Das Ziel sind die Unvollständigkeitssätze (12.2) (12.4):

Wir betrachten die Arithmetik über den natürlichen Zahlen.

Kein Beweissystem (z.B. aus Axiomen und Regeln) für die Arithmetik kann genau die „wahren“ (d.h. von $\mathfrak{N}$ erfüllten) arithmetischen Formeln formal beweisen.

Wir können sogar für jedes Beweissystem eine Π_1-Formel konstruieren,
die „wahr“ ist, aber in diesem Beweissystem keinen Beweis hat.

Teil 2

18. Januar 2020

Das Ziel sind die Unvollständigkeitssätze (12.2) (12.4):

Wir betrachten die Arithmetik über den natürlichen Zahlen.

Kein Beweissystem (z.B. aus Axiomen und Regeln) für die Arithmetik kann genau die „wahren“ (d.h. von $\mathfrak{N}$ erfüllten) arithmetischen Formeln formal beweisen.

Wir können sogar für jedes Beweissystem eine Π_1-Formel konstruieren,
die „wahr“ ist, aber in diesem Beweissystem keinen Beweis hat.

Hier wird über alle möglichen *Beweissysteme* gesprochen.
Wir haben bisher zwei spezielle Beispiele gesehen:
Natürliches Schließen + Robinson-Axiome bzw. Peano-Axiome.
Warum ist „Natürliches Schließen + alle wahren Formeln als Axiome“ kein Beweissystem?
Wie kann man über *alle* Beweissysteme sprechen?

Teil 2

18. Januar 2020

Das Ziel sind die Unvollständigkeitssätze (12.2) (12.4):

Wir betrachten die Arithmetik über den natürlichen Zahlen.

Kein Beweissystem (z.B. aus Axiomen und Regeln) für die Arithmetik kann genau die „wahren“ (d.h. von $\mathfrak{N}$ erfüllten) arithmetischen Formeln formal beweisen.

Wir können sogar für jedes Beweissystem eine Π_1-Formel konstruieren,
die „wahr“ ist, aber in diesem Beweissystem keinen Beweis hat.

Um Beweissysteme zu definieren, betrachten wir
URM-Programme und die Theorie entscheidbarer und semi-entscheidbarer Mengen. (VL 8–10)

Dann zeigen wir eine Verbindung zwischen Σ_1-Formeln und URM-Programmen bezgl. Erfüllung durch $\mathfrak{N}$ und „erfolgreicher“ Berechnungen. (VL 11)

Mit diesen Werkzeugen erhalten wir schließlich die Unvollständigkeitssätze (VL 12).

Abschließend gehen wir noch die Original-Arbeit von Gödel (1930) durch
und schauen, wie er das gemacht hat. (VL 13 . . .)

Teil 2: Unvollständigkeitssätze

VL 8: Totale und partielle berechenbare Funktionen

Es wird das Computer-Modell URM (Unlimited Register Machine) vorgestellt,
mit dem man Funktionen auf den natürlichen Zahlen berechnen kann.

Wir werden für allerlei Funktionen zeigen, wie sie von URM-Programmen berechnet werden können. Das ist (leider) technisch recht aufwendig, überzeugt uns aber davon, dass man alles, was von einem Computer-Modell erwarten kann, auch vom URM-Modell erreicht wird.

Am Ende steht die *These von Church und Turing*, nach der man alles,
was man in diesem Modell berechnen kann, auch „intuitiv“ berechnen kann. Das erlaubt uns für die Zukunft, „intuitiv“ statt formal zu argumentieren (wenn wir die Fallstricke der intuitiven Argumentation kennen).

VL 9: Entscheidbare und semi-entscheidbare Mengen

Unser Ziel ist ja, einen Unterschied zwischen

der Menge arithm. Formeln, die mit den Peano-Axiomen bewiesen werden können und

der Menge arithm. Formeln, die vom Standard-Modell $\mathfrak{N}$ erfüllt werden

zu zeigen.

Dazu „übertragen“ wir die Begriffe der totalen bzw. partiellen Berechenbarkeit von Funktionen auf Mengen und erhalten die Begriffe

- entscheidbare Menge bzw.
- semi-entscheidbare Menge.

Uns interessieren besonders die semi-entscheidbaren Mengen.

Diese Eigenschaft ist auch der gesuchte Unterschied zwischen den beiden o.g. Mengen (aber das kommt erst in einer späteren Vorlesung).

VL 10: Mengen beweisbarer Formeln sind semi-entscheidbar

Wir kennen Beweissysteme für die Arithmetik aus Axiomen und Natürlichem Schließen und wissen, wie man damit Formeln herleiten kann.

Wir zeigen nun, dass diese Mengen herleitbarer Formeln stets semi-entscheidbar sind (wenn die Axiome es auch sind). (10.5)

Unsere „Erfahrung“ mit Beweissystemen nutzen wir jetzt,
- um zwei allgemeine (abstrakte) Begriffe von Beweissystemen zu definieren,
- die in die Begriffswelt der berechenbaren Funktionen passen
- und die uns bekannten Beweissysteme enthalten.

Interessanterweise stellen sich die mit diesen Beweissystemen herleitbaren Formelmengen als genau die semi-entscheidbaren Mengen heraus. (10.7)
Das wird sich später als die gesuchte Eigenschaft der Robinson-/Peano-Arithmetik herausstellen.

VL 11: URM-Programme sind wie Σ_1-Formeln

Wir wissen, dass die Menge der von $\mathfrak{N}$ erfüllten Σ_1-Formeln semi-entscheidbar ist.

Gilt das auch umgekehrt?
Sind semi-entscheidbare Mengen auch „irgendwie“ Mengen von Σ_1-Formeln,
die von $\mathfrak{N}$ erfüllt werden?

Diese Frage wird in dieser Vorlesung mit „ja“ beantwortet (12.7).
Dazu wird gezeigt, dass (ganz grob gesagt)
URM-Programme eigentlich Σ_1-Formeln sind (und umgekehrt), und
dass das Halten des Programmes der Erfüllung der entsprechenden Formel von $\mathfrak{N}$ entspricht.

Damit erhalten wir das Werkzeug für die nächste Vorlesung,
in der gezeigt wird, dass die Menge aller von $\mathfrak{N}$ erfüllten Formeln nicht semi-entscheidbar ist.

VL 12: Die Unvollständigkeitssätze

Wir wissen, dass genau die semi-entscheidbare Mengen durch Σ_1-Formeln beschrieben werden.
Also wird z.B. das spezielle Halteproblem K durch eine Σ_1-Formel ψ_K beschrieben.

Wir kennen eine Menge, die nicht semi-entscheidbar ist:

das Komplement $\overline{K}$ des speziellen Halteproblems.

Sie wird von $\neg\psi_K$ und von keiner Σ_1-Formel beschrieben.

Das liefert die erste Version des Unvollständigkeitssatzes von Gödel:

die Menge der von $\mathfrak{N}$ erfüllten Formeln ist nicht semi-entscheidbar. (12.2)

Also gibt es für jedes Beweissystem Formeln,

die von $\mathfrak{N}$ erfüllt werden aber vom Beweissystem nicht bewiesen werden.

In der zweiten Version des Unvollständigkeitssatzes geben wir solche Formeln an. (12.4)

Vorlesung 8:
Totale und partielle URM-berechenbare Funktionen

2. Unvollständigkeitssätze

VL08: Totale und partielle URM-berechenbare Funktionen

- Unlimited Register machines: URMs
- URM-berechenbare Funktionen und die These von Church und Turing
- Kodierung von URM-Programmen durch Zahlen
- Nicht alle Funktionen sind URM-berechenbar
- Universelle URM-Programme

VL09: Entscheidbare und semi-entscheidbare Mengen

VL10: Mengen beweisbarer Formeln sind semi-entscheidbar

VL11: URM-Programme sind wie Σ_1-Formeln

VL12: Die Unvollständigkeitssätze

8.1 URM-berechenbare Funktionen

und die These von Church und Turing

In diesem Abschnitt wird das Computer-Modell URM vorgestellt,
mit dem man Funktionen auf den natürlichen Zahlen in

- berechenbare Funktionen
  (d.h. solche, für die es ein URM-Programm gibt) und
- nicht-berechenbare Funktionen
  (d.h. solche, für die es kein URM-Programm gibt)

unterscheiden kann.

Wir werden für allerlei Funktionen zeigen, wie sie von URM-Programmen berechnet werden können. Das ist (leider) technisch recht aufwendig, überzeugt uns aber davon, dass man alles, was von einem Computer-Modell erwarten kann, auch vom URM-Modell erreicht wird.
Am Ende steht die *These von Church und Turing*, nach der man alles,
was man in diesem Modell berechnen kann, auch „intuitiv“ berechnen kann. Das erlaubt uns für die Zukunft, „intuitiv“ statt formal zu argumentieren (wenn wir die Fallstricke der intuitiven Argumentation kennen).

Definition 8.1 (Syntax von URM-Programmen)

Eine *Unlimited Register Machine* (URM) besteht aus

- Registern $R_0, R_1, R_2, \ldots, R_k$ (für $k \in \mathbb{N}$), die jeweils eine nat. Zahl speichern können
- und einem URM-Programm.

Ein *URM-Programm* ist eine endliche Folge von Zahlen,
die zeilenweise mit den angegebenen Mnemonics dargestellt werden.
a, b, c stehen für beliebige natürliche Zahlen.

Zeile		Wirkung
$\mathtt{R_a} = 0$	$2^0 \cdot 3^a$	setze Register R_a auf 0
$\mathtt{R_a}$ += 1	$2^1 \cdot 3^a$	addiere 1 zum Register R_a
if $\mathtt{R_a} == \mathtt{R_b}$: goto c	$2^2 \cdot 3^a \cdot 5^b \cdot 7^c$	falls Register R_a und R_b den gleichen Inhalt haben, dann springe zu Zeile c des Programms

Sei m das Maximum aller $\mathtt{a}, \mathtt{b}$, die in den Zeilen des URM-Programms P vorkommen.
Dann benutzt P (höchstens) die Register $R_0, R_1, \ldots, R_m$,
und man sagt: P ist URM-Programm mit m Registern (plus Ausgaberegister R_0).

Beispiele für URM-Programme

Wir werden nun eine Reihe von Beispielen für URM-Programme sehen.

Später werden wir sehen, wie man URM-Programme als Zahlen kodieren kann.

Ein Ziel der folgenden Beispiel-Programme ist es, zu sehen,
dass die Funktionen zum Kodieren und Dekodieren von URM-Programmen
auch von URM-Programmen berechnet werden.

Register gleichsetzen (und abkürzende Schreibweisen benutzen)

```
0:  R0 = 0
1:  if R0 == R1 : goto 4
2:  R0 += 1
3:  if R0 == R0 : goto 1
```

Kopiere den Inhalt von R_1 nach R_0.

Register gleichsetzen (und abkürzende Schreibweisen benutzen)

```
0:  R0 = 0
1:  if R0 == R1 : goto 4
2:  R0 += 1
3:  goto 1
```

Kopiere den Inhalt von R_1 nach R_0.

Abkürzende Schreibweisen:

`goto a` springe zu Zeile *a* des Programms

Bei Benutzung abkürzender Schreibweisen weiß man,
wie man die Programme ohne abkürzende Schreibweisen schreiben kann.

Register gleichsetzen (und abkürzende Schreibweisen benutzen)

```
0:  R0 = 0
1:  while R0 != R1 :
2:      R0 += 1
```

Kopiere den Inhalt von R_1 nach R_0.

Abkürzende Schreibweisen:

`goto a`	springe zu Zeile *a* des Programms
`while Ra != Rb :`	(klar, Schleifenrumpf wird durch Einrückung gekennzeichnet)

Bei Benutzung abkürzender Schreibweisen weiß man,
wie man die Programme ohne abkürzende Schreibweisen schreiben kann.

Register gleichsetzen (und abkürzende Schreibweisen benutzen)

```
0:  R0 = 0
1:  while R0 != R1 :
2:      R0 += 1
```

Kopiere den Inhalt von R_1 nach R_0.

```
Rb = 0
while Rb != Ra :
    Rb += 1
```

Kopiere den Inhalt von R_a nach R_b.

Abkürzende Schreibweisen:

`goto a`	springe zu Zeile a des Programms
`while Ra != Rb :`	(klar, Schleifenrumpf wird durch Einrückung gekennzeichnet)
`Rb = Ra`	kopiere Inhalt von R_a nach R_b

Bei Benutzung abkürzender Schreibweisen weiß man,
wie man die Programme ohne abkürzende Schreibweisen schreiben kann.

Addieren

Addiere die Inhalte der Register R_a und R_b und schreibe das Ergebnis nach R_c.

```
Rc = Ra
Rt = 0
while Rb != Rt :
    Rc += 1
    Rt += 1
```

Abkürzende Schreibweise:

```
Rc = Ra + Rb
```

Multiplizieren und Potenzieren

$R_t = 0$
$R_c = 0$
while $R_b != R_t$:
  $R_c = R_c + R_a$
  R_t += 1

Abk.: $R_c = R_a \cdot R_b$

$R_t = 0$
$R_c = 0$
R_c += 1
while $R_b != R_t$:
  $R_c = R_c \cdot R_a$
  R_t += 1

Abk.: $R_c = R_a^{R_b}$

Dekrementieren und subtrahieren

Die Subtraktion auf den natürlichen Zahlen ist

$$a \dot{-} b = \begin{cases} a - b, & \text{falls } a > b \\ 0, & \text{sonst} \end{cases}$$

```
Ra = 0
Rt = 0
if Rb == Rt : goto Ende
Rt += 1
while Rb != Rt :
   Ra += 1
   Rt += 1
```

Abk.: `Ra = Rb-1`

`Ende` die Nummer der ersten Programmzeile „hinter“ dem Programm

```
Rz = 0
Ra = Rb
Rs = Rc
while Rs != Rz :
   Ra = Ra-1
   Rs = Rs-1
```

Abk.: `Ra = Rb- Rc`

Vorzeichenfunktionen

$$sig(x) = \begin{cases} 1, & \text{falls } x > 0 \\ 0, & \text{falls } x = 0 \end{cases} \qquad \overline{sig}(x) = 1 - sig(x)$$

```
Ra = 0
Rc = 0
if Rc == Rb : goto Ende
Ra += 1
```

Abk.: $\mathtt{R_a = sig(R_b)}$

```
Rt = sig(Rb)
Rs = 0
Rs += 1
Ra = Rs - Rt
```

Abk.: $\mathtt{R_a = \overline{sig}(R_b)}$

Vergleiche

$$\text{„}R_a > R_b\text{“} = \begin{cases} 1, & \text{falls } R_a > R_b \\ 0, & \text{sonst} \end{cases} \qquad \text{„}R_a \geqslant R_b\text{“} = \text{„}R_a + 1 > R_b\text{“}$$

$\mathtt{R_t} = \mathtt{R_a} - \mathtt{R_b}$
$\mathtt{R_c} = \mathtt{sig}(\mathtt{R_t})$

Abk.: $\mathtt{R_c} = \text{“}\mathtt{R_a} > \mathtt{R_b}\text{”}$

$\mathtt{R_t} = \mathtt{R_a}$
$\mathtt{R_t}$ += 1
$\mathtt{R_c} = \text{“}\mathtt{R_a} > \mathtt{R_t}\text{”}$

Abk.: $\mathtt{R_c} = \text{“}\mathtt{R_b} \geqslant \mathtt{R_a}\text{”}$

Aussagenlogische Verknüpfungen

$$\text{„}R_a \text{ und } R_b\text{“} = \begin{cases} 1, & \text{falls } R_a > 0 \text{ und } R_b > 0 \\ 0, & \text{sonst} \end{cases}$$

$R_c = R_a \cdot R_b$
$R_c = \mathtt{sig}(R_c)$

Abk.: $R_c = R_a \wedge R_b$

$$\text{„nicht } R_a\text{“} = \begin{cases} 1, & \text{falls } R_a = 0 \\ 0, & \text{sonst} \end{cases}$$

$R_c = \overline{\mathtt{sig}}(R_a)$

Abk.: $R_c = \,!R_a$

$$\text{„}R_a \text{ oder } R_b\text{“} = \begin{cases} 1, & \text{falls } R_a > 0 \text{ oder } R_b > 0 \\ 0, & \text{sonst} \end{cases}$$

$R_c = R_a + R_b$
$R_c = \mathtt{sig}(R_c)$

Abk.: $R_c = R_a \vee R_b$

Ausdrücke auswerten

Es ist „intuitiv klar“,
dass arithmetisch-logische Ausdrücke als URM-Programme geschrieben werden können.

Beispiel:
$R_c = R_a + \mathtt{sig}(R_a \cdot (R_b - 2))$

ist Abkürzung für

$R_t = 0$
$R_t \mathrel{+}= 1$
$R_t \mathrel{+}= 1$
$R_t = R_b - R_t$
$R_t = R_a \cdot R_t$
$R_t = \mathtt{sig}(R_t)$
$R_c = R_a + R_t$

Schleifen mit beliebigen Bedingungen

```
while <Ausdruck> :
  P
```

ist Abkürzung für

```
R_t = <Ausdruck>
R_z = 0
while R_t != R_z :
  P
  R_t = <Ausdruck>
```

if – then – else

```
if <Ausdruck> :
  P
else :
  Q
```

ist Abkürzung für

```
R_i = 0
R_z = 0
while <Ausdruck> ∧ !(R_i > R_z) :
  P
  R_i += 1
R_i = 0
while !(R_i > R_z) :
  Q
  R_i += 1
```

Ganzzahlige Division und Rest

$R_t = R_a$
$R_c = 0$
while $R_t \geqslant R_b$:
  $R_t = R_t - R_b$
  R_c += 1

Abk.: $R_c = R_a / R_b$

$R_t = R_a / R_b$
$R_s = R_t \cdot R_b$
$R_c = R_a - R_s$

Abk.: $R_c = R_a \bmod R_b$

Primzahlen

```
Rc = 0
if Ra > 1 :
  Rt = 2
  while Ra mod Rt :
    Rt += 1
  if Rt==Ra :
    Rc += 1
```

Abk.: R_c = "R_a ist Primzahl"

```
Rb = 2
Rt = 1
while Ra> Rt :
  Rb += 1
  Rs = "Rb ist Primzahl"
  Rt = Rt + Rs
```

Abk.: R_b = R_a-te Primzahl

Potenzen von Teilern finden

„maximale Potenz von t als Faktor von n“ $= \max\{k \mid t^k \text{ teilt } n\}$

```
Rc = 0
while ! ( Ra mod Rb^(Rc+1) ) :
  Rc += 1
```

Abk.: `Rc = maxPotenz von Rb in Ra`

URM-Programme – Semantik

Definition 8.2 (Konfiguration eines URM-Programms)

Sei P ein URM-Programm mit k Registern.
Eine Konfiguration von P ist ein Element von $\mathbb{N}^{k+2}$.

Eine Konfiguration $(c_0, \ldots, c_k, c_{k+1})$ eines URM-Programms $P = (I_0, \ldots, I_m)$ besteht aus

- den Inhalten $c_0, \ldots, c_k$ der Register $R_0, \ldots, R_k$ und
- der Nummer c_{k+1} (*Programmzähler*)
  der nächsten auszuführenden Programmzeile $I_{c_{k+1}}$.

Definition 8.3 (Konfigurationsübergangsrelation $\xrightarrow{P}$)

Sei $P = (I_0, I_1, \ldots, I_{s-1})$ ein URM-Programm mit k Registern,
und $c = (c_0, \ldots, c_k, pz)$ sowie $c' = (c'_0, \ldots, c'_k, pz')$ seien zwei Konfigurationen von P.
Es gilt $c \xrightarrow{P} c'$ gdw.

$pz < s$ und

- $I_{pz} = \texttt{R}_\texttt{a} \texttt{ = 0}, \quad c'_\texttt{a} = 0, \;\; c'_i = c_i$ für $i \neq \texttt{a}$, und $pz' = pz + 1$,
  oder
- $I_{pz} = \texttt{R}_\texttt{a} \texttt{ += 1}, \quad c'_a = c_\texttt{a} + 1, \;\; c'_i = c_i$ für $i \neq \texttt{a}$, und $pz' = pz + 1$,
  oder
- $I_{pz} = \texttt{if R}_\texttt{a} \texttt{ == R}_\texttt{b} \texttt{ : goto c}, \quad c'_i = c_i$ für alle i, und
  - $c_\texttt{a} = c_\texttt{b}$ und $pz' = \texttt{c}$, oder
  - $c_\texttt{a} \neq c_\texttt{b}$ und $pz' = pz + 1$.

$\xrightarrow{P}{}^*$ ist die reflexive und transitive Hülle von $\xrightarrow{P}$.

8.2 URM-berechenbare Funktionen

Definition 8.4 (URM-berechenbare Funktionen)

Eine partielle Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist URM-berechenbar,
falls es ein URM-Programm $P = (I_0, \ldots, I_{s-1})$ mit $\ell \geqslant k$ Registern gibt,
so dass für alle $a_1, \ldots, a_k \in \mathbb{N}$ gilt:

1. Wenn $f(a_1, \ldots, a_k)$ definiert ist, dann gibt es $c_1, \ldots, c_\ell, pz \in \mathbb{N}$ mit $pz \geqslant s$, so dass

$$\underbrace{(0, a_1, \ldots, a_k, 0, \ldots, 0, 0)}_{\text{Startkonfiguration von } P(a_1, \ldots, a_k)} \overset{P}{\to}{}^* \underbrace{(f(a_1, \ldots, a_k), c_1, \ldots, c_\ell, pz)}_{\text{Endkonfiguration}}$$

   (umgangssprachlich: die Berechnung von P mit Eingabe $(a_1, \ldots, a_k)$ hält mit Ausgabe $f(a_1, \ldots, a_k)$).

2. Wenn $f(a_1, \ldots, a_k)$ undefiniert ist, dann gilt für jede Konfiguration $c' = (c'_0, \ldots, c'_\ell, pz')$ mit $(0, a_1, \ldots, a_k, 0, \ldots, 0, 0) \overset{P}{\to}{}^* c'$, dass $pz' < s$

   (umgangssprachlich: die Berechnung von P mit Eingabe $(a_1, \ldots, a_k)$ hält nicht).

Die These von Church und Turing

Es gibt eine Vielzahl formaler Begriffe der Berechenbarkeit
(z.B. URM-Berechenbarkeit, Turing-Berechenbarkeit, μ-Rek., λ-B., Post-B., ...),
von denen man formal zeigen kann, dass sie äquivalent sind.

Man kennt keine Funktion, die

- „intuitiv“ berechenbar ist und
- nicht URM-berechenbar (...) ist.

These von Church und Turing

Die im intuitiven Sinne berechenbaren Funktionen
sind genau die URM-berechenbaren Funktionen.

8.3 Kodierung von URM-Programmen durch Zahlen

Im intuitiven Sinn sind folgende Funktionen berechenbar.

- gegeben: ein URM-Programm; gib die Länge des Programms aus
- gegeben: ein URM-Programm
  bestimme die Länge ℓ des Programms und ersetze alle Sprungziele $\geqslant \ell$ durch ℓ
  gib das geänderte Programm aus

Damit wir über die URM-Berechenbarkeit dieser Funktionen sprechen können,
müssen wir URM-Programme so darstellen, dass URM-Programme sie bearbeiten können:
als Zahlen.

Da die Programmzeilen eigentlich Zahlen sind,
müssen wir eine Kodierung von endlichen Folgen/Tupeln von Zahlen bestimmen.
Diese Kodierung soll einfach dekodierbar und manipulierbar sein.

Tupel $(r_1, r_2, \ldots, r_k) \in \mathbb{N}^k$ können kodiert werden mittels $\tau : \mathbb{N}^{k \geqslant 0} \to \mathbb{N}$ mit

$$\tau(r_1, r_2, \ldots, r_k) \;=\; p_0^k \cdot \prod_{i \geqslant 1} p_i^{r_i} \;,$$

wobei p_i die i-te Primzahl ist ($p_0 = 2$).

Bsp.: $(1, 2, 3, 5)$ wird kodiert durch $\tau(1, 2, 3, 5) = 2^4 \cdot 3^1 \cdot 5^2 \cdot 7^3 \cdot 11^5 = 66288591600$.

Lemma 8.5 (Das Inverse $\zeta(i, n)$ mit $\tau(\zeta(0, n), \ldots, \zeta(k, n)) = n$ ist URM-berechenbar)

Für jedes $i \in \mathbb{N}$ ist die folgende Funktion τ_i URM-berechenbar.

$$\zeta(i, n) = \begin{cases} r_i & \text{falls für } k = \max\{t \mid 2^t \text{ teilt } n\} \text{ gilt:} \\ & 1 \leqslant i \leqslant k \text{ und es gibt } r_1, \ldots, r_k \text{ mit } n = \tau(r_1, \ldots, r_k) \\ \text{undefiniert}, & \text{sonst} \end{cases}$$

Die Zeilen von URM-Programmen sind ja Zahlen.
Wir schreiben sie allerdings „umgangssprachlich lesbar“ auf.

Zeile	
umgangssprachlich	als Zahl
$\mathtt{R_a} = 0$	$2^0 \cdot 3^a$
$\mathtt{R_a}$ += 1	$2^1 \cdot 3^a$
if $\mathtt{R_a} == \mathtt{R_b}$: goto c	$2^2 \cdot 3^a \cdot 5^b \cdot 7^c$

Ein URM-Programm $P = (I_0, I_1, \ldots, I_{s-1})$ ist ein Tupel von Zahlen.

Seine Kodierung als natürliche Zahl ist $\tau(P) = \tau(I_0, I_1, \ldots, I_{s-1})$.

Beispiel: von der Zahl zum URM-Programm und weiter zur Funktion

Zahl n: 408580270875000

$= 2^3 \cdot 51072533859375$

$= 2^3 \cdot 3^4 \cdot 630525109375$

$= 2^3 \cdot 3^4 \cdot 5^6 \cdot 40353607$

$= 2^3 \cdot 3^4 \cdot 5^6 \cdot 7^9$

$= \tau(4, 6, 9)$

URM-Programm $\tau^{-1}(n)$: (auch als P_n bezeichnet)

```
if R0 == R0 : goto 0
R1 += 1
R2 = 0
```

Die von P_n berechnete Funktion wird mit φ_n bezeichnet.

Die Funktion $\varphi^{(1)}_{408580270875000} : \mathbb{N} \to \mathbb{N}$ ist die überall undefinierte Funktion,

d.h. für alle $n \in \mathbb{N}$ gilt: $\varphi^{(1)}_{408580270875000}(n)$ ist undefiniert.

Da τ nicht surjektiv ist, hat τ kein Inverses auf $\mathbb{N}$.

Wir bezeichnen mit τ^{-1}_{URM} eine Erweiterung von τ^{-1},
die jede natürliche Zahl auf ein URM-Programm abbildet.

$\tau^{-1}_{URM}(n) = (I_1, \ldots, I_s)$ mit

$$s = \max\{i \mid 2^i \text{ teilt } n\} \qquad \text{und}$$

$$I_j = \begin{cases} \max\{i \mid p_j^i \text{ teilt } n\}, & \text{falls das Maximum } i \\ & \text{eine Programmzeile ist} \\ \texttt{if } R_0 == R_0 : \texttt{goto j}, & \text{sonst} \quad \text{(Endlosschleife)} \end{cases}$$

Z.B. ist $\tau^{-1}_{URM}(n)$ für jedes ungerade n das leere Programm.

Das stört uns nicht, da uns ausreicht, dass τ injektiv ist.

Definition 8.6 (Bezeichnungen von Programmen und Funktionen)

Sei $n \in \mathbb{N}$.

- P_n ist das URM-Programm mit Kodierung n.
  Also ist $P_n = \tau_{URM}^{-1}(n)$.

  (Für jedes $n \in \mathbb{N}$ gibt es ein URM-Programm P_n.)

- $\varphi_n^{(k)}$ bezeichnet die Funktion $\mathbb{N}^k \to \mathbb{N}$,

  die durch das URM-Programm P_n berechnet wird.

- Für $\varphi_n^{(1)}$ schreiben wir auch φ_n.

8.4 Eine Funktion, die nicht URM-berechenbar ist

Definiere $\psi : \mathbb{N} \to \mathbb{N}$ mit

$$\psi(n) = \begin{cases} 0, & \text{falls } \varphi_n(n) = 1 \\ 1, & \text{sonst} \end{cases}$$

Satz 8.7

ψ ist nicht URM-berechenbar.

Beweis:

Annahme: ψ ist URM-berechenbar, d.h. $\psi = \varphi_m$ für ein $m \in \mathbb{N}$.

Dann gilt

$$\psi(m) = 1 \overset{\text{Def. } \psi}{\Rightarrow} \varphi_m(m) \neq 1 \overset{\varphi_m = \psi}{\Rightarrow} \psi(m) \neq 1 \quad \text{und}$$
$$\psi(m) \neq 1 \Rightarrow \varphi_m(m) = 1 \Rightarrow \psi(m) = 1.$$

Insgesamt: $\psi(m) = 1 \Leftrightarrow \psi(m) \neq 1$ ☞ □

8.5 Universelle Programme

Satz 8.8

Es gibt ein universelles URM-Programm U,
das beliebige URM-Programme simulieren kann.

Das heißt

- *$U(i, x)$ hält genau dann, wenn $P_i(x)$ hält, und*
- *wenn $P_i(x)$ hält, dann liefert $U(i, x)$ das gleiche Ergebnis in Register R_0.*

Den Satz kann man mit Benutzung der These von Church und Turing leicht beweisen.

Im folgenden Exkurs wird ein universelles URM-Programm konstruiert
(also die Beweisidee ohne Benutzng der These von Church und Turing).

Was haben wir in dieser Vorlesung gelernt?

- Wir kennen das URM-Modell zur Berechnung totaler und partieller Funktionen.
- Für jede intuitiv berechenbare Funktion gibt es ein URM-Programm, das sie berechnet.
- Jedes URM-Programm lässt sich durch eine natürliche Zahl darstellen.
  Also lassen sich von URM-Programmen auch URM-Programme berechnen.
- Es gibt ein universelles URM-Programm.
- Es gibt Funktionen, die nicht URM-berechenbar sind.

Exkurs: Das universelle Programm $U^{(k)}$

simuliert die Berechnung k-stelliger Funktionen.

Es erhält als Eingabe $e, x_1, \ldots, x_k$

und berechnet die gleiche Ausgabe wie $P_e(x_1, \ldots, x_k)$ – bzw. hält nicht.

Der Algorithmus arbeitet grob gesagt wie folgt.

Eingabe $e, x_1, \ldots, x_k$
der Programmcode steht bereits in R_1
schreibe den Registercode von $0, x_1, \ldots, x_k$ nach R_2
schreibe den Programmzähler 0 nach R_3
solange der Programmzähler $R_3 <$ Anzahl Zeilen des Programms in R_1:
    simuliere den durch den Programmzähler bestimmten Befehl
      des Programmcodes in R_1
      auf dem Registercode in R_2
      und setze den Programmzähler R_3 entsprechend um
schreibe den Inhalt von Register 0 aus dem Registercode nach R_0

Der Programmcode und der Registercode

Der *Programmcode* eines Programms $(I_0, I_1, \ldots, I_{k-1})$ ist $\tau(I_0, I_1, \ldots, I_p)$.

Die Anzahl der Zeilen des Programms mit Programmcode m ist $\zeta(0, m)$. (8.5)

Der *Registercode* kodiert die Inhalte aller Register einer URM.

Jedes Programm benutzt eine feste Zahl $R_0, R_1, R_2, \ldots, R_k$ von Ein- und Ausgaberegistern.

Alle Registerinhalte $r_0, r_1, r_2, \ldots, r_k$ beim Start des Programms

können als $\prod_{i=0}^{k} p_i^{r_i}$ kodiert werden.

Damit sind auch die übrigen Register mit Inhalt 0 korrekt kodiert.

Lies den Inhalt eines Registers aus dem Registercode in R_2

Dekodiere aus dem Registercode in R_2 den Inhalt des Registers,
dessen Index in R_b steht,
und schreibe ihn nach R_c.

```
Rt = R2
Rc = 0
Ru = Rb-te Primzahl
while Rt mod Ru== 0 :
  Rt = Rt / Ru
  Rc += 1
```

Abk.: R_c = Register R_{R_b} aus Registercode R_2

Schreibe den Inhalt eines Registers im Registercode in R_a

Schreibe im Registercode in R_a den Inhalt von R_c in das Register, dessen Index in R_b steht.

```
R_t = R_b-te Primzahl        (Teil 1: setze das Register im Registercode auf 0)
while !(R_a mod R_t==0) :
  R_a = R_a / R_t
R_s = R_c                    (Teil 2: setze das Register im Registercode auf R_c)
while R_s :
  R_a = R_a · R_t
  R_s = R_s - 1
```

Abk.: R_a = schreibe R_c als Reg. R_{R_b} in R_a

Berechne den Registercode beim Start der Simulation

Der Inhalt der Register $R_2, \ldots, R_{k+1}$ wird als Registercode
(für Register $(R_0,)R_1, \ldots, R_k$) nach R_2 geschrieben.

$\mathtt{R_t} = 1$ (am Anfang sind alle Register 0, also ist der Reg.code 1)

$\mathtt{R_i} = 2$ (der Index des nächsten zu kodierenden Registers)

$\mathtt{R_b} = 1$ (der Index des nächsten Registers im Konfig.code)

$\mathtt{R_t} = \mathtt{schreibe\ R_i\ als\ Reg.\ R_{R_b}\ in\ R_t}$

$\mathtt{R_i} = 3$ (der Index des nächsten zu kodierenden Registers)

$\mathtt{R_b} = 2$ (der Index des nächsten Registers im Konfig.code)

$\mathtt{R_t} = \mathtt{schreibe\ R_i\ als\ Reg.\ R_{R_b}\ in\ R_t}$

$\vdots$

$\mathtt{R_i} = \mathtt{k} + 1$ (der Index des nächsten zu kodierenden Registers)

$\mathtt{R_b} = \mathtt{k}$ (der Index des nächsten Registers im Konfig.code)

$\mathtt{R_t} = \mathtt{schreibe\ R_i\ als\ Reg.\ R_{R_b}\ in\ R_t}$

$R_2 = \mathtt{R_t}$

Abk.: $\mathtt{R_2} = \mathtt{Kodierung\ von\ 0, R_2, ..., R_{k+1}}$

Dekodierung des Zeilencodes

Jede Zeile eines URM-Programms wird als Zahl kodiert (*Zeilencode*).

Zeile

umgangssprachlich	als Zahl
$\mathtt{R_a = 0}$	$2^0 \cdot 3^a$
$\mathtt{R_a \mathrel{+}= 1}$	$2^1 \cdot 3^a$
$\mathtt{if\ R_a == R_b : goto\ c}$	$2^2 \cdot 3^a \cdot 5^b \cdot 7^c$

Der Programmcode ist die Kodierung eines Tupels von Zeilencodes.

Das universelle Programm dekodiert den Zeilencode,
indem es seine Bestandteile in die Register R_4, R_5, R_6 und R_7 schreibt.

Bsp.:
nach der Dekodierung von $\mathtt{if\ R_{10} == R_8 : goto\ 6}$
ist $R_4 = 2$, $R_5 = 10$, $R_6 = 8$ und $R_7 = 6$
nach der Dekodierung von $\mathtt{R_{13} \mathrel{+}= 1}$
ist $R_4 = 1$, $R_5 = 13$, $R_6 = 0$ und $R_7 = 0$

Dekodiere aus dem Programmcode in R_a die Zeile, deren Nummer in R_b steht, und schreibe sie nach R_c.

```
Rt = Ra
Rc = 0
Rp = Rb+1
Ru = Rp-te Primzahl
while Rt mod Ru== 0 :
  Rt = Rt / Ru
  Rc += 1
```

Abk.: `Rc = Zeile Rb aus Programmcode Ra`

Dekodiere aus dem Programmcode in R_1 die Zeile,
deren Nummer in R_b steht,
und schreibe ihre Bestandteile nach $R_4, \ldots, R_7$.

R_c = `Zeile` R_b `aus Programmcode` R_1
$R_s = 2$
R_4 = `maxPotenz von` R_s `in` R_c
$R_s = 3$
R_5 = `maxPotenz von` R_s `in` R_c
$R_s = 5$
R_6 = `maxPotenz von` R_s `in` R_c
$R_s = 7$
R_7 = `maxPotenz von` R_s `in` R_c
(falls der Zeilencode nicht korrekt war, wird die Simulation in eine Endlosschleife gehen)
`if` $R_c/(2^{R_4} \cdot 3^{R_5} \cdot 5^{R_6} \cdot 7^{R_7}) > 1$ `or (` $R_4 < 2$ `and` $R_6 + R_7 > 0$ `):`
$R_4 = 3$

Abk.: `decode next line from program code in` R_1

Simulation der einzelnen URM-Befehle: $\mathtt{R}_x = 0$

$\mathrm{sim}[\mathtt{R}_{\mathtt{R}_5} = 0]$:

simuliere $\mathtt{R}_{\mathtt{R}_5} = 0$, d.h.
setze im Registercode in R_2 den Inhalt des Registers,
dessen Index in R_5 steht,
auf 0.

$\mathtt{R}_z = 0$
$\mathtt{R}_2 =$ `schreibe` $\mathtt{R}_z$ `als Reg.` $\mathtt{R}_{\mathtt{R}_5}$ `in` $\mathtt{R}_2$
$\mathtt{R}_3$ += 1 (Programmzähler+1)

Simulation der einzelnen URM-Befehle: $\mathtt{R_x}$ += 1

sim$[\mathtt{R_{R_5}} \mathrel{+}= 1]$:

simuliere $\mathtt{R_{R_5}} \mathrel{+}= 1$, d.h.
addiere 1 im Registercode in R_2 zum Inhalt des Registers,
dessen Index in R_5 steht.

$\mathtt{R_t} = \mathtt{R_5}$-te Primzahl
$\mathtt{R_2} = \mathtt{R_2} \cdot \mathtt{R_t}$
$\mathtt{R_3} \mathrel{+}= 1$ (Programmzähler+1)

Simulation der URM-Befehle: if $R_x == R_y$: goto z

sim[if $R_{R_5} == R_{R_6}$: goto R_7]:

simuliere if $R_b == R_c$: goto d, d.h.
setze den Programmzähler in R_3 auf den Inhalt von R_7,
falls im Registercode in R_2 die Register,
    deren Indizes in R_5 und R_6 stehen,
den gleichen Inhalt haben.
Anderenfalls wird nur der Programmzähler hochgezählt.

```
Ra = Register R_R5 aus Registercode R2
Rb = Register R_R6 aus Registercode R2
if  Ra==Rb :
  R3 = R7
else :
  R3 += 1
```

Das universelle Programm $U^{(k)}$

$U^{(k)}(e, x_1, \ldots, x_k)$ simuliert $P_e(x_1, \ldots, x_k)$
und berechnet $\varphi_e(x_1, \ldots, x_k)$.

(bereite die Simulation vor)
R_2 = Kodierung von $0, R_2, ..., R_{k+1}$
$R_3 = 0$ (setze Programmzähler auf 0)
$R_8 = 2$ (wir brauchen zweimal eine 2)

(simuliere jeden Schritt des Programms)
while $R_3 <$ maxPotenz von R_8 in R_1 :
  decode next line from program code in R_1
  if R_4==0 : sim[$R_{R_5} = 0$]
  if R_4==1 : sim[R_{R_5} += 1]
  if R_4==2 : sim[if $R_{R_5} == R_{R_6}$: goto R_7]

(schreibe das Ergebnis der Simulation nach R_0)
R_0 = maxPotenz von R_4 in R_8

Vorlesung 9:
Entscheidbare und semi-entscheidbare Mengen

Stelligkeit von Funktionen

Wir haben den Begriff der Berechenbarkeit formal
für Funktionen $\mathbb{N}^k \to \mathbb{N}$ definiert.

Wir hätten das genauso für Funktionen $\mathbb{N}^k \to \mathbb{N}^\ell$ machen können.

Da man alle k-Tupel über $\mathbb{N}$ mittels der Tupelkodierung τ
als natürliche Zahlen kodieren kann,
unterscheiden wir von jetzt ab nicht zwischen Tupeln und Zahlen.

Wir werden in Zukunft die *Tupel-Konvention* benutzen:
Was für Zahlen geht, das geht auch für Tupel.

9.1 Entscheidbare Mengen

Wir wollen ja mal zeigen, dass die Menge der Formeln, die man z.B. mit den Peano-Axiomen beweisen kann, nicht die Menge der Formeln ist, die vom Standardmodell der natürlichen Zahlen erfüllt werden.

Also sind wir an Eigenschaften von *Mengen* interessiert.

In diesem Abschnitt wird die Unterscheidung zwischen totalen berechenbaren, nicht-totalen berechenbaren und nicht-berechenbaren Funktionen übertragen auf Mengen – es wird zwischen *entscheidbaren*, *semi-entscheidbaren* und *nicht-semi-entscheidbaren* Mengen unterschieden.

Letztlich interessieren uns fast nur die „schwierigeren“ nicht-entscheidbaren Mengen.
Wir werden ein paar Mengen mit dieser Eigenschaft kennenlernen.

Definition 9.1 (entscheidbare Mengen)

Eine Menge $A \subseteq \mathbb{N}^k$ heißt entscheidbar,
falls die *charakteristische Funktion* $\chi_A : \mathbb{N}^k \to \{0,1\}$ mit

$$\chi_A(n_1, \ldots, n_k) = \begin{cases} 1, & \text{falls } (n_1, \ldots, n_k) \in A \\ 0, & \text{falls } (n_1, \ldots, n_k) \notin A \end{cases}$$

berechenbar ist.

Beispiel:

$\{n \mid n \text{ ist Primzahl}\}$ ist entscheidbar, da
das URM-Programm
$\mathtt{R}_0$ = "$\mathtt{R}_1$ `ist Primzahl`"
die charakteristische Funktion χ_{prim} berechnet.

$$\chi_{prim}(n) = \begin{cases} 1, & \text{falls } n \text{ eine Primzahl ist} \\ 0, & \text{falls } n \text{ keine Primzahl ist} \end{cases}$$

Komplemente entscheidbarer Mengen

$\{n \mid n$ ist keine Primzahl$\}$ ist entscheidbar.
Die charakteristische Funktion wird durch folgendes URM-Programm berechnet.

$R_0 =$ "R_1 ist Primzahl"
$R_0 = \overline{\mathtt{sig}}(R_0)$

Lemma 9.2

Wenn A entscheidbar ist, dann ist auch $\overline{A}$ entscheidbar.

Beweis.
Sei P_A das Programm für die charakteristische Funktion von A.
Dann berechnet folgendes Programm die charakteristische Funktion von $\overline{A}$.

P_A
$R_0 = \overline{\mathtt{sig}}(R_0)$

□

Definition 9.3 (Das beschränkte Halteproblem)

$H_B^{(k)} := \{(e, x_1, \ldots, x_k, t) \mid P_e(x_1, \ldots, x_k) \text{ hält nach } \leqslant t \text{ Schritten}\}$

Lemma 9.4

$H_B^{(k)}$ ist entscheidbar.

Beweis:

Das Universelle Programm lässt sich so modifizieren, dass

(1) die Anzahl der simulierten Schritte gezählt und

(2) die Schrittsimulationsschleife nach $\leqslant t$ Wiederholungen verlassen wird.

Wenn dann der Programmzähler kleiner als die Programmlänge ist,
dann hat das simulierte Programm nicht nach t Schritten gehalten.
Anderenfalls hat es nach $\leqslant t$ Schritten gehalten. □

Weitere Entscheidungsprobleme

die wir noch betrachten werden

Definition 9.5 (Halteprobleme)

- das spezielle Halteproblem $K := \{n \mid P_n(n) \text{ hält}\}$
- das allgemeine Halteproblem $H := \{(n, m) \mid P_n(m) \text{ hält}\}$
- das 0-Halteproblem $H_0 := \{n \mid P_n(0) \text{ hält}\}$

Definition 9.6

- $TOTAL := \{n \mid \varphi_n \text{ ist eine totale Funktion}\}$

Satz 9.7

Das spezielle Halteproblem K ist nicht entscheidbar.

Beweis:
Annahme: K ist entscheidbar.
Also gibt es ein Programm S, das χ_K berechnet.
Daraus lässt sich folgendes Programm Q konstruieren.

```
            S                                   ( R_0 = χ_K(n) )
|S|:       R_1 = 0
|S| + 1:   if ! (R_0==R_1) : goto |S| + 1 :     ( if R_0 ≠ 0: Endlosschleife )
```

Q habe die Kodierung m_0, d.h. $Q = P_{m_0}$. Dann gilt:

1.) $Q(m_0)$ hält $\overset{\text{Konstr. von } Q}{\Rightarrow}$ $S(m_0) = 0$ $\overset{S \text{ berechnet } \chi_K}{\Rightarrow}$ $P_{m_0}(m_0)$ hält nicht

2.) $Q(m_0)$ hält nicht $\Rightarrow$ $S(m_0) = 1$ $\Rightarrow$ $P_{m_0}(m_0)$ hält

Also gilt: $Q(m_0)$ hält gdw. $Q(m_0)$ hält nicht. ☞

Folglich kann es das Programm S nicht geben,
d.h. K ist nicht entscheidbar. □

Satz 9.7

Das spezielle Halteproblem K ist nicht entscheidbar.

Beweis:
Annahme: K ist entscheidbar.
Also gibt es ein Programm S, das χ_K berechnet.
Daraus lässt sich folgendes Programm Q konstruieren.

	S	($R_0 = \chi_K(n)$)
$\lvert S\rvert$:	$\mathtt{R}_1 = 0$	
$\lvert S\rvert + 1$:	$\mathtt{if}\ !\,(\mathtt{R}_0 == \mathtt{R}_1)\ :\ \mathtt{goto}\ \lvert S\rvert + 1$:	(if $R_0 \neq 0$: Endlosschleife)

Q habe die Kodierung m_0, d.h. $Q = P_{m_0}$. Dann gilt:

1.)	$Q(m_0)$ hält	$\overset{\text{Konstr. von } Q}{\Rightarrow}$	$S(m_0) = 0$	$\overset{S \text{ berechnet } \chi_K}{\Rightarrow}$	$Q(m_0)$ hält nicht
2.)	$Q(m_0)$ hält nicht	$\Rightarrow$	$S(m_0) = 1$	$\Rightarrow$	$P_{m_0}(m_0)$ hält

Also gilt: $Q(m_0)$ hält gdw. $Q(m_0)$ hält nicht. 

Folglich kann es das Programm S nicht geben,
d.h. K ist nicht entscheidbar. □

Satz 9.7

Das spezielle Halteproblem K ist nicht entscheidbar.

Beweis:
Annahme: K ist entscheidbar.
Also gibt es ein Programm S, das χ_K berechnet.
Daraus lässt sich folgendes Programm Q konstruieren.

	S	($R_0 = \chi_K(n)$)
$\lvert S\rvert$:	$\mathtt{R}_1 = 0$	
$\lvert S\rvert + 1$:	$\mathtt{if}\ !\,(\mathtt{R}_0 == \mathtt{R}_1)\ :\ \mathtt{goto}\ \lvert S\rvert + 1$:	(if $R_0 \neq 0$: Endlosschleife)

Q habe die Kodierung m_0, d.h. $Q = P_{m_0}$. Dann gilt:

1.)	$Q(m_0)$ hält	$\overset{\text{Konstr. von } Q}{\Rightarrow}$	$S(m_0) = 0$	$\overset{S \text{ berechnet } \chi_K}{\Rightarrow}$	$Q(m_0)$ hält nicht
2.)	$Q(m_0)$ hält nicht	$\Rightarrow$	$S(m_0) = 1$	$\Rightarrow$	$Q(m_0)$ hält

Also gilt: $Q(m_0)$ hält gdw. $Q(m_0)$ hält nicht. ☞

Folglich kann es das Programm S nicht geben,
d.h. K ist nicht entscheidbar. □

Satz 9.8

Das allgemeine Halteproblem H ist nicht entscheidbar.

Beweis:
Annahme: H ist entscheidbar.
Dann gibt es ein Programm S, das χ_H berechnet.
Daraus lässt sich folgendes Programm Q konstruieren.

$\mathtt{R}_2 = \mathtt{R}_1$
S

Es gilt $\varphi_{\tau(Q)}(n) = \chi_K(n, n)$ für alle $n \in \mathbb{N}$.

Also ist $\varphi_{\tau(Q)} = \chi_K$.

Folglich ist K entscheidbar.
Das widerspricht (9.7). 
Also kann H nicht entscheidbar sein. □

Satz 9.9

Das 0-Halteproblem H_0 ist nicht entscheidbar.

Beweis:
Für jedes n gilt:
$P_n(n)$ hält genau dann, wenn folgendes Programm $Q_{(n)}$ mit Eingabe 0 hält.

$\mathtt{R_1 = n}$
P_n

Die Funktion f mit

$$f(n) = \tau\big(\mathtt{R_1 = 0}, \underbrace{\mathtt{R_1 \mathrel{+}= 1}, \ldots, \mathtt{R_1 \mathrel{+}= 1}}_{n \text{ Zeilen}}, \underbrace{1.\text{Zeile von } P_n, \ldots, \text{letzte Zeile von } P_n}_{\text{Zeilen von } P_n}\big)$$

ist total und berechenbar — sie berechnet $f(n) = \tau(Q_{(n)})$.

Also gilt für jedes $n \in \mathbb{N}$: $\quad n \in K$ genau dann, wenn $f(n) \in H_0$.
Wenn H_0 entscheidbar ist, dann ist folglich auch K entscheidbar.
Da K nicht entscheidbar ist (9.7), ist folglich H_0 nicht entscheidbar. □

9.2 Semi-entscheidbare Mengen

Der Begriff der Entscheidbarkeit reicht noch nicht, um zwischen den Mengen der aus den Peano-Axiomen herleitbaren Formeln und den vom Standardmodell erfüllten Formeln zu unterscheiden. Wir werden später sehen, dass beide Mengen nicht entscheidbar sind.

Um einen Unterschied zwischen den beiden Mengen zu finden, brauchen wir einen allgemeineren Begriff, der mehr Mengen umfasst:
die *Semi-Entscheidbarkeit*.

Wir werden sehen, dass das spezielle Halteproblem K diese Eigenschaft hat,
während sein Komplement $\overline{K}$ sie nicht hat.

Das ist ein Schlüssel, um einen „einfachen“ Unterschied zwischen den beiden o.g. Formelmengen zu finden.

Definition 9.10 (semi-entscheidbare Mengen)

Eine Menge $A \subseteq \mathbb{N}^k$ heißt semi-entscheidbar,
falls die partielle charakteristische Funktion $\chi'_A : \mathbb{N}^k \to \{1\}$ mit

$$\chi'_A(n_1, \ldots, n_k) = \begin{cases} 1, & \text{falls } (n_1, \ldots, n_k) \in A \\ \text{undefiniert}, & \text{falls } (n_1 \ldots, n_k) \notin A \end{cases}$$

berechenbar ist.

Beispiel: die partielle charakteristische Funktion χ'_K für das spezielle Halteproblem K ist

$$\chi'_K(n) = \begin{cases} 1, & \text{falls } P_n(n) \text{ hält} \\ \text{undefiniert}, & \text{falls } P_n(n) \text{ nicht hält} \end{cases}$$

Lemma 9.11

Das spezielle Halteproblem K ist semi-entscheidbar.

Beweis:
Die Funktion χ'_K für $K = \{x \mid P_x(x) \text{ hält}\}$ wird vom folgenden Programm Q berechnet.

$\mathtt{R}_2 = \mathtt{R}_1$
$U^{(1)}$ (simuliere $P_x(x)$)
$\mathtt{R}_0 = 1$ (sobald die Simulation hält, wird 1 ausgegeben)

$Q(n)$ startet $U^{(1)}(n, n)$ und simuliert dadurch $P_n(n)$.

Falls $P_n(n)$ hält, wird die Programmzeile $\mathtt{R}_0 = 1$ erreicht;
dadurch wird Funktionswert 1 ausgegeben.
(Wir haben den $U^{(1)}$-Teil so eingefügt, dass alle Sprünge aus dem Teil heraus in die Zeile direkt dahinter gehen.)

Falls $P_n(n)$ nicht hält, wird die Programmzeile $\mathtt{R}_0 = 1$ nicht erreicht und $Q(n)$ hält nicht.
Entsprechend ist der Funktionswert undefiniert. □

Lemma 9.12

Jede entscheidbare Menge ist semi-entscheidbar.

Beweis:

Sei A entscheidbar, und P_A berechne χ_A.
Dann wird χ'_A vom folgenden Programm berechnet.

0:	P_A	$R_0 = \chi_A(x)$
$\lvert P_A \rvert$:	$\mathtt{R}_1 = 0$	
$\lvert P_A \rvert + 1$:	$\mathtt{if\ R}_0 == \mathtt{R}_1 : \mathtt{goto}\ \lvert \mathtt{P}_\mathtt{A} \rvert + 1$	if $R_0 = 0$: Endlosschleife

□

Satz 9.13 (Entscheidbarkeit vs. Semi-Entscheidbarkeit)

A ist entscheidbar genau dann, wenn A und $\overline{A}$ semi-entscheidbar sind.

Beweis:
"$\Rightarrow$": wenn A entscheidbar ist, dann ist auch $\overline{A}$ entscheidbar (9.2).
Und entscheidbare Mengen sind auch semi-entscheidbar (9.12).

„$\Leftarrow$": seien A und $\overline{A}$ semi-entscheidbar, und P_a und P_b seien Programme für χ'_A bzw. $\chi'_{\overline{A}}$.
Dann berechnet folgender Algorithmus χ_A.

Eingabe x
$z := 0$
solange $\underbrace{\text{weder } P_a(x) \text{ noch } P_b(x) \text{ nach} \leqslant z \text{ Schritten hält}}_{(a,x,z) \notin H_B \text{ und } (b,x,z) \notin H_B}$: $z := z + 1$
falls $\underbrace{P_a(x) \text{ nach} \leqslant z \text{ Schritten hält}}_{(a,x,z) \in H_B, \text{ d.h. } \chi'_A(x) = 1}$ dann Ausgabe 1 sonst Ausgabe 0

Der Algorithmus hält bei jeder Eingabe x (d.h. er ist total), da $x \in A$ oder $x \in \overline{A}$:
d.h. für jedes x gibt es ein t mit $(a,x,t) \in H_B$ (d.h. $x \in A$) oder $(b,x,t) \in H_B$ (d.h. $x \in \overline{A}$).
Er gibt 1 aus, wenn $\chi'_A(x) = 1$ — also wenn $x \in A$, und
gibt 0 aus, wenn $\chi'_{\overline{A}}(x) = 1$ — also wenn $x \notin A$. □

Satz 9.14

$\overline{K}$ ist nicht semi-entscheidbar.

Beweis:
Nach (9.13) gilt:
wenn K nicht entscheidbar ist,
dann ist K nicht semi-entscheidbar oder $\overline{K}$ ist nicht semi-entscheidbar.

Da K nicht entscheidbar (9.7) und semi-entscheidbar (9.11) ist,
folgt also, dass $\overline{K}$ nicht semi-entscheidbar ist. □

Was haben wir in dieser Vorlesung gelernt?

- Wir kennen die Begriffe „entscheidbar“ und „semi-entscheidbar“.
- Wir kennen die Mengen H_B, K und $\overline{K}$
  und wissen, dass
  - H_B entscheidbar ist,
  - K semi-entscheidbar und nicht entscheidbar ist, und
  - $\overline{K}$ weder entscheidbar noch semi-entscheidbar ist.
- Wir kennen den Beweis für die Nicht-Entscheidbarkeit von K
  und kennen eine Vorgehensweise,
  wie man daraus die Nicht-Entscheidbarkeit anderer Mengen folgern kann.
- Wir kennen den Beweis für die Nicht-Semi-Entscheidbarkeit von $\overline{K}$.

Vorlesung 10:
Mengen beweisbarer Formeln sind semi-entscheidbar

VL 10: Mengen beweisbarer Formeln sind semi-entscheidbar

Wir zeigen in dieser Vorlesung, dass die Mengen der Formeln, die man mit Natürlichem Schließen und semi-entscheidbarer Axiomenmenge herleiten kann, semi-entscheidbar sind.
Außer *Natürlichem Schließen* gibt es noch viele andere Beweissysteme,
wie z.B. *Frege-Kalkül*, *Tableau-Kalkül*, *Resolution*.

Diese Beweissysteme haben die Gemeinsamkeit, dass

- jeder Beweis ein „endlicher Text“ ist,
  an dem man einfach erkennen kann, was bewiesen wird, und
- von jedem endlichen Text einfach entschieden werden kann, ob er ein Beweis ist.

Ein Beweissystem kann man sich also vorstellen als einen Algorithmus,
der für eine gegebene Formel einen Beweis sucht.
Das ist ein typischer Algorithmus für ein semi-entscheidbares Problem.

Man kann sich ein Beweissystem auch vorstellen als eine Funktion,
die einen „endlichen Text“ auf das abbildet, was er beweist.
Wir werden sehen, dass das genauso typisch für ein semi-entscheidbares Problem ist.

Damit wir über Formeln und Beweise im Zusammenhang mit unserem Berechnungsmodell sprechen können, müssen wir sie erstmal als natürliche Zahlen kodieren.
Eine Möglichkeit ist es, die Formel als LaTeX-String auzuschreiben,
den ASCII-Code der einzelnen Zeichen als Tupel zu nehmen
und mit der Tupel-Kodierung zu kodieren.

Bsp:

Formel	$\forall x_1(0 + x_1 = x_1)$
LaTeX-String	`\forall x_{1} ( 0 + x_{1} = x_{1} )`
Zahlen-Tupel	[92, 102, 111, 114, 97, 108, 108, 32, 120, 95, 123, 49, 125, 32, 40, 32, 48, 32, 43, 32, 120, 95, 123, 49, 125, 32, 61, 32, 120, 95, 123, 49, 125, 32, 41]
Zahl	$70997702668970256027 \cdots 250$ (4506 Stellen) $000000000000000000000000000000000000$

Diese Kodierung und die Dekodierung sind intuitiv berechenbar.

(Semi-)Entscheidbarkeit ist für Mengen (von Tupeln) natürlicher Zahlen definiert.
Im Rest der Vorlesung werden wir Mengen aus Formeln, Folgen von Sequenten etc. betrachten.
Wenn wir über ihre (Semi-)Entscheidbarkeit sprechen,
stellen wir uns alles als Zahlen kodiert vor.

Definition 10.1 (Mengen arithmetischer Formeln)

$\mathcal{Q}$-BEWEISBAR $:= \{ \alpha \mid \alpha$ ist arithmetische Σ_1-Formel mit $\mathcal{Q} \vdash_{\mathcal{P}} \alpha \}$

PA-BEWEISBAR $:= \{ \alpha \mid \alpha$ ist arithmetische Formel mit PA $\vdash_{\mathcal{P}} \alpha \}$

$\mathfrak{N}$-SAT $:= \{ \alpha \mid \alpha$ ist arithmetische Formel mit $\mathfrak{N} \models_{\mathcal{P}} \alpha \}$

10.1 Peano-Arithmetik ist semi-entscheidbar

Zuerst werden wir sehen, dass die Menge der Formeln, die man mit den Peano-Axiomen *PA* herleiten kann, semi-entscheidbar ist.

Das ist nur so eine Art „einführendes Beispiel“ dafür,
dass alle Mengen von Formeln, die man mit semi-entscheidbaren Axiommengen herleiten kann, semi-entscheidbar sind.

Für jede Axiommenge definieren wir die Menge korrekter Beweise und bewiesener Formeln als *Beweis-Prädikat*.

Definition 10.2 (das Beweis-Prädikat für die Peano-Arithmetik)

$$Bew_{PA} := \left\{ ((s_1, \ldots, s_m), \alpha) \;\middle|\; \begin{array}{l} (s_1, \ldots, s_m) \text{ ist } \vdash_{\mathcal{P}}\text{-Herleitung eines Sequenten} \\ \underbrace{\Gamma \blacktriangleright \alpha}_{=s_m} \text{ mit } \Gamma \subseteq PA \end{array} \right\}$$

Lemma 10.3

Bew_{PA} ist entscheidbar.

Beweis:
Das folgende Programm berechnet $\chi_{Bew_{PA}}$.

Eingabe $(s_1, \ldots, s_m), \alpha$ $((s_1, \ldots, s_m)$ ist eine Sequentenfolge für beliebiges m, α ist eine Formel)
für $i = 1, \ldots, m$ wiederhole
überprüfe, ob der Sequent s_i mittels einer Regel $(\to I)$, $(\to E)$, (RAA), (Hyp),
$(= E)$, $(\forall I)$, $(\forall E)$ aus vorhergehenden s_j entsteht
oder ein Axiom $\varphi \blacktriangleright \varphi$ oder $\blacktriangleright \tau = \tau$ ist
falls alle Überprüfungen positiv waren
und s_m ein Sequent der Form $\Gamma \blacktriangleright \alpha$ ist
und für jedes $\gamma \in \Gamma$ gilt $\gamma \in PA$
dann: Ausgabe 1
sonst: Ausgabe 0

Das Programm überprüft zuerst, ob die Herleitung $s_1, \ldots, s_m$ korrekt ist.
Dieser Teil der Berechnung endet nach endlicher Zeit.

Anschließend wird der letzte hergeleitete Sequent $s_m = \Gamma \blacktriangleright \beta$ betrachtet
und es wird $\beta = \alpha$ überprüft.
Γ ist eine endliche Menge.
Für jedes Element in Γ wird überprüft, ob es in *PA* ist.
Da *PA* eine entscheidbare Menge ist, geht das in endlicher Zeit.

Falls die Herleitung $s_1, \ldots, s_m$ eine korrekte Herleitung eines Sequenten $\Gamma \blacktriangleright \alpha$ ist
und Γ aus Axiomen der Peano-Arithmetik besteht,
dann hält das Programm mit Ausgabe 1.
Anderenfalls hält das Programm mit Ausgabe 0.

Also berechnet der Algorithmus die charakteristische Funktion von Bew_{PA}. □

Lemma 10.4

PA-BEWEISBAR *ist semi-entscheidbar.*

Beweis:
Das folgende Programm berechnet $\chi'_{\text{PA-BEWEISBAR}}$.

Eingabe α (α ist eine Formel)
$i := 0$
solange $(i, \alpha) \notin Bew_{PA}$:
    $i := i + 1$
Ausgabe 1

Da Bew_{PA} entscheidbar ist (10.3),
hält das Programm mit Ausgabe 1, falls die Eingabe α in PA-BEWEISBAR ist.
Anderenfalls hält das Programm nicht.
Also berechnet das Programm $\chi'_{\text{PA-BEWEISBAR}}$. □

Lemma 10.5 (Mengen von aus semi-entscheidbaren Axiomen beweisbare Formeln sind semi-entscheidbar)

Sei T semi-entscheidbar und T-BEWEISBAR $= \{\alpha \mid T \vdash_{\mathcal{P}} \alpha\}$.
Dann ist T-BEWEISBAR semi-entscheidbar.

Beweis:
Da T semi-entscheidbar ist, ist das Beweisprädikat

$$Bew_T := \left\{ ((s_1, \ldots, s_m), \alpha) \;\middle|\; \begin{array}{l} (s_1, \ldots, s_m) \text{ ist } \vdash_{\mathcal{P}}\text{-Herleitung eines Sequenten} \\ \underbrace{\Gamma \blacktriangleright \alpha}_{=s_m} \text{ mit } \Gamma \subseteq T \end{array} \right\}$$

semi-entscheidbar. Das zeigt das folgende Programm.
Es benutzt P_a zur Berechnung von χ'_T.

Eingabe $(s_1, \ldots, s_m), \alpha$ ($(s_1, \ldots, s_m)$ ist eine Sequentenfolge für beliebiges m, α ist eine Formel)
für $i = 1, \ldots, m$ wiederhole:
überprüfe, ob der Sequent s_i mittels einer Regel $(\to I)$, $(\to E)$, (RAA), (Hyp),
$(= E)$, $(\forall I)$, $(\forall E)$ aus vorhergehenden s_j entsteht
oder ein Axiom $\varphi \blacktriangleright \varphi$ oder $\blacktriangleright \tau = \tau$ ist
falls alle Überprüfungen positiv waren
und s_m ein Sequent der Form $\Gamma \blacktriangleright \alpha$ ist
dann: für jedes $\gamma \in \Gamma$ wiederhole:
starte P_a mit Eingabe γ
Ausgabe 1
sonst: Endlosschleife

Die Anfangsargumentation zur Korrektheit des Programmes ist wie im Beweis von (10.3).
Die Überprüfung von „$\Gamma \subseteq T$“ ist hier etwas anders.
Es gilt: wenn $\Gamma \subseteq T$, dann ist die Schleife „für jedes $\gamma \in \Gamma$ wiederhole: ...“ nach endlich vielen Schritten beendet.
Wenn $\Gamma \not\subseteq T$, dann wird die Schleife „für jedes $\gamma \in \Gamma$ wiederhole: ...“ nie beendet, da P_a mit Eingabe eines $\gamma \notin T$ nicht hält.
Also berechnet der Algorithmus χ'_{Bew_T}, d.h. Bew_T ist semi-entscheidbar.

Nun müssen wir noch zeigen, dass die partielle charakteristische Funktion von T-BEWEISBAR berechenbar ist.

Die totale berechenbare Paarkodierungsfunktion $\gamma(x,y) = \frac{(x+y)\cdot(x+y+1)}{2} + y$
hat die totalen berechenbaren Inversen γ_1 und γ_2 mit $\gamma_1(\gamma(a,b)) = a$ und $\gamma_2(\gamma(a,b)) = b$.

Die URM P_c (vorige Seite) berechnet χ'_{Bew_T}.
Folgendes Programm berechnet $\chi'_{T\text{-BEWEISBAR}}$:

Eingabe α (α ist eine Formel)
$i := 0$
solange $\underbrace{P_c \text{ mit Eingabe } \gamma_1(i), \alpha \text{ nicht nach } \leqslant \gamma_2(i) \text{ Schritten hält}}_{(c,\gamma_1(i),\alpha,\gamma_2(i)) \notin H_B^{(2)}}$:
  $i := i + 1$
Ausgabe 1

Das Programm hält mit Eingabe α (und gibt 1 aus) gdw.
es gibt n und s, so dass P_c mit Eingabe n, α nach s Schritten hält gdw.
$\exists i\ (i,\alpha) \in Bew_T$ (d.h. $T \vdash_{\overline{\mathcal{P}}} \alpha$) gdw. $\alpha \in T$-BEWEISBAR.

Also ist T-BEWEISBAR semi-entscheidbar. □

Folgerung 10.6

$\mathcal{Q}$-BEWEISBAR *und* PA-BEWEISBAR *sind semi-entscheidbar.*

10.2 Beweissysteme und Semi-Entscheidbarkeit

Wir haben jetzt die Peano-Arithmetik durch einen Algorithmus beschrieben, der für die eingegebene Formel einen Beweis sucht.

Etwas abstrakter können wir ein Beweissystem auffassen als
eine berechenbare Funktion f mit

$$f(\alpha) = \begin{cases} b, & b \text{ ist Beweis für Formel } \alpha \\ \text{undefiniert}, & \text{falls es keinen Beweis für } \alpha \text{ gibt} \end{cases}$$

Den Begriff des Beweissystems kann man auch unabhängig von der Arithmetik für beliebige Mengen definieren.

Sei $A \subseteq \mathbb{N}$. Ein Beweissystem für A ist
eine (partielle) berechenbare Funktion $\psi : \mathbb{N} \to \mathbb{N}$ mit Definitionsbereich A.

Eine andere Idee für ein Beweissystem ist:

ein Algorithmus, der aus einem „Beweisversuch“ die bewiesene Formel bestimmt.

Etwas abstrakter:

eine totale berechenbare Funktion f mit

$$f(b) = \begin{cases} \alpha, & b \text{ ist Beweis für Formel } \alpha \\ \top, & \text{falls } b \text{ kein korrekter Beweis ist} \end{cases}$$

Das kann man auch wieder unabhängig von der Arithmetik für beliebige Mengen definieren.

Sei $A \subseteq \mathbb{N}$. Ein Beweissystem für A ist
eine totale berechenbare Funktion $\psi : \mathbb{N} \to \mathbb{N}$ mit Wertebereich A.

Wir werden jetzt zeigen, dass beide Begriffe von Beweissystemen äquivalent sind und dass die Mengen mit Beweissystemen genau die semi-entscheidbaren Mengen sind.

Der folgende Satz zeigt, dass Beweissysteme genau die semi-entscheidbaren Mengen sind.
„Im Prinzip“ ist der Begriff der Semi-Entscheidbarkeit also nur eine Verallgemeinerung des Begriffes der Beweissysteme (und der Beweisbarkeit) für Mengen, die nicht unbedingt etwas mit Logik zu tun haben.

Notation:
- $Defi_n$ ist der Definitionsbereich von φ_n.
- $Werte_n$ ist der Wertebereich von φ_n.

Satz 10.7 (Charakterisierungen semi-entscheidbarer Mengen)

Die folgenden Aussagen sind äquivalent für alle Mengen $A \neq \emptyset$.

1. *A ist semi-entscheidbar.*
2. *A ist Definitionsbereich einer berechenbaren Funktion.*
3. *A ist Wertebereich einer totalen berechenbaren Funktion.*
4. *A ist Wertebereich einer berechenbaren Funktion.*

Bem.: $\emptyset$ ist entscheidbar und folglich auch semi-entscheidbar.

Der folgende Satz zeigt, dass Beweissysteme genau die semi-entscheidbaren Mengen sind.
„Im Prinzip“ ist der Begriff der Semi-Entscheidbarkeit also nur eine Verallgemeinerung des Begriffes der Beweissysteme (und der Beweisbarkeit) für Mengen, die nicht unbedingt etwas mit Logik zu tun haben.

Notation:
- $Defi_n$ ist der Definitionsbereich von φ_n.
- $Werte_n$ ist der Wertebereich von φ_n.

Satz 10.7 (Charakterisierungen semi-entscheidbarer Mengen)

Die folgenden Aussagen sind äquivalent für alle Mengen $A \neq \emptyset$.

1. *A ist semi-entscheidbar.*
2. *$A = Defi_n$ für ein $n \in \mathbb{N}$.*
3. *$A = Werte_n$ für ein $n \in \mathbb{N}$, wobei φ_n total ist (d.h. A hat ein Beweissystem).*
4. *$A = Werte_n$ für ein $n \in \mathbb{N}$.*

Bem.: $\emptyset$ ist entscheidbar und folglich auch semi-entscheidbar.

Beweis von Satz 10.7 (Teil 1)

z.z: Wenn A semi-entscheidbar ist,
dann ist A Definitionsbereich einer berechenbaren Funktion.

Wenn A semi-entscheidbar ist,
dann ist χ'_A berechenbar
und A ist Definitionsbereich von χ'_A.

Beweis von Satz 10.7 (Teil 2)

z.z: Wenn A Definitionsbereich einer berechenbaren Funktion ist,
dann ist A Wertebereich einer totalen berechenbaren Funktion.

Sei $A = \mathit{Defi}_q$ (d.h. A ist Definitionsbereich von φ_q).

Definiere die Funktion $\psi : \mathbb{N} \to \mathbb{N}$ für ein $c_0 \in A$ als

$$\psi(n) = \begin{cases} \gamma_1(n), & \text{falls } (q, \gamma_1(n), \gamma_2(n)) \in H_B \\ c_0, & \text{sonst .} \end{cases}$$

Da ψ berechenbar ist, ist $\psi = \varphi_r$. Außerdem ist ψ total.
Offensichtlich gilt für den Wertebereich Werte_r von ψ: $\mathit{Werte}_r \subseteq A$. $(*)$

Für jedes $x \in A$ gibt es ein $t \in \mathbb{N}$ mit $(q, x, t) \in H_B$, d.h.
für jedes $x \in A$ gibt es ein $t \in \mathbb{N}$ mit $\psi(\gamma(x, t)) = x$.
Folglich ist $A \subseteq \mathit{Werte}_r$. $(**)$

Aus $(*)$ und $(**)$ folgt $A = \mathit{Werte}_r$.

Also ist A Wertebereich einer totalen berechenbaren Funktion. ✓

Beweis von Satz 10.7 (Teil 3)

z.z.: wenn A Wertebereich einer totalen berechenbaren Funktion ist,
dann ist A Wertebereich einer berechenbaren Funktion. ✓

Beweis von Satz 10.7 (Teil 4)

z.z.: wenn A Wertebereich einer berechenbaren Funktion ist, dann ist A semi-entscheidbar.

Sei A Wertebereich der berechenbaren Funktion φ_q.

Definiere den folgenden Algorithmus R (für χ'_A).

Eingabe x
$n := 0$
solange $(q, \gamma_1(n), \gamma_2(n)) \notin H_B$ oder $P_q(\gamma_1(n)) \neq x$ wiederhole:
    $n := n + 1$
Ausgabe 1

Wenn $m \in A$, dann gibt es ein (w, t) mit „$P_q(w) = m$ in t Schritten“,
und folglich hält $R(m)$ mit Ausgabe 1, wenn $n = \gamma(w, t)$ erreicht wurde.

Wenn $m \notin A$, dann gilt für alle (w, t):
„$P_q(w)$ hält nicht nach t Schritten oder $P_q(w) \neq m$“,
und folglich hält $R(m)$ nicht.

Also berechnet R genau die Funktion χ'_A. Das heißt: A ist semi-entscheidbar. □

Bemerkungen

Wenn $A = \mathit{Werte}_q$ für eine totale berechenbare Funktion φ_q,
dann ist $A = \{\varphi_q(0), \varphi_q(1), \varphi_q(2) \ldots\}$.

Daher stammt der Begriff *rekursiv aufzählbar*,
der auch für semi-entscheidbare Mengen benutzt wird.

Entscheidbare Mengen werden häufig *rekursiv* genannt.

Daher stammen auch die Bezeichnungen REC für die
Klasse aller entscheidbaren Mengen (REC $\hat{=}$ recursive)

und RE für die
Klasse aller semi-entscheidbaren Mengen
(RE $\hat{=}$ recursively enumerable).

Was haben wir in dieser Vorlesung gelernt?

- Formeln, Sequenten und endliche Folgen von Sequenten kann man durch Zahlen kodieren, mit denen man problemlos arbeiten kann.
- Mengen von Formeln, die man aus den Robinson-Axiomen, den Peano-Axiomen oder sonstigen semi-entscheidbaren Axiommengen mit Natürlichem Schließen herleiten kann, sind semi-entscheidbar.
- Der Begriff *Menge aller mit einem Beweissystem herleitbarer Formeln* entspricht dem Begriff *semi-entscheidbare Menge*.

Wie geht es weiter?

Wir werden sehen, dass sogar der Begriff *Menge aller mit der Robinson-Arithmetik herleitbarer* Σ_1*-Formeln* dem Begriff *semi-entscheidbare Menge* entspricht.

Das ist etwas aufwendiger und kostet die ganze nächste Vorlesung.

Das liefert uns den „Trick“,
mit dem wir für jedes Beweissystem eine Formel finden können,
die man mit dem Beweissystem nicht herleiten kann, die aber vom Standardmodell der natürlichen Zahlen erfüllt wird.

Diese Resultate nennt man

die Unvollständigkeitssätze von Gödel.

Vorlesung 11: URM-Programme sind wie Σ_1-Formeln

2. Unvollständigkeitssätze

VL08: Totale und partielle URM-berechenbare Funktionen

VL09: Entscheidbare und semi-entscheidbare Mengen

VL10: Mengen beweisbarer Formeln sind semi-entscheidbar

VL11: URM-Programme sind wie Σ_1-Formeln

- Die Konfigurationsübergangsrelation $\overset{P}{\to}$ ist Δ_0-arithmetisch
- Die Konfigurationserreichbarkeitsrelation $\overset{P}{\to}^*$ ist Σ_1-arithmetisch

VL12: Die Unvollständigkeitssätze

Einleitung

Wir haben gesehen, dass Mengen herleitbarer Formeln für beliebige semi-entscheidbare Axiommengen bzw. für beliebige Beweissysteme semi-entscheidbar sind

– insbesondere $\mathcal{Q}$-BEWEISBAR und PA-BEWEISBAR.

Wir werden nun zeigen, dass $\mathfrak{N}$-SAT nicht semi-entscheidbar ist.

Daraus folgt, dass es kein Beweissystem gibt,

in dem alle von $\mathfrak{N}$ erfüllten arithmetischen Formeln ($\mathfrak{N}$-SAT) bewiesen werden.

Der Beweis, dass $\mathfrak{N}$-SAT nicht semi-entscheidbar ist, ist aufwendig.

Letztlich werden wir zeigen:

wenn $\mathfrak{N}$-SAT semi-entscheidbar ist, dann ist auch $\overline{K}$ semi-entscheidbar.

Da $\overline{K}$ nicht semi-entscheidbar ist, folgt, dass $\mathfrak{N}$-SAT ebenfalls nicht semi-entscheidbar ist.

Wie kommen wir dort hin?

Wenn $\mathfrak{N}$-SAT semi-entscheidbar ist, dann ist auch $\overline{K}$ semi-entscheidbar.

$\Uparrow$

Es gibt eine Σ_1-Formel $\psi_K(x)$, so dass für alle $n \in \mathbb{N}$ gilt:

$$n \in \overline{K} \quad \text{gdw.} \quad \mathfrak{N} \models_{\mathcal{P}} \neg\psi_K(\overline{n})$$
$$\text{d.h.} \quad n \in K \quad \text{gdw.} \quad \mathfrak{N} \models_{\mathcal{P}} \psi_K(\overline{n})$$

$\Uparrow$

Sei P ein URM-Programm für χ'_K mit Länge ℓ.

Dann gibt es eine Σ_1-Formel σ, so dass für alle $n \in \mathbb{N}$ gilt:

$$n \in K \quad \text{gdw.} \quad \mathfrak{N} \models_{\mathcal{P}} \exists t_0, \ldots, t_{k+1} : \; t_{k+1} > \overline{\ell} \wedge \sigma(0, \overline{n}, 0, \ldots, 0, t_0, \ldots, t_{k+1})$$

$\Uparrow$

Für jedes URM-Progr. P (mit k Registern) gibt es eine Δ_0-Formel δ und eine Σ_1-Formel σ, so dass für alle $s_i, t_i \in \mathbb{N}$ gilt:

1.) $(s_0, \ldots, s_{k+1}) \xrightarrow{P} (t_0, \ldots, t_{k+1})$ gdw. $\mathfrak{N} \models_{\mathcal{P}} \delta(\overline{s_0}, \ldots, \overline{s_{k+1}}, \overline{t_0}, \ldots, \overline{t_{k+1}})$

2.) $(s_0, \ldots, s_{k+1}) \xrightarrow{P}{}^{*} (t_0, \ldots, t_{k+1})$ gdw. $\mathfrak{N} \models_{\mathcal{P}} \sigma(\overline{s_0}, \ldots, \overline{s_{k+1}}, \overline{t_0}, \ldots, \overline{t_{k+1}})$

11.1 Die Konfigurationsübergangsrelation $\xrightarrow{P}$ ist Δ_0-arithmetisch

Wir wiederholen die formale Definition für die Berechnung eines URM-Programms (Definitionen (8.1)–(8.3)).

Sei $P = (I_0, I_1, \ldots, I_{s-1})$ ein URM-Programm mit Registern $R_0, \ldots, R_k$.

Eine Konfiguration von P ist ein $(k+2)$-Tupel $(c_0, \ldots, c_k, pz)$,
bei dem $c_0, \ldots, c_k \in \mathbb{N}$ die Inhalte der Register $R_0, \ldots, R_k$ sind
und $pz \in \mathbb{N}$ die Nummer der nächsten auszuführenden Programmzeile ist.

Die Konfigurationsübergangsrelation $\xrightarrow{P}$ ist die Menge aller Konfigurationspaare (a, b),
bei denen Konfiguration b durch einen Schritt des Programms P aus Konfiguration a entsteht.

$\xrightarrow{P}^*$ ist die reflexive und transitive Hülle von $\xrightarrow{P}$.

Lemma 11.1 ($\overset{P}{\to}$ ist Δ_0-arithmetisch)

Sei P ein URM-Programm mit r Registern.
Dann gibt es eine Δ_0-Formel δ_P mit $2 \cdot (r+2)$ freien Variablen,
so dass für alle $s_0, \ldots, s_{r+1}, t_0, \ldots, t_{r+1} \in \mathbb{N}$ gilt:
$(s_0, \ldots, s_{r+1}) \overset{P}{\to} (t_0, \ldots, t_{r+1})$ *gdw.* $\mathfrak{N} \models_{\mathcal{P}} \delta_P(\overline{s_0}, \ldots, \overline{s_{r+1}}, \overline{t_0}, \ldots, \overline{t_{r+1}})$.

Beweis:

Im Folgenden sei $P = (I_0, \ldots, I_{\ell-1})$ ein URM-Programm der Länge ℓ mit r Registern.

Wir konstruieren die arithmetische Δ_0-Formel $\delta_P(x_0, \ldots, x_{r+1}, y_0, \ldots, y_{r+1})$
aus folgenden Bestandteilen für die drei Arten von Programmzeilen von P.

Teilformel zur Simulation von $\mathtt{R}_\mathtt{m} = 0$

Die Ausführung von $\mathtt{R}_\mathtt{m} = 0$ wird durch die Formel $\psi_{\mathtt{R}_\mathtt{m}=0}$ beschrieben.

$\psi_{\mathtt{R}_\mathtt{m}=0}$ hat freie Variablen $x_0, \ldots, x_{r+1}, y_0, \ldots, y_{r+1}$.

$$\psi_{\mathtt{R}_\mathtt{m}=0} \quad := \quad \bigwedge_{0 \leqslant i \leqslant r, i \neq m} x_i = y_i$$

alle Register R_i mit $i \neq m$ bleiben unverändert

$$\wedge \quad y_m = 0$$

R_m wird auf 0 gesetzt

$$\wedge \quad y_{r+1} = s(x_{r+1})$$

der Programmzähler wird um 1 erhöht

Zu zeigen: Für $I_{a_{r+1}} = \mathtt{R}_\mathtt{m} = 0$ gilt
$(a_0, \ldots, a_{r+1}) \xrightarrow{P} (b_0, \ldots, b_{r+1})$ genau dann, wenn $\mathfrak{N} \models_{\mathcal{P}} \psi_{\mathtt{R}_\mathtt{m}=0}(\overline{a_0}, \ldots, \overline{a_{r+1}}, \overline{b_0}, \ldots, \overline{b_{r+1}})$.

Beobachtung 2

Sei $I_{a_{r+1}} = \mathtt{R}_{\mathtt{m}} = 0$.

Dann gilt $(a_0, \ldots, a_{r+1}) \xrightarrow{P} (b_0, \ldots, b_{r+1})$ *genau dann, wenn* $\mathfrak{N} \models_{\mathcal{P}} \psi_{\mathtt{R}_{\mathtt{m}}=0}(\overline{a_0}, \ldots, \overline{a_{r+1}}, \overline{b_0}, \ldots, \overline{b_{r+1}})$.

Beweis:

Sei $I_{a_{r+1}} = \mathtt{R}_{\mathtt{m}} = 0$.

Nach der Definition von $\xrightarrow{P}$ gilt $(a_0, \ldots, a_{r+1}) \xrightarrow{P} (b_0, \ldots, b_{r+1})$ gdw.

1. $a_i = b_i$ für $i = 0, \ldots, m-1, m+1, \ldots, r$,
2. $b_m = 0$ und
3. $b_{r+1} = a_{r+1} + 1$.

Wegen der Eigenschaften von $\mathfrak{N}$ und $\models_{\mathcal{P}}$ ist das äquivalent zu

$$\mathfrak{N} \models_{\mathcal{P}} \underbrace{\bigwedge_{0 \leqslant i \leqslant r, i \neq m} \overline{a_i} = \overline{b_i} \ \wedge \ \overline{b_m} = 0 \ \wedge \ \overline{b_{r+1}} = s(\overline{a_{r+1}})}_{\psi_{\mathtt{R}_{\mathtt{m}}=0}(\overline{a_0}, \ldots, \overline{a_{r+1}}, \overline{b_0}, \ldots, \overline{b_{r+1}})}. \qquad \checkmark$$

Teilformel zur Simulation von $\mathtt{R_m \mathrel{+}= 1}$

Die Ausführung von $\mathtt{R_m \mathrel{+}= 1}$ wird durch $\psi_{\mathtt{R_m \mathrel{+}= 1}}$ beschrieben.

$\psi_{\mathtt{R_m \mathrel{+}= 1}}$ hat freie Variablen $x_0, \ldots, x_{r+1}, y_0, \ldots, y_{r+1}$.

$$\psi_{\mathtt{R_m \mathrel{+}= 1}} \quad := \quad \bigwedge_{0 \leqslant i \leqslant r, i \neq m} x_i = y_i \qquad \text{alle Register } R_i \text{ mit } i \neq m \text{ bleiben unverändert}$$

$$\wedge \quad y_m = s(x_m) \qquad R_m \text{ wird um 1 erhöht}$$

$$\wedge \quad y_{r+1} = s(x_{r+1}) \qquad \text{der Programmzähler wird um 1 erhöht}$$

Beobachtung 3

Sei $I_{a_{r+1}} = \mathtt{R_m \mathrel{+}= 1}$.
Dann gilt $(a_0, \ldots, a_{r+1}) \xrightarrow{P} (b_0, \ldots, b_{r+1})$ *genau dann, wenn* $\mathfrak{N} \models_{\mathcal{P}} \psi_{\mathtt{R_m \mathrel{+}= 1}}(\overline{a_0}, \ldots, \overline{a_{r+1}}, \overline{b_0}, \ldots, \overline{b_{r+1}})$.

Teilformel zur Simulation von $\texttt{if } \mathtt{R_h} == \mathtt{R_m} : \texttt{goto g}$

Die Ausführung von $\texttt{if } \mathtt{R_h} == \mathtt{R_m} : \texttt{goto g}$ wird durch $\psi_{\texttt{if } \mathtt{R_h} == \mathtt{R_m}:\texttt{goto g}}$ beschrieben.
$\psi_{\texttt{if } \mathtt{R_h} == \mathtt{R_m}:\texttt{goto g}}$ hat freie Variablen $x_0, \ldots, x_{r+1}, y_0, \ldots, y_{r+1}$.

$$\psi_{\texttt{if } \mathtt{R_h} == \mathtt{R_m}:\texttt{goto g}} \quad := \quad \bigwedge_{0 \leqslant i \leqslant r} x_i = y_i$$

alle Register R_i bleiben unverändert

$$\wedge \quad x_h = x_m \to y_{r+1} = \overline{g}$$

falls $R_h = R_m$, dann wird der Programmzähler auf g gesetzt

$$\wedge \quad x_h \neq x_m \to y_{r+1} = s(x_{r+1})$$

sonst wird der Programmzähler um 1 erhöht

Beobachtung 4

Sei $I_{a_{r+1}} = \texttt{if R}_\texttt{h} \texttt{==R}_\texttt{m} : \texttt{goto g}$.
Dann gilt $(a_0, \ldots, a_{r+1}) \xrightarrow{P} (b_0, \ldots, b_{r+1})$ *genau dann, wenn*
$\mathfrak{N} \models_{\mathcal{P}} \psi_{\texttt{if R}_\texttt{h} \texttt{==R}_\texttt{m}:\texttt{goto g}}(\overline{a_0}, \ldots, \overline{a_{r+1}}, \overline{b_0}, \ldots, \overline{b_{r+1}})$.

Beweis:
Sei $I_{a_{r+1}} = \texttt{if R}_\texttt{h} \texttt{==R}_\texttt{m} : \texttt{goto g}$.
Nach der Definition von $\xrightarrow{P}$ gilt $(a_0, \ldots, a_{r+1}) \xrightarrow{P} (b_0, \ldots, b_{r+1})$ gdw.

1. $a_i = b_i$ für $i = 0, \ldots, r$,
2. aus $a_h = a_m$ folgt $b_{r+1} = g$, und
3. aus $a_h \neq a_m$ folgt $b_{r+1} = a_{r+1} + 1$.

Wie zuvor gilt $a_i = b_i$ für $i = 0, \ldots, r$ gdw. $\mathfrak{N} \models_{\mathcal{P}} \bigwedge_{0 \leqslant i \leqslant r} \overline{a_i} = \overline{b_i}$.

Für den Rest der Formel machen wir eine Fallunterscheidung.

Fall 1: $a_h = a_m$. Dann gilt $\mathfrak{N} \models_{\mathcal{P}} \overline{a_h} = \overline{a_m}$,
und es folgt $\mathfrak{N} \models_{\mathcal{P}} \overline{a_h} = \overline{a_m} \to \overline{b_{r+1}} = \overline{g}$ gdw. $\mathfrak{N} \models_{\mathcal{P}} \overline{b_{r+1}} = \overline{g}$ gdw. $b_{r+1} = g$.

Aus $\mathfrak{N} \models_{\mathcal{P}} \overline{a_h} = \overline{a_m}$ folgt außerdem $\mathfrak{N} \models_{\mathcal{P}} \overline{a_h} \neq \overline{a_m} \to \overline{b_{r+1}} = s(\overline{a_{r+1}})$.

Damit haben wir für den Fall $a_h = a_m$:

aus $a_h = a_m$ folgt $b_{r+1} = g$ gdw.
$b_{r+1} = g$ gdw. $\mathfrak{N} \models_{\mathcal{P}} \overline{a_h} = \overline{a_m} \to \overline{b_{r+1}} = \overline{g} \wedge \overline{a_h} \neq \overline{a_m} \to \overline{b_{r+1}} = s(\overline{a_{r+1}})$.

Fall 2: $a_h \neq a_m$. Dann gilt $\mathfrak{N} \not\models_{\mathcal{P}} \overline{a_h} = \overline{a_m}$ und folglich $\mathfrak{N} \models_{\mathcal{P}} \overline{a_h} = \overline{a_m} \to \overline{b_{r+1}} = \overline{g}$.

Nun gilt $\mathfrak{N} \models_{\mathcal{P}} \overline{a_h} \neq \overline{a_m} \to \overline{b_{r+1}} = s(\overline{a_{r+1}})$ gdw. $\mathfrak{N} \models_{\mathcal{P}} \overline{b_{r+1}} = s(\overline{a_{r+1}})$ gdw. $b_{r+1} = a_{r+1} + 1$.

Damit haben wir für den Fall $a_h \neq a_m$:

aus $a_h \neq a_m$ folgt $b_{r+1} = a_{r+1} + 1$ gdw.
$b_{r+1} = a_{r+1} + 1$ gdw. $\mathfrak{N} \models_{\mathcal{P}} \overline{a_h} = \overline{a_m} \to \overline{b_{r+1}} = \overline{g} \wedge \overline{a_h} \neq \overline{a_m} \to \overline{b_{r+1}} = s(\overline{a_{r+1}})$.

Zusammengefasst liefern die beiden Fälle

$(a_0, \ldots, a_{r+1}) \xrightarrow{P} (b_0, \ldots, b_{r+1})$ gdw. $\mathfrak{N} \models_{\mathcal{P}} \psi_{\texttt{if } R_h == R_m \texttt{:goto } g}(\overline{a_0}, \ldots, \overline{a_{r+1}}, \overline{b_0}, \ldots, \overline{b_{r+1}})$. ✓

Fall 1: $a_h = a_m$. Dann gilt $\mathfrak{N} \models_{\mathcal{P}} \overline{a_h} = \overline{a_m}$,
und es folgt $\mathfrak{N} \models_{\mathcal{P}} \overline{a_h} = \overline{a_m} \to \overline{b_{r+1}} = \overline{g}$ gdw. $\mathfrak{N} \models_{\mathcal{P}} \overline{b_{r+1}} = \overline{g}$ gdw. $b_{r+1} = g$.

Aus $\mathfrak{N} \models_{\mathcal{P}} \overline{a_h} = \overline{a_m}$ folgt außerdem $\mathfrak{N} \models_{\mathcal{P}} \overline{a_h} \neq \overline{a_m} \to \overline{b_{r+1}} = s(\overline{a_{r+1}})$.

Damit haben wir für den Fall $a_h = a_m$:
aus $a_h = a_m$ folgt $b_{r+1} = g$ gdw.
$b_{r+1} = g$ gdw. $\mathfrak{N} \models_{\mathcal{P}} \overline{a_h} = \overline{a_m} \to \overline{b_{r+1}} = \overline{g} \wedge \overline{a_h} \neq \overline{a_m} \to \overline{b_{r+1}} = s(\overline{a_{r+1}})$.

Fall 2: $a_h \neq a_m$. Dann gilt $\mathfrak{N} \not\models_{\mathcal{P}} \overline{a_h} = \overline{a_m}$ und folglich $\mathfrak{N} \models_{\mathcal{P}} \overline{a_h} = \overline{a_m} \to \overline{b_{r+1}} = \overline{g}$.

Nun gilt $\mathfrak{N} \models_{\mathcal{P}} \overline{a_h} \neq \overline{a_m} \to \overline{b_{r+1}} = s(\overline{a_{r+1}})$ gdw. $\mathfrak{N} \models_{\mathcal{P}} \overline{b_{r+1}} = s(\overline{a_{r+1}})$ gdw. $b_{r+1} = a_{r+1} + 1$.

Damit haben wir für den Fall $a_h \neq a_m$:
aus $a_h \neq a_m$ folgt $b_{r+1} = a_{r+1} + 1$ gdw.
$b_{r+1} = a_{r+1} + 1$ gdw. $\mathfrak{N} \models_{\mathcal{P}} \overline{a_h} = \overline{a_m} \to \overline{b_{r+1}} = \overline{g} \wedge \overline{a_h} \neq \overline{a_m} \to \overline{b_{r+1}} = s(\overline{a_{r+1}})$.

Zusammengefasst liefern die beiden Fälle

$$(a_0, \ldots, a_{r+1}) \xrightarrow{P} (b_0, \ldots, b_{r+1}) \text{ gdw. } \mathfrak{N} \models_{\mathcal{P}} \psi_{\texttt{if } \mathtt{R_h} == \mathtt{R_m} \texttt{:goto g}}(\overline{a_0}, \ldots, \overline{a_{r+1}}, \overline{b_0}, \ldots, \overline{b_{r+1}}). \quad \checkmark$$

Die Simulation eines beliebigen Programmschrittes

Nun können wir die Formel δ_P mit den freien Variablen $x_0, \ldots, x_{r+1}, y_0, \ldots, y_{r+1}$ zur Beschreibung von $\xrightarrow{P}$ definieren.

$$\delta_P \;:=\; x_{r+1} < \overline{\ell} \qquad \text{die erste Konfiguration enthält einen zulässigen Programmzähler}$$

$$\wedge \bigwedge_{0 \leqslant z < \ell} x_{r+1} = \overline{z} \to \psi_{I_z} \qquad \text{das Konfigurationspaar entspricht der Ausführung der Programmzeile } I_{x_{r+1}}$$

Beobachtung 5

Für alle $a_0, \ldots, a_{r+1}, b_0, \ldots, b_{r+1} \in \mathbb{N}$ *gilt:*
$(a_0, \ldots, a_{r+1}) \xrightarrow{P} (b_0, \ldots, b_{r+1})$ *gdw.* $\mathfrak{N} \models_{\mathcal{P}} \delta_P(\overline{a_0}, \ldots, \overline{a_{r+1}}, \overline{b_0}, \ldots, \overline{b_{r+1}})$.

Der Beweis der Beobachtung folgt aus der Konstruktion von δ_P und Beobachtungen (2)–(4).

δ_P ist eine Δ_0-Formel. □

11.2 Die Konfigurationserreichbarkeitsrelation $\overset{P}{\rightarrow}^*$ ist Σ_1-arithmetisch

Wir wollen durch URM-Rechnungen „verbundene“ Konfigurationen arithmetisch beschreiben.

Eine Grundidee dazu sieht man
bei folgender informell-arithmetischer Beschreibung der Funktion $n \mapsto 2^n$:

$2^n = p$ gdw. $\exists u$ u kodiert eine endliche Folge $u[0], u[1], \ldots, u[n]$ mit
1.) $u[0] = 1$,
2.) für alle $i < n : u[i+1] = 2 \cdot u[i]$ und
3.) $u[n] = p$.

Problem:
zur Dekodierung von endlichen Folgen/Tupeln haben wir bisher stets Potenzierung benutzt.
Wir wissen aber nicht, wie Potenzierung arithmetisch beschreibbar ist ...

11.2 Die Konfigurationserreichbarkeitsrelation $\overset{P}{\rightarrow}^*$ ist Σ_1-arithmetisch

Wir wollen durch URM-Rechnungen „verbundene“ Konfigurationen arithmetisch beschreiben.

Eine Grundidee dazu sieht man

bei folgender informell-arithmetischer Beschreibung der Funktion $n \mapsto 2^n$:

$s \overset{P}{\rightarrow}^* t$ gdw. $\exists u\ \exists m$ u kodiert eine endliche Folge $u[0], u[1], \ldots, u[m]$ mit

1.) $u[0] = s$,

2.) für alle $i < m : u[i] \overset{P}{\rightarrow} u[i+1]$ und

3.) $u[m] = t$.

Problem:
zur Dekodierung von endlichen Folgen/Tupeln haben wir bisher stets Potenzierung benutzt.
Wir wissen aber nicht, wie Potenzierung arithmetisch beschreibbar ist.

Gödels Kodierung mit sehr einfacher Dekodierung

Zur Kodierung von endlichen Folgen kann man auf Techniken zurückgreifen,
die auf dem Chinesischen Restsatz basieren.

Satz 11.2 (Gödels β-Lemma)

Für jedes $k \in \mathbb{N}^+$ und für alle $n_1, \ldots, n_k \in \mathbb{N}$ gibt es $a, b \in \mathbb{N}$,
so dass für $i = 1, 2, \ldots, k$ gilt:

$$n_i = a \bmod (b \cdot i + 1).$$

Beispiel:

```
$ python3 betafunktion.py 5 1 4 0
Berechne eine Kodierung von [5, 1, 4, 0] mit Gödels beta-Lemma:
a ist 3511362530.
b ist 120.
 5 ist a mod (b*1+1) = 3511362530 mod 121 = 5.
 1 ist a mod (b*2+1) = 3511362530 mod 241 = 1.
 4 ist a mod (b*3+1) = 3511362530 mod 361 = 4.
 0 ist a mod (b*4+1) = 3511362530 mod 481 = 0.
```

Gödels β-Lemma: Für jedes $k \in \mathbb{N}^+$ und für alle $n_1, \ldots, n_k \in \mathbb{N}$ gibt es $a, b \in \mathbb{N}$,

so dass für $i = 1, 2, \ldots, k$ gilt: $n_i = a \bmod (b \cdot i + 1)$.

Beweis:

Sei $b = (\max\{n_1, \ldots, n_k, k\})!$.

Definiere $b_i = b \cdot i + 1$ für $i = 1, \ldots, k$.

Behauptung: Für $i \neq j$ sind b_i und b_j teilerfremd $(i, j = 1, \ldots, k)$.

Annahme: b_i und b_j sind nicht teilerfremd.
Dann gibt es einen Primteiler p von b_i und b_j.
Jeder Primteiler von b_i und b_j teilt auch $b_i - b_j$ (oBdA $i > j$) – also gilt $p|(b_i - b_j)$.
Es gilt $b_i - b_j = b \cdot i + 1 - b \cdot j - 1 = b \cdot (i - j)$.
Aus $p \not| (i - j)$ folgt $p|b$.
Aus $p|(i-j)$ und $(i-j)|b$ (da $i - j < \max\{n_1, \ldots, n_k, k\}$) folgt ebenfalls $p|b$ – also gilt $p|b$.
Aus $p|b$ und $p|\underbrace{b \cdot i + 1}_{b_i}$ folgt $p = 1$.
Also ist p kein Primteiler von b_i und b_j.
Damit ist die Annahme widerlegt, und die Behauptung ist bewiesen.

Sei $m = b_1 \cdot b_2 \cdot \ldots \cdot b_k$. Wir benutzen Schreibweise $[\ell] = \{0, 1, 2, \ldots, \ell - 1\}$.

Betrachte die Funktion $f : [m] \to [b_1] \times [b_2] \times \ldots \times [b_k]$
mit $f(x) = (x \bmod b_1, x \bmod b_2, \ldots, x \bmod b_k)$.

Beh.: f ist bijektiv.

f ist injektiv: $f(u) = f(v) \Rightarrow \quad b_i | (u - v)$ für alle $i \in \{1, 2, \ldots, k\}$

$\Rightarrow \quad m | (u - v)$ (da alle b_i teilerfremd)

$\Rightarrow \quad u - v = 0$ (da $u, v < m$)

d.h. $\quad u = v$

f ist surjektiv: da f injektiv und total ist und Definitions- und Wertebereich von f gleichgroß sind.

$$\begin{aligned} |[m]| &= m \\ &= b_1 \cdot b_2 \cdot \ldots \cdot b_k \\ &= |[b_1]| \cdot |[b_2]| \cdot \ldots \cdot |[b_k]| \\ &= |[b_1] \times [b_2] \times \ldots \times [b_k]| \end{aligned}$$

Also ist f^{-1} eine total Funktion und $a = f^{-1}(n_1, n_2, \ldots, n_k)$.

□

Wir interessieren uns nur für das Dekodieren – und das ist einfach.

Fasst man a, b als Kodierung einer Folge aus k Elementen auf,
dann liefert die Funktion *sel* mit
$sel(a, b, i) := a \bmod (b \cdot i + 1)$
die Dekodierung des i-ten Elements von a, b (für $1 \leqslant i \leqslant k$).

Diese Dekodierungsfunktion ist Δ_0-arithmetisch.

Lemma 11.3 (die Dekodierung der Kodierung in (11.2) ist Δ_0-arithmetisch)

Sei $\varphi_{sel}(u, v, w, x) := x < v \cdot w + \overline{1} \ \wedge \ \exists y \leqslant u \ \ u = y \cdot (v \cdot w + \overline{1}) + x$.
Dann gilt für alle $a, b, i, x \in \mathbb{N}$

$$a \bmod (b \cdot i + 1) = x \ \ \textit{gdw.} \ \ \mathfrak{N} \models_{\overline{\mathcal{P}}} \underbrace{\overline{x} < \overline{b} \cdot \overline{i} + \overline{1} \ \wedge \ \exists y \leqslant \overline{a} \ \ \overline{a} = y \cdot (\overline{b} \cdot \overline{i} + \overline{1}) + \overline{x}}_{\varphi_{sel}(\overline{a}, \overline{b}, \overline{i}, \overline{x})} \ .$$

Bsp.: Σ_1-Beschreibung von $n \mapsto 2^n$

$\{(x, y) \mid y = 2^x\}$ wird beschrieben durch die Σ_1-Formel

$$\begin{aligned}\exists a\, \exists b: \;& \varphi_{sel}(a, b, \overline{1}, \overline{1}) \;\wedge \\ & \big(\forall i < x \quad \exists v \;\; \exists w \\ & \qquad (\varphi_{sel}(a, b, s(i), v) \wedge \varphi_{sel}(a, b, s(s(i)), w) \wedge \overline{2} \cdot v = w)\;\big) \;\wedge \\ & \varphi_{sel}(a, b, s(x), y)\end{aligned}$$

$y = 2^x$ gdw. es gibt Zahlen a und b,
so dass $1 = a \bmod (b \cdot (0 + 1) + 1)$,
$2 = a \bmod (b \cdot (1 + 1) + 1) = 2 \cdot a \bmod (b \cdot (0 + 1) + 1)$,
$4 = a \bmod (b \cdot (2 + 1) + 1) = 2 \cdot a \bmod (b \cdot (1 + 1) + 1)$,
$\vdots \qquad\qquad \vdots \qquad\qquad \vdots$
$y = a \bmod (b \cdot (x + 1) + 1) = 2 \cdot a \bmod (b \cdot ((x - 1) + 1) + 1)$.

Kodierung und Dekodierung von Tupeln fester Größe

Wir brauchen später auch noch Dekodierungsfunktionen für kodierte Tupel fester Länge, die man durch einfache arithmetische Formeln beschreiben kann.

Definition 11.4 (Kodierung von Tupeln fester Größe)

$\rho^{(2)} : \mathbb{N}^2 \to \mathbb{N}$ ist die Paarkodierungsfunktion $\rho^{(2)}(x, y) = \frac{(x+y)\cdot(x+y+1)}{2} + y$.

Für $k \geqslant 3$ ist $\rho^{(k)} : \mathbb{N}^k \to \mathbb{N}$ die k-Tupelkodierungsfunktion

$$\rho^{(k)}(x_1, \ldots, x_k) = \rho^{(2)}(x_1, \rho^{(k-1)}(x_2, \ldots, x_k)).$$

Für $k \geqslant 2$ und $1 \leqslant j \leqslant k$ ist $\rho_j^{(k)} : \mathbb{N} \to \mathbb{N}$ die Dekodierungsfunktion mit

$$\rho_j^{(k)}(\rho^{(k)}(n_1, \ldots, n_k)) = n_j.$$

Alle Funktionen $\rho^{(i)}$ sind bijektiv.

Lemma 11.5 (Kodierung von k-Tupeln ist Δ_0-arithmetisch)

Für jedes $k \geqslant 2$ gibt es eine Δ_0-Formel $\varphi^{(k)}$ mit $k+1$ freien Variablen, so dass für alle $n_1, \ldots, n_k, z \in \mathbb{N}$ gilt: $\rho^{(k)}(n_1, \ldots, n_k) = z$ gdw. $\mathfrak{N} \models_{\mathcal{P}} \varphi^{(k)}(\overline{n_1}, \ldots, \overline{n_k}, \overline{z})$.

Beweisidee:
Für $k = 2$ ist

$$\varphi^{(2)}(x, y, z) \;:=\; \exists u \leqslant z \; (\; \overline{2} \cdot u = (x + y) \cdot (x + s(y)) \wedge z = u + y \;)$$

die gesuchte Δ_0-Formel.

Für $k \geqslant 3$ leistet dann

$$\varphi^{(k)}(x_1, \ldots, x_k, z) \;:=\; \exists w \leqslant z \; \varphi^{(k-1)}(x_2, \ldots, x_k, w) \;\wedge\; \varphi^{(2)}(x_1, w, z)$$

das entsprechende. □

Lemma 11.6 (Dekodierung kodierter k-Tupel ist Δ_0-arithmetisch)

Für jedes $k \geqslant 2$ und jedes j mit $1 \leqslant j \leqslant k$ gibt es eine Δ_0-Formel $\gamma_j^{(k)}$ mit 2 freien Variablen, so dass für alle $n, m, z \in \mathbb{N}$ gilt: $\rho_j^{(k)}(n) = z$ gdw. $\mathfrak{N} \models_{\mathcal{P}} \gamma_j^{(k)}(\overline{n_1}, \overline{z})$.

Beweisidee:

- $\gamma_1^{(2)}(n, x) := \exists y \leqslant n \; \varphi^{(2)}(x, y, n)$ und $\gamma_2^{(2)}(n, y) := \exists x \leqslant n \; \varphi^{(2)}(x, y, n)$.
- Für $k \geqslant 3$ und $1 \leqslant j \leqslant k$:
  $\gamma_j^{(k)}(n, x_j) := \exists x_1 \leqslant n \cdots \exists x_{j-1} \leqslant n \; \exists x_{j+1} \leqslant n \cdots \exists x_k \leqslant n \;\; \varphi^{(k)}(x_1, \ldots, x_k, n)$.

(Die Formeln $\gamma_j^{(k)}$ werden im Beweis von (11.7) gebraucht.) □

Lemma 11.7 (die transitive Hülle von $\xrightarrow{P}$ ist Σ_1-arithmetisch)

Sei P ein URM-Programm mit r Registern.
Es gibt eine Σ_1-Formel σ_P mit $2 \cdot (r+2)$ freien Variablen,
so dass für alle $s_0, \ldots, s_{r+1}, t_0, \ldots, t_{r+1} \in \mathbb{N}$ gilt:
$(s_0, \ldots, s_{r+1}) \xrightarrow{P}{}^{*} (t_0, \ldots, t_{r+1})$ *gdw.* $\mathfrak{N} \models_{\mathcal{P}} \sigma_P(s_0, \ldots, s_{r+1}, t_0, \ldots, t_{r+1})$.

Beweis(idee):
Die Idee für die Formel hatten wir schonmal formuliert.

$s \xrightarrow{P}{}^{*} t$ gdw. $\exists a\, \exists m$ a kodiert eine endliche Folge $a[0], a[1], \ldots, a[m]$ mit

1.) $a[0] = s$,
2.) für alle $i < m : a[i] \xrightarrow{P} a[i+1]$ und
3.) $a[m] = t$.

Nun werden wir sie tatsächlich als Σ_1-Formel „implementieren“.

Im ersten Schritt bauen wir die Dekodierung gemäß β-Lemma (11.2) ein.
a und b haben die gleiche Bedeutung wie im β-Lemma, m ist die Länge der Konfigurationsfolge.
$a[i]$ (das i-te Element des in a kodierten Tupels) ist $a \bmod (b \cdot i + 1)$.
Wir müssen beachten, dass beim β-Lemma das erste in a kodierte Element $a \bmod (b \cdot 1 + 1)$ ist.

$$
\begin{aligned}
s \xrightarrow{P} t \quad \text{gdw.} \quad & \exists a \;\; \exists b \;\; \exists m : \\
& a \bmod (b+1) = s \quad \text{und} \\
& \Big(\forall i < m \;\; \exists u \;\; \exists v : \\
& \qquad a \bmod (b \cdot (i+1) + 1) = u \;\; \text{und} \;\; a \bmod (b \cdot (i+2) + 1) = v \;\; \text{und} \\
& \qquad u \xrightarrow{P} v \Big) \quad \text{und} \\
& a \bmod (b \cdot m + 1) = t
\end{aligned}
$$

$\forall i < m \; \varphi$ ist Abkürzung für $\forall i \leqslant m \; ((i = m) \vee \varphi)$.

Im zweiten Schritt berücksichtigen wir,
dass die Konfigurationen s, t, u, v eigentlich $(r+2)$-Tupel von Zahlen sind.
Wir fassen s, t, x, u, v, y als das Ergebnis von durch $\rho^{(r+2)}$ kodierte Tupel auf.
Die einzelnen Elemente der Tupel erhält man durch Dekodieren mit $\rho_i^{(r+2)}$.

$$
\begin{aligned}
&(s_1, \ldots, s_{r+2}) \xrightarrow{P} (t_1, \ldots, t_{r+2}) \ :=\\
&\qquad \exists a \ \ \exists b \ \ \exists m\\
&\qquad \exists x \ \Big(\ a \bmod (b+1) = x \text{ und } \rho_1^{(r+2)}(x) = s_1 \text{ und } \cdots \text{ und } \rho_{r+2}^{(r+2)}(x) = s_{r+2} \ \Big) \quad \text{und}\\
&\qquad \Big(\ \forall i < m \ \ \exists u \ \ \exists v :\\
&\qquad\qquad a \bmod (b \cdot (i+1) + 1) = u \ \text{ und } \ a \bmod (b \cdot (i+2) + 1) = v \ \text{ und }\\
&\qquad\qquad (\rho_1^{(r+2)}(u), \ldots, \rho_{r+2}^{(r+2)}(u)) \xrightarrow{P} (\rho_1^{(r+2)}(v), \ldots, \rho_{r+2}^{(r+2)}(v)) \ \Big) \text{ und}\\
&\qquad \exists y \ \Big(\ a \bmod (b \cdot m + 1) = y \text{ und } \rho_1^{(r+2)}(y) = t_1 \text{ und } \cdots \text{ und } \rho_{r+2}^{(r+2)}(y) = t_{r+2} \ \Big)
\end{aligned}
$$

Im dritten Schritt ersetzen wir die Dekodierungsfunktion $a \bmod (b \cdot i + 1) = s$ durch die Formel $\varphi_{sel}(a, b, i, s)$ (11.3), und am Anfang und Ende der Formel die Funktionen $\rho_j^{(r+2)}(u) = s_j$ durch die Formeln $\gamma_j^{(r+2)}(u, s_j)$ (11.6).

$$
\begin{aligned}
&(s_1, \ldots, s_{r+2}) \xrightarrow{P} (t_1, \ldots, t_{r+2}) \;:= \\
&\qquad \exists a \;\; \exists b \;\; \exists m \\
&\qquad \exists x \;\Big(\; \varphi_{sel}(a, b, s(0), x) \;\wedge\; \gamma_1^{(r+2)}(x, s_1) \;\wedge\; \cdots \;\wedge\; \gamma_{r+2}^{(r+2)}(x, s_{r+2}) \;\Big) \quad \text{und} \\
&\qquad \Big(\; \forall i < m \;\; \exists u \;\; \exists v : \\
&\qquad\qquad \varphi_{sel}(a, b, s(i), u) \;\wedge\; \varphi_{sel}(a, b, s(s(i)), v) \quad \text{und} \\
&\qquad\qquad (\rho_1^{(r+2)}(u), \ldots, \rho_{r+2}^{(r+2)}(u)) \xrightarrow{P} (\rho_1^{(r+2)}(v), \ldots, \rho_{r+2}^{(r+2)}(v)) \;\Big) \quad \text{und} \\
&\qquad \exists y \;\Big(\; \varphi_{sel}(a, b, s(m), y) \wedge \bigwedge_{j=1}^{r+2} \gamma_j^{(r+2)}(y, t_j) \;\Big)
\end{aligned}
$$

Im letzten Schritt bringen wir den Mittelteil der Formel in Ordnung.
Wir ersetzen dort die $\rho_j^{(r+2)}(u) = s_j$ durch die Formeln $\gamma_j^{(r+2)}(u, s_j)$ (11.6), und $\stackrel{P}{\to}$ wird durch die Formel δ_p ersetzt (11.1). Damit die „Funktionswerte“ von der $(r+2)$-Tupel-Dekodierung zu δ_p übertragen werden können, müssen wir die $\exists x_i$ einführen.
Damit ist die Σ_1-Formel σ_P wie folgt definiert.

$$\begin{aligned}
&\sigma_P(s_1, \ldots, s_{r+2}, t_1, \ldots, t_{r+2}) \ :=\\
&\qquad \exists a \ \ \exists b \ \ \exists m\\
&\qquad \exists x \ \big(\ \varphi_{sel}(a, b, s(0), x) \ \wedge \ \textstyle\bigwedge_{j=1}^{r+2} \gamma_j^{(r+2)}(x, s_j) \ \big) \ \wedge\\
&\qquad \Big(\ \forall i < m \ \ \exists u \ \ \exists v : \quad \varphi_{sel}(a, b, s(i), u) \ \wedge \ \varphi_{sel}(a, b, s(s(i)), v) \ \wedge\\
&\qquad\qquad \exists x_1 \cdots \exists x_{r+2} \exists y_1 \cdots \exists y_{r+2} :\\
&\qquad\qquad\qquad \bigwedge_{j=1}^{r+2} \gamma_j^{(r+2)}(u, x_j) \wedge \bigwedge_{j=1}^{r+2} \gamma_j^{(r+2)}(v, y_j) \quad \wedge \quad \delta_P(x_1, \ldots, x_{r+2}, y_1, \ldots, y_{r+2}) \ \Big)\\
&\qquad \exists y \ \big(\ \varphi_{sel}(a, b, s(m), y) \ \wedge \ \bigwedge_{j=1}^{r+2} \gamma_j^{(r+2)}(y, t_j) \ \big)
\end{aligned}$$

Gemäß (11.2)(11.3)(11.6)(11.1) leistet σ_P „offensichtlich“ das Gewünschte ... □

Satz 11.8

Es gibt eine Σ_1-Formel $\psi_K(x)$ mit einer freien Variablen, so dass für alle $n \in \mathbb{N}$ gilt:
$n \in K$ gdw. $\mathfrak{N} \models_{\mathcal{P}} \psi_K(\overline{n})$.

Beweis:
Da K semi-entscheidbar ist,
gibt es ein URM-Programm P mit r Registern und Länge ℓ für χ'_K.
Für $\xrightarrow{P}^*$ gibt es die Σ_1-Formel σ_P (11.7).
Die folgende Σ_1-Formel $\psi_K(x)$ prüft, ob man von der Startkonfiguration $(0, x, 0, \ldots, 0)$ eine Konfiguration $(z_0, \ldots, z_{r+1})$ mit $z_{r+1} \geqslant \ell$ erreichen kann. In einer solchen Konfiguration hält die Berechnung von $P(x)$. Das gilt genau dann, wenn $x \in K$.

$$\psi_K(x) \;:=\; \exists z_0\, \exists z_1\, \cdots\, \exists z_{r+1}\quad \ell \leqslant z_{r+1} \wedge \sigma_P(\, \underbrace{0, x, 0, \ldots, 0}_{\text{Startkonfiguration von } P(x)}\, ,\, \overbrace{z_0, \ldots, z_{r+1}}^{\text{Endkonfiguration}}\,)$$

Was haben wir in dieser Vorlesung gelernt?

- Wir kennen Δ_0-beschreibbare Mengen und Σ_1-beschreibbare Mengen.
- Wir wissen, wie man die Existenz einer haltenden URM-Berechnung mit einer Σ_1-arithmetischen Formel darstellen kann.
  Wir kennen Gödels β-Lemma und die Darstellung einzelner Programmschritte durch Δ_0-Formeln.
- Wir wissen, dass Beweisbarkeit einer Formel in Robinsons Σ_1-Arithmetik und Feststellen, ob eine Berechnung hält, „im Prinzip“ das Gleiche ist.

Vorlesung 12: Die Unvollständigkeitssätze

2. Unvollständigkeitssätze

12.1 Unvollständigkeitssätze von Gödel

Wir wissen bereits, dass PA-BEWEISBAR semi-entscheidbar ist (10.5).

Wir zeigen jetzt, dass $\mathfrak{N}$-SAT nicht semi-entscheidbar ist (12.1).

Damit folgt, dass PA-BEWEISBAR und $\mathfrak{N}$-SAT verschieden sind (12.3).

Für $\mathcal{Q}$-BEWEISBAR hatten wir auch eine Formel gefunden, die den Unterschied zeigt: die Formel $\forall x\ 0 + x = x$ kann man nicht aus den Robinson-Axiomen herleiten, sie wird aber vom Standardmodell $\mathfrak{N}$ erfüllt (7.3).

Wir werden zeigen, dass man eine entsprechende Formel für jedes Beweissystem konstruieren kann – d.h. insbesondere für die Peano-Axiome und für jede semi-entscheidbare Axiomenmenge für die Arithmetik.

Lemma 12.1

$\mathfrak{N}$-SAT *ist nicht semi-entscheidbar.*

Beweis:
K wird durch eine arithmetische Σ_1-Formel $\psi_K(x)$ beschrieben (11.8).
D.h. für alle $n \in \mathbb{N}$ gilt: $n \in K$ gdw. $\mathfrak{N} \models_{\mathcal{P}} \psi_K(\overline{n})$.
Folglich gilt für alle $n \in \mathbb{N}$: $n \in \overline{K}$ gdw. $\mathfrak{N} \not\models_{\mathcal{P}} \psi_K(\overline{n})$ (Satz 11.8)
gdw. $\mathfrak{N} \models_{\mathcal{P}} \neg\psi_K(\overline{n})$ (Semantik von $\neg$, da $\neg\psi_K(\overline{n})$ geschlossen ist)

Sei angenommen, dass $\mathfrak{N}$-SAT semi-entscheidbar ist.
Also wird $\chi'_{\mathfrak{N}\text{-SAT}}$ von einem URM-Programm P berechnet.
Bahauptung: folgendes Programm R berechnet die semi-charakteristische Funktion $\chi'_{\overline{K}}$ von $\overline{K}$.

R: Eingabe n
starte $P(\neg\psi_K(\overline{n}))$ (d.h. berechne $\chi'_{\mathfrak{N}\text{-SAT}}(\neg\psi_K(\overline{n}))$)

Es gilt für alle $n \in \mathbb{N}$: $n \in \overline{K} \overset{s.o.}{\Leftrightarrow} \neg\psi_K(\overline{n}) \in \mathfrak{N}\text{-SAT} \Leftrightarrow R$ hält bei Eingabe n (mit Ausgabe 1).

Also wird $\chi'_{\overline{K}}$ von R berechnet – d.h. $\overline{K}$ ist semi-entscheidbar: das ist ein Widerspruch zu (9.14).
Also ist $\mathfrak{N}$-SAT nicht semi-entscheidbar. ✓

Satz 12.2 (Unvollständigkeitssatz von Gödel)

Sei T eine semi-entscheidbare Axiomen-Menge.
Dann gilt T-BEWEISBAR $\neq \mathfrak{N}$-SAT.

Beweis:
Nach (10.5) ist T-BEWEISBAR $= \{\alpha \mid T \vdash_{\overline{\mathcal{P}}} \alpha\}$ semi-entscheidbar,
und nach (12.1) ist $\mathfrak{N}$-SAT nicht semi-entscheidbar.

Also können die beiden Mengen nicht gleich sein. ✓

Folgerung 12.3

PA-BEWEISBAR $\neq \mathfrak{N}$-SAT

Satz 12.4 (für jedes Beweissystem können wir eine „Lücke“ konstruieren)

Für jede semi-entscheidbare Menge $T \subseteq \mathfrak{N}\text{-SAT}$ gibt es ein $\ell \in \mathbb{N}$, so dass
$\neg\psi_K(\overline{\ell}) \in \mathfrak{N}\text{-SAT}$ *und* $\neg\psi_K(\overline{\ell}) \notin T$ *($\neg\psi_K(\overline{\ell})$ wird von $\mathfrak{N}$ erfüllt, aber nicht von T bewiesen).*

Beweis:

Die Menge $S = \{m \mid \neg\psi_K(\overline{m}) \in T\}$ ist semi-entscheidbar.

Eingabe n
$\chi'_T(\neg\psi_K(\overline{n}))$

Also gibt es eine URM P_ℓ, die χ'_S berechnet.

Nun gilt $\ell \notin K$, da $\ell \in K \overset{K}{\Rightarrow} \ell \in S \overset{S}{\Rightarrow} \neg\psi_K(\overline{\ell}) \in T \overset{\text{Vor.}}{\Rightarrow} \mathfrak{N} \models_{\mathcal{P}} \neg\psi_K(\overline{\ell}) \overset{\neg}{\Rightarrow} \mathfrak{N} \not\models_{\mathcal{P}} \psi_K(\overline{\ell}) \overset{(11.8)}{\Rightarrow} \ell \notin K$.

Aus $\ell \notin K$ können wir die Aussage des Satzes für ℓ folgern:

$\ell \notin K$

$\Rightarrow \ell \notin K$ und $\ell \notin S$ (da $\ell \in S$ gdw. $P_\ell(\ell)$ hält, gilt $\ell \in K \Leftrightarrow \ell \in S$)

$\Rightarrow \mathfrak{N} \not\models_{\mathcal{P}} \psi_K(\overline{\ell})$ und $\neg\psi_K(\overline{\ell}) \notin T$ (Eigenschaft von ψ_K (11.8), Definition von S)

$\Rightarrow \mathfrak{N} \models_{\mathcal{P}} \neg\psi_K(\overline{\ell})$ und $\neg\psi_K(\overline{\ell}) \notin T$ (Semantik von $\neg$) ✓

12.2 Anhang

Zum Abschluss beweisen wir noch ein paar Sätze, die die bisherigen Resultate abrunden.

- $\overline{\mathfrak{N}\text{-SAT}}$ ist nicht semi-entscheidbar.
- Semi-entscheidbare Mengen sind genau die Σ_1-beschreibbaren Mengen.

Lemma 12.5

$\overline{\mathfrak{N}\text{-SAT}}$ *ist nicht semi-entscheidbar.*

Beweis:
Für jede Formel φ gilt: entweder $\mathfrak{N} \models_{\mathcal{P}} \varphi$ oder $\mathfrak{N} \not\models_{\mathcal{P}} \varphi$.
Aufgrund der Semantik von $\neg$ ist das gleichbedeutend mit: entweder $\mathfrak{N} \models_{\mathcal{P}} \varphi$ oder $\mathfrak{N} \models_{\mathcal{P}} \neg cl(\varphi)$.
Das heißt $\varphi \in \mathfrak{N}\text{-SAT}$ gdw. $\neg cl(\varphi) \in \overline{\mathfrak{N}\text{-SAT}}$.

Sei angenommen, dass $\overline{\mathfrak{N}\text{-SAT}}$ semi-entscheidbar ist – d.h. $\chi'_{\overline{\mathfrak{N}\text{-SAT}}}$ ist berechenbar.
Dann berechnet folgendes Programm R die semi-charakteristische Funktion $\chi'_{\mathfrak{N}\text{-SAT}}$ von $\mathfrak{N}\text{-SAT}$.

R : Eingabe φ
berechne $\chi'_{\overline{\mathfrak{N}\text{-SAT}}}(\neg cl(\varphi))$

Es gilt: $\varphi \in \mathfrak{N}\text{-SAT} \overset{s.o.}{\Leftrightarrow} \neg cl(\varphi) \in \overline{\mathfrak{N}\text{-SAT}} \Leftrightarrow$ R hält bei Eingabe φ (mit Ausgabe 1).

Also ist $\mathfrak{N}\text{-SAT}$ semi-entscheidbar – das ist ein Widerspruch zu (12.1).

Folglich ist $\overline{\mathfrak{N}\text{-SAT}}$ nicht semi-entscheidbar. ✓

Definition 12.6 (Δ_0- und Σ_1-arithmetische Mengen)

Sei $\mathcal{R} \subseteq \mathbb{N}^k$ eine k-stellige Relation und
$\varphi(x_1, \ldots, x_k)$ eine arithmetische Δ_0-Formel (bzw. eine Σ_1-Formel).

φ beschreibt $\mathcal{R}$, falls für alle $c_1, \ldots, c_k \in \mathbb{N}$ gilt:

$(c_1, \ldots, c_k) \in \mathcal{R}$ genau dann, wenn $\mathfrak{N} \models_{\mathcal{P}} \varphi(\overline{c_1}, \ldots, \overline{c_k})$.

Eine Relation heißt Δ_0-airthmetisch (bzw. Σ_1-arithmetisch),
falls es eine arithmetische Δ_0-Formel (bzw. eine Σ_1-Formel) gibt, die sie beschreibt.

Σ_1-Beschreibbarkeit ist Semi-Entscheidbarkeit

Satz 12.7 (Σ_1-Beschreibbarkeit ist Semi-Entscheidbarkeit)

Sei A eine Menge. A ist Σ_1-beschreibbar genau dann, wenn A semi-entscheidbar ist.

Beweis(skizze):

„$\Rightarrow$“: Sei $\varphi(y)$ eine Σ_1-Formel.

Wir hatten gesehen (Übungsaufgabe),
dass $\varphi(y)$ äquivalent ist zu einer Formel $\exists x \psi(x, y)$, wobei $\psi(x, y)$ eine Δ_0-Formel ist.

$\mathfrak{N} \models_{\mathcal{P}} \psi(\overline{a}, \overline{b})$ kann man durch einen Aufruf `check`$(\psi(\overline{a}, \overline{b}))$ mit folgender rekursiver Methode `check`(α) entscheiden:

`check`(α):

- wenn $\alpha = \bot$, dann `return False`
- wenn α ein Atom $\sigma = \tau$ ist, dann `return` $\sigma^{\mathfrak{N}} = \tau^{\mathfrak{N}}$
- wenn $\alpha = \beta \to \gamma$, dann `return (not check(`$\beta$`)) or check(`$\gamma$`)`
- wenn $\alpha = \exists x \leqslant \tau\, \beta(x)$, dann `return` $\bigvee_{n=0}^{\tau^{\mathfrak{N}}}$ `check(`$\beta(\overline{n})$`)`
- wenn $\alpha = \forall x \leqslant \tau\, \beta(x)$, dann `return` $\bigwedge_{n=0}^{\tau^{\mathfrak{N}}}$ `check(`$\beta(\overline{n})$`)`

Offensichtlich gilt $\mathfrak{N} \models_{\mathcal{P}} \psi(\overline{a}, \overline{b})$ gdw. `check(`$\psi(\overline{a}, \overline{b})$`)`.

Die semi-charakteristische Funktion von $\{n \mid \mathfrak{N} \models\!\!\!\models_{\mathcal{P}} \exists y\, \psi(y, \overline{n})\}$ wird vom folgenden Algorithmus berechnet:

```
Eingabe n
y=0
solange not check(ψ(ȳ, n̄)) :
      y = y+1
Ausgabe 1
```

Wenn $\mathfrak{N} \models\!\!\!\models_{\mathcal{P}} \varphi(\overline{n})$,
dann gibt es ein $y \in \mathbb{N}$ mit $\mathfrak{N} \models\!\!\!\models_{\mathcal{P}} \psi(\overline{y}, \overline{n})$ und der Algorithmus hält und gibt 1 aus.

Wenn $\mathfrak{N} \not\models\!\!\!\models_{\mathcal{P}} \varphi(\overline{n})$,
dann gibt es kein $y \in \mathbb{N}$ mit $\mathfrak{N} \models\!\!\!\models_{\mathcal{P}} \psi(\overline{y}, \overline{n})$ und der Algorithmus hält nicht.

Also berechnet der Algorithmus die semi-charakteristische Funktion der von $\varphi(x)$ beschriebenen Menge. □

„$\Leftarrow$“: (wie der Beweis von (11.8) mit A statt K)

Da A semi-entscheidbar ist,

gibt es ein URM-Programm P mit r Registern und Länge ℓ für χ'_K.

Für $\xrightarrow{P}^*$ gibt es die Σ_1-Formel σ_P (11.7).

Die folgende Σ_1-Formel $\psi_A(x)$ prüft, ob man von der Startkonfiguration $(0, x, 0, \ldots, 0)$ eine Konfiguration $(z_0, \ldots, z_{r+1})$ mit $z_{r+1} \geqslant \ell$ erreichen kann.

In einer solchen Konfiguration hält die Berechnung von $P(x)$.

Das gilt genau dann, wenn $x \in A$.

$$\psi_A(x) \;:=\; \exists z_0\, \exists z_1\, \cdots\, \exists z_{r+1} \quad \ell \leqslant z_{r+1} \wedge \sigma_P(\, \underbrace{0, x, 0, \ldots, 0}_{\text{Startkonfiguration von } P(x)}\, ,\, \overbrace{z_0, \ldots, z_{r+1}}^{\text{Endkonfiguration}}\,)$$

✓

Was haben wir in dieser Vorlesung gelernt?

- Wir kennen Gödels Unvollständigkeitssatz in verschiedenen Fassungen und können die Sätze beweisen.