Paarkodierung

Katharina Wendler

15. Januar 2025

1 Paarkodierung

Wir haben nun bereits gesehen wie man die reellen Zahlen und Folgen von reellen Zahlen konstruktiv definieren kann. Der nächste Schritt wäre nun sich Summen und Reihen von reellen Zahlen anzuschauen. Damit wir mit diesen sinnvoll arbeiten können, brauchen wir das Cauchy-Produkt bzw. allgemeiner Doppelsummen.
Doppelsummen können wir konstruktiv beschreiben durch Paarkodierungen. Für das Cauchy-Produkt brauchen wir insbesondere eine Paarkodierung um Produkte $x_i y_j$ ordnen zu können. Deshalb werden wir im Folgenden näher auf die Wurzelkodierung, die Kodierung über Gauß Summen und über quadratische Gauß Summen eingehen.
Das **Ziel** dieses Vortrags wird insbesondere sein, die Gleichwertigkeit der quadratische Gauß-Summen Kodierung und der Wurzelkodierung zu zeigen. Für nähere Details zu den Beweisen und Aussagen, die hier ausgelassen oder nicht bewiesen werden, wird in Minlog auf natpair.scm verwiesen.
Wir verwenden die Variablen k, i, j, n, m, l vom Typ $\mathbb{N}$ und ml, ij vom Typ $\mathbb{N} \times \mathbb{N}$.

1.1 Wurzelkodierung

Die Kodierung mit Wurzeln folgt aus der folgenden Beobachtung.

Bemerkung 1. Wir können jede natürliche Zahl $k \in \mathbb{N}$ schreiben als

$$k = m^2 + l \quad \text{mit } l \leq m + m \tag{1}$$

für gewisse $m, l \in \mathbb{N}$. Es lässt sich insbesondere zeigen, dass diese Darstellung eindeutig ist. (RtCtoMlUniq)

Aus dem Paar (m, l) das wir nach der obigen Bemerkung für jedes $k \in \mathbb{N}$ er halten können wir wiederrum Koordinaten (i, j) folgern, sodass die Wurzelkodierung für beispielsweise $k \leq 16$ wie folgt veranschaulicht werden kann:

3	12	13	14	15	
2	6	7	8	11	
1	2	3	5	10	
0	0	1	4	9	16
	0	1	2	3	4

Tabelle 1: Wurzelkodierung

Wir können nun definieren die Konversion von k zu (m, l) definieren durch:

Definition 1.1 (RtCToMl)
```
(add-program-constant "RtCToMl" (py "nat=>nat yprod nat"))
(add-computation-rules
    "RtCToMl Zero" "Zero pair Zero"
    "RtCToMl(Succ k)"
    "([ml][if (rht ml<lft ml+lft ml)
              (lft ml pair Succ rht ml)
              (Succ lft ml pair Zero)])(RtCToMl k)")
```

Wir können sehen, dass RtCToMl total ist.

Bemerkung 2 (RtCToMlStep)**.** Für beliebige $k, m, l \in \mathbb{N}$ gilt, wenn $(m, l) = \text{RtCToMl } (k)$, dann ist

$$\text{RtCToMl } (k+1) = \begin{cases} (m, l+1) & l < m + m \\ (m+1, 0) & \text{sonst.} \end{cases}$$

Für diese Programmkonstante kann man folgendes zeigen:

Proposition 1.1 (RtCToMlProp1 und RtCToMlProp2)
Für alle k, m, l gilt

$$(m \; pair \; l) = RtCToMl(k) \quad \longleftrightarrow \quad k = m * m + l \quad andnc \quad l <= m + m$$

Wir können nun eine weitere totale Programmkonstante definieren und wollen anschließend zeigen, dass diese Inverse zu RtCToMl ist.

Definition 1.2 (RtMlToC)
```
(add-program-constant "RtMlToC" (py "nat yprod nat=>nat"))
(add-computation-rules
    "RtMlToC(m pair l)" "m*m+l")
```

Satz 1.1 (RtMlToCToMlId)
```
(set-goal "all k RtMlToC(RtCToMl k)=k")
```

Satz 1.2 (RtCToMlToCId)
```
(set-goal "all ml(rht ml<=lft ml+lft ml -> RtCToMl(RtMlToC ml)=ml)")
```

Diese Aussage lässt sich auch zeigen wenn man setzt ml = (m pair l).

Durch diese beiden Theoreme wird deutlich, dass RtMlToC und RtCToMl wirklich Inverse sind. Wir können jetzt eindeutig die (m, l) für jedes $k \in \mathbb{N}$ bestimmen, wollen jetzt aber (m, l) noch in die passenden Koordinaten (i, j) für unsere Wurzelkodierung umwandeln. Dazu definieren wir zwei weitere totale „Programmkonstanten“:

Definition 1.3 (RtMlToIj)

```
(add-program-constant "RtMlToIj" (py "nat yprod nat=>nat yprod nat"))
(add-computation-rules
    "RtMlToIj(m pair l)" "[if (l<m) (m pair l) (l- -m pair m)]")
```

Definition 1.4 (RtIjToMl)

```
(add-program-constant "RtIjToMl" (py "nat yprod nat=>nat yprod nat"))
(add-computation-rules
    "RtIjToMl(i pair j)" "(i max j) pair [if (j<i) j (j+i)]")
```

Damit diese Definitionen wirklich sinnvoll sind, müssen wir auch hier wieder zeigen, dass RtMlToIj und RtIjToMl invers sind. Dafür brauchen wir ein zusätzliches Lemma (RtIjToMlEst).

Satz 1.3 (RtMlToIjToMlId und RtIjToMlToIjId)
Zum einen gilt

```
(set-goal "all ij RtMlToIj(RtIjToMl ij)=ij"
```

und zum anderen

```
(set-goal
    "all ml
        (rht ml<=lft ml+lft ml ->
            RtIjToMl(RtMlToIj ml)=ml )")
```

Somit können wir nun allgemeiner totale Programmkonstanten definieren die uns $k \in \mathbb{N}$ dekodieren bzw. Koordinaten (i, j) kodieren.

Definition 1.5 (RtC und RtD)
Um (i, j) zu kodieren nutzen wir

```
(add-program-constant "RtC" (py "nat yprod nat=>nat"))
(add-computation-rules
    "RtC ij" "RtMlToC(RtIjToMl ij)")
```

und um die Koordinaten von k zu ermitteln haben wir

```
(add-program-constant "RtD" (py "nat=>nat yprod nat"))
(add-computation-rules
    "RtD k" "RtMlToIj(RtCToMl k)")
```

Wie schon zu vermuten ist, können wir insbesondere zeigen, dass *RtC* und *RtD* invers zu einander sind:

Satz 1.4 (RtDCId und RtCDId)
Die Programmkonstanten *RtC* und *RtD* sind invers zueinander, d.h. wir können zeigen

```
(set-goal "all ij RtD(RtC ij)=ij")
(set-goal "all k RtC(RtD k)=k")
```

Ferner lassen sich noch viele weitere nützliche Eigenschaften und Schranken für *RtC* und *RtD* zeigen (siehe natpair.scm). Durch diese Eigenschaften können wir schließlich folgende Schranken für *RtC* und *RtD* ermitteln:

Proposition 1.2 (RtCBd)

```
(set-goal
    "all n,i,j
        (i<=n ->
            j<=n ->
                RtC(i pair j)<=n*n+n+n)")
```

Auch hier können wir statt i und j die Variable ij verwenden mit $\mathrm{rht}\, ij = j$ und $\mathrm{lft}\, ij = i$.

Proposition 1.3 (RtDBd)

```
(set-goal
    "all n,i,j,k
        (k<=n*n+n+n ->
            (i pair j)=RtD k ->
                    (i max j)<=n)")
```

1.2 Kodierung mittels Gauß-Summen

Eine andere Art der Paarkodierung ist die Kodierung über Gauß Summen.

Definition 1.6 (Gauß Summe)

```
(add-program-constant "Gs" (py "nat=>nat"))
(add-computation-rules
    "Gs Zero" "Zero"
    "Gs(Succ n)" "Succ(Gs n)+n")
```

Hier müssen wir prüfen, dass Gs eine totale Programmkonstante und wirklich die Gauß Summe darstellt ist, d.h es gilt $\forall n\colon\ Gs(n+1) = Gs(n) + (n+1)$. Weiterhin können wir die Gaußsche Summenformel $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ beweisen.
Wie bei der Wurzelkodierung wollen wir nun für jedes k eine eindeutige Darstellung durch ein Paar (m, l) mittels der Gauß-Summe finden und anschließend noch die passenden Koordinaten (i, j) zu k finden.

Bemerkung 3. Für jede Zahl k gibt es eindeutige (m, l) mit $l \leq m$, sodass $k = Gs(m) + l$ (siehe für Eindeutigkeit (GsCtoMlUniq) in Minlog). Die Kodierung erfolgt dann „in Diagonalen“, sieht also wie folgt aus:

4	10				
3	6	11			
2	3	7	12		
1	1	4	8	13	
0	0	2	5	9	14
	0	1	2	3	4

Tabelle 2: Kodierung durch Gs

Bemerke, man kann (m, l) bzgl. der obigen kodierung auch so interpretieren, dass m die Diagonalen nummeriert und l die Entfernung des Codes k vom anfang der Diagonale angibt.

Nun wollen wir Programmkonstanten für die Konversion von k zu (m, l) und umgekehrt einführen.

Definition 1.7 (GsCToMl)
```
(add-program-constant "GsCToMl" (py "nat=>nat yprod nat"))
(add-computation-rules
    "GsCToMl Zero" "Zero pair Zero"
    "GsCToMl(Succ k)"
    "([ml][if (rht ml<lft ml)
              (lft ml pair Succ rht ml)
              (Succ lft ml pair Zero)])(GsCToMl k)")
```

Anders gesagt haben wir folgende „Schrittberechnungsregel“ - für $(m, l) = GsCtoMl(k)$ ist

$$GsCToMl(k+1) = \begin{cases} (m, l+1) & \text{falls } l < m \\ (m+1, 0) & \text{sonst} \end{cases}$$

Bemerkung 4 (GsCToMlProp)**.** Es lässt sich insbesondere zeigen, dass für k, m, l gilt

$$(m, l) = GsCToMl(k) \quad \Leftrightarrow \quad k = Gs(m) + l \quad andnc \quad l \leq m$$

Mit dieser Bemerkung folgt die Existenz der Darstellung von k über die Gauß-Summe und wir können nun definieren:

Definition 1.8 (GsMlToC)
```
(add-program-constant "GsMlToC" (py "nat yprod nat=>nat"))
(add-computation-rules
     "GsMlToC(m pair l)" "Gs m+l")
```

Ähnlich wie zuvor für die Wurzelkodierung können wir zeigen, dass diese beiden Programmkonstanten total und invers zueinander sind.

Satz 1.5 (GsMlToCToMlId)
```
(set-goal "all k GsMlToC(GsCToMl k)=k")
```

Satz 1.6 (GsCToMlToCId)

```
(set-goal
    "all ml
        (rht ml<=lft ml ->
                GsCToMl(GsMlToC ml)=ml)")
```

Ferner haben wir eine Bijektion zwischen den Paaren (m, l) und den Koordinaten (i, j), die durch folgende totale Programmkonstante gegeben sind:

Definition 1.9 (GsMlToIj)

```
(add-program-constant "GsMlToIj" (py "nat yprod nat=>nat yprod nat"))
(add-computation-rules
 "GsMlToIj(m pair l)" "l pair m - -l")
```

Bemerkung 5. Da wir auf natürlichen Zahlen arbeiten, definieren wir den Minus-Operator etwas anders wie auf den reellen Zahlen, nämlich:

$$m - -l = \begin{cases} m - l & \text{falls } l < m \\ 0 & \text{sonst} \end{cases}$$

Definition 1.10 (GsIjToMl)

```
(add-program-constant "GsIjToMl" (py "nat yprod nat=>nat yprod nat"))
(add-computation-rules
     "GsIjToMl(i pair j)" "i+j pair i")
```

Auch hier können wir zeigen, dass $GsIjToMl$ und $GsMlToIj$ bijektiv agieren, denn es gilt für m, l, ij mit $(m, l) = GsIjToMl(ij)$, dass $l \leq m$ und somit lässt sich leicht nachrechnen

Satz 1.7 (GsMlToIjoMlId)

```
(set-goal "all ij GsMlToIj(GsIjToMl ij)=ij")
```

Satz 1.8 (GsIjToMlToIjId)

```
(set-goal "all ml(rht ml<=lft ml -> GsIjToMl(GsMlToIj ml)=ml)")
```

Zu guter letzt können wir nun mit den obigen Programmkonstanten zwei weitere totale Programmkonstanten definieren, die die Kodierung bzw. Dekodierung von k zu den Koordinaten (i, j) definieren.

Definition 1.11 (GsC)

```
(add-program-constant "GsC" (py "nat yprod nat=>nat"))
(add-computation-rules
     "GsC(i pair j)" "GsMlToC(GsIjToMl(i pair j))")
```

Definition 1.12 (GsD)

```
(add-program-constant "GsD" (py "nat=>nat yprod nat"))
(add-computation-rules
    "GsD k" "GsMlToIj(GsCToMl k)")
```

Aus den Definitionen und den vorherigen Bijektionen können wir nun auch direkt ableiten, dass GsC und GsD invers zueinander sind:

Satz 1.9 (GsDCId und GsCDId)
Es gilt

```
(set-goal "all ij GsD(GsC ij)=ij")
```

und

```
(set-goal "all k GsC(GsD k)=k")
```

Für GsC und GsD lassen sich nun auch wieder mehrere Eigenschaften und Schranken beweisen, für näheres verweisen wir hier auch auf natpair.scm.

1.3 Quadratische Gauß Summe

Um nun die Kodierung durch Gauß-Summen mit der Wurzelkodierung in Relation zusetzen, brauchen wir wir eine Kodierung, die auf der quadratischen Gauß-Summe basiert. Bei der Wurzelkodierung haben wir die Koordinaten in Tabelle 1 durch ein n-Quadrat eingeschränkt. Bei der Gauß-Summen Kodierung haben wir dies jedoch nicht getan und deshalb brauchen wir nun die quadratischen Gauß-Summen.

Bemerkung 6. Die Kodierung, die wir gerne hätten, sieht für ein 3-Quardat wie folgt aus

3	6	10	13	15
2	3	7	11	14
1	1	4	8	12
0	0	2	5	9
	0	1	2	3

Tabelle 3: Kodierung durch quadratische Gs ($n = 3$ und $k < (n+1)^2 = 16$)

Man sieht leicht, dass dies im unteren Dreieck der Gs-Kodierung entspricht, im oberen Dreieck aber nicht, somit brauchen wir eine Fallunterscheidung zwischen diesen Dreiecken. Wir unterscheiden für $k < (n+1)^2$ in einem n-Quadrat wie folgt:

- k liegt im oberen Dreieck, wenn $k \geq Gs(n+1)$
- k liegt im unteren Dreieck, wenn $k < Gs(n+1)$

Um die Kodierung für das obere Dreieck zu definieren brauchen wir folgende Definitionen:

Definition 1.13 (decreasing Gauß sum)

```
(add-program-constant "Gds" (py "nat=>nat=>nat"))
(add-computation-rules
 "Gds n Zero" "Zero"
 "Gds n(Succ m)" "(Gds n m)+(n - -m)")
```

Es folgt schnell, dass Gds total ist und $Gds(n,m) = \sum_{i=0}^{m-1}(n--i)$ (NatEqGdsSum).
Wir können noch folgende weitere Eigenschaften zeigen:

- Es gilt $Gds(n,m+1) = Gds(n,m) + n--m$.
- Es gilt $Gds(n,n) = Gs(n)$.
- Für jede Zahl $k = Gds(n,m) + l$ mit $l \leq n--m$ sind m, l eindeutig bestimmt.

Angenommen wir betrachten k im oberen Dreieck, so sei $k_0 := k - Gs(n+1)$. Bemerke, dass dann wegen $k < (n+1)^2$ gilt $k_0 < Gs(n)$. Wir wollen nun für ein gegebenes n und k_0 das Paar (m,l) bestimmen, dafür müssen wir wiederum eine Unterscheidung treffen und definieren somit zwei totale Programmkonstanten:

Definition 1.14 (GdsCToMlLt)
```
(add-program-constant "GdsCToMlLt" (py "nat=>nat=>nat yprod nat"))
(add-computation-rules
     "GdsCToMlLt n Zero" "Zero pair Zero"
     "GdsCToMlLt n(Succ k)"
     "([ml][if (Succ rht ml<n- -lft ml)
               (lft ml pair Succ rht ml)
               (Succ lft ml pair Zero)])
      (GdsCToMlLt n k)")
```

Definition 1.15 (GdsCToMlEq)
```
(add-program-constant "GdsCToMlEq" (py "nat=>nat yprod nat"))
(add-computation-rules
     "GdsCToMlEq Zero" "Zero pair Zero"
     "GdsCToMlEq(Succ n)" "n pair Zero")
```

Es gilt nun für diese Definitionen:

Proposition 1.4 (GdsCToMlLtProp und GdsCToMlEqProp)
(1) Es gilt für $k+1 < Gs(n)$ und $(m,l) = GdsCToMlLt(n,k)$, dass

$$Gds(n,m) + l = k \quad \text{und} \quad l < n--m$$

(2) Es gilt für $k+1 = Gs(n)$ und $(m,l) = GdsCToMlEq(n)$, dass

$$Gds(n,m) + l = k \quad \text{und} \quad l < n--m$$

Diese Proposition erlaubt uns nun $GdsCToMl$ sinnvoll zu definieren:

Definition 1.16 (GdsCToMl)
```
(add-program-constant "GdsCToMl" (py "nat=>nat=>nat yprod nat"))
(add-computation-rules
     "GdsCToMl n k"
     "[if (Succ k<Gs n)
          (GdsCToMlLt n k)
          (GdsCToMlEq n)]")
```

Zudem gilt nun ähnlich zur obigen Proposition

Proposition 1.5

```
(set-goal
    "all n,k,ml,m,l(
        ml=GdsCToMl n k ->
                m=lft ml ->
                 l=rht ml ->
                     k<Gs n ->
                     Gds n m+l=k andnc l<n--m)")
```

Die zu *GdsCToMl* inverse Programmkonstante ist nun definiert durch:

Definition 1.17

```
(add-program-constant "GdsMlToC" (py "nat=>nat yprod nat=>nat"))
(add-computation-rules
    "GdsMlToC n(m pair l)" "(Gds n m)+l")
```

Das die totalen Programmkonstanten *GdsMlToC* und *GdsCToMl* inverse zueinander sind folgt aus den Sätzen:

Satz 1.10 (GdsCToMlToCId und GdsMlToCToMlId)

```
(set-goal
    "all m,n,l
        (m < n ->
             l < n- -m ->
                 GdsCToMl n(GdsMlToC n(m pair l))=(m pair l))")
```

und

```
(set-goal
    "all n,k
        (k < Gs n ->
            GdsMlToC n(GdsCToMl n k)=k)")
```

Erneut können wir an dieser Stelle zwei totale Programmkonstanten definieren, um für das Paar (m, l) die Koordinaten (i, j) zu bestimmen und umgekehrt.

Definition 1.18 (GqMlToIj)

```
(add-program-constant "GqMlToIj"
                        (py "nat=>nat yprod nat=>nat yprod nat"))
(add-computation-rules
     "GqMlToIj n(m pair l)" "Succ(m+l)pair (n- -l)")
```

Definition 1.19 (GqIjToMl)

```
(add-program-constant "GqIjToMl"
                        (py "nat=>nat yprod nat=>nat yprod nat"))
(add-computation-rules
     "GqIjToMl n(i pair j)" "Pred(i+j- -n)pair (n- -j)")
```

Für $GqIjToMl$ und $GqMlToIj$ lässt sich auch wieder zeigen, dass sie mit gewissen Bedingungen Inverse sind - siehe (GqIjToMlToIjId) und (GqMlToIjToMlId) in natpair.scm. Die bisherigen Definitionen erlauben es uns nun durch die folgenden Programmkonstanten k zu dekodieren und (i, j) zu kodieren.

Definition 1.20 (GqC)
```
(add-program-constant "GqC" (py "nat=>nat yprod nat=>nat"))
(add-computation-rules
 "GqC n(i pair j)"
 "[if (i+j<=n) (GsC(i pair j))
   (Gs(Succ n)+Gds n(lft(GqIjToMl n(i pair j)))
              +rht(GqIjToMl n(i pair j)))]")
```

Definition 1.21 (GqD)
```
(add-program-constant "GqD" (py "nat=>nat=>nat yprod nat"))
(add-computation-rules
     "GqD n k"
     "[if (k<Gs(Succ n))
          (GsD k)
          (GqMlToIj n(GdsCToMl n(k- -Gs(Succ n))))]")
```

Wir können nun mit den folgenden Einschränkungen zeigen, dass auch GqC und GqD invers sind.

Satz 1.11 (GqCDId und GqDCId)
```
(set-goal "all n,k (k<=n*n+n+n -> GqC n(GqD n k)=k)")
```

und

```
(set-goal "all n,ij(lft ij<=n -> rht ij<=n -> GqD n(GqC n ij)=ij)"
```

Auch hier können wir wieder Schranken für GqC und GqD finden. Die im Anschluss aufgeführte Schranke werden wir im nächsten Kapitel nutzen um die Äquivalenz der Kodierungen zu zeigen.

Satz 1.12 (GqCBd und GqCBdMod)
```
(set-goal "all n,i,j(i<=n -> j<=n -> GqC n(i pair j)<=n*n+n+n)")
```

Schreiben wir statt i, j hier ij so erhalten wir (GqCBdMod).

1.4 Relation der Kodierungen

Betrachten wir nun ein festes n und das zugehörige n-Quadrat, so wollen wir zeigen, dass $k < (n+1)^2$ sowohl mit Wurzelkodierung wie auch mit der quadratischen Gauß-Summen Kodierung als Koordinaten (i, j) dargestellt werden kann und es eine Bijektion zwischen den Koordinaten der Kodierungen gibt. Dazu definieren wir folgende totale Programmkonstanten:

Definition 1.22 (GqCRtD)
```
(add-program-constant "GqCRtD" (py "nat=>nat=>nat"))
(add-computation-rules
     "GqCRtD n k" "[if (k<=n*n+n+n) (GqC n(RtD k)) k]")
```

Definition 1.23 (RtCGqD)
```
(add-program-constant "RtCGqD" (py "nat=>nat=>nat"))
(add-computation-rules
     "RtCGqD n k" "[if (k<=n*n+n+n) (RtC(GqD n k)) k]")
```

Dann können wir (mit Minlog) die Bijektion der Koordinaten zeigen.

Satz 1.13 (RtCGqDGqCRtDId)
```
(set-goal "all n,k RtCGqD n(GqCRtD n k)=k")
```

Satz 1.14 (GqCRtDRtCGqDId)
```
(set-goal "all n,k GqCRtD n(RtCGqD n k)=k")
```

Insbesondere haben wir auch folgende Schranken für $GqCRtD$ und $RtCGqD$.

Lemma 1.1 (Bd)
Sei $F = GqCRtD$ oder $F = RtCGqD$, dann gilt für alle n, k:

$$k \leq n^2 + 2n \quad \Rightarrow \quad F(n,k) \leq n^2 + 2n$$