Philosophie der Informatik

Philosophie der Informatik steht zur Informatik im gleichen Verhältnis wie die Philosophie der Mathematik zur Mathematik und die Philosophie der Technologie zur Technologie.

Aufgrund der abstrakten Natur der Informatik haben viele essentielle Fragen der Philosophie der Mathematik ein Analogon in der philosophie der Informatik; Aufgrund der technischen Natur der Informatik haben viele essentielle Fagen der Philosophie der Technik ein Analogon in der Philosophie der Informatik.

Außerdem gibt es Überscheidungen/Verbindungen zur Sprachphilosophie, zur Philosophie des Geistes (?) und zur Wissenschaftsphilosophie.

Im folgenden wird eine Reihe von eng verknüpften Themenfeldern bearbeitet, die das Skelett der Displin ausmachen: * Spezifikation (specification) * Implementierung (Implementation) * Semantik (semantics) * Programme (programs) * Programmieren (programming) * Korrektheit (correctness) * Abstraktion (abstraction) * Berechnung (computation)

Computationale Artefakte (computational artefacts)

Was sind die Gegenstände der Informatik? Was ist die Dinge die in so vielen Bereichen auf unser Leben Einfluss nehmen?

Die banale Antwort: Die Entitäten die ein Programmierer konstruiert, die Artefakte der Informatik, computationale Artefakte (computational Artefakts).

Ihrer Natur, Spezifikation, Design und Konstruktion ist ein großer Teil des Inhalts der Philosophie der Informatik gewidmet.

Dualität (duality)

Folklore: Computationale Artefakte zerfallen in zwei Kategorien: 1. Software 2. Hardware

Wie wird die unterscheidung zwischen Software und Hardware getroffen?

Klassisch wird die Unterscheidung durch eine Einteilung in abstrakte und physikalische Objekte vorgenommen.

Diese Einteilung ist mangelhaft: Programme – abstrakte Objekte – die unter dieser Einteilung der Software zugerechnet werden können auch als physikalische Schaltungen implementiert werden.

Antworten auf diesen Einwand: * Es gibt keinen Unterschied zwischen Hard- und Software. Hardware ist eine spezielle Unterform von Software. Die Unterscheidung ist ontologisch irrelevant. * Es gibt einen Unterschied aber der kann

innerhalb einer ontologischen Theorie erfasst werden, die feiner aufgelöst ist als die Unterteilung in physikalische und abstrakte Objekte.

Unabhängig davon, ob die Einteilung in Software und Hardware sauber formuliert werden kann, gibt es eine Übereinstimmung darin, dass ein Programm, obwohl es sich dabei um ein abstraktes Objekt handelt, in Hardware implementiert werden kann.

Daher haben Programme dualer Natur (Type/Token?): See haben sowohl eine abstrakte als auch physikalische Erscheinung; einerseits haben sie eine abstrakte Erscheinung die uns erlaubt über sie nachzudenken und zu schlussfolgern, unabhängig von einer physikalischen Manifestation. Das trifft auf Abstrakte Datentypen zu (Cardelli 1985), * Beispiel Listentype: Der Listentype besteht aus einem Trägertyp/Containertyp zusammen mit Operationen welche die Konstruktion und Manipulation von Listen erlauben. Selbst wenn diese nicht explizit angegeben werden, sind diese Operationen durch verschiedene Axiome festgelegt die ihre Eigenschaften festlegen (Idempotenz, Umkehroperationen, ...). * Beispiel Stapeltyp: 'Pop' ist die Umkehroperation zu 'Push', usw.

Unter Verwendung dieser Eigenschaften können Wahrheiten über diese Objekte (in diesem Fall Typen) unter Verwendung mathematischer Methoden gefunden werden – unabhängig von einer konkreten Implementierung.

Die Verwendung mathematischer Werkzeuge ist elementar: Es ist unmöglich korrekte Programme zu entwerfen ohne zu erörtern (reason) was sie tun sollen (intended to do).

Damit haben Computationale Artefakte eine abstrakte Erscheingung, die losgelöst von ihrer physikalischen Implementierung ist.

Die Notwendigkeit abstrakter Objekte anhand derer Eigenschaften physikalischer Objekte ermittelt werden können ist der Informatik nicht exklusiv gegeben:

Duhem: physikalisches Experiment.

Wittgenstein: Kinematik (wichtig! nachtragen!)

Das gleiche was für das physikalische Experiment gilt, gilt auch für einen Rechner: es erfordert die Verwendung einer abstrakte Maschine welche die Funktione eines komputers modelliert.

Genauso wie computationale Artefakte abstrakten Charakter haben, haben sie auch einen physikalischen Charakter. Dieser physikalische Charakter erst macht sie in der physikalischen Welt nutzbar. Das ist offensichtlich wahr für Maschinen trifft für Programme aber im selben Maß zu.

Dazu Dijkstra: Ein Programmierer entwirft Algorithmen, bestimmt für die mechanische Ausführung, bestimmt einen realen oder vorstellbaren Computer zu kontrollieren.

Unter dieser dualen Perspektive ist die Informatik nicht eine rein mathematische Disziplin die vollständig losgelöst von der physikalischen Welt ist; um Verwendet

zu werden benötigen diese Objekte Substanz – diese Beobachtung stellt eine deutliche Verbindung zur Philosophie der Technologie her: technische Artefakte.

Technische Artefakte

Die Menge der technischen Artefakte umfasst die menschgemachten Objekte des täglichen Gebrauchs wie Taschentücher, Schraubendreher, ...

Es handelt sich um vorsätzlich angefertige Objekte. Dies ist ein essentieller Teil davon ein technisches Artefakt zu sein.

Beispiel: Ein physikalisches Objekt welches zufälligerweise arithmetische Berechnungen ausführt ist nicht ob seiner selbst ein Rechner. Dieser teleologische (zweckorientierte) Aspekt unterscheidet technische Artefakte von anderen Objekten. Technische Artefakte haben damit eine duale Natur, festgelegt von zwei Eigenschaftsmengen: 1. Funktionale Eigenschaften 2. Strukturelle Eigenschaften

Funktionale Eigenschaften treffen Aussagen darüber was ein Objekt tut. Beispiel: Ein Auto dient dem Transport.

Strukturelle Eigenschaften treffen Aussagen betreffen den physikalischen Aufbau; Sie beinhalten Gewicht, Farbe, Abmessungen, Gestalt, ...

Der begriff des technischen Artefakte ermöglicht es einige zentrale Fragen und Anliegen der Philosophie der Informatik zu konzeptualisieren und organisieren.

Spezifikation und Funktion

Spezifikation und Funktion sind der grundlegende Gegenstand des ersten Aspekts technischer Artefakte: ihrer funktionellen Eigenschaften. (teleologischen Eigenschaften?)

Bist zu Fertigstellung eines finalen Systems wird oft eine ganze Reihe von Paaren 'Spezifikation' * 'Artefakt' Paaren mit unterschiedlichen Abstraktionsgraden modelliert (unterschiedlicher Auflösungsgrad der Modellierung).

Schritte: 1. Spezifikation (Konjunktion von Eigenschaften). 2. Implementierung. 3. Überprüfung der Korrektheit der Implementierung gemäß der Spezifikation. Eine korrekte Implementierung ist dabei der Beweis der in der Spezifikation angegebenen Eigenschaften.

Diese Ablauffolge ist Gegenstand einer ganzen Reihe von konzeptioneller Fragen und und Probleme. (vgl. auch Dörner: Zielsetzung, Modellbildung, Planung, Durchführung, Analyse)

Definition

(Was hat der Abschnitt hier mit einer Definition zu tun? Definition von was? von Spezifikationen? gabs nicht auch einen SEP-Eintrag zu Definitionen?)

Spezifikationen können auf verschiedene Weisen formuliert werden: * Informal, Umgangssprachlich (zunehmend weniger verbreitet) * Formal (zunehmend häufiger) formale Sprachen für die Programmspezialisierung * VDM * Z * UML (großes Spektrum) Diese Sprachen unterscheiden sich im Bezug auf die ihnen zu grunde liegende Ontologie (wird ein wichtiger Punkt bei der Typentheorie), und den sprachlichen Mitteln die sie zur Modellierung bereitstellen (Expressivität der Sprache).

Spezifikationssprache Z: Basierend auf der Prädikatenlogik und der Mengenlehre. Sie wird für die Spezifikation von Programmmodulen und Programmen verwendet.

Spezifikationssprache UML: Die Spezifikationssprache UML hat eine reiche Ontologie und einen großen umfang expressiver Sprachmittel (hohe Expressivität). (Anscheinend doch nicht so schlecht da mathematische Formalismen, nacharbeiten)

(Was ist mit Aussagenlogik/Prädikatenlogik in SAt/SMT-Solvern?) (Was it mit der Typentheorie als Spezifikationssprache?)

Was ist die logische Funktion (?) dieser Sprachen? Oberflächlich: Ausdrücke einer formalen Sprache.

Wenn die unterliegende Ontologie explizit gemacht wird, offenbart sich, das jede dieser Sprachen eine formale Ontologie ist (?) welche auf einem natürlche Art in eine Typentheorie übertragen werden kann (Turner 2009a).

Unter dieser Interpretation sind diese Ausdrücke stipulative definitionen. (stipulative - new meaning for the sake of the argument?) Als solche definiert jede ein neues abstraktes Objekt innerhalb der formalen Ontologie ihre Systems.

Definitionen als Spezifikationen

Für sich selbst genommen muss eine Definition nicht die Spezifikation eines Dings sein; sie kann dann lediglich ein Teil mathematischer Untersuchungen sein.

(hier wird doch von einem sehr speziellen Definitionsbegriff ausgegangen, oder?)

Wann ist eine Definition eine Spezifikation? Mutmaßlich wenn die Definition so interpretiert wird, dass sie Vorgaben zur konstruktion eines Artefacts enthält.

Es ist diese vorsätzliche Interpretation der Definition, die sie zu der Spezifikation eines Programms oder Systems werden lässt. (Beispielsweise Idempotenz, ...)

Über einen Abgleich mit dieser als Spezifikation interpretierten Definition kann dann entschieden werden ob ein System korrekt implementiert wurde. Sie liefert die Kriterien der Korrektheit oder Fehlerhaftigkeit des Systems. So interpretiert ist die Rolle einer Spezifikation normativ.

Auf allen Ebenen der Abstraktion bleibt die logische Rolle von Spezifikationen gleich: Sie liefert ein Kriterium über die Korrektheit oder Fehlerhafitgkeit des Systems.

(Dieser Normative Teil ist essentieller Teil jeder funktionalen (teleologischen) Theorie).

Das ist eine Idealisierung: Während der Entwicklungsphase kann die Spezifikation überarbeitet werden: * geänderte Anforderungen * widersprüchliche Forderungen – das System kann so nicht gebaut werden. (logische, physikalische, Ressourcen ... Gründe)

Die Logische Funktionen von Definitionen ist nicht die wissenschaftlicher Theorien (normativ vs deskriptiv) (Turner 2011): Wenn ein Artefakt nicht der Spezifikation entspricht ist das Artefakt fehlerhaft und nicht die Spezifikation; Wenn die Theorie eines Artefakts nicht dem Artefakt entspricht ist die Theorie fehlerhaft und nicht das Artefakt.

(wissenschaftliche Theorie: deskriptive funktionale Beschreibung; Spezifikation: normative funktionale Beschreibung)

Darüber hinaus gibt es auch zeitliche Unterschiede: Eine Spezifikation wird vor der Implementierung eines Systems erstellt, eine Theorie des Systems danach. Die Spezifikation übernimmt damit noch eine weitere Rolle: Die einer Anweisung an den Konstrukteur des Systems.

Der Unterschied zwischen wissenschaftlicher Theorie und Spezifikation eines Systems besteht in ihrere Interpretation (und temporalen Aspekten) – ihre Darstellung und ihre Darstellungssprache können identische sein.

Abstrakte Artefakte

Software ist das Produkt einer Serie von Ebenen mit einem Abnehmenden Abstraktionsgrad, wobei sowohl die ersten Ebenen von Spezifikation und Artefakt abstrakt sind.

Beispiel: Eine in logischer Notation formalisierte Definition wird als Spezifikation eines linguistischen Programms aufgefasst.

Die Definition des linguistische Programm mit der ihm zugeordneten Semantik wird als Spezifikation eines physikalischen Geräts interpretiert. Das bedeutet, dass auch abstrakte Entitäten als Artefakte zugelassen sind.

Die zwischenstuflicher abstrakter Artefakte ist essentiell; ohne sie wäre die Konstruktion komplexer computationaler Artefakte unmöglich.

(Auch Implementierungen sind Definitionen die widerum als Spezifikationen interpretiert werden können.)

Was bedeutet das für die Dualitätsthese? Diese ist immer noch gültig: Allerdings erfassen strukturelle Beschreibungen jetzt nicht nur die Eigenschaften physikalischer sondern auch abstrakter Objekte. (Erweiturung der Dualitätsthese um abstrakte Objekte.)

Beispiel:

Spezifikation eines abstrakten Stacks in Prädikatenlogischer Notation Implementierung der Spezifikation in einer Programmiersprache. Diese Implementierung ist immer noch ein abstraktes Objekt ohne physikalische Eigenschaften.

Implementierung in der Programmiersprache als Spezifikation für die Implementierung des Stack als physikalisches Gerät.

Funktionstheorie

(hier: Intentional (Functional) im teleologischen Sinn. vorsicht: Intention ist nicht Intension!!!!!!!)

Wie genau die physikalische und intentionale Konzeptualisierung unserer Welt in Beziehung stehen bleibt ein ärgerliches Problem, wie die lange Geschichte des Körper-Seele-Problems bezeugt.

```
Diese Situation betrifft auch unser Verständnis technischer Artefakte:
Ein konzeptuelles Gerüst/Bezugssystem welches physikalische und
intentionale (funktionale) Aspekte von technischen Artefakten vereint
ist bislang noch nicht existent.
(Kroes and Meijers 2006:2)
```

Die Literatur über technische Artefakte enthält zwei Theorien darüber wie sich die physikalischen und intentionalen Aspekte eines Systems zueinander verhalten: 1. kausale Rollentheorien 2. intentionale Theorien

Kausale Rollentheorie

Die kausalen Rollentheorien behaupten, dass aktuale physikalische Eigenschaften die Funktion eines Objekts bestimmen; ohne das konkrete physikalische Objekt und dessen faktischen Eigenschaften kann es keine Artefakte geben (keine abstrakte Interpretation?).

Gegenargumente/Kritik an kausalen Rollentheorien: Diese Theorie verhindert die Formulierung eines Korrektheitskriteriums: die Funktion des Geräts wird festgelegt davon was das Gerät eigentlich tut. (Pearl hahaha, operationale Semantik?)

Argumente: Kripke, Kroes

Eine abstrakte Maschine liefert ein Korrektheitskriterium für eine physikalische Maschine. Ohne diese unabhängige Definition liegt kein Kriterium für Korrektheit oder die Fehlerhaftigkeit der Maschine vor.

intentionale Theorien

Die intentionalen Theorien behaupten das Agenten Artefakten Funktionen zuschreiben. Objekte und ihre Komponenten verfügen nur über Funktionen wenn sie zur Erreichung eines Ergenisses/Ziels beitragen.

Wie genau wird die Funktion von den Zielen des Agenten festgelegt?

Interpretation 1:

Funktion wird festgelegt von den mentalen Zuständen eines Agentens, den Schöpfern und Benutzern technischer Objekte. In dieser reinen Form fällt es den intentionalen Theorien schwer, die Menge der Dinge zu beschränken die sie als Artefakt verstehen. (physikalische Komponente wird vernachlässigt)

Wie kann der mentale Zustand eines Agenten die Funktion eines Geräts festlegen, welches Additionen durchführen soll.

Argument: auch Kripke

Interpetation 2: Wittgenstein/Pears

Jeder der intentional handelt muss zwei Dinge wissen: 1. Welche Aktivität er gerade betreibt 2. Wann er die Aktivität erfolgreich beendet hat.

Nach dieser Ansicht ist die Herstellung von Korrektheit eine extern beobachtbare Aktivität. Die Beziehung zwischen Definition und Artefakt manifestiert sich darin die Definition als Maßstab für die Korrektheit des Geräts anzusetzen.

Es muss mir möglich sein meine Überzeugung, dass das Gerät funktioniert zu begründen: das geschieht unter Referenz auf die abstrakte Definition.

Der Inhalt der Funktion wird durch die abstrakte Definition festgelegt, aber die Intention sie als Spezifikation zu verwenden wird durch ihre Verwendung als solche manifestiert.

Implementation

Die Implementierung ist die Realisation/Umsetzung einer Spezifikation.

Beispiele: * Implementierung einer UML Spezifikation in Java * Implementierung eines abstrakten Algorithmuses in C * Implementierung eines abstrakten Datentyps in Haskell

Implementierung ist oftmals ein indirekter, vielstufiger Prozess, wobei jede Stufe mit einem Spezifikation - Implementierungs - Paar einhergeht.

Was aber ist eine Implementierung? Gibt es für ihre Natur nur eine Auffasung oder verschiedene?

Was ist Implementierung?

Bislang detaillierteste philosophische Untersuchung der Implementierung stammt von Rapaport (1999, 2005): er legt dar, dass eine Implementierung in Gebiete umfasst: * eine syntaktische (die Abstraktion) * eine semantische (die Implementierung)

Die vollständige Definition einer Implementierung schließt darüber hinaus noch einen weiteren – bislang versteckten – Begriff ein: das Medium.

I ist eine Implementierung von A im Medium M. I : Semantische Komponente A : Abstraktion (soll das das gleiche sein wie die Spezifikation?) M : Medium

Das Medium kann sowohl abstrakt als auch konkrete sein.

Example: Implementation of a programming language: * syntaktische Domäne: die eigentliche Programmiersprache * semantische Domäne: Interpretation der Sprache auf einer abstrakten Maschine * Medium (der Interpretation): abstrakte Maschine

Weiter führt er an, dass es keinen intrinsischen Unterschied gibt weleche der Domänen syntaktisch und welche semantisch ist. Dies wird durch die Asymmetrie der Implementierungsabbidlung festgelegt.

Beispiel: Ein physikalischer Berechnungsprozess der die Implementierung eines linguistisch definiterten Programms; das gleiche linguistische Programm kann die Rolle der Semantischen Domäne für einen abstrakten Algorithmus spielen.

Diese Assymetrie ist gleich zu der Spezifizierung/Artefakt-verbindung.

Implementierung als semantische Interpretation

Normative Auffassung von Interpretation.

Die semantische Domäne wird immer als semantische Repräsentation der syntaktischen behandelt; sie schließt eine semantische Lücke zwischen der Abstraktion und der Implementierung in dem Sinn, dass die Implementierung Details ergänzt (???).

Dies ist eine referentielle Sicht von Semantik: die syntaktische Domäne bezeichnet eine andere Domäne welche ihre Bedeutung festlegt. Referentielle/Denotationale Semantiken als fundamental zu betrachten hat eine lange Tradition in der Informatik (mehr in Abschnitt 4).

Im weiteren Verlauf dieses Abschnitts nur die zentrale Rolle der Semantik (unabhängig von ihrer Art (???)).

normative Rolle der Semantik

Eine Sicht auf die Semantik ist es, diese als normativ zu betrachten. Die genaue Form dieser Normativitätseinschränkung ist Gegenstand der aktuellen

Debatte, allerdings herrscht weitestgehende Übereinstimmung einer minimalen Einschränkung/Anforderung: Ein semantischer Ansatz muss festlegen was es genau bedeutet einen Ausdruck korrekt zu verwenden.

Die Tatsache, dass ein Ausdruck etwas bedeutet impliziert, dass es eine ganze Reihe von normativen Wahrheiten über meinen Umgang mit diesem Ausdruck gibt; namentlich, dass mein Umgang damit korrekt ist im Bezug auf bestimmte Objekte und nicht im Bezug auf andere ...

Die Normativität der Bedeutung ist damit nur ein neuer Name der bekannten Tatsache, dass unabhängig von eine wahrheitstheoretischen (truth-theoretic) oder ??? (assertion-theoretic) Auffassung von Bedeutung (meaning), bedeutungstragende Ausdrücke Bedingungen für ihren korrekten Gebrauch besitzen.

Kripkes Erkenntnis war es zu realisieren, dass durch diese Beobachtung eine Adäquatheitsbedingung (Minimalanforderung) für semantische Theorien darstellt:
jeder mögliche Kandidat für die Eigenschaft aufgrund derer ein Ausdruck eine Bedeutung hat muss so sein, das er die Normativität von Bedeutung begründet -- es muss möglich sein anhand jeder bedeutungsverschaffenden Eigenschaft eines Ausdrucks abzulesen wie dieser korrekt gebraucht wird.

(Im Prinzip: die Bedeutung legt den korrekten Gebrauch eines mit ihr assoziierten Ausdrucks fest und schränkt damit dessen Benutzung ein.)

Unter der Annahme, dass diese minimale Anforderung von jeder adäquaten Semantischen Theorie erfüllt werden muss, kann Implementierung (immer, manchmal) als semantische Interpretation aufgefasst werden?

Beispiel 1: Interpretation von einer Sprache in einer anderen Sprache Hier sind sowohl die syntaktische als auch die semantische Domäne Sprachen. Allerdings liefert die Zielsprache kein Korrektheitskriterium wenn sie nicht selbst eine Semantik mit einem solchen Kriterum hat (dann ist die Bedeutgung transitv). Eine uninterpretierte Sprache kann kein solches Kriterium bereitstellen.

Beispiel 2: Abstraktion ist eine Sprache, Semantik die Mengenlehre Dies ist ein Beispiel für denotationelle Semantik. Hierbei gibt es ein Korrektheitskriterium. Unser Verständnis der Mengenlehre liefert dieses. Allerdings gibt es dabei ein wichtiges Problem: Notwendiges Kriterium einer Implementierung ist es (nach weiteren implementierungen) physikalische realisierbar zu sein. Dies ist bei der Mengenlehre nicht der fall. (hat der oben nicht gesagt, dass Implementierungen auch abstrakt sein dürfen?)

Beispiel 3: Abstraktion ist ein Stack, Semantik ein konkreter Array Kein solches Kriterium möglich da der Array den durch den Stack festgelegten Axiomen gehorchen soll, nicht umgekehrt.

Beispiel 4: Abstraktion abstrakte Maschine, Semantik konkrete Maschine Gleiches Problem wie bei der Funktionstheorie

Das sind alles sehr gewichtige Einwände gegen die Auffassung von Implementierung als semantischer Interpretation (normativ).

Spezifikation und Implementierung

Alternativer Ansatz zur Implementierung (Turner 2012, 2013).

Folgendes Beispiel: Der Datentyp endlicher Mengen wird durch den Datentyp Liste implementiert. Jede dieser Strukturen erfüllt einige einfache Axiome.

Implementierung: * endliche Mengen -> Listen * Vereinigungsoperation -> concatenation * ...

Dies ist eine mathematische Beziehung wobei die Axiome der endlichen Mengen als Spezifikation des Artefakts dienen, welches wiederum im Medium des Typs Liste implementiert wird.

Es erscheint als sei die logische Verbindungs zwischen dem Typ 'endlicher Menge' und 'Liste' der von Spezifikation und Artefakt. Die Abbildung muss keine direkte Korrespondenz sein (Operation auf operation, ...) aber die Eigenschaften der Liste und ihrere Operationen müssen die Axiome der Mengenlehre erfüllen. In mathematischer Terminologie: Der Listentyp muss ein Modell (im Sinne der Modelltheorie) für die Mengenaxiome sein.

Wenn dieser Ansatz richtig ist, dann wird Implementierung am besten als Relation von zwischen Spezifikation und Artefakt beschrieben; Implementierung ist nicht semantische Interpretation.

Das setzt ein unabhängiges Konzept von Semantik voraus um eine das Konzept der Korrektheit einer Implementierung formulieren zu können (???).

Semantik (semantics)

Wie kann die Semantik einer Sprache gegeben werden? Was sind dabei die zentralen Punkte? Es gibt viele Arten von Semantiken (Gordon, Milne and Strachy, ...)

Die wichtigste Unterscheidung: * operationale Semantiken * denotationale Semantiken

Operationale Semantiken

(Reifiziert Auswertungsschritte, Programmäquivalenz ...) (Für alle Arten von Programmiersprachen möglich?)

Begründet von Landin (1964). In seiner logischen Darstellung handelt es sich um einen Auswertungsmechanismus wo die Auswertungsrelation (in ihrer simpelsten Form) wie folgt dargestellt wird:

P downarrow c

Das Programm P konvergiert zu einer kanonischen Form (canonical form) c (Big Step semantics).

Operationale Semantiken werden als eine Menge von Regeln gegeben, die die Auswertung eines komplexen Programmes durch die Auswertung seiner Teile festlegen.

(hier beispiel einfügen)

Diese kanonischen Formenen (auch Normalformen) sind wiederum terme (Ausdrücke) in der Programmiersprache die anhande der gegebenen Regeln nicht weiter reduziert werden können (Fixpunkte module Evaluation).

Aus dem Grund, dass es sich um Terme der Sprache handelt werden operationale Semantiken oftmals als unbefriedigent betrachtet. Nach dieser Kritik muss die Semantik einer Sprache zu irgend einem Zeitpunkt der Interpretation mathematisch sein.

```
Man kann mit der Erklärung der Eigenscaften einer Sprache mit rein syntaktischen
Regeln offensichtlich weit kommen.

Eine dieser Sprachen ist das Lambdakalkül, und dieses kann als rein formales
System mit bestimmten Konvertierungsregeln dargestellt werden.

Aber wir müssen uns immer bewusst sein, dass wenn wir auf diese Weise
vorgehen, alles was wir tun ist Symbolmanipulation -- wir haben keine
Ahnung wovon wir eigentlich sprechen.

Um irgendein wirkliches Problem zu lösen müssen wir eine semantische Interpretation
angeben. Wir müssen zum Beispiel sagen: "Diese Symbole repräsentieren die
Ganzzahlen (Stoy 1977)
```

Operationale Semantik ist rein syntaktischer Natur. Die Relation setzt nur syntaktische Objekte zueinander in Beziehung. Dies sagt nichts darüber aus wovon wir reden.

Wenn den Konstanten der Sprache nicht unabhängig davon eine mathematische Bedeutung zugeschrieben wird kommen wir an keinem Punkt des Konvertierungsprozesses zu den semantischen Grundlagen; Wir reduzieren nur ein syntaktisches Objekt zu einem anderen.

Aus dieser unzulänglichkeit ergibt sich die Forderung nach einem mathematischeren Ansatz.

Denotationale Semantiken

Offensichtlich referenzieren programmiersprachen (bzw sind Programmiersprachen Notationen für) abstrakte mathematische Objekte und nicht für syntaktische.

Denotationale Semantiken setzen jedes syntaktische Objekt P mit einem Mathematischen Objekt in Beziehung. Weiter erfolgt dies auf kompositionale Weise: Die Denotation komplexer Programme wird durch die Denotation ihrer Teilprogramme festgelegt (Freges Kompositionalität ???).

Diese mathematischen Objekte können Mengentheoretisch, kategorientheoretisch oder typentheoretisch gegeben werden. Programme beziehen sich dann auf konkrete mathematische Objekte. Dieser Ansatz bezieht sich auf eine klare Trennung zwischen syntaktischen und mathematischen Objekten. (Die Bedeutung eines Ausdrucks ist das im zugeordnete mathematische Objekt.)

Programmiersprachen als axiomatische Theorien

Ansatz: Mathematische Theorien wie die Mengenlehre und die Kategorientheorie sind axiomatische Theorien. Das ist es was sie mathematisch macht. Das ist in der modernen axiomatischen Formulierung der Mathematik implizit (Hilbert, Bourbaki). Brief von Hilbert and Frege, 29 Dezember 1899

```
Meiner Meinung nach kann ein Konzept logisch nur durch seine Beziehungen
zu anderen Objekten festgelegt werden.
Diese Beziehungen, formalisiert als Aussagen, nenne ich Axiome und komme
damit zu dem Schluss, dass Axiome (vll. zusammen mit Aussagen die
Konzepten Namen zuordnen) die Definitionen von Konzepten sind.
Ich habe mir diese Sicht nicht ausgedacht weil ich nichts besseres zu tun hatte,
ich sah mich in sie hineingezwängt von den Anforderungen der logischen
Schlussfolgerung und der logischen Konstruktion einer Theorie.
...
Jede Theorie kann immer auf eine unendliche Menge von Systemen
basaler Elemente angewandt werden.
```

Der axiomatische Ansatz muss nicht formal sein, solange er präzise ist und mathematisches Schlussfolgern zulässt. Wenn man dies als notwendige Bedingung für den mathematischen Status akzeptiert, schließt das operationale Ansätze aus? Zunächst sieht es dannach aus: Programme werden auf Normalformen reduziert ohne Axiomatischen Definitionen zu folgen. (Turner 2009) argumentiert, das hier an der falschen Stelle nach Axiomen gesucht wird: letztere sind nicht in den interpretierten Konstanten, sondern in den Auswertungsregeln, in der Evaluationstheorie die von downarrow gegeben wurde.

Wenn es also richtig ist, dass sowohl die operationale als auch die denotationale Semantik eine axiomatisierung (der Sprache???) darstellen, ist es, solange beide in ihren Axiomenmengen übereinstimmen egal sein welche Form der Semantik wir als formale mathematische Theorie der Sprache betrachten.

Leider stimmten beide Konzepte nicht überein: Das Konzept der Äquivalenz ist im operationalen Ansatz feinstufiger (Halteproblem).

Dies hat zu einigen speziellen Spieltheoretischen Denotationalen Semantiken geführt. Von Praktikern wird der operationale Ansatz als fundamentaler betrachtet, dem trägt die Tatsache Rechnung, dass denotationale Semantiken erst später gesucht werden und mit den operationalen Ansätzen übereinstimmen müssen (cartesian closed categories für das Lambdakalkül, ...). Es gibt damit keinen metaphysischen Unterschied ziwschen dem Mengentheoretischen Ansatz und dem Operationalen, aber der Operationale wird als der definitive angesehen. (Warum benötigt man denotationale Semantiken? kann man damit auch etwas beweisen oder gehts nur drum diese Interpretations lücke zu schließen? ist dass dann ein Isomorphismus? gelten alle Aussagen der denotationalen Semantik nach Rückabbildung wieder in der Programmiersprache? WARUM IST DIE DENOTATIONALE SEMANTIK SO EIN WICHTIGES WERKZEUG? Vermutlich wird das hier noch weiter am mathematischen Korrektheitsbegriff und der Programmontologie erörtert.) Diese Sicht der Programmiersprachen sit die Sicht der Informatik: Programmiersprachen via ihrer operationalen Definition sind mathematische Theorien der Berechnung (Computation).

Programmiersprachen sind in der Regel Werkzeuge, keine eleganten mathematischen Theorien; es ist (oftmals) sehr schwer sie mit mathematischen Mitteln zu erforschen. Verhindert dies, dass sie als mathematische Theorien aufgefasst werden können? (Ich schätze der Gute redet hier von sochen Sachen wie Java. ausserdem, was ist mit Turingmaschinen und dem LK? Das sind doch sicherlich mathematische Theorien. Achtung: der macht hier mathematische Theorie an axiomatisierbarkeit fest.) Haskell ist in diesem sinn ja auch keine Implementierung des Lambdakalküls wegen seq und so weiter. ...)

Die Implementierung von Programmiersprachen

Turner 2009: Programmiersprachen sind nicht nur mathematische Objekte; sie sollten am besten als technische Artefakte verstanden werden.

Während ihre axiomatische Definition ihnen Funktionalität gibt, brauchen sie auch eine Implementierung.

In der Sprache der technischen Artefakte muss die strukturale Beschreibung einer Sprache beschreiben wie das realisiert wird: sie muss genau erklären wie die Konstrukte der Sprache implementiert werden.

Beispiel: Zuweisungsoperation

```
x := E
```

Eine physikalische Implementierung kann die folgende Form annehmen: * Berechne (physikalisch) den Wert E. * Lege das (physikalische (Terminal)Symbol) des Wertes E an der physikalischen Position x ab.

Das ist eine Beschreibung wie Zuweisung physikalisch realiziert werden kann. Es ist eine physikalische Beschreibung des Auswertungsprozesses.

Dabei wird von vielen Details der tatsächlichen Maschine abstrahiert: das wäre Bestandteil der strukturellen Beschreibung des zu Grunde liegenden Rechners als Implementierungsmedium.

Turner schließt daraus, dass eine Programmiersprache ein komplexes Paket von syntax/semantics (Funktion) sind zusamment mit ihrer Implementierung als Struktur (???).

Einige sind der Meinung, das die physikalische Implementierung die Semantik einer Sprache definiert. (Verbreitete Ansicht) (wie ist das dann mit dem Korrektheitskriterium?).

gleiche Kritik wie kausale Funktionstheorie.

Die Ontology von Programmen

Was für eine Art von Ding? Sind die mathematische/symbolische Objekte oder konkrete physikalische Dinge? (Es gibt sogar Ansätze, dass Programme eine neue Art von Entität darstellt)

Die genaue Natur von Programmen ist schwer zu ergründen. Einerseits sind sie verwand mit technologieschen Gegenständen. Andererseits können sie nur schwer mit üblichen Erfindungen verglichen werden. Sie beinhalten weder physikalische Prozesse, noch physikalische Produkte, sondern Methoden der Organisation und Verwaltung.

Sie erinnern diesbezüglich an literarische Werke obwohl sie an Maschinen addressiert sind.

(Duale Natur (abstrakt, konkret) der Ontologie)

Programme als mathematische Objekte (Type?)

Was ist der Inhalt der Behauptung, dass es sich bei Programmen um mathematische Objekte handelt? In der Rechtsliteratur scheint das Konzpet vorzuherrschen dass es sich bei Programmen um symbolische Objekte handelt die manipuliert werden können.

Wichtigstes Argument dafür, dass es sich bei Programmen um mathematische Objekte handelt; dies betrifft ihre Semantik in der Programmiersprachen wie axiomatische Theorien behandelt werden. Aus dieser Sicht sind Programme Elemente in einer Berechnungstheorie (Theorie of Computation).

(Schwierige Situation in der Gesetzgebung – kein Copyright auf rein abstrakte Entitäten.)

Programme als technische Artefakte (Token?)

Obwohl es Übereinstimmung gibt, dass Programme eine abstrakte Erscheinung/Form haben, haben sie auch eine konkrete physikalische Manifestation welche ihre Verwendung als 'Ursachen von Berechnungen' vereinfacht.

Abstrakt und Konkret

Angenommen Programme seien dualer Natur. Welche Beziehung herrscht zwischen diesen beiden Existenzformen?

Vorschlag 1: Programme, als textuelle Objekte, verursachen mechanische Prozesse. Die Idee scheint das das textuelle Objekt irgendwie den machanischen Prozess verursacht.

Diese Idee stößt aber auch auf Widerspruch (Colburn): Nicht der symbolische Text selber löst einen mechanischen Prozess aus, es ist die physikalische Manifestation, die diesen Effekt hat. Software ist eine konkrete Abstrakion welche ein Beschreibungsmedium hat (der Text, die Abstraktion) und ein Medium auf dem es ausgeführt wird (eine konkrete Implementierung in Halbleiterelementen). (Siehe Dualismus)

Vorschlag 2 (Fetzer): Abstrake Programme sind so etwas wie eine wissenschaftliche Theorie: Ein Programm kann als die Theorie seiner Implementierung angesehen werden – Programme als kausale Modelle. (Problem: Theorie ist deskriptiv, Vertauschung von Ursache und Wirkung ???)

Vorschlag 3: Ein abstraktes Programm (festgelegt durch seine Semantik) legt die Funktion des Artefakts fest.

Die Funktion des Programms, ausgedrückt in seiner Semantik, welche die physikalische Implementierung festlegt und ein Kriterium der Korrektheit und Fehlerhafitgkeit festlegt.

Programme als computationale Artefakte haben damit sowohl einen abstrakten Aspekt der festlegt was sie machen und einen physikalischen Aspekt der Ihnen hilft physikalische Dinge zu verursachen.

Programme und Spezifikation

Was ist der Unterschied zwischen Programmen und Spezifikation? Vorschlag: Spezifikation klärt das 'Was' aber nicht das 'Wie' (Unterschied deklarative vs imperative Programmiersprache).

Beispiel: Spezifikation in VDM SQRTP (x:real, y:real) Pre: x >= 0 Post: y*y = x and y >= 0

This is the spezifikation of the square root function with the precondition that the input is positive. Es handelt sich um eine funktionale Beschreibung: Sie

beschreibt *was* getan wird aber nicht *wie*. Unterscheidung: deskriptive vs imperative Programmierung.

Programme sind imperativer Natur und beschreiben das 'wie', Spezifikationen sind deklarativer Natur und beschreiben das 'was'.

Diese Unterscheidung ist aber nur für das imperative Programmierparadigma zulässig. Auf logische oder funktionale Programmiersprachen trifft es nicht zu. In der Praxis: Ein in Haskell geschriebenes Programm kann als Spezifikation für ein in C geschriebens Programm dienen.

Andere Unterscheidung: Die Beeinflussungsrichtung – welche Seite ist die normative. (Wieder eine Reihe von Paaren).

Korrektheit

Wie kann die Korrektheit eines Programms festgelegt werden? Das grundlegende Problem geht von der dualen Natur von Programmen aus, und was Korrektheit bei Programmen genau bedeutet und im Bezug auf was.

Wenn ein Programm ein mathematisches Ding ist hat es mathematische Eigenschaften wenn es ein physikalisches Ding ist dann physikalische.

Mathematische Korrektheit

Tony Hoare: Proponent der mathematischen Perspektive – Korrektheit ist eine mathematische Sache. Darzulegen, dass ein Programm korrekt gemäß einer Spezifikation ist verlangt nur nach einem mathematischen Beweis.

```
Programmieren ist eine exakte Wissenschaft in dem Sinn,
dass alle Eigenschaften eines Programms und alle Konsequenzen seiner
Ausführung prinzipiell durch logisches Schlussfolgern aus dem Text
geschlossen werden können.
```

Beispiel: Spezifikation der Quadratwurzelfunktion Was bedeutet es wenn ein Programm sie erfüllt?

Mit Bezug auf seine abstrakte Semantik generiert eine Programm eine Relation Rp zwischen seinem Input und seinem Output: Seine Extension.

Die Korrektheitsbedingung fordert, dass diese Relation die oben genannte Spezifikation erfüllt, was wiederum zur Folge hat, dass das Programm modulo seiner semantischen Interpretation die Spezifikation erfüllt.

Die Spezifikation C ist eine Beziehung zwischen zwei formalen Objekten und prinzipiell kann die Korrektheit damit mathematisch festgelegt werden.

Im Bezug auf die mathematische Gestalt eines Programms lässt sich diese Auffassung von Korrektheit kaum in Frage stellen. Es gibt aber die weitere, im Folgenden aufgelistete Einwände.

Die Herausforderung der Komplexität

```
Programmierer sind immer von Komplexität umgeben; wir können sie nicht
vermeiden. Unsere Anwendungen sind komplex da wir unsere Computer auf
immer kompliziertere Weise verwenden wollen.
Programmieren ist komplex, da es eine große Menge von sich im Konflikt
befindlichen Zielen bei den Projekten gibt.
Wenn unser basales Werkzeug, die Sprache in der wir unsere Programme
entwerfen und schreiben auch kompliziert ist, dann wird die Sprache
selber ein Teil des Problems anstatt ein Teil der Lösung.
(Hoare 1981: 10)
```

Im geeigneten mathematischen Rahmen ist es theoretisch möglich die Korrektheit eines linguistischen Programms relativ zu seiner Spezifikation zu beweisen.

Reale Software ist aber sehr komplex. In solchen Fällen kann es aus praktischen Gesichtspunkten unmöglich sein ihre Korrektheit zu beweisen. Man könnte versuchen diese Tatsache zu entschärfen in dem man einen Theorembeweiser einsetzt; dessen Korrektheit muss allerdings auch bewiesen werden (verified tool chain).

Auch ein verifizierter Theorembeweiser löst das Problem nicht vollständig. Sowohl aus praktischen wie auch aus theoretischen Gründen ist eine menschliche Beteiligung nach wir vor vorausgesetzt (??? welche theoretischen und praktischen Gründe???). In den meisten Fällen werden Beweise von Hand unter Verwendung von Beweisassistenzsystemen erstellt. Trotzdem hilft dieser Ansatz enorm die Sicherheit von Programmen zu erhöhen.

Ob lange und teilweise mechanisch erzeugte Beweise im klassischen Sinn mathematische Beweise sein können ist Gegenstand der Debatte (Four Color Theorem).

(Methoden klassische Beweise zu strukturieren: Kompositionalistät durch Lemmata. Die Mathematik schreitet voran dadurch, dass sie neue mathematische Konzepte entwickelt durch die ehemals unmöglich zu führende und zu verstehende Beweise möglich werden. Simples Beispiel: Exponentennoation Kategorientheorie In der Mathematik geht es nicht nur um Beweise, sondern auch um die Erfindung neuer Abstraktionen und Notationen)

Anmerkung: Die Grad an Korrektheit, den eine Spezifikation vorschreibt variiert natürlich. Widerspiegelt sich durch die unterschieldiche Expressivität von Typensystemen und UML Diagramme als Beispiel. (nochmal nacharbeiten was er da will) (UML nur strukturelle Spezifikation)

Die Empirische Herausforderung

```
Das Konzept von Programmverifikation schein auf einer Doppeldeutigkeit
zu beruhen.
```

Algorithmen als logische Strukturen können Gegenstand deduktiver Verifikation sein.
Programme, als kausale Modelle dieser Strukturen sind das nicht.
Der Erfolg von Programmverifikation als eine universell anwendbare und komplett verlässliche Methode um das Verhalten eines Programmes zu garantieren ist nicht einmal eine theoretische Möglichkeit.
(Fetzer 1988: 1)

Wenn die Korrektheit eines Programms, sein Kompiler und die Korrekheit der Hardware eines Computers alle mit mathematischer Sicherheit bewiesen wurden, ist es möglich große Gewissheit in die Ergebnisse eines Programms zu legen, und ihre Eigenschaften mit einer Gewissheit vorherzusagen die nur von der verlässlichkeit der Elektronik eingeschränkt wird.
(Hoare 1969L 579)

Physikalischen Instanzen von Programmen liegen physikalische Systeme zu Grunde unvorhersehbares Verhalten kann sich aus den kausalen Zusammenhängen ergeben. Selbst wenn Theorembeweiser und Beweisassistenten verwendet werden handelt es sich nur um empirisches Wissen

(wie ja alles Wissen ist also nichts neues, auch alle mathematiker die einen bestimmten Beweis durchgearbeitet haben können sich an genau der gleichen Stelle irren oder vielleicht doch, ist doch die alte debatte mit a priori und a posteriori wissen.)

Bei der Ausführeung eines Beweises wird davon ausgegangen, dass all die relevanten zugrunde liegenden Schichten von Hard- und Software korrekt sind (Gegenentwurf: verified toolchain).

In der Regel geht die Überprüfung der Spezifikation nicht einmal so weit - Softwareentwickler führen kaum klassischen Beweise ihrer Software durch, auch Theorembeweiser kommen kaum zum Einsatz.

Das Optimum des abgleiches besteht normalerweise in der ausführung einiger Tests (Unit testing, test driven development) (zitat: can only check for presence of error not its absence) Aus einer transfinitin eingabemenge immer nur finit viele Fälle. Dies ist keine Korrektheit im mathematischen Sinn sondern nur eine Überprüfung einer kleinen Teilmenge der möglichen Eingaben.

Physikalische Korrektheit

Was bedeutet es für ein physikalisches Gerät seiner Spezifikation zu entsprechen? Wann ist ein physikalisches System eine korrekte Implementierung?

Idee: Ansatz der einfachen Abbildung (simple mapping account)

Nach dem Ansatz der einfachen Abbildung ist ein physikalisches system S einer korrekte Implementierung einer abstrakten Spezifikation C im Fall, dass 1. Es

eine Abbildung von den physkikalischen Zuständen die S einnehmen kann zu den physikalischen Zuständen die von der abstrakten Spezifikation C definiert werden, so dass 2. Die Zustandstransitionen des physikalischen Systems die Zustandstransitionen des abstrakten Systems wiederspiegeln.

(Also ein Homomorphismus (Isomorphismus??) zwischen dem physikalischen und dem abstrakten system.)

Beispiel: einfache abstrakte maschine über dem Alphabet {0,1} mit 4 zuständen: (0,0), (0,1), (1,0), (1,1) vll besser anderes Beispiel.

ERGÄNZEN

Programming

Was ist die Natur des Programmierens? Mathematische Aktivität, Wissenschaft, Ingenieurswissenschaft, oder Kunst?

Programmieren als mathematische Aktivität

Programmkonstruktion durch Ableitung von Spezifikationen auf mathematisch präzise Weise durch Programmtransformationen basierend auf algebraische Gesetzen.

Dies kann als alternativer Ansatz für das Korrektheitsproblem gesehen werden: Programme werden von ihrer Spezifikation durch wahrheitserhaltende Regeln abgeleitet.

(Beispiel: Kombinator durch Beweisverfahren, Gleichungen, ...) Forschung: * Morgan (1994): Transformationaler Ansatz für imperative Programmiersprachen * Henson (1987): Transformationaler Ansatz für funktionale Programmiersprachen * Löf (1982): Konstruktive Mathematik um Programme aus konstruktiven Existenzbeweisen abzuleiten (Curry-Howard-Isomorphismus?)

Ergänzen, könnte ganz hilfreich sein

Programmierung als Ingenieurstätigkeit (engineering design)

Programmierung in weiteren Sinn und in der Praxis befasst sich mit der Erstellung technischer Artefakte und nicht mathematischer Objekte. In diesem Bild ist die Programmierung eine Ingenieurstätigkeit.

Konzepte: * Strucutred Programming (Kompositionalität) * Data Encapsulation (Objekt-Orientierung?) vermeidung von 'leaky-modules'.

Programmierung Konstruktion von Theorien (theory construction)

Abstraktion

(Was hat das mit dem Modellentwurf zu tun?)

Abstraktion vereinfacht die Informatik. Ohne sie wäre die entwicklung von einfachen numerischen Algorithmen zur Komplexität von Luftraumüberwachungssystemen, interaktiven Beweisassistenten, und Computerspielen möglich.

Sie maniferstiert sich in den expressiven Typensystemen moderner Programmier- und Spezifikationssprachen und liegt dem Entwurf dieser Sprachen mit ihren eingebauten Abstraktionsmechanismen zu grunde.

Abstraktion steht hinter der Erfindung von Konzepten wie Polymorphie, Datenabstrakion, Klassen, Schemata, Design-Patterns, Macros, Vererbung.

Was ist die Natur von Abstraktion in den Computerwissenschaften? Hat sie nur eine Form? Ist es die selbe Art von Abstraktion die man in der Mathematik vorfindet?

Die generelle Natur von Abstraktion

Ansatz 1: Abstraktion als XXX

Skemp (1987):

```
Abstraktion ist eine Aktivität bei denen uns Gemeinsamkeiten ... in useren
Erfahrungen bewusst werden.
Klassifikation bedeutet unsere Erfahrungen gemäß dieser Gemeinsamkeiten zu
bündeln.
Eine Abstraktion ist eine Art von bleibender Veränderung, das Ergebnis von
Abstraktion, welches uns erlaubt Gemeinsamkeiten von neuen Erfahreungen mit
einer bereits bekannten Klasse von Erfahrungen zu erkennen.
(Universalien?)
Um zwischen der Abstraktion als Aktivität und Abstraktion als dem Endprodukt
zu unterscheiden nennen wir letzteres ein Konzept.
(Skemp 1987: 5)
```

Hoare (1973):

```
Abstraktion entsteht aus dem Erkennen von Gemeinsamkeiten zwischen
verschiedenen Objekten, Situationen und Prozessen in der realen Welt,
und den Entschluss sich auf diese Gemeinsamkeiten zu konzentrieren
und in diesem Kontext die Unterschiede zu ignorieren.

(Leibniz, Identitätskriterium?)
```

Abstraktion in diesem Sinne umfasst das Ignorieren konkreter Eigenschaften; aus dieser wissentlichen Unkenntniss entsteht ein neues Konzept.

(in diesem Sinn aber sehr umstritten, siehe Abstrakte Objekte weitere Ideen dazu in diesem Artikel)

Abstraktion in der Informatik

In der Informatik tritt Abstraktion in vielen verschiedenen Formen auf. (hier keine systematische Beschreibung siehe Goguen & Burstall).

Abstraktion 1: prozedurale Abstraktion (procedural abstraction)

(Anwendung: wiederhohlter Code)

Prozedurale Abstraktion: Programmtext, unter umständen parametrisiert, wird ein Name gegeben (procedural abstraktion). Durch die Formulierung einer Prozedur wird die Existenz eines neuen Objekts festgelegt; die strukturellen Gemeinsamkeit ist der gemeinsame Code.

Formal ist das die Abstraktion des Lambdakalküls. Bei Typsisierten Varianten kann der Parameter sogar ein Type sein, was zu den verschiedenen Polymorphiemechanismen führt, welche in mathematischen Theorien wie dem Lambdakalkül zweiter Stufe beschrieben werden können.

Abstraktion 2: Abstraktion vom Mechanismus (mechanical Abstraktion)

Abstraktion von den unterliegenden Mechanismen der ausführenden Maschine. Erlaubt das Erstellen komplexer Lösungen ohne genaue Kenntnis von den Operationen der Maschine zu haben.

Beispiel: Rekursion (Tail Call Optimization?) Rekursion implementiert als Stack; der Benutzer muss nichts über die Implementierung wissen.

Abstraktion 3: Ontologische Abstraktion (type abstraktion)

Die Typenstruktur einer Programmiersprache legt die Ontologie der Sprache fest: die Arten von Entitäten die für die Formulierung und Lösung von Problemen verwendet werden können. Zu einem großen Teil legen Typen den Abstraktionsgrad einer Sprache fest. Ein große Menge von Typenkonstruktoren erlaubt ein expressives Repräsentationssystem. Abstrakte und Rekursive Typen sind ein häufiges Beispiel.

Abstraktionsebenen (Stratifizierung von Abstraktionen)

In der Mathematik, Philosophie und der Informatik gibt es Abstraktionsebenen;

Verstecken von Information (Information Hiding)

These 1: (Abstraktion in der klassischen Mathematik) Sobald in der Mathematik eine Abstraktion begründet wurde wird ihre physikalisches Vorbild zurückgelassen.

Unter dieser Ansatz ist eine Abstraktion abgeschlossen: Ein abstraktes mathematisches Objekt bezieht seine Bedeutung nur aus dem System innerhalb dessen es definiert worden ist. Abgeschlossenheit ist von größter Wichtigkeit. Es gibt keine Bezüge nach außen. (siehe Hilbert zum Axiomatischen Ansatz).

Damit begründen einige Philosophen, dass Abstraktion in der Informatik grundsätzlich verschieden von der Abstraktion in der Mathematik ist.

Sie behaupten, dass computationale Abstraktion eine 'Implementierungsspur' hinterlassen muss (Implementation trace); die Information ist versteckt aber nicht zerstört. Jedes Detail, dass auf einer Ebene der Abstraktion nicht bedacht wurde muss in einer anderen Ebene berücksichtigt werden. Auf jeder Ebene hängen Computationale Artefakte von Implementierungen ab. Das stimmt mit der Sichtweise überein, dass computationale Artefakte sowohl Struktur als auch Funktion haben: sie haben eine abstrakte als auch eine konkrete Gestalt in Form ihrere Implementierung.

Diese Trennung ist allerdings nicht so eindeutig wie es auf den ersten Blick scheint ... Auch in der Informatik werden Objekte erzeugt, die ihre Bedeutung nur aus ihrer Beziehung zu anderen Objekten in dem System beziehen.

Abstrakte Konzepte könnten keine normative Funktion haben ohne das ihnen solche unabhängigen Bedeutungen zukommen könnten.

Weiter gilt das oben gesagte nur für die klassische Mathematik; Die konstruktive Mathematik ist der Informatik in dem Sinn Ähnlich als das es auch hier eine 'Implementierungsspur' geben muss: Die details der Implementierung können aus den Beweisen entnommen werden. (???)

Weiteres Argument: mathematische Abstraktionen lassen ihre physikalischen Wurzeln nicht komplett hinter sich. ERGÄNZEN

Dadurch erscheint es, dass eine Abgrenzung zwischen Abstraktion in der Informatik und Abstraktion in der Mathematik nicht so scharf getroffen werden kann.

Eine konzeptuelle Differenz bleibt aber bestehen: die Dominanz des Abstraktionspartners Mathematik ist dazu da die Welt zu modellieren, treffen die Prognosen nicht zu wird die Mathematik geändert; Relation: ModelTheory/Relaität In der Informatik: Relation Spezifikation/Artefakt.

Bei Fehlerhaftigkeiten: In der Mathematik wird die Theorie verändert, In der Informatik die Implementierung.

Brechnung (Computation)

Die ursprüngliche Motivation für eine mathematische Analyse des Berechnungsbegriffs kam aus der mathematischen Logik. Ihre Ursprünge liegen in Hilbert's Frage nach der

`Entscheidbarkeit der Prädikatenlogik`

Gibt es eine Entscheidungsprozedur, einen Algorithmus, der für einen beliebeigen Satz der Prädikatenlogik zeigt, dass er beweisbar ist (das er ein Modell besitzt?) => Entscheidungsproblem. (ist das Halteproblem nicht nur die spezielle Formulierung des Halteproblems?).

Um diese Frage beantworten zu können musste der informale Berechenbarkeitsbegriff formalisiert/mathematisiert werden.

Der heute definitive Ansatz ist der von Turing (1936).

Die Geburt der formalen Berechnung (Computatbility)

Es gab mehrere formalisierungen des Berechnungsbegriffs: * Gödel-Herbrand (1934): Generelle Rekursive Funktionen * Church-Kleene (1931-1939): Lambda-Kalkül

Kleen konnte zeigen, das diese beiden Formalisierungen extensional gleich waren. Dies ist die Grundlage der extensional gleichen Chruchschen These: * die Berechenbaren Funktionen sind die Generellen Rekursiven Funktionen * die berechenbaren Funktionen sind die im Lambdakalkül formulierbaren Funktionen

Gödel zeigte sich von beiden Ansätzen nicht überzeugt. Er akzeptierte nicht, dass es sich um die intentional richtige formalisierung einer mechanischen Methode handelte. Sie waren nur extensionale ??? Charakterisierungen mit wenigt motivationaler Rechtfertigung.

Turings Vorschlag reifizierte ein abstraktes Berechnungsmodell – die Turingmaschinen – und schien damit begründet den richtigen Begriff einzufangen. Weiter konnte Turing zeigen, das sein Berechnungbegriff und das Lambdakalkül extensionsgleich waren.

Turings Version der These (Turingthese): Berechenbare Funktionen sind von Turingmaschine ausführbar.

Alle Programmiersprachen sind Turing-Berechenbar; general purpose programmiersprachen sind Turing-vollständig: sie umfassen alle kontrollstrukturen die notwendig sind eine universelle turingmaschine zu simulieren.

Was ist an Turings Vorschlag so besonders?

Die Natur der Berechnung

Turing legte große Betonung auf die Natur der basalen Anweisungen der Turingmaschine. Die folgende Ausführung legt nahe, dass die Anweisungen ohne reflexion/nachzudenken ausgeführt werden können. Atomare Operationen können auf Maschinen implementiert werden, welche die Bedeutung der Anweisungen weder kennen noch kennen können.

Ausführung ...

Beurteilungen von Ergebnissen sind einer Maschine nicht möglich. Berechnungen haben dann rein syntaktische Natur.

Berechnungen als Vorschriften Vorschriften müssen auch von einer Maschine interpretierbar sein.

Sprache berschreibt eine mechanische Berechnung; das ist auch das imperative Paradigma.

(Gewonnen hats weils intuitiver war als das Lambda-kalkül)

Gibt es so etwas wie bedeutungslose Anweisungen oder Operationen in einer Programmiersprache. Ist es möglich, dass wir einer Anweisung folgen ohne seine Bedeutung zu rechtfertigen? Tragen diese Anweisungen wirklich keine Bedeutung?

Diese Vorschrift ist alles nur nicht bedeutungslos. Sie ist so einfach, dass man sich einen Agenten vorstellen kann der sie mechanisch vornimmt (...) Unabhängig davon wie simpel diese Regel ist, sie sagt einem Trotzdem was zu tun ist.

Compositionality

Es besteht ein offensichtlicher Konflikt zwischen der Auffassung, das Anweisung keine Bedeutung tragen und dem Kompositionalitätsprinzip. In seiner modernen Form wurde das Kompositionalitätsprinzip von Frege formuliert:

```
Die Möglichkeit Sätze zu verstehen, die wir vorher noch nie gehört haben
beruht darauf, das wir denn Sinn eines Satzes aus dem Sinn seiner
Konstituenten bestimmen können. (Frege 1914)
```

Kompetente Sprecher können Sätze verstehen die sie noch nie zuvor gehört haben. Nach Proponenten ist das nur möglich wenn sie die struktur des Ausdrucks sowie die Bedeutung seiner Konstituenten berücksichtigen.

Für Programmiersprachen bedeutet dies, dass ohne Verständnis ihrer atomaren Anweisungen Auch komplexere Ausdrücke keine Bedeutung hätten.

Die physikalische Turing-These

Viele nehmen an, dass die Church-Turing-These die tatsächliche physikalische Berechnung beschreibt. Dabei wird zwischen einer reinen mathematischen Interpretation einer berechenbaren Funktion und einer physikalischen Interpretation unterschieden. Das Konzept soll beide erfassung.