toolchains für Entwickelnde	Tools
Idris	Polymorphic und Dependent Types	Korrektheit
	Foreign Function Interface (FFI)	Interoperabilität
	Compiler-unterstütztes interaktives editieren	-
	Eager Evaluation	-
	Typen als First-Class-Citizen	-
	Do-Notation und Idiom Brackets	-
	State Machines	-

Fortsetzung auf nächster Seite

Tabelle 4.3 – *Fortsetzung von vorheriger Seite*

Sprache	Feature	Zuordnung
	Effects	-
Flint	Caller Protections	Sicherheit
	Type States	Korrektheit
	Immutability by default	-
	Asset Types	-
	Protection Blocks	Sicherheit
	Finate Loops	-
	Initialisers	-
	Limited Fallback Functions	Sicherheit
	Safe Arithmetic	Sicherheit
Formality	Accessible Syntax	Erlernbarkeit
	Fast and portable "by design"	Performance
	An elegant underlying Type Theory	Korrektheit
	An optimal high-order evaluator	Performance
	Guaranteed termination	Korrektheit
Huff	Macros	Ausdrucksstärke
	Jump Tables	Ausdrucksstärke
	Syntactic Sugar	Ausdrucksstärke
	No Push Opcodes	Sicherheit
Lira	Secure Foundation	Sicherheit
	Non-Turing complete	Nicht Turing-Vollständig
	Kompakte Sprachdefinition	Erlernbarkeit
Michelson	Primitive und High-Level-Datentypen	Ausdrucksstärke
	Typsystem	Korrektheit
	Compiler Output der selbst geschrieben werden kann	Erlernbarkeit
	Formale Verifikation	Korrektheit
	Monomorphie	Eingeschränkter Polymorphismus
Liquidity	Abdeckung der Michelson Programmiersprache	-
	Verschiedene Syntax-Systeme	Erlernbarkeit
	High-Level-Typen	Korrektheit
	Lokale Variablen anstatt Stackmanipulation	-
	Polymmorphimus	Polymorphismus
	Michelson Dekompiler	Tools
	Liquidity Complier	Tools
	Modularisierung	Wartbarkeit
fi	Einfache Basisstruktur	Erlernbarkeit
	Lokale Variablen	Korrektheit
	Strikte Funktionen	Korrektheit

Fortsetzung auf nächster Seite

Tabelle 4.3 – *Fortsetzung von vorheriger Seite*

Sprache	**Feature**	**Zuordnung**
	Funktion für Typumwandlung	Korrektheit
	Variablen Modifier	Lesbarkeit
	Globale Konstanten	-
LIGO	Mehrsprachigkeit	Erlernbarkeit
	Starke Typisierung	Korrektheit
	Leichte Integration	Kompatibilität
	Modularität	Wartbarkeit
Move	First-Class-Resourcen	Korrektheit
	Flexibilität	-
	Sicherheit	Korrektheit
	Verifizierbarkeit	Korrektheit
RIDE	Unveränderbare Variablen	Korrektheit
	Kein gas	Kein Gas
	Complexity	Effizienz
	Nicht Turing-Vollständig	Nicht Turing-Vollständig
Obsidian	Ownership	Ownership
	Zustandsorientiertes Programmieren	Korrektheit
	Lineares Typsystem	Korrektheit
Pact	Unveränderbare Variablen	Korrektheit
	Turing-incomplete	Nicht Turing-Vollständig
	Besondere Conditionals	Korrektheit
	Typsicherheit	Korrektheit
	Keine Null-Werte	Keine Null-Werte
	Keine Lambdas, Makros oder Eval-Ausdrücke	Performance
	Human-readable,on-ledger code	Lesbarkeit
	Atomic execution (transactions)	Korrektheit
	Modules, Schemas und Tabellen	Wartbarkeit
	Module-guarded tables	-
	Single-sig and multi-sig public-key authorization	-
	Multi-step "pacts"for confidential execution	-
	Key rotation support	-
	Designed for direct integration with industrial databases	-
GPLs	Verbreitung	Erlernbarkeit
	kontinuierliche Weiterentwicklung	-

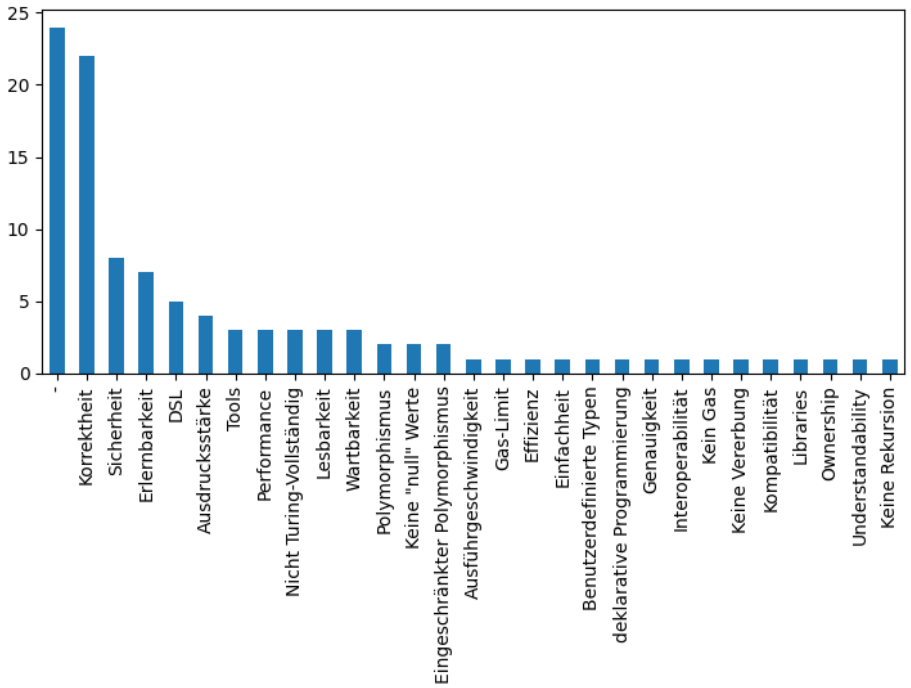

Abbildung 4.1: Häufung vereinheitlichter Anforderungen gängiger Smart-Contract-Sprachen

4.2 Zuordnung Spracheigenschaften zu Smart Contract Anforderungen

In diesem Kapitel werden die in Kapitel 4.1 gefunden Anforderungen **Ownership**, **Correctness**, **Expressiveness** und **Understandability** genauer betrachtet. Es wird untersucht, welche Spracheigenschaften diesen Anforderungen zugeordnet werden können und inwiefern die Anforderungen diese Spracheigenschaften beeinflussen.

4.2.1 Ausdrucksstärke

Die Ausdrucksstärke einer Sprache ist eine Eigenschaft, die zum Ausdruck bringt, wie gut Anwendende ihre Ideen und Absichten mit dieser Sprache ausdrücken können. Das, was viele Entwickelnde intuitiv bewerten würden, hat Landin [Lan66] bereits 1966 erforscht und damit den ersten Schritt in Richtung der formalen Definition dieser Eigenschaft gemacht. In seiner Arbeit stellt Landin eine neu konzeptionierte Programmiersprache, beziehungsweise Sprachfamilie If you See What I Mean (ISWIM) vor, welche aber eine abstrakte Idee blieb und nie umgesetzt wurde. ISWIM ist eine imperative Programmiersprache mit einem funktionalen Kern, der auf dem Lambda-Kalkül basiert und

somit Funktionen höherer Ordnung und statische Sichtbarkeit von Variablen mitbringt. Zuweisungen, veränderliche Variablen und der von Landin sogenannte *program-point* (auch als J-Operator bekannt [Lan65]), der als Kontrollmechanismus für die Auswertung von Ausdrücken eingesetzt werden kann und als eine Art von Sprung-Instruktion gesehen werden kann, erweitern den funktionalen Kern mit imperativen Eigenschaften. Zudem besitzt ISWIM, als eine der ersten Sprachen, eine *where*-Notation, um wie im mathematischen Vorbild, zusätzliche Erklärungen und Definitionen festhalten zu können.

Der ausschlaggebende Punkt bezüglich der Ausdrucksstärke war aber die Differenzierung der Sprache in einen *essentiellen Kern* sowie weitere Teile, die rein dekorativer Natur waren und über Umwege auch mit dem essentiellen Kern hätten ausgedrückt werden können. Aufbauend auf der Idee von Landin gab es weitere informale Betrachtungen der Ausdrucksstärke. Sussmann und Steele haben zu der von ihnen entwickelten Sprache *Scheme* [SS75], die Ausdrucksstärke von imperativen Erweiterungen, wie Zuweisungen, Sprung-Befehlen, *Call by Reference*- oder *Call by Name*-Aufrufen, untersucht [SS76].

Felleisen [Fel91] hat letztendlich ein formales System entwickelt, mit welchem zum ersten Mal die Ausdrucksstärke von Sprachkonstrukten formal überprüft werden kann. Mit dem System von Felleisen lassen sich Aussagen treffen wie dass eine *repeat*-Schleife genauso ausdrucksstark wie eine *while*-Schleife ist. Weiters kann festgehalten werden, dass das Erweitern einer Sprache um ein neues Konstrukt, wie zum Beispiel eine while-Schleife, die Sprache ausdrucksstärker macht, da in der ursprüngliche Variante der Sprache gewisse Aussagen nicht darstellbar sind, jedoch in der erweiterte Sprache nun ausgedrückt werden können.

Die formale Definition der Ausdrucksstärke ist natürlich sehr wichtig in der Entwicklung einer Programmiersprache und kann genutzt werden, um die Programmiersprache sinnvoll zu erweitern. Abseits der Sprachentwicklung ist dieser Ansatz aber sehr unpraktikabel, da das Ziel in den meisten Fällen nicht ist konkrete Vergleiche zweier Sprachversionen zu erstellen, sondern allgemein ein Gefühl für die Ausdrucksstärke einer Sprache zu gewinnen.

[Red15] vergleicht die Ausdrucksstärke vieler verschiedener Sprachen anhand der Zeilen Code pro Commit, unter der Annahme, dass durch jeden Commit dem Programm ein konzeptionelles Stück hinzufügt wurde. Die Idee dahinter ist eine sehr praktische, die die Anzahl der Zeilen an Programmcode in Relation zur Ausdrucksstärke stetzt, durch die Hypothese, dass wenig Programmcode darauf hindeutet, dass die eingesetzte Sprache ausdrucksstark ist und es leicht möglich ist, die Absicht der Entwickelnden durch die Grundbausteine der Sprache abzubilden.

Berkholz untersucht einen beträchtlichen Datensatz an Open-Source-Projekten (über sieben Millionen Projektmonate), die mit den unterschiedlichsten Sprachen entwickelt wurden. Abbildung 4.2 zeigt eine Visualisierung der Daten - eine nach dem Median der Ausdrucksstärke geordnete Darstellung der bekanntesten Programmiersprachen.

Seine allgemeinen Erkenntnisse anhand der vorliegenden Daten lassen sich wie folgt zusammenfassen:

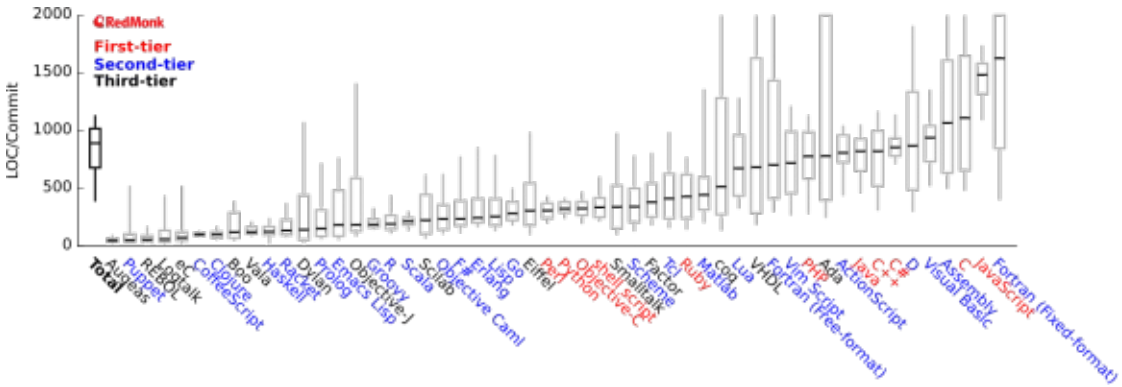

Abbildung 4.2: Box-Whisker-Plot: Verteilung von Codezeilen pro Commit pro Monat für etwa 20 Jahre, gewichtet nach der Anzahl der Commits in einem Monat. Quelle: `http://dberkholz-media.redmonk.com/dberkholz/files/2013/03/expressiveness_weighted2.png`

- Funktionale Sprachen sind sehr ausdrucksstark
- DSLs sind ausdrucksstark
- Die Ausdrucksstärke ist nicht abhängig davon, ob eine Sprache direkt interpretiert oder zuerst übersetzt wird.
- Sprachen mit *interaktivem Modus* neigen dazu nicht sehr ausdrucksstark zu sein.

4.2.2 Verständlichkeit

Die Verständlichkeit ist eine Eigenschaft, die misst, wie leicht Programmcode von Lesenden verstanden werden kann. Die Lesbarkeit von Programmcode ist somit auch eine Eigenschaft, die mit der Verständlichkeit zusammenspielt. Es muss aber differenziert werden, da die Lesbarkeit für sich eine selbstständige Eigenschaft darstellt und nicht direkt etwas über die Verständlichkeit von Programmcode aussagt.

In der Bemessung der Qualität einer Software nach ISO/IEC 9126-1:2001 [ISO20b] wurde Verständlichkeit als eine Untercharakteristik der Bedienbarkeit von Software genannt. Im Jahr 2011 wurde der bisherige ISO Standard ersetzt durch ISO/IEC 25010:2011 [ISO20a], in welchem der Begriff Verständlichkeit ersetzt wurde durch Angemessenheitserkennung (engl. *appropriateness recognizability*) und definiert wird als *"Degree to which users can recognize whether a product or system is appropriate for their needs."*.

Scalabrino et al. [SBV+19] haben in ihrer Arbeit den aktuellen Stand zur automatisierten Bestimmung der Verständlichkeit sehr gut erarbeitet. Sie haben eine empirische Studie zur Verständlichkeit durchgeführt mit 121 Messparametern, welche teilweise der bisherigen Literatur entnommen wurden und teilweise selbst definiert wurden. Die Studie wurde mit 63 Entwickelnden durchgeführt und hatte zum Ziel, eine Korrelation zwischen den eigenen

und bisher in der Literatur genannten Parametern und der Verständlichkeit empirisch zu beweisen. In drei Klassen unterteilt wurden codebezogene, dokumentationsbezogene und personenbezogene Messparameter in der Studie betrachtet. Das Ergebnis war ein sehr überraschendes. Bezüglich keinem der 121 Messparameter konnte eine mittlere oder starke Korrelation zur Verständlichkeit empirisch nachgewiesen werden. Vereinzelte Werte wie die maximale Zeilenlänge oder die Erfahrung der Entwickelnden zeigten eine schwache Korrelation zur Verständlichkeit.

Verständlichkeit ist somit eine sehr komplexe Eigenschaft, die durch ihre sehr subjektive Wahrnehmung schwer definiert werden kann und daher ein sehr spannendes offenes Forschungsgebiet darstellt.

4.2.3 Ownership

In der Grundidee von Smart Contracts war Ownership schon ein wichtiger Bestandteil. Szabo [Sza98] hat bereits im Jahr 1998 überlegt, wie man physischen Eigentum digital anonym abbilden, sichern oder transferieren kann. Im gleichen Jahr haben Clarke, Potter und Noble [CPN98] gezeigt, wie mittels Ownership Typen das Aliasing-Problem von objektorientierten Sprachen gelöst werden kann und haben damit den Grundstein für die weitere Forschung an Ownership Typen gesetzt. In ihrer Arbeit beschreiben sie, wie Ownership Typen ein statisches Typsystem bilden, anhand dem die Sichtbarkeit und der Zugriff auf Objektreferenzen kontrolliert werden kann, um Schutz vor ungewollten Zugriffen und Änderungen zu sichern.

Clarke, Östlund, Sergey und Wrigstadt [CÖSW13] haben im Jahr 2013, 15 Jahre nach der ersten Betrachtung von Ownership Typen für das Aliasing-Problem, eine Erhebung der Entwicklung und der Anwendung von Ownership Typen durchgeführt. Sie zeigen in ihrer Arbeit, dass es unterschiedliche Modelle gibt Ownership abzubilden, abhängig davon wie strikt die Einschränkungen und somit der Schutz sind. Das Model *Owners-as-Dominators* ist das strikteste und erlaubt in einer topologischen Objektstruktur den Zugriff auf interne Objekte nur über eine Stelle, welche hierarchisch definiert ist. Das bedeutet, dass in einem Objektgraphen der Owner eines Objektes immer nur genau ein direktes Eltern-Objekt dieses Objektes sein kann. Ausgehend von diesem Modell gibt es unterschiedliche Varianten, die durch immer mehr Lockerungen entstanden sind.

Müller und Poetzsch-Heffter [MPH99] haben, basierend auf dem Owner-as-Dominator Model, in ihrem *Universe Typsystem* das Model *Owner-as-Modifiers* eingeführt. Durch die Lockerung für Read-only-Referenzen, bleiben ungewollte Zugriffe und Änderungen geschützt und zugleich sind lesende Zugriffe außerhalb des Owners möglich.

Aldrich und Chambers [AC04] sind einen Schritt weiter gegangen und haben die topologische Objektstruktur teilweise aufgehoben, indem sie mit ihrem Model *Ownership Domains* die innere Struktur eines Objektes feiner unterteilen lassen. Ownership Domains erlauben die Unterteilung der Objektstruktur in mehrere Bereiche und für jeden dieser Bereiche ist der Zugriff separat geregelt. Dies ermöglicht den Zugriff von außen für gewisse Elemente

des Objektes zu ermöglichen, während andere weiterhin dem Owner-as-Dominator Model entsprechend topologisch, durch den einen Owner, gesichert sind.

Cameron et al. [CDNS07] haben letztendlich die topologische Objektstruktur komplett aufgehoben und haben mit ihrem Model von *Multiple Ownership* statt der Baumstruktur, mit immer nur einem Owner, eine Directed Acyclic Graph (DAG)-Struktur, mit der Möglichkeit mehrere Owner festzulegen, eingeführt. Für ihr Model haben sie die kompakte, objektorientierte und imperative Programmiersprache **M**ultiple **O**wnership for **J**ava-like **O**bjects (MOJO) entwickelt, welche ein Typsystem besitzt, welches mehrfache Owner nach ihrem Model unterstützt. Zudem haben sie zugleich einen Beweis für die Korrektheit ihres Systems beigelegt.

Coblenz et al. [COE+19] nutzen für ihre Smart-Contract-Programmiersprache Obsidian die Grundprinzipien der bisherigen Forschung zu Ownership, einem System, in dem ein Owner den Lebenszyklus eines Objektes, die Zugriffe und Modifikationen verantwortet. Sie nutzen dieses Konzept aber nicht wie prinzipiell vorgesehen zur Datenkapselung, sondern für die korrekte Abbildung von Typzuständen und Zustandswechseln. Zustandswechsel können nur von Referenzen mit annotiertem `@Owned`-Typ durchgeführt werden.

Betrachtet man die bisherige Forschung und die aktuellen Anwendungen, so ist zu sehen, dass Ownership ein Konzept ist, welches zwar unterschiedliche Formen annehmen kann, aber durch spezielle Schlüsselwörter, die die Programmiersprache bereitstellen muss, und einem zugehörigen Typsystem umgesetzt werden muss.

4.2.4 Korrektheit

Korrektheit ist eine Eigenschaft eines Computerprogrammes, die beschreibt, ob dieses Computerprogramm sich konform einer Spezifikationen verhält. Um die Korrektheit eines Computerprogrammes bezogen auf eine Spezifikation zu beweisen, muss mit Hilfe der formalen Semantik und der Hilfe von formalen Methoden, diese Eigenschaft verifiziert werden. Diese formalen Methoden zur Verifikation können grob in zwei Gruppen unterteilt werden - Model Checking und Deduktive Verifikation. Beim Model Checking wird die Spezifikation meinst mit einer Formel in temporaler Logik formalisiert und anhand spezieller Model Checking Algorithmen wird vollautomatisch, ohne weitere Benutzerinteraktionen, versucht in dem Computerprogramm eine Ausführung zu finden, die nicht der Spezifikation entspricht. Bei deduktiven Verfahren wird in einer Logik (z.B.: Aussagenlogik, Prädikatenlogik, Logik höherer Ordnung, usw.), anhand des Computerprogrammes und der Spezifikation eine Hypothese aufgestellt, die in weiterer Folge mit Hilfe von automatisierten Theorembeweisern (meistens SMT-Solver) oder interaktiven Theorembeweisern (z.B.: Coq[25], HOL[26] oder Isabelle[27]) bewiesen oder widerlegt wird.

Es zeigt sich also, dass Korrektheit eine Eigenschaft ist, die anhand der formal spezifizierten Syntax und Semantik einer Sprache und entsprechenden formalen Methoden, die diese

[25] `https://coq.inria.fr`
[26] `https://hol-theorem-prover.org`
[27] `https://isabelle.in.tum.de`

Spezifikationen verarbeiten können, formal verifiziert werden kann. Korrektheit wird als nicht durch eine Spracheigenschaft erreicht, sondern durch einen Prozess sichergestellt.

Die Untersuchueng von Nawaz et al. [NML+19] zeigt einen durchaus interessanten Aspekt auf. Nawaz et al. haben über 40 Theorembeweiser bezüglicher unterschiedlicher Parameter untersucht und dabei auch die Programmiersprache und das Sprachparadigma berücksichtigt.

Mehr als 40% der Theorembeweiser sind dem deklarativem Sprachparadigma entsprechend, nur 12 % dem imperativen Sprachparadigma entsprechend und der Rest multiparadigmatisch umgesetzt.

Aucch wenn man anhand der aktuellen Theorembeweiser und der eingesetzten Sprachen eher darauf schließen könnte, dass die Basis eines Theorembeweisers eine deklarative - im Speziellen funktionale - Sprache sein muss, gibt es auch aktuelle Ansätze [Abd17], die versuchen zu zeigen, dass objektorientierte Theorembeweiser nicht weniger mächtig sind.

4.3 Validierung von Solidity

Solidity befindet sich aktuell in der Version v0.7.3[28]. Diese Version wird anhand der Informationen, die die Dokumentation bereitstellt, gegen die im vorherigen Kapitel erarbeiteten Anforderungen und Eigenschaften evaluiert.

Anforderung	Spracheigenschaft	Solidity
Ausdrucksstärke	Funktionale Sprache, DSL	DSL
Verständlichkeit	–	Unzureichende Informationen
Ownership	–	Kein Ownership-Konzept
Korrektheit	–	Keine formale Spezifikation

Tabelle 4.4: Validierung von Solidity bezüglich Anforderungen und Spracheigenschaften

4.3.1 Ausdrucksstärke

Wie in Kapitel 4.2.1 beschrieben, sind funktionale Sprachen sehr ausdrucksstark, domänenspezifische Sprachen ausdrucksstark und Sprachen mit einem *interaktiven Modus* neigen eher dazu, nicht sehr ausdrucksstark zu seien.

Solidity ist eine objektorientierte, High-Level-Programmiersprache für Smart Contracts. Die Sprache ist somit von Grund auf nicht funktional und ein interaktiver Modus ist bei Solidity auch nicht zu finden. Die Sprache wird übersetzt in EVM-Bytecode und auf der EVM ausgeführt.

Da Soldity explizit für die Entwicklung von Smart Contracts auf der Blockchain Ethereum entwickelt wurde, wird sie als DSL gezählt. Anstatt Klassen werden `contracts` verwendet, es gibt dezidierte Dateitypen, wie `address`, speziell für die Erkennung anderer Smart Contracts auf der Blockchain, sowie global verfügbare Variablen und Funktionen wie `block.timestamt` oder `gasleft()`, um Informationen der Blockchain abzufragen.

4.3.2 Verständlichkeit

Verständlichkeit ist ein offenes Forschungsgebiet und lässt sich aktuell nicht einzelnen Spracheigenschaften zuordnen. Somit ist eine Evaluierung von Solidity bezüglich der Verständlichkeit mit den vorliegenden Informationen nicht möglich. Empirische Studien, die die Verständlichkeit von Solidity mit einer ausreichend großen Menge an Teilnehmenden erfragt, wären bestimmt eine Hilfestellung für diese offene Frage.

4.3.3 Ownership

Ownership ist eine weitere Anforderung, die sich nicht durch einzelne Spracheigenschaften erfüllen lässt. Ownership ist ein Konzept, welches bei der Entwicklung der Sprache

[28] `https://solidity.readthedocs.io/en/v0.7.3/`

berücksichtigt werden muss und in dieser Sprache auch eine große Rolle spielt. Solidity hat dieses Konzept nicht im Grundgerüst der Sprache inkludiert.

4.3.4 Korrekheit

Korrekheit ist ebenfalls eine Anforderung, die nicht durch einzelne Spracheigenschaften erfüllbar ist. Wie bereits zu Beginn in der Kapitel 2 beschrieben, ist die formale Spezifikation von EVM-Bytecode oder Soldity selbst, ob direkt oder über den Umweg einer Übersetzung in eine andere Sprache, ein aktuell sehr stark betrachtetes Forschungsgebiet. Solidity selbst wurde nicht von Anfang formal spezifiziert und somit auch nicht hinsichtlich der Korrektheit entwickelt.

KAPITEL 5

Zusammenfassung

Dieses abschließende Kapitel soll die Arbeit und die gewonnen Erkenntnisse nochmal zusammenfassen, die Ergebnisse zur Diskussion stellen und Möglichkeiten für zukünftige Forschung aufzeigen.

5.1 Zusammenfassung und Erkenntnisse

Anforderungen an Smart Contracts werden in dieser Arbeit von vier unterschiedlichen Blickwinkeln betrachtet.

Historische Anforderungen Vor fast 25 Jahren, bei der Entstehung der Grundidee zu Smart Contracts, wurden die vier Aspekte Observability, Verifiability, Privity und Enforceability als Grundprinzipien genannt. Zudem kommt noch, als Abbildung von Eigentum, das Prinzip von **Ownership** hinzu. Auch wenn zu diesem Zeitpunkt Smart Contracts noch sehr stark auf traditionelles Vertragswesen bezogen waren, liefern diese Punkte die ersten Anforderungen an Smart Contracts.

Anforderungen anhand aktueller Anwendungen Betrachtet man aktuelle Anwendungen von Smart Contracts, so lassen sich diese grob in die fünf Bereiche Finanzen, Notariatsakt, Spiele, Wallelts und Libraries zusammenfassen. Innerhalb dieser Anwendungen werden dazu die neun Entwurfsmuster Token, Authorization, Oracle, Randomness, Poll, Time Contraint, Termination, Math und Fork-Check verwendet.

Anforderungen anhand der Sprachentwicklung Aus dem Blickwinkel der allgemeinen Sprachentwicklung betrachtet, kommen grundsätzlich die vier traditionellen Anforderungen an Programmiersprachen **Korrektheit**, Performance, **Ausdrucksstärke**, Übersetzungsgeschwindigkeit auf. Aus der aktuellen Forschung zur Sprachentwick-

lung kommen noch die anwenderorientierte Anforderungen **Verständlichkeit**, einfache Schlussfolgerung, Modifizierbarkeit und Erlernbarkeit hinzu.

Anforderungen anhand gängiger Smart-Contract-Sprachen Betrachtet man die gängigen Smart-Contract-Sprachen, so ist das mit Abstand am häufigsten beworbene Feature die Korrektheit. Zu den weiteren Top fünf Features zählen Sicherheit, Erlernbarkeit, DSL und Ausdrucksstärke.

5.2 Diskussion der Forschungsfragen

Die Frage, welche zu Beginn in Kapitel 1 formuliert wurde und diese gesamte Arbeit gelenkt hat, wird nun beantwortet und diskutiert.

Welche spezifischen Eigenschaften muss eine Sprache für die Entwicklung von Smart Contracts besitzen?

- Was sind die Anforderung an eine Sprache für die Entwicklung von Smart Contracts?

  Anhand der vier in dieser Arbeit betrachteten Blickwinkel, muss eine Sprache für die Entwicklung von Smart Contracts, folgende Anforderungen erfüllen: **Ownership**, **Korrektheit**, **Ausdrucksstärke** und **Verständlichkeit**

- In welcher Beziehung stehen diese Anforderungen mit den Eigenschaften der Sprache?

  Von den vier Anforderung lässt sich nur die Ausdrucksstärke zum Teil den spezifischen Spracheigenschaften, **funktionale Sprache** und **DSL**, zuordnen.

- Welche dieser Spracheigenschaften besitzt Solidity und inwieweit ist Solidity geeignet für die Entwicklung von Smart Contracts?

  Die einzige wünschenswerte Eigenschaft, die Solidity besitzt, ist, dass Solidty eine DSL ist. Bezüglich der Verständlichkeit sind nicht ausreichend Informationen vorhanden um eine Aussage zu treffen. Soldity besitzt kein Ownership-Konzept und ist hinsichtlich der Korrektheit noch keine vollständige formale Spezifikation. **Solidty besitzt somit nur eine der vier erarbeiteten Eigenschaften**, um den aktuellen Anforderungen der Smart-Contract-Entwicklung zu genügen. Für die de facto Standardsprache für Smart Contracts ist dies eine **unzureichende Leistung**.

5.3 Zukünftige Forschung

In Kapitel 4.1.4 wurde nur ein Teil der gängigen Smart-Contract-Sprachen betrachtet und dabei zudem der Fokus auf Sprachen, die auf der Blockchain Ethereum ausgeführt werden,

gelegt. Es gibt viele weitere Programmiersprachen und es werden auch kontinuierlich neue Sprachen entwickelt. Zukünftige Arbeiten könnten weitere Sprachen betrachten oder auch prüfen, ob neue Features zu den bereits in dieser Arbeit untersuchten Sprachen - im Speziellen Solidity - hinzugefügt wurden.

Eine weitere Herausforderung, die in zukünftigen Arbeiten genauer betrachtet werden könnte, ist die Zusammenfassung und Zuordnung zu Anfoderungen der von den verschiedenen Sprachen beworbenen Features. Manchmal wird der Hintergedanke zu einem Feature offen kommuniziert, doch meist wird dieser nicht explizit genannt. In dieser Arbeit wurde zum Beispiel das Typsystem der Korrektheit zugeordnet, um über die unterschiedlichsten Sprachen gruppieren zu können. Eine andere Zuordnung oder eine feinere Unterscheidung kann zu anderen Ergebnissen führen.

In dieser Arbeit wurde auch ersichtlich, dass sich der konkrete Anwendungsfall von Smart Contracts, der in der Entwicklung von neuen Programmiersprachen die richtige Richtung weisen soll, noch nicht kristallisiert hat. Durch das Aufkommen von neuen Anwendungsgebieten wird es in Zukunft auch weitere Forschung in diesem Bereich geben müssen.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Akronyme

DAG Directed Acyclic Graph. 66

DSL Domain Specific Language. 46, 47, 64, 68, 72

EDSL Embedded Domain Specific Language. 44

EVM Ethereum Virtual Machine. 1, 5–8, 11, 38, 39, 46, 47, 68, 69

ewasm Ethereum WebAssembly. 42

FFI Foreign Function Interface. 43

GPL General Purpose Language. 43

ICO Initial Coin Offering. 32–34, 36

ISWIM If you See What I Mean. 62, 63

JVM Java Virtual Machine. 56

MOJO **M**ultiple **O**wnership for **J**ava-like **O**bjects. 66

SDK Software Development Kit. 56

SMT Satisfiability Modulo Theory. 8, 66

Wasm WebAssembly. 42

Literaturverzeichnis

[ABC17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A Survey of Attacks on Ethereum Smart Contracts (SoK). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, volume 10204 LNCS, pages 164–186. Springer Berlin Heidelberg, July 2017.

[Abd17] Moez A. AbdelGawad. Object-Oriented Theorem Proving (OOTP): First Thoughts. *arXiv*, pages 1–11, 2017.

[AC04] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3086:1–25, 2004.

[Alf19] Gabriel Alfour. Updates about LIGO and Marigold · LIGO. `https://ligolang.org/blog/2019/07/11/ligo-update`, Jul 2019. [Online; accessed 8. Jan. 2020].

[Azt19] AztecProtocol. huff. `https://github.com/AztecProtocol/huff`, Sep 2019. [Online; accessed 1. Jan. 2020].

[BCD+19] Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. Move : A Language With Programmable Resources. `https://developers.libra.org/docs/assets/papers/libra-move-a-language-with-programmable-resources.pdf`, 2019. [Online; accessed 23. Nov. 2019].

[BCH+19] Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. Mi-cho-coq, a framework for certifying tezos smart contracts, 2019.

[Blo19] Bloqboard. Digital Asset Lending via Open Protocols 2018. `https://bloqboard.com/Bloqboard_Research-Digital_Asset_Lending_via_Open_Protocols_2018.pdf`, Oct 2019. [Online; accessed 24. Nov. 2019].

[BP17] Massimo Bartoletti and Livio Pompianu. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security*, pages 494–509, Cham, 2017. Springer International Publishing.

[Bra13] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. *ACM SIGPLAN Notices*, 48(9):133–144, nov 2013.

[Bra16] Edwin Brady. State Machines All The Way Down An Architecture for Dependently Typed Applications. [Online; accessed 12. Jan. 2020], 2016.

[BSZB+16] Karthikeyan Bhargavan, Nikhil Swamy, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, and Thomas Sibut-Pinote. Formal Verification of Smart Contracts. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security - PLAS'16*, pages 91–96, New York, New York, USA, 2016. ACM Press.

[But16] Vitalik Buterin. r/ethereum - Explaining EIP 150, Oct 2016. [Online; accessed 27. Aug. 2018].

[CAMS18] Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. Interdisciplinary programming language design. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2018, pages 133–146, New York, NY, USA, 2018. ACM.

[CDNS07] Nicholas R. Cameron, Sophia Drossopoulou, James Noble, and Matthew J. Smith. Multiple ownership. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, pages 441–460, 2007.

[Cob17] Michael Coblenz. Obsidian: A Safer Blockchain Programming Language. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 97–99. IEEE, may 2017.

[Cob20] Michael Coblenz. Obsidian Programming Language, Jan 2020. [Online; accessed 11. Jan. 2020].

[COE+19] Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. Obsidian: Typestate and assets for safer blockchain programming, 2019.

[CÖSW13] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. *Lecture Notes in Computer Science (including subseries*

Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 7850:15–58, 2013.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, 33(10):48–64, 1998.

[Dai16] Phil Daian. Analysis of the DAO exploit, Aug 2016. [Online; accessed 27. Aug. 2018].

[EMEHR17] Benjamin Egelund-Müller, Martin Elsman, Fritz Henglein, and Omri Ross. Automated Execution of Financial Contracts on Blockchains. *Business and Information Systems Engineering*, 59(6):457–467, 2017.

[Eth19] Ethereum. Ethereum Improvement Proposals. `https://eips.ethereum.org/EIPS/eip-20`, Nov 2019. [Online; accessed 24. Nov. 2019].

[eTo20] eToreX. Lira - eToroX. `https://www.etorox.com/lira`, Jan 2020. [Online; accessed 1. Jan. 2020].

[fC20] fi Code. Learn fi - Documentation. `https://learn.fi-code.com`, Jan 2020. [Online; accessed 4. Jan. 2020].

[Fel91] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1):35 – 75, 1991.

[Fli19] Flintlang. The Flint Programming Language. `https://docs.flintlang.org`, Nov 2019. [Online; accessed 23. Nov. 2019].

[FMMT18] G. Fenu, L. Marchesi, M. Marchesi, and R. Tonelli. The ico phenomenon and its relationships with ethereum smart contract environment. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 26–32, March 2018.

[FS14] Peng Fu and Aaron Stump. Self types for dependently typed lambda encodings. In Gilles Dowek, editor, *Rewriting and Typed Lambda Calculi*, pages 224–239, Cham, 2014. Springer International Publishing.

[GJ97] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts*. John Wiley \& Sons, Inc., New York, NY, USA, 3rd edition, 1997.

[GMS18] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10804 LNCS:243–269, feb 2018.

[HGB19] Mike Hearn and Richard Gendal Brown. Corda: A distributed ledger. https://docs.corda.net/_static/corda-technical-whitepaper.pdf, Aug 2019. [Online; accessed 12. Jan. 2020].

[Ico19] Icodata. ICOdata - ICO 2018 Statistics. https://www.icodata.io/stats/2018, Dec 2019. [Online; accessed 26. Dec. 2019].

[IGRS16] Florian Idelberger, Guido Governatori, Régis Riveret, and Giovanni Sartor. Evaluation of Logic-Based Smart Contracts for Blockchain Systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9718, pages 167–183, 07 2016.

[Il19] Idris-lang. Idris | A Language with Dependent Types. https://www.idris-lang.org, Nov 2019. [Online; accessed 23. Nov. 2019].

[ISO20a] ISO/IEC 25010:2011 systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models. https://www.iso.org/standard/35733.html, Sep 2020. [Online; accessed 7. Sep. 2020].

[ISO20b] ISO/IEC 9126-1:2001 software engineering — product quality — part 1: Quality model. https://www.iso.org/standard/22749.html, Sep 2020. [Online; accessed 7. Sep. 2020].

[Kad19] Kadena. Pact Language Reference — Pact Language Reference 3.2.1 documentation. https://pact-language.readthedocs.io/en/stable, Dec 2019. [Online; accessed 11. Jan. 2020].

[Lan65] P. J. Landin. Correspondence between algol 60 and church's lambda-notation: Part i. *Commun. ACM*, 8(2):89–101, February 1965.

[Lan66] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, March 1966.

[Lar18] Eric Larcheveque. 2018: A record-breaking year for crypto exchange hacks - coindesk. https://www.coindesk.com/2018-a-record-breaking-year-for-crypto-exchange-hacks, 12 2018. (Accessed on 08/18/2019).

[LCO+16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, pages 254–269, New York, New York, USA, 2016. ACM Press.

[Led19] Ledger. Hack Flasback: The Mt.Gox Hack - The Most Iconic Exchange Hack | Ledger. https://www.ledger.com/hack-flasback-the-

mt-gox-hack-the-most-iconic-exchange-hack, Feb 2019. [Online; accessed 24. Nov. 2019].

[Lib19] Libra. Move Language · Libra. https://developers.libra.org/docs/crates/move-language, Nov 2019. [Online; accessed 23. Nov. 2019].

[Lig20] Ligolang. LIGO · LIGO is a friendly smart-contract language for Tezos. https://ligolang.org, Jan 2020. [Online; accessed 8. Jan. 2020].

[ML17a] Anastasia Mavridou and Aron Laszka. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. *CoRR*, abs/1711.0, nov 2017.

[Ml17b] Michelson-lang. Why Michelson? https://www.michelson-lang.com/why-michelson.html, Dec 2017. [Online; accessed 23. Nov. 2019].

[moo19] moonad. Formality. https://github.com/moonad/Formality, Dec 2019. [Online; accessed 1. Jan. 2020].

[MP08] CONOR MCBRIDE and ROSS PATERSON. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, jan 2008.

[MPH99] P Müller and A Poetzsch-Heffter. Universes: A Type System for Controlling Representation Exposure. In *Programming Languages and Fundamentals of Programming*, 1999.

[NML+19] M. Saqib Nawaz, Moin Malik, Yi Li, Meng Sun, and Ikram Ullah Lali. A Survey on Theorem Provers in Formal Methods. *arXiv*, 2019.

[O'C17] Russell O'Connor. Simplicity: A New Language for Blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security - PLAS '17*, pages 107–120, New York, New York, USA, 2017. ACM Press.

[OCa19] OCamlPro. liquidity. https://github.com/OCamlPro/liquidity, Oct 2019. [Online; accessed 23. Nov. 2019].

[PE16] Jack Pettersson and Robert Edström. *Safer smart contracts through type-driven development*. PhD thesis, Chalmers University of Technology and University of Gothenburg, 2016.

[PH06] Peter Pepper and Petra Hofstedt. *Polymorphe und abhängige Typen*, pages 145–156. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[Pla19] Waves Platform. Waves Smart Asset Applications: Whitelists, Blacklists and Interval Trading. https://blog.wavesplatform.com/waves-smart-asset-applications-whitelists-blacklists-and-

interval-trading-4169f11f8690, Mar 2019. [Online; accessed 11. Jan. 2020].

[Pla20] Waves Platform. About RIDE · GitBook. https://docs.wavesplatform.com/en/ride/about-ride.html, Jan 2020. [Online; accessed 11. Jan. 2020].

[Pop17] Stuart Popejoy. The Pact Smart-Contract Language. [Online; accessed 11. Jan. 2020], Jun 2017.

[Red15] Programming languages ranked by expressiveness. https://redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness, Apr 2015. [Online; accessed 2. Sep. 2020].

[Reu18] Arseny Reutov. Predicting random numbers in ethereum smart contracts. https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620, 2 2018. (Accessed on 08/18/2019).

[Ria18] Yos Riady. Common Smart Contract Vulnerabilities and How To Mitigate Them. https://yos.io/2018/10/20/smart-contract-vulnerabilities-and-how-to-mitigate-them, Oct 2018. [Online; accessed 12. Jan. 2020].

[SBV+19] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically Assessing Code Understandability. *IEEE Transactions on Software Engineering*, pages 1–18, 2019.

[Sch18] Franklin Schrans. *Writing safe smart contracts in Flint*. PhD thesis, Imperial College London, London, 2018.

[SED18] Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. Writing safe smart contracts in flint. In *ACM International Conference Proceeding Series*, volume Part F1376, pages 218–219. Imperial College London, jun 2018.

[Sol19] Solidity. Solidity — Solidity 0.5.13 documentation. https://solidity.readthedocs.io/en/v0.5.13, Nov 2019. [Online; accessed 23. Nov. 2019].

[SPZ+20] Amritraj Singh, Reza M. Parizi, Qi Zhang, Kim-Kwang Raymond Choo, and Ali Dehghantanha. Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. *Computers & Security*, 88(October):101654, jan 2020.

[SS75] Gerald Jay Sussman and Guy L. Steele. Scheme: A Interpreter for Extended Lambda Calculus, 1975.

[SS76] Guy Lewis Steele and Gerald Jay Sussman. Lambda: The ultimate imperative, 1976.

[st19] s tikhomirov. smart-contract-languages. https://github.com/s-tikhomirov/smart-contract-languages, Oct 2019. [Online; accessed 1. Jan. 2020].

[Swe16] Martin Swende. Announcement of imminent hard fork for EIP150 gas cost changes, Oct 2016. [Online; accessed 27. Aug. 2018].

[Sza96] Nick Szabo. Smart Contracts: Building Blocks for Digital Markets Copyright, 1996.

[Sza97] Nick Szabo. Formalizing and Securing Relationships on Public Networks. *First Monday*, 2(9):28, 1997.

[Sza98] Nick Szabo. Secure Property Titles with Owner Authority, 1998.

[Tez19] Tezos. Michelson: the language of Smart Contracts in Tezos — Tezos (master branch, 2019/12/30 16:41) documentation. https://tezos.gitlab.io/whitedoc/michelson.html, Dec 2019. [Online; accessed 2. Jan. 2020].

[Tok19] TokensEconomy. Comparison of major 37 Blockchain smart contract platforms by Tokens-Economy. https://platform.tokens-economy.com/table.html, Apr 2019. [Online; accessed 26. Dec. 2019].

[Vyp19] Vyper. Vyper — Vyper documentation. https://vyper.readthedocs.io/en/v0.1.0-beta.14, Nov 2019. [Online; accessed 23. Nov. 2019].

[Woo14] Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger. (Accessed on 08/18/2019), 2014.