



Berechenbarkeit, Entscheidbarkeit, Aufzählbarkeit

Vorlesungsskript SS 2011

© Christoph Walther

Fachgebiet Programmiermethodik
Fachbereich Informatik
Technische Universität Darmstadt

29. März 2011

INHALTSVERZEICHNIS

| | | |
|----------|---|-----------|
| 1 | Einleitung | 5 |
| 2 | Rechner, Algorithmen, Programme | 10 |
| 2.1 | Rechner und Programmiersprachen | 10 |
| 2.2 | Die Programmiersprache \mathcal{P} | 12 |
| 2.3 | Semantik von \mathcal{P} -Programmen | 21 |
| 3 | Berechenbare Funktionen | 26 |
| 3.1 | \mathcal{P} -berechenbare Funktionen | 26 |
| 3.2 | Die Churchsche These | 27 |
| 3.3 | Nicht-Konstruktivität und Konstruktivität | 29 |
| 4 | Gödelisierungen | 32 |
| 4.1 | Kodierung von Datentypen | 32 |
| 4.2 | Die Paar-Funktion | 34 |
| 4.3 | Stelligkeit berechenbarer Funktionen | 39 |
| 4.4 | Kodierung von Programmen | 41 |
| 5 | Algorithmisch unlösbare Aufgaben | 45 |
| 5.1 | Abzählbarkeit der \mathcal{P} -berechenbaren Funktionen | 45 |
| 5.2 | Nicht-berechenbare Funktionen | 47 |
| 5.3 | Das Halteproblem | 49 |
| 6 | Universelle Funktion und Interpretierer | 51 |
| 7 | Entscheidbarkeit und Semi-Entscheidbarkeit | 53 |
| 7.1 | Entscheidbare Probleme | 53 |
| 7.2 | Endliche Probleme | 55 |
| 7.3 | Semi-entscheidbare Probleme | 56 |

| | | |
|-----------|---|------------|
| 8 | Schrittfunktionen | 61 |
| 8.1 | Die Schrittfunktion eines \mathcal{P} -Programms | 61 |
| 8.2 | Die universelle Schrittfunktion | 64 |
| 8.3 | Schrittweise Abarbeitung von \mathcal{P} -Programmen | 66 |
| 9 | Rekursive Aufzählbarkeit | 70 |
| 9.1 | Aufzählungsverfahren | 70 |
| 9.2 | Abzählbarkeit und rekursive Aufzählbarkeit | 76 |
| 9.3 | Definitions- und Bildbereiche berechenbarer Funktionen | 76 |
| 10 | Unentscheidbare Probleme | 79 |
| 10.1 | Das Totalitätsproblem | 79 |
| 10.2 | Reduzierbarkeit | 80 |
| 10.3 | Das <i>s-m-n</i> -Theorem | 82 |
| 10.4 | Programmäquivalenz | 85 |
| 10.5 | Der Satz von <i>Rice</i> | 88 |
| 10.6 | Weitere Probleme | 90 |
| 11 | Primitiv-rekursive und μ-rekursive Funktionen | 93 |
| 11.1 | Primitiv-rekursive Funktionen | 93 |
| 11.2 | Definitionsprinzipien für primitiv-rekursive Funktionen | 96 |
| 11.3 | Werteverlaufsrekursion | 100 |
| 11.4 | Ein Rechenmodell für primitiv-rekursive Funktionen | 102 |
| 11.5 | loop-berechenbare Funktionen | 104 |
| 11.6 | Die <i>Ackermann</i> -Funktion | 114 |
| 11.7 | μ -rekursive Funktionen | 116 |
| 12 | Turingmaschinen | 133 |
| 12.1 | Definition und Rechenmodell | 133 |
| 12.2 | Turing-berechenbare Funktionen | 138 |
| 12.3 | Berechnungsvollständigkeit von \mathcal{P} | 140 |
| 13 | Begriffe und Schreibweisen | 146 |
| | Literaturverzeichnis | 147 |
| | Index | 148 |

Vorwort

Dieses Skript ist eine überarbeitete und erweiterte Version des Vorlesungsskripts meiner im SS 2001 erstmalig gehaltenen Vorlesung über Berechenbarkeitstheorie. Da es bei dieser Thematik um fundamentale Grundlagen geht, ist es erforderlich, mit mathematischer Präzision zu arbeiten. Dieser Forderung kann jedoch in einer Vorlesung vom Umfang 2+1 nicht durchgehend nachgekommen werden. Man muß daher an gewissen Stellen an die "Vorstellungskraft" des Auditoriums appellieren, wenn man den Stoff in beschränkter Zeit präsentieren will. Konkret bezieht sich dieser Vorbehalt auf den Nachweis der \mathcal{P} -Berechenbarkeit der universellen Funktion und der universellen Schrittfunktion. Diese Auslassungen sind zugegebenermaßen höchst unbefriedigend, aber aus Zeitgründen unvermeidbar.

Herrn Markus Aderhold – als Hörer der damaligen Vorlesung mit den Schwächen des Originalskripts bestens vertraut – danke ich für sehr sorgfältiges Korrekturlesen, Hinweise auf Fehler und konstruktive Verbesserungsvorschläge.

Bei aller Sorgfalt wird das Skript trotzdem noch Fehler enthalten.¹ Entsprechende Hinweise auf Schreibfehler und inhaltliche Schwächen nehme ich gerne unter Chr.Walther@informatik.tu-darmstadt.de entgegen.

Darmstadt, im Mai 2007

Das Skript wurde für das Sommersemester 2008 überarbeitet. Korrigiert wurden einige Schreibfehler sowie einige Ergänzungen in Definitionen eingefügt (und Beweise dann entsprechend angepaßt). Insbesondere wird jetzt für die Abarbeitung von Programmen unter einer Laufzeitbeschränkung ein einfacheres Kostenmaß verwendet.

Darmstadt, im März 2008

Das Skript wurde für das Sommersemester 2009 überarbeitet. Korrigiert wurden lediglich einige Schreibfehler.

Darmstadt, im April 2009

Das Skript wurde für das Sommersemester 2011 überarbeitet. Neben der Korrektur einiger Schreibfehler und der Ergänzung durch Satz 9.7 wurde der Beweis von Satz 11.9 durch Verzicht auf die Werteverlaufsrekursion stark vereinfacht und die Reihenfolge einiger Abschnitte in Kapitel 11 umgestellt.

Darmstadt, im März 2011

¹ Einige Indexeinträge vermeiden Umlaute, also 'ae' anstatt 'ä' u.s.w. Dies sind weder Schreibfehler noch Ausdruck einer eigenwilligen Rechtschreibung, sondern vielmehr Notwendigkeiten, um Fehler meiner \TeX -Installation zu umgehen.

1. EINLEITUNG

In der Berechenbarkeitstheorie wird untersucht, welche Aufgaben durch Rechner gelöst werden können und welche Aufgaben prinzipiell nicht durch Rechner lösbar sind. Es geht hier also um die Möglichkeiten und insbesondere um die *prinzipiellen Grenzen* des Problemlösens mittels Rechnern. Die Ergebnisse der Berechenbarkeitstheorie gehören damit zum fundamentalen Grundlagenwissen der Informatik.¹ Daher sollte jeder Informatiker – unabhängig davon, ob er sich für theoretische Fragestellungen interessiert oder nicht – die essentiellen Grundvoraussetzungen seines Berufs kennen. Er sollte beispielsweise wissen,

- warum es keine gängige Programmiersprache gibt, in der nur terminierende Programme geschrieben werden können,
- warum es kein Entwicklungswerkzeug gibt, das feststellen kann, ob die optimierte Version eines Programms (beispielsweise eines Übersetzers für eine Programmiersprache) – abgesehen von verbesserter Laufzeit – das gleiche leistet wie die nicht optimierte Originalversion des Programms,
- warum es kein Testwerkzeug gibt, mit dem festgestellt werden kann, ob die Abarbeitung eines Programms mit bestimmten Eingaben terminiert oder nicht,
- warum jedes Verfahren, mit dem die Terminierung von Programmen bewiesen werden kann, beim Terminierungsnachweis unendlich vieler (terminierender) Programme versagt,
- warum es kein Verifikationswerkzeug gibt, mit dem für jedes Programm festgestellt werden kann, ob das Programm eine bestimmten Aufgabe – etwa das Sortieren von Listen, die Berechnung der Primfaktorzerlegung von natürlichen Zahlen u.s.w. – löst oder nicht löst.

¹ Historisch handelt es sich um Ergebnisse der mathematischen Logik ab den 30er Jahren des letzten Jahrhunderts, also aus einer Zeit, in der die Vorstellung programmierbarer Rechner eine “Vision” ohne jeglichen Realitätsbezug war. Die Informatik, deren “Geburt” als eigenständige Wissenschaft etwa 35 Jahre später datiert, hat diese Ergebnisse als Grundprämissen ihres Fachs übernommen.

Anworten auf diese (und weitere) Fragen findet man in diesem Skript. Dabei werden nur solche Ergebnisse der Berechenbarkeitstheorie vorgestellt, die ein Informatiker mindestens kennen sollte. Es handelt sich hier also nur um eine Einführung in das Gebiet – für Vertiefungen wird auf das Literaturverzeichnis mit einer Auswahl der umfangreichen Literatur zu diesem Thema verwiesen.

Da sich die Berechenbarkeitstheorie aus der mathematischen Logik, also einem Teilgebiet der Mathematik, entwickelt hat, ist offensichtlich, daß man hier um Beweise nicht herumkommt. Diese Beweise sind genaugenommen nicht kompliziert. Verständnisschwierigkeiten resultieren vielmehr daraus, daß die abstrakten Begriffsbildungen nicht verstanden werden. Anders gesagt, viele Studenten haben Schwierigkeiten, die mathematischen Konzepte in Bezug zu ihrer realen Informatikpraxis zu setzen, was zum Verständnis der Inhalte jedoch unabdingbar ist.

Beispielsweise werden Programme “gödelisiert”, d.h. durch natürliche Zahlen kodiert, damit man Programme wie Daten verarbeiten kann. Der Übersetzer einer Programmiersprache, der ja einen Programmtext als Eingabe erwartet, erhält in der Berechenbarkeitstheorie eine natürliche Zahl als Eingabe, die lediglich die Zahldarstellung eines Programmtextes ist. Zum Verständnis von Sätzen und Beweisen muß man deshalb diese Zahlen dann nur als *Repräsentation* von Programmtexten “lesen”. Erkennt man jedoch diesen Bezug nicht, so kann man auch die entsprechenden Sätze und schon gar nicht deren Beweise verstehen.

Um Aussagen über die Berechnungen programmierbarer Rechner zu erhalten, muß ein Maschinenmodell zugrunde gelegt werden. Der überwiegende Teil der Lehrbücher dieses Gebiets verwendet dazu *Turingmaschinen*. Dies hat historische Wurzeln, denn in einer Zeit, in der es keine programmierbaren Rechner (geschweige denn Programmiersprachen) gab, mußte man sich mit dem mathematischen Modell eines Rechners – der Turingmaschine – begnügen. Die Turingmaschine besitzt alle Merkmale eines Rechners (Programm- und Datenspeicher, Rechenwerk), und folglich sind alle Ergebnisse, die man über Turingmaschinen erhält, auf beliebige Rechner mit beliebigen Betriebssystemen und beliebigen Programmiersprachen übertragbar.²

Allerdings ist die Programmierung von Turingmaschinen äußerst aufwendig, und Turingprogramme sind nur mit großer Mühe zu verstehen. Folglich steht man bei Verwendung von Turingmaschinen vor einem Dilemma: Entweder verwendet man in Beweisen andauernd Formulierungen der Form “*wie man sich leicht vorstellen kann, gibt es eine Turingmaschine die Dies-und-Das leistet*”, ersetzt Beweise durch Graphiken und andere Illustrationen u.s.w. und appelliert

² Anstatt ein JAVA Programm unter *Windows XP* auf einem INTEL-Chip auszuführen, kann man also genausogut (die Hardwareimplementierung) einer Turingmaschine verwenden. Aus Effizienzgründen sollte man dies jedoch tunlichst unterlassen.

damit ständig an die Vorstellungskraft der Studenten, oder aber man entwickelt eine Programmiermethodik für Turingmaschinen, mittels derer dann Beweise mathematisch präzise angegeben werden können, d.h. so, wie es in einem Gebiet, in dem es um fundamentale Grundlagen geht, eben erforderlich ist. Mit der zweiten Lösung erhält man zwar die erforderliche mathematische Präzision, allerdings zu einem hohen Preis: Zum einen müssen die Studenten in der Programmierung von Turingmaschinen ausgebildet werden – eine Fähigkeit, die, abgesehen von Bearbeitung der Übungsaufgaben, völlig nutzlos ist. Zum anderen werden die entscheidenden Beweisideen und die Kernaussagen des Gebiets ständig durch die Turingprogrammierung verschleiert und schlimmstenfalls sogar verdeckt. Sätze und deren Beweise werden dadurch genauso “barock” wie die Turingprogramme selbst.

Aber müssen überhaupt Turingmaschinen verwendet werden? Um die zentralen Ergebnisse der Berechenbarkeitstheorie zu erhalten, ist ein Maschinenmodell erforderlich, bei dem Programme Programme verarbeiten können. Dies ist jedoch schon bei Programmierung in irgendeiner Assembler- bzw. Maschinensprache möglich, und folglich verwenden einige Lehrbücher eine sogenannte *Registermaschine* oder *random access machine (RAM)* als Modell eines Rechners. Zwar stellt die Assemblerprogrammierung einen großen Fortschritt gegenüber der Turingprogrammierung dar, sie besitzt im Prinzip jedoch die gleichen Nachteile wie die Turingprogrammierung, allerdings in stark abgeschwächter Form.

Da jedoch auch Programme einer höheren Programmiersprache Programme (dieser Programmiersprache) verarbeiten können (etwa ein in C geschriebener C-Übersetzer oder ein in JAVA geschriebener JAVA-Interpreter), kann man genausogut eine höhere Programmiersprache als Ausgangspunkt wählen. In einigen Programmiersprachen, wie etwa LISP, werden Programme und Daten überhaupt nicht unterschieden, die Verarbeitung von Programmen durch Programme kann dort direkt – also ohne den Umweg über Kodierungen von Programmen – implementiert werden. Einige (wenige) Lehrbücher zur Berechenbarkeitstheorie, wie beispielweise das hier ausdrücklich zur Vertiefung empfohlene [Jones(1997)], verwenden eine solche Programmiersprache als Ausgangspunkt für ihre Präsentation.

Andere Programmiersprachen, wie etwa C oder JAVA, erlauben keine direkte Verarbeitung von Programmen. Dort geht man dann den Umweg über geeignete Datenstrukturen, indem Programme etwa als Texte (*string, array of character*) “kodiert” werden. Auch solche Programmiersprachen werden gelegentlich in Lehrbüchern verwendet, etwa in [Kfoury et al.(1982)Kfoury, Moll, and Arbib] und [Wagner(1994)]. Dies ist genauso akzeptabel wie die Verwendung einer Programmiersprache, in der Programme direkt verarbeitet werden können, denn die Repräsentation von Programmen durch Datenstrukturen der Sprache ist gängige Praxis in der Informatik.

Um das Verständnis der Vorlesungsinhalte nicht unnötig zu erschweren, verwenden wir deshalb in diesem Skript eine einfache (höhere) Programmiersprache – \mathcal{P} genannt – zur Implementierung von Algorithmen. Diese Programmiersprache enthält nur Sprachkonzepte, die man auch in jeder anderen höheren Programmiersprache findet. Damit ist jedermann, der wenigstens mit *einer* höheren Programmiersprache vertraut ist, sofort in der Lage \mathcal{P} -Programme zu verstehen und zu schreiben.

Bei Definition dieser Programmiersprache müssen wir entscheiden, wie ausdrucksstark die Sprache gewählt werden soll. Diese Frage ist deshalb von Bedeutung, da viele Beweise in der Berechenbarkeitstheorie *konstruktiv* geführt werden. Beispielsweise zeigt man, daß ein Problem durch einen Rechner lösbar ist, indem man ein Programm angibt, das die Lösung berechnet. Ebenso kann man die algorithmische Unlösbarkeit von Problemen zeigen: Man führt einen Widerspruchsbeweis und nimmt dafür die Lösbarkeit des Problems an. Damit muß es ein Programm geben, das eine Lösung berechnet. Dieses Programm verwendet man dann zur Konstruktion weiterer Programme, von denen eines offensichtlich Unmögliches leistet, und damit einen Widerspruch offenbar macht.

Kurzum, viele Beweise erfordern hier Programmierung, und je ausdrucksstärker die verwendete Programmiersprache ist, desto einfacher ist es solche Programme zu schreiben. Nur wird es dann auch aufwendiger – und bei Verwendung gängiger Programmiersprachen praktisch unmöglich – Beweise über Berechnungen von Programmen dieser Sprache zu führen.³ Dieses Problem vermeidet man bei Verwendung einer sehr einfachen Programmiersprache. Allerdings ist dann die Programmierung mühselig, und damit werden Beweise aufwendiger. Abhilfe ist hier, die sehr einfache Sprache durch weitere Sprachkonzepte ausdrucksstärker zu gestalten, wobei diese zusätzlichen Sprachkonzepte jedoch nur als abkürzende Schreibweisen für Programmfragmente der Ausgangssprache aufgefaßt werden. Damit wird die Programmierung komfortabler, und bei Beweisen kann dann je nach konkreter Aufgabenstellung frei gewählt werden, ob man die erweiterte Sprache verwendet oder aber nur die Ausdrücke der einfachen Kernsprache betrachtet.

Unter Verwendung von \mathcal{P} -Programmen führen wir nachfolgend diejenigen zentralen Grundbegriffe der Berechenbarkeitstheorie ein und beweisen die Kernaussagen des Gebiets, die jeder Informatiker kennen sollte.

³ Es reicht natürlich nicht, ein Programme nur anzugeben, sondern man muß auch zeigen, daß dieses Programm auch das Gewünschte leistet. D.h., Programme müssen *verifiziert* werden – darin steckt dann der eigentliche Beweis. Ohne Werkzeugunterstützung ist dies bei gängigen Programmiersprachen jedoch ein hoffnungsloses Unterfangen.

Daran anschließend betrachten wir mit den μ -rekursiven Funktionen ein alternatives Berechenbarkeitsmodell, das man als eine Art *funktionale Programmiersprache* auffassen kann. Die Untersuchung des Zusammenhangs zwischen μ -rekursiven Funktionen und (imperativen) \mathcal{P} -Programmen sowie darauf aufbauend der Äquivalenznachweis beider Berechenbarkeitsmodelle bildet den Kern dieses Kapitels.

Abschließend werden Turingmaschinen sowie ihr Zusammenhang mit \mathcal{P} -Programmen betrachtet, denn dieses Berechenbarkeitsmodell ist nach wie vor der de-facto Standard des Gebiets (und verwandter Gebiete, wie etwa der Komplexitätstheorie), und sollte daher jedem Informatiker in seinen Grundzügen geläufig sein.⁴

⁴ Dieses Kapitel steht nicht im Widerspruch zu den zuvor geäußerten Vorbehalten, denn diese beziehen sich ja nicht auf die Turingmaschine an sich, sondern lediglich auf die Didaktik, die die Turingmaschine zum Ausgangspunkt der Einführung in die Grundbegriffe der Berechenbarkeitstheorie macht.

2. RECHNER, ALGORITHMEN, PROGRAMME

2.1. Rechner und Programmiersprachen

Die einfachste Methode nachzuweisen, daß eine Aufgabe mittels eines Rechners lösbar ist, besteht darin, in irgendeiner Programmiersprache ein Programm zu schreiben, das diese Aufgabe löst. Wie soll man aber nun von einer Aufgabe zeigen, daß sie prinzipiell nicht mit Rechnern lösbar ist? Dies zu behaupten bedeutet ja, daß wir nicht nur eine Aussage über alle konkreten Rechner und alle bislang definierten Programmiersprachen treffen, sondern sogar über alle Rechner, die überhaupt noch nicht entworfen wurden und über alle Programmiersprachen, die man sich irgendwann einmal ausdenken kann.

Ausgangspunkt der Untersuchungen ist der Begriff des *Algorithmus*. Darunter soll allgemein ein *Verfahren* verstanden werden, das *automatisch* ausgeführt werden kann. Soll ein Algorithmus auf einem *Rechner* ausgeführt werden, so muß dieser Algorithmus in einer *Programmiersprache* repräsentiert, d.h. aufgeschrieben, werden, deren *Programme* durch den betreffenden Rechner *ausführbar* sind. Ein *Programm* ist also nur eine *Darstellung* eines bestimmten *Algorithmus* in einer bestimmten *Programmiersprache*.

Beispielsweise kann man Felder (“*arrays*”) mit dem *Quicksort*-Algorithmus ordnen. Alternativ können zur Lösung dieser Aufgabe auch andere Algorithmen, wie z.B. *Mergesort*, *Heapsort*, *Bubblesort* u.s.w. verwendet werden. Der Unterschied zwischen diesen Algorithmen besteht darin, daß sie die gestellte Aufgabe (“*ordne ein Feld*”) in unterschiedlicher Art lösen.

Sollen Felder durch einen Rechner geordnet werden, so wählen wir zunächst eine Programmiersprache wie etwa LISP, JAVA, C, C⁺⁺, ... , die auf dem betreffenden Rechner ausführbar ist, und *implementieren* dann den Sortieralgorithmus unserer Wahl, also z.B. *Quicksort*, in der gewählten Programmiersprache. Ergebnis dieser Bemühung ist dann beispielsweise eine *Implementierung* von *Quicksort* in JAVA, die jetzt auf dem gewählten Rechner ausgeführt werden kann.

Mit der Implementierung eines Sortierverfahrens in einer ausführbaren Programmiersprache kann man offensichtlich davon ausgehen, daß das Sortieren von Feldern zu denjenigen Aufgaben gehört, die mittels Rechnern lösbar sind. Jedoch

stellt sich die Frage, was allgemein unter “Ist Aufgabe x durch einen Rechner lösbar?” genau zu verstehen ist.

Mit Definition einer Programmiersprache muß nicht nur deren *Syntax* angegeben werden, sondern auch deren *Semantik*. Mit der Definition der Semantik wird festgelegt, was die Abarbeitung eines Programms der Programmiersprache auf einem Rechner bewirken soll. Dabei wird von konkreten Rechnern abstrahiert: Wird ein Programm auf einem (beliebigen) Rechner ausgeführt, so werden die Speicherinhalte des Rechners verändert. Die “Wirkung” der Abarbeitung eines Programms kann damit abstrakt als Veränderung des initialen Speicherinhalts zu dem Speicherinhalt, der nach Abarbeitung des Programms vorliegt, beschrieben werden. Mit diesem Ansatz müssen wir nicht mehr *alle* Rechner in ihrer Vielseitigkeit und Unterschiedlichkeit betrachten: Jeder Rechner besitzt einen Speicher, der verändert werden kann, und dies reicht uns zur Modellierung *aller* Rechner.

Allerdings haben wir mit Definition der Semantik einer Programmiersprache implizit weitere Annahmen über Rechner getroffen, denn wir fordern ja, daß Rechner auch das *leisten können*, was wir durch die Semantik festgelegt haben. Man muß sich hier also einigen, welche Fähigkeiten von einem Rechner mindestens zu verlangen sind. Dies geschieht dadurch, daß gleichsam eine Art “Standardrechner” definiert wird, dessen Fähigkeiten der Maßstab über alle Aussagen ist, die wir über *beliebige* Rechner treffen. Ein allgemein akzeptierter “Standardrechner” ist die *Turingmaschine*. Kann man nun zeigen, daß eine Programmiersprache (mit gegebener Semantik) diesem Standard genügt, so ist mit dem Nachweis der nicht-Programmierbarkeit (bzw. der Programmierbarkeit) eines Lösungsverfahrens für ein Problem in dieser Programmiersprache zugleich die prinzipielle Unlösbarkeit (bzw. Lösbarkeit) des Problems durch alle Rechner, die dem Turing-Standard genügen, gezeigt. Es bleibt dann nur noch nachzuweisen, daß die Programmiersprache (mit gegebener Semantik) auch tatsächlich dem Turing-Standard genügt. Dies geschieht dadurch, indem man einen Interpreter für eine Turingmaschine in der gegebenen Programmiersprache implementiert. Gelingt dies, so können alle Berechnungen einer Turingmaschine auch durch Programme der gegebenen Programmiersprache durchgeführt werden. Eine Programmiersprache, in der man eine Turingmaschine implementieren kann, wird auch *Turing-vollständig* oder allgemeiner *berechnungsvollständig* genannt.

Hat man nun von einem Problem nachgewiesen, daß kein Programm einer *bestimmten* berechnungsvollständigen Programmiersprache ein Lösungsverfahren für dieses Problem implementiert, so weiß man, daß dieses Problem durch kein Programm *irgendeiner* Programmiersprache gelöst werden kann. In so einem Fall geht man davon aus, daß das Problem prinzipiell nicht mit Rechnern lösbar ist.¹

¹ Bei dieser Schlußfolgerung unterstellt man implizit, daß kein Rechner *mehr* leisten kann

2.2. Die Programmiersprache \mathcal{P}

Mit den Überlegungen des letzten Abschnitts können wir die Frage “Ist Aufgabe x durch einen Rechner lösbar?” mit “nein” beantworten, wenn wir nachweisen können, daß kein Lösungsverfahren für die Aufgabe x als Programm einer berechnungsvollständigen Programmiersprache implementiert werden kann.² Da wir zur Untersuchung der Unlösbarkeit von Aufgaben durch Rechner nur *eine* berechnungsvollständige Programmiersprache betrachten müssen, ist jetzt eine solche Programmiersprache auszuwählen.

In einem ersten Schritt abstrahieren wir von einer konkreten Programmiersprache, denn es ist offenbar für die weiteren Überlegungen unerheblich, ob wir einen Algorithmus in LISP, JAVA, PASCAL, C, C++, ... oder irgendeiner anderen Programmiersprache implementieren. Wir verwenden statt dessen eine einfache Programmiersprache \mathcal{P} , die für unsere Untersuchungen ausreicht. Damit abstrahieren wir von den vielfältigen und reichhaltigen Sprachkonzepten gängiger Programmiersprachen, die für die Programmierung zweifellos nützlich sind, aber unsere Untersuchungen aufwendiger machen würden. Anders gesagt, man kann in den gängigen Programmiersprachen sicher elegantere und “bessere” Programme schreiben als in der von uns verwendeten Programmiersprache \mathcal{P} . Da dieser Aspekt jedoch hier keine Rolle spielt, können wir uns auf eine einfache Programmiersprache beschränken.

Definition 2.1. (Programmiersprache \mathcal{P})

Die Programmiersprache \mathcal{P} ist wie folgt definiert:

- Der einzige Datentyp in \mathcal{P} ist die Menge \mathbb{N} der natürlichen Zahlen. Dabei sind natürliche Zahlen beliebiger Größe erlaubt.
- Ein \mathcal{P} -Programm ist eine \mathcal{P} -Prozedur, die wir in der Form

`procedure PROC(x_1, \dots, x_k) \Leftarrow PROCBODY`

angeben. Dabei sind x_1, \dots, x_k paarweise verschiedene Bezeichner für die formalen Parameter der Prozedur und PROCBODY ist der Rumpf der Prozedur.

als die Turingmaschine. In Abschnitt 3.2 wird die Gültigkeit dieser Annahme diskutiert.

² Um die *Lösbarkeit* einer Aufgabe festzustellen, reicht offenbar die Angabe eines Lösungsverfahrens als Programm *irgendeiner* – also nicht notwendigerweise berechnungsvollständigen – Programmiersprache.

- Der Prozedurrumpf *PROCBODY* einer Prozedur ist ein Block, d.h. ein Ausdruck der Form

`begin var y_1, \dots, y_n ; STA; return(y_i) end`

wobei `var y_1, \dots, y_n` eine nicht-leere Folge von Deklarationen lokaler Variabler gefolgt von einer Programmanweisung `STA` und einer (der deklarierten) lokalen Variablen `y_i` (als Argument von `return`) ist.³ Dabei müssen alle lokalen Variablen einer solchen Deklarationsfolge verschieden bezeichnet sowie verschieden von den formalen Parametern sein. Formale Parameter und lokale Variable werden auch als Programmvariable bezeichnet.

- Als Programmausdrücke `EXPR` sind erlaubt:

- natürliche Zahlen `0, 1, 2, ...`,
- formale Parameter der Prozedur,
- lokale Variable der Prozedur,
- Ausdrücke der Form `SUCC(EXPR')`, sowie
- Ausdrücke der Form `PRED(EXPR')`,

wobei `EXPR'` jeweils ein Programmausdruck ist.⁴

- Als Programmanweisungen verwenden wir

- die Leeranweisung `SKIP`,
- die Zuweisung `y := EXPR`, wobei `EXPR` ein Programmausdruck und `y` eine lokale Variable ist,
- die bedingte Anweisung `if EXPR then STA1 else STA2 end_if`, wobei `EXPR` ein Programmausdruck und `STA1, STA2` Programmanweisungen sind,
- die Hintereinanderausführung von Anweisungen `STA1;STA2`, wobei `STA1` und `STA2` Programmanweisungen sind, sowie

³ Der Ausdruck `return(y_i)` dient lediglich zur Angabe der lokalen Variablen, in der das Ergebnis einer \mathcal{P} -Prozedur abgespeichert ist. Dabei wird `return` nur aus Lesbarkeitsgründen verwendet, man könnte genausogut eine lokale Variable ohne `return` zu Angabe des Ergebnisses erlauben.

⁴ `SUCC` steht für die Nachfolgerfunktion (engl. *successor*) und `PRED` steht für die Vorgängerfunktion (engl. *predecessor*).

- die Schleifenanweisung `while` `EXPR` `do` `STA` `end_while`, wobei `EXPR` ein Programmausdruck und `STA` eine Programmanweisung ist.

\mathcal{P} ist die Menge aller \mathcal{P} -Programme, und $\mathcal{P}[k] \subseteq \mathcal{P}$ ist die Menge aller \mathcal{P} -Programme mit k formalen Parametern. ■

Abbildung 2.1 zeigt ein \mathcal{P} -Programm, also einen *Text* aus $\mathcal{P}[2]$, der die syntaktischen Forderungen von Definition 2.1 erfüllt.

Da wir die Programmiersprache \mathcal{P} im folgenden häufig verwenden, führen wir einige zusätzliche Programmkonstrukte ein, die die Programmierung vereinfachen und \mathcal{P} -Programme lesbarer gestalten. Dabei stellen diese Erweiterungen lediglich *abkürzende Schreibweisen* dar. Wir geben jeweils an, was genau durch ein zusätzliches Programmkonstrukt abgekürzt wird, und definieren damit die Semantik, d.h. die Bedeutung, dieser neuen Ausdrücke durch Ausdrücke der Originalsprache.

Definition 2.2. (Erweiterung 1 der Programmiersprache \mathcal{P})

1. Wir erlauben Blöcke auf der rechten Seite von Zuweisungen, d.h. wir dürfen auch

```

procedure P'(x1, ..., xk) <=
begin var y1, ..., yn, y;
...
y := begin var y'1, ..., y'm; STA; return(y'j) end;
...
return(...)
end

```

schreiben, wobei `STA` eine Programmanweisung ist, in der die im Block deklarierten lokalen Variablen y'_1, \dots, y'_m sowie Programmvariable des umgebenden Blocks vorkommen dürfen – formale Parameter x_i und lokale Variable y_i von P' sind in `STA` also erlaubt.

Wir fassen solche Prozeduren als Abkürzung für

```

procedure P(x1, ..., xk) <=
begin var y1, ..., yn, y, z1, ..., zm;
...
STA[y'1/z1, ..., y'm/zm];
y := zj;
...
return(...)
end

```



```

procedure EQUAL(x1,x2) <=
begin var n,m,false,res;
  n := x1; m := x2;
  while n do
    n := PRED(n);
    if m then m := PRED(m) else false := 1 end_if
  end_while;
  if false
  then res := 0
  else if m then res := 0 else res := 1 end_if
  end_if;
  return(res)
end

```

Abbildung 2.1: Eine \mathcal{P} -Prozedur zur Berechnung von $x_1 = x_2$

auf, wobei z_1, \dots, z_m Bezeichner für lokale Variable sind, die alle verschieden von den Programmvariablen der Prozedur P' sind und $STA[y'_1/z_1, \dots, y'_m/z_m]$ aus STA entsteht, indem jedes Vorkommen der lokalen Variablen y'_j in STA durch z_j ersetzt wird. Die Umbenennungen stellen sicher, daß bei Namenskonflikten zwischen lokalen Variablen in einem Block und Programmvariablen der Prozedur (durch identische Benennungen) immer die lokalen Variablen des innersten Blocks verwendet werden.⁵

- Wir erlauben Blöcke anstelle von Programmausdrücken auch in bedingten Anweisungen und in Schleifenanweisungen, d.h. wir dürfen sowohl

```

procedure P'(x1, ..., xk) <=
begin var y1, ..., yn;
  ...
  if begin var y'1, ..., y'm; STA; return(y'j) end
  then STA1
  else STA2
  end_if;
  ...
  return(...)
end

```

⁵ In Beispielen werden wir – falls bequem – auf die Umbenennung einer lokalen Variablen y'_j aus dem Rumpf des Blocks zu z_j verzichten, falls y'_j verschieden von allen Programmvariablen der Prozedur ist.

als auch

```
procedure Q'(x1, ..., xk) <=  
begin var y1, ..., yn;  
  ...  
  while begin var y'1, ..., y'm; STA; return(y'j) end  
    do STA1 end_while;  
  ...  
  return(...)  
end
```

schreiben, wobei STA eine Programmanweisung ist, in der die im Block deklarierten lokalen Variablen y'_1, \dots, y'_m sowie Programmvariable des umgebenden Blocks vorkommen dürfen – formale Parameter x_i und lokale Variable y_j von P' bzw. Q' sind in STA also erlaubt.

Wir fassen solche Prozeduren als Abkürzung für

```
procedure P(x1, ..., xk) <=  
begin var y1, ..., yn, y;  
  ...  
  y := begin var y'1, ..., y'm; STA; return(y'j) end;  
  if y then STA1 else STA2 end_if;  
  ...  
  return(...)  
end
```

und

```
procedure Q(x1, ..., xk) <=  
begin var y1, ..., yn, y;  
  ...  
  y := begin var y'1, ..., y'm; STA; return(y'j) end;  
  while y do  
    STA1;  
    y := begin var y'1, ..., y'm; STA; return(y'j) end  
  end_while;  
  ...  
  return(...)  
end
```

auf, wobei y ein Bezeichner einer lokalen Variable ist, der verschieden von den Programmvariablen der Prozeduren P' und Q' ist.

3. Wir verwenden Hilfsprozeduren in \mathcal{P} -Programmen, um diese besser zu strukturieren und lesbarer zu gestalten. Dazu erlauben wir Prozeduraufrufe der Form $\text{PROC}(\text{EXPR}_1, \dots, \text{EXPR}_k)$ an allen Stellen in einem Programm bzw. in einer anderen Prozedur, an denen ein Programmausdruck stehen darf. Dabei sind $\text{EXPR}_1, \dots, \text{EXPR}_k$ Programmausdrücke, die keine Prozeduraufrufe enthalten, die zu (direkt oder gegenseitig) rekursiv definierten Programmen führen.⁶ Für eine \mathcal{P} -Prozedur

```
procedure PROC( $x_1, \dots, x_k$ ) <=
begin var  $y_1, \dots, y_n$ ; STA; return( $y_i$ ) end
```

wird ein Prozeduraufruf $\text{PROC}(\text{EXPR}_1, \dots, \text{EXPR}_k)$ in einer \mathcal{P} -Prozedur P als Abkürzung für den Block

```
begin var  $y_1, \dots, y_n$ ;
  STA[ $x_1/\text{EXPR}_1, \dots, x_k/\text{EXPR}_k$ ];
  return( $y_i$ )
end
```

aufgefaßt, wobei $\text{STA}[x_1/\text{EXPR}_1, \dots, x_k/\text{EXPR}_k]$ aus STA entsteht, indem jedes Vorkommen eines formalen Parameters x_i im Rumpf STA von PROC durch den korrespondierenden aktuellen Parameter EXPR_i ersetzt wird. ■

Wir erhalten ein \mathcal{P} -Programm P aus einem Programm P' der erweiterten Sprache wie folgt:

1. Wir eliminieren alle Prozeduraufaufrufe in P', so wie in Definition 2.2(3) angegeben. Dies gelingt immer, da Prozeduraufrufe, die zu (direkt oder gegenseitig) rekursiv definierten Programmen führen, ausgeschlossen sind.
2. In einem zweiten Schritt eliminieren wir alle Blöcke in bedingten und in Schleifenanweisungen, so wie in Definition 2.2(2) angegeben.
3. Nach Schritt 2. erhält man ein Programm der erweiterten Sprache, in dem keine Prozeduraufrufe vorhanden sind und Blöcke nur noch auf der rechten Seite von Zuweisungen vorkommen. Diese werden so wie in Definition 2.2(1) angegeben eliminiert. Resultat ist dann ein \mathcal{P} -Programm P, das den Forderungen von Definition 2.1 genügt.

Um die Programmierung weiter zu vereinfachen, führen wir zusätzliche Schreibweisen ein:

⁶ Mit dieser Forderung erzwingen wir, daß Prozeduraufaufrufe immer ersetzt werden können. Bei einer Prozedur der Form $\text{procedure P}(x) <= \text{begin var } y; y := P(\text{EXPR}); \text{return}(y) \text{ end}$ wäre dies nicht möglich.

```

case EXPR of
  n1: STA1
  n2: STA2
  ⋮
  nk: STAk
  other: STAother
end_case

if EXPR = n1
  then STA1
  else if EXPR = n2
    then STA2
    else ...
      ⋮
      if EXPR = nk
        then STAk
        else STAother
      end_if
    end_if
  ...
end_if

```

Abbildung 2.2: case-Anweisungen als Abkürzungen für bedingte Anweisungen

```

procedure MINUS(y1,y2) <=
begin var n,m;
  n := y1; m := y2;
  while m do
    n := PRED(n); m := PRED(m)
  end_while; return(n)
end

procedure GE(y1,y2) <=
begin var n,m,res;
  n := y1; m := y2;
  while n ^ m do
    n := PRED(n); m := PRED(m)
  end_while;
  if m = 0 then res := 1 end_if;
  return(res)
end

procedure GGT(y1,y2) <=
begin var n,m,res;
  n := y1;
  m := y2;
  while n ^ m do
    if GE(n,m)
      then n := MINUS(n,m)
      else m := MINUS(m,n)
    end_if
  end_while;
  if n
    then res := n
    else res := m
  end_if;
  return(res)
end

```

Abbildung 2.3: Ein \mathcal{P} -Programm mit Hilfsprozeduren zur Berechnung des *ggt*

Definition 2.3. (Erweiterung 2 der Programmiersprache \mathcal{P})

1. Da wir natürliche Zahlen in den bedingten Anweisungen und in den Schleifenanweisungen wie Wahrheitswerte auffassen, ist es praktisch, logische Operatoren auch für natürliche Zahlen zu erlauben. Wir verwenden daher für Programmausdrücke EXPR1 und EXPR2
 - “ $\neg\text{EXPR1}$ ” als Abkürzung für “begin var y; if EXPR1 then y := 0 else y := 1 end_if; return(y) end”,
 - “ $\text{EXPR1} \vee \text{EXPR2}$ ” als Abkürzung für “begin var y; if EXPR1 then y := 1 else y := EXPR2 end_if; return(y) end”, und
 - “ $\text{EXPR1} \wedge \text{EXPR2}$ ” als Abkürzung für “begin var y; if EXPR1 then y := EXPR2 else y := 0 end_if; return(y) end”.
2. Wir erlauben Programmausdrücke der Form $\text{EXPR}' = \text{EXPR}''$, die wir als Abkürzung für den Aufruf $\text{EQUAL}(\text{EXPR}', \text{EXPR}'')$ einer Prozedur EQUAL , definiert wie in Abbildung 2.1, auffassen.
3. Wir erlauben einseitig bedingte Anweisungen und schreiben “if EXPR then STA end_if” als Abkürzung für “if EXPR then STA else SKIP end_if”.
4. Wie erlauben *case*-Anweisungen, d.h. Fallunterscheidungen, die wir als Abkürzung für eine bedingte Anweisung auffassen, siehe Abbildung 2.2. Dabei ist EXPR ein Programmausdruck, n_1, \dots, n_k mit $k \geq 1$ sind natürliche Zahlen und $\text{STA}_1, \dots, \text{STA}_k, \text{STA}_{\text{other}}$ sind Programmanweisungen.
5. Wir erlauben eine zweite Form von Schleifenanweisungen und schreiben “repeat STA until EXPR ” als Abkürzung für “ STA ; while $\neg\text{EXPR}$ do STA end_while”. ■

Mit diesen Erweiterungen unserer Programmiersprache läßt sich jetzt beispielsweise ein GGT-Programm wie in Abbildung 2.3 angeben. Aus diesem Programm entsteht das \mathcal{P} -Programm aus Abbildung 2.4 durch Elimination der Hilfsprozeduren und der logischen Operatoren, so wie in Definitionen 2.2 und 2.3 angegeben.

Um die Lesbarkeit von \mathcal{P} -Programmen weiter zu erhöhen, verwenden wir offensichtliche abkürzende Schreibweisen, also etwa $x \neq y$ anstatt $\neg x = y$. Im folgenden werden wir die Programmiersprache \mathcal{P} sowohl in ihrer ursprünglichen Definition als auch in ihren Erweiterungen verwenden, je nachdem, wie es für den jeweiligen Zweck am bequemsten ist. Dabei dürfen wir jedoch bei Untersuchungen über \mathcal{P} -Programme immer davon ausgehen, daß ein \mathcal{P} -Programm in der ursprünglichen

```

procedure GGT(y1,y2) <=
begin var n,m,res,loop;
n := y1; m := y2;
loop := begin var r;
            if n then r := m else r := n end_if;
            return(r)
        end;
while loop do
    if begin var x,y,res,loop; x := n; y := m; res := 0;
        loop := begin var r;
                    if x then r := y else r := x end_if;
                    return(r)
                end;
            while loop do
                x := PRED(x); y := PRED(y);
                if x then loop := y else loop := x end_if
            end_while;
            if y then SKIP else res := 1 end_if;
            return(res)
        end
    then n := begin var x,y; x := n; y := m;
                while y do
                    x := PRED(x); y := PRED(y)
                end_while;
                return(x)
            end; loop := n
    else m := begin var x,y; x := m; y := n;
                while y do
                    x := PRED(x); y := PRED(y)
                end_while;
                return(x)
            end; loop := m
        end_if
    end_while;
    if n then res := n else res := m end_if;
    return(res)
end

```

Abbildung 2.4: Ein \mathcal{P} -Programm zur Berechnung des *ggt* von y_1 und y_2

Form nach Definition 2.1 gegeben ist, denn die Programmkonstrukte der erweiterten Sprache können immer – wie in Definitionen 2.2 und 2.3 angegeben – eliminiert werden. Dort, wo es erheblich ist, ob ein \mathcal{P} -Programm in der ursprünglichen oder in der erweiterten Form gegeben ist, werden wir explizit darauf hinweisen.

Da rekursiv definierte Prozeduren verboten sind, dürfen wir insbesondere davon ausgehen, daß jedes Programm aus genau einer Prozedur besteht, denn Prozeduraufrufe können immer durch den instantiierten Rumpf der aufgerufenen Prozedur ersetzt werden. Deshalb verwenden wir im folgenden die Begriffe *Programm* und *Prozedur* auch als Synonyme. Beispielsweise definiert die Prozedur **MINUS** aus Abbildung 2.3 ein Programm, und wir sagen dann z.B. auch, daß das *Programm* **MINUS** aufgerufen wird. Andererseits können wir auch sagen, daß die *Prozedur* **GGT** aufgerufen wird, wobei wir dann stillschweigend voraussetzen, daß die Prozeduren, die bei Definition von **GGT** verwendet wurden, also **MINUS** und **GE**, zusammen mit **GGT** gegeben sind.

2.3. Semantik von \mathcal{P} -Programmen

Die Programme der Programmiersprache \mathcal{P} sind zunächst einmal nur syntaktische Objekte, also Worte über einem bestimmten Alphabet. Da wir mit Programmen Algorithmen beschreiben wollen, müssen wir jetzt definieren, was es bedeuten soll, wenn ein \mathcal{P} -Programm auf einem *Rechner abgearbeitet* wird. Wir müssen also die *Semantik* von \mathcal{P} , d.h. die *Bedeutung* der Ausdrücke und Anweisungen der Programmiersprache \mathcal{P} definieren. Zwar ist in den meisten Fällen intuitiv klar, was die Abarbeitung eines \mathcal{P} -Programms bewirkt. Um aber Mißverständnisse und Fehlinterpretationen zu vermeiden, muß die Semantik von \mathcal{P} -Programmen präzise angegeben werden, etwa um zu klären, was das Ergebnis von **PRED**(0) und was die Bedeutung der Leeranweisung **SKIP** ist, was geschieht, wenn auf eine nicht-initialisierte lokale Variable zugegriffen wird, u.s.w. Die Semantik muß aber vor allem deshalb mathematisch präzise definiert werden, damit wir Aussagen über die Berechnungen, die bei Ausführung von \mathcal{P} -Programmen stattfinden, *formal beweisen* können.

Wir definieren jetzt eine *operationale* Semantik von \mathcal{P} -Programmen, indem wir angeben, wie genau \mathcal{P} -Programme abgearbeitet werden sollen.⁷ Dazu benötigen wir zunächst einen *Speicher* beliebiger Größe. Diesen Speicher modellieren wir durch eine Teilmenge M des kartesischen Produkts $\mathbb{N} \times \mathbb{N}$, wobei $\langle i, n \rangle \in M$ be-

⁷ Man kann die Semantik von Programmen auch nicht-*operational* angeben und diese etwa *axiomatisch* oder *denotational* definieren. Diese verschiedenen Konzepte zur Semantikdefinition besitzen alle Vor- und Nachteile; für unsere Untersuchungen ist die operationale Form zweckmäßig.

deutet, daß in der *Speicherzelle* von M mit *Adresse* i der *Wert* n abgespeichert ist. Den Programmvariablen \mathbf{z} ordnen wir (beginnend mit 0) fortlaufende Adressen $i_{\mathbf{z}}$ aus \mathbb{N} zu, und $M(i_{\mathbf{z}})$ bezeichnet dann den Wert, der in M für eine Programmvariable \mathbf{z} abgespeichert ist.

Definition 2.4. (Speichermodell)

Für die Funktion $cont: 2^{\mathbb{N} \times \mathbb{N}} \times \mathbb{N} \rightarrow \mathbb{N}$ definiert durch

$$cont(M, i) := \begin{cases} n & , \text{ falls } \langle i, n \rangle \in M \text{ und } n < m \\ & \text{für alle } m \in \mathbb{N} \text{ mit } \langle i, m \rangle \in M \\ 0 & , \text{ falls } \langle i, m \rangle \notin M \text{ für alle } m \in \mathbb{N} \end{cases}$$

ist $M(i)$ definiert als $cont(M, i)$.⁸ M^{init} bezeichnet den initialen Speicher mit $M^{init}(i) = 0$ für jedes $i \in \mathbb{N}$.

Durch Ersetzen des Speicherinhalts an der Adresse j durch einen Wert n erhält man den Speicher $M[j \leftarrow n]$, für den $M[j \leftarrow n](i) = n$, falls $i = j$, und andernfalls $M[j \leftarrow n](i) = M(i)$ gilt.

Für eine Programmvariable \mathbf{z} mit Adresse $i_{\mathbf{z}}$ steht $M(\mathbf{z})$ als Abkürzung für $M(i_{\mathbf{z}})$ und $M[\mathbf{z} \leftarrow n]$ als Abkürzung für $M[i_{\mathbf{z}} \leftarrow n]$. ■

Bemerkung 2.5. Zwar ist der Adressraum unseres Speichers endlich (denn \mathcal{P} -Programme sind endliche Texte, und damit gibt es auch nur endliche viele Programmvariable in einem \mathcal{P} -Programm), nicht jedoch der Speicher insgesamt. Da jede natürliche Zahl im Speicher abgelegt werden kann, ist die Größe des Speichers unbegrenzt. Dies wird sofort klar, wenn wir die Programmiersprache \mathcal{P} auf einem konkreten Rechner implementieren, denn dann muß eine Repräsentation für natürliche Zahlen gewählt werden. Stellt man beispielsweise natürliche Zahlen durch Bitworte dar, so sind diese Bitworte zwar immer endlich, sie können jedoch beliebig lang sein (siehe auch Abschnitt 12.2). Damit existiert keine Größenbeschränkung bzgl. des verwendeten realen Speichers. Durch unsere Modellierung als Teilmenge von $\mathbb{N} \times \mathbb{N}$ scheint dagegen ein endlicher Speicherbedarf bei Abarbeitung eines Programms auszureichen. Für unsere Untersuchungen ist dies unerheblich. In der Komplexitätstheorie, in der auch der Speicherbedarf bei Lösung von Problemen untersucht wird, muß dagegen ein Speichermodell gewählt werden, das quantitative Aspekte explizit macht. ■

⁸ Da mit M nur eine *Relation* definiert wird, ist beispielsweise $\langle 2, 7 \rangle \in M$ und $\langle 2, 13 \rangle \in M$ nicht von vornherein ausgeschlossen. Da das Auslesen des Speichers M aber offenbar durch eine *Funktion* (hier: $cont$) modelliert werden muß, ist für solche Fälle eine (willkürliche) Auswahl erforderlich (im Beispiel erhält man $M(2) = 7$). Unsere Verwendung des Speichers M wird jedoch sicherstellen, daß solch willkürliche Auswahl nie erforderlich ist.

Die *Auswertung* von Programmausdrücken und die *Ausführung* von Programm-
anweisungen definieren wir jetzt durch die Funktionen

$$value : 2^{\mathbb{N} \times \mathbb{N}} \times expressions_P \rightarrow \mathbb{N}$$

und

$$eval : 2^{\mathbb{N} \times \mathbb{N}} \times statements_P \mapsto 2^{\mathbb{N} \times \mathbb{N}} .$$

Dabei bezeichnet $expressions_P$ die Menge aller Programmausdrücke und $statements_P$ die Menge aller Programmanweisungen in einem \mathcal{P} -Programm P . Für eine gegebene Speicherbelegung M_P und einen Programmausdruck $EXPR$ gibt $value(M_P, EXPR) \in \mathbb{N}$ den *Wert* von $EXPR$ bezüglich der Speicherbelegung M_P an. Durch Ausführung einer Programmanweisung STA wird aus einer gegebenen Speicherbelegung M_P eine *neue Speicherbelegung* gebildet, d.h. das Ergebnis der Ausführung von STA unter der Speicherbelegung M_P ist die Speicherbelegung $eval(M_P, STA) \subseteq \mathbb{N} \times \mathbb{N}$.

Definition 2.6. (Auswertungsfunktion *value*)

Für ein \mathcal{P} -Programm P definieren wir die Auswertung von Programmausdrücken unter einer endlichen Speicherbelegung $M_P \subseteq \mathbb{N} \times \mathbb{N}$ mittels der Auswertungsfunktion $value : 2^{\mathbb{N} \times \mathbb{N}} \times expressions_P \rightarrow \mathbb{N}$ durch:

$$(v1) \quad value(M_P, n) := n, \text{ falls } n \in \mathbb{N},$$

$$(v2) \quad value(M_P, z) := M_P(z), \text{ falls } z \text{ eine Programmvariable ist,}$$

$$(v3) \quad value(M_P, SUCC(EXPR)) := 1 + value(M_P, EXPR),$$

$$(v4) \quad value(M_P, PRED(EXPR)) := \begin{cases} 0, & \text{falls } value(M_P, EXPR) = 0 \\ m, & \text{falls } value(M_P, EXPR) = m + 1 . \end{cases} \quad \blacksquare$$

Definition 2.7. (Ausführungsfunktion *eval*)

Für ein \mathcal{P} -Programm P definieren wir die Ausführung von Programmanweisungen unter einer endlichen Speicherbelegung $M_P \subseteq \mathbb{N} \times \mathbb{N}$ mittels der Ausführungsfunktion $eval : 2^{\mathbb{N} \times \mathbb{N}} \times statements_P \mapsto 2^{\mathbb{N} \times \mathbb{N}}$ durch:

$$(e1) \quad eval(M_P, SKIP) := M_P,$$

$$(e2) \quad eval(M_P, x := EXPR) := M_P[x \leftarrow value(M_P, EXPR)],$$

$$(e3) \quad \begin{array}{l} \text{if } \text{EXPR} \\ \text{then } \text{STA1} \\ \text{else } \text{STA2} \\ \text{end_if} \end{array} := \begin{cases} \text{eval}(M_P, \text{STA1}), & \text{falls } \text{value}(M_P, \text{EXPR}) > 0 \\ \text{eval}(M_P, \text{STA2}), & \text{falls } \text{value}(M_P, \text{EXPR}) = 0, \end{cases}$$

$$(e4) \quad \text{eval}(M_P, \text{STA1}; \text{STA2}) := \text{eval}(\text{eval}(M_P, \text{STA1}), \text{STA2}),$$

$$(e5) \quad \begin{array}{l} \text{while } \text{EXPR} \\ \text{do } \text{STA} \\ \text{end_while} \end{array} := \begin{cases} M_P, & \text{falls } \text{value}(M_P, \text{EXPR}) = 0, \text{ und sonst} \\ \text{eval}(\text{eval}(M_P, \text{STA}), \begin{array}{l} \text{while } \text{EXPR} \\ \text{do } \text{STA} \\ \text{end_while} \end{array}) . \quad \blacksquare \end{cases}$$

Die Zuweisung ist also (wenig überraschend) die einzige Programmanweisung, mit der der Speicher verändert wird. Die Ausführung der Leeraanweisung **SKIP** hat keine Wirkung, und bei Ausführung der bedingten Anweisung wird lediglich zwischen zwei Anweisungsalternativen ausgewählt. Bei der Hintereinanderausführung **STA1;STA2** von Programmanweisungen wird zunächst **STA1** unter der Speicherbelegung M_P ausgeführt, und anschließend die Programmanweisung **STA2** unter der nach Ausführung von **STA1** gegebenen Speicherbelegung. Die Ausführung der Schleifenanweisung hat keine Wirkung, falls die Schleifenbedingung **EXPR** nicht erfüllt ist. Anderfalls wird der Schleifenrumpf **STA** unter der gegebenen Speicherbelegung einmal ausgeführt, und daran anschließend wieder die gesamte Schleifenanweisung. Damit wird der Schleifenrumpf solange ausgeführt, bis eine Speicherbelegung resultiert, unter der die Schleifenbedingung **EXPR** nicht mehr erfüllt ist.

Offensichtlich bezeichnet *value* eine totale Funktion, d.h. die Auswertung von Programmausdrücken gelingt immer. Dagegen ist die Ausführungsfunktion *eval* im allgemeinen eine *nicht-totale* Funktion, da wir auch *nicht-terminierende* Programme definieren können, etwa durch

```

procedure CYCLEk(x1, ..., xk) <=
begin var y;
  while 1 do SKIP end_while;
  return(y)
end .

```

(2.1)

Mittels *value* und *eval* können wir jetzt die Semantik eines \mathcal{P} -Programms P mathematisch präzise angeben, und zwar als eine *Funktion*, die die *Eingabewerte* von P in das *Ergebnis* der *Abarbeitung* von P abbildet:

Definition 2.8. (Semantik von \mathcal{P} -Programmen)

Sei

```
procedure P( $\mathbf{x}_1, \dots, \mathbf{x}_k$ ) <=
begin var  $y_1, \dots, y_n$ ; STA; return( $y_j$ ) end
```

ein \mathcal{P} -Programm, sei $(n_1, \dots, n_k) \in \mathbb{N}^k$, und sei $M_P^{(n_1, \dots, n_k)}$ eine Teilmenge von $\mathbb{N} \times \mathbb{N}$ mit ⁹

$$\begin{aligned} M_P^{(n_1, \dots, n_k)}(\mathbf{x}_i) &= n_i && \text{für alle } i \in \{1, \dots, k\} \text{ und} \\ M_P^{(n_1, \dots, n_k)}(\mathbf{y}_i) &= 0 && \text{für alle } i \in \{1, \dots, n\} . \end{aligned} \quad (2.2)$$

Dann ist die durch P berechnete Funktion $\llbracket P \rrbracket : \mathbb{N}^k \mapsto \mathbb{N}$ definiert durch

$$\llbracket P \rrbracket(n_1, \dots, n_k) := \text{value}(\text{eval}(M_P^{(n_1, \dots, n_k)}, \text{STA}), \mathbf{y}_j) .$$

Die Semantik $\llbracket P(t_1, \dots, t_k) \rrbracket$ eines Programmaufrufs (bzw. Prozeduraufrufs) $P(t_1, \dots, t_k)$ mit variablenfreien Programmausdrücken t_1, \dots, t_k als aktuelle Parameter definieren wir als $\llbracket P \rrbracket(n_1, \dots, n_k)$ mit $n_i = \text{value}(M_P^{\text{init}}, t_i)$ für alle $i \in \{1, \dots, k\}$. ■

Die *Semantik*, d.h. die *Bedeutung*, eines Programms P ist also die durch P berechnete *Funktion* $\llbracket P \rrbracket$.¹⁰ Für diese Funktion gilt $\llbracket P \rrbracket(n_1, \dots, n_k) = n$ genau dann, wenn P gestartet mit Eingabe n_1, \dots, n_k als Ergebnis n liefert. Anders gesagt, bei Eingabe von $P(t_1, \dots, t_k)$ auf der Tastatur eines Rechners wird $\llbracket P \rrbracket(n_1, \dots, n_k)$ (mit $n_i = \text{value}(M_P^{\text{init}}, t_i)$) als Ergebnis angezeigt, vorausgesetzt natürlich, die Abarbeitung des Programms terminiert. Beispielsweise gilt $\llbracket \text{GGT} \rrbracket(n_1, n_2) = \text{ggt}(n_1, n_2)$ für das Programm **GGT** aus Abbildung 2.4 (und damit auch für das Programm **GGT** aus Abbildung 2.3).

Die Funktion $\llbracket P \rrbracket$ ist genau dann *total*, wenn P für *jede* Eingabe *terminiert*. Dies ist z.B. für das Programm **GGT** der Fall, und somit gilt $\llbracket \text{GGT} \rrbracket : \mathbb{N}^2 \rightarrow \mathbb{N}$. Dagegen terminiert das Programm **CYCLE**₁ $\in \mathcal{P}[1]$ aus (2.1) für keine Eingabe, und folglich gilt $\llbracket \text{CYCLE}_1 \rrbracket = \omega_{\mathbb{N} \rightarrow \mathbb{N}}$.

⁹ Leere Parameterlisten, also $k = 0$, sind erlaubt.

¹⁰ Da wir die Semantik von \mathcal{P} -Programmen mittels der Funktionen *value* und *eval algorithmisch* definiert haben, haben wir für \mathcal{P} -Programme eine *operationale* Semantik angegeben.



3. BERECHENBARE FUNKTIONEN

3.1. \mathcal{P} -berechenbare Funktionen

Mit der Definition der Semantik von \mathcal{P} -Programmen können wir jetzt die Aussage “Aufgabe x ist lösbar durch einen Rechner” präziser formulieren. Wir definieren eine Funktion $\phi : \mathbb{N}^k \mapsto \mathbb{N}$ und fragen: “Gibt es ein \mathcal{P} -Programm **PROG**, so daß $\phi = \llbracket \text{PROG} \rrbracket$ gilt?”. Beispielsweise formulieren wir die Aufgabe, die n -te Primzahl anzugeben, wie folgt: Sei $\text{prim} : \mathbb{N} \rightarrow \mathbb{N}$ definiert als $\text{prim}(n) = n$ -te Primzahl, also $\text{prim}(0) = 1$, $\text{prim}(1) = 2$, $\text{prim}(2) = 3$, $\text{prim}(3) = 5$, $\text{prim}(4) = 7$ u.s.w.¹ Die Aufgabe “ n -te Primzahl angeben” ist genau dann durch einen Rechner lösbar, wenn $\text{prim} = \llbracket \text{PRIM} \rrbracket$ für ein \mathcal{P} -Programm **PRIM** gilt. Allgemein fragen wir also danach, ob eine Funktion ϕ \mathcal{P} -berechenbar ist.²

Definition 3.1. (\mathcal{P} -berechenbare Funktionen)

Eine Funktion $\phi : \mathbb{N}^k \mapsto \mathbb{N}$ ist \mathcal{P} -berechenbar genau dann, wenn $\phi = \llbracket \text{PROG} \rrbracket$ für ein $\text{PROG} \in \mathcal{P}[k]$ gilt. $\llbracket \mathcal{P} \rrbracket$ mit $\llbracket \mathcal{P} \rrbracket := \{ \llbracket \text{PROG} \rrbracket \mid \text{PROG} \in \mathcal{P} \}$ ist die Menge aller \mathcal{P} -berechenbaren Funktionen. ■

Die Aufgabe “ n -te Primzahl angeben” ist natürlich durch einen Rechner lösbar, da man ein Programm $\text{PRIM} \in \mathcal{P}[1]$ mit $\text{prim} = \llbracket \text{PRIM} \rrbracket$ leicht angeben kann. Ebenso sind alle überall undefinierten arithmetischen Funktionen \mathcal{P} -berechenbar:

Satz 3.2. $\omega_{\mathbb{N}^k \mapsto \mathbb{N}}$ ist \mathcal{P} -berechenbar.

Beweis. Für $\text{CYCLE}_k \in \mathcal{P}[k]$, gegeben wie (2.1), gilt $\llbracket \text{CYCLE}_k \rrbracket(n_1, \dots, n_k) = \perp$ für alle $(n_1, \dots, n_k) \in \mathbb{N}^k$, folglich $\omega_{\mathbb{N}^k \mapsto \mathbb{N}} = \llbracket \text{CYCLE}_k \rrbracket$, und damit ist $\omega_{\mathbb{N}^k \mapsto \mathbb{N}}$ \mathcal{P} -berechenbar. ■

¹ Für unsere Zwecke ist es praktisch, auch 1 zu verwenden. Daher definieren wir $\text{prim}(0) = 1$.

² In der englischsprachigen Literatur wird der Begriff “recursive” für “berechenbar” verwendet (“computable” ist dagegen seltener). Es gibt also sowohl “recursive procedures” (dt. “rekursive Prozeduren”) als auch “recursive functions” (dt. “berechenbare Funktionen”). Genaugenommen müßte man von “recursively defined procedures” bzw. “rekursiv definierten Prozeduren” sprechen. Da dies aber oft unterlassen wird, kann es zu Mißverständnissen kommen.



Die Komposition \mathcal{P} -berechenbarer Funktionen liefert wieder eine \mathcal{P} -berechenbare Funktion:

Satz 3.3. Seien $\phi : \mathbb{N}^k \mapsto \mathbb{N}$ und $\psi : \mathbb{N} \mapsto \mathbb{N}$ \mathcal{P} -berechenbare Funktionen. Dann ist auch $\psi \circ \phi : \mathbb{N}^k \mapsto \mathbb{N}$ \mathcal{P} -berechenbar.

Beweis. Seien $\text{PSI} \in \mathcal{P}[1]$ und $\text{PHI} \in \mathcal{P}[k]$ mit $\psi = \llbracket \text{PSI} \rrbracket$ und $\phi = \llbracket \text{PHI} \rrbracket$ gegeben durch

| | | |
|---|-----|---|
| <pre> procedure PSI(x) <= begin var y1, ..., yn; STA_ψ; return(y_ψ) end </pre> | und | <pre> procedure PHI(x1, ..., xk) <= begin var y1, ..., ym; STA_φ; return(y_φ) end . </pre> |
|---|-----|---|

Für das \mathcal{P} -Programm $\text{COMPOSE} \in \mathcal{P}[k]$, gegeben durch

```

procedure COMPOSE(x1, ..., xk) <=
begin var y1, ..., yn;
  STAψ[x/begin var y1, ..., ym; STAφ; return(yφ) end];
  return(yψ)
end

```

gilt dann $\llbracket \text{COMPOSE} \rrbracket(n_1, \dots, n_k) = \psi(\phi(n_1, \dots, n_k))$ für alle $(n_1, \dots, n_k) \in \mathbb{N}^k$, und somit ist $\psi \circ \phi$ \mathcal{P} -berechenbar. ■

3.2. Die Churchsche These

Der bislang verwendete Berechenbarkeitsbegriff stützt sich auf die Programmiersprache \mathcal{P} . Wir lösen uns von dieser Einschränkung und definieren allgemeiner:

Eine arithmetische Funktion ϕ ist genau dann *intuitiv berechenbar*, wenn ϕ durch *irgendein algorithmisches Verfahren* berechnet wird.

Offenbar ist jede \mathcal{P} -berechenbare Funktion auch *intuitiv* berechenbar, denn es gilt dann $\llbracket \text{PROG} \rrbracket = \phi$ für ein \mathcal{P} -Programm PROG , und mit PROG wird offenbar ein algorithmisches Verfahren zur Berechnung von ϕ angegeben. Gilt jedoch $\phi \neq \llbracket \text{PROG} \rrbracket$ für jedes \mathcal{P} -Programm PROG , so können wir keine Aussage darüber treffen, ob ϕ intuitiv berechenbar ist: Entweder ist ϕ nicht intuitiv berechenbar, und

dann kann es natürlich auch kein \mathcal{P} -Programm PROG mit $\llbracket \text{PROG} \rrbracket = \phi$ geben, oder aber ϕ ist nicht \mathcal{P} -berechenbar, da die Programmiersprache \mathcal{P} ungeeignet ist, um einen Algorithmus für ϕ anzugeben. Anders gesagt, mit dem Nachweis, daß eine Funktion ϕ nicht \mathcal{P} -berechenbar ist, wird zunächst nur eine Aussage über die Programmiersprache \mathcal{P} getroffen, nicht jedoch – wie eigentlich erwünscht – über die Funktion ϕ . Dies gilt nicht nur für die Programmiersprache \mathcal{P} , sondern für jeden Formalismus zur Beschreibung von Algorithmen. Man begegnet diesem Dilemma wie folgt:

Zu Beginn des letzten Jahrhunderts wurden verschiedene Berechenbarkeitsbegriffe – also Formalismen zur Beschreibung von Algorithmen – definiert, darunter

- der λ -Kalkül von *A. Church*,
- die *Turingmaschine* von *A. Turing* und
- die *μ -rekursiven Funktionen* von *S. C. Kleene*.³

Man konnte zeigen, daß alle diese Formalismen *berechnungsäquivalent* sind, d.h. jede Funktion, die mit einem der Formalismen berechnet werden kann, kann auch mit den jeweils anderen beiden berechnet werden. Da sich diese Formalismen in ihren Definitionen stark unterscheiden, jedoch gleichmächtig sind, liegt es nahe, von einem intuitiven Berechenbarkeitsbegriff auszugehen, der sich nicht auf einen bestimmten Berechnungsformalismus (also etwa eine bestimmte Programmiersprache) stützt. Anders gesagt, man geht davon aus, daß jede *intuitiv* berechenbare Funktion auch durch den λ -Kalkül (die Turingmaschine oder die μ -rekursiven Funktionen) berechnet wird. Diese Vermutung wird ausgedrückt durch die

Churchsche These

Jede *intuitiv berechenbare* Funktion ist durch den λ -Kalkül (die Turingmaschine, die μ -rekursiven Funktionen) berechenbar.

Die Churchsche These ist ein *Postulat*, d.h. kein Satz, den man beweisen kann. Allenfalls könnte man die Churchsche These *widerlegen*, indem man ein Verfahren

³ Diese Aufzählung ist nicht erschöpfend. Es gibt noch weitere Berechnungsformalismen, genau genommen beliebig viele, die das gleiche leisten wie die hier genannten.

angibt, von dem übereinstimmend behauptet werden kann, daß es ein *algorithmisches* Verfahren ist, und dann nachweist, daß dieses Verfahren *nicht* durch einen der zuvor genannten Berechnungsformalisten implementiert werden kann.

Bei der Churchschen These handelt es sich damit um eine *empirische* Aussage, mit der von Beobachtungen auf eine allgemeine Gesetzmäßigkeit geschlossen wird. Dies ist auch in anderen Bereichen der Wissenschaft nicht unüblich. Beispielsweise ergibt sich die Unmöglichkeit eines *perpetuum mobile*, also eines Geräts, das (physikalische) Arbeit verrichtet ohne Energie aufzunehmen, aus dem Energieerhaltungssatz der Physik, also einem nicht-beweisbaren Naturgesetz bzw. Postulat, von dessen Wahrheit man aufgrund von Beobachtungen solange ausgeht, bis das Gegenteil nachgewiesen ist.

Mit der Churchschen These emanzipiert man sich von einem konkreten Berechenbarkeitsbegriff: Unter der Voraussetzung dieser These darf man eine Funktion ϕ als *nicht berechenbar* bezeichnen, wenn man gezeigt hat, daß diese Funktion nicht durch den λ -Kalkül (bzw. die Turingmaschine oder die μ -rekursiven Funktionen) berechnet wird. Man kann dabei auch die nicht-Berechenbarkeit durch irgendeinen anderen Formalismus verwenden, vorausgesetzt man hat zuvor gezeigt, daß dieser Formalismus *berechnungsvollständig*, d.h. *berechnungsäquivalent* mit dem λ -Kalkül (bzw. der Turingmaschine oder den μ -rekursiven Funktionen) ist.

In unserem Fall ist also nachzuweisen, daß die Programmiersprache \mathcal{P} berechnungsvollständig ist, siehe Kapitel 11 und 12.⁴ Im Vorgriff auf diesen Beweis dürfen wir – unter Annahme der Churchschen These – im folgenden davon ausgehen, daß jede nicht \mathcal{P} -berechenbare Funktion auch nicht intuitiv berechenbar ist.

3.3. Nicht-Konstruktivität und Konstruktivität

Um zu beweisen, daß eine gegebene Funktion ϕ \mathcal{P} -berechenbar ist, müssen wir die Existenz eines \mathcal{P} -Programms `PROG` mit $\phi = \llbracket \text{PROG} \rrbracket$ nachweisen. Dies kann beispielsweise dadurch geschehen, daß wir ein *konkretes* Programm angeben, etwa ein \mathcal{P} -Programm zur Berechnung der n -ten Primzahl, um nachzuweisen, daß die Funktion, die jedem $n \in \mathbb{N}$ die n -te Primzahl zuordnet, \mathcal{P} -berechenbar ist.

Die Angabe eines konkreten \mathcal{P} -Programms ist zwar *hinreichend* zum Nachweis der \mathcal{P} -Berechenbarkeit einer Funktion, nicht aber *notwendig*. Anders gesagt, man kann die Existenz eines \mathcal{P} -Programms auch nachweisen, *ohne* dieses Programm *konkret anzugeben*. Dies ist in der Mathematik nicht unüblich, denn ein Existenzbeweis erfordert nicht unbedingt, daß das existierende Objekt auch aus

⁴ Präzise formuliert ist bei Verwendung von \mathcal{P} -Programmen mit “Berechnungsformalismus” die *Abarbeitung der Aufrufe* von \mathcal{P} -Programmen gemäß Definition 2.8 gemeint.

dem Beweis rekonstruiert werden kann. In diesem Fall spricht man von einem *unkonstruktiven* Beweis. Dazu zwei einfache Beispiele:

Angenommen, wir wollen die Aussage “ $\forall i \in \mathbb{N} \exists j \in \mathbb{N}. j > i$ ” beweisen. Der Beweis ist offensichtlich – man setzt einfach $j := i + 1$. Dieser Beweis ist *konstruktiv*, denn aus dem Beweis können wir eine “Lösung” für j , nämlich $i + 1$, sofort ablesen.⁵

Als Beispiel für einen *unkonstruktiven* Beweis betrachten wir die Aussage “ $\exists q, r \in \overline{\mathbb{Q}}. q^r \in \mathbb{Q}$ ”, wobei \mathbb{Q} die Menge der rationalen Zahlen und $\overline{\mathbb{Q}}$ die Menge der irrationalen Zahlen bezeichnet. Zum Beweis unterscheiden wir folgende Fälle:

1. *Fall* $\sqrt{2}^{\sqrt{2}} \in \mathbb{Q}$: Dann gilt die Aussage für $q = r = \sqrt{2}$.
2. *Fall* $\sqrt{2}^{\sqrt{2}} \notin \mathbb{Q}$: Dann gilt $\sqrt{2}^{\sqrt{2}} \in \overline{\mathbb{Q}}$ und die Aussage gilt für $q = \sqrt{2}^{\sqrt{2}}$ und $r = \sqrt{2}$, denn $q^r = \sqrt{2}^{\sqrt{2}\sqrt{2}} = \sqrt{2}^2 = 2$.

Dieser Beweis ist *unkonstruktiv*, denn wir können eine “Lösung” für q und r nicht aus dem Beweis ermitteln.

Genauso wie in diesem Beispiel können wir beweisen, daß eine Funktion \mathcal{P} -berechenbar ist, ohne ein \mathcal{P} -Programm, das diese Funktion berechnet, *konkret* anzugeben.

Betrachten wir als Beispiel die Funktion $\xi : \mathbb{N} \rightarrow \mathbb{N}$ mit



$$\xi(n) := \begin{cases} 1 & \text{falls die Dezimalbruchentwicklung von } \pi \text{ mindestens} \\ & \text{ } n \text{ aufeinanderfolgende Ziffern 1 enthält} \\ 0 & \text{, andernfalls .} \end{cases}$$

Wir zeigen, daß ξ \mathcal{P} -berechenbar ist: Angenommen, in der Dezimalbruchentwicklung von π kommen höchstens k_0 aufeinanderfolgende Ziffern 1 vor. Dann gilt $\xi = \xi_1$ mit

$$\xi_1(n) := \begin{cases} 1 & \text{, falls } n \leq k_0 \\ 0 & \text{, falls } n > k_0 . \end{cases} \quad (3.1)$$

Andernfalls definieren wir $\xi_2(n) := 1$ und erhalten $\xi = \xi_2$. Es ist offensichtlich, daß sowohl ξ_1 als auch ξ_2 \mathcal{P} -berechenbar sind, etwa durch \mathcal{P} -Programme **PROG1**

⁵ Nicht-triviale Beispiele für konstruktive Beweise findet man z.B. in der Theorie der Formalen Sprachen. Beispielsweise läßt sich aus dem Beweis des Satzes “Für jeden nicht-deterministischen endlichen Automaten *NFA* existiert ein deterministischer endlicher Automat *DFA*, so daß beide Automaten die gleiche Sprache akzeptieren” unmittelbar ein algorithmisches Verfahren ablesen, das aus jedem *NFA* einen äquivalenten *DFA* synthetisiert.

und **PROG2**. Damit ist auch ξ \mathcal{P} -berechenbar. Allerdings haben wir auch hier einen *unkonstruktiven* Beweis geführt – die \mathcal{P} -Programme **PROG1** und **PROG2** kann man leicht angeben, nur ist aus dem Beweis nicht ersichtlich, ob wir **PROG1** oder **PROG2** verwenden müssen, um ξ zu berechnen.

Generell gilt, daß eine Funktion mit einer Fallunterscheidung, die nicht abhängig von den Parametern der Funktion ist, auch dann berechenbar sein kann, wenn nicht algorithmisch festgestellt werden kann, welcher der Fälle zutrifft.

Beispielsweise gilt für die Funktionen $\phi_1, \phi_2, \phi, \psi : \mathbb{N} \mapsto \mathbb{N}$ mit

$$\phi(n) := \begin{cases} \phi_1(n) & , \text{ falls } \forall m. \psi(m) \neq \perp \\ \phi_2(n) & , \text{ falls } \exists m. \psi(m) = \perp \end{cases}$$

daß ϕ berechenbar ist, falls ϕ_1 und ϕ_2 berechenbar sind: Entweder ist ψ *total*, und dann gilt $\phi(n) = \phi_1(n)$, oder aber ψ ist *nicht total*, und damit gilt dann $\phi(n) = \phi_2(n)$. In Abschnitt 10.1 werden wir zeigen, daß man nicht algorithmisch feststellen kann, ob eine Funktion (wie ψ) total ist oder nicht. Damit kann auch nicht algorithmisch festgestellt werden, ob nun $\phi(n) = \phi_1(n)$ oder $\phi(n) = \phi_2(n)$ gilt. Dies ändert jedoch nichts daran, daß ϕ (wegen ϕ_1 und ϕ_2) berechenbar ist.

Definiert man hingegen eine Funktion $\phi' : \mathbb{N} \mapsto \mathbb{N}$ durch

$$\phi'(n) := \begin{cases} \phi_1(n) & , \text{ falls } \psi(n) \neq \perp \\ \phi_2(n) & , \text{ falls } \psi(n) = \perp \end{cases}$$

so ist ϕ' selbst bei berechenbaren ϕ_1, ϕ_2 und ψ *nicht notwendigerweise* berechenbar, denn um $\phi'(n)$ zu bestimmen muß festgestellt werden, ob $\psi(n) = \perp$ gilt oder nicht. In Abschnitt 5.3 zeigen wir, daß nicht algorithmisch feststellbar ist, ob eine berechenbare Funktion (wie ψ) für eine beliebige Eingabe $n \in \mathbb{N}$ ein Ergebnis $m \in \mathbb{N}$ berechnet. Damit kann ϕ' nicht berechenbar sein.

Der Unterschied zwischen ϕ und ϕ' besteht also darin, daß die Fallunterscheidung in der Definition von ϕ' *abhängig vom Parameter n* der Funktion ϕ' ist. Solch eine Abhängigkeit führt jedoch nicht notwendigerweise zu nicht-Berechenbarkeit: Wenn algorithmisch feststellbar ist, welcher der Fälle zutrifft, so kann eine solchermaßen definierte Funktion durchaus berechenbar sein – die Funktion ξ_1 aus (3.1) ist ein Beispiel dafür.

4. GÖDELISIERUNGEN

4.1. Kodierung von Datentypen

Da wir uns bei Definition der Programmiersprache \mathcal{P} auf die natürlichen Zahlen \mathbb{N} als einzigem Datentyp beschränkt haben, stellt sich die Frage, wie die Berechenbarkeit von nicht-arithmetischen Funktionen, also Funktionen, die nicht auf \mathbb{N} definiert sind, untersucht werden kann.

Beispielsweise kann man Listen von natürlichen Zahlen mit dem *Quicksort*-Algorithmus ordnen. Die Funktion $sort : \mathbb{N}^* \rightarrow \mathbb{N}^*$, die jeder Liste l eine geordnete Permutation von l zuordnet, ist also sicher intuitiv berechenbar. Wir können sogar ein konkretes \mathcal{P} -Programm für das Sortieren von Listen angeben, indem wir Listen von natürlichen Zahlen durch natürliche Zahlen *kodieren*. Man nennt solche Kodierungen auch nach K. Gödel *Gödelisierungen*.

Um eine Liste $\langle n_1, \dots, n_k \rangle$ durch eine natürliche Zahl zu kodieren, ordnen wir dieser Liste mit der Funktion $\lambda : \mathbb{N}^* \rightarrow \mathbb{N}$ die natürliche Zahl $\lambda(\langle n_1, \dots, n_k \rangle) :=$

$$prim(1)^{n_k+1} prim(2)^{n_{k-1}+1} \dots prim(k-1)^{n_2+1} prim(k)^{n_1+1}$$

zu, wobei $prim(n)$ die n -te Primzahl bezeichnet. Wir nennen $\lambda(\langle n_1, \dots, n_k \rangle)$ die *Kodierung* von $\langle n_1, \dots, n_k \rangle$. Die leere Liste $\langle \rangle$ wird mit 1 kodiert, d.h. $\lambda(\langle \rangle) = 1$.

Da $prim(n)$, x^y und die Multiplikation natürlicher Zahlen \mathcal{P} -berechenbar sind, ist die Funktion λ *algorithmisch*. Mit "algorithmisch" ist gemeint, daß es ein algorithmisches Verfahren gibt, mit dem $\lambda(\langle n_1, \dots, n_k \rangle)$ aus einer Eingabe $\langle n_1, \dots, n_k \rangle$ ermittelt wird. Wir können λ nicht *\mathcal{P} -berechenbar* nennen, da wir diesen Begriff für die *arithmetischen* Funktionen $\mathbb{N}^k \mapsto \mathbb{N}$ reserviert haben.

Wegen der Eindeutigkeit der Primfaktorzerlegung ist die Funktion λ *injektiv*, und damit werden verschiedenen Listen l_1 und l_2 verschiedene Kodierungen $\lambda(l_1)$ und $\lambda(l_2)$ zugeordnet. Jeder natürlichen Zahl $m \in \text{Bild}(\lambda)$ können wir daher mit der Umkehrfunktion $\lambda^{-1} : \mathbb{N} \mapsto \mathbb{N}^*$ von λ *genau eine* Liste $\langle n_1, \dots, n_k \rangle$ von natürlichen Zahlen zuordnen, d.h. $\lambda^{-1}(m) = \langle n_1, \dots, n_k \rangle$, falls $\lambda(\langle n_1, \dots, n_k \rangle) = m$. Beispielsweise gilt $\lambda(\langle 0, 1, 0, 2 \rangle) = 4200$ und $\lambda^{-1}(4200) = \langle 0, 1, 0, 2 \rangle$, denn $4200 = 2^3 \times 3^1 \times 5^2 \times 7^1$. Dagegen ist $\lambda^{-1}(6600)$ undefiniert, denn $6600 = 2^3 \times 3^1 \times 5^2 \times 11^1$, λ ist also nicht *surjektiv*. Dabei ist offensichtlich sowohl die Umkehrfunktion λ^{-1}

algorithmisch, als auch die Funktion $\lambda^? : \mathbb{N} \rightarrow \{0, 1\}$ mit $\lambda^?(m) = 1$ gdw. $m \in \lambda(\mathbb{N}^*)$.

Allgemein muß eine Kodierung $\gamma : D \rightarrow \mathbb{N}$ einer Menge D folgenden Eigenschaften genügen:¹

Definition 4.1. (Gödelisierung)

Sei D eine Menge und sei $\gamma : D \rightarrow \mathbb{N}$. Dann ist γ eine Gödelisierung von D genau dann, wenn gilt:

1. γ ist injektiv,
2. γ ist algorithmisch,
3. $\gamma^? : \mathbb{N} \rightarrow \{0, 1\}$ mit “ $\gamma^?(m) = 1$ gdw. $m \in \gamma(D)$ ” ist berechenbar, und
4. $\gamma^{-1} : \text{Bild}(\gamma) \rightarrow D$ ist algorithmisch. ■

Es ist offensichtlich, daß λ eine Gödelisierung von \mathbb{N}^* ist. Um jetzt Listen $\langle n_1, \dots, n_k \rangle$ mit einer \mathcal{P} -Prozedur zu sortieren, kodieren wir diese Listen zunächst mit λ durch natürliche Zahlen $\lambda(\langle n_1, \dots, n_k \rangle)$. Diese Kodierung dient dann als Eingabe für einen Sortieralgorithmus $\text{QSORT} \in \mathcal{P}[1]$, der als Ergebnis eine natürliche Zahl m berechnet. Das Ergebnis m des Prozeduraufrufs muß dann mit λ^{-1} in eine Liste von natürlichen Zahlen *dekodiert* werden. Damit kann man das *Quicksort*-Verfahren mittels eines \mathcal{P} -Programms QSORT angeben, für das

$$\lambda^{-1}([\text{QSORT}](\lambda(\langle n_1, \dots, n_k \rangle))) = \text{sort}(\langle n_1, \dots, n_k \rangle)$$

gilt. Da λ eine Gödelisierung und $[\text{QSORT}]$ \mathcal{P} -berechenbar ist, ist das Sortieren von Listen mit Angabe von QSORT ebenfalls algorithmisch lösbar.

Die Prozedur QSORT ist dabei sicher umständlich zu schreiben. Beispielsweise muß man die größte Primzahl $p \neq 1$ und eine natürliche Zahl n_1 mit $\lambda(\langle n_1, \dots, n_k \rangle) \bmod p^{n_1+1} = 0$ berechnen, um das erste Element n_1 einer Liste $\langle n_1, n_2, \dots, n_k \rangle$ zu ermitteln. Um die (Kodierung der) Restliste $\langle n_2, \dots, n_k \rangle$ zu berechnen, muß $\lambda(\langle n_1, \dots, n_k \rangle)$ durch p^{n_1+1} dividiert werden. Das zweite Element der Liste erhält man dann als erstes Element dieser Restliste, u.s.w. Dieser umständliche Zugriff auf Listenelemente ist der Preis dafür, daß wir nur \mathbb{N} als Datentyp in \mathcal{P} -Programmen zulassen.

¹ Genau genommen müssen wir von einer Gödelisierungs- bzw. einer Kodierungsfunktion γ sprechen, um γ von den Elementen in $\text{Bild}(\gamma)$ zu unterscheiden. Wir verzichten auf diese sprachliche Feinheit, wenn aus dem Kontext ersichtlich ist, was gerade genau gemeint ist.

| $n_1 \setminus n_2$ | 0 | 1 | 2 | 3 | 4 | ... |
|---------------------|----------|----------|----------|----------|----|-----|
| 0 | 0 | 1 | 3 | 6 | 10 | ... |
| 1 | 2 | 4 | 7 | 11 | 16 | ... |
| 2 | 5 | 8 | 12 | 17 | 23 | ... |
| 3 | 9 | 13 | 18 | 24 | 31 | ... |
| 4 | 14 | 19 | 25 | 32 | 40 | ... |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

Abbildung 4.1: Wertetabelle der Paar-Funktion π^2

Allgemein können wir jeden Datentyp durch natürliche Zahlen kodieren. Da jedes Element eines Datentyps im Speicher eines Rechners endlich darstellbar ist, kann man jedes dieser Elemente auch kodieren. Man stelle sich dazu vor, daß jedes Datenobjekt im Speicher eines Rechners durch eine endliche Bitfolge repräsentiert wird. Diese Bitfolge kann man als natürliche Zahl auffassen, wodurch man die Kodierung des Datenobjekts erhält. Die zugehörige Kodierungsfunktion erfüllt dabei offenbar die Forderungen, die mit Definition 4.1 an eine Gödelisierung gestellt werden.

Wesentlich dabei ist, daß Elemente eines Datentyps auch *endlich* darstellbar sind. Beispielsweise kann man keinen Datentyp für die Menge \mathbb{R} der reellen Zahlen definieren, denn reelle Zahlen sind i.A. *unendlich große* Objekte. Man behilft sich deswegen in Programmiersprachen mit einem Datentyp `real` $\subsetneq \mathbb{R}$, der eine *endliche Approximation* der reellen Zahlen darstellt. Damit sind Elemente des Datentyps `real` endliche Objekte und somit auch durch natürliche Zahlen kodierbar.

4.2. Die Paar-Funktion

In diesem Abschnitt betrachten wir eine Gödelisierung von $\mathbb{N} \times \mathbb{N}$, die für weitere Untersuchungen nützlich ist.

Definition 4.2. (Paar-Funktion)

Die Paar-Funktion (*engl.* pairing function) $\pi^2: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ ist definiert durch

$$\pi^2(n_1, n_2) := \frac{1}{2}(n_1 + n_2)(n_1 + n_2 + 1) + n_1. \quad \blacksquare$$

Abbildung 4.1 gibt einige Funktionswerte der Paar-Funktion an.

| n | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
|--------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-----|
| (n_1, n_2) | (0,0) | (0,1) | (1,0) | (0,2) | (1,1) | (2,0) | (0,3) | (1,2) | (2,1) | ... |

Abbildung 4.2: Wertetabelle der Umkehrfunktion π^{-2} von π^2

Satz 4.3. Die Paar-Funktion π^2 ist

1. \mathcal{P} -berechenbar,
2. eine Gödelisierung von $\mathbb{N} \times \mathbb{N}$,
3. surjektiv.

Beweis. Übung. ■

Mit Satz 4.3(1) dürfen wir im folgenden also die Existenz einer \mathcal{P} -Prozedur PAIR^2 mit $[[\text{PAIR}^2]] = \pi^2$ voraussetzen. Mit Satz 4.3(2,3) ist π^2 *bijektiv*, und damit kommt jede natürliche Zahl n in Abbildung 4.1 *genau* einmal vor. Damit können wir jeder natürlichen Zahl n ein Paar (n_1, n_2) von natürlichen Zahlen *eindeutig* zuordnen, nämlich dasjenige Paar (n_1, n_2) , für das $\pi^2(n_1, n_2) = n$ gilt. Daraus folgt, daß die Umkehrfunktion $\pi^{-2} : \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ von π^2 , s. Abbildung 4.2, ebenfalls *bijektiv* ist.² Mit Satz 4.3(2) ist π^{-2} auch algorithmisch, und folglich sind die Funktionen $\pi_1^2 : \mathbb{N} \rightarrow \mathbb{N}$ und $\pi_2^2 : \mathbb{N} \rightarrow \mathbb{N}$ mit

$$\pi_1^2(\pi^2(n_1, n_2)) = n_1, \quad \pi_2^2(\pi^2(n_1, n_2)) = n_2 \quad \text{und} \quad \pi^2(\pi_1^2(n), \pi_2^2(n)) = n \quad (4.1)$$

berechenbar. Abbildung 4.3 gibt einige Funktionswerte von π_1^2 und π_2^2 an. Wir zeigen, daß auch π_1^2 und π_2^2 \mathcal{P} -berechenbar sind:

Satz 4.4. $\pi_1^2 : \mathbb{N} \rightarrow \mathbb{N}$ und $\pi_2^2 : \mathbb{N} \rightarrow \mathbb{N}$ mit (4.1) sind \mathcal{P} -berechenbar.

Beweis. Mit Satz 4.3(1) ist π^2 \mathcal{P} -berechenbar und damit existiert ein Programm $\text{PAIR}^2 \in \mathcal{P}[2]$ mit $[[\text{PAIR}^2]] = \pi^2$. Sei $\text{PAIR}_1^2 \in \mathcal{P}[1]$ definiert wie in Abbildung 4.4. Dieses Programm erzeugt systematisch eine Folge $(0, 0); (1, 0), (1, 1); (2, 0), (2, 1), (2, 2); (3, 0), \dots$ von Paaren (n_1, n_2) natürlicher Zahlen mit $n_1 \geq n_2$. Jedes Paar wird mit dem Programm PAIR^2 daraufhin überprüft, ob $\pi^2(n_1, n_2) = n$ gilt. Verläuft der Test positiv, so gilt $\pi_1^2(n) = n_1$. Andernfalls wird $\pi^2(n_2, n_1) = n$

² Aus Gründen der Lesbarkeit schreiben wir π^{-2} anstatt des formal korrekten $(\pi^2)^{-1}$.

überprüft, denn dann gilt $\pi_1^2(n) = n_2$. Mit $\text{stop} := 1$ wird für beide Schleifen der Abbruch erzwungen, wenn einer der Tests positiv verläuft, und PAIR_1^2 hält mit Ergebnis n_1 bzw. n_2 , also mit $\pi_1^2(n)$. Da (1) mit der **repeat**-Schleife alle Paare $(n_1, n_2) \in \mathbb{N} \times \mathbb{N}$ mit $n_1 \geq n_2$ erzeugt werden, (2) für alle Paare $(n_1, n_2) \in \mathbb{N} \times \mathbb{N}$ entweder $n_1 \geq n_2$ oder $n_2 \geq n_1$ gilt und (3) π^2 surjektiv ist, wird entweder der Test $\text{PAIR}^2(\mathbf{n1}, \mathbf{n2}) = \mathbf{n}$ oder der Test $\text{PAIR}^2(\mathbf{n2}, \mathbf{n1}) = \mathbf{n}$ nach endlich vielen Schritten positiv beantwortet, und damit hält PAIR_1^2 für jede Eingabe n . Folglich gilt $\llbracket \text{PAIR}_1^2 \rrbracket = \pi_1^2$, und damit ist π_1^2 \mathcal{P} -berechenbar.

Man erhält ein Programm $\text{PAIR}_2^2 \in \mathcal{P}[1]$ mit $\llbracket \text{PAIR}_2^2 \rrbracket = \pi_2^2$ aus dem Programm PAIR_1^2 , indem man lediglich die Zuweisungen $\text{res} := \mathbf{n1}$ und $\text{res} := \mathbf{n2}$ im Rumpf von PAIR_1^2 vertauscht. ■

Mit der Paar-Funktion können wir auch endliche Listen (beliebiger Länge) von natürlichen Zahlen und damit dann insbesondere (in Abschnitt 4.3) die Listen der aktuellen Parameter von Prozeduraufrufen gödelisieren.

Bemerkung 4.5. Allgemein definiert man den abstrakten Datentyp $\text{list}[\mathbb{N}]$ durch die Funktionssignatur

$$\begin{array}{ll} \varepsilon : \rightarrow \text{list}[\mathbb{N}] & \text{hd} : \text{list}[\mathbb{N}] \rightarrow \mathbb{N} \\ \text{add} : \mathbb{N} \times \text{list}[\mathbb{N}] \rightarrow \text{list}[\mathbb{N}] & \text{tl} : \text{list}[\mathbb{N}] \rightarrow \text{list}[\mathbb{N}] \end{array}$$

sowie die Gleichungen

$$\begin{array}{ll} \text{hd}(\varepsilon) = 0 & \text{hd}(\text{add}(n, k)) = n \\ \text{tl}(\varepsilon) = \varepsilon & \text{tl}(\text{add}(n, k)) = k \end{array} \quad (4.2)$$

Dabei steht ε für die leere Liste, mit add wird aus einer natürlichen Zahl und einer Liste eine neue Liste gebildet, mit hd (für head) erhält man das erste Element einer nicht-leeren Liste, und mit tl (für tail) die Liste ohne das erste Listenelement. Die Liste $\langle 0, 1, 0 \rangle$ wird beispielsweise als $\text{add}(0, \text{add}(1, \text{add}(0, \varepsilon)))$ geschrieben.

Mit der Paar-Funktion kodieren wir Listen durch natürliche Zahlen. Dabei übernimmt π^2 die Rolle von add , 0 kodiert ε , π_1^2 entspricht hd und π_2^2 steht für tl . Da in \mathcal{P} nur die natürlichen Zahlen als Datentyp zur Verfügung stehen, repräsentiert eine natürliche Zahl sowohl eine natürliche Zahl als auch eine Liste (von natürlichen Zahlen). Mit $\pi^2(0, 0) = 0$ darf 0 jedoch keine natürliche Zahl repräsentieren, denn man erhielte sonst 0 sowohl als Kodierung von ε als auch als Kodierung von $\text{add}(0, \varepsilon)$, also der Liste $\langle 0 \rangle$. Anders gesagt, man kann bei Kodierung einer Liste $\langle n_1, n_2, \dots, n_k \rangle$ durch $\pi^2(n_1, \pi^2(n_2, \pi^2(\dots, \pi^2(n_{k-1}, n_k) \dots)))$ nicht das Listenende erkennen. Um dieses Problem zu umgehen, werden Listen $\langle n_1, n_2, \dots, n_k \rangle$ durch $\pi^2(n_1 + 1, \pi^2(n_2 + 1, \pi^2(\dots, \pi^2(n_k + 1, 0) \dots)))$ kodiert. Bei dieser Kodierung von Listen ist unmittelbar einsichtig, daß mit (4.1) die Gleichungen (4.2) für die kodierten Listen auch tatsächlich gelten.

| n | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
|--------------|---|---|---|---|---|---|---|---|---|-----|
| $\pi_1^2(n)$ | 0 | 0 | 1 | 0 | 1 | 2 | 0 | 1 | 2 | ... |
| $\pi_2^2(n)$ | 0 | 1 | 0 | 2 | 1 | 0 | 3 | 2 | 1 | ... |

Abbildung 4.3: Wertetabelle der Funktionen π_1^2 und π_2^2

```

procedure PAIR12(n) <=
begin var n1,n2,stop,res;
repeat
n2 := 0;
while GE(n1,n2) ∧ ¬stop do
if PAIR2(n1,n2) = n
then res := n1;
stop := 1
else if PAIR2(n2,n1) = n
then res := n2;
stop := 1
else n2 := SUCC(n2)
end_if
end_if
n1 := SUCC(n1)
until stop;
return(res)
end

```

Abbildung 4.4: Ein \mathcal{P} -Programm für π_1^2

Definition 4.6. (Verallgemeinerte Paar-Funktion)

Die verallgemeinerte Paar-Funktion $\pi^k: \mathbb{N}^k \rightarrow \mathbb{N}$ ist für jedes $k \geq 1$ definiert durch

- $\pi^1(n_1) := n_1,$
- $\pi^2(n_1, n_2) := \frac{1}{2}(n_1 + n_2)(n_1 + n_2 + 1) + n_1,$ und
- $\pi^{k+3}(n_1, \dots, n_{k+3}) := \pi^2(n_1, \pi^{k+2}(n_2, \dots, n_{k+3})).$ ³ ■

Satz 4.7. Für jedes $k \geq 1$ ist die verallgemeinerte Paar-Funktion $\pi^k: \mathbb{N}^k \rightarrow \mathbb{N}$

1. \mathcal{P} -berechenbar,
2. eine Gödelisierung von \mathbb{N}^k
3. surjektiv.

Beweis. Für $k = 1$ ist die Aussagen trivial. Für $k = 2$ ist die Behauptung bereits mit Satz 4.3 gezeigt. Für $k = k' + 3$ zeigt man die Behauptung durch Induktion über k' . ■

Mit Satz 4.7 dürfen wir im folgenden also für jedes $k \geq 1$ die Existenz einer \mathcal{P} -Prozedur PAIR^k mit $[[\text{PAIR}^k]] = \pi^k$ voraussetzen.

Satz 4.8. Für $k \geq 1$ und $h \in \{1, \dots, k\}$ existieren \mathcal{P} -berechenbare Funktionen $\pi_h^k: \mathbb{N} \rightarrow \mathbb{N}$ mit

1. $\pi_h^k(\pi^k(n_1, \dots, n_k)) = n_h,$ und
2. $\pi^k(\pi_1^k(n), \dots, \pi_h^k(n), \dots, \pi_k^k(n)) = n.$

Beweis. Für $k = 1$ ist die Aussage trivial. Für $k = 2$ ist die Behauptung bereits mit Satz 4.4 gezeigt. Für $k = k' + 3$ zeigt man die Behauptung durch Induktion über k' . ■

Mit Satz 4.8 dürfen wir im folgenden also für jedes $k \geq 1$ und jedes $h \in \{1, \dots, k\}$ die Existenz einer \mathcal{P} -Prozedur PAIR_h^k mit $[[\text{PAIR}_h^k]] = \pi_h^k$ voraussetzen.

Bemerkung 4.9. Mit der verallgemeinerten Paar-Funktion π^k können wir Listen $\langle n_1, \dots, n_k \rangle$ durch $\pi^k(n_1, \dots, n_k)$ – anstatt durch $\pi^2(n_1+1, \pi^2(n_2+1, \pi^2(\dots, \pi^2(n_k+1, 0) \dots)))$, vgl. Bemerkung 4.5 – kodieren, denn für $\pi_h^k(n) = 0$ kann n nur die Kodierung einer Liste $\langle n_1, n_2, \dots, n_k \rangle$ mit $n_h = 0$ sein. Das Listenende muß hier nicht gesondert durch 0 gekennzeichnet werden, da mit π^k nur Listen einer festen Länge k kodiert werden.

³ Der Fall $k = 0$ wird absichtsvoll ausgeschlossen, denn π^0 könnte nur eine Konstante sein, also eine Funktion die nicht injektiv (und auch nicht surjektiv) ist, und deshalb keine Gödelisierung sein kann.

4.3. Stelligkeit berechenbarer Funktionen

Mit der verallgemeinerten Paar-Funktion π^k kann man jedes k -Tupel (n_1, \dots, n_k) natürlicher Zahlen eindeutig durch eine natürliche Zahl kodieren und jede Komponente n_h dieses k -Tupels mit π_h^k zurückgewinnen, vgl. Bemerkung 4.9. Damit können wir jede \mathcal{P} -berechenbare Funktion $\phi : \mathbb{N}^k \mapsto \mathbb{N}$ durch eine *einstellige* \mathcal{P} -berechenbare Funktion $\phi' : \mathbb{N} \mapsto \mathbb{N}$ darstellen:

Satz 4.10. *Für jede \mathcal{P} -berechenbare Funktion $\phi : \mathbb{N}^k \mapsto \mathbb{N}$ existiert eine \mathcal{P} -berechenbare Funktion $\phi' : \mathbb{N} \mapsto \mathbb{N}$, so daß gilt:*

1. $\phi'(n) = \phi(\pi_1^k(n), \dots, \pi_h^k(n), \dots, \pi_k^k(n))$ und
2. $\phi(n_1, \dots, n_k) = \phi'(\pi^k(n_1, \dots, n_k))$.⁴

Beweis. (1) Mit Satz 4.8 gibt es Prozeduren $\text{PAIR}_1^k, \dots, \text{PAIR}_k^k \in \mathcal{P}[1]$ so daß $\pi_h^k = \llbracket \text{PAIR}_h^k \rrbracket$ für jedes $h \in \{1, \dots, k\}$ gilt. Sei $\text{PHI} \in \mathcal{P}[k]$ mit $\llbracket \text{PHI} \rrbracket = \phi$ gegeben durch

```

procedure PHI(x1, ..., xk) <=
begin var y1, ..., yn;
  STA $\phi$ ;
  return(y $\phi$ )
end .

```

Für die Prozedur $\text{PHI}' \in \mathcal{P}[1]$, gegeben durch

```

procedure PHI'(x) <=
begin var y;
  y := PHI(PAIR $_1^k$ (x), ..., PAIR $_k^k$ (x));
  return(y)
end

```

gilt dann $\llbracket \text{PHI}' \rrbracket(n) = \llbracket \text{PHI} \rrbracket(\llbracket \text{PAIR}_1^k \rrbracket(n), \dots, \llbracket \text{PAIR}_k^k \rrbracket(n))$ für alle $n \in \mathbb{N}$, also $\llbracket \text{PHI}' \rrbracket(n) = \phi(\pi_1^k(n), \dots, \pi_k^k(n))$. Folglich gilt $\llbracket \text{PHI}' \rrbracket = \phi'$, und damit ist ϕ' \mathcal{P} -berechenbar.

(2) Folgt unmittelbar aus (1) und Satz 4.8(1). ■

Als Konsequenz von Satz 4.10 können wir immer davon ausgehen, daß jede \mathcal{P} -berechenbare Funktion ϕ *einstellig* ist, also durch ein Programm aus $\mathcal{P}[1]$ berechnet werden kann. Dies kann man sich folgendermaßen veranschaulichen:

⁴ Im Fall $k = 0$ liest man 1. $\phi'(n) = \phi$ sowie 2. $\phi = \phi'(n)$.



Ein Programm $\text{PROG} \in \mathcal{P}[k]$ erwartet eine Liste (n_1, \dots, n_k) aktueller Parameter mit k Elementen als Eingabe. Diese Liste können wir mit der verallgemeinerten Paar-Funktion π^k kodieren.⁵ Wir erhalten so eine $\mathcal{P}[1]$ -Prozedur PROG' , in deren Rumpf dann mittels der $\mathcal{P}[1]$ -Prozeduren PAIR_h^k auf die aktuellen Parameter n_h der kodierten Parameterliste zugegriffen werden kann, vgl. die Prozedur PHI' in Satz 4.10.

Für die $\mathcal{P}[2]$ -Prozedur MINUS aus Abbildung 2.3 erhält man beispielsweise die $\mathcal{P}[1]$ -Prozedur

```

procedure MINUS'(x) <=
begin var n,m;
  n := PAIR12(x); m := PAIR22(x);
  while m do
    n := PRED(n); m := PRED(m)
  end_while;
  return(n)
end .

```

Um jetzt etwa $3-2$ zu berechnen, wird – anstatt die Prozedur MINUS mit den aktuellen Parametern 3 und 2 aufzurufen – die Prozedur MINUS' mit dem aktuellen Parameter 18 aufgerufen, denn $\pi^2(3, 2) = 18$. In der ersten Zuweisung der Prozedur MINUS' erhält n dann den Wert 3 ($= \pi_1^2(18)$), und in der zweiten Zuweisung erhält m den Wert 2 ($= \pi_2^2(18)$). Jetzt wird 2 in der *while*-Schleife von 3 subtrahiert, und das Ergebnis des Prozeduraufrufs $\text{MINUS}'(18)$ ist 1.

Verwendet man nur $\mathcal{P}[1]$ -Programme, so stellt sich die Frage, wie denn im Beispiel der aktuelle Parameter 18 als Kodierung der Parameterliste $(3, 2)$ berechnet werden soll. Bei Aufruf einer $\mathcal{P}[k]$ -Prozedur $\text{PROG}(n_1, \dots, n_k)$ im Rumpf eines \mathcal{P} -Programms P geschieht dies mittels der \mathcal{P} -Prozedur PAIR^k , d.h. der Prozeduraufruf wird im Rumpf von P durch $\text{PROG}'(\text{PAIR}^k(n_1, \dots, n_k))$ ersetzt. Im Rumpf der Prozedur GGT aus Abbildung 2.3 würden daher die Prozeduraufrufe $\text{MINUS}(n, m)$ und $\text{MINUS}(m, n)$ durch $\text{MINUS}'(\text{PAIR}^2(n, m))$ bzw. $\text{MINUS}'(\text{PAIR}^2(m, n))$ ersetzt. Da wir Prozeduraufrufe immer eliminieren können, vgl. Definition 2.2, ist die Verwendung der $\mathcal{P}[k]$ -Prozedur PAIR^k hier unkritisch.

Bei einem $\mathcal{P}[k]$ -Programm als “*Hauptprogramm*”, etwa dem $\mathcal{P}[2]$ -Programm GGT aus Abbildung 2.4 oder dem $\mathcal{P}[2]$ -Programm MINUS (als “*Hauptprogramm*” und nicht als Hilfsprozedur aufgefaßt) können wir jedoch nicht so vorgehen, denn für diese Programme kennen wir ja keine Aufrufe, die wir modifizieren könnten. Anders gesagt, diese Programme werden *außerhalb* unserer \mathcal{P} -Programme auf-

⁵ Dies gilt natürlich nur für den Fall $k \geq 1$, denn π^0 ist nicht definiert. Die Anpassung der nachfolgenden Diskussion auf den Fall $k = 0$ ist jedoch trivial, so daß wir darauf verzichten.

gerufen, also muß *dort* die Kodierung der aktuellen Parameter $\mathbf{n1}$ und $\mathbf{n2}$ zu $\pi^2(\mathbf{n1}, \mathbf{n2})$ vorgenommen werden. Für uns ist daher belanglos, wie die Kodierung der Parameterliste $(3, 2)$ mittels π^2 zu 18 berechnet wird, wenn etwa $ggt(3, 2)$ oder $3-2$ mittels der $\mathcal{P}[1]$ -Programme \mathbf{GGT}' bzw. \mathbf{MINUS}' berechnet werden sollen. Diese beiden $\mathcal{P}[1]$ -Programme *interpretieren* 18 als Kodierung der Liste $(3, 2)$, denn die Programme aus $\mathcal{P}[1]$ arbeiten anstatt direkt auf natürlichen Zahlen auf kodierten Listen von natürlichen Zahlen (die selbst wieder natürliche Zahlen sind). Wie diese Kodierungen genau berechnet werden ist für unsere Untersuchungen genauso unerheblich wie die Frage, wie die Gödelisierung irgendeines Datentyps D nun genau berechnet wird. Wesentlich ist nur, daß die Kodierungsfunktion π^k für \mathbb{N}^k den Forderungen von Definition 4.1 für eine Gödelisierung genügt, und dies ist mit Satz 4.7 der Fall.

Damit bleibt nur noch die Frage, wie denn ein $\mathcal{P}[1]$ -Programm $\mathbf{PAIR}^{2'}$ für \mathbf{PAIR}^2 aussehen könnte. Da wir behaupten, daß jede k -stellige berechenbare Funktion als 1-stellige berechenbare Funktion dargestellt werden kann, so muß es ja ein $\mathcal{P}[1]$ -Programm $\mathbf{PAIR}^{2'}$ mit (*) $\llbracket \mathbf{PAIR}^{2'} \rrbracket(n_1, n_2) = \pi^2(n_1, n_2) = \llbracket \mathbf{PAIR}^{2'} \rrbracket(\pi^2(n_1, n_2))$ geben. Dies gilt offenbar, falls $\llbracket \mathbf{PAIR}^{2'} \rrbracket$ die Identitätsfunktion ist, also definieren wir

```

procedure  $\mathbf{PAIR}^{2'}(\mathbf{x}) <=$ 
begin var  $\mathbf{y}$ ;  $\mathbf{y} := \mathbf{x}$ ; return( $\mathbf{y}$ ) end .

```

Wir werden daher im folgenden arithmetische Funktionen $\phi : \mathbb{N}^k \mapsto \mathbb{N}$ in vielen Fällen anstatt in der Form $\phi(n_1, \dots, n_k) := \mathcal{DEF}[n_1, \dots, n_k]$ in der Form $\phi : \mathbb{N} \mapsto \mathbb{N}$ mit $\phi(\pi^k(n_1, \dots, n_k)) := \mathcal{DEF}[n_1, \dots, n_k]$ definieren, wobei dann mit Satz 4.8 $\phi(n) = \mathcal{DEF}[\pi_1^k(n), \dots, \pi_k^k(n)]$ gilt.

4.4. Kodierung von Programmen

Mit Programmen irgendeiner Programmiersprache PS können auch Programme (dieser Programmiersprache) verarbeitet werden. Beispielsweise können wir einen Übersetzer in der Programmiersprache PS schreiben, der PS -Programme als Eingabe erhält und diese dann in eine Zielsprache übersetzt. Mit einem in PS geschriebenen Editor können wir PS -Programme editieren, mit einem in PS geschriebenen Datenkompressionsprogramm können wir PS -Programme platzsparend abspeichern, mit einem in PS geschriebenen Interpretierer können wir PS -Programme ausführen, u.s.w.

In einigen Programmiersprachen, wie z.B. Assemblersprachen oder LISP, werden Programme und Daten überhaupt nicht unterschieden, so daß die Verarbeitung von Programmen durch Programme direkt implementiert werden kann. In

anderen Sprachen, wie z.B. JAVA oder C, werden Programme und Daten unterschiedlich behandelt. Hier müssen dann Programme durch einen bestimmten Datentyp, wie z.B. den Datentyp *string* oder als Feld von Zeichen (*array integer of character*), kodiert werden, um Programme durch Programme zu verarbeiten.

Die Fähigkeit von Programmen, Programme zu verarbeiten, ist wesentlich für die Ergebnisse der Berechenbarkeitstheorie. Wir kodieren daher (wie schon zuvor Datentypen) \mathcal{P} -Programme durch natürliche Zahlen, um \mathcal{P} -Programme mit \mathcal{P} -Programmen verarbeiten zu können:

\mathcal{P} -Programme sind endliche Worte über einem endlichen Alphabet $A_{\mathcal{P}}$, die den syntaktischen Konventionen von Definition 2.1 genügen. Es gilt also $\mathcal{P} \subseteq A_{\mathcal{P}}^*$. Das Alphabet $A_{\mathcal{P}}$ enthält alle Buchstaben in Groß- und Kleinschreibung, die Ziffern $0, \dots, 9$ und Sonderzeichen wie " $<$ ", " $=$ ", " $,$ ", " $:$ ", " $;$ ", " $;$ ", " $)$ ", " $($ ", \dots . Damit können wir die Elemente von $A_{\mathcal{P}}$ durch Bitworte über $\{0, 1\}$ mit einer festen Länge ℓ kodieren. Für $\ell = 8$ können wir beispielsweise 256 verschiedene Zeichen in $A_{\mathcal{P}}$ darstellen – das sollte reichen, um \mathcal{P} -Programme zu schreiben. Die Zuordnung von ℓ -Bit Worten zu den Symbolen von $A_{\mathcal{P}}$ legen wir durch eine endliche Kodetabelle fest. Diese Tabelle repräsentiert den Graphen einer Funktion $code : A_{\mathcal{P}} \rightarrow \{0, 1\}^{\ell}$. Da wir verschiedenen Symbolen in $A_{\mathcal{P}}$ verschiedene ℓ -Bit Worte zuordnen, ist $code$ injektiv. Damit können wir jetzt jedes \mathcal{P} -Programm $P = a_1 \dots a_k \in A_{\mathcal{P}}^*$ durch eine natürliche Zahl $\natural P := dual^{-1}(code(a_1) \oplus \dots \oplus code(a_k))$ gödelisieren, wobei $dual^{-1} : \{0, 1\}^+ \rightarrow \mathbb{N}$ eine natürliche Zahl aus deren Dualzahldarstellung berechnet (und damit z.B. $dual^{-1}(11010) = 26$ gilt).

Definition 4.11. (Programmkode $\natural P$) Sei

- $A_{\mathcal{P}} \neq \emptyset$ das endliche Alphabet von \mathcal{P} , d.h. $\mathcal{P} \subseteq A_{\mathcal{P}}^*$,
- $code : A_{\mathcal{P}} \rightarrow \{0, 1\}^{\ell}$ mit $\ell = \lceil \log_2(|A_{\mathcal{P}}|) \rceil$ eine injektive Funktion, die jedes Symbol aus $A_{\mathcal{P}}$ in ein ℓ -Bit Wort abbildet,⁶
- $dual_{\ell} : \mathbb{N} \rightarrow \{\{0, 1\}^{\ell}\}^+$ diejenige Funktion, die jede natürliche Zahl n in ihre in Bit-Blöcke der Länge ℓ eingeteilte Dualzahldarstellung $dual_{\ell}(n)$ abbildet, d.h. in eine endliche Zeichenreihe minimaler Länge von Bits 0 und 1, deren Länge ein Vielfaches von ℓ ist,⁷ und sei
- $\oplus : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^*$ diejenige Funktion, die zwei Bitworte α und β in ihre Konkatenation $\alpha\beta$ abbildet.

⁶ $\lceil \log_2(n) \rceil \in \mathbb{N}$ ist der nach oben gerundete Logarithmus von n zur Basis 2.

⁷ Dieser Forderung kann immer durch Voranstellen von 0-Bits genügt werden. Dies ist erforderlich, da sonst $dual(dual^{-1}(code(a_1) \oplus \dots \oplus code(a_k))) = code(a_1) \oplus \dots \oplus code(a_k)$ nicht garantiert wäre. Man erhielte so beispielsweise (für $\ell = 2$) $dual(dual^{-1}(01)) = dual(1) = 1 \neq 01$, während $dual_{\ell}(1) = 01$ gilt.

Dann ist die Programmkodierungsfunktion $\natural : \mathcal{P} \rightarrow \mathbb{N}$ von \mathcal{P} gegeben durch

$$\natural(a_1 \dots a_k) := \text{dual}^{-1}(\text{code}(a_1) \oplus \dots \oplus \text{code}(a_k)) .$$

Für $P \in \mathcal{P}$ schreiben wir $\natural P$ anstatt $\natural(P)$ und nennen $\natural P$ den Kode von P . Die Menge $\natural\mathcal{P}$ aller Kodes der Programmiersprache \mathcal{P} ist definiert durch $\natural\mathcal{P} := \{\natural P \in \mathbb{N} \mid P \in \mathcal{P}\}$.⁸ ■



Satz 4.12. Die Programmkodierungsfunktion \natural ist eine Gödelisierung von \mathcal{P} .

Beweis. (1) “ $\natural : \mathcal{P} \rightarrow \mathbb{N}$ ist injektiv”: Da alle Elemente von $\text{Bild}(\text{code})$ Bitworte der Länge ℓ und code nach Voraussetzung injektiv ist, gilt $\text{code}(a_1) \oplus \dots \oplus \text{code}(a_k) \neq \text{code}(a'_1) \oplus \dots \oplus \text{code}(a'_k)$ falls $a_1 \dots a_m \neq a'_1 \dots a'_k$. Mit der Bijektivität von dual_ℓ ist \natural dann injektiv.

(2) “ \natural ist algorithmisch”: Um $\natural(a_1 \dots a_k)$ für ein Wort $a_1 \dots a_k \in A_{\mathcal{P}}^*$ zu berechnen, bearbeitet man $a_1 \dots a_k$ Zeichen für Zeichen von links nach rechts und ersetzt dabei jedes Symbol a_i durch $\text{code}(a_i)$, so wie in der Kodetabelle angegeben. Dieses Verfahren ist offenbar algorithmisch. Die ℓ -Bit Worte $\text{code}(a_i)$ werden mit \oplus zu einem Bitwort $\text{code}(a_1) \oplus \dots \oplus \text{code}(a_k)$ konkateniert, was auch algorithmisch bewerkstelligt werden kann. Schließlich wird $\text{code}(a_1) \oplus \dots \oplus \text{code}(a_k)$ in die natürliche Zahl $\text{dual}^{-1}(\text{code}(a_1) \oplus \dots \oplus \text{code}(a_k))$ umgewandelt. Auch diese Umwandlung ist algorithmisch, und damit ist auch \natural algorithmisch.

(3) “ $\natural? : \mathbb{N} \rightarrow \{0, 1\}$ mit ‘ $\natural?(n) = 1$ gdw. $n \in \natural\mathcal{P}$ ’ ist berechenbar”: Die Programmiersprache \mathcal{P} ist eine sehr einfache Programmiersprache, die nur Konstrukte enthält, die wir (mitunter in syntaktischen Abwandlungen) bereits aus implementierten Programmiersprachen kennen. Damit dürfen wir davon ausgehen, daß man (in \mathcal{P}) einen Syntaxanalysator (engl. Parser) $\text{parse} : \{\mathbf{O}, \mathbf{I}\}^+ \rightarrow \{0, 1\}$ für \mathcal{P} implementieren kann, für den $\text{parse}(\text{code}(a_1) \oplus \dots \oplus \text{code}(a_k)) = 1$ für alle $a_1 \dots a_k \in \mathcal{P}$ und $\text{parse}(\text{code}(a_1) \oplus \dots \oplus \text{code}(a_k)) = 0$ für alle $a_1 \dots a_k \in A_{\mathcal{P}}^* \setminus \mathcal{P}$ gilt, und man erhält damit $\natural?(n) := \text{parse}(\text{dual}_\ell(n))$. Da dual_ℓ und parse algorithmisch sind, ist $\natural?$ berechenbar.

Für $\natural?(n) = 1$ gilt $\text{dual}_\ell(n) = \alpha_1 \oplus \dots \oplus \alpha_k$ mit $\alpha_i \in \{\mathbf{O}, \mathbf{I}\}^\ell$ für jedes α_i sowie $\text{parse}(\alpha_1 \oplus \dots \oplus \alpha_k) = 1$. Mit der Definition von parse gilt $\text{code}^{-1}(\alpha_1) \dots \text{code}^{-1}(\alpha_k) \in \mathcal{P}$, und folglich $\natural(\text{code}^{-1}(\alpha_1) \dots \text{code}^{-1}(\alpha_k)) \in \natural\mathcal{P}$. Mit der Definition von \natural erhält man dann $n \in \natural\mathcal{P}$, denn

$$\begin{aligned} & \natural(\text{code}^{-1}(\alpha_1) \dots \text{code}^{-1}(\alpha_k)) \\ &= \text{dual}^{-1}(\text{code}(\text{code}^{-1}(\alpha_1)) \oplus \dots \oplus \text{code}(\text{code}^{-1}(\alpha_k))) \\ &= \text{dual}^{-1}(\alpha_1 \oplus \dots \oplus \alpha_k) \\ &= n . \end{aligned} \tag{4.3}$$

⁸ Bei der Definition von $\natural\mathcal{P}$ gehen wir davon aus, daß \mathcal{P} nach Definition 2.1 gegeben ist, also insbesondere keine Prozeduraufrufe enthält.

Für $\mathfrak{h}^{-1}(n) \neq 1$ gilt $\text{parse}(\alpha_1 \oplus \dots \oplus \alpha_k) = 0$, wobei $\alpha_1 \oplus \dots \oplus \alpha_k = \text{dual}_\ell(n)$ und $\alpha_i \in \{0, 1\}^\ell$ für jedes α_i gilt. Also gilt $\text{code}^{-1}(\alpha_1) \dots \text{code}^{-1}(\alpha_k) \notin \mathcal{P}$ mit der Definition von parse , und folglich $n \notin \mathfrak{h}\mathcal{P}$ mit (4.3). Damit sind auch die Forderungen erfüllt, die an \mathfrak{h}^{-1} gestellt werden.

(4) “ $\mathfrak{h}^{-1} : \text{Bild}(\mathfrak{h}) \rightarrow \mathcal{P}$ ist algorithmisch”: Mit dual_ℓ kann jede natürliche Zahl algorithmisch in ihre Dualzahldarstellung überführt werden. Für $n \in \mathfrak{h}\mathcal{P}$ erhält man so $\mathfrak{h}^{-1}(n) = \alpha_1 \oplus \dots \oplus \alpha_k$ für ein $k \in \mathbb{N}$, wobei $\alpha_i \in \{0, 1\}^\ell$ für jedes α_i gilt, denn $n \in \mathfrak{h}\mathcal{P}$. Das Bitwort $\alpha_1 \oplus \dots \oplus \alpha_k$ bearbeitet man jetzt in ℓ -Bit Abständen von links nach rechts und ersetzt dabei mittels der Kodetabelle jedes ℓ -Bit Wort α_i durch $\text{code}^{-1}(\alpha_i)$. Auch dieses Verfahren ist algorithmisch, und damit ist \mathfrak{h}^{-1} mit $\mathfrak{h}^{-1}(n) := \text{code}^{-1}(\alpha_1) \dots \text{code}^{-1}(\alpha_k)$ auch algorithmisch. ■

Da die Programmkodierungsfunktion \mathfrak{h} eine Gödelisierung von \mathcal{P} ist, ist \mathfrak{h} *injektiv*, d.h. $P = P'$ falls $\mathfrak{h}P = \mathfrak{h}P'$ für je zwei Programme $P, P' \in \mathcal{P}$. Damit werden *verschiedenen Programmen* durch \mathfrak{h} *verschiedene Codes* zugeordnet. Die Programmkodierungsfunktion \mathfrak{h} ist jedoch nicht *surjektiv*, d.h. es gibt natürliche Zahlen n mit $\mathfrak{h}P \neq n$ für alle $P \in \mathcal{P}$, und damit gilt $\mathfrak{h}\mathcal{P} \subsetneq \mathbb{N}$. Dies ist einleuchtend, denn andernfalls würde ja *jedes* Bitwort ein Programm repräsentieren. Für eine beliebige natürliche Zahl n können wir jedoch mit \mathfrak{h}^{-1} feststellen, ob n der Code eines Programms ist, d.h. ob $n = \mathfrak{h}P$ für ein $P \in \mathcal{P}$ gilt.



5. ALGORITHMISCH UNLÖSBARE AUFGABEN

5.1. Abzählbarkeit der \mathcal{P} -berechenbaren Funktionen

Mit Hilfe der Codes von \mathcal{P} -Programmen können wir eine wichtige Eigenschaft der \mathcal{P} -berechenbaren Funktionen beweisen, nämlich die *Abzählbarkeit* der Menge $\llbracket \mathcal{P} \rrbracket$ aller \mathcal{P} -berechenbaren Funktionen. Zum Nachweis definieren wir zunächst:

Definition 5.1.

Für jedes $i \in \mathbb{N}$ ist die Funktion $\varphi_i : \mathbb{N} \mapsto \mathbb{N}$ definiert durch

$$\varphi_i := \begin{cases} \llbracket P \rrbracket & , \text{ falls } i = \natural P \text{ für ein } P \in \mathcal{P} \\ \omega & , \text{ falls } i \notin \natural \mathcal{P} . \quad \blacksquare \end{cases}$$

Für $i \in \natural \mathcal{P}$ ist φ_i also diejenige Funktion, die von dem \mathcal{P} -Programm mit Kode i berechnet wird. Also gilt $\varphi_{\natural P} = \llbracket P \rrbracket$ für jedes Programm $P \in \mathcal{P}$. Falls $i \notin \natural \mathcal{P}$, so ordnen wir i die überall undefinierte Funktion zu, d.h. es gilt dann $\varphi_i = \omega$. Man beachte, daß φ_i *wohldefiniert* ist, denn mit der Injektivität von \natural (vgl. Satz 4.12) gibt es *höchstens* ein Programm P mit $\natural P = i$, und mit Satz 4.10 dürfen wir annehmen, daß $\llbracket P \rrbracket : \mathbb{N} \mapsto \mathbb{N}$ für jedes Programm $P \in \mathcal{P}$ gilt.

Nun können wir die Abzählbarkeit aller \mathcal{P} -berechenbaren Funktionen einfach beweisen:

Satz 5.2. $\llbracket \mathcal{P} \rrbracket = \{\varphi_i \mid i \in \mathbb{N}\}$, d.h. $\llbracket \mathcal{P} \rrbracket$ ist abzählbar.

Beweis. “ \subseteq ” Sei $\phi \in \llbracket \mathcal{P} \rrbracket$. Dann gilt $\phi = \llbracket P \rrbracket$ für ein $P \in \mathcal{P}$ und damit $\phi = \varphi_{\natural P}$. Also gilt $\phi \in \{\varphi_i \mid i \in \mathbb{N}\}$, denn $\natural P \in \mathbb{N}$.

“ \supseteq ” Falls $i \notin \natural \mathcal{P}$, so gilt $\varphi_i = \omega = \llbracket \text{CYCLE}_1 \rrbracket$ mit Satz 3.2, und mit $\text{CYCLE}_1 \in \mathcal{P}$ dann $\varphi_i \in \llbracket \mathcal{P} \rrbracket$. Andernfalls gilt $i = \natural P$ für ein Programm $P \in \mathcal{P}$ und damit $\varphi_{\natural P} = \llbracket P \rrbracket$, also ebenfalls $\varphi_i \in \llbracket \mathcal{P} \rrbracket$. ■

Wesentlich im Beweis von Satz 5.2 ist, daß den \mathcal{P} -berechenbaren Funktionen ein Programm zugeordnet werden kann und Programme in natürliche Zahlen *kodiert* werden können, woraus dann die Abzählbarkeit von \mathcal{P} folgt. Die folgenden Sätze zeigen, daß der Beweis von Satz 5.2 ohne diese Voraussetzungen nicht gelingt:

Satz 5.3. Sei N eine abzählbare Menge, d.h. $N = \{n_i \mid i \in \mathbb{N}\}$, und sei $M \subseteq N$. Dann ist M abzählbar.

Beweis. Für $M = \emptyset$ gilt die Aussage trivialerweise. Für $M \neq \emptyset$ sei $m' \in M$ fest gewählt. Wir definieren für jedes $i \in \mathbb{N}$

$$m_i := \begin{cases} n_i & , \text{ falls } n_i \in M \\ m' & , \text{ falls } n_i \notin M . \end{cases}$$

Mit $M \subseteq N$ gibt es für jedes $m \in M$ ein $i \in \mathbb{N}$ mit $m = n_i$, und damit gilt $m_i = m$, d.h. $M = \{m_i \mid i \in \mathbb{N}\}$. ■

Mit Satz 5.3 ist also jede Teilmenge einer abzählbaren Menge auch abzählbar. Umgekehrt bedeutet dies, daß jede Obermenge einer nicht-abzählbaren Menge auch nicht abzählbar ist. Wir geben jetzt eine nicht-abzählbare Menge von Funktionen an.

Satz 5.4. Sei \mathcal{F}_{tot} die Menge aller totalen Funktionen $\mathbb{N} \rightarrow \mathbb{N}$. Dann ist \mathcal{F}_{tot} nicht abzählbar.

Beweis. Wir nehmen an, daß \mathcal{F}_{tot} abzählbar ist, und leiten daraus mit einem sogenannten Diagonalisierungsbeweis einen Widerspruch her: Mit der Annahme gilt $\mathcal{F}_{tot} = \{\phi_i \mid i \in \mathbb{N}\}$ und $\gamma \in \mathcal{F}_{tot}$ für die Funktion $\gamma : \mathbb{N} \rightarrow \mathbb{N}$ mit $\gamma(n) := \phi_n(n) + 1$. Mit $\gamma \in \mathcal{F}_{tot}$ gilt $\gamma = \phi_k$ für ein $k \in \mathbb{N}$, und man erhält $\phi_k(k) = \gamma(k) = \phi_k(k) + 1$. Also muß $\phi_k(k) = \perp$ gelten, d.h. $\phi_k \notin \mathcal{F}_{tot}$. ▼ Damit kann \mathcal{F}_{tot} nicht abzählbar sein. ■

Mit Satz 5.4 können wir jetzt zeigen, daß die Menge *aller* arithmetischen Funktionen auch nicht abzählbar ist:

Satz 5.5. Sei \mathcal{F} die Menge aller Funktionen $\mathbb{N} \mapsto \mathbb{N}$. Dann ist \mathcal{F} nicht abzählbar.

Beweis. Wir führen einen Widerspruchsbeweis und nehmen an, daß \mathcal{F} abzählbar ist. Mit Satz 5.3 und $\mathcal{F}_{tot} \subseteq \mathcal{F}$ gilt dann, daß \mathcal{F}_{tot} abzählbar ist, also ein Widerspruch zu Satz 5.4. Damit kann \mathcal{F} nicht abzählbar sein. ■

Man beachte, daß Satz 5.5 nicht im Widerspruch zu Satz 5.2 steht, denn Satz 5.2 macht nur eine Aussage über *berechenbare* Funktionen, während Satz 5.5 eine Aussage über *alle* (arithmetischen) Funktionen macht. Entscheidend ist hier also die Forderung nach *Berechenbarkeit*.

5.2. Nicht-berechenbare Funktionen

Mit der Abzählbarkeit der \mathcal{P} -berechenbaren Funktionen können wir jetzt ein zentrales Ergebnis der Berechenbarkeitstheorie beweisen, nämlich daß es Aufgaben gibt, die prinzipiell *nicht algorithmisch lösbar* sind. D.h. es gibt Aufgaben, deren Lösung nicht durch einen Rechner berechnet werden kann. Formal heißt das, daß es *nicht-berechenbare Funktionen* gibt:

Korollar 5.6. *Es gibt eine Funktion $\phi : \mathbb{N} \mapsto \mathbb{N}$ mit $\phi \notin \llbracket \mathcal{P} \rrbracket$.*

Beweis. *Es gilt $\llbracket \mathcal{P} \rrbracket \subseteq \mathcal{F}$ für die Menge \mathcal{F} aller Funktionen $\phi : \mathbb{N} \mapsto \mathbb{N}$, mit Satz 5.2 ist $\llbracket \mathcal{P} \rrbracket$ abzählbar und mit Satz 5.5 ist \mathcal{F} nicht abzählbar. Damit gilt $\mathcal{F} \setminus \llbracket \mathcal{P} \rrbracket \neq \emptyset$. ■*

Der Beweis von Korollar 5.6 ist unkonstruktiv, vgl. Abschnitt 3.3, d.h. eine konkrete nicht-berechenbare Funktion wird in dem Beweis nicht angegeben. Dies holen wir jetzt nach:

Definition 5.7. (Selbstanwendungsfunktion)

Die Selbstanwendungsfunktion $self : \mathbb{N} \rightarrow \mathbb{N}$ ist definiert durch

$$self(n) := \begin{cases} 0 & , \text{ falls } \varphi_n(n) = \perp \\ 1 & , \text{ falls } \varphi_n(n) \neq \perp . \quad \blacksquare \end{cases}$$

Untersuchen wir zunächst, welche Ergebnisse die Funktion $self$ liefert: Für $n \notin \natural\mathcal{P}$ gilt $\varphi_n = \omega$, folglich $\varphi_n(n) = \perp$, und damit $self(n) = 0$. Andernfalls gilt $\varphi_n = \llbracket P \rrbracket$ für ein $P \in \mathcal{P}$, wobei $n = \natural P$. Um $self(n)$ zu bestimmen, führen wir das Programm P mit der Eingabe $\natural P$ aus, d.h. wir wenden P auf seinen eigenen Programmcode $\natural P$ an. Hält P mit dieser Eingabe, so gilt $\varphi_n(n) \neq \perp$ und damit $self(n) = 1$. Andernfalls erhalten wir $self(n) = 0$. Es gilt also

$$self(\natural P) = \begin{cases} 0 & , \text{ falls } \llbracket P \rrbracket(\natural P) = \perp \\ 1 & , \text{ falls } \llbracket P \rrbracket(\natural P) \neq \perp . \end{cases}$$

Wir zeigen jetzt, daß die Selbstanwendungsfunktion nicht berechenbar ist. Dazu führen wir einen Widerspruchsbeweis, d.h. wir nehmen an, $self$ sei berechenbar, und leiten daraus dann einen Widerspruch her.

Satz 5.8. Die Selbstanwendungsfunktion $self : \mathbb{N} \rightarrow \mathbb{N}$ ist nicht berechenbar.

Beweis. Angenommen, $self$ sei berechenbar. Dann gibt es eine \mathcal{P} -Prozedur $SELF$ mit $self = \llbracket SELF \rrbracket$. Mit $SELF$ konstruieren wir jetzt eine weitere \mathcal{P} -Prozedur $KURIOS$:

```

procedure KURIOS(n) <=
begin var x, res;
  if SELF(n) then x := CYCLE1(x) end_if;
  res := 1;
  return(res)
end .

```

Es gilt

$$\llbracket KURIOS \rrbracket(n) = \begin{cases} 1 & , \text{ falls } self(n) = 0 \\ \perp & , \text{ falls } self(n) \neq 0 \end{cases} ,$$

und mit der Definition von $self$ erhält man

$$\llbracket KURIOS \rrbracket(n) = \begin{cases} 1 & , \text{ falls } \varphi_n(n) = \perp \\ \perp & , \text{ falls } \varphi_n(n) \neq \perp . \end{cases}$$

Für $n = \natural KURIOS$ erhalten wir insbesondere

$$\llbracket KURIOS \rrbracket(\natural KURIOS) = \begin{cases} 1 & , \text{ falls } \llbracket KURIOS \rrbracket(\natural KURIOS) = \perp \\ \perp & , \text{ falls } \llbracket KURIOS \rrbracket(\natural KURIOS) \neq \perp . \end{cases}$$

Damit haben wir einen Widerspruch hergeleitet, denn es gilt $\llbracket KURIOS \rrbracket(\natural KURIOS) = \perp$ gdw. $\llbracket KURIOS \rrbracket(\natural KURIOS) \neq \perp$. Die Anwendung der Prozedur $KURIOS$ auf ihre eigene Kodierung $\natural KURIOS$ führt also zu einem Widerspruch, und damit kann $self$ keine berechenbare Funktion sein. ■

Der Nachweis, daß $self$ nicht berechenbar ist, erscheint zunächst wie ein mathematisches ‘‘Kunststück’’. Die Schritte der Herleitung sind einfach nachvollziehbar, weniger einleuchtend ist jedoch, wie man auf die gewählte Konstruktion eigentlich kommt, und *warum* sich daraus ein Widerspruch ergibt. Es ist daher zum Verständnis wichtig, daß man sich die Argumentationsweise und die dabei verwendeten Fakten noch einmal genau vor Augen führt:

1. Mit der Behauptung, daß $self$ berechenbar sei, dürfen wir die Existenz des Programms $SELF$ voraussetzen.

2. Unter Verwendung von **SELF** und dem Programm CYCLE_1 können wir ein weiteres Programm, nämlich **KURIOS**, angeben.
3. Mit der Definition von **KURIOS** erhalten wir eine berechenbare Funktion, nämlich $\llbracket \text{KURIOS} \rrbracket$.
4. $\llbracket \text{KURIOS} \rrbracket(n)$ ist insbesondere dann definiert, wenn n der Kode eines Programms P ist, also $n = \natural P$, und P aufgerufen mit $\natural P$ nicht terminiert.
5. Damit terminiert **KURIOS** insbesondere dann, wenn **KURIOS** mit dem Kode $\natural P$ eines Programms P aufgerufen wird, und P aufgerufen mit $\natural P$ *nicht* terminiert.
6. Folglich terminiert **KURIOS** aufgerufen mit $\natural \text{KURIOS}$ insbesondere dann, wenn **KURIOS** aufgerufen mit $\natural \text{KURIOS}$ *nicht* terminiert. ▼
7. Der Widerspruch entsteht also dadurch, daß wir die Existenz des Programms **SELF** annehmen, $\varphi_n(n) = \perp$ jedoch (für beliebige n) nicht durch einen Algorithmus festgestellt werden kann.

5.3. Das Halteproblem

Die nicht berechenbare Selbstanwendungsfunktion *self* mutet zunächst wenig nützlich an, und man fragt sich unwillkürlich, ob es auch "sinnvolle" Funktionen gibt, die nicht berechenbar sind.

Jeder Programmierer hat schon die leidvolle Erfahrung gemacht, daß sein Programm nicht terminiert. Üblicherweise wird ein Programm mit gewissen Eingaben getestet. Erhält man ein Resultat, so ist der Terminierungstest bestanden. Andernfalls gibt es zwei Möglichkeiten: Entweder terminiert das Programm mit der Eingabe nicht, oder aber es dauert nur etwas länger, bis ein Ergebnis berechnet ist. Es ist daher wünschenswert, ein Testwerkzeug **HALT** zu entwickeln, das als Eingabe den Kode $\natural P$ eines Programms P sowie eine Eingabe $m \in \mathbb{N}$ für dieses Programm erhält. Bei Ausführung von **HALT** soll nach endlicher Zeit 0 ausgegeben werden, wenn die Ausführung von P mit m nicht terminiert. Andernfalls soll **HALT** das Ergebnis 1 ausgeben.

Bevor wir **HALT** entwickeln, betrachten wir vorsichtshalber die von **HALT** berechnete Funktion. Dabei müssen wir feststellen, daß es kein Programm **HALT** mit den gewünschten Eigenschaften geben kann:

Satz 5.9. Die Haltefunktion $halt : \mathbb{N} \rightarrow \mathbb{N}$, gegeben durch

$$halt(\pi^2(i, n)) := \begin{cases} 0 & , \text{ falls } \varphi_i(n) = \perp \\ 1 & , \text{ falls } \varphi_i(n) \neq \perp , \end{cases}$$

ist nicht berechenbar.¹

Beweis. Wir führen einen Widerspruchsbeweis und nehmen an, daß es ein \mathcal{P} -Programm `HALT` mit $\llbracket \text{HALT} \rrbracket = halt$ gibt. Mit `HALT` können wir auch das \mathcal{P} -Programm

```

procedure PROG(x) <=
begin var y;
  y := HALT(PAIR2(x, x));
  return(y)
end

```

definieren. Es gilt $\llbracket \text{PROG} \rrbracket(n) = \llbracket \text{HALT} \rrbracket(\pi^2(n, n))$, und mit der Definition von *halt* erhält man

$$\llbracket \text{PROG} \rrbracket(n) = \begin{cases} 0 & , \text{ falls } \varphi_n(n) = \perp \\ 1 & , \text{ falls } \varphi_n(n) \neq \perp , \end{cases}$$

also $\llbracket \text{PROG} \rrbracket = self$, vgl. Definition 5.7. Mit Satz 5.8 ist $\llbracket \text{PROG} \rrbracket$ nicht berechenbar, und damit kann es kein \mathcal{P} -Programm `HALT` mit $\llbracket \text{HALT} \rrbracket = halt$ geben. ■

Mit Satz 5.9 ist es also prinzipiell unmöglich, das gewünschte Testwerkzeug `HALT` zu implementieren. Ursache ist **hier**, daß man für ein beliebiges Programm P und beliebige Eingaben m für dieses Programm nicht algorithmisch feststellen kann, ob P ausgeführt mit m **anhält oder nicht hält**.²

¹ Siehe Abschnitt 4.3 zur Bedeutung der Schreibweise $halt(\pi^2(i, n)) := \dots$.

² Entscheidend ist der Fall, in dem P mit Eingabe m *nicht* hält. Natürlich kann man algorithmisch feststellen, ob P mit Eingabe m hält – man läßt P einfach “laufen” und falls P hält, ist das ja feststellbar (etwa durch Anzeige eines Ergebnisses). Genau dieser Idee werden wir später – etwa im Beweis von Satz 7.11 – folgen.

6. UNIVERSELLE FUNKTION UND INTERPRETIERER

In Abschnitt 5.1 wurde mit Satz 5.2 gezeigt, daß die Menge aller \mathcal{P} -berechenbaren Funktionen abzählbar ist, d.h. $\llbracket \mathcal{P} \rrbracket = \{\varphi_i \mid i \in \mathbb{N}\}$. Damit können wir definieren:

Definition 6.1. (Universelle Funktion)

Die universelle Funktion $u_{\mathcal{P}} : \mathbb{N} \mapsto \mathbb{N}$ ist gegeben durch

$$u_{\mathcal{P}}(\pi^2(i, n)) := \varphi_i(n). \quad \blacksquare$$

Die Funktion $u_{\mathcal{P}}$ wird *universell* genannt, da mit $u_{\mathcal{P}}$ jede Funktion $\varphi_i \in \llbracket \mathcal{P} \rrbracket$ dargestellt werden kann. Mit $\varphi_{\natural P} = \llbracket P \rrbracket$, vgl. Definition 5.1, erhält man

$$u_{\mathcal{P}}(\pi^2(\natural P, n)) = \llbracket P \rrbracket(n) \quad (6.1)$$

und für die Prozedur

```
procedure P(x) <=
begin var y1, ..., yn; STA; return(y) end
```

mit Definition 2.8 dann

$$u_{\mathcal{P}}(\pi^2(\natural P, n)) = \text{value}(\text{eval}(M_{\mathcal{P}}^{(n)}, \text{STA}), \mathbf{y}). \quad (6.2)$$

Wegen (6.2) wird $u_{\mathcal{P}}$ auch der *Interpreter* der Programmiersprache \mathcal{P} genannt.

Wir überlegen jetzt, ob die Funktion $u_{\mathcal{P}}$ \mathcal{P} -berechenbar ist: Um $u_{\mathcal{P}}(\pi^2(i, n))$ zu berechnen, muß zuerst festgestellt werden, ob $i \in \natural \mathcal{P}$ gilt. Dies gelingt mit der \mathcal{P} -berechenbaren Funktion $\natural?$, vgl. Satz 4.12. Mit $i = \natural P$ für ein $P \in \mathcal{P}$ erhält man $u_{\mathcal{P}}(\pi^2(\natural P, n))$, indem man aus $\natural P$ das \mathcal{P} -Programm P bildet, also den Programmcode $\natural P$ mittels \natural^{-1} in das \mathcal{P} -Programm P *dekodiert*, und dann P auf n anwendet, vgl. (6.1). Andernfalls gilt $\varphi_i = \omega$ und damit $u_{\mathcal{P}}(\pi^2(i, n)) = \llbracket \text{CYCLE}_1 \rrbracket(n)$.

Damit ist $u_{\mathcal{P}}$ \mathcal{P} -berechenbar und folglich dürfen wir die Existenz eines $\mathcal{P}[1]$ -Programms **APPLY** mit

$$u_{\mathcal{P}}(m) = \llbracket \text{APPLY} \rrbracket(m) \quad (6.3)$$

annehmen, für das $\llbracket \text{APPLY} \rrbracket(\pi^2(\natural P, n)) = \llbracket P \rrbracket(n)$ gilt. Das \mathcal{P} -Programm **APPLY** überprüft bei Eingabe m mittels $\natural?$, ob $\pi_1^2(m)$ der Kode eines \mathcal{P} -Programms P ist. Im positiven Fall wird dann P mit $\pi_2^2(m)$ als Eingabe ausgeführt. Für $\pi_1^2(m) \notin \natural\mathcal{P}$ ruft **APPLY** die \mathcal{P} -Prozedur **CYCLE**₁ auf, d.h. der Aufruf **APPLY**(m) terminiert in diesem Fall nicht. Damit *implementiert* **APPLY** den Interpretierer der Programmiersprache \mathcal{P} . Mit dem Interpretierer **APPLY** können wir \mathcal{P} -Programme P , die als Daten $\natural P$ repräsentiert werden, *durch ein \mathcal{P} -Programm* ausführen.

Bemerkung 6.2. Den Nachweis, daß $u_{\mathcal{P}}$ \mathcal{P} -berechenbar ist, haben wir hier nur informell geführt. Eigentlich müßte diese Behauptung mit einem streng formalen Beweis nachgewiesen werden, und zwar indem **APPLY** konkret angegeben und damit dann (6.3) bewiesen wird. Wir verzichten darauf, da dies hier einen zu großen beweistechnischen Aufwand erfordert.

Bemerkung 6.3. Als **APPLY** kann man sich genausogut ein Programm vorstellen, das als Eingabe eine endliche Folge $a_1 \dots a_n$ von ASCII Zeichen erhält, feststellt, ob $a_1 \dots a_n$ ein \mathcal{P} -Programm P nach Definition 2.1 ist, im positiven Fall P ausführt, und im negativen Fall nicht terminiert. Dies ist natürlich unpraktisch, denn man erwartet sinnvollerweise bei Ausführungsversuch eines vermeintlichen \mathcal{P} -Programms mit syntaktischen Fehlern, wie etwa falsche Klammerung, nicht-deklarierte lokale Variable u.s.w., einen Hinweis des Rechners anstatt eine Endlosberechnung. Sinnvoller wäre also ein $\mathcal{P}[1]$ -Programm **APPLY'** mit

$$\llbracket \text{APPLY}' \rrbracket(m) = \begin{cases} 0 & , \text{ falls } \pi_1^2(m) \notin \natural\mathcal{P} \\ 1 + \llbracket P \rrbracket(\pi_2^2(m)) & , \text{ falls } \pi_1^2(m) = \natural P , \end{cases}$$

d.h. **APPLY'** hält bei Eingabe “fehlerhafter” \mathcal{P} -Programme immer mit Ergebnis 0 (und bei Eingabe “korrekter” \mathcal{P} -Programme P mit $\llbracket P \rrbracket(\pi_2^2(m)) + 1$, falls die Ausführung von P terminiert). Da solch “praktische” Erwägungen für unsere Untersuchungen jedoch belanglos sind, reicht uns die Definition von **APPLY** so wie in (6.3) angegeben.

7. ENTSCHIEDBARKEIT UND SEMI-ENTSCHEIDBARKEIT

7.1. Entscheidbare Probleme

Die Frage aus Abschnitt 5.3, ob ein beliebiges Programm P ausgeführt mit einer beliebigen Eingabe $m \in \mathbb{N}$ anhält oder nicht, wird das *Halteproblem* genannt. Allgemein formulieren wir Probleme, die durch einen Rechner gelöst werden sollen, wie folgt:

Sei $M \subseteq \mathbb{N}$ und sei $m \in \mathbb{N}$. Das gestellte Problem besteht dann darin festzustellen, ob “ $m \in M$ ” wahr oder falsch ist. Das Halteproblem H kann man jetzt durch

$$H := \{\pi^2(n, m) \in \mathbb{N} \mid \varphi_n(m) \neq \perp\}$$

formulieren. Für beliebige $n, m \in \mathbb{N}$ soll hier festgestellt werden, ob “ $\pi^2(n, m) \in H$ ” wahr oder falsch ist.

Die Aufgabe $\varphi_n(n) \neq \perp$ festzustellen, vgl. Abschnitt 5.2, nennt man auch das *Selbstanwendbarkeitsproblem* (oder das *spezielle Halteproblem*) S , das man durch

$$S := \{n \in \mathbb{N} \mid \varphi_n(n) \neq \perp\}$$

formulieren kann. Für beliebige $n \in \mathbb{N}$ soll dabei festgestellt werden, ob “ $n \in S$ ” wahr oder falsch ist.

Die Antwort auf die Frage “ $m \in M$?” wollen wir natürlich berechnen lassen, und allgemein untersuchen wir Probleme daraufhin, ob sie durch einen Rechner gelöst werden können oder nicht. Ein Problem, wie z.B. das Halteproblem, nennen wir im folgenden *entscheidbar*, wenn ein Rechner immer – also in endlicher Zeit – eine korrekte Antwort liefern kann. Wir definieren daher:

Definition 7.1. (Entscheidbare Menge, charakteristische Funktion)

Sei $M \subseteq \mathbb{N}$ und sei $\chi_M : \mathbb{N} \rightarrow \{0, 1\}$ mit

$$\chi_M(m) := \begin{cases} 0 & , \text{ falls } m \notin M \\ 1 & , \text{ falls } m \in M . \end{cases}$$

Dann ist M entscheidbar genau dann, wenn χ_M berechenbar ist. Die Funktion χ_M wird die charakteristische Funktion von M genannt. Ein \mathcal{P} -Programm PROG mit $\chi_M = \llbracket \text{PROG} \rrbracket$ wird ein Entscheidungsverfahren für M genannt.¹ ■

Das Halteproblem ist genau dann entscheidbar, wenn die charakteristische Funktion $\chi_H : \mathbb{N} \rightarrow \{0, 1\}$ berechenbar ist. Mit der Definition von H erhalten wir

$$\chi_H(\pi^2(n, m)) = \begin{cases} 0 & , \text{ falls } \varphi_n(m) = \perp \\ 1 & , \text{ falls } \varphi_n(m) \neq \perp . \end{cases}$$

Da $\chi_H = \text{halt}$ für die nicht-berechenbare Funktion halt aus Satz 5.9 gilt, ist H nicht entscheidbar. Man erhält also:

Korollar 7.2. *Das Halteproblem H ist nicht entscheidbar.*

Mit Satz 5.8 ist auch die Selbstanwendungsfunktion self nicht berechenbar, und damit gilt:

Korollar 7.3. *Das Selbstanwendbarkeitsproblem S ist nicht entscheidbar.*

Betrachten wir ein weiteres Problem: Ist entscheidbar, ob eine natürliche Zahl eine Primzahl ist? Um dies zu beantworten, betrachten wir die charakteristische Funktion $\chi_{\mathbb{P}} : \mathbb{N} \rightarrow \{0, 1\}$ für die Menge \mathbb{P} aller Primzahlen

$$\chi_{\mathbb{P}}(n) = \begin{cases} 0 & , \text{ falls } n \notin \mathbb{P} \\ 1 & , \text{ falls } n \in \mathbb{P} . \end{cases}$$

Da wir offensichtlich ein Programm $\text{PRIM} \in \mathcal{P}[1]$ mit $\chi_{\mathbb{P}} = \llbracket \text{PRIM} \rrbracket$ angeben können, ist die Menge aller Primzahlen entscheidbar.

Die Entscheidbarkeit von Problemen überträgt sich auf deren Vereinigung und Schnitt:

Satz 7.4. *Für $M_1, M_2 \subseteq \mathbb{N}$ sind $M_1 \cup M_2$ und $M_1 \cap M_2$ entscheidbar, falls M_1 und M_2 entscheidbar sind.*

¹ In der englischsprachigen Literatur wird neben “decidable” auch der Begriff “recursive” für “entscheidbar” verwendet. Es gibt also sowohl “recursive functions” als auch “recursive problems” (s. auch die Fußnote auf Seite 26).

Beweis. Seien DECIDE_{M_1} , DECIDE_{M_2} \mathcal{P} -Prozeduren mit $\llbracket \text{DECIDE}_{M_1} \rrbracket = \chi_{M_1}$ und $\llbracket \text{DECIDE}_{M_2} \rrbracket = \chi_{M_2}$, und seien die Prozeduren $\text{DECIDE}_{M_1 \cup M_2}$, $\text{DECIDE}_{M_1 \cap M_2} \in \mathcal{P}$ gegeben durch

```

procedure DECIDE_{M_1 \cup M_2}(x) <=
begin var res;
  if DECIDE_{M_1}(x) then res := 1 else res := DECIDE_{M_2}(x) end_if;
  return(res)
end

```

und

```

procedure DECIDE_{M_1 \cap M_2}(x) <=
begin var res;
  if DECIDE_{M_1}(x) then res := DECIDE_{M_2}(x) else res := 0 end_if;
  return(res)
end .

```

Dann gilt offensichtlich $\llbracket \text{DECIDE}_{M_1 \cup M_2} \rrbracket = \chi_{M_1 \cup M_2}$ sowie $\llbracket \text{DECIDE}_{M_1 \cap M_2} \rrbracket = \chi_{M_1 \cap M_2}$, und damit sind sowohl $M_1 \cup M_2$ als auch $M_1 \cap M_2$ entscheidbar. ■

7.2. Endliche Probleme

Eine Konsequenz von Definition 7.1 ist, daß *endliche* Probleme immer entscheidbar sind:

Satz 7.5. $M \subseteq \mathbb{N}$ mit $|M| < \infty$ ist entscheidbar.

Beweis. Für $M = \emptyset$ ist $\chi_M : \mathbb{N} \rightarrow \{0, 1\}$ mit $\chi_M(m) := 0$ für alle $m \in \mathbb{N}$ die charakteristische Funktion von M , und offenbar ist χ_M berechenbar.

Andernfalls gilt $M = \{m_0, \dots, m_n\}$ für gewisse $m_0, \dots, m_n \in \mathbb{N}$ und damit gilt $\llbracket \text{DECIDE}_M \rrbracket = \chi_M$ für die \mathcal{P} -Prozedur DECIDE_M aus Abbildung 7.1. Folglich ist M entscheidbar. ■

Der Beweis von Satz 7.5 ist *nicht konstruktiv*, vgl. Abschnitt 3.3, denn die Elemente m_0, \dots, m_n von M müssen ja nicht bekannt sein. Beispielsweise folgt aus Satz 7.5, daß das Problem

$$H_{<10^{10}} := \{\pi^2(n, m) \in \mathbb{N} \mid n, m < 10^{10} \text{ und } \varphi_n(m) \neq \perp\}$$

entscheidbar ist, denn $H_{<10^{10}} \subseteq H$ ist endlich. Wie folgende Überlegung zeigt, ist $H_{<10^{10}}$ entscheidbar, auch wenn wir die Elemente von $H_{<10^{10}}$ nicht kennen:

```

procedure DECIDE_M(x) <=
begin var res;
  case x of
    m0: res := 1
    m2: res := 1
    ...
    mn: res := 1
  other: res := 0
end_case;
return(res)
end .

```

Abbildung 7.1: Ein Entscheidungsverfahren für eine endliche Menge

Allgemein gibt es für jedes $k \in \mathbb{N}$ 2^{k^2} Teilmengen N von \mathbb{N} , so daß $\pi_1^2(j), \pi_2^2(j) < k$ für jedes $j \in N$ gilt, und es gilt $N = H_{<k}$ für genau eine dieser Teilmengen. Jede dieser Teilmengen N ist endlich, wobei wir alle Elemente von N *kennen*, und wir können daher wie im Beweis von Satz 7.5 für jedes N ein Programm `DECIDE_M` schreiben, das die charakteristische Funktion χ_N von N berechnet. Unter den endlich vielen, nämlich 2^{k^2} Programmen, befindet sich dann auch das Programm, das die charakteristische Funktion von $H_{<k}$ berechnet. Wir können also tatsächlich ein Programm angeben, mit dem $H_{<k}$ entschieden werden kann, wir wissen nur nicht, welches der 2^{k^2} Programme das Gewünschte leistet, vgl. Abschnitt 3.3.

7.3. Semi-entscheidbare Probleme

Da *Probleme* in unserem Zusammenhang **Mengen sind**, können wir auch das Komplement solcher Mengen betrachten. Für das Halteproblem erhalten wir beispielsweise das komplementäre Problem

$$\overline{H} = \{ \pi^2(n, m) \in \mathbb{N} \mid \varphi_n(m) = \perp \} .$$

Offensichtlich ist auch \overline{H} nicht entscheidbar, denn $\pi^2(n, m) \in \overline{H}$ gdw. $\pi^2(n, m) \notin H$. Da wir die charakteristische Funktion χ_H mit Hilfe der charakteristischen Funktion $\chi_{\overline{H}}$ bilden können, d.h.

$$\chi_H(\pi^2(n, m)) = \begin{cases} 0 & , \text{ falls } \chi_{\overline{H}}(\pi^2(n, m)) = 1 \\ 1 & , \text{ falls } \chi_{\overline{H}}(\pi^2(n, m)) = 0 , \end{cases}$$

folgt aus der Unentscheidbarkeit von H sofort, daß auch \overline{H} nicht entscheidbar ist. Allgemein gilt:

Satz 7.6. $M \subseteq \mathbb{N}$ ist genau dann entscheidbar, wenn das komplementäre Problem \overline{M} entscheidbar ist.

Beweis. “ \Rightarrow ” Wenn M entscheidbar ist, so gilt $\chi_M = \llbracket \text{DECIDE_M} \rrbracket$ für eine \mathcal{P} -Prozedur DECIDE_M . Für die \mathcal{P} -Prozedur $\text{DECIDE_}\overline{M}$ mit

```

procedure DECIDE_ $\overline{M}$ (x) <=
begin var res;
  if DECIDE_M(x) then res := 0 else res := 1 end_if;
  return(res)
end

```

gilt $\chi_{\overline{M}} = \llbracket \text{DECIDE_}\overline{M} \rrbracket$, und damit ist \overline{M} entscheidbar.

“ \Leftarrow ” Offensichtlich, denn $\overline{\overline{M}} = M$. ■

Mit Satz 7.5 und Satz 7.6 folgt sofort die Entscheidbarkeit kofiniter Mengen:

Korollar 7.7. $M \subseteq \mathbb{N}$ mit $|\overline{M}| < \infty$ ist entscheidbar.

Bezüglich **der** Entscheidbarkeit gibt es also keinen Unterschied zwischen einem Problem und seinem Komplementärproblem. Begnügen wir uns jedoch mit einer schwächeren Forderung, so können wir doch einen Unterschied feststellen.

Für das Halteproblem H erhalten wir immer eine positive Antwort, **wenn es überhaupt eine positive Antwort gibt**: Für $n, m \in \mathbb{N}$ gibt es ja (mindestens) ein Programm $\text{PROG}_n \in \mathcal{P}$ mit $\varphi_n = \llbracket \text{PROG}_n \rrbracket$. Wir können daher PROG_n mit Eingabe m ausführen. Falls $\pi^2(n, m) \in H$, so gilt $\varphi_n(m) \neq \perp$, d.h. PROG_n ausgeführt mit Eingabe m hält auf jeden Fall mit irgendeinem Ergebnis. Umgekehrt gilt $\pi^2(n, m) \in H$, falls PROG_n ausgeführt mit Eingabe m irgendein Ergebnis liefert.

Anders gesagt, für $n, m \in \mathbb{N}$ starten wir PROG_n mit m und erhalten immer eine korrekte Antwort, falls $\pi^2(n, m) \in H$ gilt. Gilt jedoch $\pi^2(n, m) \notin H$, so erhalten wir i.A. nie eine Antwort. Das ist zugegebenermaßen unbefriedigend, aber mehr kann man nicht erreichen, da H nicht entscheidbar ist.

Probleme, für die zumindest immer eine positive Antwort berechnet werden kann, nennt man *semi-entscheidbar*:

Definition 7.8. (Semi-entscheidbare Menge, semi-charakteristische Funktion)
Für eine Menge $M \subseteq \mathbb{N}$ definieren wir die Funktion $\tilde{\chi}_M : \mathbb{N} \mapsto \{1\}$ durch

$$\tilde{\chi}_M(m) := \begin{cases} \perp & , \text{ falls } m \notin M \\ 1 & , \text{ falls } m \in M . \end{cases}$$

Für eine Funktion $\psi : \mathbb{N} \mapsto \{0, 1\}$ schreiben wir $\psi \cong_M \tilde{\chi}_M$, falls $\psi(m) \neq 1$ für $m \notin M$ und $\psi(m) = 1$ für $m \in M$.

Eine Menge M ist semi-entscheidbar genau dann, wenn eine berechenbare Funktion ψ mit $\psi \cong_M \tilde{\chi}_M$ existiert. Jede solche Funktion ψ wird eine semi-charakteristische Funktion von M genannt. Ein \mathcal{P} -Programm **PROG** mit $\tilde{\chi}_M \cong_M \llbracket \text{PROG} \rrbracket$ wird ein semi-Entscheidungsverfahren für M genannt. ■

Die charakteristische Funktion χ_M eines Problems M erfüllt offenbar die Forderung, die wir an eine *semi*-charakteristische Funktion stellen, und folglich ist jedes entscheidbare Problem auch semi-entscheidbar:

Satz 7.9. Wenn M entscheidbar ist, so ist M auch semi-entscheidbar.

Beweis. Offensichtlich, denn $\chi_M \cong_M \tilde{\chi}_M$ und χ_M ist nach Voraussetzung berechenbar. ■

Im Unterschied zu charakteristischen Funktionen sind semi-charakteristische Funktionen nicht eindeutig bestimmt. Für eine Funktion ψ mit $\psi \cong_M \tilde{\chi}_M$ ist $\psi(m)$ im Fall $m \notin M$ *unterspezifiziert*, d.h. nicht eindeutig festgelegt. Allgemein gilt für $m \notin M$ immer $\psi(m) = 0$ oder $\psi(m) = \perp$ für eine semi-charakteristische Funktion ψ . Mit Satz 7.5 folgt sofort, daß $\psi(m) = \perp$ für unendlich viele $m \in \mathbb{N}$ gelten muß, falls M nicht entscheidbar ist.² Mit der Existenz *irgendeiner* berechenbaren semi-charakteristischen Funktion ist *die* semi-charakteristische Funktion $\tilde{\chi}_M$ jedoch immer berechenbar:

Satz 7.10. $\tilde{\chi}_M$ ist berechenbar, falls M semi-entscheidbar ist.

Beweis. Wenn M semi-entscheidbar ist, so existiert ein semi-Entscheidungsverfahren $\text{PROG} \in \mathcal{P}$, d.h. $\tilde{\chi}_M \cong_M \llbracket \text{PROG} \rrbracket$. Für die \mathcal{P} -Prozedur **SEMI_DECIDE_M** gegeben durch

```

procedure SEMI_DECIDE_M(x) <=
begin var res;
  if PROG(x) then res := 1 else res := CYCLE1(x) end_if;
  return(res)
end

```

² Damit ist natürlich nicht ausgeschlossen, daß bei unentscheidbarem M auch $\psi(m) = 0$ für unendlich viele $m \in \mathbb{N}$ gelten *kann*.

gilt dann

$$\llbracket \text{SEMI_DECIDE_M} \rrbracket(m) = \begin{cases} 1, & \text{falls } \llbracket \text{PROG} \rrbracket = 1 \\ \perp, & \text{falls } \llbracket \text{PROG} \rrbracket = 0 \\ \perp, & \text{falls } \llbracket \text{PROG} \rrbracket = \perp . \end{cases}$$

und damit

$$\llbracket \text{SEMI_DECIDE_M} \rrbracket(m) = \begin{cases} 1 & , \text{ falls } m \in M \\ \perp & , \text{ falls } m \notin M , \end{cases}$$

also $\llbracket \text{SEMI_DECIDE_M} \rrbracket = \tilde{\chi}_M$. ■

Wir können jetzt beweisen, daß das Halteproblem semi-entscheidbar ist:

Satz 7.11. *Das Halteproblem H ist semi-entscheidbar.*

Beweis. Sei $\text{SEMI_DECIDE_H} \in \mathcal{P}$ definiert durch

```

procedure SEMI_DECIDE_H(x) <=
begin var y,z;
  y := 1;
  z := APPLY(x);
  return(y)
end .

```

Dann gilt

$$\llbracket \text{SEMI_DECIDE_H} \rrbracket(\pi^2(n, m)) = \begin{cases} \perp & , \text{ falls } \llbracket \text{APPLY} \rrbracket(\pi^2(n, m)) = \perp \\ 1 & , \text{ falls } \llbracket \text{APPLY} \rrbracket(\pi^2(n, m)) \neq \perp \end{cases}$$

und mit $\llbracket \text{APPLY} \rrbracket(\pi^2(n, m)) = \varphi_n(m)$ dann

$$\llbracket \text{SEMI_DECIDE_H} \rrbracket(\pi^2(n, m)) = \begin{cases} \perp & , \text{ falls } \varphi_n(m) = \perp \\ 1 & , \text{ falls } \varphi_n(m) \neq \perp . \end{cases}$$

Also gilt $\tilde{\chi}_H = \llbracket \text{SEMI_DECIDE_H} \rrbracket$, und damit ist H semi-entscheidbar. ■

Korollar 7.12. *Das Selbstanwendungsproblem S ist semi-entscheidbar.*

Beweis. *Für die \mathcal{P} -Prozedur*

```

procedure SEMI_DECIDE_S(x) <=
begin var y;
  y := SEMI_DECIDE_H(PAIR2(x, x));
  return(y)
end

```

zeigt man leicht $\tilde{\chi}_S = \llbracket \text{SEMI_DECIDE_S} \rrbracket$. ■

Betrachten wir nun das Komplementärproblem \overline{H} : Hier gilt $\varphi_n(m) = \perp$, falls $\pi^2(n, m) \in \overline{H}$, d.h. APPLY ausgeführt mit Eingabe $\pi^2(n, m)$ hält *nie* mit einem Ergebnis. Offenbar können wir nicht analog der Konstruktion im Beweis von Satz 7.11 vorgehen, um ein Programm $\text{SEMI_DECIDE}_{\overline{H}} \in \mathcal{P}$ anzugeben, für das $\tilde{\chi}_{\overline{H}} \cong_{\overline{H}} \llbracket \text{SEMI_DECIDE}_{\overline{H}} \rrbracket$ gilt.

Damit ist natürlich noch nicht bewiesen, daß \overline{H} nicht semi-entscheidbar ist. Es könnte ja gelingen, ein Programm $\text{SEMI_DECIDE}_{\overline{H}}$ mit $\llbracket \text{SEMI_DECIDE}_{\overline{H}} \rrbracket \cong_{\overline{H}} \tilde{\chi}_{\overline{H}}$ auf andere Weise zu definieren. Nehmen wir einmal an, dies gelingt uns tatsächlich. Dann können wir das folgende Experiment durchführen:

Wir besorgen uns 2 Rechner und installieren auf dem einen Rechner das Programm SEMI_DECIDE_H, und auf dem anderen Rechner wird SEMI_DECIDE_{\overline{H}} installiert. Für beliebige $n, m \in \mathbb{N}$ starten wir dann SEMI_DECIDE_H mit Eingabe $\pi^2(n, m)$ auf Rechner 1 und SEMI_DECIDE_{\overline{H}} mit der gleichen Eingabe auf Rechner 2. Gilt $\pi^2(n, m) \in H$, so hält Rechner 1 nach endlicher Zeit mit Ergebnis 1. Gilt jedoch $\pi^2(n, m) \notin H$, so hält Rechner 2 nach endlicher Zeit mit Ergebnis 1. Wir bekommen so immer in endlicher Zeit eine korrekte Antwort für das Halteproblem und haben damit doch noch ein Entscheidungsverfahren für das Halteproblem gefunden. Da dies im Widerspruch zu Satz 7.2 steht, gibt es entweder kein Programm SEMI_DECIDE_H (als semi-Entscheidungsverfahren für H) oder kein Programm SEMI_DECIDE_{\overline{H}} (als semi-Entscheidungsverfahren für \overline{H}). Da wir jedoch mit Satz 7.11 die Existenz von SEMI_DECIDE_H nachgewiesen haben, kann es kein Programm SEMI_DECIDE_{\overline{H}} als semi-Entscheidungsverfahren für \overline{H} geben, und folglich ist \overline{H} nicht semi-entscheidbar.

Nun kann man an der Argumentationsweise kritisieren, daß wir bislang immer nur *einen* Rechner betrachtet haben, wenn wir die Berechenbarkeit einer Funktion bzw. die Entscheidbarkeit eines Problems untersucht haben. In Abschnitt 8.3 werden wir daher ohne Rückgriff auf mehrere Rechner beweisen, daß \overline{H} nicht semi-entscheidbar ist.

8. SCHRITTFUNKTIONEN

8.1. Die Schrittfunktion eines \mathcal{P} -Programms

Das Experiment vom Ende des letzten Abschnitts kann man auch mit einem einzigen Rechner durchführen. Wenn dieser Rechner ein Betriebssystem besitzt, mit dem mehrere Prozesse verwaltet werden können, so starten wir `SEMI_DECIDE_H` und `SEMI_DECIDE_H̄` jeweils als einen eigenen Prozeß. Das Betriebssystem konfigurieren wir so, daß jedem der beiden Prozesse immer abwechselnd eine feste Prozessorzeit zugeordnet wird. Mit dieser Vorgehensweise erhalten wir dann das gleiche Resultat wie zuvor.

Die Grundidee ist hier also, daß die Programme `SEMI_DECIDE_H̄` und `SEMI_DECIDE_H` Schritt für Schritt “verzahnt” abgearbeitet werden. Dies ist erforderlich, denn wir können ja nicht die Programme hintereinander ausführen, also etwa zuerst `SEMI_DECIDE_H̄` und dann `SEMI_DECIDE_H`. Gilt bei dieser Programmreihenfolge $\pi^2(n, m) \in H$, so hält `SEMI_DECIDE_H̄` nicht, und `SEMI_DECIDE_H` wird nie aufgerufen, um $\pi^2(n, m) \in H$ festzustellen.

Die Prozessorzeit, die wir im Beispiel jedem der beiden Prozesse abwechselnd zugeordnet haben, modellieren wir formal mit der *Anzahl der Schritte*, die bei Abarbeitung eines Programms erlaubt sind. Jedes Programm wird ja von einem Rechner “Schritt für Schritt” abgearbeitet. Auf der untersten Maschinenebene besteht ein “Schritt” aus dem *fetch&execute*-Zyklus des Prozessors, bei dem dasjenige Bitwort, auf das der Befehlszähler verweist, aus dem Speicher gelesen, ein Maschinenbefehl gemäß dem gelesenen Bitwort ausgeführt und abschließend der Befehlszähler verändert wird, bevor der nächste *fetch&execute*-Zyklus beginnt.

Man kann einen Abarbeitungsschritt auch auf der Ebene einer höheren Programmiersprache – wie etwa \mathcal{P} – definieren, indem man abstrakt die *Interpretationskosten* eines Programms definiert. Damit kann man dann die Abarbeitung eines Programms durch Angabe einer *Laufzeitbeschränkung*, d.h. durch Angabe einer maximal erlaubten Schrittzahl *kontrollieren*, und so eine *beschränkte Ausführung* von *Programmanweisungen* implementieren. Dabei ist ein realistisches Kostenmaß für unsere Zwecke unerheblich: Da lediglich die Terminierung von Programmausführungen sichergestellt werden muß und nicht-Terminierung nur

bei Ausführung von **while**-Schleifen auftreten kann, reicht es hier die Anzahl der Ausführungen von Schleifenrumpfen zu begrenzen.

Wir realisieren diese Idee durch Angabe einer *totalen* Funktion

$$step : \mathbb{N} \times 2^{\mathbb{N} \times \mathbb{N}} \times \text{statements}_P \rightarrow \mathbb{N}$$

mit der die Kosten bei Ausführung von Programmanweisungen bestimmt werden: Für $step(b, M_P, STA)$ erhält man 0 als Ergebnis, falls die initial vorgegebene Laufzeitbeschränkung b nicht ausreicht, um die Programmanweisung STA unter der Speicherbelegung M_P durch *eval* auszuwerten. Andernfalls ist die nach Abarbeitung von STA verbleibende Restschrittzahl das Ergebnis von $step(b, M_P, STA)$, und es gilt dann $b \geq step(b, M_P, STA) > 0$. Da in diesem Fall garantiert ist, daß die gegebene Laufzeitbeschränkung b ausreicht um die Programmanweisung STA durch *eval* auszuführen, ist diese Programmausführung unkritisch, denn eine nicht-terminierende Abarbeitung von STA ist dann ausgeschlossen.

Definition 8.1. (Schrittzählfunktion *step*)

Die Schrittzählfunktion

$$step : \mathbb{N} \times 2^{\mathbb{N} \times \mathbb{N}} \times \text{statements}_P \rightarrow \mathbb{N}$$

für Programmanweisungen in \mathcal{P} unter einer Laufzeitbeschränkung $b \in \mathbb{N}$ und einer Speicherbelegung $M_P \subseteq \mathbb{N} \times \mathbb{N}$ ist definiert durch:

- (s1) $step(0, M_P, STA) := 0$,
- (s2) $step(b + 1, M_P, SKIP) := b + 1$,
- (s3) $step(b + 1, M_P, x := \text{EXPR}) := b + 1$,
- (s4) $step(b + 1, M_P, \text{if EXPR then STA1 else STA2 end_if})$
 $:= \begin{cases} step(b + 1, M_P, STA1), & \text{falls } value(M_P, \text{EXPR}) > 0 \\ step(b + 1, M_P, STA2), & \text{falls } value(M_P, \text{EXPR}) = 0, \end{cases}$
- (s5) $step(b + 1, M_P, STA1; STA2) :=$
 $:= step(step(b + 1, M_P, STA1), eval(M_P, STA1), STA2),$
- (s6) $step(b + 1, M_P, \text{while EXPR do STA end_while})$
 $:= \begin{cases} b + 1, & \text{falls } value(M_P, \text{EXPR}) = 0, \\ step(step(b, M_P, STA), \\ \quad eval(M_P, STA), \quad , \text{ andernfalls} \\ \text{while EXPR do STA end_while}) . \end{cases}$ ■

Die beschränkte Ausführung von Programmanweisungen unterscheidet sich von der unbeschränkten Form also lediglich darin, daß die Verarbeitung durch (s1) abbricht, da dann die vorgegebene Schrittzahl erschöpft ist. Der folgende Satz illustriert den Zusammenhang zwischen beschränkter und unbeschränkter Abarbeitung von \mathcal{P} -Programmen:

Satz 8.2. Für alle $P \in \mathcal{P}$, $\text{STA} \in \text{statements}_P$, $M_P \in 2^{\mathbb{N} \times \mathbb{N}}$ und alle $b \in \mathbb{N}$ gilt:

1. $\text{eval}(M_P, \text{STA}) \neq \perp \Leftrightarrow \exists b_0 \in \mathbb{N} \forall b \geq b_0. \text{step}(b, M_P, \text{STA}) > 0$
2. $\text{step}(b, M_P, \text{STA}) > 0 \Leftrightarrow \text{eval}(M_P, \text{STA}) \neq \perp .$

Beweis. Durch strukturelle Induktion über STA . ■

Mit Satz 8.2(1) gibt es für jede unbeschränkte Ausführung einer Programmanweisung STA mit Ergebnis $M'_P \neq \perp$ eine Laufzeitbeschränkung b_0 , so daß $\text{step}(b, M_P, \text{STA})$ mit jeder Laufzeitbeschränkung $\geq b_0$ erfolgreich ist. Umgekehrt gilt mit (2), daß mit $\text{step}(b, M_P, \text{STA}) > 0$ die Abarbeitung von STA **gelingt**.

Mit Satz 8.2(1) folgt insbesondere, daß step *monoton* ist:

Korollar 8.3. (Monotonie von step)

Für alle $P \in \mathcal{P}$, $\text{STA} \in \text{statements}_P$, $M_P \in 2^{\mathbb{N} \times \mathbb{N}}$ und alle $b, b' \in \mathbb{N}$ mit $b \geq b'$ gilt

$$\text{step}(b, M_P, \text{STA}) \geq \text{step}(b', M_P, \text{STA}). \quad \blacksquare$$

Mit Hilfe der beschränkten Abarbeitung können wir jetzt für jedes Programm $P \in \mathcal{P}[k]$ eine Funktion definieren, die für beliebige Argumente n_1, \dots, n_k und eine Laufzeitbeschränkung b angibt, ob die Berechnung von $\llbracket P \rrbracket(n_1, \dots, n_k)$ mit weniger als b Schritten gelingt:

Definition 8.4. (Schrittfunktion eines \mathcal{P} -Programms)

Für ein \mathcal{P} -Programm

```

procedure P( $\mathbf{x}_1, \dots, \mathbf{x}_k$ ) <=
begin var  $y_1, \dots, y_n$ ;  $\text{STA}$ ; return( $y$ ) end

```

ist die durch P berechnete Schrittfunktion $\llbracket P^{STEP} \rrbracket : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ definiert durch

$$\llbracket P^{STEP} \rrbracket(b, n_1, \dots, n_k) := \text{step}(b, M_P^{(n_1, \dots, n_k)}, \text{STA}) . \quad \blacksquare$$

Satz 8.5. Für alle $n_1, \dots, n_k \in \mathbb{N}$ gilt:

1. $(\forall b \in \mathbb{N}. \llbracket P^{STEP} \rrbracket(b, n_1, \dots, n_k) = 0) \curvearrowright \llbracket P \rrbracket(n_1, \dots, n_k) = \perp$, und
2. $\llbracket P \rrbracket(n_1, \dots, n_k) = \perp \curvearrowright (\forall b \in \mathbb{N}. \llbracket P^{STEP} \rrbracket(b, n_1, \dots, n_k) = 0)$.

Beweis. Mit Satz 8.2. ■

Mit Satz 8.5(1) scheitert die Berechnung von $\llbracket P \rrbracket(n_1, \dots, n_k)$, falls die Schrittfunktion für jede Laufzeitbeschränkung **0 als Ergebnis** liefert. Umgekehrt gilt mit (2), daß die Schrittfunktion für jede Laufzeitbeschränkung 0 berechnet, falls die Berechnung von $\llbracket P \rrbracket(n_1, \dots, n_k)$ erfolglos ist. Als Kontraposition von Satz 8.5 erhält man

- 1'. $\llbracket P \rrbracket(n_1, \dots, n_k) \neq \perp \curvearrowright \exists b \in \mathbb{N}. \llbracket P^{STEP} \rrbracket(b, n_1, \dots, n_k) > 0$, und
- 2'. $(\exists b \in \mathbb{N}. \llbracket P^{STEP} \rrbracket(b, n_1, \dots, n_k) > 0) \curvearrowright \llbracket P \rrbracket(n_1, \dots, n_k) \neq \perp$.

Mit (1') findet man für jede erfolgreiche Berechnung von $\llbracket P \rrbracket(n_1, \dots, n_k)$ eine Laufzeitbeschränkung b , so daß die Schrittfunktion ein Ergebnis > 0 liefert. Umgekehrt gilt mit (2'), daß die Berechnung von $\llbracket P \rrbracket(n_1, \dots, n_k)$ erfolgreich ist, falls die Schrittfunktion für irgendeine Laufzeitbeschränkung ein Ergebnis > 0 berechnet.

In Abschnitt 11.7 werden wir zeigen, daß $\llbracket P^{STEP} \rrbracket$ allein durch ein **loop-Programm** – das ist ein \mathcal{P} -Programm in dem **while**-Anweisungen nur in der Art von **for**-Schleifen verwendet werden – implementiert werden kann.

8.2. Die universelle Schrittfunktion

Genauso, wie wir für jedes \mathcal{P} -Programm P die Funktion $\llbracket P \rrbracket$ mittels der universellen Funktion $\mathbf{u}_{\mathcal{P}}$ berechnen können, vgl. Kapitel 6, können wir die Schrittfunktion $\llbracket P^{STEP} \rrbracket$ von P universell berechnen. Wir definieren daher:

Definition 8.6. (Universelle Schrittfunktion)

Die universelle Schrittfunktion $\mathbf{s}_{\mathcal{P}} : \mathbb{N} \rightarrow \mathbb{N}$ ist gegeben durch

$$\mathbf{s}_{\mathcal{P}}(\pi^3(i, b, n)) = \begin{cases} 0, & \text{falls } i \notin \mathfrak{b}\mathcal{P} \\ \llbracket P^{STEP} \rrbracket(b, n), & \text{falls } i = \mathfrak{b}P. \quad \blacksquare \end{cases}$$

Mit der Funktion $s_{\mathcal{P}}$ kann die Schrittfunktion eines jeden Programms P dargestellt werden. Man erhält

$$s_{\mathcal{P}}(\pi^3(\natural P, b, n)) = \llbracket P^{STEP} \rrbracket(b, n) \quad (8.1)$$

und für die Prozedur

```
procedure P(x) <=
begin var y1, . . . , yn; STA; return(y) end
```

mit Definition 8.4 dann

$$s_{\mathcal{P}}(\pi^3(\natural P, b, n)) = step(b, M_P^{(n)}, STA) . \quad (8.2)$$

Um $s_{\mathcal{P}}(\pi^3(i, b, n))$ zu berechnen, muß zuerst festgestellt werden, ob $i \in \natural \mathcal{P}$ gilt.¹ Dies gelingt mit der \mathcal{P} -berechenbaren Funktion $\natural?$ vgl. Satz 4.12. Für $i \notin \natural \mathcal{P}$ ist 0 das Ergebnis der Berechnung. Andernfalls gilt $i = \natural P$ für ein $P \in \mathcal{P}$ und man erhält $s_{\mathcal{P}}(\pi^3(i, b, n))$, indem aus $\natural P$ das \mathcal{P} -Programm P gebildet, also der Programmcode $\natural P$ mittels \natural^{-1} in das \mathcal{P} -Programm P *dekodiert*, und dann P auf n angewendet wird. Bei der Abarbeitung von P wird jedoch (im Unterschied zur Interpretation mit der universellen Funktion $u_{\mathcal{P}}$) die Laufzeitbeschränkung b berücksichtigt. Ist die Berechnung beendet – was aufgrund der Laufzeitbeschränkung immer eintritt – wird abschließend überprüft, ob der Abbruch der Berechnung aufgrund der Laufzeitbeschränkung erfolgte. In diesem Fall ist das Ergebnis 0, und andernfalls $b' > 0$, vgl. (8.2).

Damit ist $s_{\mathcal{P}}$ \mathcal{P} -berechenbar und folglich dürfen wir die Existenz eines $\mathcal{P}[1]$ -Programms **STEP** mit

$$s_{\mathcal{P}}(m) = \llbracket \mathbf{STEP} \rrbracket(m) \quad (8.3)$$

annehmen, für das $\llbracket \mathbf{STEP} \rrbracket(\pi^3(\natural P, b, n)) = \llbracket P^{STEP} \rrbracket(b, n)$ gilt. Das \mathcal{P} -Programm **STEP** überprüft bei Eingabe m mittels $\natural?$, ob $\pi_1^3(m)$ der Kode eines \mathcal{P} -Programms P ist. Im positiven Fall wird dann P mit $\pi_2^3(m)$ als Eingabe unter der Laufzeitbeschränkung $\pi_3^3(m)$ ausgeführt. Für $\pi_1^3(m) \notin \natural \mathcal{P}$ hält **STEP** sofort **mit 0**.

¹ Den Nachweis, daß $s_{\mathcal{P}}$ \mathcal{P} -berechenbar ist, führen wir – wie schon zuvor bei der universellen Funktion u – nur *informell*. Wir verzichten auf einen streng *formalen* Beweis, da dies hier einen zu großen beweistechnischen Aufwand erfordert.

```

procedure DECIDE_M(x) <=
begin var b, continue, res;
  b := 1; continue := 1;
  while continue do
    if STEP(PAIR3(⊥SEMI_DECIDE_M, b, x))
    then res := SEMI_DECIDE_M(x);
      continue := 0
    else if STEP(PAIR3(⊥SEMI_DECIDE_M̄, b, x))
    then res := ¬SEMI_DECIDE_M̄(x);
      continue := 0
    else b := SUCC(b)
    end_if
  end_if
end_while;
return(res)
end

```

Abbildung 8.1: Zum Beweis von Satz 8.7 – Ein Entscheidungsverfahren für M

8.3. Schrittweise Abarbeitung von \mathcal{P} -Programmen

Mit der Prozedur STEP können wir die “verzahnte” Abarbeitung mehrerer \mathcal{P} -Programme P_1, \dots, P_j , die in Form von Daten $\perp P_1, \dots, \perp P_j$ vorliegen, *kontrollieren*, indem wir diese Programme *schrittweise* abarbeiten. Damit steht uns jetzt eine Beweistechnik zur Verfügung, mit der wir – ohne Rückgriff auf mehrere Rechner oder die Vorstellung eines Betriebssystems mit *time sharing* – nachweisen können, daß das Komplement \overline{H} des Halteproblems H nicht semi-entscheidbar ist.

Um dies zu beweisen zeigen wir zunächst, daß jedes semi-entscheidbare Problem genau dann entscheidbar ist, wenn dessen Komplementärproblem ebenfalls semi-entscheidbar ist:

Satz 8.7. M ist genau dann entscheidbar, wenn M und \overline{M} semi-entscheidbar sind.

Beweis. “ \Rightarrow ” Folgt mit Satz 7.6 und Satz 7.9.

“ \Leftarrow ” Seien SEMI_DECIDE_M und SEMI_DECIDE_M̄ semi-Entscheidungsverfahren für M und \overline{M} . Für die Prozedur DECIDE_M $\in \mathcal{P}$ aus Abbildung 8.1 gilt dann

$$\llbracket \text{DECIDE_M} \rrbracket(m) = \begin{cases} \llbracket \text{SEMI_DECIDE_M} \rrbracket(m) & , \text{ falls } \llbracket \text{SEMI_DECIDE_M} \rrbracket(m) \neq \perp \\ \neg \llbracket \text{SEMI_DECIDE_M̄} \rrbracket(m) & , \text{ falls } \llbracket \text{SEMI_DECIDE_M̄} \rrbracket(m) \neq \perp . \end{cases}$$



Für $m \in M$ gilt $\llbracket \text{SEMI_DECIDE_M} \rrbracket(m) = 1$ und damit $\llbracket \text{DECIDE_M} \rrbracket(m) = 1$. Für $m \notin M$ gilt $\llbracket \text{SEMI_DECIDE_M} \rrbracket(m) = 1$ und damit $\llbracket \text{DECIDE_M} \rrbracket(m) = 0$. Also gilt

$$\llbracket \text{DECIDE_M} \rrbracket(m) = \begin{cases} 1 & , \text{ falls } m \in M \\ 0 & , \text{ falls } m \notin M , \end{cases}$$

d.h. DECIDE_M ist ein Entscheidungsverfahren für M .² ■

Entscheidend im Beweis von Satz 8.7 ist die Verwendung der Prozedur STEP .³ Da weder für SEMI_DECIDE_M noch für SEMI_DECIDE_M garantiert ist, daß diese Prozeduren halten, müssen wir die Ausführung dieser Prozeduren geeignet kontrollieren: Gilt $\text{STEP}(\text{PAIR}^3(\text{SEMI_DECIDE_M}, b, m)) = 1$ so hält $\text{SEMI_DECIDE_M}(m)$ mit Satz 8.5(2), und wir können das Ergebnis für $\text{DECIDE_M}(m)$ aus dem Ergebnis von $\text{SEMI_DECIDE_M}(m)$ bilden. Andernfalls testen wir, ob $\text{STEP}(\text{PAIR}^3(\text{SEMI_DECIDE_M}, b, m)) = 1$ gilt. Im positiven Fall hält dann $\text{SEMI_DECIDE_M}(m)$, und wir bilden das Ergebnis für $\text{DECIDE_M}(m)$ aus dem Ergebnis von $\text{SEMI_DECIDE_M}(m)$. Gilt jedoch weder $\text{STEP}(\text{PAIR}^3(\text{SEMI_DECIDE_M}, b, m)) = 1$ noch $\text{STEP}(\text{PAIR}^3(\text{SEMI_DECIDE_M}, b, m)) = 1$, so erhöhen wir die erlaubte Laufzeitbeschränkung b , um dann anschließend die Tests mit dem erhöhten Schrittzähler durchzuführen.

Da entweder $m \in M$ oder $m \in \overline{M}$ gilt muß einer der beiden Prozeduraufrufe $\text{SEMI_DECIDE_M}(m)$ oder $\text{SEMI_DECIDE_M}(m)$ auf jeden Fall halten. Dafür werden b_m Schritte benötigt, wobei b_m nicht von vornherein bekannt ist. Da die Laufzeitbeschränkung b in der Schleife nach b_m Schleifendurchläufen den Wert $b_m + 1$ erhält, verläuft einer der beiden Tests $\text{STEP}(\text{PAIR}^3(\text{SEMI_DECIDE_M}, b, m)) = 1$ und $\text{STEP}(\text{PAIR}^3(\text{SEMI_DECIDE_M}, b, m)) = 1$ im $b_m + 1$ -ten Schleifendurchlauf positiv. Damit wird die Schleife beendet, und $\text{DECIDE_M}(m)$ hält mit korrektem Ergebnis.



Mit Satz 8.7 können wir jetzt beweisen, daß \overline{H} tatsächlich nicht semi-entscheidbar ist:

Korollar 8.8. \overline{H} ist nicht semi-entscheidbar.

Beweis. Angenommen, \overline{H} sei semi-entscheidbar. Da H semi-entscheidbar ist, vgl. Satz 7.11, folgt mit Satz 8.7, daß H entscheidbar ist, also ein Widerspruch zu Satz 7.2. Folglich ist \overline{H} nicht semi-entscheidbar. ■

² Die Initialisierung “ $b := 1$ ” der Laufzeitbeschränkung b und deren Erhöhung “ $b := \text{SUCC}(b)$ ” in der Prozedur DECIDE_M aus Abbildung 8.1 sind willkürlich. Man kann genausogut “ $b := n_1$ ” und “ $b := b + n_2$ ” verwenden, sofern $n_1, n_2 > 0$ gilt.

³ In der englischsprachigen Literatur wird die Beweistechnik der “verzahnten” Abarbeitung von Programmen auch als “dovetailing” bezeichnet.

```

procedure PSI(x) <=
begin var b, continue, res, y, z;
  b := 1; continue := 1;
  y := PAIR12(x); z := PAIR22(x)
  while continue do
    if STEP(PAIR3(⊥, SIGMA, b, z))
      then res := SIGMA(z);
           continue := 0
    else if STEP(PAIR3(y, b, y))
      then res := THETA(z);
           continue := 0
    else b := SUCC(b)
    end_if
  end_if
end_while;
return(res)
end .

```

Abbildung 8.2: Die Prozedur PSI für Beispiel 8.10

Genauso zeigt man:

Korollar 8.9. \overline{S} ist nicht semi-entscheidbar.

Wir illustrieren die Verwendung von STEP mit einem weiteren Beispiel:

Beispiel 8.10. Seien $\theta, \sigma : \mathbb{N} \mapsto \mathbb{N}$ beliebige berechenbare Funktionen und sei $\psi : \mathbb{N} \mapsto \mathbb{N}$ definiert durch

$$\psi(\pi^2(y, z)) = \begin{cases} \theta(z) & , \text{ falls } \varphi_y(y) \neq \perp \\ \sigma(z) & , \text{ falls } \varphi_y(y) = \perp . \end{cases}$$

Dann ist ψ i.A. nicht berechenbar, denn für $\theta(z) = 1$ und $\sigma(z) = 0$ (für alle $z \in \mathbb{N}$) würde beispielsweise $\chi_S(y) = \psi(\pi^2(y, z))$ für alle $y, z \in \mathbb{N}$ gelten, und mit der Berechenbarkeit von ψ wäre das Selbstanwendbarkeitsproblem S dann im Widerspruch zu Satz 7.3 entscheidbar. Das Problem besteht offensichtlich darin, daß mit der Unentscheidbarkeit von $\varphi_y(y) = \perp$ bei Berechnung von $\psi(\pi^2(y, z))$ nicht zwischen $\theta(z)$ und $\sigma(z)$ ausgewählt werden kann.

Angenommen, θ ist eine Erweiterung von σ , d.h. es gilt $\sigma(z) = \perp$ oder $\sigma(z) = \theta(z)$ für alle $z \in \mathbb{N}$. Obwohl $\varphi_y(y) = \perp$ nicht entscheidbar ist, können wir jetzt

$\psi(\pi^2(y, z))$ doch berechnen: Seien SIGMA, THETA, PSI $\in \mathcal{P}$ -Prozeduren mit $\llbracket \text{SIGMA} \rrbracket = \sigma$, $\llbracket \text{THETA} \rrbracket = \theta$ und PSI definiert wie in Abbildung 8.2. Wir zeigen, daß $\llbracket \text{PSI} \rrbracket = \psi$ gilt, und damit, daß ψ berechenbar ist:

Angenommen, es gilt $\varphi_y(y) \neq \perp$: Dann gilt $\text{STEP}(\text{PAIR}^3(\mathbf{y}, \mathbf{k}, \mathbf{y})) = 1$ für ein $k \in \mathbb{N}$. Nehmen wir weiter an, daß $\text{STEP}(\text{PAIR}^3(\natural\text{SIGMA}, \mathbf{h}, \mathbf{z})) = 1$ für ein $h \leq k$ gilt. Man erhält $\llbracket \text{PSI} \rrbracket(\pi^2(y, z)) = \sigma(z)$ und $\sigma(z) = \theta(z)$, also $\llbracket \text{PSI} \rrbracket(\pi^2(y, z)) = \theta(z)$, denn mit $\text{STEP}(\text{PAIR}^3(\natural\text{SIGMA}, \mathbf{h}, \mathbf{z})) = 1$ gilt $\sigma(z) \neq \perp$. Also gilt $\llbracket \text{PSI} \rrbracket(\pi^2(y, z)) = \psi(\pi^2(y, z))$. Andernfalls gilt $\text{STEP}(\text{PAIR}^3(\natural\text{SIGMA}, \mathbf{h}, \mathbf{z})) = 0$ für alle $h \leq k$, und man erhält $\llbracket \text{PSI} \rrbracket(\pi^2(y, z)) = \theta(z) = \psi(\pi^2(y, z))$.

Nehmen wir jetzt an, daß $\varphi_y(y) = \perp$ gilt. Dann gilt auch $\text{STEP}(\text{PAIR}^3(\mathbf{y}, \mathbf{k}, \mathbf{y})) = 0$ für alle $k \in \mathbb{N}$. Nehmen wir weiter an, daß $\text{STEP}(\text{PAIR}^3(\natural\text{SIGMA}, \mathbf{h}, \mathbf{z})) = 1$ für ein $h \in \mathbb{N}$ gilt. Man erhält $\llbracket \text{PSI} \rrbracket(\pi^2(y, z)) = \sigma(z) = \psi(\pi^2(y, z))$. Andernfalls gilt $\text{STEP}(\text{PAIR}^3(\natural\text{SIGMA}, \mathbf{h}, \mathbf{z})) = 0$ für alle $h \in \mathbb{N}$, und man erhält $\llbracket \text{PSI} \rrbracket(\pi^2(y, z)) = \perp = \sigma(z) = \psi(\pi^2(y, z))$, denn mit $\text{STEP}(\text{PAIR}^3(\natural\text{SIGMA}, \mathbf{h}, \mathbf{z})) = 0$ (für alle $h \in \mathbb{N}$) gilt $\sigma(z) = \perp$. ■

9. REKURSIVE AUFZÄHLBARKEIT

9.1. Aufzählungsverfahren

Ein semi-Entscheidungsverfahren $\text{SEMI_DECIDE_M} \in \mathcal{P}$ implementiert ein *akzeptierendes* Erkennungsprinzip für eine Menge $M \subseteq \mathbb{N}$: Für $m \in \mathbb{N}$ erhalten wir *immer* eine positive Antwort, d.h. 1, falls $m \in M$, und (vielleicht) *manchmal* auch eine negative Antwort, also 0, falls $m \notin M$. Diesem Erkennungsprinzip stellen wir ein *generatives* Prinzip gegenüber: Wenn man die Elemente einer Menge M *systematisch erzeugen* kann, so kann man aus solch einem Erzeugungsverfahren in einfacher Weise ein semi-Entscheidungsverfahren bilden. Den Begriff “*systematisch erzeugen*” erfassen wir formal mit dem Begriff der *rekursiven Aufzählbarkeit*:

Definition 9.1. (Rekursive Aufzählbarkeit)

Eine Menge $M \subseteq \mathbb{N}$ ist genau dann rekursiv aufzählbar, **wenn** $M = \emptyset$ oder andernfalls **eine Funktion** $\alpha_M : \mathbb{N} \rightarrow M$ existiert, so daß gilt:

1. $\alpha_M(\mathbb{N}) = M$, d.h. α_M ist surjektiv, und
2. α_M ist berechenbar.

Eine Funktion $\alpha_M : \mathbb{N} \rightarrow \mathbb{N}$, für die (1) und (2) gilt, wird eine Aufzählungsfunktion von M genannt. Ein \mathcal{P} -Programm PROG mit $\alpha_M = \llbracket \text{PROG} \rrbracket$ heißt ein Aufzählungsverfahren für M . ■

Jede rekursiv aufzählbare Menge ist auch semi-entscheidbar: Für ein $m \in \mathbb{N}$ muß man nur alle Elemente von M Schritt für Schritt erzeugen, also M *aufzählen*, und nach jedem Aufzählungsschritt überprüfen, ob das zuletzt erzeugte Element identisch m ist. Verläuft der Test positiv, so hält man mit positiver Antwort (also 1) an, denn dann gilt ja $m \in M$. Für $m \in M$ muß das Verfahren in jedem Fall halten, denn mit der Surjektivität der Aufzählungsfunktion wird m nach endlich vielen Schritten erzeugt.

Satz 9.2. Wenn $M \subseteq \mathbb{N}$ rekursiv aufzählbar ist, so ist M semi-entscheidbar.

Beweis. Wenn $M = \emptyset$, so gilt $\tilde{\chi}_M = \llbracket \text{CYCLE}_1 \rrbracket$, d.h. $\text{CYCLE}_1 \in \mathcal{P}$ ist ein semi-Entscheidungsverfahren für M .¹ Andernfalls gibt es ein Aufzählungsverfahren $\text{ENUMERATE}_M \in \mathcal{P}$ für M . Für das Programm $\text{SEMI_DECIDE}_M \in \mathcal{P}$ mit

```

procedure SEMI_DECIDE_M(x) <=
begin var i, res;
  while ENUMERATE_M(i) ≠ x do i := SUCC(i) end_while;
  res := 1;
  return(res)
end

```

gilt

$$\llbracket \text{SEMI_DECIDE}_M \rrbracket(m) = \begin{cases} \perp & , \text{ falls } \llbracket \text{ENUMERATE}_M \rrbracket(i) \neq m \text{ für alle } i \in \mathbb{N} \\ 1 & , \text{ falls } \llbracket \text{ENUMERATE}_M \rrbracket(i) = m \text{ für ein } i \in \mathbb{N} . \end{cases}$$

Da ENUMERATE_M ein Aufzählungsverfahren für M ist, gilt $\llbracket \text{ENUMERATE}_M \rrbracket(\mathbb{N}) = M$, und damit gibt es für jedes $m \in M$ ein (kleinstes) $i \in \mathbb{N}$ mit $\text{ENUMERATE}_M(i) = m$. Für $m \in M$ bricht die Schleife in SEMI_DECIDE_M also nach i Schleifendurchläufen ab, und man erhält das Ergebnis 1. Gilt dagegen $m \notin M$, so bricht die Schleife nie ab und $\text{SEMI_DECIDE}_M(m)$ hält nicht. Damit gilt

$$\llbracket \text{SEMI_DECIDE}_M \rrbracket(m) = \begin{cases} \perp & , \text{ falls } m \notin M \\ 1 & , \text{ falls } m \in M , \end{cases}$$

und folglich $\tilde{\chi}_M = \llbracket \text{SEMI_DECIDE}_M \rrbracket$. Damit ist M semi-entscheidbar. ■

Rekursive Aufzählbarkeit ist nicht nur *hinreichend* für semi-Entscheidbarkeit, sondern auch *notwendig*, d.h. aus der semi-Entscheidbarkeit einer Menge M folgt auch deren rekursive Aufzählbarkeit. Zum Beweis konstruieren wir aus einem semi-Entscheidungsverfahren SEMI_DECIDE_M für M eine Prozedur ENUMERATE_M für M , die eine Aufzählungsfunktion α_M für M implementiert. Zur Konstruktion benötigen wir die Paar-Funktion sowie die universelle Schrittfunktion:

Mittels der Paar-Funktion interpretieren wir ein $i \in \mathbb{N}$ als ein Paar $(b, m) \in \mathbb{N} \times \mathbb{N}$, d.h. $i = \pi^2(b, m)$, und überprüfen mittels SEMI_DECIDE_M unter der Laufzeitbeschränkung b , ob $m \in M$ gilt. Im positiven Fall definieren wir $\alpha_M(i) := m$. Andernfalls wurde entweder $m \notin M$ durch SEMI_DECIDE_M festgestellt oder aber die

¹ Anstatt CYCLE_1 kann man natürlich auch eine \mathcal{P} -Prozedur wie `procedure ZERO(x) <= begin var y; y := 0; return(y) end` verwenden, für die dann $\tilde{\chi}_M \cong_M \llbracket \text{ZERO} \rrbracket$ gilt.

```

procedure ENUMERATE_M(i) <=
begin var b, element, res;
  b := PAIR12(i);
  element := PAIR22(i);
  res := m0;
  if STEP(PAIR3(⊥SEMI_DECIDE_M,b,element))
    then if SEMI_DECIDE_M(element) then res := element end_if
  end_if;
  return(res)
end

```

Abbildung 9.1: Die Prozedur ENUMERATE_M zum Beweis von Satz 9.3

Ausführung von SEMI_DECIDE_M mit Eingabe m scheiterte an der Laufzeitbeschränkung b . In diesen Fällen definieren wir $\alpha_M(i) := m_0$, wobei $m_0 \in M$ beliebig aber fest gewählt ist. Damit gilt $\alpha_M(i) \in M$ in jedem Fall, d.h. das implementierte Verfahren ENUMERATE_M ist *korrekt*.

Da für jedes $m \in M$ ein $b_m \in \mathbb{N}$ existiert, so daß SEMI_DECIDE_M in b_m Schritten “ $m \in M$ ” feststellen kann, gilt $m = \alpha_M(\pi^2(b_m + 1, m))$. Damit ist α_M surjektiv, und folglich ist das implementierte Verfahren ENUMERATE_M auch *vollständig*.

Satz 9.3. Wenn $M \subseteq \mathbb{N}$ semi-entscheidbar ist, so ist M rekursiv aufzählbar.

Beweis. Für $M = \emptyset$ ist die Behauptung trivialerweise wahr. Andernfalls können wir ein beliebiges Element $m_0 \in M$ wählen. Da M semi-entscheidbar ist, gibt es ein semi-Entscheidungsverfahren SEMI_DECIDE_M $\in \mathcal{P}[1]$ für M . Für die Prozedur ENUMERATE_M $\in \mathcal{P}[1]$ aus Abbildung 9.1 hält die Ausführung von ENUMERATE_M(i) für beliebiges $i \in \mathbb{N}$, denn die Ausführung von STEP(PAIR³(⊥SEMI_DECIDE_M,b,element)) hält immer und SEMI_DECIDE_M(element) wird nur dann ausgeführt, wenn ein Ergebnis in weniger als b Schritten berechnet werden kann.

Weiter gilt $\llbracket \text{ENUMERATE_M} \rrbracket(i) \in M$, denn für $\llbracket \text{ENUMERATE_M} \rrbracket(i) = m$ gilt entweder $m = m_0 \in M$ oder $\llbracket \text{SEMI_DECIDE_M} \rrbracket(m) = 1$, und damit ebenfalls $m \in M$. Also ist $\llbracket \text{ENUMERATE_M} \rrbracket : \mathbb{N} \rightarrow \mathbb{N}$ eine totale berechenbare Funktion mit $\llbracket \text{ENUMERATE_M} \rrbracket(\mathbb{N}) \subseteq M$.

Es ist jetzt noch zu zeigen, daß $\llbracket \text{ENUMERATE_M} \rrbracket(\mathbb{N}) = M$ gilt. D.h. wir zeigen, daß für jedes $m \in M$ ein $i \in \mathbb{N}$ mit $\llbracket \text{ENUMERATE_M} \rrbracket(i) = m$ existiert: Für jedes $m \in M$ gilt $\llbracket \text{SEMI_DECIDE_M} \rrbracket(m) = 1$, und folglich liefert SEMI_DECIDE_M(m) nach endlich vielen, also etwa b_m Schritten, das Ergebnis 1. Für $i := \pi^2(b_m + 1, m)$ erhält man $\llbracket \text{ENUMERATE_M} \rrbracket(i) = m$, denn mit $\llbracket \text{PAIR}_1^2 \rrbracket = \pi_1^2$ und $\llbracket \text{PAIR}_2^2 \rrbracket = \pi_2^2$ gilt $b = b_m + 1$ und $\text{element} = m$.

Folglich gilt $\llbracket \text{STEP} \rrbracket(\text{PAIR}^3(\text{SEMIDECIDE_M}, b_m + 1, m)) = \llbracket \text{SEMIDECIDE_M} \rrbracket(m) = 1$ und damit $\llbracket \text{ENUMERATE_M} \rrbracket(i) = \pi_2^2(i) = \pi_2^2(\pi^2(b_m + 1, m)) = m$. Also ist `ENUMERATE_M` ein Aufzählungsverfahren für M , d.h. M ist rekursiv aufzählbar. ■

Korollar 9.4. $M \subseteq \mathbb{N}$ ist genau dann semi-entscheidbar, wenn M rekursiv aufzählbar ist.

Beweis. Offensichtlich mit Satz 9.2 und Satz 9.3. ■

Semi-Entscheidbarkeit und rekursive Aufzählbarkeit sind also äquivalente Begriffe, die lediglich unterschiedlich definiert sind. Wir machen uns dies zunutze, indem wir bei den weiteren Untersuchungen jeweils denjenigen Begriff wählen, dessen Definition für uns am bequemsten ist.

Bemerkung 9.5. Mit der Beweisidee von Satz 9.2 überlegt man sich leicht, daß jede Menge M , deren Elemente durch Herleitung mittels eines Regelsystems definiert sind, semi-entscheidbar ist. Man erzeugt dazu die Elemente von M in systematischer Weise, etwa indem die Regelanwendungen durch eine Breitensuche organisiert werden, und vergleicht die erzeugten Elemente jeweils mit demjenigen m , für daß “ $m \in M$?” überprüft werden soll. Gilt $m \in M$, so wird m auf diese Weise nach endlicher Zeit erzeugt, so daß man ein positives Ergebnis erhält. Für $m \notin M$ muß das Erzeugungsverfahren dagegen nicht terminieren. Beispielsweise folgt so, daß die Menge der allgemeingültigen prädikatenlogischen Formeln semi-entscheidbar ist, denn aufgrund der Vollständigkeit der Prädikatenlogik 1. Stufe, ist jede allgemeingültige Formel durch einen prädikatenlogischen Kalkül herleitbar. Weitere Beispiele sind Wortmengen, die durch generative Grammatiken (Chomsky-Grammatiken) definiert werden. In gewissen Fällen kann so auch die semi-Entscheidbarkeit des Komplements einer Menge bewiesen werden. Beispielsweise gilt für kontextsensitive Sprachen (= Typ-1 Sprachen), daß ein Wort aufgrund der Längenbeschränkung der Grammatikregeln innerhalb einer (durch das Wort gegebenen) Suchtiefe herleitbar sein muß. Wird also ein m nicht innerhalb der Suchtiefe gefunden, so ist $m \in \overline{M}$ nach endlicher Zeit festgestellt.

Mittels rekursiver Aufzählbarkeit kann man zeigen, daß jede unendliche, rekursiv aufzählbare Menge eine unendliche, entscheidbare Teilmenge enthält. Zum Beweis benötigen wir den Begriff *aufsteigend rekursiv aufzählbar*. Damit ist gemeint, daß die Aufzählungsfunktion α_M einer Menge M *monoton* ist, also $i < j \curvearrowright \alpha_M(i) < \alpha_M(j)$ für alle $i, j \in \mathbb{N}$ gilt. Folgende Überlegung zeigt, daß aufsteigend rekursiv aufzählbare Mengen entscheidbar sind: Wie in Satz 9.2 erzeugen wir mittels der Aufzählungsfunktion α_M systematisch die Elemente von M und vergleichen diese mit $x \in \mathbb{N}$. Für $\alpha_M(i) = x$ gilt $x \in M$, die Suche kann also mit



positivem Ergebnis beendet werden. Andernfalls prüfen wir, ob $\alpha_M(i) > x$ gilt. Ist dies der Fall, so kann x nicht in $\alpha_M(\mathbb{N})$ enthalten sein, denn aufgrund der Monotonie von α_M gilt dann $\alpha_M(j) > x$ für alle $j > i$. Die Suche kann also mit negativem Ergebnis abgebrochen werden. Ansonsten muß weiter gesucht werden. Die Suche endet jedoch nach endlich vielen Schritten, denn wegen der Monotonie von α_M gilt nach endlich vielen Schritten $\alpha_M(i) \geq x$.

Satz 9.6. $M \subseteq \mathbb{N}$ ist entscheidbar, falls M aufsteigend rekursiv aufzählbar ist.

Beweis. Für $M = \emptyset$ gilt $\chi_M(n) = 0$ für alle $n \in \mathbb{N}$. Damit ist M entscheidbar, denn χ_M ist trivialerweise berechenbar. Andernfalls gibt es ein monotones Aufzählungsverfahren $\text{ENUMERATE_M} \in \mathcal{P}$ für M . Für das Programm $\text{DECIDE_M} \in \mathcal{P}$ mit

```

procedure DECIDE_M(x) <=
begin var i, res;
  while ENUMERATE_M(i) < x do i := SUCC(i) end_while;
  if ENUMERATE_M(i) = x then res := 1 end_if;
  return(res)
end

```

gilt

$$\llbracket \text{DECIDE_M} \rrbracket(m) = \begin{cases} 0 & , \text{ falls } \llbracket \text{ENUMERATE_M} \rrbracket(i) > m \text{ für ein } i \in \mathbb{N} \\ 1 & , \text{ falls } \llbracket \text{ENUMERATE_M} \rrbracket(i) = m \text{ für ein } i \in \mathbb{N} . \end{cases}$$

Da ENUMERATE_M ein Aufzählungsverfahren für M ist, gilt $\llbracket \text{ENUMERATE_M} \rrbracket(\mathbb{N}) = M$, und damit gibt es für jedes $m \in M$ ein (kleinstes) $i \in \mathbb{N}$ mit $\text{ENUMERATE_M}(i) = m$. Für $m \in M$ bricht die Schleife in DECIDE_M also nach i Schleifendurchläufen ab, und man erhält das Ergebnis 1. Für $m \notin M$ gilt $\llbracket \text{ENUMERATE_M} \rrbracket(j) > m$ für ein (kleinstes) $j \in \mathbb{N}$ aufgrund der Monotonie von $\llbracket \text{ENUMERATE_M} \rrbracket$. Also bricht die Schleife dann nach j Schritten ab, und man erhält das Ergebnis 0. Damit gilt

$$\llbracket \text{DECIDE_M} \rrbracket(m) = \begin{cases} 0 & , \text{ falls } m \notin M \\ 1 & , \text{ falls } m \in M , \end{cases}$$

und folglich $\chi_M = \llbracket \text{DECIDE_M} \rrbracket$. Damit ist M entscheidbar. ■



Die Existenz einer unendlichen und entscheidbaren Teilmenge einer unendlichen, rekursiv aufzählbaren Menge M zeigen wir jetzt folgendermaßen: Aus einem Aufzählungsverfahren α_M von M bilden wir eine berechenbare Funktion $\beta : \mathbb{N} \rightarrow M$, so daß β monoton ist. Damit ist $\beta(\mathbb{N})$ sowohl unendlich als auch aufsteigend rekursiv aufzählbar, also mit Satz 9.6 auch entscheidbar.

Satz 9.7. Sei $M \subseteq \mathbb{N}$ unendlich und rekursiv aufzählbar. Dann enthält M eine unendliche entscheidbare Menge.

Beweis. Sei α_M eine Aufzählungsfunktion für M und sei $\text{ALPHA_M} \in \mathcal{P}$ mit $\llbracket \text{ALPHA_M} \rrbracket = \alpha_M$. Wie definieren das \mathcal{P} -Programm **BETA** durch

```

procedure BETA(i) <=
begin var j, k, b;
  b := ALPHA_M(0);
  while k < i
    do if b < ALPHA_M(j)
      then k := k + 1;
         b := ALPHA_M(j);
      end_if;
      j := j + 1
    end_while;
  return(b)
end

```

Angenommen, es gibt ein $i \in \mathbb{N}$ so daß $i < j \curvearrowright \alpha_M(i) \geq \alpha_M(j)$ für alle $j \in \mathbb{N}$. Dann enthält M höchstens $i + (\alpha_M(i) - 1)$ Elemente und ist somit endlich. ▼ Also gibt es für jedes $i \in \mathbb{N}$ ein $j \in \mathbb{N}$, so daß $(*) i < j \wedge \alpha_M(i) < \alpha_M(j)$ gilt. Sei $n_0 := 0$ sowie $n_{k+1} := \min \{j \in \mathbb{N} \mid n_k < j \wedge \alpha_M(n_k) < \alpha_M(j)\}$. Wegen $(*)$ ist n_k wohldefiniert, und es gilt $\alpha_M(n_k) < \alpha_M(n_{k+1})$ für alle $k \in \mathbb{N}$. Folglich ist $\beta : \mathbb{N} \rightarrow M$ definiert durch $\beta(i) := \alpha_M(n_i)$ monoton. Für die **while**-Anweisung der Prozedur **BETA** gilt die Schleifeninvariante

$$\begin{array}{l}
\{\mathbf{b} := \alpha_M(n_k)\} \\
\text{while } k < i \text{ do } \dots \text{ end_while} \\
\{\mathbf{b} := \alpha_M(n_k)\}
\end{array} \tag{9.1}$$

denn für $k = h$ und $\mathbf{b} = \alpha_M(n_h)$ vor Ausführung des Schleifenrumpfs erhält man nach dessen Ausführung $k = h + 1$ und $\mathbf{b} = \alpha_M(n_{h+1})$, falls $\mathbf{b} < \text{ALPHA_M}(j)$, und $k = h$ und $\mathbf{b} = \alpha_M(n_h)$ andernfalls. Da der Wert der Programmvariablen j bei jeder Ausführung des Schleifenrumpfs erhöht wird, terminiert die **while**-Schleife wegen $(*)$. Dann gilt $k = i$ und folglich $\mathbf{b} = \alpha_M(n_k) = \alpha_M(n_i) = \beta(i)$. Also gilt $\beta = \llbracket \text{BETA} \rrbracket$, somit ist $\beta(\mathbb{N})$ mittels β aufsteigend rekursiv aufzählbar und mit Satz 9.6 entscheidbar. Schließlich ist $\beta(\mathbb{N})$ unendlich, denn β ist monoton. ■



Beispielsweise ist die Menge AG der allgemeingültigen Formeln der Prädikatenlogik erster Stufe semi-entscheidbar, also rekursiv aufzählbar. Ein Beispiel für eine unendliche, entscheidbare Teilmenge von AG ist die Menge TAU der Tautologien, also der wahren aussagenlogischen Formeln, s.a. Abschnitt 10.6.

9.2. Abzählbarkeit und rekursive Aufzählbarkeit

Es ist wichtig, zwischen den Begriffen der *Abzählbarkeit* und der *rekursiven Aufzählbarkeit* einer Menge M zu unterscheiden.² In beiden Fällen kann $M \neq \emptyset$ als Bild einer totalen und surjektiven Aufzählungsfunktion $\alpha : \mathbb{N} \rightarrow M$ dargestellt werden, d.h. $M = \alpha(\mathbb{N})$. Für die *rekursive Aufzählbarkeit* wird zusätzlich gefordert, daß α eine *berechenbare* Funktion ist. Daraus folgt, daß jede rekursiv aufzählbare Menge auch abzählbar ist. Umgekehrt gilt das jedoch nicht. Insbesondere können wir Satz 5.3 nicht für rekursive Aufzählbarkeit formulieren. Anders gesagt, eine Teilmenge einer rekursiv aufzählbaren Menge ist nicht notwendigerweise auch rekursiv aufzählbar, wie man leicht zeigt:

Die Menge \mathbb{N} ist trivialerweise rekursiv aufzählbar, denn die Identitätsfunktion ist offenbar berechenbar. Damit ist \mathbb{N} natürlich ebenso trivialerweise abzählbar. Mit $\overline{H} \subseteq \mathbb{N}$ ist dann auch das Komplement \overline{H} des Halteproblems abzählbar, vgl. Satz 5.3. \overline{H} ist jedoch nicht rekursiv aufzählbar, denn \overline{H} ist nicht semi-entscheidbar, vgl. Satz 8.8 und Satz 9.2. Wir haben also mit \overline{H} eine abzählbare, aber nicht rekursiv aufzählbare Teilmenge einer rekursiv aufzählbaren Menge, nämlich \mathbb{N} , angegeben.

9.3. Definitions- und Bildbereiche berechenbarer Funktionen

Semi-Entscheidbarkeit bzw. rekursive Aufzählbarkeit kann durch die Definitions- und Bildbereiche berechenbarer Funktionen charakterisiert werden:

Satz 9.8. $M \subseteq \mathbb{N}$ ist genau dann semi-entscheidbar, wenn $M = \text{Def}(\phi)$ für eine berechenbare Funktion $\phi : \mathbb{N} \mapsto \mathbb{N}$.

Beweis. “ \Rightarrow ” Mit $\phi := \tilde{\chi}_M$.

“ \Leftarrow ” Sei $M = \text{Def}(\phi)$ für eine berechenbare Funktion $\phi : \mathbb{N} \mapsto \mathbb{N}$, und sei $\text{PHI} \in \mathcal{P}[1]$ mit $\llbracket \text{PHI} \rrbracket = \phi$. Dann gilt $\llbracket \text{SEMI_DECIDE_M} \rrbracket = \tilde{\chi}_M$ für die Prozedur $\text{SEMI_DECIDE_M} \in \mathcal{P}[1]$ aus Abbildung 9.2, denn $\text{PHI}(m)$ terminiert genau dann, wenn $m \in \text{Def}(\phi)$, d.h. wenn $m \in M$. ■

Satz 9.9. $M \subseteq \mathbb{N}$ ist genau dann semi-entscheidbar, wenn $M = \text{Bild}(\phi)$ für eine berechenbare Funktion $\phi : \mathbb{N} \mapsto \mathbb{N}$.

² Wir folgen hier dem allgemeinen Sprachgebrauch. Man könnte genauso von *Aufzählbarkeit* oder von *rekursiver Abzählbarkeit* sprechen. Anders gesagt, mit “auf” und “ab” wird keine weitere Unterscheidung ausgedrückt.

```

procedure SEMI_DECIDE_M(m) <=
begin var y,z;
  y := PHI(m);
  z := 1;
  return(z)
end

```

Abbildung 9.2: Ein semi-Entscheidungsverfahren für $Def(\phi)$

Beweis. “ \Rightarrow ” Mit Satz 9.3 gibt es eine berechenbare Aufzählungsfunktion $\alpha_M : \mathbb{N} \rightarrow \mathbb{N}$ mit $\alpha_M(\mathbb{N}) = M$, d.h. $M = Bild(\alpha_M)$.

“ \Leftarrow ” Sei $M = Bild(\phi)$ für eine berechenbare Funktion $\phi : \mathbb{N} \mapsto \mathbb{N}$, und sei $PHI \in \mathcal{P}[1]$ mit $\llbracket PHI \rrbracket = \phi$. Für die \mathcal{P} -Prozedur SEMI_DECIDE_M aus Abbildung 9.3 gilt dann

$$\llbracket SEMI_DECIDE_M \rrbracket(m) = \begin{cases} 1 & , \text{ falls } m = \phi(n) \text{ für ein } n \in \mathbb{N} \\ \perp & , \text{ falls } m \neq \phi(n) \text{ für alle } n \in \mathbb{N} \end{cases}$$

und mit $M = Bild(\phi)$ folglich

$$\llbracket SEMI_DECIDE_M \rrbracket(m) = \begin{cases} 1 & , \text{ falls } m \in M \\ \perp & , \text{ falls } m \notin M \end{cases}$$

also $\llbracket SEMI_DECIDE_M \rrbracket = \tilde{\chi}_M$. ■

Mit Satz 9.8 ist der Definitionsbereich einer berechenbaren Funktion mindestens semi-entscheidbar. Für Programme, die partielle Funktionen mit *entscheidbarem* Definitionsbereich berechnen, kann man die nicht-Terminierung einfach ausschließen, indem man vor Programmausführung entscheidet, ob die Eingabe Element des Definitionsbereichs ist. Beispielsweise resultiert ein Prozeduraufruf QUOT(EXPR, 0) zur Division in gängigen Programmiersprachen in einem Laufzeitfehler und nicht in einem unendlichen Programmablauf, denn es ist natürlich entscheidbar, ob der Divisor identisch 0 ist oder nicht.

Formal kann man den Unterschied zwischen “harmlosen” partiellen berechenbaren Funktionen, wie z.B. $\llbracket QUOT \rrbracket$, und “inhärent” partiellen berechenbaren Funktionen, wie z.B. $\tilde{\chi}_H$, wie folgt charakterisieren:

Definition 9.10. Seien $\phi, \psi : \mathbb{N} \mapsto \mathbb{N}$. Dann ist ϕ weniger definiert oder gleich ψ , kurz $\phi \sqsubseteq \psi$, gdw. $\phi(n) = \perp$ oder $\phi(n) = \psi(n)$ für alle $n \in \mathbb{N}$ gilt.

```

procedure SEMI_DECIDE_M(m) <=
begin var b,n,i,res;
  while res = 0 do
    n := PAIR12(i);
    b := PAIR22(i);
    if STEP(PAIR3(⊥PHI,b,n))
      then if m = PHI(n) then res := 1 else i := SUCC(i) end_if
      else i := SUCC(i)
    end_if
  end_while;
return(res)
end

```

Abbildung 9.3: Ein semi-Entscheidungsverfahren für $Bild(\phi)$

Satz 9.11. Sei $\phi : \mathbb{N} \mapsto \mathbb{N}$ berechenbar, so daß $Def(\phi)$ entscheidbar ist. Dann gibt es eine totale berechenbare Funktion $\psi : \mathbb{N} \rightarrow \mathbb{N}$ mit $\phi \sqsubseteq \psi$.

Beweis. Sei $m_0 \in \mathbb{N}$ beliebig aber fest gewählt, sei PHI eine \mathcal{P} -Prozedur mit $\llbracket PHI \rrbracket = \phi$ und sei DECIDE_DEF_PHI $\in \mathcal{P}$ [1] ein Entscheidungsverfahren für $Def(\phi)$. Für die \mathcal{P} -Prozedur PSI mit

```

procedure PSI(n) <=
begin var res;
  if DECIDE_DEF_PHI(n) then res := PHI(n) else res := m0 end_if;
  return(res)
end

```

gilt dann

$$\llbracket PSI \rrbracket(n) = \begin{cases} \phi(n) & , \text{ falls } n \in Def(\phi) \\ m_0 & , \text{ falls } n \notin Def(\phi) . \end{cases}$$

Die Prozedur PSI terminiert, denn DECIDE_DEF_PHI terminiert und PHI(n) terminiert, wenn DECIDE_DEF_PHI(n) gilt. Für $\psi := \llbracket PSI \rrbracket$ gilt dann $\phi \sqsubseteq \psi$, ψ ist berechenbar und ψ ist total. ■

Mit Satz 9.11 kann also jede partielle berechenbare Funktion ϕ mit entscheidbarem Definitionsbereich zu einer totalen berechenbaren Funktion ψ erweitert werden, die auf $Def(\phi)$ das gleiche leistet wie ϕ . In der Praxis macht man sich dies zunutze, indem man Programme, die partielle Funktionen berechnen, anstatt in eine Endlosschleife zu geraten mit Laufzeitfehlern terminieren läßt, wenn die Eingabe außerhalb des Definitionsbereichs einer solchen Funktion liegt.

10. UNENTSCHEIDBARE PROBLEME

10.1. Das Totalitätsproblem

Da wir das Halteproblem nicht entscheiden können, probieren wir einen anderen Ansatz, um nicht-terminierende Programme zu vermeiden. Man könnte ja versuchen eine Programmiersprache zu entwerfen, mit deren Programmen genau die totalen (berechenbaren) Funktionen implementiert werden können. Damit würde ein nicht-terminierendes Programm schon durch einen Übersetzer ausgeschlossen. Ein Gelingen dieses Ansatzes setzt jedoch voraus, daß das *Totalitätsproblem*

$$TOT = \{i \in \mathbb{N} \mid \varphi_i(x) \neq \perp \text{ für alle } x \in \mathbb{N}\}$$

entscheidbar ist. Das Totalitätsproblem ist jedoch noch nicht einmal rekursiv aufzählbar:

Satz 10.1. *TOT ist nicht rekursiv aufzählbar.*

Beweis. Wir führen einen Widerspruchsbeweis und nehmen an, daß *TOT* rekursiv aufzählbar sei. Dann gibt es eine totale und berechenbare Aufzählungsfunktion $\alpha : \mathbb{N} \rightarrow \mathbb{N}$ mit $\alpha(\mathbb{N}) = TOT$. Folglich gibt es eine \mathcal{P} -Prozedur ALPHA mit $\llbracket \text{ALPHA} \rrbracket = \alpha$, und mit der \mathcal{P} -Prozedur PSI, definiert durch

```
procedure PSI(x) <=  
begin var y;  
  y := SUCC(APPLY(PAIR2(ALPHA(x), x)));  
  return(y)  
end
```

erhält man eine berechenbare Funktion $\llbracket \text{PSI} \rrbracket$, für die $\llbracket \text{PSI} \rrbracket(j) = 1 + \mathbf{u}_{\mathcal{P}}(\pi^2(\alpha(j), j)) = 1 + \varphi_{\alpha(j)}(j)$ gilt. Mit $\alpha(j) \in TOT$ ist $\varphi_{\alpha(j)}$ total. Damit ist auch $\llbracket \text{PSI} \rrbracket$ total, und folglich gilt $\natural\text{PSI} \in TOT$. Mit der Surjektivität von α gibt es ein $i \in \mathbb{N}$, so daß $\alpha(i) = \natural\text{PSI}$. Damit gilt $\llbracket \text{PSI} \rrbracket = \varphi_{\alpha(i)}$ und man erhält $\llbracket \text{PSI} \rrbracket(i) = 1 + \varphi_{\alpha(i)}(i) = 1 + \llbracket \text{PSI} \rrbracket(i)$. Dies ist aber unmöglich, denn $\llbracket \text{PSI} \rrbracket$ ist total, also gilt $\llbracket \text{PSI} \rrbracket(i) \in \mathbb{N}$ und damit $\llbracket \text{PSI} \rrbracket(i) \neq 1 + \llbracket \text{PSI} \rrbracket(i)$. \blacktriangledown Also ist *TOT* nicht rekursiv aufzählbar. \blacksquare

Bemerkung 10.2. *Im Beweis von Satz 10.1 wird die gleiche Argumentationsweise, nämlich die Diagonalisierung, verwendet wie im Beweis von Satz 5.4. Trotzdem folgt Satz 10.1 nicht aus Satz 5.4, da dieser eine Aussage über die Menge aller totalen Funktionen macht und aus der nicht-Abzählbarkeit einer Menge – hier $\mathcal{F}_{tot} = \{\phi : \mathbb{N} \rightarrow \mathbb{N}\}$ – nicht folgt, daß eine Teilmenge – hier $\{\varphi_i : \mathbb{N} \rightarrow \mathbb{N} \mid i \in TOT\}$ – auch nicht abzählbar ist. Umgekehrt folgt auch Satz 5.4 nicht aus Satz 10.1, denn TOT ist abzählbar, und so kann nicht mit Satz 5.3 auf die nicht-Abzählbarkeit von \mathcal{F}_{tot} geschlossen werden.*

Eine Konsequenz von Satz 10.1 ist, daß es für jede Programmiersprache, in der *nur* terminierende Programme geschrieben werden können – wir werden solch eine Programmiersprache in Kapitel 11 definieren –, eine totale berechenbare Funktion gibt, die durch kein Programm dieser Programmiersprache berechnet werden kann. Verwendet man dagegen eine Programmiersprache, in der *alle* totalen berechenbaren Funktionen programmiert werden können – was für \mathcal{P} und für alle gängigen Programmiersprachen gilt –, so muß man in Kauf nehmen, daß auch nicht-terminierende Programme in dieser Programmiersprache geschrieben werden können.

Eine weitere Konsequenz von Satz 10.1 ist, daß jedes *Terminierungsbeweisverfahren*, also ein Verfahren, das für *terminierende* Prozeduren als Eingabe einen Terminierungsbeweis für diese Prozedur berechnet, *unvollständig* ist. Anders gesagt, für jedes (korrekte) Terminierungsbeweisverfahren kann man eine terminierende Prozedur angeben, für die das Verfahren *keinen* Beweis berechnen kann. Dabei ist natürlich nicht ausgeschlossen, daß man die Totalität von berechenbaren Funktionen nicht doch beweisen oder widerlegen kann. Mit Satz 10.1 ist es nur unmöglich, ein Verfahren anzugeben, mit dem die Totalität *jeder* totalen berechenbaren Funktion bewiesen wird.

10.2. Reduzierbarkeit

In den Abschnitten 3.3 und 7.2 wurde schon gezeigt, daß man die Berechenbarkeit einer Funktion nachweisen kann, ohne konkret ein Programm anzugeben, das diese Funktion berechnet. Ursache war dort, daß man für eine berechenbare Funktion ϕ eine Menge von Programmen angeben kann, so daß eines dieser Programme die Funktion auch tatsächlich berechnet, jedoch unbekannt ist, *welches* dieser Programme konkret das Gewünschte leistet.

In diesem Abschnitt betrachten wir eine andere Vorgehensweise, um die Berechenbarkeit einer Funktion nachzuweisen, ohne dabei ein konkretes Programm anzugeben. Seien beispielsweise $\phi : \mathbb{N} \rightarrow \mathbb{N}$ und $\psi : \mathbb{N} \rightarrow \mathbb{N}$ Funktionen mit

$\psi(n) = \phi(\pi^2(n, n))$ für alle $n \in \mathbb{N}$. Um zu zeigen, daß ψ \mathcal{P} -berechenbar ist, reicht es offensichtlich nachzuweisen, daß ϕ berechenbar ist. Wir müssen dann für ψ kein konkretes Programm $P \in \mathcal{P}$ mit $\psi = \llbracket P \rrbracket$ angeben. Wir haben hier die Aufgabe, die Berechenbarkeit von ψ zu beweisen, auf die Aufgabe, die Berechenbarkeit von ϕ nachzuweisen, *reduziert*.

Man kann natürlich einwenden, daß wir hier ja nur eine Aufgabe durch eine andere Aufgabe ersetzt haben, und somit nichts gewonnen ist. Daher versucht man, ein Problem auf ein *gelöstes* Problem zu *reduzieren*. Beispielsweise können wir die Frage, ob die Quadratfunktion x^2 berechenbar ist, auf die Frage reduzieren, ob die Multiplikationsfunktion $x \times y$ berechenbar ist, denn $x^2 = x \times x$. Da wir wissen, daß $x \times y$ berechenbar ist, ist die Berechenbarkeit von x^2 ohne Angabe eines konkreten Programms bewiesen.

Wir präzisieren jetzt den Begriff “reduzierbar” und definieren:

Definition 10.3. Ein Problem $M \subseteq \mathbb{N}$ ist genau dann auf ein Problem $N \subseteq \mathbb{N}$ reduzierbar, wenn es eine totale und berechenbare Funktion $\varrho : \mathbb{N} \rightarrow \mathbb{N}$ gibt, so daß $m \in M \Leftrightarrow \varrho(m) \in N$ für alle $m \in \mathbb{N}$ gilt. In diesem Fall schreiben wir kurz $M \preceq_{\varrho} N$. ■

Die Reduzierbarkeit von M auf N bedeutet also, M in *berechenbarer* Weise, nämlich mittels ϱ , als *Spezialfall* von N darzustellen. Dies kann dann ausgenutzt werden, um von der (semi-)Entscheidbarkeit von N auf die (semi-)Entscheidbarkeit von M zu schließen:

Satz 10.4. Für $M \subseteq \mathbb{N}$ und $N \subseteq \mathbb{N}$ mit $M \preceq_{\varrho} N$ gilt:

1. Wenn N entscheidbar ist, so ist auch M entscheidbar.
2. Wenn N semi-entscheidbar ist, so ist auch M semi-entscheidbar.

Beweis. (1) Mit Satz 3.3 ist $\chi_N \circ \varrho$ berechenbar, denn χ_N und ϱ sind nach Voraussetzung berechenbar. Wenn $m \in M$, so gilt $\varrho(m) \in N$ mit der Definition von ϱ , und mit der Definition von χ_N gilt dann $\chi_N(\varrho(m)) = 1$. Für $m \notin M$ gilt $\varrho(m) \notin N$ mit der Definition von ϱ , und mit der Definition von χ_N gilt dann $\chi_N(\varrho(m)) = 0$. Also gilt $\chi_N \circ \varrho = \chi_M$.

(2) Wie in (1) zeigt man $\tilde{\chi}_N \circ \varrho \cong_M \tilde{\chi}_M$. ■

Beispielsweise können wir so einfach zeigen, daß das Selbstanwendbarkeitsproblem S semi-entscheidbar ist, indem S auf das Halteproblem H reduziert wird (vgl. den Beweis von Korollar 7.12): Wir definieren einfach $\varrho : \mathbb{N} \rightarrow \mathbb{N}$ durch

$\varrho(n) := \pi^2(n, n)$, und offensichtlich ist ϱ berechenbar. Es gilt $n \in S \Leftrightarrow \varrho(n) \in H$ für alle $n \in \mathbb{N}$, und da H semi-entscheidbar ist, ist mit Satz 10.4(2) auch S semi-entscheidbar.

Anstatt Satz 10.4 direkt in einem Beweis zu verwenden, wird häufig die Kontraposition von Satz 10.4 angewandt:

Korollar 10.5. Für $M \subseteq \mathbb{N}$ und $N \subseteq \mathbb{N}$ mit $M \preceq_{\varrho} N$ gilt:

1. Wenn M nicht entscheidbar ist, so ist auch N nicht entscheidbar.
2. Wenn M nicht semi-entscheidbar ist, so ist auch N nicht semi-entscheidbar.

Beispielsweise können wir mit Korollar 10.5 einfach zeigen, daß das Halteproblem H unentscheidbar ist, indem das Selbstanwendbarkeitsproblem S mit $\varrho(n) := \pi^2(n, n)$ auf H reduziert wird (vgl. den Beweis von Satz 5.9). Da S nicht entscheidbar ist, ist mit Korollar 10.5(1) auch H nicht entscheidbar (siehe auch Bemerkung 10.10).

10.3. Das *s-m-n*-Theorem

Sei `procedure EXP(y, z) <= BODY_EXP` eine \mathcal{P} -Prozedur mit $\llbracket \text{EXP} \rrbracket(y, z) = z^y$.¹ Dann können wir mit

$$\text{procedure SQUARE}(z) \text{ <= BODY_EXP}[y/2]$$

eine Prozedur zur Berechnung der Quadratfunktion angeben. Dabei entsteht der Prozedurrumpf `BODY_EXP[y/2]` von `SQUARE` aus dem Prozedurrumpf `BODY_EXP` von `EXP`, indem in `BODY_EXP` jedes Vorkommen des formalen Parameters `y` durch die Konstante 2 ersetzt wird. Damit gilt $\llbracket \text{SQUARE} \rrbracket(z) = \llbracket \text{EXP} \rrbracket(2, z) = z^2$, d.h. wir haben aus einer 2-stelligen Funktion eine 1-stellige Funktion gewonnen, indem wir für einen formalen Parameter, nämlich `y`, einen festen Wert, nämlich 2, eingesetzt haben. Man sagt auch, die Prozedur `EXP` wurde *parametrisiert*.

Um allgemein parametrisierte Prozeduren zu erhalten, erlauben wir jetzt in einem \mathcal{P} -Programm für eine gegebene Prozedur

$$\text{procedure PROC}(y_1, \dots, y_m, z_1, \dots, z_n) \text{ <= BODY_PROC}$$

¹ In diesem und in den folgenden Abschnitten verwenden wir Prozeduren mit *mehreren* Parametern und arithmetische Funktionen $\phi : \mathbb{N} \times \dots \times \mathbb{N} \mapsto \mathbb{N}$ mit *mehreren* Argumenten. Dies geschieht allein aus Gründen der Lesbarkeit und bedeutet keine Abänderung unseres formalen Rahmens, vgl. Abschnitt 4.3.

Prozeduraufrufe der Form

$$\text{PROC}[k_1, \dots, k_m](\text{EXPR}_1, \dots, \text{EXPR}_n)$$

mit $k_1, \dots, k_m \in \mathbb{N}$, wobei wir uns jede *parametrisierte Prozedur* $\text{PROC}[k_1, \dots, k_m]$ durch

$$\begin{aligned} \text{procedure } \text{PROC}[k_1, \dots, k_m](z_1, \dots, z_n) <= \\ \text{BODY_PROC}[y_1/k_1, \dots, y_m/k_m] \end{aligned} \quad (10.1)$$

definiert vorstellen. Damit stehen uns im Beispiel beliebig viele Prozeduren

$$\begin{aligned} \text{procedure } \text{EXP}[0](z) <= \text{BODY_EXP}[y/0] \\ \text{procedure } \text{EXP}[1](z) <= \text{BODY_EXP}[y/1] \\ \text{procedure } \text{EXP}[2](z) <= \text{BODY_EXP}[y/2] \\ \text{procedure } \text{EXP}[3](z) <= \text{BODY_EXP}[y/3] \\ \dots \end{aligned}$$

zur Verfügung, für die $\llbracket \text{EXP}[k] \rrbracket(z) = \llbracket \text{EXP} \rrbracket(k, z) = z^k$ gilt, wobei k eine beliebige, aber *feste* natürliche Zahl ist.

Es stellt sich natürlich sofort die Frage, warum wir in einem Programm Aufrufe parametrisierter Prozeduren, wie z.B. $\text{EXP}[2](\text{EXPR})$, verwenden sollen, anstatt einfach die Originalprozedur mit den entsprechenden aktuellen Parametern, also etwa $\text{EXP}(2, \text{EXPR})$, aufzurufen. Der Grund ist, daß wir mittels Parametrisierung den Code eines Programms in Abhängigkeit von aktuellen Parametern berechnen können: Für jedes Programm $P \in \mathcal{P}$ gilt $\llbracket P \rrbracket = \varphi_{\natural P}$, also beispielsweise $\llbracket \text{EXP} \rrbracket = \varphi_{\natural \text{EXP}}$ und $\llbracket \text{EXP}[k] \rrbracket = \varphi_{\natural \text{EXP}[k]}$ für jedes $k \in \mathbb{N}$. Mit (10.1) erhält man offenbar $\natural \text{EXP}[k]$ allein aus $\natural \text{EXP}$ und k , denn $\text{EXP}[k]$ wird allein aus EXP und k gebildet. Anders gesagt, es gilt $\natural \text{EXP}[k] = s(\natural \text{EXP}, k)$ für eine Funktion $s : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, und man erhält damit $\llbracket \text{EXP}[k] \rrbracket = \varphi_{s(\natural \text{EXP}, k)}$.²

Diese Konstruktion kann man natürlich mit beliebigen Programmen und beliebigen Parametern durchführen.³ Wir zeigen, daß die Funktion s , mit der der

² Dabei gehen wir davon aus, daß EXP selbst keine Prozeduraufrufe enthält. Dies kann immer durch "Einkopieren" der aufgerufenen Prozedur erreicht werden, denn rekursiv definierte Prozeduren sind verboten, vgl. Definition 2.2. Diese Forderung ist notwendig, da EXP sonst von weiteren Prozeduren P_1, \dots, P_j abhängt und wir folglich $\natural \text{EXP}[k] = s(\natural \text{EXP}, k, P_1, \dots, P_j)$ definieren müßten.

³ Unsere Notation zur Parametrisierung von Prozeduren hat allerdings die Schwäche, daß nur formale Parameter *vom Anfang* der Parameterliste durch Konstanten aus \mathbb{N} ersetzt werden können. Beispielsweise gilt auch $\llbracket \text{EXP} \rrbracket(y, 2) = 2^y$, die 2er Potenz kann aber in unserer Notation nicht durch Parametrisierung von EXP gewonnen werden. Um dieser Schwäche abzuweichen, müßte unsere Notation entsprechend erweitert werden. Wir verzichten darauf, da die verwendete Notation für unsere Zwecke ausreicht.

Kode eines parametrisierten Programms aus dem Ursprungsprogramm und den Parametern gewonnen wird, berechenbar und total ist:

Satz 10.6. (*s-m-n-Theorem*) Sei $i \in \mathbb{N}$ mit $\varphi_i : \mathbb{N}^{m+n} \mapsto \mathbb{N}$. Dann gibt es eine berechenbare und totale Funktion $s_n^m : \mathbb{N}^{m+1} \rightarrow \mathbb{N}$ mit

$$\varphi_{s_n^m(i, y_1, \dots, y_m)}(z_1, \dots, z_n) = \varphi_i(y_1, \dots, y_m, z_1, \dots, z_n) \quad (10.2)$$

für alle $(y_1, \dots, y_m, z_1, \dots, z_n) \in \mathbb{N}^{m+n}$.

Beweis. Sei

procedure PROC($y_1, \dots, y_m, z_1, \dots, z_n$) <= BODY_PROC

eine Prozedur mit $\natural\text{PROC} = i$. Dann berechnen wir $s_n^m(i, y_1, \dots, y_m)$ wie folgt: Jedem m -Tupel $(y_1, \dots, y_m) \in \mathbb{N}^m$ ordnen wir die Prozedur

$$\begin{aligned} \text{procedure PROC}[y_1, \dots, y_m](z_1, \dots, z_n) <= \\ \text{BODY_PROC}[y_1/y_1, \dots, y_m/y_m] \end{aligned} \quad (10.3)$$

zu, und damit gilt

$$\llbracket \text{PROC}[y_1, \dots, y_m] \rrbracket(z_1, \dots, z_n) = \llbracket \text{PROC} \rrbracket(y_1, \dots, y_m, z_1, \dots, z_n). \quad (10.4)$$

Mit $\text{PROC}[y_1, \dots, y_m]$ definieren jetzt

$$s_n^m(\natural\text{PROC}, y_1, \dots, y_m) := \natural\text{PROC}[y_1, \dots, y_m]. \quad (10.5)$$

Da wir $\text{PROC}[y_1, \dots, y_m]$ mit (10.3) allein aus $\natural\text{PROC}$ und y_1, \dots, y_m bilden, ist s_n^m berechenbar: Wir dekodieren $\natural\text{PROC}$ mit \natural^{-1} zu der Prozedur PROC , ersetzen dann die formalen Parameter y_i im Rumpf von PROC durch y_i und berechnen schließlich mit \natural den Kode $\natural\text{PROC}[y_1, \dots, y_m]$ der so erhaltenen Prozedur $\text{PROC}[y_1, \dots, y_m]$. Da die Funktionen \natural und \natural^{-1} total und algorithmisch sind, ist s_n^m total und berechenbar. Es gilt

$$\begin{aligned} & \varphi_{s_n^m(\natural\text{PROC}, y_1, \dots, y_m)}(z_1, \dots, z_n) \\ &= \varphi_{\natural\text{PROC}[y_1, \dots, y_m]}(z_1, \dots, z_n) \quad , \text{ mit (10.5),} \\ &= \llbracket \text{PROC}[y_1, \dots, y_m] \rrbracket(z_1, \dots, z_n) \quad , \text{ denn } \varphi_{\natural\text{PROC}[y_1, \dots, y_m]} = \llbracket \text{PROC}[y_1, \dots, y_m] \rrbracket, \\ &= \llbracket \text{PROC} \rrbracket(y_1, \dots, y_m, z_1, \dots, z_n) \quad , \text{ mit (10.4),} \\ &= \varphi_{\natural\text{PROC}}(y_1, \dots, y_m, z_1, \dots, z_n) \quad , \text{ denn } \llbracket \text{PROC} \rrbracket = \varphi_{\natural\text{PROC}}, \end{aligned}$$

und damit ist (10.2) bewiesen. ■

Wir illustrieren die Anwendung des *s-m-n*-Theorems mit dem Beweis des folgenden Satzes:

Satz 10.7. *Es gibt eine berechenbare und totale Funktion $h : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, so daß für alle Funktionen $\varphi_j : \mathbb{N}^n \mapsto \mathbb{N}$ und $\varphi_i : \mathbb{N} \mapsto \mathbb{N}$ gilt:*

$$\varphi_{h(i,j)} = \varphi_i \circ \varphi_j. \quad (10.6)$$

Beweis. Sei $\theta : \mathbb{N}^{n+2} \mapsto \mathbb{N}$ definiert durch $\theta(i, j, x_1, \dots, x_n) := \varphi_i(\varphi_j(x_1, \dots, x_n))$. Für die \mathcal{P} -Prozedur

```

procedure THETA(i, j, x1, ..., xn) <=
begin var y;
  y := APPLY(PAIR2(i, APPLY(PAIRn+1(j, x1, ..., xn))));
  return(y)
end

```

gilt dann $\llbracket \text{THETA} \rrbracket = \theta$, damit ist θ berechenbar, und man erhält $\theta = \varphi_{\llbracket \text{THETA} \rrbracket}$. Mit dem *s-m-n*-Theorem 10.6 ist dann $h : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ mit $h(i, j) := s_n^2(\llbracket \text{THETA} \rrbracket, i, j)$ total und berechenbar, und mit

$$\begin{aligned}
\varphi_{h(i,j)}(x_1, \dots, x_n) &= \varphi_{s_n^2(\llbracket \text{THETA} \rrbracket, i, j)}(x_1, \dots, x_n) && \text{, mit Definition } h, \\
&= \varphi_{\llbracket \text{THETA} \rrbracket}(i, j, x_1, \dots, x_n) && \text{, mit dem s-m-n-Theorem 10.6,} \\
&= \theta(i, j, x_1, \dots, x_n) && \text{, mit Definition THETA,} \\
&= \varphi_i(\varphi_j(x_1, \dots, x_n)) && \text{, mit Definition } \theta .
\end{aligned}$$


ist (10.6) bewiesen. ■

10.4. Programmäquivalenz

Wenn wir ein Programm schreiben, um eine bestimmte Aufgabe mittels eines Rechners zu lösen, müssen wir uns vergewissern, daß das Programm auch das Geforderte leistet. Wir müssen also unsere Programme *verifizieren*. Dabei stellt sich natürlich die Frage, wie wir angeben, was genau das Programm leisten soll, d.h. wie man ein Programm *spezifiziert*.

Ein Ansatz dazu besteht darin, daß wir ein alternatives Programm angeben und dann zeigen, daß beide Programme die gleiche Funktion berechnen, d.h. daß beide Programme *äquivalent* sind. Natürlich ist dieser Ansatz nicht für jedes Programm praktikabel – es ergibt offenbar keinen Sinn etwa einen Übersetzer zu spezifizieren, indem man einen zweiten Übersetzer schreibt und dann zeigt, daß

beide Programme das gleiche leisten. Für bestimmte Aufgaben ist dieser Ansatz jedoch trotzdem nützlich:

In der Informatik sind wir auch an *effizienten* Programmen interessiert, d.h. an Programmen, die möglichst wenig Speicher bzw. Rechenzeit benötigen. Nun kommt es oft vor, daß für eine Aufgabe sowohl ein effizienter, jedoch komplizierter Algorithmus als auch ein einfacher, aber ineffizienter Algorithmus existiert. Beispielsweise ist *InsertionSort*, also “Sortieren durch Einfügen”, einfach aber ineffizient, während *Heapsort* ein effizientes Sortierverfahren ist, das jedoch eine wesentlich kompliziertere Programmstruktur als *InsertionSort* aufweist. Damit ist auch die Verifikation von *Heapsort* wesentlich aufwendiger als die Verifikation von *InsertionSort*. Man kann jedoch *Heapsort* auch dadurch verifizieren, indem man zunächst die Korrektheit von *InsertionSort* direkt zeigt, und dann in einem zweiten Schritt nachweist, daß *InsertionSort* und *Heapsort* äquivalent sind, also die gleichen Funktionen berechnen. Bei diesem Verifikationsansatz verwenden wir also *InsertionSort*, um *Heapsort* zu *spezifizieren*. 

Den Nachweis der Programmäquivalenz wollen wir natürlich einem Rechner überlassen. Anders gesagt, wir wollen ein Werkzeug entwickeln, mit dem für beliebige Programme P_1 und P_2 entschieden werden kann, ob $\llbracket P_1 \rrbracket = \llbracket P_2 \rrbracket$ gilt. Bevor wir beginnen dieses Werkzeug zu entwickeln, untersuchen wir natürlich, ob diese Aufgabe überhaupt lösbar ist. Formal heißt das, wir untersuchen, ob das *Äquivalenzproblem*

$$EQV := \{ \pi^2(i, j) \in \mathbb{N} \mid \varphi_i = \varphi_j \}$$

entscheidbar ist.

Wir beginnen dazu mit einem einfacheren Problem: Wir wollen feststellen, ob ein beliebiges Programm P die Identitätsfunktion berechnet, d.h. ob $\llbracket P \rrbracket(n) = n$ für alle $n \in \mathbb{N}$ gilt. Formal heißt das, wir untersuchen die Entscheidbarkeit des *Identitätsproblems*

$$ID := \{ i \in \mathbb{N} \mid \varphi_i(n) = n \text{ für alle } n \in \mathbb{N} \}.$$

Folgender Satz zeigt jedoch, daß ID noch nicht einmal semi-entscheidbar ist:

Satz 10.8. *ID ist nicht semi-entscheidbar.*

Beweis. Wir führen einen Widerspruchsbeweis und nehmen an, daß ID semi-entscheidbar sei, d.h. $\tilde{\chi}_{ID}$ sei berechenbar. Sei $\psi : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$ definiert durch

$$\psi(i, j) := \begin{cases} j & , \text{ falls } \varphi_i(j) \neq \perp, \\ \perp & , \text{ falls } \varphi_i(j) = \perp. \end{cases}$$

Für das Programm $\text{PSI} \in \mathcal{P}$ mit

```

procedure PSI(i, j) <=
begin var y, z;
  y := j;
  z := APPLY(PAIR2(i, j));
  return(y)
end

```

gilt $\psi = \llbracket \text{PSI} \rrbracket$, d.h. ψ ist mit PSI berechenbar und folglich gilt $\psi = \varphi_{\text{PSI}}$. Mit dem s-m-n-Theorem 10.6 gilt dann

$$\varphi_{\text{PSI}}(i, j) = \varphi_{s_1^1(\text{PSI}, i)}(j). \quad (10.7)$$

Wir zeigen jetzt, daß für alle $i \in \mathbb{N}$ gilt:

$$(\forall j \in \mathbb{N}. \varphi_i(j) \neq \perp) \Leftrightarrow s_1^1(\text{PSI}, i) \in ID. \quad (10.8)$$

“ \Rightarrow ” Mit $\varphi_i(j) \neq \perp$ für alle $j \in \mathbb{N}$ gilt $\psi(i, j) = j$ mit Definition von ψ , also $\varphi_{s_1^1(\text{PSI}, i)}(j) = j$ für alle $j \in \mathbb{N}$ mit (10.7), und folglich $s_1^1(\text{PSI}, i) \in ID$.

“ \Leftarrow ” Gilt $\varphi_i(j) = \perp$ für ein $j \in \mathbb{N}$, so gilt $\psi(i, j) = \perp$ mit Definition von ψ , also $\varphi_{s_1^1(\text{PSI}, i)}(j) \neq j$ mit (10.7), und folglich $s_1^1(\text{PSI}, i) \notin ID$. Damit ist Behauptung (10.8) bewiesen.

Sei jetzt die Funktion $\tau : \mathbb{N} \mapsto \mathbb{N}$ definiert durch $\tau(i) := \tilde{\chi}_{ID}(s_1^1(\text{PSI}, i))$. Dann gilt mit der Definition von $\tilde{\chi}_{ID}$

$$\tau(i) = \begin{cases} 1 & , \text{ falls } s_1^1(\text{PSI}, i) \in ID, \\ \perp & , \text{ falls } s_1^1(\text{PSI}, i) \notin ID. \end{cases} \quad (10.9)$$

Mit Satz 3.3 ist τ berechenbar, denn τ entsteht durch Funktionalkomposition aus den berechenbaren Funktionen $\tilde{\chi}_{ID}$ und s_1^1 . Mit (10.8) können wir (10.9) umformen zu

$$\tau(i) = \begin{cases} 1 & , \text{ falls } \varphi_i(j) \neq \perp \text{ für alle } j \in \mathbb{N}, \\ \perp & , \text{ falls } \varphi_i(j) = \perp \text{ für ein } j \in \mathbb{N}. \end{cases} \quad (10.10)$$

Damit gilt $\tau = \tilde{\chi}_{TOT}$, d.h. wir haben im Widerspruch zu Satz 10.1 gezeigt, daß das Totalitätsproblem TOT semi-entscheidbar und damit rekursiv aufzählbar ist.

▼ Also ist $\tilde{\chi}_{ID}$ nicht berechenbar, d.h. ID ist nicht semi-entscheidbar. ■

Da das Identitätsproblem ID ein Spezialfall des Äquivalenzproblems EQV ist, können wir ID auf EQV reduzieren und damit zeigen:

Korollar 10.9. *EQV ist nicht semi-entscheidbar.*

Beweis. Die Identitätsfunktion ist offensichtlich berechenbar, etwa durch ein Programm IDENT. Sei $\varrho : \mathbb{N} \rightarrow \mathbb{N}$ definiert durch $\varrho(i) := \pi^2(\uparrow \text{IDENT}, i)$. Dann ist ϱ offensichtlich berechenbar, und es gilt $i \in ID \Leftrightarrow \varrho(i) \in EQV$, d.h. $ID \preceq_{\varrho} EQV$. Mit Satz 10.8 und Korollar 10.5 ist dann auch EQV nicht semi-entscheidbar. ■

Bemerkung 10.10. Der Beweis von Korollar 10.9 illustriert die Nützlichkeit von Satz 10.4 in Form von Korollar 10.5: Es ist oft schwieriger, die (semi-)Unentscheidbarkeit eines allgemeinen Problems – wie etwa EQV – direkt zu beweisen, anstatt die (semi-)Unentscheidbarkeit eines Spezialfalls des Problems – wie etwa ID – direkt nachzuweisen und dann eine berechenbare Reduktion des Spezialfalls auf das allgemeine Problem anzugeben.



Als Konsequenz von Korollar 10.9 ist es also unmöglich, das gewünschte Verifikationswerkzeug zu implementieren. Damit ist natürlich nicht ausgeschlossen, daß man die Äquivalenz von bestimmten Programmen beweisen (oder auch widerlegen) kann. Mit Korollar 10.9 ist es nur unmöglich, ein (korrektes) Verfahren anzugeben, mit dem die Äquivalenz für alle äquivalenten Programme bewiesen werden kann.

10.5. Der Satz von Rice

Wir betrachten noch einmal das Verifikationsproblem, das schon in Abschnitt 10.4 angesprochen wurde. Im allgemeinen ist man an *Eigenschaften* der Funktionen, die von einem Programm berechnet werden, interessiert. Für die Prozedur GGT aus Abbildung 2.4 möchte man beispielsweise wissen, ob $\llbracket \text{GGT} \rrbracket$ eine *idempotente* Funktion ist, d.h. $\llbracket \text{GGT} \rrbracket(x, x) = x$, ob $\llbracket \text{GGT} \rrbracket$ *assoziativ* ist, d.h. $\llbracket \text{GGT} \rrbracket(x, \llbracket \text{GGT} \rrbracket(y, z)) = \llbracket \text{GGT} \rrbracket(\llbracket \text{GGT} \rrbracket(x, y), z)$, oder ob $\llbracket \text{GGT} \rrbracket$ *kommutativ* ist, d.h. $\llbracket \text{GGT} \rrbracket(x, y) = \llbracket \text{GGT} \rrbracket(y, x)$.

Um beispielsweise die Assoziativität von $\llbracket \text{GGT} \rrbracket$ zu verifizieren, betrachten wir die Menge $\Phi_{ass} = \{\phi \in \llbracket \mathcal{P} \rrbracket \mid \phi \text{ ist assoziativ}\}$ und versuchen dann zu entscheiden, ob $\llbracket \text{GGT} \rrbracket \in \Phi_{ass}$ gilt. In Φ_{ass} sind alle berechenbaren arithmetischen assoziativen Funktionen enthalten, also etwa auch die Addition und die Multiplikation, aber z.B. nicht die (abgerundete) Division. Im Unterschied zum Äquivalenzproblem EQV aus Abschnitt 10.4, bei dem entschieden werden soll, ob zwei Programme die gleiche Funktion berechnen, fragen wir hier allgemein, ob die Funktion, die durch ein bestimmtes Programm berechnet wird, eine bestimmte Eigenschaft hat.

Natürlich stellt sich die Frage, wie man Mengen von Funktionen, wie z.B. Φ_{ass} , *repräsentiert*. Da wir nur berechenbare Funktionen betrachten, können wir anstatt

Φ_{ass} die Menge \mathcal{P}_{ass} aller Programme P mit $\llbracket P \rrbracket(x, \llbracket P \rrbracket(y, z)) = \llbracket P \rrbracket(\llbracket P \rrbracket(x, y), z)$ betrachten. Diese Menge kodieren wir dann, wodurch wir die Menge $\natural\Phi_{ass} := \natural\mathcal{P}_{ass} = \{i \in \mathbb{N} \mid \varphi_i \in \Phi_{ass}\}$ erhalten. Anstatt “ $\llbracket \text{GGT} \rrbracket \in \Phi_{ass}$?” fragen wir jetzt also, ob $\natural\text{GGT} \in \natural\Phi_{ass}$ gilt, denn $\llbracket \text{GGT} \rrbracket \in \Phi_{ass}$ gdw. $\natural\text{GGT} \in \natural\Phi_{ass}$.

Damit dieser Ansatz zur Programmverifikation gelingt, muß $\natural\Phi_{ass}$ *entscheidbar* sein. Dies ist jedoch nicht der Fall, wie folgender Satz zeigt:

Satz 10.11. (Satz von Rice) Sei $\emptyset \subsetneq \Phi \subsetneq \llbracket \mathcal{P} \rrbracket$ und sei $\natural\Phi = \{i \in \mathbb{N} \mid \varphi_i \in \Phi\}$. Dann ist $\natural\Phi$ nicht entscheidbar.

Beweis. Sei $\theta : \mathbb{N} \mapsto \mathbb{N}$ mit $\theta \in \llbracket \mathcal{P} \rrbracket$ und $\theta \neq \omega$ beliebig gewählt, sei THETA eine \mathcal{P} -Prozedur mit $\llbracket \text{THETA} \rrbracket = \theta$ und sei die \mathcal{P} -Prozedur PROC gegeben durch

```

procedure PROC(i, j) <=
begin var res;
  if SEMI_DECIDE_S(i)
    then res := THETA(j)
    else res := CYCLE1(j)
  end_if;
return(res)
end

```

(10.11)

wobei SEMI_DECIDE_S mit $\llbracket \text{SEMI_DECIDE_S} \rrbracket = \tilde{\chi}_S$ ein semi-Entscheidungsverfahren für das Selbstanwendungsproblem S ist, vgl. Korollar 7.12. Mit dem s-m-n-Theorem 10.6 gilt dann

$$\varphi_{s_1^1(\natural\text{PROC}, i)}(j) = \varphi_{\natural\text{PROC}}(i, j) = \llbracket \text{PROC} \rrbracket(i, j) \quad (10.12)$$

und damit ist $\varrho : \mathbb{N} \rightarrow \mathbb{N}$, definiert durch

$$\varrho(i) := s_1^1(\natural\text{PROC}, i), \quad (10.13)$$

total und berechenbar. Weiter gilt mit (10.11)

$$\llbracket \text{PROC} \rrbracket(i, j) = \begin{cases} \theta(j) & , \text{ falls } i \in S \\ \omega(j) & , \text{ falls } i \notin S \end{cases} \quad (10.14)$$

und für $i \in S$ damit dann $\llbracket \text{PROC} \rrbracket(i, j) = \theta(j)$ für alle $j \in \mathbb{N}$, also

$$i \in S \Rightarrow \varphi_{s_1^1(\natural\text{PROC}, i)} = \theta. \quad (10.15)$$

mit (10.12). Für $i \notin S$ gilt $\llbracket \text{PROC} \rrbracket(i, j) = \omega(j)$ für alle $j \in \mathbb{N}$, und damit

$$i \notin S \Rightarrow \varphi_{s_1^1(\natural\text{PROC}, i)} = \omega. \quad (10.16)$$

Angenommen, es gilt $\omega \notin \Phi$. Mit $\Phi \neq \emptyset$ gilt dann o.B.d.A. $\theta \in \Phi$,⁴ und damit

$$\begin{aligned} i \in S &\Leftrightarrow \varphi_{s_1^1(\mathfrak{h}\text{PROC}, i)} \in \Phi && , \text{ mit (10.15) und (10.16),} \\ &\Leftrightarrow s_1^1(\mathfrak{h}\text{PROC}, i) \in \mathfrak{h}\Phi && , \text{ mit Definition von } \mathfrak{h}\Phi, \\ &\Leftrightarrow \varrho(i) \in \mathfrak{h}\Phi && , \text{ mit (10.13),} \end{aligned}$$

d.h. $S \preceq_{\varrho} \mathfrak{h}\Phi$. Mit der Unentscheidbarkeit von S , vgl. Korollar 7.3, folgt dann mit Korollar 10.5, daß $\mathfrak{h}\Phi$ nicht entscheidbar ist.

Andernfalls gilt $\omega \in \Phi$ und mit $\Phi \subsetneq \llbracket \mathcal{P} \rrbracket$ dann o.B.d.A. $\theta \notin \Phi$.⁵ Man erhält

$$\begin{aligned} i \notin S &\Leftrightarrow \varphi_{s_1^1(\mathfrak{h}\text{PROC}, i)} \in \Phi && , \text{ mit (10.16) und (10.15),} \\ &\Leftrightarrow s_1^1(\mathfrak{h}\text{PROC}, i) \in \mathfrak{h}\Phi && , \text{ mit Definition von } \mathfrak{h}\Phi, \\ &\Leftrightarrow \varrho(i) \in \mathfrak{h}\Phi && , \text{ mit (10.13),} \end{aligned}$$

d.h. $\overline{S} \preceq_{\varrho} \mathfrak{h}\Phi$. Mit der Unentscheidbarkeit von \overline{S} , vgl. Korollar 8.9, folgt dann ebenfalls, mit Korollar 10.5, daß $\mathfrak{h}\Phi$ nicht entscheidbar ist. ■

Der Satz von Rice zeigt also, daß es unmöglich ist zu entscheiden, ob die von einem Programm berechnete Funktion bestimmte Eigenschaften besitzt. Ausgenommen sind dabei die “trivialen” Eigenschaften, die für kein Programm – Fall $\Phi = \emptyset$ – und für alle Programme – Fall $\Phi = \llbracket \mathcal{P} \rrbracket$ – gelten, denn dann gilt $\mathfrak{h}\Phi = \emptyset$ bzw. $\mathfrak{h}\Phi = \mathfrak{h}\mathcal{P}$, und damit ist $\mathfrak{h}\Phi$ trivialerweise bzw. mittels $\mathfrak{h}?$ (vgl. Satz 4.12) entscheidbar.

Mit Satz 10.11 ist natürlich nicht ausgeschlossen, daß man für gewisse Funktionen irgendwelche Eigenschaften nachweisen kann, und so etwa der Nachweis von $\llbracket \text{GGT} \rrbracket \in \Phi_{\text{ass}}$ trotzdem gelingt. Mit dem Satz von Rice ist es dagegen unmöglich, ein terminierendes (und korrektes) Verfahren anzugeben, mit dem beispielsweise $\llbracket \text{PROG} \rrbracket \in \Phi_{\text{ass}}$ für *jedes* Programm $\text{PROG} \in \mathcal{P}$ bewiesen oder widerlegt werden kann.

10.6. Weitere Probleme

Wir geben hier ohne Beweis (einige) weitere Ergebnisse aus der Berechenbarkeitstheorie bzw. der mathematischen Logik an – weiteres dazu findet man in den Lehrbüchern des Literaturverzeichnisses:

⁴ Mit $\omega \notin \Phi$ und $\Phi \neq \emptyset$ enthält Φ mindestens eine Funktion $\phi : \mathbb{N} \mapsto \mathbb{N}$ mit $\phi \neq \omega$. Da $\theta : \mathbb{N} \mapsto \mathbb{N}$ mit $\theta \in \llbracket \mathcal{P} \rrbracket$ und $\theta \neq \omega$ *beliebig* gewählt war, gilt alles, was bislang über θ gezeigt wurde, auch für ϕ . Wir dürfen daher tatsächlich “o.B.d.A.” $\theta \in \Phi$ annehmen.

⁵ Mit $\omega \in \Phi$ und $\Phi \subsetneq \llbracket \mathcal{P} \rrbracket$ enthält $\llbracket \mathcal{P} \rrbracket$ mindestens eine Funktion $\phi : \mathbb{N} \mapsto \mathbb{N}$ mit $\phi \neq \omega$ und $\phi \notin \Phi$. Da $\theta : \mathbb{N} \mapsto \mathbb{N}$ mit $\theta \in \llbracket \mathcal{P} \rrbracket$ und $\theta \neq \omega$ *beliebig* gewählt war, gilt alles, was bislang über θ gezeigt wurde, auch für ϕ . Wir dürfen daher tatsächlich “o.B.d.A.” $\theta \notin \Phi$ annehmen.

1. Die Menge TAU der Tautologien, also der wahren aussagenlogischen Formeln, ist entscheidbar (zum Beweis implementiere man die Wahrheitstafel-Methode).
2. Die Menge AG der allgemeingültigen Formeln der Prädikatenlogik erster Stufe ist semi-entscheidbar, vgl. Anmerkung 9.5. Konsequenz ist, daß man einen Automatischen Beweiser implementieren kann, der für jede allgemeingültige Formel einen Beweis berechnet.
3. Die Menge \overline{AG} der nicht-allgemeingültigen Formeln der Prädikatenlogik erster Stufe ist nicht semi-entscheidbar. Konsequenz ist, daß der Automatische Beweiser aus 2. für unendlich viele (nicht-allgemeingültige) Formeln beim Beweisversuch in eine Endlosschleife gerät.
4. Aus 3. folgt (mit Satz 8.7): Die Menge AG der allgemeingültigen Formeln der Prädikatenlogik erster Stufe ist unentscheidbar.
5. Die Menge Pr aller wahren Formeln, die über die Nachfolgerfunktion und die Addition in \mathbb{N} Aussagen treffen, ist entscheidbar (Presburger Arithmetik).
6. Die Menge Ar der wahren Formeln der Arithmetik, d.h. alle wahren Formeln, die über die Nachfolgerfunktion, die Addition und die Multiplikation (und eventuell weitere arithmetische Funktion) in \mathbb{N} Aussagen treffen, ist nicht semi-entscheidbar (Erster Gödelscher Unvollständigkeitssatz). Konsequenz ist, daß jedes korrekte Beweisverfahren für Aussagen der Arithmetik unvollständig ist, d.h. für gewisse wahre Formeln keinen Beweis berechnen kann.
7. Aus 6. folgt (mit Satz 8.7): Die Menge Ar der wahren Formeln der Arithmetik ist unentscheidbar.
8. Die Menge \overline{Ar} der falschen Formeln der Arithmetik ist nicht semi-entscheidbar (Erster Gödelscher Unvollständigkeitssatz). Konsequenz ist, daß jedes korrekte Widerlegungsverfahren für Aussagen der Arithmetik unvollständig ist, d.h. für gewisse falsche Formeln kein Gegenbeispiel berechnen kann.
9. Die Menge Ar^\forall der wahren universellen Formeln der Arithmetik, d.h. der wahren Formeln der Form $\forall \dots \forall \phi$ mit “ ϕ ist quantorfrei”, ist nicht semi-entscheidbar.
10. Die Menge Ar^\exists der wahren existentiellen Formeln der Arithmetik, d.h. der wahren Formeln der Form $\exists \dots \exists \phi$ mit “ ϕ ist quantorfrei”, ist semi-entscheidbar (denn $\mathbb{N} \times \dots \times \mathbb{N}$ ist rekursiv aufzählbar, so daß man alle Be-

gungen für die existentiell quantifizierten Variablen systematisch erzeugen und ϕ damit in berechenbarer Weise überprüfen kann).

11. Die Menge \mathcal{E}_P aller wahren Formeln, die Aussagen über Eigenschaften eines Programms P einer berechnungsvollständigen Programmiersprache treffen, ist nicht semi-entscheidbar (folgt aus 6.). Konsequenz ist, daß jedes korrekte Verifikationsverfahren für Programme einer berechnungsvollständigen Programmiersprache unvollständig ist, d.h. für gewisse wahre Aussagen keinen Beweis berechnen kann.
12. Aus 11. folgt (mit Satz 8.7): Die Menge \mathcal{E}_P aller wahren Formeln, die Aussagen über Eigenschaften eines Programms P einer berechnungsvollständigen Programmiersprache treffen, ist unentscheidbar.
13. Die Menge $\overline{\mathcal{E}_P}$ aller falschen Formeln, die Aussagen über Eigenschaften eines Programms P einer berechnungsvollständigen Programmiersprache treffen, ist nicht semi-entscheidbar (folgt aus 8.). Konsequenz ist, daß jedes korrekte Widerlegungsverfahren für Programmaussagen unvollständig ist, d.h. für gewisse falsche Aussagen kein Gegenbeispiel berechnen kann.
14. Die Menge \mathcal{E}_P^\forall der wahren universellen Formeln über Eigenschaften eines Programms P einer berechnungsvollständigen Programmiersprache ist nicht semi-entscheidbar (folgt aus 9.).
15. Die Menge \mathcal{E}_P^\exists der wahren existentiellen Formeln über Eigenschaften eines Programms P einer berechnungsvollständigen Programmiersprache ist semi-entscheidbar (folgt aus 10.). Konsequenz ist, daß man ein vollständiges Testwerkzeug für \mathcal{E}_P^\forall implementieren kann, d.h. ein Testwerkzeug, das jede falsche – als universelle Formel formulierte – Behauptung über ein Programm widerlegen kann. Mit 14. gerät solch ein Testwerkzeug allerdings für unendlich viele (wahre, universell formulierte) Programmaussagen in eine Endlosschleife.

11. PRIMITIV-REKURSIVE UND μ -REKURSIVE FUNKTIONEN

In diesem Abschnitt stellen wir ein weiteres Rechenmodell vor, die auf *S. C. Kleene* zurückgehenden μ -rekursiven Funktionen. Für den Informatiker ist dieses Rechenmodell auch deshalb interessant, da die Definition μ -rekursiver Funktionen in gewisser Weise als eine Vorform der *applikativen Programmierung*, also der *funktionalen Programmierung*, aufgefaßt werden kann.

μ -rekursive Funktionen werden mittels zweier Definitionsprinzipien gebildet: Zunächst wird die Klasse der *primitiv-rekursiven Funktionen* eingeführt. Man kann zeigen, daß diese Klasse genau die arithmetischen Funktionen enthält, die man mittels sogenannter *loop*-Programme erhält. Dies sind \mathcal{P} -Programme, die anstatt *while*-Anweisungen nur Schleifenanweisungen in der Art von *for*-Schleifen enthalten, so wie sie aus einigen Programmiersprachen (beispielsweise FORTRAN, PASCAL oder ADA) bekannt sind. Mit dieser Einschränkung ist offensichtlich, daß nicht jede berechenbare Funktion primitiv-rekursiv sein kann, denn jedes *loop*-Programm muß ja terminieren, und somit können nicht-totale berechenbare Funktionen nicht primitiv-rekursiv sein.

Um dieser Einschränkung zu begegnen, wird der sogenannte (unbeschränkte) μ -Operator eingeführt, durch den beliebige – also auch *nicht-terminierende* – Schleifen modelliert werden können. Man erhält mittels des μ -Operators und der primitiv-rekursiven Funktionen die Klasse der μ -rekursiven Funktionen, von der wir zeigen, daß sie identisch der Klasse der \mathcal{P} -berechenbaren Funktionen ist.

11.1. Primitiv-rekursive Funktionen

Das Definitionsprinzip für primitiv-rekursive Funktionen ist vergleichbar dem Definitionsprinzip von Prozeduren in funktionalen Programmiersprachen. Ausgangspunkt sind die sogenannten *Grundfunktionen*, die man sich als bereits implementiert vorstellen kann. Wie schon bei Definition der Programmiersprache \mathcal{P} werden nur solche Grundfunktionen vorausgesetzt, die für die Definition des Rechenmodells unabdingbar sind.

Definition 11.1. (Grundfunktionen)

Als Grundfunktionen werden bezeichnet

1. die 0-stellige Funktion $Z : \rightarrow \mathbb{N}$ mit $Z := 0$,
2. die 1-stellige Funktion $S : \mathbb{N} \rightarrow \mathbb{N}$ mit $S(x) := x + 1$, und die
3. die k -stelligen Projektionsfunktionen $P_i^k : \mathbb{N}^k \rightarrow \mathbb{N}$ mit $P_i^k(x_1, \dots, x_k) := x_i$ für jedes $k > 0$ und alle i mit $1 \leq i \leq k$. ■

Mit bereits definierten Funktionen können weitere Funktionen definiert werden. Beispielsweise können wir mittels bereits definierter Funktionen *half* und *times* für abgerundete Division durch 2 und Multiplikation eine Funktion *sum-up* durch $sum-up(x) := half(times(x, S(x)))$ definieren. Man nennt dieses Definitionsprinzip "Definition durch Einsetzung".

Definition 11.2. (Definition durch Einsetzung)

Seien $f : \mathbb{N}^k \mapsto \mathbb{N}$, $g : \mathbb{N}^m \mapsto \mathbb{N}$ und $h_1, \dots, h_m : \mathbb{N}^k \mapsto \mathbb{N}$ Funktionen mit $k, m \geq 0$ sowie

$$f(\vec{x}) := g(h_1(\vec{x}), \dots, h_m(\vec{x})) .$$

Dann ist die Funktion f definiert durch Einsetzung (mittels der Funktionen g und h_1, \dots, h_m).¹ ■

Allein mit Grundfunktionen und durch Einsetzung können noch keine interessanten Funktionen, wie etwa Addition, Multiplikation u.s.w., definiert werden. Wir erlauben daher zusätzlich die Definition von Funktionen durch ein *Rekursionsprinzip*:

Definition 11.3. (Definition durch primitive Rekursion)

Seien $f : \mathbb{N}^{k+1} \mapsto \mathbb{N}$, $g : \mathbb{N}^k \mapsto \mathbb{N}$ und $h : \mathbb{N}^{k+2} \mapsto \mathbb{N}$ Funktionen mit $k \geq 0$ sowie

1. $f(0, \vec{x}) := g(\vec{x})$ und
2. $f(x + 1, \vec{x}) := h(f(x, \vec{x}), x, \vec{x})$.

Dann ist die Funktion f definiert durch primitive Rekursion (mittels der Funktionen g und h). ■

¹ \vec{x} steht als Abkürzung für ein k -Tupel x_1, \dots, x_k von Variablensymbolen.

$$\begin{array}{ll}
\Rightarrow & plus(1, 2) \\
\Rightarrow & S_1^3(plus(0, 2), 0, 2) \quad , \text{ mit (iii)} \\
\Rightarrow & S(P_1^3(plus(0, 2), 0, 2)) \quad , \text{ mit (i)} \\
\Rightarrow & P_1^3(plus(0, 2), 0, 2) + 1 \quad , \text{ mit Definition 11.1(2.)} \\
\Rightarrow & plus(0, 2) + 1 \quad , \text{ mit Definition 11.1(3.)} \\
\Rightarrow & P_1^1(2) + 1 \quad , \text{ mit (ii)} \\
\Rightarrow & 3 \quad , \text{ mit Definition 11.1(3.)}
\end{array}$$

Abbildung 11.1: Berechnung von $plus(1, 2)$ mit primitiv-rekursiven Funktionen

Mit Grundfunktionen, Definition durch Einsetzung und durch primitive Rekursion erhalten wir eine einfache funktionale Programmiersprache. Funktionen, die durch Programme dieser Programmiersprache berechnet werden können, werden *primitiv-rekursiv* genannt:

Definition 11.4. (Primitiv-rekursive Funktionen)

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ mit $k \geq 0$ heißt primitiv-rekursiv, gdw. gilt:

1. f ist eine Grundfunktion, oder
2. f ist definiert durch Einsetzung mittels primitiv-rekursiver Funktionen, oder
3. f ist definiert durch primitive Rekursion mittels primitiv-rekursiver Funktionen. ■

Beispielsweise können wir die Addition natürlicher Zahlen als primitiv-rekursive Funktion “programmieren”. Wir definieren dazu

- (i) $S_1^3(x_1, x_2, x_3) := S(P_1^3(x_1, x_2, x_3))$,
- (ii) $plus(0, y) := P_1^1(y)$, und
- (iii) $plus(x + 1, y) := S_1^3(plus(x, y), x, y)$.

Nach den Definitionen 11.4(1.) und 11.1 sind S , P_1^1 und P_1^3 primitiv-rekursive Funktionen. Die Funktion S_1^3 entsteht durch Einsetzung nach den Forderungen von Definition 11.2, und ist damit nach Definition 11.4(2.) ebenfalls primitiv-rekursiv. Damit genügen die Definitionen (ii) und (iii) den Forderungen (1.) und (2.) von Definition 11.4, also ist $plus$ eine primitiv-rekursive Funktion.

Durch Anwenden der definierenden Gleichungen *links := rechts* von links nach rechts können wir Ausdrücke über primitiv-rekursiven Funktionen ausrechnen. Abbildung 11.1 zeigt als Beispiel die Berechnung von $plus(1, 2)$.

Eine Funktion ist primitiv-rekursiv, wenn sie nach Definition 11.4 gebildet werden kann. Anders gesagt, man muß unterscheiden zwischen der *Definition* einer Funktion und der *Funktion* selbst. Beispielsweise ist die durch $plus'(0, y) := P_1^1(y)$ und $plus'(x + 1, y) := plus'(x, S(y))$ definierte *Funktion* primitiv-rekursiv (denn man zeigt leicht $plus'(n, m) = plus(n, m)$ für alle $n, m \in \mathbb{N}$), $plus'$ ist jedoch *nicht* primitiv-rekursiv *definiert*, da die Forderung (2.) von Definition 11.4 nicht erfüllt ist.²

11.2. Definitionsprinzipien für primitiv-rekursive Funktionen

Mit Definition 11.4 wird ein starres Schema zur Definition primitiv-rekursiver Funktionen angegeben. Konsequenz ist, daß die Programmierung umständlich und unübersichtlich ist, wie schon anhand der Definition von $plus$ ersichtlich. Als Abhilfe gehen wir so vor, wie schon zuvor bei der Programmiersprache \mathcal{P} : Wir führen abkürzende Schreibweisen ein, die jedoch immer auf die Originalforderungen zurückgeführt werden können. Nachfolgend verwenden wir dann sowohl die Originalsprache als auch die komfortablere, erweiterte Sprache, je nachdem, wie es für den jeweiligen Zweck am bequemsten ist.

Lemma 11.5. (Primitiv-rekursive Hilfsfunktionen)

1. Die Funktionen $C_n^k : \mathbb{N}^k \rightarrow \mathbb{N}$ mit $C_n^k(x_1, \dots, x_k) = n$ sind für jedes $k \in \mathbb{N}$ und jedes $n \in \mathbb{N}$ primitiv-rekursiv.
2. Die Funktion $pred : \mathbb{N} \rightarrow \mathbb{N}$, definiert durch $pred(0) = 0$ und $pred(x+1) := x$ ist primitiv-rekursiv.
3. Die Funktion $if : \mathbb{N}^3 \rightarrow \mathbb{N}$ mit $if(0, y_1, y_2) = y_2$ und $if(x + 1, y_1, y_2) := y_1$ ist primitiv-rekursiv.
4. Die Paar-Funktion π^2 und ihre Projektionsfunktionen π_1^2 und π_2^2 sind primitiv-rekursiv.

Beweis. Übung. ■

² Solche Unterscheidungen zwischen Syntax und Semantik sind auch von anderswo bekannt. Beispielsweise ist eine *Grammatik* mit den beiden Regeln $S \rightarrow AA$ und $A \rightarrow a$ nicht regulär, die dadurch definierte *Sprache* $\{aa\}$ dagegen schon.

Lemma 11.6. (Definition durch allgemeine Einsetzung)

Seien $g_1 : \mathbb{N}^{m_1} \mapsto \mathbb{N}, \dots, g_j : \mathbb{N}^{m_j} \mapsto \mathbb{N}$, sei t ein Ausdruck, in dem neben den Grundfunktionen nur Funktionssymbole aus $\{g_1, \dots, g_j\}$, nur Variablensymbole aus $\{x_1, \dots, x_k\}$ und natürliche Zahlen n vorkommen, und sei die Funktion $f : \mathbb{N}^k \mapsto \mathbb{N}$ mit $k > 0$ definiert durch

$$f(x_1, \dots, x_k) := t .$$

Dann kann f durch Grundfunktionen und Einsetzung definiert werden.

Beweis. Wir zeigen die Behauptung durch strukturelle Induktion über t .

Induktionsanfang “ $t = n \in \mathbb{N}$ ”: Durch Induktion über n zeigt man leicht $S_n = n$ für jede Funktion $S_n : \rightarrow \mathbb{N}$ definiert durch $S_0 := Z$ und $S_{n+1} := S(S_n)$. Jede Funktion S_n genügt damit der Forderung von Definition 11.2, und mit $f(x_1, \dots, x_k) = S_n$ genügt f ebenfalls dieser Forderung.

Induktionsanfang “ $t = x_i$ ”: Dann gilt $f(x_1, \dots, x_k) := P_i^k(x_1, \dots, x_k)$, und folglich genügt f der Forderung von Definition 11.2.

Induktionsschritt “ $t = g(t_1, \dots, t_m)$ mit $g \in \{g_1, \dots, g_j\}$ ”: Wir definieren für jedes $i \in \{1, \dots, m\}$ die Funktionen $f_i : \mathbb{N}^k \mapsto \mathbb{N}$ durch $f_i(x_1, \dots, x_k) := t_i$, von denen wir nach Induktionshypothese annehmen dürfen, daß sie alternativ den Forderungen von Definition 11.2 genügend definiert werden können. Mit $f(x_1, \dots, x_k) := g(f_1(x_1, \dots, x_k), \dots, f_m(x_1, \dots, x_k))$ genügt dann auch f der Forderung von Definition 11.2. ■

Satz 11.7. (Definition durch strukturelle Rekursion)

Seien $g_1 : \mathbb{N}^{m_1} \mapsto \mathbb{N}, \dots, g_j : \mathbb{N}^{m_j} \mapsto \mathbb{N}$ Funktionen, sei $f : \mathbb{N}^{k+1} \mapsto \mathbb{N}$ eine Funktion, und seien t_1 und t_2 Ausdrücke, in denen neben den Grundfunktionen nur Funktionssymbole aus $\{f, g_1, \dots, g_j\}$, das Variablensymbol x , die Variablensymbole aus \vec{x} und natürliche Zahlen n vorkommen. Sei weiter die Funktion f definiert durch:

1. $f(0, \vec{x}) := t_1$, so daß f und x nicht in t_1 vorkommen, und
2. $f(x + 1, \vec{x}) := t_2$, so daß $t' = f(x, \vec{x})$ für jeden Teilausdruck $t' = f(\dots)$ von t_2 gilt.

Dann kann f durch Grundfunktionen, Einsetzung und primitive Rekursion definiert werden.

Beweis. Mit Lemma 11.6 gibt es eine durch Grundfunktionen und Einsetzung definierte Funktion $g : \mathbb{N}^k \mapsto \mathbb{N}$ mit (i) $g(\vec{x}) = t_1$.

Sei jetzt z eine Variable und sei $C[z]$ ein Ausdruck mit $t_2 = C[z/f(x, \vec{x})]$.³ Dann ist $C[z]$ ein Ausdruck, in dem außer der Variablen z höchstens die Variablen

³ $C[z/t]$ entsteht aus $C[z]$, indem jedes Vorkommen von z in $C[z]$ durch t ersetzt wird.

x, \vec{x} , Funktionssymbole aus $\{g_1, \dots, g_j\}$ sowie natürliche Zahlen n vorkommen. Nach Lemma 11.6 kann somit eine Funktion $h : \mathbb{N}^{k+2} \mapsto \mathbb{N}$ mit $h(z, x, \vec{x}) = C[z]$ durch Einsetzung definiert werden. Es gilt (ii) $f(x+1, \vec{x}) = t_2 = C[z/f(x, \vec{x})] = h(f(x, \vec{x}), x, \vec{x})$, und mit (i) und (ii) kann f dann den Forderungen von Definition 11.4 genügend definiert werden. ■

Satz 11.7 erlaubt eine wesentlich einfachere Definition von primitiv-rekursiven Funktionen und legt die Vermutung nahe, daß die Funktionen, mit denen wir in der Praxis umgehen, überwiegend primitiv-rekursiv sind. So genügen etwa die Definitionen der Multiplikation und der Fakultätsfunktion in nachfolgendem Beispiel nicht dem starren Schema von Definition 11.4, die so definierten Funktionen sind jedoch nach Satz 11.7 primitiv-rekursiv.

Beispiel 11.8. (Definition durch strukturelle Rekursion)

1. $times(0, y) := 0$ und $times(x+1, y) := plus(y, times(x, y))$.
2. $fact(0) := 1$ und $fact(x+1) := plus(times(fact(x), x), fact(x))$. ■

Mit Satz 11.7 kann man ein weiteres, mächtiges Definitionsprinzip für primitiv-rekursive Funktionen angeben, nämlich die Definition von primitiv-rekursiven Funktionen durch gegenseitige Rekursion. Dieses Definitionsprinzip wird im folgenden noch nützlich sein.

Satz 11.9. (Gegenseitige Rekursion)

Sei $k \geq 0$ und seien $g_1, \dots, g_j : \mathbb{N}^k \mapsto \mathbb{N}$ sowie $h_1, \dots, h_j : \mathbb{N}^{j+1+k} \mapsto \mathbb{N}$ primitiv-rekursive Funktionen. Dann sind die Funktionen $f_1, \dots, f_j : \mathbb{N}^{k+1} \mapsto \mathbb{N}$, gegeben durch

1. $f_i(0, \vec{x}) := g_i(\vec{x})$ und
2. $f_i(x+1, \vec{x}) := h_i(f_1(x, \vec{x}), \dots, f_j(x, \vec{x}), x, \vec{x})$,

für alle $i \in \{1, \dots, j\}$ primitiv-rekursiv.

Beweis. Sei $F : \mathbb{N}^{k+1} \mapsto \mathbb{N}$ definiert durch

$$F(0, \vec{x}) := \pi^j(g_1(\vec{x}), \dots, g_j(\vec{x})) \tag{11.1}$$

$$F(x+1, \vec{x}) := \pi^j(h_1(\pi_1^j(F(x, \vec{x})), \dots, \pi_j^j(F(x, \vec{x})), x, \vec{x}), \dots, h_j(\pi_1^j(F(x, \vec{x})), \dots, \pi_j^j(F(x, \vec{x})), x, \vec{x})). \tag{11.2}$$

Wir zeigen

$$f_i(x, \vec{x}) = \pi_i^j(F(x, \vec{x})) \quad (11.3)$$

für alle $i \in \{1, \dots, j\}$ durch Induktion über x : Im Induktionsanfang gilt $f_i(0, \vec{x}) = g_i(\vec{x}) = \pi_i^j(\pi^j(g_1(\vec{x}), \dots, g_j(\vec{x}))) = \pi_i^j(F(0, \vec{x}))$ mit Definition von f_i und 11.1. Im Induktionsschritt erhält man

$$\begin{aligned} f_i(x+1, \vec{x}) &= h_i(f_1(x, \vec{x}), \dots, f_j(x, \vec{x}), x, \vec{x}) && \text{, mit Definition von } f_i \\ &= h_i(\pi_1^j(F(x, \vec{x})), \dots, \pi_j^j(F(x, \vec{x})), x, \vec{x}) && \text{, mit Induktionshypothese} \\ &= \pi_i^j(F(x+1, \vec{x})). && \text{, mit 11.2.} \end{aligned}$$

Mit 11.3 ist jede Funktion f_i primitiv-rekursiv, denn F ist mit Satz 11.7 primitiv-rekursiv. ■

Beispiel 11.10. (Gegenseitige Rekursion) Die Funktionen $even : \mathbb{N} \mapsto \mathbb{N}$ und $odd : \mathbb{N} \mapsto \mathbb{N}$, definiert durch

1. $even(0) := C_1^0 = 1$,
2. $even(x+1) := P_2^2(even(x), odd(x)) = odd(x)$,
3. $odd(0) := C_0^0 = 0$, und
4. $odd(x+1) := P_1^2(even(x), odd(x)) = even(x)$

sind mit Satz 11.9 primitiv-rekursiv. Für die Begriffe aus Satz 11.9 ergibt sich

1. $F(0) := \pi^2(C_1^0, C_0^0) = \pi^2(1, 0)$ sowie
2. $F(x+1) := \pi^2(P_2^2(\pi_1^2(F(x)), \pi_2^2(F(x))), P_1^2(\pi_1^2(F(x)), \pi_2^2(F(x))))$
 $= \pi^2(\pi_2^2(F(x)), \pi_1^2(F(x))),$

und mit 11.3 dann $even(x) = \pi_1^2(F(x))$ sowie $odd(x) = \pi_2^2(F(x))$. Für F erhält man z.B.

- $F(1) = \pi^2(\pi_2^2(F(0)), \pi_1^2(F(0))) = \pi^2(0, 1)$,
- $F(2) = \pi^2(\pi_2^2(F(1)), \pi_1^2(F(1))) = \pi^2(1, 0)$,
- $F(3) = \pi^2(\pi_2^2(F(2)), \pi_1^2(F(2))) = \pi^2(0, 1)$,
- ... ■

11.3. Werteverlaufsrekursion

Bislang steht uns nur die strukturelle Rekursion zur Definition von Funktionen zur Verfügung, denn bei rekursiver Definition einer Funktion f ist im Rekursionsfall nur der *direkte* Vorgänger x des Rekursionsparameters $x + 1$ im rekursiven Aufruf erlaubt. Beispielsweise ist so nicht ersichtlich, ob die *Fibonacci*-Funktion $fib : \mathbb{N} \rightarrow \mathbb{N}$, definiert durch

$$\begin{aligned} (1.) \quad & fib(0) := 1 \text{ und} \\ (2.) \quad & fib(x + 1) := if(x, plus(fib(x), fib(pred(x))), 1) \end{aligned} \quad (11.4)$$

primitiv-rekursiv ist, denn Definitionsgleichung (11.4-2.) genügt (wegen des rekursiven Aufrufs $fib(pred(x))$) weder Forderung (2.) von Definition 11.3 noch Forderung (2.) von Satz 11.7.

In gängigen funktionalen Programmiersprachen dürfen Funktionen sogar *Funktionsparameter* besitzen. Beispielsweise kann man dort eine Funktion $SUM : \mathbb{N} \times (\mathbb{N} \mapsto \mathbb{N}) \mapsto \mathbb{N}$ (Eingabe von SUM ist also eine natürliche Zahl sowie eine *Funktion*, die natürliche Zahlen in natürliche Zahlen abbildet) als

$$\begin{aligned} (1.) \quad & SUM(0, F) := F(0) \text{ und} \\ (2.) \quad & SUM(x + 1, F) := F(x + 1) + SUM(x, F) \end{aligned} \quad (11.5)$$

angeben, und damit dann durch

$$\begin{aligned} (1.) \quad & exp2(0) := 1 \text{ und} \\ (2.) \quad & exp2(x + 1) := S(SUM(x, exp2)) \end{aligned} \quad (11.6)$$

die Exponentiation zur Basis 2 definieren (es gilt $exp2(n) = 2^n$).⁴ Hier ist noch weniger ersichtlich, ob mit (11.6-1.) und (11.6-2.) eine primitiv-rekursive Funktion definiert wird.

Um dies zu untersuchen, betrachten wir die Funktion SUM eingehender: Es gilt offensichtlich $SUM(x, F) = F(x) + F(x - 1) + \dots + F(0)$. Die Summanden können wir auch in einer Liste $\langle F(x), F(x - 1), \dots, F(0) \rangle$ abspeichern, um dann die Summe der Listenelemente zu bilden. Für $exp2(x + 1)$ erhält man so die Liste $\langle exp2(x), exp2(x - 1), \dots, exp2(0) \rangle$. Diese Liste beschreibt den *Werteverlauf* bei Berechnung von $exp2(x + 1)$.

Für eine Funktion $exp2^* : \mathbb{N} \mapsto \mathbb{N}$, mit der man die π^2 -Kodierung der Werteverlaufsliste von $exp2$ erhält, d.h.

$$exp2^*(x + 1) := \pi^2(exp2(x), \pi^2(exp2(x - 1), \dots, \pi^2(exp2(0), 0) \dots)) \quad (11.7)$$

⁴ Auch die Paar-Funktion kann durch $pair^2(x, y) := minus(SUM(plus(x, y), S), pred(y))$ mittels SUM definiert werden, denn man zeigt leicht $pair^2(x, y) = \pi^2(x, y)$.

gilt

$$\text{exp2}(x) = \text{S}(\text{sum}(x, \text{exp2}^*(x))) \quad (11.8)$$

denn man erhält $\text{exp2}(x)$ ja durch Aufsummieren der Elemente der (dekodierten) Liste $\text{exp2}^*(x)$. Hierbei verwenden wir die Hilfsfunktionen ntl , nth und sum , definiert durch⁵

$$\begin{aligned} (1.) \quad \text{ntl}(0, y) &:= y & (4.) \quad \text{sum}(0, y) &:= 0 \\ (2.) \quad \text{ntl}(x+1, y) &:= \pi_2^2(\text{ntl}(x, y)) & (5.) \quad \text{sum}(x+1, y) &:= \\ (3.) \quad \text{nth}(x, y) &:= \pi_1^2(\text{ntl}(x, y)) & & \text{plus}(\text{nth}(x, y), \text{sum}(x, y)) . \end{aligned} \quad (11.9)$$

Mit Satz 11.7 sind ntl , nth und sum primitiv-rekursive Funktionen. Mit (11.8) kann exp2 allein durch (i) Einsetzung mittels (ii) primitiv-rekursiver Funktionen, nämlich **if** und sum , und (iii) der Werteverlaufsfunktion exp2^* definiert werden. Wir zeigen, daß allein aus (i) – (iii) folgt, daß exp2 primitiv-rekursiv ist.

Definition 11.11. (Werteverlaufsfunktion)

Für eine Funktion $f : \mathbb{N}^{k+1} \mapsto \mathbb{N}$ ist die Werteverlaufsfunktion $f^* : \mathbb{N}^{k+1} \mapsto \mathbb{N}$ definiert durch

1. $f^*(0, \vec{x}) := 0$ und

2. $f^*(x+1, \vec{x}) := \pi^2(f(x, \vec{x}), f^*(x, \vec{x}))$. ■

Da mit der Paar-Funktion π^2 endliche Listen durch natürliche Zahlen kodiert werden (π^2 entspricht dabei dem Listenkonstruktor, mit π_1^2 erhält man das erste Element einer nicht-leeren Liste und mit π_2^2 die Liste ohne das erste Listenelement), repräsentiert $f^*(x+1, \vec{x})$ die π^2 -Kodierung der Liste $\langle f(x, \vec{x}), f(x-1, \vec{x}), \dots, f(0, \vec{x}) \rangle$.⁶ Wir zeigen, daß jede Funktion f (und ihre Werteverlaufsfunktion f^*) *primitiv-rekursiv* sein muß, falls f allein durch Einsetzung mittels einer primitiv-rekursiven Funktion h und ihrer Werteverlaufsfunktion f^* definiert werden kann:

⁵ Bei diesen Funktionen wird y als die π^2 -Kodierung einer linearen Liste aufgefaßt. Mit $\text{ntl}(x, y)$ erhält man dann die (Kodierung der) Liste, die aus der Dekodierung von y durch Elimination der ersten x Elemente entsteht. Mit $\text{nth}(x, y)$ erhält man das $x+1$. Listenelement (der Dekodierung) von y . Mit $\text{sum}(x, y)$ werden schließlich die ersten x Elemente der (dekodierten) Liste y aufsummiert.

⁶ Da die durch $f^*(x, \vec{x})$ kodierte Liste genau x Elemente enthält, die Länge dieser Liste also bekannt ist, muß die natürliche Zahl 0 nicht von der Kodierung 0 der leeren Liste unterschieden werden, vgl. Bemerkung 4.5. Beispielsweise stellt die Definition von sum in (11.9-4 und 11.9-5.) sicher, daß nicht versucht wird, das erste Element der leeren Liste zu bestimmen, falls $x \leq$ der Länge der aus Dekodierung von y gewonnenen Liste ist.

Satz 11.12. (Werteverlaufsrekursion, engl. *course-of-values recursion*)
 Sei $h : \mathbb{N}^{k+2} \mapsto \mathbb{N}$ primitiv-rekursiv und sei $f : \mathbb{N}^{k+1} \mapsto \mathbb{N}$ eine Funktion mit

$$f(x, \vec{x}) = h(f^*(x, \vec{x}), x, \vec{x}) . \quad (11.10)$$

Dann ist f primitiv-rekursiv.

Beweis. Für alle $x \in \mathbb{N}$, $\vec{x} \in \mathbb{N}^k$ gilt

$$f^*(0, \vec{x}) = 0 \quad (11.11)$$

mit Definition 11.11(1.) sowie

$$f^*(x+1, \vec{x}) = \pi^2(h(f^*(x, \vec{x}), x, \vec{x}), f^*(x, \vec{x})) \quad (11.12)$$

denn

$$\begin{aligned} & f^*(x+1, \vec{x}) \\ = & \pi^2(f(x, \vec{x}), f^*(x, \vec{x})) \quad , \text{ mit Definition 11.11(2.)} \\ = & \pi^2(h(f^*(x, \vec{x}), x, \vec{x}), f^*(x, \vec{x})) \quad , \text{ mit (11.10) .} \end{aligned}$$

Mit Satz 11.7 ist f^* wegen (11.11) und (11.12) primitiv-rekursiv, und folglich ist f mit (11.10) primitiv-rekursiv.⁷ ■

Mit Nachweis von (11.8) rechtfertigt Satz 11.12 also die Behauptung, daß *exp2* tatsächlich primitiv-rekursiv ist.⁸ Genauso folgt mit Satz 11.12 aus dem Nachweis von

$$fib(x) = \text{if}(x, \text{plus}(\pi_1^2(fib^*(x)), \pi_1^2(\pi_2^2(fib^*(x)))), 1) \quad (11.13)$$

daß die *Fibonacci*-Funktion primitiv-rekursiv ist.

11.4. Ein Rechenmodell für primitiv-rekursive Funktionen

Bei Definition der Programmiersprache \mathcal{P} wurde eine explizite Trennung zwischen *Syntax* und *Semantik* vorgenommen, d.h., zum einen wurde in Definition 2.1 festgelegt, welche formalsprachlichen Ausdrücke überhaupt als Programme aufgefaßt

⁷ Das ‘‘Gegenstück’’ zur *Werteverlaufsrekursion* ist die *Noethersche Induktion* mit der üblichen (transitiven) $<$ -Relation auf \mathbb{N} . Hier zeigt man eine Behauptung $\forall x:\mathbb{N}. P(x)$ durch Nachweis von $P(0)$ und $\forall x:\mathbb{N}. (\forall x':\mathbb{N}. x' < x+1 \rightarrow P(x')) \rightarrow P(x+1)$, während man bei der *Peano-Induktion* (dem ‘‘Gegenstück’’ zur *primitiven Rekursion*) nur $\forall x:\mathbb{N}. P(x) \rightarrow P(x+1)$ als Schrittfall zur Verfügung hat.

⁸ Es gilt $\text{exp2}(x+1) = 1 + \text{exp2}(x) + \dots + \text{exp2}(0)$ mit (11.5) und (11.6). Mit (11.9-5.) gilt $\text{sum}(x+1, y) = n_0 + \dots + n_x$ für die π^2 -Kodierung y einer Liste $\langle n_x, \dots, n_0 \rangle$, und folglich $S(\text{sum}(x+1, \text{exp2}^*(x+1))) = 1 + \text{exp2}(0) + \dots + \text{exp2}(x)$.

werden, und zum anderen, was die Bedeutung solcher Programme ist. Mit Angabe einer *operationalen* Semantik wurde in Definition 2.7 zugleich ein Rechenmodell für \mathcal{P} -Programme definiert.

Es fällt auf, daß bei Definition des Begriffs der primitiv-rekursiven Funktionen solch eine explizite Trennung zwischen Syntax und Semantik zu fehlen scheint, und insbesondere, daß überhaupt kein Rechenmodell angegeben wird. Dies hat sicher historische Gründe, denn zu Beginn des letzten Jahrhunderts gab es noch keine programmierbaren Rechner, geschweige denn Programmiersprachen.

Gleichwohl gibt es auch hier eine Unterscheidung zwischen Syntax und Semantik, nur wird diese eben nicht weiter betont:⁹ Die Definitionen 11.1, 11.2 und 11.3 geben die Syntax einer (fiktiven) Programmiersprache PRF für primitiv-rekursive Funktionen an, denn dort wird festgelegt, welche objektsprachlichen Ausdrücke überhaupt erlaubt sind, um primitiv-rekursive Funktionen zu definieren. Die Semantik dieser Ausdrücke wird dann mit Definition 11.4 gegeben, denn hier ist von semantischen Objekten – nämlich *Funktionen* – die Rede, die ausschließlich durch Gleichungen definiert werden, die den Definitionen 11.1, 11.2 und 11.3 genügen.

Ein Rechenmodell wird – im Unterschied zu \mathcal{P} -Programmen und den in Kapitel 12 vorgestellten *Turingmaschinen* – dabei jedoch nicht formal definiert. Dies ist vielmehr *implizit* durch die Anwendung der definierenden Gleichungen gegeben. Man kann solch ein Rechenmodell beispielsweise durch einen Kalkül explizit machen: Wir definieren diesen Kalkül in Abbildung 11.2 durch Angabe von 6 Regelschemata, die man unmittelbar aus den Definitionen 11.1, 11.2 und 11.3 gewinnt.¹⁰ Mit diesem Kalkül wird dann eine Relation \Rightarrow definiert durch: $t_1 \Rightarrow t_2$ gdw. $\frac{t_1}{t_2}$ eine Regel des Kalküls ist. Gerechnet wird jetzt durch schrittweise Anwendung der Kalkülregeln, wobei gilt: $n \in \mathbb{N}$ ist das *Ergebnis der Berechnung* von $f(n_1, \dots, n_k)$ gdw.¹¹

$$f(n_1, \dots, n_k) \Rightarrow^+ n.$$

Beispielsweise entsteht die Berechnung von $plus(1, 2)$ aus Abbildung 11.1 durch Anwendung dieser Kalkülregeln. Mit \Rightarrow wird damit das Rechenmodell für primitiv-rekursive Funktionen explizit gemacht. Man kann so einem nach Definition 11.4 gegebenen Funktionssymbol f die primitiv-rekursive Funktion $[[f]] : \mathbb{N}^k \rightarrow \mathbb{N}$

⁹ Ohne solch eine Unterscheidung könnten die primitiv-rekursiven – und darauf aufbauend die μ -rekursiven – Funktionen auch schwerlich ein Beitrag zur Berechenbarkeitstheorie sein.

¹⁰ Man erhält hier eine *Regel* aus einem *Regelschema*, in dem n durch irgendeine natürliche Zahl und x sowie jedes x_i durch einen beliebigen Ausdruck (bestehend aus Funktionssymbolen und natürlichen Zahlen) oder eine natürlich Zahl ersetzt wird. Schema (Pr3) darf nur auf Funktionen angewendet werden, die durch primitive Rekursion definiert wurden, und ist erforderlich, um etwa $plus(plus(1, 2), 3) \Rightarrow^+ plus(3, 3) (\Rightarrow^+ 6)$ zu erhalten. Anderfalls wäre keine Regel auf $plus(plus(1, 2), 3)$ anwendbar.

¹¹ \Rightarrow^+ bezeichnet die transitive Hülle von \Rightarrow .

| | | | | | |
|-------|---|-------|------------------------------------|-------|--|
| (Z) | $\frac{Z}{0}$ | (S) | $\frac{S(x)}{x+1}$ | (P) | $\frac{P_i^k(x_1, \dots, x_k)}{x_i}$ |
| (E) | $\frac{f(\vec{x})}{g(h_1(\vec{x}), \dots, h_m(\vec{x}))}$ | (Pr1) | $\frac{f(0, \vec{x})}{g(\vec{x})}$ | (Pr2) | $\frac{f(n+1, \vec{x})}{h(f(n, \vec{x}), n, \vec{x})}$ |
| (Pr3) | $\frac{f(x, \vec{x})}{f(x', \vec{x})}, \text{ falls } x \Rightarrow x'$ | | | | |

Abbildung 11.2: Ein Kalkül für primitiv-rekursive Funktionen

durch “ $\llbracket f \rrbracket(n_1, \dots, n_k) = n$ gdw. $f(n_1, \dots, n_k) \Rightarrow^+ n$ ” als operationale Semantik zuordnen, und $\llbracket \text{PRF} \rrbracket$ bezeichnet dann die Klasse aller primitiv-rekursiven Funktionen.

11.5. loop-berechenbare Funktionen

Man kann zeigen, daß jede primitiv-rekursive Funktion \mathcal{P} -berechenbar ist. Es kann sogar eine noch stärkere Aussage bewiesen werden:

Mit den **while**-Anweisungen der Programmiersprache \mathcal{P} kann man beliebige Schleifenprogramme schreiben. Insbesondere auch solche Programme, die nur Schleifen in der Art von *for*-Schleifen enthalten, die aus Programmiersprachen wie FORTRAN, PASCAL oder ADA bekannt sind. Solche Schleifen terminieren immer, denn die Anzahl der Abarbeitungen des Schleifenrumpfs ist immer durch einen festen Wert begrenzt. Wir nennen \mathcal{P} -Programme, die nur Schleifen in der Art von *for*-Schleifen enthalten **loop-Programme**, und Funktionen, die durch **loop**-Programme berechnet werden können, **loop-berechenbar**. Man kann zeigen, daß die Klasse der primitiv-rekursiven Funktionen identisch der Klasse der **loop**-berechenbaren Funktionen ist.

Definition 11.13. (loop-Programme, Programmiersprache $\mathcal{P}_{\text{loop}}$)

Wir erlauben in \mathcal{P} -Programmen **loop**-Anweisungen, d.h. zusätzliche Programm-anweisungen der Form

loop y do STA **end_loop**

wobei y eine lokale Variable, die sogenannte Laufvariable der **loop**-Anweisung, und STA eine Programmanweisung ist, in der y nicht vorkommt.



Ein `loop`-Programm ist ein \mathcal{P} -Programm, in dem (anstatt `while`-Anweisungen) nur `loop`-Anweisungen vorkommen.¹² $\mathcal{P}_{\text{loop}}$ bezeichnet die Menge aller `loop`-Programme, und $\mathcal{P}_{\text{loop}}[k] \subseteq \mathcal{P}[k]$ ist die Menge aller $\mathcal{P}_{\text{loop}}$ -Programme mit k formalen Parametern. ■

Die Semantik eines `loop`-Programms definieren wir durch Übersetzung in die Programmiersprache \mathcal{P} :

Definition 11.14. (Semantik von `loop`-Programmen)

Eine `loop`-Anweisung der Form “`loop y do STA end_loop`” wird aufgefaßt als abkürzende Schreibweise für eine `while`-Anweisung der Form

```
while y ≠ 0 do
  STA;
  y := PRED(y)
end_while . ■
```

In Definition 11.13 garantiert die Forderung für Laufvariable, daß diese im Rumpf einer `loop`-Anweisung nicht verändert werden können. Damit wird sichergestellt, daß `loop`-Anweisungen auch tatsächlich genauso wie *for*-Schleifen abgearbeitet werden. Andernfalls wären `loop`-Anweisungen der Form `loop y do y := SUCC(y) end_loop` möglich, mit denen man durch

```
while y ≠ 0 do
  y := SUCC(y);
  y := PRED(y)
end_while
```

nicht-terminierende Programmfragmente erhielte.¹³

Beispiel 11.15. (`loop`-Programme) Abbildung 11.3 definiert `loop`-Programme zur Berechnung von Addition, Multiplikation und der Fakultätsfunktion, sowie die \mathcal{P} -Programme für Addition und Multiplikation, die bei Elimination der Hilfsprozeduren und der Übersetzung der `loop`-Anweisungen in `while`-Anweisungen entstehen. ■

¹² Insbesondere ist P ein `loop`-Programm, wenn P weder `while`- noch `loop`-Anweisungen enthält.

¹³ Um nicht-Terminierung auszuschließen reicht es, lediglich *Zuweisungen* an Laufvariable zu verbieten. Die Verwendung von Laufvariablen y in Programmausdrücken wird nur aus beweistechnischen Gründen verboten, da dies andernfalls den Beweis von Satz 11.20 durch die *implizite* Zuweisung $y := \text{PRED}(y)$ unnötig erschweren würde. Dort, wo ein “lesender” Zugriff auf die Laufvariable erforderlich ist, muß dann ein “Doppelgänger” verwendet werden, wie etwa die Programmvariable z in der Prozedur `FAC` aus Abbildung 11.3.

```

procedure PLUS(x,y) <=
begin var res,z;
  res := y;
  z := x;
  loop z do res := SUCC(res) end_loop;
  return(res)
end

procedure TIMES(x,y) <=
begin var res,z;
  res := 0;
  z := x;
  loop z do res := PLUS(res,y) end_loop;
  return(res)
end

procedure PLUS(x,y) <=
begin var res,z;
  res := y; z := x;
  while z ≠ 0 do res := SUCC(res); z := PRED(z) end_while;
  return(res)
end

procedure TIMES(x,y) <=
begin var res',z',res,z;
  res' := 0; z' := x;
  while z' ≠ 0 do
    res := y; z := res';
    while z ≠ 0 do res := SUCC(res); z := PRED(z) end_while;
    res' := res;
    z' := PRED(z')
  end_while;
  return(res')
end

procedure FAC(x) <=
begin var res,y,z;
  res := 1;
  y := x;
  z := x;
  loop y do
    res := TIMES(z,res);
    z := PRED(z)
  end_loop;
  return(res)
end

```

Abbildung 11.3: $\mathcal{P}_{\text{loop}}$ -Programme (und ihre \mathcal{P} -Übersetzungen) zur Berechnung der Addition, der Multiplikation sowie der Fakultätsfunktion

| | |
|---|--|
| <pre> procedure P'_f(x, x1, ..., xk) <= begin var y, z, R; R := P_g(x1, ..., xk); y := x; z := x; loop z do R := P_h(R, MINUS(x, y), x1, ..., xk); y := PRED(y) end_loop; return(R) end </pre> | <pre> procedure P_f(x, x1, ..., xk) <= begin var y, z, R; R := P_g(x1, ..., xk); y := x; z := x; while z ≠ 0 do R := P_h(R, MINUS(x, y), x1, ..., xk); y := PRED(y); z := PRED(z) end_while; return(R) end </pre> |
|---|--|

Abbildung 11.4: Die Prozeduren P'_f und P_f zum Beweis von Satz 11.16

Wir zeigen nun mit den folgenden Sätzen, daß die Klasse der `loop`-berechenbaren Funktionen identisch der Klasse der primitiv-rekursiven Funktionen ist.

Satz 11.16. *Jede primitiv-rekursive Funktion ist $\mathcal{P}_{1\text{oop}}$ -berechenbar.*

Beweis. *Ausgehend von der induktiven Definition der primitiv-rekursiven Funktionen definieren wir für jede primitiv-rekursive Funktion $f : \mathbb{N}^k \mapsto \mathbb{N}$ ein $\mathcal{P}_{1\text{oop}}$ -Programm $P_f \in \mathcal{P}_{1\text{oop}}[k]$ mit $\llbracket P_f \rrbracket = f$.*

Induktionsanfang “ f ist eine Grundfunktion”: Für die $\mathcal{P}_{1\text{oop}}$ -Programme

```

procedure Z() <= begin var y; y := 0; return(y) end
procedure S(x) <= begin var y; y := SUCC(x); return(y) end
procedure P_i^k(x_1, ..., x_k) <= begin var y; y := x_i; return(y) end

```

gilt $\llbracket Z \rrbracket = Z$, $\llbracket S \rrbracket = S$ und $\llbracket P_i^k \rrbracket = P_i^k$.

Induktionsschritt “ f ist definiert durch Einsetzung”: Sei f gegeben durch $f(\vec{x}) := g(h_1(\vec{x}), \dots, h_m(\vec{x}))$. Dann sind die Funktionen h_1, \dots, h_m und g primitiv-rekursiv und somit nach Induktionsvoraussetzung $\mathcal{P}_{1\text{oop}}$ -berechenbar. Mit dem Beweis von Satz 3.3 ist dann auch f $\mathcal{P}_{1\text{oop}}$ -berechenbar, denn in diesem Beweis werden keine `while`-Anweisungen verwendet.

Induktionsschritt “ f ist definiert durch primitive Rekursion”: Sei f gegeben durch $f(0, \vec{x}) := g(\vec{x})$ und $f(x + 1, \vec{x}) := h(f(x, \vec{x}), x, \vec{x})$. Dann sind die Funktionen g und h primitiv-rekursiv und somit nach Induktionsvoraussetzung $\mathcal{P}_{\text{loop}}$ -berechenbar, etwa durch $\mathcal{P}_{\text{loop}}$ -Prozeduren P_g und P_h . Damit definieren wir eine $\mathcal{P}_{\text{loop}}$ -Prozedur P'_f , aus der wir durch Elimination der `loop`-Anweisung die \mathcal{P} -Prozedur P_f erhalten, s. Abbildung 11.4. Für die `while`-Anweisung dieser Prozedur gilt bei Aufruf von $\mathsf{P}_f(n, \vec{n})$ die Schleifeninvariante

$$\begin{aligned} & \{n \geq y \wedge R = f(n - y, \vec{n})\} \\ & \text{while } z \neq 0 \text{ do } \dots \text{end_while} \\ & \{n \geq y \wedge R = f(n - y, \vec{n})\} \end{aligned} \tag{11.14}$$

denn für $n \geq y \wedge R = f(n - y, \vec{n})$ sowie $y \neq 0$ erhält man nach einmaliger Ausführung des Schleifenrumpfs $R = \llbracket \mathsf{P}_h \rrbracket (f(n - (y + 1), \vec{n}), n - (y + 1), \vec{n}) = h(f(n - (y + 1), \vec{n}), n - (y + 1), \vec{n}))$, da mit der Induktionshypothese $\llbracket \mathsf{P}_h \rrbracket = h$ gilt. Für $n' := n - (y + 1)$ gilt damit $R = h(f(n', \vec{n}), n', \vec{n}) = f(n' + 1, \vec{n})$. Mit $n \geq y$ vor Ausführung des Schleifenrumpfs gilt nach Ausführung $n \geq y + 1$, also $n > y$, damit dann $n' + 1 = n - y$ und somit $R = f(n - y, \vec{n})$.

Mit $n = y$ und $R = \llbracket \mathsf{P}_g \rrbracket (\vec{n}) = g(\vec{n}) = f(0, \vec{n}) = f(n - y, \vec{n})$ ist die Schleifen-*vorbedingung* auch vor der erstmaligen Ausführung der `while`-Anweisung erfüllt, und folglich gilt $R = f(n - y, \vec{n})$ mit (11.14) nach der letztmaligen Ausführung des Schleifenrumpfs. Da $y = 0$ nach Terminierung der `while`-Anweisung gilt, erhält man schließlich $R = f(n, \vec{n})$ als Ergebnis der Ausführung von $\mathsf{P}_f(n, \vec{n})$. ■

Bemerkung 11.17. Mit den Begriffen aus Abschnitt 11.4 lautet die Behauptung von Satz 11.16:

$$\forall \phi \in \mathbb{N}^k \rightarrow \mathbb{N}. \phi \in \llbracket \text{PRF} \rrbracket \curvearrowright \phi \in \llbracket \mathcal{P}_{\text{loop}} \rrbracket. \tag{11.15}$$

Eine äquivalente Formulierung von (11.15) ist die Aussage

$$\forall f \in \text{PRF}. \exists P \in \mathcal{P}_{\text{loop}}. \llbracket f \rrbracket = \llbracket P \rrbracket. \tag{11.16}$$

Angenommen, wir geben eine totale Funktion $\mathsf{c} : \text{PRF} \rightarrow \mathcal{P}_{\text{loop}}$ an, so daß wir

$$\forall f \in \text{PRF}. \llbracket f \rrbracket = \llbracket \mathsf{c}(f) \rrbracket \tag{11.17}$$

beweisen können. Dann ist auch (11.16) – und folglich (11.15) – gezeigt, denn (11.17) impliziert offensichtlich (11.16). Wenn wir sogar eine algorithmische Funktion c angeben, so beschreibt c einen Compiler, der Programme der (fiktiven) Programmiersprache PRF für primitiv-rekursive Funktionen in äquivalente $\mathcal{P}_{\text{loop}}$ -Programme übersetzt.

Tatsächlich haben wir Satz 11.16 durch Nachweis der stärkeren Behauptung (11.17) bewiesen, denn wir haben für jede Definition f einer primitiv-rekursiven Funktion ein $\mathcal{P}_{\text{loop}}$ -Programm $c(f)$ – im Beweis durch \mathcal{P}'_f bezeichnet – angegeben, das die gleiche Funktion wie f berechnet. Da der Beweis von Satz 11.16 konstruktiv ist (vgl. Abschnitt 3.3), ist die Funktion c sogar algorithmisch: Wir können aus dem Beweis direkt ablesen, wie zu einer gegebenen Definition f einer primitiv-rekursiven Funktion ein äquivalentes $\mathcal{P}_{\text{loop}}$ -Programm $c(f)$ zu konstruieren ist.

Anders gesagt, Satz 11.16 wurde bewiesen, indem ein Compiler für PRF-Programme zunächst spezifiziert und dann gezeigt wurde, daß dieser Compiler auch korrekt ist.

Zum Nachweis, daß jedes loop -Programm eine primitiv-rekursive Funktion berechnet, simulieren wir die Interpretation von loop -Programmen durch primitiv-rekursive Funktionen. Dazu müssen wir zeigen, daß die Interpretationsfunktionen value und eval , vgl. Definitionen 2.6 und 2.7, primitiv-rekursiv sind (vorausgesetzt natürlich, daß eval nur auf $\mathcal{P}_{\text{loop}}$ -Programme angewendet wird). Wir definieren dazu:

Definition 11.18. ($\llbracket \text{EXPR} \rrbracket$ und $\llbracket \text{STA}(\mathbf{y}) \rrbracket$)

Sei EXPR ein Programmausdruck von \mathcal{P} , in dem genau k Programmvariable $\mathbf{y}_1, \dots, \mathbf{y}_k$ vorkommen, und sei $M_{\mathcal{P}}^{\vec{n}} \subseteq \mathbb{N} \times \mathbb{N}$ mit $\vec{n} = (n_1, \dots, n_k)$, so daß $\text{value}(M_{\mathcal{P}}^{\vec{n}}, \mathbf{y}_i) = n_i$ für jedes $i \in \{1, \dots, k\}$. Dann ist die Funktion $\llbracket \text{EXPR} \rrbracket : \mathbb{N}^k \rightarrow \mathbb{N}$ definiert durch

$$\llbracket \text{EXPR} \rrbracket(\vec{n}) := \text{value}(M_{\mathcal{P}}^{\vec{n}}, \text{EXPR}) .$$

Für eine Programmanweisung STA von \mathcal{P} , in der genau k lokale Variable $\mathbf{y}_1, \dots, \mathbf{y}_k$ und l formale Parameter $\mathbf{y}_{k+1}, \dots, \mathbf{y}_{k+l}$ vorkommen, sei $M_{\mathcal{P}}^{\vec{n}} \subseteq \mathbb{N} \times \mathbb{N}$ mit $\vec{n} = (n_1, \dots, n_{k+l})$ gegeben durch $\text{value}(M_{\mathcal{P}}^{\vec{n}}, \mathbf{y}_j) = n_j$ für jedes $j \in \{1, \dots, k+l\}$. Wir definieren für jedes $i \in \{1, \dots, k\}$ eine Funktion $\llbracket \text{STA}(\mathbf{y}_i) \rrbracket : \mathbb{N}^{k+l} \mapsto \mathbb{N}$ durch

$$\llbracket \text{STA}(\mathbf{y}_i) \rrbracket(\vec{n}) := \text{value}(\text{eval}(M_{\mathcal{P}}^{\vec{n}}, \text{STA}), \mathbf{y}_i) . \quad \blacksquare$$

Wir zeigen zuerst, daß value primitiv-rekursiv ist:

Satz 11.19. Für jeden Programmausdruck EXPR ist $\llbracket \text{EXPR} \rrbracket$ primitiv-rekursiv.

Beweis. Seien $\mathbf{y}_1, \dots, \mathbf{y}_k$ die Programmvariablen von EXPR , und sei $M_{\mathcal{P}}^{\vec{n}} \subseteq \mathbb{N} \times \mathbb{N}$ mit $\vec{n} = (n_1, \dots, n_k)$, so daß $\text{value}(M_{\mathcal{P}}^{\vec{n}}, \mathbf{y}_i) = n_i$ für jedes $i \in \{1, \dots, k\}$. Wir zeigen die Behauptung durch strukturelle Induktion über EXPR .

Induktionsanfang “ $\text{EXPR} = \mathbf{m} \in \mathbb{N}$ ”: Es gilt $\llbracket \text{EXPR} \rrbracket(\vec{n}) = \text{value}(M_P^{\vec{n}}, \mathbf{m}) = \mathbf{m} = C_m^k(\vec{n})$, und mit Lemma 11.5(1) ist $\llbracket \text{EXPR} \rrbracket$ primitiv-rekursiv.

Induktionsanfang “ $\text{EXPR} = \mathbf{y}_i$ ”: Es gilt $\llbracket \text{EXPR} \rrbracket(\vec{n}) = \text{value}(M_P^{\vec{n}}, \mathbf{y}_i) = n_i = P_i^k(\vec{n})$, und mit Definition 11.4(1) ist $\llbracket \text{EXPR} \rrbracket$ primitiv-rekursiv.

Induktionsschritt “ $\text{EXPR} = \text{SUCC}(\text{EXPR}')$ ”: Es gilt

$$\begin{aligned} & \llbracket \text{EXPR} \rrbracket(\vec{n}) \\ &= \text{value}(M_P^{\vec{n}}, \text{SUCC}(\text{EXPR}')) \\ &= 1 + \text{value}(M_P^{\vec{n}}, \text{EXPR}') \\ &= 1 + \llbracket \text{EXPR}' \rrbracket(\vec{n}) \\ &= S(\llbracket \text{EXPR}' \rrbracket(\vec{n})) \end{aligned}$$

und mit Definition 11.4(2) ist $\llbracket \text{EXPR} \rrbracket$ primitiv-rekursiv, denn die Funktion $\llbracket \text{EXPR}' \rrbracket$ ist nach Induktionshypothese primitiv-rekursiv.

Induktionsschritt “ $\text{EXPR} = \text{PRED}(\text{EXPR}')$ ”: Man erhält

$$\llbracket \text{EXPR} \rrbracket(\vec{n}) = \begin{cases} 0 & , \text{ falls } \text{value}(M_P^{\vec{n}}, \text{EXPR}') = 0 \\ m & , \text{ falls } \text{value}(M_P^{\vec{n}}, \text{EXPR}') = m + 1 \end{cases}$$

und folglich

$$\llbracket \text{EXPR} \rrbracket(\vec{n}) = \begin{cases} \llbracket \text{EXPR}' \rrbracket(\vec{n}), & \text{ falls } \llbracket \text{EXPR}' \rrbracket(\vec{n}) = 0 \\ \llbracket \text{EXPR}' \rrbracket(\vec{n}) - 1, & \text{ falls } \llbracket \text{EXPR}' \rrbracket(\vec{n}) > 0 . \end{cases}$$

Damit gilt $\llbracket \text{EXPR} \rrbracket(\vec{n}) = \text{pred}(\llbracket \text{EXPR}' \rrbracket(\vec{n}))$, und mit Lemma 11.5(2) sowie Definition 11.4(2) ist $\llbracket \text{EXPR} \rrbracket$ primitiv-rekursiv, denn die Funktion $\llbracket \text{EXPR}' \rrbracket$ ist nach Induktionshypothese primitiv-rekursiv. ■

Als nächstes zeigen wir, daß die Interpretationsfunktion *eval* ebenfalls primitiv-rekursiv ist, solange diese nur auf Anweisungen von **loop**-Programmen angewendet wird: Für eine Programmanweisung **STA** berechnen wir die Speicherbelegung, die bei Ausführung von **STA** entsteht, und bestimmen damit dann den Wert jeder **lokalen Variablen** nach Ausführung von **STA**. Mit $\llbracket \text{STA}(\mathbf{y}) \rrbracket$ erhält man so für **jede lokale Variable** \mathbf{y} von **STA** eine *primitiv-rekursive* Funktion, die eine Speicherbelegung (vor Ausführung von **STA**) in den Wert der lokalen Variablen \mathbf{y} (nach Ausführung von **STA**) abbildet.

Satz 11.20. Für jede Anweisung STA eine eines $\mathcal{P}_{\text{loop}}$ -Programms und jede lokale Variable y in STA ist $\llbracket \text{STA}(y) \rrbracket$ primitiv-rekursiv.

Beweis. Seien y_1, \dots, y_k die lokalen Variablen und y_{k+1}, \dots, y_{k+l} die formalen Parameter von STA, und sei $M_P^{\vec{n}} \subseteq \mathbb{N} \times \mathbb{N}$ mit $\vec{n} = (n_1, \dots, n_{k+l})$, so daß $\text{value}(M_P^{\vec{n}}, y_j) = n_j$ für jedes $j \in \{1, \dots, k+l\}$. Wir zeigen die Behauptung für jedes y_i mit $i \in \{1, \dots, k\}$ durch strukturelle Induktion über STA.

Induktionsanfang “STA = SKIP”: Es gilt

$$\begin{aligned} & \llbracket \text{STA}(y_i) \rrbracket(\vec{n}) \\ &= \text{value}(\text{eval}(M_P^{\vec{n}}, \text{SKIP}), y_i) \quad , \text{ mit Def. 11.18} \\ &= \text{value}(M_P^{\vec{n}}, y_i) \quad , \text{ mit Def. 2.7} \\ &= n_i \quad , \text{ mit Def. 2.6} \\ &= P_i^{k+l}(\vec{n}), \end{aligned}$$

und mit Definition 11.4(1) ist $\llbracket \text{STA}(y_i) \rrbracket$ primitiv-rekursiv.

Induktionsanfang “STA = $y_h := \text{EXPR}$ ”: Es gilt

$$\begin{aligned} & \llbracket \text{STA}(y_i) \rrbracket(\vec{n}) \\ &= \text{value}(\text{eval}(M_P^{\vec{n}}, y_h := \text{EXPR}), y_i) \quad , \text{ mit Def. 11.18} \\ &= \text{value}(M_P^{\vec{n}} [y_h \leftarrow \llbracket \text{EXPR} \rrbracket(\vec{n})], y_i) \quad , \text{ mit Def. 2.7} \\ &= \text{value}(M_P^{(n_1, \dots, n_{h-1}, \llbracket \text{EXPR} \rrbracket(\vec{n}), n_{h+1}, \dots, n_{k+l})}, y_i). \quad , \text{ mit Def. 2.4} \end{aligned}$$

Für $i = h$ erhält man damit $\llbracket \text{STA}(y_i) \rrbracket(\vec{n}) = \llbracket \text{EXPR} \rrbracket(\vec{n})$, und mit Satz 11.19 ist $\llbracket \text{STA}(y_i) \rrbracket$ primitiv-rekursiv. Für $i \neq h$ gilt $\llbracket \text{STA}(y_i) \rrbracket(\vec{n}) = n_i = P_i^{k+l}(\vec{n})$, und mit Definition 11.4(1) ist $\llbracket \text{STA}(y_i) \rrbracket$ auch in diesem Fall primitiv-rekursiv.

Induktionsschritt “STA = if EXPR then STA1 else STA2 end_if”: Man erhält

$$\llbracket \text{STA}(y_i) \rrbracket(\vec{n}) = \begin{cases} \text{value}(\text{eval}(M_P^{\vec{n}}, \text{STA1}), y_i), & \text{falls } \text{value}(M_P^{\vec{n}}, \text{EXPR}) > 0 \\ \text{value}(\text{eval}(M_P^{\vec{n}}, \text{STA2}), y_i), & \text{falls } \text{value}(M_P^{\vec{n}}, \text{EXPR}) = 0, \end{cases}$$

und folglich

$$\llbracket \text{STA}(y_i) \rrbracket(\vec{n}) = \begin{cases} \llbracket \text{STA1}(y_i) \rrbracket(\vec{n}), & \text{falls } \llbracket \text{EXPR} \rrbracket(\vec{n}) > 0 \\ \llbracket \text{STA2}(y_i) \rrbracket(\vec{n}), & \text{falls } \llbracket \text{EXPR} \rrbracket(\vec{n}) = 0. \end{cases}$$

Damit gilt $\llbracket \text{STA}(y_i) \rrbracket(\vec{n}) = \text{if}(\llbracket \text{EXPR} \rrbracket(\vec{n}), \llbracket \text{STA1}(y_i) \rrbracket(\vec{n}), \llbracket \text{STA2}(y_i) \rrbracket(\vec{n}))$, und mit Lemma 11.5(3), Satz 11.19 und Definition 11.4(2) ist $\llbracket \text{STA}(y_i) \rrbracket$ primitiv-rekursiv,

denn $\llbracket \text{STA1}(y_i) \rrbracket$ und $\llbracket \text{STA2}(y_i) \rrbracket$ sind nach Induktionsvoraussetzung primitiv-rekursiv.

Induktionsschritt “ $\text{STA} = \text{STA1}; \text{STA2}$ ”: Für $(*)$ $n'_j := \text{value}(\text{eval}(M_P^{\vec{n}}, \text{STA1}), y_j)$, falls $j \in \{1, \dots, k\}$, und $n'_j := n_j$, falls $j \in \{k+1, \dots, k+l\}$, erhält man

$$\begin{aligned}
& \llbracket \text{STA}(y_i) \rrbracket(\vec{n}) \\
&= \text{value}(\text{eval}(M_P^{\vec{n}}, \text{STA1}; \text{STA2}), y_i) && , \text{ mit Def. 11.18} \\
&= \text{value}(\text{eval}(\text{eval}(M_P^{\vec{n}}, \text{STA1}), \text{STA2}), y_i) && , \text{ mit Def. 2.7} \\
&= \text{value}(\text{eval}(M_P^{\vec{n}}[y_1 \leftarrow n'_1, \dots, y_k \leftarrow n'_k], \text{STA2}), y_i) && , \text{ mit Def. 2.8 und } (*) \\
&= \text{value}(\text{eval}(M_P^{\vec{n}'}, \text{STA2}), y_i) && , \text{ mit Def. 2.4 und } (*).
\end{aligned}$$

Damit gilt $\text{value}(\text{eval}(M_P^{\vec{n}'}, \text{STA2}), y_i) = \llbracket \text{STA2}(y_i) \rrbracket(\vec{n}')$ sowie $n'_i = \llbracket \text{STA1}(y_i) \rrbracket(\vec{n})$, also

$$\llbracket \text{STA}(y_i) \rrbracket(\vec{n}) = \llbracket \text{STA2}(y_i) \rrbracket(\llbracket \text{STA1}(y_1) \rrbracket(\vec{n}), \dots, \llbracket \text{STA1}(y_k) \rrbracket(\vec{n})) .$$

Nach Induktionsvoraussetzung sind $\llbracket \text{STA1}(y_i) \rrbracket$ und $\llbracket \text{STA2}(y_i) \rrbracket$ primitiv-rekursiv, und folglich ist $\llbracket \text{STA}(y_i) \rrbracket$ nach Definition 11.4(2) ebenfalls primitiv-rekursiv.

Induktionsschritt “ $\text{STA} = \text{loop } y_h \text{ do } \text{STA}' \text{ end_loop}$ ”: Wir nehmen o.B.d.A. $h = 1$ an und definieren für jedes $i' \in \{2, \dots, k\}$ und $\vec{m} := (n_2, \dots, n_{k+l})$ eine Funktion $f_{i'} : \mathbb{N}^{k+l} \mapsto \mathbb{N}$ durch¹⁴

$$f_{i'}(n, \vec{m}) := \text{value}(\text{eval}^{(n)}(M_P^{\vec{m}}, \text{STA}'), y_{i'}) \quad (11.18)$$

und damit gilt¹⁵

$$\llbracket \text{STA}(y_{i'}) \rrbracket(\vec{n}) = f_{i'}(n_1, \vec{m}) . \quad (11.19)$$

Man erhält

$$\boxed{f_{i'}(0, \vec{x}) = P_{i'-1}^{k+l-1}(\vec{x})} \quad (11.20)$$

denn

$$\begin{aligned}
& f_{i'}(0, \vec{m}) \\
&= \text{value}(\text{eval}^{(0)}(M_P^{\vec{m}}, \text{STA}'), y_{i'}) && , \text{ mit (11.18)} \\
&= \text{value}(M_P^{\vec{m}}, y_{i'}) && , \text{ mit Def. eval}^{(n)} \\
&= n_{i'} && , \text{ mit Def. 2.6} \\
&= P_{i'-1}^{k+l-1}(\vec{m})
\end{aligned}$$

¹⁴ Es gelte $\text{eval}^{(0)}(M_P, \text{STA}) := M_P$ und $\text{eval}^{(n+1)}(M_P, \text{STA}) := \text{eval}(\text{eval}^{(n)}(M_P, \text{STA}), \text{STA})$.

¹⁵ Es reicht, lediglich die Speicherbelegung $M_P^{\vec{m}}$ zu betrachten, da die Laufvariable y_1 nicht in STA' vorkommt.

sowie

$$f_{i'}(x+1, \vec{x}) = \llbracket \text{STA}'(\mathbf{y}_{i'}) \rrbracket (f_2(x, \vec{x}), \dots, f_k(x, \vec{x}), \vec{z}) \quad (11.21)$$

mit $\vec{z} := (x_{k+1}, \dots, x_{k+l})$ denn für (i) $n'_j := \text{value}(\text{eval}^{(n)}(M_P^{\vec{m}}, \text{STA}'), \mathbf{y}_j)$, falls $j \in \{2, \dots, k\}$, und $n'_j := n_j$, falls $j \in \{k+1, \dots, k+l\}$, und (ii) $\vec{m}' := (n'_2, \dots, n'_{k+l})$ erhält man

$$\begin{aligned} & f_{i'}(n+1, \vec{m}) \\ = & \text{value}(\text{eval}^{(n+1)}(M_P^{\vec{m}}, \text{STA}'), \mathbf{y}_{i'}) && , \text{ mit (11.18)} \\ = & \text{value}(\text{eval}(\text{eval}^{(n)}(M_P^{\vec{m}}, \text{STA}'), \text{STA}'), \mathbf{y}_{i'}) && , \text{ mit Def. } \text{eval}^{(n)} \\ = & \text{value}(\text{eval}(M_P^{\vec{m}}[\mathbf{y}_2 \leftarrow n'_2, \dots, \mathbf{y}_k \leftarrow n'_k], \text{STA}'), \mathbf{y}_{i'}) && , \text{ mit Def. 2.8 und (i)} \\ = & \text{value}(\text{eval}(M_P^{\vec{m}'}, \text{STA}'), \mathbf{y}_{i'}) && , \text{ mit Def. 2.4 und (ii)} \\ = & \llbracket \text{STA}'(\mathbf{y}_{i'}) \rrbracket (\vec{m}') && , \text{ mit Definition 11.18} \\ = & \llbracket \text{STA}'(\mathbf{y}_{i'}) \rrbracket (f_2(n, \vec{m}), \dots, f_k(n, \vec{m}), n_{k+1}, \dots, n_{k+l}) && , \text{ mit (i), (ii) und (11.18)}. \end{aligned}$$

Wegen (11.20) und (11.21) sind alle Funktionen $f_{i'}$ nach Satz 11.9 primitiv-rekursiv, denn alle Funktionen $\llbracket \text{STA}'(\mathbf{y}_{i'}) \rrbracket$ sind nach Induktionsvoraussetzung primitiv-rekursiv. Mit (11.19) gilt

$$\llbracket \text{STA}(\mathbf{y}_{i'}) \rrbracket (\vec{n}) = f_{i'}(\vec{n})$$

und folglich sind auch die Funktionen $\llbracket \text{STA}(\mathbf{y}_{i'}) \rrbracket$ primitiv-rekursiv. Mit $\llbracket \text{STA}(\mathbf{y}_1) \rrbracket (\vec{n}) := 0$ ist schließlich auch $\llbracket \text{STA}(\mathbf{y}_1) \rrbracket$ primitiv-rekursiv. ■

Der Nachweis, daß jede loop-berechenbare Funktion auch primitiv-rekursiv ist, folgt jetzt unmittelbar:

Korollar 11.21. Jede $\mathcal{P}_{\text{loop}}$ -berechenbare Funktion ist primitiv-rekursiv.

Beweis. Sei

```
procedure P( $\mathbf{x}_1, \dots, \mathbf{x}_k$ ) <=
begin var  $\mathbf{y}_1, \dots, \mathbf{y}_l$ ; STA; return( $\mathbf{y}_i$ ) end
```

ein $\mathcal{P}_{\text{loop}}$ -Programm. Dann gilt $\llbracket P \rrbracket (\vec{n}) = \text{value}(\text{eval}(M_P^{\vec{n}}, \text{STA}), \mathbf{y}_i) = \llbracket \text{STA}(\mathbf{y}_i) \rrbracket (\vec{n})$, und mit Satz 11.20 ist $\llbracket P \rrbracket$ primitiv-rekursiv. ■

Bemerkung 11.22. Auch der Beweis von Satz 11.20 ist konstruktiv, denn für jede lokale Variable \mathbf{y}_i in einer Programmanweisung STA wird eine konkrete Definition g_i für eine primitiv-rekursive Funktion $\llbracket g_i \rrbracket$ angegeben, für die dann $\llbracket g_i \rrbracket = \llbracket \text{STA}(\mathbf{y}_i) \rrbracket$ gilt. Somit wird auch Korollar 11.21 (wie schon dessen Rückrichtung in Satz 11.16) durch Spezifikation und Verifikation eines Compilers $c : \mathcal{P}_{\text{loop}} \rightarrow \text{PRF}$ bewiesen, vgl. Anmerkung 11.17.

11.6. Die Ackermann-Funktion

Man zeigt leicht, daß jede primitiv-rekursive Funktion *total* ist:

Satz 11.23. *Jede primitiv-rekursive Funktion ist total.*

Beweis. Sei f eine beliebige primitiv-rekursive Funktion. Wir zeigen die Totalität von f durch Induktion über den Aufbau der primitiv-rekursiven Funktionen.

Induktionsanfang “ f ist eine Grundfunktion”: In diesem Fall ist die Behauptung offensichtlich.

Induktionsschritt “ f ist definiert durch Einsetzung”: Für $f(\vec{x}) := g(h_1(\vec{x}), \dots, h_m(\vec{x}))$, vgl. Definition 11.2, sind die Funktionen h_1, \dots, h_m und g nach Induktionsvoraussetzung *total*. Damit muß auch f *total* sein.

Induktionsschritt “ f ist definiert durch primitive Rekursion”: Für f , gegeben wie in Definition 11.4, definieren wir für jedes $x \in \mathbb{N}$ eine Funktion $f_x : \mathbb{N}^k \rightarrow \mathbb{N}$ mit $f_x(\vec{x}) := f(x, \vec{x})$ und zeigen durch Induktion über x , daß jede dieser Funktionen *total* ist.

Induktionsanfang $x = 0$: Es gilt $f_0(\vec{x}) = f(0, \vec{x}) = g(\vec{x})$, also ist f_0 *total*, da g nach Induktionsvoraussetzung (der äußeren Induktion) *total* ist.

Induktionsschritt $x = x' + 1$: Es gilt $f_{x'+1}(\vec{x}) = f(x' + 1, \vec{x}) = h(f(x', \vec{x}), x', \vec{x}) = h(f_{x'}(\vec{x}), x', \vec{x})$. Nach Induktionsvoraussetzung (der inneren Induktion) ist $f_{x'}$ *total*, und h ist nach Induktionsvoraussetzung (der äußeren Induktion) ebenfalls *total*. Folglich muß auch $f_{x'+1}$ *total* sein.

Damit ist auch f *total*, denn für beliebige natürliche Zahlen x, \vec{x} gilt $f(x, \vec{x}) = f_x(\vec{x})$, und mit der Totalität von f_x ist dann auch $f(x, \vec{x})$ definiert. ■

Aus Abschnitt 10.1 wissen wir, daß es für jede Programmiersprache, in der nur totale Funktionen programmiert werden können, eine totale berechenbare Funktion gibt, die durch kein Programm dieser Programmiersprache berechnet werden kann. Fassen wir das Definitionsprinzip für primitiv-rekursive Funktionen als Programmiersprache auf, so folgt mit Satz 11.23 sofort, daß es eine totale berechenbare Funktion gibt, die nicht primitiv-rekursiv ist.

Ein konkretes Beispiel dafür wurde zuerst von *W. Ackermann* angegeben und später von *R. Peter* vereinfacht. Man definiert die (vereinfachte) *Ackermann-Funktion* $ack: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ durch

1. $ack(0, y) := y + 1$,
2. $ack(x + 1, 0) := ack(x, 1)$, und
3. $ack(x + 1, y + 1) := ack(x, ack(x + 1, y))$.

Satz 11.24. Die Ackermann-Funktion ist total.

Beweis. Sei $(x, y) \gg (x', y')$ gdw. $x > x'$ oder $x = x'$ und $y > y'$. Wir zeigen $ack(x, y) \in \mathbb{N}$ für alle $x, y \in \mathbb{N}$ durch Induktion über \gg .

Induktionsanfang $(0, y)$: Dann gilt $ack(0, y) := y + 1 \in \mathbb{N}$.

Induktionsschritt $(x + 1, 0)$: Nach Induktionsvoraussetzung gilt $ack(x, 1) \in \mathbb{N}$ und mit $ack(x + 1, 0) = ack(x, 1)$ dann auch $ack(x + 1, 0) \in \mathbb{N}$.

Induktionsschritt $(x + 1, y + 1)$: Nach Induktionsvoraussetzung gilt $ack(x + 1, y) \in \mathbb{N}$ sowie dann auch $ack(x, ack(x + 1, y)) \in \mathbb{N}$. Mit $ack(x + 1, y + 1) = ack(x, ack(x + 1, y))$ erhält man $ack(x + 1, y + 1) \in \mathbb{N}$. ■

Es gilt:

Satz 11.25. Die Ackermann-Funktion ist nicht primitiv-rekursiv.

Zum Beweis von Satz 11.25 zeigt man, daß ack stärker als jede primitiv-rekursive Funktion wächst. Einen formalen Beweis dafür findet man beispielsweise in [Hermes(1971)] und [Wagner(1994)].

Bemerkung 11.26. In Abschnitt 11.1 wurden verallgemeinerte Definitionsschemata für primitiv-rekursive Funktionen angegeben, wobei jedoch geschachtelte Rekursionen, d.h. Ausdrücke der Form $f(\dots f(\dots))$ auf der rechten Seite einer definierenden Gleichung nicht erlaubt wurden. Mit Satz 11.25 wird auch der Grund für dieses Verbot offensichtlich. Dies bedeutet natürlich nicht, daß mit geschachtelten Rekursionen nur Funktionen definiert werden können, die nicht primitiv-rekursiv sind. Beispielsweise wird mit $f(0) := 0$ und $f(x + 1) := 1 + f(f(x))$ eine primitiv-rekursive Funktion definiert, denn man zeigt leicht $f(n) = n$ für alle $n \in \mathbb{N}$ durch Induktion über n . Allerdings wandelt man bei geschachtelten Rekursionen auf einem schmalen Grat: Die (fast genauso wie f definierten) Funktionen f' und f'' , gegeben durch $f'(0) := 1$ und $f'(x + 1) := 1 + f'(f'(x))$ sowie $f''(0) := 0$ und $f''(x + 1) := 2 + f''(f''(x))$ sind nicht total (und mit Satz 11.23 dann auch nicht primitiv-rekursiv).

11.7. μ -rekursive Funktionen

Da mit primitiver Rekursion nicht alle berechenbaren Funktionen definiert werden können, wird ein zusätzliches Definitionsprinzip eingeführt:

Definition 11.27. (Definition durch Minimierung, μ -Operator)

Sei $f : \mathbb{N}^{k+1} \mapsto \mathbb{N}$ eine Funktion mit $k \geq 0$ und sei die Funktion $\mu f : \mathbb{N}^k \mapsto \mathbb{N}$ gegeben durch

$$\mu f(\vec{x}) := \begin{cases} x & , \text{ falls } f(x, \vec{x}) = 0 \text{ und } f(x', \vec{x}) \in \mathbb{N} \setminus \{0\} \text{ für alle } x' < x \\ \perp & , \text{ andernfalls.} \end{cases}$$

Dann ist die Funktion μf definiert durch (unbeschränkte) Minimierung (der Funktion f). ■

Beispiel 11.28. (Definition durch Minimierung)

1. Es gilt $\mu plus(0) = 0$ und $\mu plus(y + 1) = \perp$ sowie $\mu times(y) = 0$.
2. Sei $g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ definiert durch $g(x, y) = y - x^2$. Es gilt $\mu g(y) = z$ gdw. $y - z^2 = 0$ und $y - \bar{z}^2 \in \mathbb{N} \setminus \{0\}$ für alle $\bar{z} < z$, d.h. $\bar{z}^2 < y \leq z^2$, also $\bar{z} < \sqrt{y} \leq z$. Damit gilt $\mu g(y) = \lceil \sqrt{y} \rceil$, d.h. $\mu g(y)$ approximiert \sqrt{y} als die kleinste natürliche Zahl, deren Quadrat größer oder gleich y ist.
3. Sei $f_h : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$ definiert durch $f_h(x, y) = (h(x) - y) + (y - h(x))$ für eine berechenbare Funktion $h : \mathbb{N} \mapsto \mathbb{N}$. Mit $\mu f_h(y) = z$ gilt für alle $z' < z$:

$$\begin{aligned} & (h(z) - y) + (y - h(z)) = 0 \wedge (h(z') - y) + (y - h(z')) \in \mathbb{N} \setminus \{0\} \\ \text{gdw.} & \quad h(z) - y = 0 \wedge y - h(z) = 0 \wedge ((h(z') - y) \neq 0 \vee (y - h(z')) \neq 0) \\ \text{gdw.} & \quad h(z) \leq y \leq h(z) \wedge (h(z') > y \vee y > h(z')) \\ \text{gdw.} & \quad h(z) = y \neq h(z') . \end{aligned}$$

Damit berechnet $\mu f_h(y)$ das kleinste h -Urbild z von y , falls $y \in \text{Bild}(h)$ und $\{0, \dots, z - 1\} \subseteq \text{Def}(h)$. Für $y \notin \text{Bild}(h)$ gilt $\mu f_h(y) = \perp$, und folglich $\text{Def}(\mu f_h) \subseteq \text{Bild}(h)$.

- (i) Ist h nicht total, so gilt $|\text{Def}(\mu f_h)| < \infty$, d.h. die Minimierung nicht-totaler Funktionen ist wenig nützlich, vgl. Bemerkung 11.31. Man erhält z.B. $\text{Def}(\mu f_{\log_2}(y)) = \emptyset$, denn $\log_2(0) = \perp$.¹⁶

¹⁶ Mit $\log_2(2^x) := x$ und $\log_2(y) = \perp$, falls y keine 2er-Potenz ist.

- (ii) Ist h total, so gilt $\text{Def}(\mu f_h) = \text{Bild}(h)$, also z.B. $\mu f_S(0) = \perp$ und $\mu f_S(y+1) = y$ sowie $\mu f_{2^x}(y) = \log_2(y)$, falls y eine 2er-Potenz ist, und $\mu f_{2^x}(y) = \perp$ andernfalls.
- (iii) Ist h total und surjektiv, so ist μf_h ebenfalls total, wie beispielsweise bei $\mu f_{\lfloor x/2 \rfloor}(y) = 2y$. ■

Ist $f : \mathbb{N}^{k+1} \mapsto \mathbb{N}$ eine berechenbare Funktion, so können wir $\mu f(\vec{n})$ für alle $\vec{n} \in \mathbb{N}^k$ allein mittels f berechnen: Für $n := 0$ starten wir die Berechnung von $f(n, \vec{n})$. Gilt $f(n, \vec{n}) = \perp$, so endet diese Berechnung nicht, folglich endet auch die Berechnung von $\mu f(\vec{n})$ nicht, und wir erhalten $\mu f(\vec{n}) = \perp$. Andernfalls testen wir, ob $f(n, \vec{n}) = 0$ gilt. Im positiven Fall ist dann n das Ergebnis von $\mu f(\vec{n})$. Im negativen Fall setzen wir $n := n + 1$ und beginnen von vorne. Damit erhalten wir das kleinste n mit $f(n, \vec{n}) = 0$ sowie $f(n', \vec{n}) \in \mathbb{N} \setminus \{0\}$ für alle $n' < n$ als Ergebnis von $\mu f(\vec{n})$, falls ein n mit $f(n, \vec{n}) = 0$ überhaupt existiert. Andernfalls gilt entweder (i) $f(n, \vec{n}) \in \mathbb{N} \setminus \{0\}$ für alle $n \in \mathbb{N}$ oder aber (ii) $f(n, \vec{n}) \notin \mathbb{N}$ und $f(n', \vec{n}) \in \mathbb{N} \setminus \{0\}$ für alle $n' < n$. In Fall (i) hält die Schleife nie, und damit ist \perp das Ergebnis von $\mu f(\vec{n})$. In Fall (ii) gilt $f(n, \vec{n}) = \perp$, und folglich erhält man auch hier \perp als Ergebnis von $\mu f(\vec{n})$.

Anhand dieser Überlegung ist offensichtlich, daß man mittels des μ -Operators ein Konzept in Art einer Schleife erhält, denn die Funktion f wird solange mit (festem \vec{n} und) jeweils um 1 erhöhtem Argument n aufgerufen, bis das Schleifenabbruchkriterium “ $f(n, \vec{n}) = 0$ ” erfüllt ist. Damit erhält man ein explizites Rechenmodell für (die nachfolgend definierten) μ -rekursive Funktionen durch Erweiterung des Kalküls aus Abschnitt 11.4 um drei zusätzliche Regelschemata wie in Abbildung 11.5.¹⁷ Hierbei ist $\mu \hat{f}$ eine Hilfsfunktion, die die “Hochzählschleife” des μ -Operators modelliert: Mit (M1) wird die Laufvariable dieser Schleife mit 0 initialisiert, mit (M2) erhält man das Ergebnis von $\mu f(\vec{n})$, falls ein solches existiert, und mit (M3) wird die Laufvariable bei Fehlschlag von (M2) um 1 erhöht.

Damit wird auch die Motivation für die Forderung (*) “ $f(x', \vec{x}) \in \mathbb{N} \setminus \{0\}$ für alle $x' < x$ ” aus Definition 11.27 ersichtlich: Ersetzt man (*) durch (**) “ $f(x', \vec{x}) \neq 0$ für alle $x' < x$ ”, also “ $f(x', \vec{x}) = \perp$ oder $f(x', \vec{x}) \in \mathbb{N} \setminus \{0\}$ für alle $x' < x$ ”, so kann μf (selbst für eine berechenbare Funktion f) nicht notwendigerweise iterativ berechnet werden:

¹⁷ Bei diesen Regelschemata steht \vec{x} für eine Folge beliebiger Ausdrücke, bestehend aus Funktionssymbolen und natürlichen Zahlen. Der Wert $n \in \mathbb{N}$ für die Laufvariable darf nur durch (M1) – (M3) gesetzt werden.

$$\begin{array}{l}
\text{(M1)} \quad \frac{\mu f(\vec{x})}{\mu \hat{f}(0, \vec{x})} \\
\text{(M2)} \quad \frac{\mu \hat{f}(n, \vec{x})}{n}, \text{ falls } f(n, \vec{x}) \Rightarrow^+ 0 \\
\text{(M3)} \quad \frac{\mu \hat{f}(n, \vec{x})}{\mu \hat{f}(n+1, \vec{x})}, \text{ falls } f(n, \vec{x}) \Rightarrow^+ n'+1 \in \mathbb{N}
\end{array}$$

Abbildung 11.5: Erweiterung des Rechenmodells für den μ -Operator

Beispiel 11.29. (μ -Operator) Sei die Funktion $f : \mathbb{N} \mapsto \mathbb{N}$ definiert durch $f(0) := \perp$ und $f(x+1) := 0$. Nach Definition 11.27 gilt $\mu f = \perp$, denn $f(0) := \perp$, und man erhält genau dieses Ergebnis bei iterativer Berechnung von f . Ersetzt man jedoch Forderung (*) in Definition 11.27 durch die Forderung (**), so gilt $\mu f = 1$, die iterative Berechnung von f liefert jedoch \perp . ■

Ausgehend von den Grundfunktionen erlauben wir jetzt bei Angabe einer Funktion neben “Definition durch Einsetzung” und “Definition durch primitive Rekursion” auch “Definition durch Minimierung”. Man erhält so die Klasse der μ -rekursiven Funktionen als eine echte Obermenge der primitiv-rekursiven Funktionen:

Definition 11.30. (μ -rekursive Funktionen)

Eine Funktion $f : \mathbb{N}^k \mapsto \mathbb{N}$ mit $k \geq 0$ heißt μ -rekursiv, gdw. gilt:

1. f ist eine Grundfunktion, oder
2. f ist definiert durch Einsetzung mittels μ -rekursiver Funktionen, oder
3. f ist definiert durch primitive Rekursion mittels μ -rekursiver Funktionen, oder
4. f ist definiert durch Minimierung einer μ -rekursiven Funktion. ■

Bemerkung 11.31. Der μ -Operator ist natürlich nicht prinzipiell auf das erste Argument einer zu minimierenden Funktion beschränkt, denn man kann leicht mittels μf und Grundfunktionen einen μ -Operator $\mu f.x_i$ definieren, der das i .te Argument x_i einer Funktion f minimiert.

In manchen Lehrbüchern wird bei Definition des μ -Operators “ $f(x, \vec{x}) = 1$ ” anstatt “ $f(x, \vec{x}) = 0$ ” gefordert. Es sollte offensichtlich sein, daß solche Unterschiede für die definierte Theorie unerheblich sind. Des weiteren findet man in einigen Texten lediglich die schwächere Forderung (**) “ $f(x', \vec{x}) \neq 0$ für alle $x' < x$ ” anstatt Forderung (*) “ $f(x', \vec{x}) \in \mathbb{N} \setminus \{0\}$ für alle $x' < x$ ”. Dafür wird dann jedoch bei Definition der μ -rekursiven Funktionen “4. f ist definiert durch Minimierung einer totalen μ -rekursiven Funktion” anstatt 11.30(4) verlangt, s.a. Beispiel 11.28(3.(i)). Diese Einschränkung stellt sicher, daß die Berechnung von μf durch die iterative Berechnung von f trotzdem gelingt. (In Beispiel 11.29 darf der μ -Operator dann nicht auf die Funktion f angewendet werden).

Trotzdem gibt es einen gravierenden Unterschied zwischen diesen beiden Alternativen, und zwar in der Frage, ob wir die Notation zur Definition μ -rekursiver Funktionen als eine (fiktive) Programmiersprache μRF auffassen können: Von einer Programmiersprache wird sinnvollerweise gefordert, daß entschieden werden kann, ob ein vorliegender Text den syntaktischen Konventionen der Sprache genügt, vgl. die Entscheidungsfunktion *parse* im Beweis von Satz 4.12(3.1). Es muß also entscheidbar sein, ob ein Text tatsächlich ein Programm der Programmiersprache darstellt. Für unsere Definition der μ -rekursiven Funktionen trifft das sicherlich zu, denn es ist entscheidbar, ob eine Funktion als Grundfunktion, durch Einsetzung, durch primitive Rekursion oder durch Minimierung definiert wurde. Fordert man jedoch “4. f ist definiert durch Minimierung einer totalen μ -rekursiven Funktion” in Definition 11.30, so verliert man die Entscheidbarkeit. Jetzt muß ein Syntaxanalysator (als eine berechenbare Funktion *parse*) entscheiden, ob eine μ -rekursive Funktion total ist, um die korrekte Anwendung des μ -Operators zu überprüfen. Nach Satz 10.1 ist jedoch die Totalität einer berechenbaren Funktion noch nicht einmal semi-entscheidbar, und damit ist die Alternativdefinition μ -rekursiver Funktionen als Programmiersprache ungeeignet.

Schränkt man die Anwendbarkeit des μ -Operators noch weiter ein, und zwar auf primitiv-rekursiv definierte Funktionen, so ist die resultierende Programmiersprache offensichtlich entscheidbar. Allerdings stellt sich dann die Frage, ob diese Einschränkung nicht auch eine Einschränkung der so definierten Klasse der μ -rekursiven Funktionen (im Vergleich zu Definition 11.30) nach sich zieht. **Mit dem Beweis des Kleeneschen Normalformatsatzes (Korollar 11.41) werden wir zeigen, daß dies nicht der Fall ist.**

Bemerkung 11.32. Die (fiktive) Programmiersprache μRF ist nicht strikt, da Funktionsaufrufe call-by-name ausgewertet werden (man spricht in solch einem Fall von lazy evaluation). Beispielsweise gilt $P_1^2(42, \mu S) \Rightarrow^+ 42$ sowie $\text{times}(0, \mu S) \Rightarrow^+ 0$ für die primitiv-rekursive Funktion *times* aus Beispiel 11.8, obwohl die Berechnung von μS nicht terminiert (denn $\mu S \not\Rightarrow^+ n$ für jedes $n \in \mathbb{N}$).

Funktionen, die durch primitive Rekursion definiert wurden, sind (wegen Regelschema (Pr3) aus Abbildung 11.2) dagegen strikt bzgl. ihres Rekursionsarguments. Beispielsweise gilt $\text{times}(\mu S, 0) \not\leq^+ n$ für alle $n \in \mathbb{N}$. Insbesondere ist damit auch die Funktion if aus Lemma 11.5 strikt in ihrem Bedingungsargument, d.h. $\text{if}(b, \dots, \dots) \not\leq^+ n$ für alle $n \in \mathbb{N}$, wenn $b \not\leq^+ n$ für alle $n \in \mathbb{N}$.

Beispiel 11.33. (μ -rekursive Funktionen) Alle Funktionen aus Beispiel 11.28 sind μ -rekursiv. Weiter gilt für die Funktionen aus diesem Beispiel:

1. μplus ist nicht primitiv-rekursiv, μtimes ist primitiv-rekursiv.
2. μg ist primitiv-rekursiv.
3. μf_{\log_2} , μf_S und μf_{2^x} sind nicht primitiv-rekursiv, $\mu f_{\lfloor x/2 \rfloor}$ ist primitiv-rekursiv. ■

Wir zeigen jetzt, daß die Klasse der μ -rekursiven Funktionen identisch der Klasse der \mathcal{P} -berechenbaren Funktionen ist. Den Beweis für die Behauptung, daß μ -rekursive Funktionen \mathcal{P} -berechenbar sind, gewinnen wir aus dem Beweis der Behauptung, daß primitiv-rekursive Funktionen $\mathcal{P}_{\text{loop}}$ -berechenbar sind (Satz 11.16). Wir müssen diesen Beweis lediglich noch für den Fall ergänzen, in dem eine Funktion mittels Minimierung definiert wird. Dies gelingt, da jetzt beliebige `while`-Anweisungen zur Verfügung stehen:

Satz 11.34. Jede μ -rekursive Funktion ist \mathcal{P} -berechenbar.

Beweis. Ausgehend von der induktiven Definition der μ -rekursiven Funktionen definieren wir für jede μ -rekursive Funktion $f : \mathbb{N}^k \mapsto \mathbb{N}$ ein \mathcal{P} -Programm $P_f \in \mathcal{P}[k]$ mit $\llbracket P_f \rrbracket = f$.

Induktionsanfang “ f ist eine Grundfunktion”:
Wir definieren P_f wie im Beweis von Satz 11.16.

Induktionsschritt “ f ist definiert durch Einsetzung”:
Man zeigt die Behauptung wie im Beweis von Satz 11.16.

Induktionsschritt “ f ist definiert durch primitive Rekursion”:
Man zeigt die Behauptung wie im Beweis von Satz 11.16, da die schwächere Induktionsvoraussetzung “die Funktionen g und h sind μ -rekursiv” ausreicht, um die μ -Rekursivität von f zu zeigen.

Induktionsschritt “ f ist definiert durch Minimierung”:
Sei f gegeben durch Minimierung einer μ -rekursiven Funktion $g : \mathbb{N}^{k+1} \mapsto \mathbb{N}$, also $f = \mu g$. Dann ist

g nach Induktionsvoraussetzung \mathcal{P} -berechenbar, etwa durch eine \mathcal{P} -Prozedur P_g .
Damit definieren wir die \mathcal{P} -Prozedur

```

procedure  $P_f(x_1, \dots, x_k) \Leftarrow$ 
begin var res;
  res := 0;
  while  $P_g(res, x_1, \dots, x_k) \neq 0$  do
    res := SUCC(res)
  end_while;
  return(res)
end

```

und zeigen $\llbracket P_f \rrbracket(\vec{n}) = \mu g(\vec{n})$ für alle $\vec{n} \in \mathbb{N}^k$:

Fall “ $g(n, \vec{n}) = 0$ für ein $n \in \mathbb{N}$ sowie $g(n', \vec{n}) \in \mathbb{N} \setminus \{0\}$ für alle $n' < n$ ”:
Dann gilt $\mu g(\vec{n}) = n$ und die Berechnung von $P_g(res, x_1, \dots, x_k)$ terminiert solange
 $res = n' \leq n$ gilt. Da der Schleifentest für alle $n' < n$ erfolgreich ist, wird res
solange hochgezählt bis $res = n$ gilt. Mit $\llbracket P_g \rrbracket(res, \vec{n}) = 0$ wird die Abarbeitung der
Schleife beendet, und die Ausführung von $P_f(x_1, \dots, x_k)$ terminiert mit Ergebnis
 $n = \mu g(\vec{n})$.

Fall “ $g(n, \vec{n}) = \perp$ für ein $n \in \mathbb{N}$ sowie $g(n', \vec{n}) \in \mathbb{N} \setminus \{0\}$ für alle $n' < n$ ”: Dann
gilt $\mu g(\vec{n}) \notin \mathbb{N}$ und die Berechnung von $P_g(res, x_1, \dots, x_k)$ terminiert solange $res =$
 $n' < n$ gilt. Da der Schleifentest für alle $n' < n$ erfolgreich ist, wird res solange
hochgezählt bis $res = n$ gilt. Hier terminiert die Berechnung von $P_g(res, x_1, \dots, x_k)$
jedoch nicht mehr. Also terminiert auch die Ausführung von $P_f(x_1, \dots, x_k)$ nicht,
und somit gilt $\llbracket P_f \rrbracket(\vec{n}) = \perp = \mu g(\vec{n})$.

Fall “ $g(n, \vec{n}) \in \mathbb{N} \setminus \{0\}$ für alle $n \in \mathbb{N}$ ”: Dann gilt $\mu g(\vec{n}) \notin \mathbb{N}$ und die Berech-
nung von $P_g(res, x_1, \dots, x_k)$ terminiert immer. Da mit $\llbracket P_g \rrbracket(n, n_1, \dots, n_k) \neq 0$ der
Schleifentest jedoch für alle n erfolgreich ist, terminiert die **while**-Schleife in P_f
nicht. Also terminiert die Ausführung von $P_f(x_1, \dots, x_k)$ nicht, und somit gilt
 $\llbracket P_f \rrbracket(\vec{n}) = \perp = \mu g(\vec{n})$. ■

Bemerkung 11.35. Wie bei Beweis von Satz 11.16 wird auch Satz 11.34 durch
Spezifikation eines Compilers $c : \mu\text{RF} \rightarrow \mathcal{P}$ und dessen Korrektheitsnachweis be-
wiesen, vgl. Anmerkung 11.17.

Zum Beweis, daß jede \mathcal{P} -berechenbare Funktion auch μ -rekursiv ist, müssen
wir analog zum Beweis von Satz 11.20 jeder Programmanweisung **STA** und jeder

lokalen Variablen \mathbf{y} in STA eine Funktion $f_{\text{STA},\mathbf{y}}$ zuordnen, die eine Speicherbelegung (vor Ausführung von STA) in den Wert der lokalen Variablen $\bar{\mathbf{y}}$ (nach Ausführung von STA) abbildet. Da jetzt jedoch auch beliebige while-Anweisungen betrachtet werden, ist $f_{\text{STA},\mathbf{y}}$ nicht notwendigerweise total und damit i.A. auch nicht primitiv-rekursiv. Man kann jedoch zeigen, daß jede Funktion $f_{\text{STA},\mathbf{y}}$ primitiv-rekursiv ist, wenn STA unter einer Laufzeitbeschränkung ausgeführt wird, vgl. Abschnitt 8.1. Wir definieren dazu für eine Programmanweisung STA und eine lokale Variable \mathbf{y} in STA eine Funktion $\langle \text{STA}(\mathbf{y}) \rangle$ (anstatt $f_{\text{STA},\mathbf{y}}$) mit $\langle \text{STA}(\mathbf{y}) \rangle(b, \bar{\mathbf{n}}) = \llbracket \text{STA}(\mathbf{y}) \rrbracket(\bar{\mathbf{n}})$, falls die Ausführung von STA unter der Laufzeitbeschränkung b gelingt. Den Erfolg (oder Mißerfolg) der beschränkten Ausführung von STA bestimmen wir mittels einer weiteren primitiv-rekursiven Funktion $\langle \text{STA} \rangle$ mit $\langle \text{STA} \rangle(b, \bar{\mathbf{n}}) > 0$ gdw. wenn die Ausführung von STA unter der Laufzeitbeschränkung b gelingt. Anders gesagt, so wie die Ausführungsfunktion $eval$ durch $\llbracket \text{STA}(\mathbf{y}) \rrbracket$ modelliert wird, so modelliert $\langle \text{STA} \rangle$ die Schrittzählfunktion $step$ aus Definition 8.1.

Definition 11.36. ($\langle \text{STA} \rangle$ und $\langle \text{STA}(\mathbf{y}) \rangle$)

Für eine Programmanweisung STA von \mathcal{P} , in der genau k lokale Variable $\mathbf{y}_1, \dots, \mathbf{y}_k$ und l formale Parameter $\mathbf{y}_{k+1}, \dots, \mathbf{y}_{k+l}$ vorkommen, sei $M_{\mathcal{P}}^{\bar{\mathbf{n}}} \subseteq \mathbb{N} \times \mathbb{N}$ mit $\bar{\mathbf{n}} = (n_1, \dots, n_{k+l})$ gegeben durch $value(M_{\mathcal{P}}^{\bar{\mathbf{n}}}, \mathbf{y}_j) = n_j$ für jedes $j \in \{1, \dots, k+l\}$. Wir definieren eine Funktion $\langle \text{STA} \rangle : \mathbb{N}^{k+l+1} \rightarrow \mathbb{N}$ und für jedes $i \in \{1, \dots, k\}$ eine Funktion $\langle \text{STA}(\mathbf{y}_i) \rangle : \mathbb{N}^{k+l+1} \rightarrow \mathbb{N}$ durch

$$\begin{aligned} \langle \text{STA} \rangle(b, \bar{\mathbf{n}}) &:= step(b, M_{\mathcal{P}}^{\bar{\mathbf{n}}}, \text{STA}) \\ \langle \text{STA}(\mathbf{y}_i) \rangle(b, \bar{\mathbf{n}}) &:= \begin{cases} 0, & \text{falls } \langle \text{STA} \rangle(b, \bar{\mathbf{n}}) = 0 \\ \llbracket \text{STA}(\mathbf{y}_i) \rrbracket(\bar{\mathbf{n}}), & \text{andernfalls.} \end{cases} \quad \blacksquare \end{aligned}$$

Die Funktion $\langle \text{STA} \rangle$ implementiert die Schrittfunktion eines \mathcal{P} -Programms

```
procedure P( $\mathbf{x}_1, \dots, \mathbf{x}_k$ ) <=
begin var  $\mathbf{y}_1, \dots, \mathbf{y}_n$ ; STA; return( $\mathbf{y}$ ) end
```

d.h. es gilt $\langle \text{STA} \rangle(b, \bar{\mathbf{n}}) = \llbracket P^{STEP} \rrbracket(b, \bar{\mathbf{n}})$, vgl. Definition 8.4. Da diese Funktion die Anzahl nicht benötigter Ausführungsschritte angibt, erhält man:

Lemma 11.37. Für jede Programmanweisung STA gilt $b \geq \langle \text{STA} \rangle(b, \bar{\mathbf{n}})$ für alle $b \in \mathbb{N}$ und alle $\bar{\mathbf{n}} \in \mathbb{N}^k$.

Beweis. Beweis durch strukturelle Induktion über STA. \blacksquare

Der Nachweis, daß die Funktionen $\Downarrow \text{STA}(\mathbf{y}_i) \Downarrow$ primitiv-rekursiv sind, folgt im wesentlichen dem Beweis, daß die Funktionen $\llbracket \text{STA}(\mathbf{y}_i) \rrbracket$ für loop-Programme primitiv-rekursiv sind (siehe Satz 11.20). Um jedoch der Beweisidee auch für allgemeine while-Anweisungen zu folgen, müssen diese zuvor in loop-Anweisungen umgeformt werden: Man ersetzt dabei eine while-Anweisung

STA = while EXPR do STA' end_while

durch das Programmfragment

```
y := b;  
loop y do  
  if EXPR then STA'; b := PRED(b) end_if  
end_loop (11.22)
```

um die Berechnung der nach Ausführung der while-Anweisung verbleibenden Schrittzahl $\Downarrow \text{STA} \Downarrow (b, \vec{n})$ anhand der Ausführung der loop-Anweisung in (11.22) zu bestimmen. Dies gelingt, da loop-Anweisungen immer terminieren. Stehen b Schritte vor Abarbeitung der loop-Anweisung zur Verfügung, so gilt $\Downarrow \text{STA} \Downarrow (b, \vec{n}) = b - w$ nach Abarbeitung, wobei w den Laufzeitbedarf bei Ausführung der while-Anweisung angibt, also die Kosten, die durch die Ausführungen des Schleifenrumpfs STA' anfallen, plus die Anzahl n_0 der Ausführungen des Schleifenrumpfs (siehe Definition 8.1). Damit erhält man $\Downarrow \text{STA} \Downarrow (b, \vec{n}) = 0$, falls die zur Abarbeitung der while-Schleife anfallenden Laufzeitkosten größer oder gleich b sind (was insbesondere dann der Fall ist, wenn die Abarbeitung der while-Schleife nicht terminiert).

Im Beweis des folgenden Satzes 11.38 wird die Berechnung von $\Downarrow \text{STA} \Downarrow (b, \vec{n})$ durch die primitiv-rekursive Funktion g modelliert (siehe die Gleichungen (11.26) im Beweis). Damit dieser Ansatz zur Berechnung von $\Downarrow \text{STA} \Downarrow (b, \vec{n})$ korrekt ist, muß die Laufvariable \mathbf{y} in (11.22) mit einem hinreichend großen Wert initialisiert werden. Es reicht dabei \mathbf{y} mit b zu initialisieren – dies ist die Aussage von Gleichung (11.27) im Beweis – denn im Fall $b > w$ ist die Initialisierung $\mathbf{y} := b$ unkritisch, da im Rumpf der loop-Anweisung keine Anweisungen mehr ausgeführt werden, sobald der Schleifentest EXPR nach n_0 Ausführungen des Schleifenrumpfs STA' scheitert. Im Fall $b \leq w$ reicht die Initialisierung $\mathbf{y} := b$ ebenfalls, da man nach b Ausführungen des Rumpfs der loop-Anweisung das korrekte Ergebnis $\Downarrow \text{STA} \Downarrow (b, \vec{n}) = 0$ erhält. Da $\Downarrow \text{STA} \Downarrow$ durch Einsetzung mittels der primitiv-rekursiven Funktion g definiert werden kann – Gleichung (11.29) im Beweis – ist auch gezeigt, daß $\Downarrow \text{STA} \Downarrow$ primitiv-rekursiv ist.

Die Berechnung der Werte $\Downarrow \text{STA}(\mathbf{y}_i) \Downarrow (b, \vec{n})$ für die lokalen Variablen \mathbf{y}_i wird ebenfalls mittels der loop-Anweisung (11.22) durchgeführt. Wie im Beweis von

Satz 11.20 für $\mathcal{P}_{\text{loop}}$ -Programme werden primitiv-rekursive Funktionen f_i durch gegenseitige Rekursion definiert, die jetzt jedoch einen zusätzlichen Parameter b für die zur Verfügung stehende Schrittzahl besitzen. Mit dem gleichen Argument wie zuvor gilt auch hier, daß die Initialisierung der Laufvariablen \mathbf{y} mit b ausreicht, um $\Downarrow\text{STA}(\mathbf{y}_i)\langle(b, \vec{n})$ zu bestimmen, denn ein Mißerfolg wird nach b Schritten erkannt (Fall $b \leq w$) und überflüssige Abarbeitungen des loop-Schleifenrumpfs (Fall $b > w$) sind unkritisch, da keine Anweisungen mehr ausgeführt werden. Dies ist die Aussage von Gleichung (11.32) im Beweis. Da $\Downarrow\text{STA}(\mathbf{y}_i)\langle$ durch Einsetzung mittels der primitiv-rekursiven Funktion f_i angegeben werden kann, ist auch $\Downarrow\text{STA}(\mathbf{y}_i)\langle$ primitiv-rekursiv.

Satz 11.38. Für jede Anweisung STA eines \mathcal{P} -Programms und jede lokale Variable \mathbf{y} in STA sind die Funktionen $\Downarrow\text{STA}\langle$ und $\Downarrow\text{STA}(\mathbf{y})\langle$ primitiv-rekursiv.

Beweis. Seien $\mathbf{y}_1, \dots, \mathbf{y}_k$ die lokalen Variablen und $\mathbf{y}_{k+1}, \dots, \mathbf{y}_{k+l}$ die formalen Parameter von STA, und sei $M_P^{\vec{n}} \subseteq \mathbb{N} \times \mathbb{N}$ mit $\vec{n} = (n_1, \dots, n_{k+l})$, so daß $\text{value}(M_P^{\vec{n}}, \mathbf{y}_j) = n_j$ für jedes $j \in \{1, \dots, k+l\}$. Wir zeigen die Behauptung für jedes \mathbf{y}_i mit $i \in \{1, \dots, k\}$ durch strukturelle Induktion über STA (unter häufiger Verwendung von Definitionen 2.7 und 8.1).

Induktionsanfang “STA = SKIP oder STA = $\mathbf{y}_h := \text{EXPR}$ ”: Es gilt $\text{step}(b, M_P^{\vec{n}}, \text{STA}) = b$, und damit ist $\Downarrow\text{STA}\langle$ definiert durch “ $\Downarrow\text{STA}\langle(y, \vec{z}) := y$ ” primitiv-rekursiv. Mit Definition 11.36 gilt

$$\Downarrow\text{STA}(\mathbf{y}_i)\langle(y, \vec{z}) = \text{if}(y, \llbracket\text{STA}(\mathbf{y}_i)\rrbracket(\vec{z}), 0)$$

und somit ist auch $\Downarrow\text{STA}(\mathbf{y}_i)\langle$ primitiv-rekursiv, denn $\llbracket\text{STA}(\mathbf{y}_i)\rrbracket$ ist – wie im Beweis von Satz 11.20 gezeigt – in diesem Fall primitiv-rekursiv.

Induktionsschritt “STA = if EXPR then STA1 else STA2 end_if”: Für $b = 0$ gilt $\Downarrow\text{STA}\langle(b, \vec{n}) = 0$. Andernfalls erhält man

$$\Downarrow\text{STA}\langle(b, \vec{n}) = \begin{cases} \Downarrow\text{STA1}\langle(b, \vec{n}), & \text{falls } \llbracket\text{EXPR}\rrbracket(\vec{n}) > 0 \\ \Downarrow\text{STA2}\langle(b, \vec{n}), & \text{falls } \llbracket\text{EXPR}\rrbracket(\vec{n}) = 0 \end{cases}$$

und damit

$$\Downarrow\text{STA}\langle(y, \vec{z}) = \text{if}(y, \text{if}(\llbracket\text{EXPR}\rrbracket(\vec{z}), \Downarrow\text{STA1}\langle(y, \vec{z}), \Downarrow\text{STA2}\langle(y, \vec{z})), 0). \quad (11.23)$$

$\llbracket\text{EXPR}\rrbracket$ ist nach Satz 11.19 primitiv-rekursiv, die Funktionen $\Downarrow\text{STA1}\langle$ und $\Downarrow\text{STA2}\langle$ sind nach Induktionsvoraussetzung primitiv-rekursiv, und somit ist auch $\Downarrow\text{STA}\langle$ primitiv-rekursiv.

Weiter gilt $\Downarrow \text{STA}(\bar{y}_i) \Downarrow (b, \bar{n}) = 0$, falls $\Downarrow \text{STA} \Downarrow (b, \bar{n}) = 0$, und sonst $\Downarrow \text{STA}(\bar{y}_i) \Downarrow (b, \bar{n}) = \llbracket \text{STA}(\bar{y}_i) \rrbracket(\bar{n})$. Für $\llbracket \text{EXPR} \rrbracket(\bar{n}) > 0$ erhält man $\Downarrow \text{STA} \Downarrow (b, \bar{n}) = \Downarrow \text{STA1} \Downarrow (b, \bar{n}) > 0$, also $\Downarrow \text{STA1}(\bar{y}_i) \Downarrow (b, \bar{n}) = \llbracket \text{STA1}(\bar{y}_i) \rrbracket(\bar{n}) = \llbracket \text{STA}(\bar{y}_i) \rrbracket(\bar{n})$. Genauso erhält man $\llbracket \text{STA}(\bar{y}_i) \rrbracket(\bar{n}) = \Downarrow \text{STA2}(\bar{y}_i) \Downarrow (b, \bar{n})$ im Fall $\llbracket \text{EXPR} \rrbracket(\bar{n}) = 0$. Damit gilt

$$\Downarrow \text{STA}(\bar{y}_i) \Downarrow (y, \bar{z}) = \text{if}(\Downarrow \text{STA} \Downarrow (y, \bar{z}), \\ \text{if}(\llbracket \text{EXPR} \rrbracket(\bar{z}), \Downarrow \text{STA1}(\bar{y}_i) \Downarrow (y, \bar{z}), \Downarrow \text{STA2}(\bar{y}_i) \Downarrow (y, \bar{z})), \\ 0).$$

und somit ist $\Downarrow \text{STA}(\bar{y}_i) \Downarrow$ primitiv-rekursiv, denn $\Downarrow \text{STA} \Downarrow$ ist mit (11.23), $\llbracket \text{EXPR} \rrbracket$ ist nach Satz 11.19 und $\Downarrow \text{STA1}(\bar{y}_i) \Downarrow$ und $\Downarrow \text{STA2}(\bar{y}_i) \Downarrow$ sind nach Induktionsvoraussetzung primitiv-rekursiv.

Induktionsschritt “ $\text{STA} = \text{STA1}; \text{STA2}$ ”: Für $b = 0$ erhält man $\Downarrow \text{STA} \Downarrow (b, \bar{n}) = 0$. Andernfalls gilt $\Downarrow \text{STA} \Downarrow (b, \bar{n}) = \Downarrow \text{STA2} \Downarrow (\Downarrow \text{STA1} \Downarrow (b, \bar{n}), \llbracket \text{STA1}(\bar{y}_1, \dots, \bar{y}_k) \rrbracket(\bar{n}))$.¹⁸

Für $\Downarrow \text{STA1} \Downarrow (b, \bar{n}) = 0$ erhält man $\Downarrow \text{STA} \Downarrow (b, \bar{n}) = \Downarrow \text{STA2} \Downarrow (0, \dots) = 0$. Andernfalls gilt $\Downarrow \text{STA1}(\bar{y}_1, \dots, \bar{y}_k) \Downarrow (b, \bar{n}) = \llbracket \text{STA1}(\bar{y}_1, \dots, \bar{y}_k) \rrbracket(\bar{n})$ und somit

$$\Downarrow \text{STA} \Downarrow (y, \bar{z}) = \text{if}(y, \\ \text{if}(\Downarrow \text{STA1} \Downarrow (y, \bar{z}), \\ \Downarrow \text{STA2} \Downarrow (\Downarrow \text{STA1} \Downarrow (y, \bar{z}), \Downarrow \text{STA1}(\bar{y}_1, \dots, \bar{y}_k) \Downarrow (y, \bar{z})), \\ 0) \\ 0). \quad (11.24)$$

Nach Induktionsvoraussetzung sind die Funktionen $\Downarrow \text{STA1} \Downarrow$, $\Downarrow \text{STA2} \Downarrow$ sowie $\Downarrow \text{STA1}(\bar{y}_i) \Downarrow$ primitiv-rekursiv, und somit ist auch $\Downarrow \text{STA} \Downarrow$ primitiv-rekursiv.

Für $\Downarrow \text{STA} \Downarrow (b, \bar{n}) = 0$ gilt $\Downarrow \text{STA}(\bar{y}_i) \Downarrow (b, \bar{n}) = 0$, und andernfalls mit (11.24) sowohl (i) $\Downarrow \text{STA1} \Downarrow (b, \bar{n}) > 0$ als auch (ii) $\Downarrow \text{STA2} \Downarrow (\Downarrow \text{STA1} \Downarrow (b, \bar{n}), \Downarrow \text{STA1}(\bar{y}_1, \dots, \bar{y}_k) \Downarrow (b, \bar{n})) > 0$. Man erhält

$$\begin{aligned} & \Downarrow \text{STA}(\bar{y}_i) \Downarrow (b, \bar{n}) \\ = & \llbracket \text{STA}(\bar{y}_i) \rrbracket(\bar{n}) && \text{, mit Def. 11.36} \\ = & \llbracket \text{STA2}(\bar{y}_i) \rrbracket(\llbracket \text{STA1}(\bar{y}_1, \dots, \bar{y}_k) \rrbracket(\bar{n})) && \text{, mit Def. 2.7} \\ = & \llbracket \text{STA2}(\bar{y}_i) \rrbracket(\Downarrow \text{STA1}(\bar{y}_1, \dots, \bar{y}_k) \Downarrow (b, \bar{n})) && \text{, mit (i) \& Def. 11.36} \\ = & \Downarrow \text{STA2}(\bar{y}_i) \Downarrow (\Downarrow \text{STA1} \Downarrow (b, \bar{n}), \Downarrow \text{STA1}(\bar{y}_1, \dots, \bar{y}_k) \Downarrow (b, \bar{n})) && \text{, mit (ii) \& Def. 11.36.} \end{aligned}$$

Damit gilt

$$\Downarrow \text{STA}(\bar{y}_i) \Downarrow (y, \bar{z}) = \text{if}(\Downarrow \text{STA} \Downarrow (y, \bar{z}), \\ \Downarrow \text{STA2}(\bar{y}_i) \Downarrow (\Downarrow \text{STA1} \Downarrow (y, \bar{z}), \Downarrow \text{STA1}(\bar{y}_1, \dots, \bar{y}_k) \Downarrow (y, \bar{z})), \\ 0)$$

¹⁸ $\llbracket \text{STA}(\bar{y}_1, \dots, \bar{y}_k) \rrbracket(\bar{z})$ steht für das $k+l$ -Tupel $\llbracket \text{STA}(\bar{y}_1) \rrbracket(\bar{z}), \dots, \llbracket \text{STA}(\bar{y}_k) \rrbracket(\bar{z}), z_{k+1}, \dots, z_{k+l}$ und $\Downarrow \text{STA}(\bar{y}_1, \dots, \bar{y}_k) \Downarrow (y, \bar{z})$ für das $k+l$ -Tupel $\Downarrow \text{STA}(\bar{y}_1) \Downarrow (y, \bar{z}), \dots, \Downarrow \text{STA}(\bar{y}_k) \Downarrow (y, \bar{z}), z_{k+1}, \dots, z_{k+l}$.

und somit ist $\Downarrow\text{STA}(\mathbf{y}_i)\Downarrow$ primitiv-rekursiv, denn $\Downarrow\text{STA}\Downarrow$ ist mit (11.24) primitiv-rekursiv, und die Funktionen $\Downarrow\text{STA1}\Downarrow$, $\Downarrow\text{STA1}(\mathbf{y}_i)\Downarrow$ und $\Downarrow\text{STA2}(\mathbf{y}_i)\Downarrow$ sind nach Induktionsvoraussetzung primitiv-rekursiv.

Induktionsschritt “ $\text{STA} = \text{while EXPR do STA' end_while}$ ”: Es gilt mit Definitionen 8.1 und 11.36

$$\begin{aligned} \Downarrow\text{STA}\Downarrow(0, \vec{n}) &= 0 \\ \Downarrow\text{STA}\Downarrow(b+1, \vec{n}) &= \begin{cases} b+1, & \text{falls } \llbracket\text{EXPR}\rrbracket(\vec{n}) = 0 \\ \Downarrow\text{STA}\Downarrow(\Downarrow\text{STA}'\Downarrow(b, \vec{n}), \Downarrow\text{STA}'(\mathbf{y}_1, \dots, \mathbf{y}_k)\Downarrow(b, \vec{n})), & \text{sonst,} \end{cases} \end{aligned} \quad (11.25)$$

denn $\llbracket\text{STA}'(\mathbf{y}_1, \dots, \mathbf{y}_k)\rrbracket(\vec{n}) = \Downarrow\text{STA}'(\mathbf{y}_1, \dots, \mathbf{y}_k)\Downarrow(b, \vec{n})$ wenn $\Downarrow\text{STA}'\Downarrow(b, \vec{n}) > 0$.

Wir definieren eine Funktion $g : \mathbb{N}^{k+l+2} \rightarrow \mathbb{N}$ durch

$$\begin{aligned} g(0, b, \vec{n}) &= b \\ g(n+1, b, \vec{n}) &= \text{if}(\llbracket\text{EXPR}\rrbracket(\vec{n}), \\ &\quad \text{if}(\Downarrow\text{STA}'\Downarrow(\text{pred}(b), \vec{n}), \\ &\quad \quad g(n, \Downarrow\text{STA}'\Downarrow(\text{pred}(b), \vec{n}), \Downarrow\text{STA}'(\mathbf{y}_1, \dots, \mathbf{y}_k)\Downarrow(\text{pred}(b), \vec{n})), \\ &\quad \quad 0), \\ &\quad b) \end{aligned} \quad (11.26)$$

und zeigen zunächst

$$n \geq b \curvearrowright g(n, b, \vec{n}) = g(b, b, \vec{n}) \quad (11.27)$$

für alle $n, b \in \mathbb{N}$ und alle $\vec{n} \in \mathbb{N}^{k+l}$ durch Induktion über n .

Induktionsanfang “ $n = 0$ ”: Mit $n \geq b$ gilt $n = b$ und damit die Behauptung.

Induktionsschritt “ $n = n' + 1$ ”: Für $b = 0$ gilt $\Downarrow\text{STA}'\Downarrow(\text{pred}(b), \vec{n}) = \Downarrow\text{STA}'\Downarrow(0, \vec{n}) = 0$ und man erhält damit $g(n' + 1, 0, \vec{n}) = \text{if}(\llbracket\text{EXPR}\rrbracket(\vec{n}), 0, 0) = 0 = g(0, 0, \vec{n})$.

Für $b = b' + 1$ gilt nach Voraussetzung $n' + 1 \geq b' + 1$, also $n > n' \geq b' \geq \Downarrow\text{STA}'\Downarrow(b', \vec{n})$ (mit Lemma 11.37). Als Induktionshypothesen erhält man folglich

$$n' \geq b^* \curvearrowright g(n', b^*, \vec{r}) = g(b^*, b^*, \vec{r})$$

sowie

$$b' \geq b^* \curvearrowright g(b', b^*, \vec{r}) = g(b^*, b^*, \vec{r})$$

für alle $b^* \in \mathbb{N}$ und $\vec{r} \in \mathbb{N}^{k+l}$, und für $\vec{m} := \Downarrow\text{STA}'(\mathbf{y}_1, \dots, \mathbf{y}_k)\Downarrow(b', \vec{n})$ damit dann insbesondere

$$g(n', \Downarrow\text{STA}'\Downarrow(b', \vec{n}), \vec{m}) = g(b', \Downarrow\text{STA}'\Downarrow(b', \vec{n}), \vec{m}). \quad (11.28)$$

Im Fall $\llbracket \text{EXPR} \rrbracket(\vec{n}) > 0$ erhält man

$$\begin{aligned}
 & g(n' + 1, b' + 1, \vec{n}) \\
 = & \text{if}(\Downarrow \text{STA}'\langle (b', \vec{n}), g(n', \Downarrow \text{STA}'\langle (b', \vec{n}), \vec{m} \rangle), 0 \rangle, \text{mit Def. } g) \\
 = & \text{if}(\Downarrow \text{STA}'\langle (b', \vec{n}), g(b', \Downarrow \text{STA}'\langle (b', \vec{n}), \vec{m} \rangle), 0 \rangle, \text{mit (11.28)}) \\
 = & g(b' + 1, b' + 1, \vec{n}) \quad \text{, mit Def. } g
 \end{aligned}$$

und andernfalls $g(n' + 1, b' + 1, \vec{n}) = b' + 1 = g(b' + 1, b' + 1, \vec{n})$.

Jetzt zeigen wir

$$\Downarrow \text{STA}\langle (b, \vec{n}) = g(b, b, \vec{n}) \quad (11.29)$$

für alle $b \in \mathbb{N}$ und alle $\vec{n} \in \mathbb{N}^{k+l}$ durch Induktion über b .

Induktionsanfang “ $b = 0$ ”: Es gilt $\Downarrow \text{STA}\langle (0, \vec{n}) = 0 = g(0, 0, \vec{n})$.

Induktionsschritt “ $b = b' + 1$ ”: Es gilt $b' + 1 > b' \geq \Downarrow \text{STA}'\langle (b', \vec{n})$ (mit Lemma 11.37) und als Induktionsvoraussetzung dann

$$\Downarrow \text{STA}\langle (\Downarrow \text{STA}'\langle (b', \vec{n}), \vec{r} \rangle = g(\Downarrow \text{STA}'\langle (b', \vec{n}), \Downarrow \text{STA}'\langle (b', \vec{n}), \vec{r} \rangle), \vec{r} \rangle \quad (11.30)$$

für beliebige $\vec{r} \in \mathbb{N}^{k+l}$.

Für $\llbracket \text{EXPR} \rrbracket(\vec{n}) = 0$ gilt $\Downarrow \text{STA}\langle (b' + 1, \vec{n}) = b' + 1 = g(b' + 1, b' + 1, \vec{n})$, und im Fall $\llbracket \text{EXPR} \rrbracket(\vec{n}) > 0$ und $\Downarrow \text{STA}'\langle (b', \vec{n}) = 0$ erhält man

$$\begin{aligned}
 & \Downarrow \text{STA}\langle (b' + 1, \vec{n}) \\
 = & \Downarrow \text{STA}\langle (\Downarrow \text{STA}'\langle (b', \vec{n}), \dots \rangle, \dots \rangle \quad \text{, mit (11.25)} \\
 = & \Downarrow \text{STA}\langle (0, \dots \rangle \quad \text{, denn } \Downarrow \text{STA}'\langle (b', \vec{n}) = 0 \\
 = & 0 \quad \text{, mit Definitionen. 8.1 und 11.36} \\
 = & \text{if}(0, \dots, 0) \\
 = & \text{if}(\Downarrow \text{STA}'\langle (b', \vec{n}), \dots, 0 \rangle, \text{denn } \Downarrow \text{STA}'\langle (b', \vec{n}) = 0 \\
 = & g(b' + 1, b' + 1, \vec{n}) \quad \text{, mit Def. } g .
 \end{aligned}$$

Andernfalls gilt

$$\begin{aligned}
 & \Downarrow \text{STA}\langle (b' + 1, \vec{n}) \\
 = & \Downarrow \text{STA}\langle (\Downarrow \text{STA}'\langle (b', \vec{n}), \Downarrow \text{STA}'(\vec{y}_1, \dots, \vec{y}_k)\langle (b', \vec{n}) \rangle), \dots \rangle \quad \text{, mit (11.25)} \\
 = & g(\Downarrow \text{STA}'\langle (b', \vec{n}), \Downarrow \text{STA}'\langle (b', \vec{n}), \Downarrow \text{STA}'(\vec{y}_1, \dots, \vec{y}_k)\langle (b', \vec{n}) \rangle), \dots \rangle \quad \text{, mit (11.30)} \\
 = & g(b', \Downarrow \text{STA}'\langle (b', \vec{n}), \Downarrow \text{STA}'(\vec{y}_1, \dots, \vec{y}_k)\langle (b', \vec{n}) \rangle), \dots \rangle \quad \text{, mit (11.27)} \\
 = & g(b' + 1, b' + 1, \vec{n}) \quad \text{, mit Def. } g .
 \end{aligned}$$

Mit (11.26) ist g primitiv-rekursiv, denn $\llbracket \text{EXPR} \rrbracket$ ist nach Satz 11.19 primitiv-rekursiv, und die Funktionen $\langle \text{STA}' \rangle$ und $\langle \text{STA}'(\mathbf{y}_i) \rangle$ sind nach Induktionsvoraussetzung primitiv-rekursiv.¹⁹ Mit (11.29) ist dann auch $\langle \text{STA} \rangle$ primitiv-rekursiv.

Zur Angabe von $\langle \text{STA}(\mathbf{y}_i) \rangle$ definieren wir jetzt für jedes $i \in \{1, \dots, k\}$ eine Funktion $f_i : \mathbb{N}^{k+l+2} \mapsto \mathbb{N}$ durch

$$f_i(n, b, \vec{n}) := \begin{cases} 0, & \text{falls } \langle \text{STA} \rangle(b, \vec{n}) = 0 \\ \text{value}(\text{eval}^{(n)}(M_P^{\vec{n}}, \text{if EXPR then STA' end_if}), \mathbf{y}_i), & \text{sonst.} \end{cases} \quad (11.31)$$

Unter der Annahme (i) " $\langle \text{STA} \rangle(b, \vec{n}) > 0$ " gilt (ii) $\langle \text{STA}(\mathbf{y}_i) \rangle(b, \vec{n}) = \text{value}(\text{eval}^{(n_0)}(M_P^{\vec{n}}, \text{STA}'), \mathbf{y}_i)$ für ein $n_0 \in \mathbb{N}$, d.h. der Schleifenrumpf STA' wird genau n_0 mal ausgeführt. Für $h \in \{1, \dots, n_0\}$ seien w_h die Ausführungskosten der h -ten Ausführung von STA' , und w bezeichne die Ausführungskosten von STA . Dann ergibt sich w mit Definition 8.1 als Summe der Kosten für die n_0 Ausführungen des Schleifenrumpfs plus n_0 für n_0 Ausführungen des Schleifenrumpfs, also $w = w_1 + \dots + w_{n_0} + n_0$. Es gilt $\langle \text{STA} \rangle(b, \vec{n}) = b - w$ und mit (i) dann (iii) $b > w \geq n_0$. Daraus folgt

$$\langle \text{STA}(\mathbf{y}_i) \rangle(b, \vec{n}) = f_i(b, b, \vec{n}). \quad (11.32)$$

denn es gilt

$$n' \geq n_0 \quad \text{gdw.} \quad (11.33)$$

$$\text{value}(\text{eval}^{(n')}(M_P^{\vec{n}}, \text{if EXPR then STA' end_if}), \text{EXPR}) = 0$$

für alle $n' \in \mathbb{N}$ mit Definition von n_0 , und mit Definition 2.7 dann

$$\begin{aligned} & \langle \text{STA}(\mathbf{y}_i) \rangle(b, \vec{n}) \\ = & \text{value}(\text{eval}^{(n_0)}(M_P^{\vec{n}}, \text{STA}'), \mathbf{y}_i) \quad , \text{ mit (ii)} \\ = & \text{value}(\text{eval}^{(b-n_0)}(\text{eval}^{(n_0)}(M_P^{\vec{n}}, \text{STA}'), \text{SKIP}), \mathbf{y}_i) \\ = & \text{value}(\text{eval}^{(b)}(M_P^{\vec{n}}, \text{if EXPR then STA' end_if}), \mathbf{y}_i) \quad , \text{ mit (iii) und (11.33)} \\ = & f_i(b, b, \vec{n}) \quad , \text{ mit (11.31)}. \end{aligned}$$

¹⁹ g ist mit (11.26) durch eine *tail-rekursive* Definition mit *struktureller Rekursion* gegeben. Solche Funktionen können uniform in äquivalente `loop`-Programme umgeformt werden, woraus mit Korollar 11.21 folgt, daß solchermaßen definierte Funktionen primitiv-rekursiv sind.

Man erhält

$$f_i(0, y, \vec{z}) = \text{if}(\Downarrow \text{STA} \langle (y, \vec{z}), P_{i+1}^{k+l+1}(y, \vec{z}), 0 \rangle) \quad (11.34)$$

denn für $\Downarrow \text{STA} \langle (b, \vec{n}) \rangle = 0$ gilt $f_i(0, b, \vec{n}) = 0$, und andernfalls erhält man

$$\begin{aligned} & f_i(0, b, \vec{n}) \\ = & \text{value}(\text{eval}^{(0)}(M_P^{\vec{n}}, \text{if EXPR then STA}' \text{end_if}), y_i) \quad , \text{ mit (11.31)} \\ = & \text{value}(M_P^{\vec{n}}, y_i) \quad , \text{ mit Def. } \text{eval}^{(n)} \\ = & n_i \\ = & P_{i+1}^{k+l+1}(b, \vec{n}) . \end{aligned}$$

Des weiteren gilt

$$\begin{aligned} f_i(x+1, y, \vec{z}) = \text{if}(\Downarrow \text{STA} \langle (y, \vec{z}), \\ \text{if}(\llbracket \text{EXPR} \rrbracket(\vec{f}(x, y, \vec{z})), \\ \Downarrow \text{STA}'(y_i) \langle (y, \vec{f}(x, y, \vec{z})), \\ f_i(x, y, \vec{z}) \rangle, \\ 0) \end{aligned} \quad (11.35)$$

denn für $\Downarrow \text{STA} \langle (b, \vec{n}) \rangle = 0$ erhält man $f_i(n+1, b, \vec{n}) = 0$.²⁰

Andernfalls gilt $\Downarrow \text{STA} \langle (b, \vec{n}) \rangle > 0$ und im Fall (*) $\llbracket \text{EXPR} \rrbracket(\vec{f}(n, b, \vec{n})) > 0$ wegen (11.33) dann $n < n_0$. Für

$$M_P^{\vec{f}(n, b, \vec{n})} := \text{eval}^{(n)}(M_P^{\vec{n}}, \text{if EXPR then STA}' \text{end_if}) \quad (11.36)$$

und

$$b' := \text{step}^{(n)}(b, M_P^{\vec{n}}, \text{if EXPR then STA}' \text{end_if}) \quad (11.37)$$

erhält man^{21,22}

$$\Downarrow \text{if EXPR then STA}' \text{end_if} \langle (b', \vec{f}(n, b, \vec{n})) \rangle > 0. \quad (11.38)$$

²⁰ Wir schreiben $\vec{f}(x, y, \vec{z})$ als Abkürzung für das $k+l$ -Tupel $f_1(x, y, \vec{z}), \dots, f_k(x, y, \vec{z}), z_{k+1}, \dots, z_{k+l}$ und $\vec{f}(n, b, \vec{n})$ als Abkürzung für das $k+l$ -Tupel $f_1(n, b, \vec{n}), \dots, f_k(n, b, \vec{n}), n_{k+1}, \dots, n_{k+l}$.

²¹ Es gelte $\text{step}^{(0)}(b, M_P, \text{STA}) := b$ und $\text{step}^{(n+1)}(b, M_P, \text{STA}) := \text{step}(\text{step}^{(n)}(b, M_P, \text{STA}), \text{eval}^{(n)}(M_P, \text{STA}), \text{STA})$.

²² $\Downarrow \text{STA} \langle (b, \vec{n}) \rangle$ gibt die nach Ausführung der while-Schleife verbleibenden Schritte an. $\Downarrow \text{if EXPR then STA}' \text{end_if} \langle (b', \vec{f}(n, b, \vec{n})) \rangle$ gibt die Schritte an, die nach $n+1$ -maliger Ausführung der if-Anweisung verbleiben (mit (11.37) entspricht b' den verbleibenden Schritten nach n

Mit $\llbracket \text{EXPR} \rrbracket(\vec{f}(n, b, \vec{n})) > 0$ gilt

$$\Downarrow \text{if } \text{EXPR} \text{ then } \text{STA}' \text{ end_if} \Downarrow (b', \vec{f}(n, b, \vec{n})) = \Downarrow \text{STA}' \Downarrow (b', \vec{f}(n, b, \vec{n}))$$

und mit (11.38) dann $\Downarrow \text{STA}' \Downarrow (b', \vec{f}(n, b, \vec{n})) > 0$. Mit Lemma 11.37 gilt $b \geq b'$, und mit Korollar 8.3 erhält man

$$\Downarrow \text{STA}' \Downarrow (b, \vec{f}(n, b, \vec{n})) > 0. \quad (11.39)$$

Damit gilt:

$$\begin{aligned} & f_i(n+1, b, \vec{n}) \\ = & \text{value}(\text{eval}^{(n+1)}(M_P^{\vec{n}}, \text{if } \text{EXPR} \text{ then } \text{STA}' \text{ end_if}, y_i)) \quad , \text{ mit (11.31)} \\ = & \text{value}(\text{eval}(M_P^{\vec{f}(n, b, \vec{n})}, \text{if } \text{EXPR} \text{ then } \text{STA}' \text{ end_if}, y_i)) \quad , \text{ mit (11.36)} \\ = & \llbracket \text{if } \text{EXPR} \text{ then } \text{STA}' \text{ end_if}(y_i) \rrbracket(\vec{f}(n, b, \vec{n})) \quad , \text{ mit Def. 11.18} \\ = & \llbracket \text{STA}'(y_i) \rrbracket(\vec{f}(n, b, \vec{n})) \quad , \text{ mit (*)} \\ = & \Downarrow \text{STA}' \Downarrow (y_i) \Downarrow (b, \vec{f}(n, b, \vec{n})) \quad , \text{ mit (11.39)} \\ & \text{und Definition 11.36.} \end{aligned}$$

Im Fall (**) $\llbracket \text{EXPR} \rrbracket(\vec{f}(n, b, \vec{n})) = 0$ gilt $n \geq n_0$ wegen (11.33), und man erhält damit

$$\begin{aligned} & f_i(n+1, b, \vec{n}) \\ = & \text{value}(\text{eval}^{(n+1)}(M_P^{\vec{n}}, \text{if } \text{EXPR} \text{ then } \text{STA}' \text{ end_if}, y_i)) \quad , \text{ mit (11.31)} \\ = & \text{value}(\text{eval}^{(n_0)}(M_P^{\vec{n}}, \text{if } \text{EXPR} \text{ then } \text{STA}' \text{ end_if}, y_i)) \quad , \text{ mit (11.33)} \\ = & f_i(n_0, b, \vec{n}) \quad , \text{ mit (11.31)} \\ = & f_i(n, b, \vec{n}) \quad , \text{ mit (11.33).} \end{aligned}$$

Wegen (11.34) und (11.35) sind alle Funktionen f_i nach Satz 11.9 primitiv-rekursiv, denn $\llbracket \text{EXPR} \rrbracket$ ist nach Satz 11.19 primitiv-rekursiv, $\Downarrow \text{STA}' \Downarrow$ ist mit (11.29) primitiv-rekursiv, und $\Downarrow \text{STA}' \Downarrow (y_i) \Downarrow$ ist nach Induktionsvoraussetzung primitiv-rekursiv. Mit (11.32) ist dann auch $\llbracket \text{STA}'(y_i) \rrbracket$ primitiv-rekursiv. ■

Eine bemerkenswerte Konsequenz von Satz 11.38 ist, daß die beschränkte Interpretation von \mathcal{P} -Programmen allein durch ein loop-Programm implementiert werden kann.

Ausführungen, abzüglich der Kosten für eine weitere Ausführung). Mit $n < n_0$ gilt $n+1 \leq n_0$, d.h. $\Downarrow \text{if } \text{EXPR} \text{ then } \text{STA}' \text{ end_if} \Downarrow (b', \vec{f}(n, b, \vec{n}))$ entspricht höchstens den verbleibenden Schritten nach n_0 -maliger Ausführung des while-Schleifenrumpfs abzüglich der Kosten für die n_0 Ausführungen des Schleifenrumpfs. Damit gilt $\Downarrow \text{if } \text{EXPR} \text{ then } \text{STA}' \text{ end_if} \Downarrow (b', \vec{f}(n, b, \vec{n})) \geq \Downarrow \text{STA}' \Downarrow (b, \vec{n})$, und mit $\Downarrow \text{STA}' \Downarrow (b, \vec{n}) > 0$ folgt dann (11.38).

Aus Kapitel 8 wissen wir, daß durch beschränkte Ausführung auch die *unbeschränkte* Ausführung von \mathcal{P} -Programmen realisiert werden kann, indem \mathcal{P} -Programme mittels schrittweise erhöhter Laufzeitbeschränkung ausgeführt werden. Gelingt die Ausführung bei unbeschränkter Interpretation, so gelingt sie auch bei beschränkter Interpretation sobald die erforderliche Mindestlaufzeitbeschränkung erreicht ist, vgl. Satz 8.2(1). Andernfalls terminiert das Verfahren nicht.

Im Beweis des folgenden Satzes 11.39 modellieren wir die schrittweise Erhöhung der Laufzeitbeschränkung durch den μ -Operator, mit dem ja eine "Hochzähl-schleife" zur Verfügung steht: Für eine aktuelle Laufzeitbeschränkung b entscheiden wir mittels $\Downarrow\text{STA}\Downarrow$, ob b groß genug ist, um STA auszuführen. Im positiven Fall berechnen wir das Ergebnis mit $\llbracket\text{STA}(y)\rrbracket$. Andernfalls wird die Laufzeitbeschränkung auf $b+1$ erhöht, und der nächste Ausführungsversuch begonnen. Diese Überlegung liegt dem Beweis zu Grunde, daß jede \mathcal{P} -berechenbare Funktion auch μ -rekursiv ist:

Satz 11.39. *Jede \mathcal{P} -berechenbare Funktion ist μ -rekursiv.*

Beweis. *Sei*

```
procedure P( $x_1, \dots, x_k$ )  $\leq$ =
begin var  $y_1, \dots, y_m$ ; STA; return( $y_r$ ) end
```

ein \mathcal{P} -Programm, sei $M_P^{\vec{n}} \subseteq \mathbb{N} \times \mathbb{N}$ mit $\vec{n} := (n_1, \dots, n_k)$ und $\text{value}(M_P^{\vec{n}}, x_i) = n_i$ für jedes $i \in \{1, \dots, k\}$ sowie $\text{value}(M_P^{\vec{n}}, y_j) = 0$ für jedes $j \in \{1, \dots, m\}$. Dann gilt $\llbracket P \rrbracket(\vec{n}) = \text{value}(\text{eval}(M_P^{\vec{n}}, \text{STA}), y_r) = \llbracket \text{STA}(y_r) \rrbracket(\vec{n})$.

Wir definieren eine Funktion $\Downarrow\text{STA}\Downarrow : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ durch

$$\Downarrow\text{STA}\Downarrow(b, \vec{n}) := \text{if}(\Downarrow\text{STA}\Downarrow(b, \vec{n}), 0, 1)$$

und zeigen

$$\llbracket \text{STA}(y_r) \rrbracket(\vec{n}) = \Downarrow\text{STA}\Downarrow(y_r) \Downarrow(\mu \Downarrow\text{STA}\Downarrow(\vec{n}), \vec{n}). \quad (11.40)$$

Fall " $\mu \Downarrow\text{STA}\Downarrow(\vec{n}) = \perp$ ": Es gilt $\Downarrow\text{STA}\Downarrow(b, \vec{n}) = 0$ für alle $b \in \mathbb{N}$, mit Satz 8.2 folgt $\text{eval}(M_P^{\vec{n}}, \text{STA}) = \perp$ und damit $\llbracket \text{STA}(y_r) \rrbracket(\vec{n}) = \perp = \Downarrow\text{STA}\Downarrow(y_r) \Downarrow(\mu \Downarrow\text{STA}\Downarrow(\vec{n}), \vec{n})$.²³

Fall " $\mu \Downarrow\text{STA}\Downarrow(\vec{n}) = b \in \mathbb{N}$ ": Es gilt $\Downarrow\text{STA}\Downarrow(b, \vec{n}) > 0$, und mit Definition 11.36 folgt $\llbracket \text{STA}(y_r) \rrbracket(\vec{n}) = \Downarrow\text{STA}\Downarrow(y_r) \Downarrow(b, \vec{n}) = \Downarrow\text{STA}\Downarrow(y_r) \Downarrow(\mu \Downarrow\text{STA}\Downarrow(\vec{n}), \vec{n})$.

Mit (11.40) gilt $\llbracket P \rrbracket(\vec{n}) = \Downarrow\text{STA}\Downarrow(y_r) \Downarrow(\mu \Downarrow\text{STA}\Downarrow(\vec{n}), \vec{n})$, mit Satz 11.38 sind $\Downarrow\text{STA}\Downarrow(y_r) \Downarrow$ und $\Downarrow\text{STA}\Downarrow$ primitiv-rekursiv, damit ist auch $\Downarrow\text{STA}\Downarrow$ primitiv-rekursiv, folglich ist $\mu \Downarrow\text{STA}\Downarrow$ und damit auch $\llbracket P \rrbracket$ μ -rekursiv. ■

²³ Der Fall $\Downarrow\text{STA}\Downarrow(b', \vec{n}) = \perp$ und $\Downarrow\text{STA}\Downarrow(b, \vec{n}) = 0$ für $b' < b$ ist ausgeschlossen, denn $\Downarrow\text{STA}\Downarrow$ ist primitiv-rekursiv und damit total.

Bemerkung 11.40. Auch der Beweis von Satz 11.38 ist konstruktiv, denn für jede lokale Variable \bar{y}_i in einer Programmanweisung STA wird eine konkrete Definition g_i für eine primitiv-rekursive Funktion $\llbracket g_i \rrbracket$ angegeben, für die $\llbracket g_i \rrbracket = \Downarrow \text{STA}(\bar{y}_i) \Downarrow$ gilt. Im Beweis von Satz 11.34 wird damit dann eine konkrete Definition g für eine μ -rekursive Funktion $\llbracket g \rrbracket$ angegeben, für die $\llbracket g_i \rrbracket = \llbracket \text{STA}(\bar{y}_i) \rrbracket$ gilt. Somit wird auch dieser Satz (wie schon dessen Rückrichtung in Satz 11.34) durch Spezifikation und Verifikation eines Compilers $c : \mathcal{P} \rightarrow \mu\text{RF}$ bewiesen, vgl. Anmerkung 11.35.

Mit Satz 11.39 erhalten wir unmittelbar:

Korollar 11.41. (Kleenescher Normalformsatz für μ -rekursive Funktionen)
Für jede μ -rekursive Funktion $f : \mathbb{N}^k \mapsto \mathbb{N}$ existieren primitiv-rekursive Funktionen $g, h : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$, so daß $f(\vec{x}) = h(\mu g(\vec{x}), \vec{x})$ für alle $\vec{x} \in \mathbb{N}^k$.

Beweis. Mit dem Beweis von Satz 11.34 erhalten wir für f ein \mathcal{P} -Programm P_f mit $f = \llbracket P_f \rrbracket$ und daraus mit dem Beweis von Satz 11.39 dann $f(\vec{x}) = \Downarrow \text{STA}(\bar{y}_r) \Downarrow (\mu \Downarrow \text{STA}(\vec{x}), \vec{x})$ mit primitiv-rekursivem $\Downarrow \text{STA}(\bar{y}_r) \Downarrow$ und $\Downarrow \text{STA} \Downarrow$. ■

Bemerkung 11.42. Konsequenz von Korollar 11.41 ist insbesondere, daß bei Definition μ -rekursiver Funktionen die Anwendung des μ -Operators auf primitiv-rekursiv definierte Funktionen beschränkt werden kann. Mit dieser Forderung (anstatt der in Anmerkung 11.31 diskutierten Alternativen) bleibt die resultierende Programmiersprache μRF entscheidbar.

12. TURINGMASCHINEN

Turingmaschinen sind in der Berechenbarkeitstheorie ein allgemein akzeptiertes mathematisches Modell für programmierbare Rechner, siehe Kapitel 1 und Abschnitt 3.2. In diesem Kapitel werden Turingmaschinen definiert und anschließend bewiesen, daß Turingmaschinen nicht mehr leisten können als \mathcal{P} -Programme. Mit diesem Nachweis ist dann gezeigt, daß die Programmiersprache \mathcal{P} berechnungsvollständig ist und somit – unter Voraussetzung der Gültigkeit der Churchschen These – jede arithmetische Funktion, die nicht \mathcal{P} -berechenbar ist, auch nicht intuitiv berechenbar ist.

12.1. Definition und Rechenmodell

Definition 12.1. (Turingmaschine)

Eine Turingmaschine TM ist ein 7-Tupel $(Z, z_{start}, z_{stop}, \Gamma, \square, \Sigma, \delta)$ für das gilt:

- Z ist die endliche Menge der Zustände von TM ,
- $z_{start} \in Z$ ist der Anfangszustand von TM ,
- $z_{stop} \in Z$ ist der Endzustand von TM ,
- Γ ist das endliche Arbeitsalphabet von TM ,
- $\square \in \Gamma$ ist ein ausgezeichnetes Symbol des Arbeitsalphabets von TM ,
- $\Sigma \subseteq \Gamma \setminus \{\square\}$ ist das nicht-leere Eingabealphabet von TM ,
- δ ist eine totale Funktion $\delta : (Z \setminus \{z_{stop}\}) \times \Gamma \rightarrow Z \times \Gamma \times \{\leftarrow, \circlearrowleft, \rightarrow\}$, die Übergangsfunktion von TM .

Die Funktionen $f, l : \Gamma^* \rightarrow \Gamma$ und $r, b : \Gamma^* \rightarrow \Gamma^*$ (für first, last, rest und butlast) sind definiert durch

- $f(\varepsilon) := l(\varepsilon) := \square$,
- $r(\varepsilon) := b(\varepsilon) := \varepsilon$, sowie

- $f(a\alpha) := l(\alpha) := a$
- $r(a\alpha) := b(\alpha) := \alpha$.

mit $a \in \Gamma$ und $\alpha \in \Gamma^*$. Ein Wort $\alpha z \beta \in \Gamma^* Z \Gamma^*$ mit $\alpha \beta \in \Gamma^*$ wird eine Konfiguration der Turingmaschine genannt. Wir definieren die Konfigurationsrelation \vdash_{TM} als die kleinste Relation auf $\Gamma^* Z \Gamma^*$ mit:

1. $\alpha z \beta \vdash_{TM} b(\alpha) z' l(\alpha) b' r(\beta)$, falls $\delta(z, f(\beta)) = (z', b', \leftarrow)$, siehe (12.1),
2. $\alpha z \beta \vdash_{TM} \alpha z' b' r(\beta)$, falls $\delta(z, f(\beta)) = (z', b', \circ)$, siehe (12.2), oder
3. $\alpha z \beta \vdash_{TM} \alpha b' z' r(\beta)$, falls $\delta(z, f(\beta)) = (z', b', \rightarrow)$, siehe (12.3).

Eine Konfiguration $\alpha' z' \beta'$ mit $\alpha z \beta \vdash_{TM} \alpha' z' \beta'$ heißt Folgekonfiguration der Konfiguration $\alpha z \beta$. Eine Konfiguration $\alpha z_{stop} \beta$ heißt Endkonfiguration.

Die von einer Turingmaschine TM berechnete Funktion $\phi_{TM} : \Sigma^* \mapsto \Sigma^*$ ist definiert durch $\phi_{TM}(\beta) := \beta'$ genau dann, wenn es $\alpha', \gamma' \in \Gamma^*$ sowie ein $c \in \Gamma \setminus \Sigma$ gibt, so daß für alle $\alpha, \gamma \in \{\square\}^*$ gilt: $\alpha z_{start} \beta \gamma \vdash_{TM}^* \alpha' z_{stop} \beta' c \gamma'$. ■

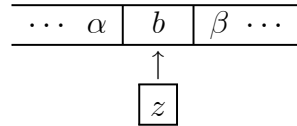
Die Arbeitsweise einer Turingmaschine kann man sich wie folgt vorstellen: Die Turingmaschine besitzt einen unendlich großen Speicher, das nach beiden Seiten unbegrenzte *Arbeitsband*, auf dem Worte aus Γ^* gespeichert sind. Das Arbeitsband ist in Speicherzellen unterteilt, wobei in jeder Speicherzelle ein Symbol aus Γ gespeichert ist. Da jedes Wort aus Γ^* endliche Länge besitzt, wird unterstellt, daß in den restlichen Speicherzellen das Sonderzeichen \square abgespeichert ist. Ein Arbeitsband, auf dem das Wort $a_1 a_2 \dots a_{j-1} a_j \in \Gamma^+$ gespeichert ist, illustrieren wir durch:

| | | | | | | | | | | |
|-----|---|---|----------------|----------------|-----|------------------|----------------|---|---|-----|
| ... | □ | □ | a ₁ | a ₂ | ... | a _{j-1} | a _j | □ | □ | ... |
|-----|---|---|----------------|----------------|-----|------------------|----------------|---|---|-----|

Weiter besitzt die Turingmaschine einen *endlichen Speicher*, der durch die endliche Menge der Zustände modelliert wird, sowie einen *Lesekopf*, der auf eine Speicherzelle des Arbeitsbands zeigt.

Die Turingmaschine arbeitet synchron, d.h. in jedem Arbeitsschritt werden der Zustand, das Arbeitsband und die Position des Lesekopfs verändert. Der "Arbeitszustand" der Turingmaschine kann dabei zu jedem Zeitakt durch eine Konfiguration $\alpha z b \beta \in \Gamma^* Z \Gamma^+$ mit $\alpha \beta \in \Gamma^*$ beschrieben werden, in der die Maschine im Zustand z ist, links vom Lesekopf das Wort α und rechts daran anschließend das Wort $b \beta$ auf dem Arbeitsband abgespeichert ist. Der Lesekopf zeigt also auf

das erste Zeichen b von $b\beta$. Eine Konfiguration ist damit ein ‘‘Schnappschuß’’ des ‘‘Arbeitszustands’’ der Maschine zu einem gegebenen Zeitpunkt.



Eine Konfiguration $\alpha z b \beta \in \Gamma^ Z \Gamma^+$*

Die Folgekonfiguration, die in einem Arbeitsschritt entsteht, wird durch die Übergangsfunktion δ bestimmt. Da δ einen endlichen Graphen besitzt – denn sowohl Z als auch Γ sind endlich – kann die Übergangsfunktion für $\Gamma = \{b_1, \dots, b_l\}$ und $Z = \{z_1, \dots, z_{k+1}\}$ mit Endzustand z_{k+1} durch eine endliche Tabelle mit Einträgen der Form

| $Z \setminus \Gamma$ | b_1 | b_2 | \cdots | b_{l-1} | b_l |
|----------------------|------------------------|------------------------|----------|----------------------------|------------------------|
| z_1 | $\delta(z_1, b_1)$ | $\delta(z_1, b_2)$ | \cdots | $\delta(z_1, b_{l-1})$ | $\delta(z_1, b_l)$ |
| z_2 | $\delta(z_2, b_1)$ | $\delta(z_2, b_2)$ | \cdots | $\delta(z_2, b_{l-1})$ | $\delta(z_2, b_l)$ |
| \cdots | \cdots | \cdots | \cdots | \cdots | \cdots |
| z_{k-1} | $\delta(z_{k-1}, b_1)$ | $\delta(z_{k-1}, b_2)$ | \cdots | $\delta(z_{k-1}, b_{l-1})$ | $\delta(z_{k-1}, b_l)$ |
| z_k | $\delta(z_k, b_1)$ | $\delta(z_k, b_2)$ | \cdots | $\delta(z_k, b_{l-1})$ | $\delta(z_k, b_l)$ |

Der Graph der Übergangsfunktion δ als Tabelle

dargestellt werden. Linearisiert man diese Tabelle zu einer endlichen Liste

| | | | | |
|----------------|------------------------|----------|----------------|------------------------|
| z_1, b_1 | $\delta(z_1, b_1)$ | | \cdots | \cdots |
| z_1, b_2 | $\delta(z_1, b_2)$ | | z_k, b_1 | $\delta(z_k, b_1)$ |
| \cdots | \cdots | \cdots | z_k, b_2 | $\delta(z_k, b_2)$ |
| z_1, b_{l-1} | $\delta(z_1, b_{l-1})$ | | \cdots | \cdots |
| z_1, b_l | $\delta(z_1, b_l)$ | | z_k, b_{l-1} | $\delta(z_k, b_{l-1})$ |
| \cdots | \cdots | | z_k, b_l | $\delta(z_k, b_l)$ |

Das Programm einer Turingmaschine als Turingtafel

so erhält man die *Turingtafel*, d.h. das Programm der Turingmaschine. Damit ist ein *Turingprogramm* eine endliche Folge von Tripeln $(z, b, (z', b', D))$ mit $z, z' \in Z$, $b, b' \in \Gamma$, $D \in \{\leftarrow, \circlearrowleft, \rightarrow\}$ und $\delta(z, b) = (z', b', D)$.

In jedem Arbeitstakt liest die Turingmaschine in der Konfiguration $\alpha z b \beta$ – also im Zustand z mit Lesekopf positioniert auf die Speicherzelle mit Inhalt b – das Zeichen aus der Speicherzelle des Arbeitsbands, auf die der Lesekopf zeigt, und reagiert gemäß den Anweisungen des Programms. Mit $\delta(z, b) = (z', b', D)$ geht die Maschine in den neuen Zustand z' über, überschreibt das Zeichen, auf das der Lesekopf zeigt, mit b' und bewegt den Lesekopf um eine Position nach links, falls $D = \leftarrow$, oder nach rechts, falls $D = \rightarrow$. Für $D = \circlearrowleft$ bleibt die Position des Lesekopfs unverändert.¹ Damit gibt es drei Möglichkeiten, eine Folgekonfiguration zu erhalten, vgl. Definition 12.1:

$$\begin{array}{ccc}
 \overline{\dots \alpha \mid a \mid b \mid \beta \dots} & & \overline{\dots \alpha \mid a \mid b' \mid \beta \dots} \\
 \uparrow & \vdash_{TM} & \uparrow \\
 \boxed{z} & & \boxed{z'}
 \end{array} \tag{12.1}$$

Folgekonfiguration für $\delta(z, b) := (z', b', \leftarrow)$ mit Definition 12.1(1)

$$\begin{array}{ccc}
 \overline{\dots \alpha \mid b \mid \beta \dots} & & \overline{\dots \alpha \mid b' \mid \beta \dots} \\
 \uparrow & \vdash_{TM} & \uparrow \\
 \boxed{z} & & \boxed{z'}
 \end{array} \tag{12.2}$$

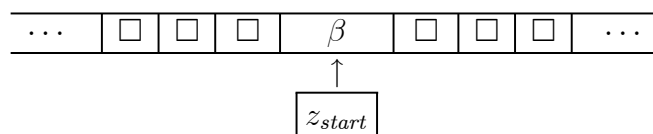
Folgekonfiguration für $\delta(z, b) = (z', b', \circlearrowleft)$ mit Definition 12.1(2)

$$\begin{array}{ccc}
 \overline{\dots \alpha \mid b \mid c \mid \beta \dots} & & \overline{\dots \alpha \mid b' \mid c \mid \beta \dots} \\
 \uparrow & \vdash_{TM} & \uparrow \\
 \boxed{z} & & \boxed{z'}
 \end{array} \tag{12.3}$$

Folgekonfiguration für $\delta(z, b) = (z', b', \rightarrow)$ mit Definition 12.1(3)

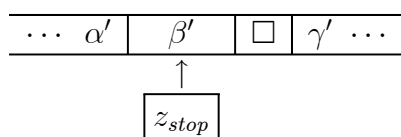
¹ Natürlich *muß* die Maschine weder den Zustand noch das Arbeitsband ändern: Mit $z' := z$ bzw. $b' := b$ bleibt die Maschine im gleichen Zustand bzw. läßt das Arbeitsband unverändert.

Um $\phi_{TM}(\beta)$ für ein $\beta \in \Sigma^*$ zu berechnen, wird die Turingmaschine in der Konfiguration



Startkonfiguration zur Berechnung von $\phi_{TM}(\beta)$

gestartet. Daran anschließend werden Schritt für Schritt Folgekonfigurationen berechnet. Falls dabei eine Endkonfiguration



Endkonfiguration bei Berechnung von $\phi_{TM}(\beta)$

mit $\beta' \in \Sigma^*$ erreicht wird, so hält die Turingmaschine mit β' als Ergebnis der Berechnung. Falls nie eine Endkonfiguration erreicht wird, so gilt $\phi_{TM}(\beta) = \perp$.

Bemerkung 12.2. Die Definition der Turingmaschine ist in der Literatur nicht einheitlich. Manche Autoren verwenden Maschinen mit mehreren Arbeitsbändern, mit einem oder mehreren einseitig begrenzten Arbeitsbändern, mit mehreren Endzuständen, mit partiellen Übergangsfunktionen, u.s.w. Allen diesen Varianten ist gemeinsam, daß sie äquivalent sind, d.h. daß sie die gleichen Funktionen berechnen können. Die verschiedenen Versionen sind allein durch den Zweck motiviert, der in einem bestimmten Anwendungskontext verfolgt wird. Des weiteren können Turingmaschinen auch indeterministisch definiert werden, ohne jedoch dadurch mehr Funktionen berechnen zu können. In der Berechenbarkeitstheorie spielt es daher keine Rolle, ob die deterministische oder die indeterministische Definition einer Turingmaschine verwendet wird. Bei Untersuchungen zur Komplexität von Problemen ist der Unterschied dagegen relevant.

12.2. Turing-berechenbare Funktionen

Um arithmetische Funktionen durch eine Turingmaschine zu berechnen, müssen natürliche Zahlen durch die Worte des Eingabealphabets dargestellt werden. Wir verwenden hier Bitworte, vgl. Abschnitt 4.4, d.h. wir stellen natürliche Zahlen n mittels einer Funktion $dual : \mathbb{N} \rightarrow \{0, 1\}^+$ als Worte $dual(n) \in \{0, 1\}^+$ dar.²

Definition 12.3. (Turing-berechenbare Funktionen)

Eine Funktion $\phi : \mathbb{N} \mapsto \mathbb{N}$ ist Turing-berechenbar genau dann, wenn

$$\phi(n) = \begin{cases} \perp & , \text{ falls } \phi_{TM}(dual(n)) \notin \text{Bild}(dual) \\ dual^{-1}(\phi_{TM}(dual(n))) & , \text{ falls } \phi_{TM}(dual(n)) \in \text{Bild}(dual) \end{cases}$$

für eine Turingmaschine $TM = (Z, z_{start}, z_{stop}, \{0, 1, \square\}, \square, \{0, 1\}, \delta)$ und für alle $n \in \mathbb{N}$ gilt. ■

Beispiel 12.4. Wir zeigen, daß die Vorgängerfunktion $pred : \mathbb{N} \rightarrow \mathbb{N}$ mit $pred(0) := 0$ und $pred(n+1) := n$ Turing-berechenbar ist: Sei die Turingmaschine TM_{PRED} gegeben durch

$$TM_{PRED} := (Z, z_{start}, z_{stop}, \{0, 1, \square\}, \square, \{0, 1\}, \delta)$$

mit $Z := \{z_{start}, z_{one}, z_{done}, z_{back}, z_{dec}, z_{strip}, z_{stop}\}$ und δ definiert durch das Turingprogramm

| | | | | | |
|-----|--------------------|---------------------------------|------|----------------------|--|
| (1) | $z_{start}, 0$ | $z_{stop}, 0, \circlearrowleft$ | (10) | $z_{back}, 0$ | $z_{back}, 0, \leftarrow$ |
| (2) | $z_{start}, 1$ | $z_{one}, 1, \leftrightarrow$ | (11) | $z_{back}, 1$ | $z_{back}, 1, \leftarrow$ |
| (3) | $z_{one}, 0$ | $z_{one}, 0, \leftrightarrow$ | (12) | z_{back}, \square | $z_{strip}, \square, \leftrightarrow$ |
| (4) | $z_{one}, 1$ | $z_{one}, 1, \leftrightarrow$ | (13) | z_{start}, \square | $z_{start}, \square, \circlearrowleft$ |
| (5) | z_{one}, \square | $z_{done}, \square, \leftarrow$ | (14) | z_{dec}, \square | $z_{dec}, \square, \circlearrowleft$ |
| (6) | $z_{done}, 1$ | $z_{back}, 0, \leftarrow$ | (15) | z_{done}, \square | $z_{done}, \square, \circlearrowleft$ |
| (7) | $z_{done}, 0$ | $z_{dec}, 1, \leftarrow$ | (16) | $z_{strip}, 0$ | $z_{stop}, \square, \leftrightarrow$ |
| (8) | $z_{dec}, 0$ | $z_{dec}, 1, \leftarrow$ | (17) | $z_{strip}, 1$ | $z_{stop}, 1, \circlearrowleft$ |
| (9) | $z_{dec}, 1$ | $z_{back}, 0, \leftarrow$ | (18) | z_{strip}, \square | $z_{strip}, \square, \circlearrowleft$ |

Dann geht die Maschine bei Eingabe 0 mit (1) in den Endzustand über, Ergebnis der Berechnung ist 0. Bei Eingabe eines Wortes 1... geht die Maschine vom

² Da \mathbb{N} unendlich ist, können wir \mathbb{N} nicht als Eingabealphabet verwenden (vgl. Bemerkung 2.5). Die Verwendung von Bitworten ist nicht zwingend – jedes (endliche, nicht-leere) Alphabet ist geeignet.

Startzustand mit (2) in den Zustand z_{one} über und liest mit (3) und (4) das Eingabewort ein. Ist das letzte Zeichen des Eingabeworts eingelesen, so geht die Maschine mit (5) in den Zustand z_{done} über, der Lesekopf wird nach links bewegt und steht dann auf dem letzten Zeichen s des Eingabeworts.

Fall $s = 1$: Mit (6) geht die Maschine in den Zustand z_{back} über und ersetzt dabei das letzte Zeichen 1 durch 0 (um 1 vom Eingabewort zu subtrahieren).

Fall $s = 0$: Mit (7) geht die Maschine in den Zustand z_{dec} über und ersetzt dabei das letzte Zeichen 0 durch 1 (um 1 vom Eingabewort zu subtrahieren). Solange die Maschine im Zustand z_{dec} 0 liest, wird mit (8) 0 durch 1 ersetzt und der Lesekopf nach links bewegt (um den Übertrag, der bei Subtraktion von 1 entstand, im Eingabewort weiterzureichen). Wenn im Zustand z_{dec} 1 gelesen wird, so wird dieses Zeichen mit (9) durch 0 ersetzt (der Übertrag ist jetzt verarbeitet), und die Maschine geht in den Zustand z_{back} über. (Da die Maschine nur in den Zustand z_{done} gelangt, wenn 1 mindestens einmal im Eingabewort vorkommt, jedoch $s \neq 1$ für das letzte Zeichen s des Eingabeworts gilt, ist gewährleistet, daß die Maschine tatsächlich irgendwann 1 im Zustand z_{dec} liest.)

Im Zustand z_{back} wird mit (10) und (11) der Lesekopf solange nach links bewegt, bis \square gelesen wird. Jetzt ist der Anfang des beschriebenen Arbeitsbands wieder erreicht, die Maschine geht mit (12) in den Zustand z_{strip} über, ersetzt mit (16) ein führendes 0-Bit, bewegt den Lesekopf nach rechts und geht in den Endzustand z_{stop} über.³ Falls im Zustand z_{strip} jedoch 1 gelesen wird, so geht die Maschine statt dessen mit (17) in den Endzustand z_{stop} über. Damit steht der Lesekopf in jedem Fall auf dem ersten Zeichen des um 1 subtrahierten Eingabeworts.⁴

Somit gilt:

1. $\phi_{TM_{PRED}}(dual(n)) \in Bild(dual)$ sowie
2. $0 = dual^{-1}(\phi_{TM_{PRED}}(dual(0)))$ und
3. $n = dual^{-1}(\phi_{TM_{PRED}}(dual(n + 1)))$,



also $pred(n) = dual^{-1}(\phi_{TM}(dual(n)))$, d.h. $pred$ ist Turing-berechenbar. ■

³ Dies ist erforderlich, da bei Dekrementierung einer 2er Potenz, z.B. 100, zunächst ein Bitwort mit führendem 0-Bit – im Beispiel 011 – entsteht.

⁴ Regel (18) dient nur dazu, die Forderung nach einer *totalen* Übergangsfunktion zu erfüllen – diese Regel kann nie angewendet werden.

12.3. Berechnungsvollständigkeit von \mathcal{P}

Wir zeigen jetzt, daß jede Turing-berechenbare arithmetische Funktion auch \mathcal{P} -berechenbar ist. Zum Beweis übersetzen wir das Programm einer Turingmaschine TM in ein \mathcal{P} -Programm \mathbf{TM} und zeigen, daß \mathbf{TM} die gleiche Funktion berechnet wie TM .

Sei $TM = (Z, z_0, z_k, \{\square, \mathbf{O}, \mathbf{l}\}, \square, \{\mathbf{O}, \mathbf{l}\}, \delta)$ eine Turingmaschine mit $Z = \{z_0, \dots, z_k\}$. Aus TM konstruieren wir *uniform* das \mathcal{P} -Programm \mathbf{TM} wie in Abbildung 12.1. Dabei repräsentieren wir die Zeichen des Arbeitsalphabets mittels $\# : \{\square, \mathbf{O}, \mathbf{l}\} \rightarrow \{0, 1, 2\}$ durch natürliche Zahlen, und zwar mit $\#\square := 0$, $\#\mathbf{O} := 1$ und $\#\mathbf{l} := 2$. Die Zustände aus Z werden durch die natürlichen Zahlen aus $\{0, \dots, k\}$ dargestellt.

Eine Konfiguration $a_1 \dots a_m z_i b_1 \dots b_n$ von TM wird durch $\#a_1 \dots \#a_m i \#b_1 \dots \#b_n$ kodiert. Im Programm \mathbf{TM} wird eine solche Konfiguration durch die lokalen Variablen **alpha**, **head**, **beta** und **state** repräsentiert, wobei gilt:

- **alpha** = $\pi^2(\#a_m, \pi^2(\#a_{m-1}, \pi^2(\dots \pi^2(\#a_2, \pi^2(\#a_1, 0)) \dots))$,
- **head** = $\#b_1$,
- **beta** = $\pi^2(\#b_2, \pi^2(\#b_3, \pi^2(\dots \pi^2(\#b_{n-1}, \pi^2(\#b_n, 0)) \dots))$, und
- **state** = i .

Für ein Eingabewort $b_1 \dots b_n \in \{\mathbf{O}, \mathbf{l}\}^*$ wird \mathbf{TM} mit $\pi^2(\#b_1, \pi^2(\#b_2, \pi^2(\dots \pi^2(\#b_{n-1}, \pi^2(\#b_n, 0)) \dots))$ aufgerufen. Mit den ersten vier Zuweisungen im \mathcal{P} -Programm \mathbf{TM} werden die lokalen Variablen durch die Repräsentation $0 \dots 00\#b_1 \dots \#b_n 0 \dots 0$ der Startkonfiguration $\square \dots \square z_0 b_1 \dots b_n \square \dots \square$ initialisiert.

In der *while*-Schleife wird anschließend das durch δ gegebene Turingprogramm abgearbeitet. Dieses Turingprogramm wird durch geschachtelte *case*-Anweisungen repräsentiert. In jedem Fall einer inneren *case*-Anweisung befindet sich die Maschine im Zustand **state** $\neq k$ und liest das Zeichen **head**. Für **state** = i und **head** = $\#b_j$ ändert das Programm die lokalen Variablen **alpha**, **head**, **beta** und **state** gemäß dem Wert von $\delta(z_i, b_j)$. Dies geschieht durch die Programmfragmente in Abbildung 12.2.

Gilt vor einem Schleifendurchlauf **state** = k , so ist der Endzustand erreicht, und die *while*-Schleife bricht ab.

```

procedure TM(x) <=
begin var alpha, head, beta, state, y;
  alpha := 0;      head := PAIR12(x);
  beta := PAIR22(x); state := 0;
  while state ≠ k do
    case state of
      0 : ...
        :
      i : case head of
            :
            #bj: DELTA-zi-bj
            :
          end_case
        :
      k-1 : ...
      other : SKIP
    end_case
  end_while;
  y := STRIP(PAIR2(head, beta));
  return(y)
end

```

Abbildung 12.1: Ein \mathcal{P} -Programm zur Simulation einer Turingmaschine

| | | |
|---|--|--|
| <pre> state := m; head := PAIR₁²(alpha); alpha := PAIR₂²(alpha); beta := PAIR²(#b_n, beta); </pre> | <pre> state := m; head := PAIR₁²(beta); alpha := PAIR²(#b_n, alpha); beta := PAIR₂²(beta); </pre> | <pre> state := m; alpha := PAIR₂²(alpha); head := #b_n; </pre> |
| $\delta(z_i, b_j) := (z_m, b_h, \leftarrow)$ | $\delta(z_i, b_j) := (z_m, b_h, \circlearrowright)$ | $\delta(z_i, b_j) := (z_m, b_h, \leftrightarrow)$ |

Abbildung 12.2: Programmfragmente des \mathcal{P} -Programms TM für $DELTA$ - z_i - b_j

Für

- **alpha** = $\pi^2(\#a'_{m'}, \pi^2(\#a'_{m'-1}, \pi^2(\dots \pi^2(\#a'_2, \pi^2(\#a'_1, 0)) \dots))$,
- **head** = $\#b'_1$, und
- **beta** = $\pi^2(\#b'_2, \pi^2(\#b'_3, \pi^2(\dots \pi^2(\#b'_{n'-1}, \pi^2(\#b'_{n'}, 0)) \dots))$

befindet sich die Turingmaschine TM dann in der Endkonfiguration $a'_1 \dots a'_{m'} z_k b'_1 \dots b'_{n'}$, repräsentiert durch $\#a'_1 \dots \#a'_{m'} k \#b'_1 \dots \#b'_{n'}$. Damit gilt $\phi_{TM}(b_1 \dots b_n) = b'_1 \dots b'_{n''} \in \{\mathbf{O}, \mathbf{I}\}^*$ für ein n'' mit $0 \leq n'' \leq n'$ und $b'_{n''+1} = \square$. Das \mathcal{P} -Programm **TM** berechnet mittels einer Prozedur **STRIP** die Repräsentation $\pi^2(\#b'_1, \pi^2(\#b'_2, \pi^2(\dots \pi^2(\#b'_{n''-1}, \pi^2(\#b'_{n''}, 0)) \dots))$ von $\phi_{TM}(b_1 \dots b_n)$ und hält mit diesem Ergebnis.

Für einen formalen Beweis der \mathcal{P} -Berechenbarkeit von Turing-berechenbaren Funktionen muß das \mathcal{P} -Programm **TM** verifiziert werden. Da dies manuell, d.h. ohne Rechnerunterstützung durch ein geeignetes Verifikationswerkzeug, zu aufwendig ist, skizzieren wir hier nur den Beweis.

Zunächst definieren wir eine Funktion $encode : \{\square, \mathbf{O}, \mathbf{I}\}^* \rightarrow \mathbb{N}$ durch $encode(\varepsilon) := 0$ und $encode(a_1 \dots a_m) := \pi^2(\#a_1, \pi^2(\#a_2, \pi^2(\dots \pi^2(\#a_{m-1}, \pi^2(\#a_m, 0)) \dots))$, mit der wir dann das Lemma

Lemma 12.5. *Wenn $\alpha z_i \beta \vdash_{TM} \alpha' z_h \beta'$, dann*

$$\begin{aligned} & \{\mathbf{alpha} = encode(\overleftarrow{\alpha}), \mathbf{state} = i, \mathbf{head} = \#f(\beta), \mathbf{beta} = encode(r(\beta))\} \\ & \quad \mathbf{case\ state\ of\ \dots\ end_case} \\ & \{\mathbf{alpha} = encode(\overleftarrow{\alpha'}), \mathbf{state} = h, \mathbf{head} = \#f(\beta'), \mathbf{beta} = encode(r(\beta'))\} \end{aligned}$$

formulieren können.⁵ Mit einem Beweis von Lemma 12.5 wird verifiziert, daß der Rumpf der *while*-Schleife der \mathcal{P} -Prozedur **TM** die Übergangsfunktion δ korrekt implementiert. Der Beweis wird (unter Verwendung von Definition 2.8) durch Fallunterscheidung nach den drei möglichen Fällen, eine Folgekonfiguration zu bilden, geführt, vgl. Definition 12.1(1,2,3) und Abbildung 12.2. Mit einem weiteren Lemma

⁵ $\overleftarrow{\alpha}$ entsteht durch Umkehrung der Reihenfolge der Symbole in α . $\{P\} \mathbf{STA} \{Q\}$ ist die übliche *Hoare*-Notation und bedeutet: Wenn die Aussage P (über Programmvariable) vor Ausführung der Programmanweisung **STA** gilt und die Ausführung von **STA** terminiert, so gilt die Aussage Q nach Ausführung von **STA**.

Lemma 12.6. Wenn $\alpha z_i \beta \vdash_{TM}^{(\ell)} \alpha' z_h \beta'$, dann

$$\begin{aligned} & \{\text{alpha} = \text{encode}(\overleftarrow{\alpha}), \text{state} = i, \text{head} = \#f(\beta), \text{beta} = \text{encode}(r(\beta))\} \\ & \quad (\text{case state of } \dots \text{end_case})^\ell \\ & \{\text{alpha} = \text{encode}(\overleftarrow{\alpha'}), \text{state} = h, \text{head} = \#f(\beta'), \text{beta} = \text{encode}(r(\beta'))\} \end{aligned}$$

wird verifiziert, daß alle Folgekonfiguration einer Konfiguration durch die Ausführung der *while*-Schleife in TM korrekt modelliert werden. Der Beweis wird – unter Verwendung von Lemma 12.5 – durch Induktion über die Länge ℓ der Konfigurationsfolge geführt. Mit Lemma 12.6 erhält man dann

Lemma 12.7. Wenn $\alpha z_i \beta \vdash_{TM}^* \alpha' z_k \beta'$, dann

$$\begin{aligned} & \{\text{alpha} = \text{encode}(\overleftarrow{\alpha}), \text{state} = i, \text{head} = \#f(\beta), \text{beta} = \text{encode}(r(\beta))\} \\ & \quad \text{while state} \neq k \text{ do case state of } \dots \text{end_case end_while} \\ & \{\text{alpha} = \text{encode}(\overleftarrow{\alpha'}), \text{state} = k, \text{head} = \#f(\beta'), \text{beta} = \text{encode}(r(\beta'))\} \end{aligned}$$

und damit

Lemma 12.8. $\llbracket \text{TM} \rrbracket(\text{encode}(\beta)) = \text{encode}(\phi_{TM}(\beta))$ für alle $\beta \in \{\square, \mathbf{O}, \mathbf{1}\}^*$.

Jetzt kann gezeigt werden, daß die Programmiersprache \mathcal{P} berechnungsvollständig ist:

Satz 12.9. Wenn $\phi : \mathbb{N} \mapsto \mathbb{N}$ Turing-berechenbar ist, so ϕ auch \mathcal{P} -berechenbar.

Beweis. Sei eine Turingmaschine TM mit

$$\phi(n) = \begin{cases} \perp & , \text{ falls } \phi_{TM}(\text{dual}(n)) \notin \text{Bild}(\text{dual}) \\ \text{dual}^{-1}(\phi_{TM}(\text{dual}(n))) & , \text{ falls } \phi_{TM}(\text{dual}(n)) \in \text{Bild}(\text{dual}) \end{cases} \quad (12.4)$$

gegeben wie in Definition 12.3. Sei weiter $\text{decode} : \mathbb{N} \mapsto \{\square, \mathbf{O}, \mathbf{1}\}^*$ definiert durch $\text{decode}(i) = \perp$, falls $i \notin \text{Bild}(\text{encode})$, und andernfalls $\text{decode}(0) := \varepsilon$ und $\text{decode}(i+1) := \#^{-1}(\pi_1^2(i+1))\text{decode}(\pi_2^2(i+1))$. Man zeigt leicht $\text{decode}(\text{encode}(\alpha)) = \alpha$ für alle $\alpha \in \{\square, \mathbf{O}, \mathbf{1}\}^*$ durch strukturelle Induktion über α . Mit Lemma 12.8 gilt dann

$$\text{decode}(\llbracket \text{TM} \rrbracket(\text{encode}(\text{dual}(n)))) = \phi_{TM}(\text{dual}(n)) \quad (12.5)$$

für alle $n \in \mathbb{N}$ und folglich

$$\text{dual}^{-1}(\text{decode}(\llbracket \text{TM} \rrbracket(\text{encode}(\text{dual}(n)))))) = \text{dual}^{-1}(\phi_{TM}(\text{dual}(n))) \quad (12.6)$$

mit $dual^{-1}(\alpha) = \perp$, falls $\alpha \notin \text{Bild}(dual)$. Die Funktionen $dual$ und $encode$ sind algorithmisch, und offensichtlich ist (*) $encode \circ dual : \mathbb{N} \rightarrow \mathbb{N}$ \mathcal{P} -berechenbar, etwa durch eine \mathcal{P} -Prozedur **PACK**. Ebenso sind die Funktionen $dual^{-1}$ und $decode$ algorithmisch, und ebenso offensichtlich ist (**) $dual^{-1} \circ decode : \mathbb{N} \mapsto \mathbb{N}$ mit $dual^{-1}(decode(i)) = \perp$, falls $i \notin \text{Bild}(encode)$ oder $decode(i) \notin \text{Bild}(dual)$, \mathcal{P} -berechenbar, etwa durch eine \mathcal{P} -Prozedur **UNPACK**. Damit erhält man für die \mathcal{P} -Prozedur

```

procedure PHI-TM(y) <=
begin var z;
  z := UNPACK(TM(PACK(y)));
  return(z)
end

```

(12.7)

schließlich

$$\begin{aligned}
& \llbracket \text{PHI-TM} \rrbracket(n) \\
&= \llbracket \text{UNPACK} \rrbracket(\llbracket \text{TM} \rrbracket(\llbracket \text{PACK} \rrbracket(n))) && , \text{ mit (12.7)} \\
&= dual^{-1}(decode(\llbracket \text{TM} \rrbracket(encode(dual(n))))) && , \text{ mit (*) und (**)} \\
&= dual^{-1}(\phi_{TM}(dual(n))) && , \text{ mit (12.6)}
\end{aligned}$$

also $\llbracket \text{PHI-TM} \rrbracket(n) = \phi(n)$ mit (12.4). ■

Bemerkung 12.10. Die Behauptung von Satz 12.9 entspricht den Behauptungen der Sätze 11.16 und 11.34 für primitiv-rekursive bzw. μ -rekursive (anstatt Turing-berechenbare) Funktionen und wird wie diese durch Definition und Verifikation eines Compilers bewiesen (vgl. Abbildungen 12.1 und 12.2 sowie Anmerkungen 11.17 und 11.35).

Mit (der Kontraposition von) Satz 12.9 ist gezeigt, daß jede Funktion $\phi : \mathbb{N} \mapsto \mathbb{N}$, die nicht \mathcal{P} -berechenbar ist, auch nicht Turing-berechenbar ist. Somit ist ϕ dann – unter der Voraussetzung der Gültigkeit der Churchschen These – auch nicht intuitiv berechenbar.

Mit Satz 12.9 ist jedoch noch nicht bewiesen, daß “ \mathcal{P} -berechenbar” und “Turing-berechenbar” äquivalente Begriffe sind, dazu fehlt noch:

Satz 12.11. Wenn $\phi : \mathbb{N} \mapsto \mathbb{N}$ \mathcal{P} -berechenbar ist, so ϕ auch Turing-berechenbar.

Um diesen Satz direkt zu beweisen, muß ein Interpretierer für \mathcal{P} -Programme mittels einer Turingmaschine implementiert werden, ein beweistechnisch sehr aufwendiges Unterfangen. Mit Satz 11.39 kann man alternativ zeigen, daß jede μ -rekursive Funktion auch Turing-berechenbar ist. Einen Beweis dazu findet man etwa in [Hermes(1971)] oder in [Lewis and Papadimitriou(1981)].

Akzeptiert man die Gültigkeit der Churchschen These, so ist die Behauptung von Satz 12.11 jedoch kaum überraschend. Denn angenommen, diese Behauptung ist falsch, so existiert ein \mathcal{P} -Programm **PROG** mit $\llbracket \mathbf{PROG} \rrbracket \neq \phi_{TM}$ für jede Turingmaschine TM , und folglich wäre die Churchsche These widerlegt.

13. BEGRIFFE UND SCHREIBWEISEN

- Für eine Menge N bezeichnet
 - 2^N die *Potenzmenge* von N ,
 - \overline{N} das *Komplement* von N ,
 - N^k die Menge aller endlichen Folgen von Elementen aus N der Länge k , und auch das kartesische Produkt $N_1 \times \dots \times N_k$ mit $N_i = N$ für alle $i \in \{1, \dots, k\}$,
 - N^* die Menge aller endlichen Folgen (beliebiger Länge) von Elementen aus N , und
 - N^+ die Menge $N^* \setminus \{\varepsilon\}$, d.h. N^* ohne die leere Folge ε .
- Mit $\phi : M \mapsto N$ bezeichnen wir eine Funktion von M nach N und nennen
 - ϕ *arithmetisch*, gdw. $N = \mathbb{N}$ und $M = \mathbb{N}^k$ für ein $k \in \mathbb{N}$,
 - ϕ *total* (in Kurzschreibweise $\phi : M \rightarrow N$), gdw. $\phi(m)$ für alle $m \in M$ definiert ist, d.h. $\phi(m) \in N$ für alle $m \in M$ gilt,
 - ϕ *partiell*, gdw. ϕ nicht total ist, d.h. $\phi(m)$ für mindestens ein $m \in M$ undefiniert ist, also $\phi(m) \notin N$ gilt (wir schreiben in einem solchen Fall auch $\phi(m) = \perp$),¹
 - $Def(\phi) \subseteq M$ mit $Def(\phi) := \{m \in M \mid \phi(m) \neq \perp\}$ den *Definitionsbereich* von ϕ ,²
 - $Bild(\phi) \subseteq N$ mit $Bild(\phi) := \{n \in N \mid \phi(m) = n \text{ für ein } m \in M\}$ den *Bildbereich* von ϕ , und

¹ Das Symbol \perp ist ein *Metazeichen* (d.h. eine Kurzschreibweise) für Undefiniertheit und nicht etwa das *Ergebnis* der Anwendung von ϕ auf m . Anders gesagt, im Unterschied zu einer natürlichen Zahl als Ergebnis von $\phi(m)$ kann man mit \perp nicht “rechnen”.

² “ $\phi(m) \neq \perp$ ” steht für “ $\exists n \in N. \phi(m) = n$ ” und “ $\phi(m) \neq n$ ” steht für “ $\phi(m) = \perp$ oder $\exists n' \in N \setminus \{n\}. \phi(m) = n'$ ”.

- (falls ϕ injektiv ist) $\phi^{-1} : N \mapsto M$ die Umkehrfunktion von ϕ , d.h. es gilt $\phi^{-1}(\phi(m)) = m$, falls $\phi(m) \neq \perp$, und $\phi(\phi^{-1}(n)) = n$, falls $\phi^{-1}(n) \neq \perp$.
- Für zwei Funktionen $\phi : N_1 \times \dots \times N_k \mapsto N$ und $\psi : N \mapsto N$ ist die *Funktionalkomposition* $\psi \circ \phi : N_1 \times \dots \times N_k \mapsto N$ von ψ und ϕ gegeben durch $\psi \circ \phi(n_1, \dots, n_k) = \psi(\phi(n_1, \dots, n_k))$, und wir definieren $\psi^{(k)} : N \mapsto N$ durch $\psi^{(0)}(n) = n$ und $\psi^{(i+1)}(n) = \psi^{(i)} \circ \psi(n) = \psi \circ \psi^{(i)}(n)$.
- Mit $\omega_{M \mapsto N}$ bezeichnen wir die *überall undefinierte* Funktion von M nach N , d.h. die Funktion, für die $\phi(m) = \perp$ für alle $m \in M$ gilt. Für $M = N = \mathbb{N}$ schreiben wir kurz ω anstatt $\omega_{\mathbb{N} \mapsto \mathbb{N}}$.
- Für $r \in \mathbb{R}$ ist $\lfloor r \rfloor$ (genannt *floor* r) die größte natürliche Zahl n mit $n \leq r$ und $\lceil r \rceil$ (genannt *ceiling* r) die kleinste natürliche Zahl n mit $r \leq n$. ■

LITERATURVERZEICHNIS

- [Hermes(1971)] Hermes, H., *Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit - Einführung in die Theorie der rekursiven Funktionen*, 2nd ed., Springer-Verlag, Berlin, Heidelberg, New York, 1971.
- [Jones(1997)] Jones, N. D., *Computability and Complexity: From a Programming Perspective*, The MIT Press, Cambridge, Massachusetts, London, England, 1997.
- [Kfoury et al.(1982)Kfoury, Moll, and Arbib] Kfoury, A. J., R. N. Moll, and M. A. Arbib, *A Programming Approach to Computability*, 2nd ed., Springer-Verlag, Berlin, Heidelberg, New York, 1982.
- [Lewis and Papadimitriou(1981)] Lewis, H., and C. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall, 1981.
- [Mendelson(1964)] Mendelson, E., *Mathematical Logic*, D. Van Nostrand, 1964.
- [Rogers Jr.(1988)] Rogers Jr., H., *Theory of Recursive Functions and Effective Computability*, The MIT Press, Cambridge, Massachusetts, 1988.
- [Wagner(1994)] Wagner, K. W., *Theoretische Informatik — Grundlagen und Modelle*, Springer, Berlin, Heidelberg, New York, 1994.

INDEX

- Äquivalenzproblem, 86
- Abarbeitung von \mathcal{P} -Programmen, 24
 - beschränkt, 63
- abzählbar, 45, 76
- ack*, 114
- Ackermann*, *W.*, 114
- Ackermann*-Funktion, 114
- add*, 36
- algorithmisch, 32
- Algorithmus, 10
- APPLY, 52
- Arbeitsband, 134
- Arithmetik, 91
- aufsteigend rekursiv aufzählbar, 73
- Aufzählungs
 - funktion, 70
 - monoton, 73
 - verfahren, 70
 - monoton, 74
- Ausführung
 - funktion, 23
 - von Programmanweisungen, 23
- Auswertung
 - funktion, 23
 - von Programmausdrücken, 23
- b, 133
- berechenbar
 - intuitiv, 27
 - loop-, 104
 - \mathcal{P} -, 26
 - $\mathcal{P}_{\text{loop}}$ -, 104
 - Turing-, 138
- berechnungs
 - äquivalent, 28
 - vollständig, 11, 29
- Bildbereich, 146
- Bild*(ϕ), 146
- Block, 13
- C_n^k , 96
- call-by-name, 119
- case, 19
- charakteristische Funktion, 54
- Church*, *A.*, 28
- Churchsche These, 28
- code*, 42
- COMPOSE, 27
- computable*, 26
- cont*, 22
- course-of-values recursion, 102
- CYCLE_k, 24
- decidable*, 54
- decode*, 143
- Definition
 - durch allgemeine Einsetzung, 97
 - durch Einsetzung, 94
 - durch gegenseitige Rekursion, 98
 - durch Minimierung, 116
 - durch primitive Rekursion, 94
 - durch strukturelle Rekursion, 97
- Definitionsbereich, 146
- Def*(ϕ), 146
- Diagonalisierung, 46, 80
- dovetailing*, 67
- dual*, 42, 138

*dual*_ℓ, 42
 Einsetzung, 94
 allgemeine, 97
encode, 142
 Endkonfiguration, 134
 entscheidbar, 54
 semi-, 58
 Entscheidungsverfahren, 54
 semi-, 58
 EQUAL, 15
EQV, 86
eval, 23
eval^(*n*), 112
 EXP, 82
*expressions*_{*P*}, 23
 f, 133
 FAC, 106
Fibonacci-Funktion, 100
 Folgekonfiguration, 134, 136
 Funktion
 Ackermann-, 114
 algorithmisch, 32
 arithmetisch, 146
 berechnete, 25, 134
 charakteristische, 54
 Fibonacci-, 100
 intuitiv berechenbar, 27
 loop-berechenbar, 104
 μ-rekursiv, 28, 93, 118
 ℙ-berechenbar, 26
 ℙ_{loop}-berechenbar, 104
 partiell, 146
 primitiv-rekursiv, 93, 95
 semi-charakteristische, 58
 strikt, 119
 total, 146
 Turing-berechenbar, 138
 überall undefiniert, 147
 universelle, 51
 Werteverlauf, 101
 Funktionalkomposition, 147
Gödel, K., 32, 91
 Gödelisierung, 33
 GE, 18
 gegenseitige Rekursion, 98
 GGT, 18, 20
 Grundfunktion, 94
H, 53
half, 94
 HALT, 49
halt, 50
 Haltefunktion, 50
 Halteproblem, 49, 53
 spezielles, 53
hd, 36
ID, 86
 Identitätsproblem, 86
 if, 96
 if, 13
 Induktion, 102
 initialer Speicher, 22
 Interpretationskosten, 61
 Interpretierer, 51, 52
 intuitiv berechenbar, 27
Kleene, S. C., 28, 93
 Kleenescher Normalformsatz, 132
 Kode, 43
 -tabelle, 42
 Kodierung
 von Datentypen, 32
 von Listen, 32, 36, 38
 von Programmen, 41
 Konfiguration, 134
 Konfigurationsrelation, 134

KURIOS, 48
 l, 133
 λ -Kalkuel, 28
 Laufvariable, 104
 Laufzeitbeschränkung, 61
 lazy evaluation, 119
 Lesekopf, 134
 Listen, 36
 lokale Variable, 13
 loop, 104
 -Anweisung, 104
 -Programm, 93, 104
 -berechenbar, 104

 MINUS, 18
 monoton
 Aufzählungsfunktion, 73
 Aufzählungsverfahren, 74
 μ -Operator, 93, 116, 118, 132
 μ -rekursive Funktion, 28, 93, 118
 μ RF, 119, 132

nth, 101
ntl, 101

 \mathcal{P} , 12
 -Programm, 12
 erweitert, 14, 19
 Interpretierer, 51, 52
 parametrisiert, 83
 -Prozedur, 12
 parametrisiert, 83
 -berechenbar, 26
 $\mathcal{P}_{\text{loop}}$, 104
 -berechenbar, 104
 $\mathcal{P}_{\text{loop}}[k]$, 105
 P_i^k , 94
 Paar-Funktion, 34, 96
 verallgemeinert, 38

 PACK, 144
pairing function, 34, 38
 PAIR^2 , 35
 PAIR_1^2 , 35
 PAIR_2^2 , 36
 PAIR^k , 38
 PAIR_h^k , 38
 Parameter
 aktuell, 17, 25
 formal, 12
 Parametrisierung, 82
Peter, R., 114
 PHI-TM, 144
 PLUS, 106
 PRED, 13
 pred, 96
 Presburger Arithmetik, 91
 PRF, 103
 primitiv-rekursive Funktion, 93, 95
 primitive Rekursion, 94
 Problem, 53
 endlich, 55
 entscheidbar, 54
 kofinit, 57
 komplementär, 57
 reduzierbar, 81
 semi-entscheidbar, 58
 Programm
 -anweisung, 13
 Ausführung von, 23
 -ausdruck, 13
 Auswertung von, 23
 -kodierungsfunktion, 43
 -variable, 13
 loop-, 93, 104
 Projektionsfunktion, 94
 Prozedur, 12
 -aufruf, 17
 -rumpf, 12

- parametrisiert, 83
- r, 133
- RAM, 7
- random access machine, 7
- recursive*, 26, 54
- reduzierbar, 81
- Registermaschine, 7
- Rekursion
 - gegenseitige, 98
 - primitive, 94
 - strukturelle, 97
 - Werteverlauf, 102
- rekursiv aufzählbar, 70, 76
 - aufsteigend, 73
- return, 13
- Rice, H.G.*, 89
- S, 53
- S, 94
- s-m-n*-Theorem, 84
- $s_{\mathcal{P}}$, 64
- Schrittfunktion
 - universelle, 64
 - von \mathcal{P} -Programmen, 63
- Schrittzählfunktion, 62
- Selbstanwendbarkeitsproblem, 53
- Selbstanwendungsfunktion, 47
- self*, 47
- Semantik, 11
 - axiomatisch, 21
 - denotational, 21
 - operational, 21, 25
 - von loop-Programmen, 105
 - von $\mathcal{P}_{\text{loop}}$ -Programmen, 105
 - von \mathcal{P} -Programmen, 14, 25
 - von Programmaufrufen, 25
 - von Prozeduraufrufen, 25
- semi
 - Entscheidungsverfahren, 58
 - charakteristische Funktion, 58
 - entscheidbar, 58
- SKIP, 13
- Speicher, 21, 134
 - adresse, 22
 - bedarf, 22
 - modell, 22
 - zelle, 22, 134
 - initial, 22
- statements_P*, 23
- STEP, 65
- step*, 62
- step⁽ⁿ⁾*, 129
- strikt, 119
- strukturelle Rekursion, 97
- SUCC, 13
- sum*, 101
- sum-up*, 94
- TIMES, 106
- tl*, 36
- TM, 140
- TOT, 79
- Totalitätsproblem, 79
- Turing, A.*, 28
- Turing
 - berechenbar, 138
 - maschine, 6, 11, 28, 133, 137
 - indeterministisch, 137
 - programm, 136
 - tafel, 136
 - vollständig, 11
- $u_{\mathcal{P}}$, 51
- Übergangsfunktion, 133
- Umkehrfunktion, 147
- universelle
 - Funktion, 51
 - Schrittfunktion, 64
- UNPACK, 144

value, 23
 Werteverlaufs
 -funktion, 101
 -rekursion, 102
while, 14
 Z, 94
 $\omega_{M \mapsto N}$, 147
 ω , 147
 N^* , 146
 N^k , 146
 \bar{N} , 146
 N^+ , 146
 2^N , 146
 α_M , 70
 $\bar{\alpha}$, 142
 $A_{\mathcal{P}}$, 42
 $[[P]]$, 25
 $[[\mathcal{P}]]$, 26
 $[[P^{STEP}]]$, 63
 $[[PRF]]$, 104
 $[[EXPR]]$, 109
 $[[STA(y)]]$, 109
 $| \rangle STA(y) \langle |$, 122
 $| \rangle STA \langle |$, 122
 $[[STA(y_1, \dots, y_k)]]$, 125
 $| \rangle STA(y_1, \dots, y_k) \langle |$, 125
 O, 42
 I, 42
 \square , 133
 \perp , 146
 χ_M , 53
 $\tilde{\chi}_M$, 58
 \cong_M , 58
 δ , 133
 \mathcal{F}_{tot} , 46
 $M \mapsto N$, 146
 ϕ^{-1} , 147
 $M \rightarrow N$, 146
 $\psi^{(k)}$, 147
 $\psi \circ \phi$, 147
 Γ , 133
 $\gamma^?$, 33
 \wedge , 19
 \vee , 19
 \neg , 19
 ε , 36, 146
 \sqsubseteq , 77
 $M(i)$, 22
 M^{init} , 22
 $M_P^{(n_1, \dots, n_k)}$, 25
 $M[j \leftarrow n]$, 22
 $M(\mathbf{z})$, 22
 $M[\mathbf{z} \leftarrow n]$, 22
 \mathbb{N} , 12
 $\natural P$, 43
 $\natural \mathcal{P}$, 43
 $\mathcal{P}[k]$, 14
 φ_i , 45
 ϕ_{TM} , 134
 π^2 , 34, 96
 π_1^2 , 35, 96
 π_2^2 , 35, 96
 π^{-2} , 35
 π^k , 38, 41
 π_h^k , 38, 41
 \Rightarrow , 103
 \Rightarrow^+ , 103
 \preceq_{ϱ} , 81
 Σ , 133
 $\langle i, n \rangle$, 21
 \leftrightarrow , 133
 \hookrightarrow , 133
 \circlearrowleft , 133
 \vdash_{TM} , 134
 $[\dots]$, 147
 $[\dots]$, 147
 \bar{x} , 94

\vec{f} , 129

Z , 133

z_{start} , 133

z_{stop} , 133