Repräsentation von Datenstrukturen im π-Kalkül

Seminar Theorie kommunizierender Systeme: Der π-Kalkül
Ausarbeitung

1. Einleitung

In dieser Arbeit wird die Darstellung von beliebigen Daten- und Datenstrukturen im π-Kalkül untersucht. Diese Untersuchung wird in diesem Abschnitt mit Hilfe von CCS motiviert. Abschnitt 2 stellt daraufhin prinzipielle Darstellungsidee anhand der Repräsentation einiger einfacher Daten vor. 3 fasst die gewonnenen Erfahrungen aus den bisherigen Abschnitten zusammen.

In CCS kann man einen Buffer der Größe n wie folgt darstellen:

$$Buff^{(n)} := \sum_u in_u.Buff_u^{(n)}$$

$$Buff_{\vec{v},w}^{(n)} := \begin{cases} \sum_u in_u.Buff_{u,\vec{v},w}^{(n)} + \overline{out_{\vec{w}}}.Buff_{\vec{v}}^{(n)} & (|\vec{v}|<n-1) \\ \overline{out_w}.Buff_{\vec{v}}^{(n)} & (|\vec{v}|=n-1) \end{cases}$$

Die Namen in bzw. out stellen Einfüge- bzw. Ausleseoperationen dar. An diesem Beispiel lassen sich zwei Probleme erkennen: Zum einen muss das Kalkül um parametrisierte Prozessbezeichner erweitert werden(hier $Buff$). Diese Änderung zeigt, das das zugrundeliegende Kalkül unvollständig ist. Zum anderen muss die Wertemenge V der möglichen Bufferinhalte endlich sein, da sonst das LTS des Buffers nicht mehr regulär wäre. Dieser Kompromiss reduziert die Expressivität erheblich. Im π-Kalkül hingegen können Datenstrukturen ohne solche Kompromisse dargestellt werden.

2. Darstellung

Der π-Kalkül ist eine Sprache zur Beschreibung nebenläufiger Systeme. Diese Systeme setzen sich aus *Prozessen* zusammen. Daher müssen Datenstruktur im π-Kalkül als *Prozesse* dargestellt werden, eine etwas ungewöhnliche Interpretation einer Datenstruktur. Dementsprechend müssen Berechnungen mit solchen Daten als *Reaktionen* repräsentiert werden. Aber was sollte bei einer solchen Reaktion passieren?

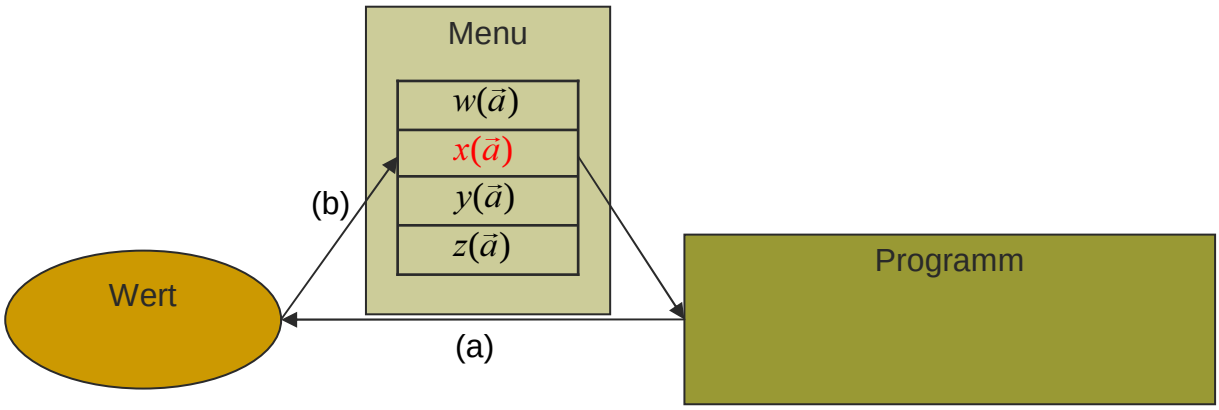

Abbildung 1 – Darstellungsidee

Im Pi-Kalkül sind wir in der Lage, Namen zu verschicken, in seiner polyadischen Version können wir sogar eine ganze Reihe von Namen versenden. Genau diese Fähigkeit hilft dabei, die Grundidee der Darstellung zu realisieren (s. Abbildung 1): Ein Wert im Pi-Kalkül ist ein separater Prozess, der über einen speziellen Kanal ein *Menu* erwartet. Wird dieses *Menu* gesendet (a), wählt der Wert eine *Menuoption* (x) aus (b) und darf gegebenfalls noch selbst eine *Spezifizierung* mitsenden ($\vec{a}$).
Eine denkbare Analogie zu dieser Darstellung wäre zum Beispiel eine Multiple-Choice-klausur mit nur einer Aufgabe: Die Studenten haben die Auswahl zwischen mehreren vorgegeben Antworten, wobei einige davon noch genauere Angaben erfordern (z.B. eine Rechenaufgabe, die Optionen sind verschiedene Formeln, die Antworten die Anwendung der Formel auf die von der Aufgabe vorgegebenen Zahlen). Die Studenten stellen hier die Werte dar, die die Klausur verteilenden Assistenten sind das berechnende Programm, und die Klausuren sind das *Menu*. Die Assistenten berechnen je nach Antwort eine Punktzahl.
In den folgenden Abschnitten werden einige Beispieldaten vorgestellt, die alle das oben vorgestellte *Menuprotokoll* implementieren. Es werden ausserdem einige Operatoren auf diesen Daten definiert und deren Korrektheit gezeigt.

2.1 endliche Datenmengen

Die einfachste noch interessante endliche Menge ist die Menge der Wahrheitswerte $\{true, false\}$. Mít der oben vorgestellten Darstellungsidee lassen sich die Wahrheits-werte recht einfach repräsentieren:

$$True(k) := k(t\ f).\overline{t}$$
$$False(k) := k(t\ f).\overline{f}$$

Beide Werte empfangen ein zweistelliges *Menu* über einen abstrakten Kanal, eine „Lokation“ k; Handelt es sich um $true$, wird t selektiert, sonst f. Entsprechend lässt sich ein Konditional definieren:

$$Cond(P,Q)(k) := (\text{new}\, t\ f)\overline{k}\langle t\ f\rangle.(t.P + f.Q)$$

$Cond$ ist ein abgeleiteter Operator auf Prozessen und erwartet an k einen Wahrheitswert. Diesem wird ein mit new generiertes Menu gesendet, und bei $true$ mit P, sonst mit Q weiter gerechnet. Genau dieses Verhalten von $Cond$ wollen wir nun zunächst formalisieren und anschliessend beweisen:

$$(1)\ \ Cond(P,Q)\langle k\rangle \mid True\langle k\rangle \rightarrow^{*} P$$
$$(2)\ Cond(P,Q)\langle k\rangle \mid False\langle k\rangle \rightarrow^{*} Q$$

(1) lässt sich leicht durch Nachrechnen der Reaktionen modulo struktureller Kongruenz zeigen:

$$\begin{aligned}
&Cond(P,Q)\langle k\rangle \mid True\langle k\rangle = ((\text{new}\, t\, f)\overline{k}\langle t\, f\rangle.(t.P + f.Q)) \mid k(t\, f).\overline{t} \\
&\equiv (\text{new}\, t\, f)(\overline{k}\langle t\, f\rangle.(t.P + f.Q) \mid k(t\, f).\overline{t}) \\
&\rightarrow (\text{new}\, t\, f)(t.P + f.Q \mid \overline{t}) && (a) \\
&\rightarrow (\text{new}\, t\, f)(P) && (b) \\
&\equiv P
\end{aligned}$$

Der Beweis von (2) ist analog.
Wie erwartet finden genau zwei Reaktionen statt: Das Menu wird an den Wert gesendet (a), der Wert selektiert eine Menuoption (b). Der Beweis veranschaulicht aber auch ein dramatisches Problem dieser Darstellung:

Eine Datenstruktur wird durch Rechnung zerstört!

Dieses Problem wird in 2.4 wieder aufgegriffen, und, wie der geneigte Leser wohl schon vermutet, durch Replikation gelöst.
Grössere endliche Mengen lassen sich nach exakt dem gleichen Schema repräsentieren, man erweitert lediglich das Menu (hier z.B. Wochentage):

$$Monday(k) := k(mo\, tu\, we\, th\, fr\, sa\, su).\overline{mo}$$
$$\ldots$$
$$DayCond(P,Q,R,S,T,U,V)(k) := (\text{new}\, mo\, tu\, we\, th\, fr\, sa\, su)(mo.P + \ldots)$$

2.2 unendliche Datenmengen

2.2.1 natürliche Zahlen

Die einfachste unendliche Menge ist die Menge der natürlichen Zahlen $\mathbb{N}$. Natürliche Zahlen werden in der Regel rekursiv definiert: Es gibt einen Null, und jede Zahl grösser Null ist der Nachfolger einer um 1 kleineren Zahl. Für eben jene Rekursion benötigen wir nun die in 2.1 noch vernachlässigten Spezifizierungen $\vec{a}$ aus Abbildung 1 (siehe Abbildung 2): Eine Zahl erhält ein Menu mit zwei Optionen: *null*, falls die Zahl Null ist, ansonsten *succ(k)*, wobei es sich bei *k* um die Lokation ihres Vorgängers handelt. Diese Lokation wird dem rechnenden Programm, wie in Abbildung 2 veranschaulicht, gesendet.

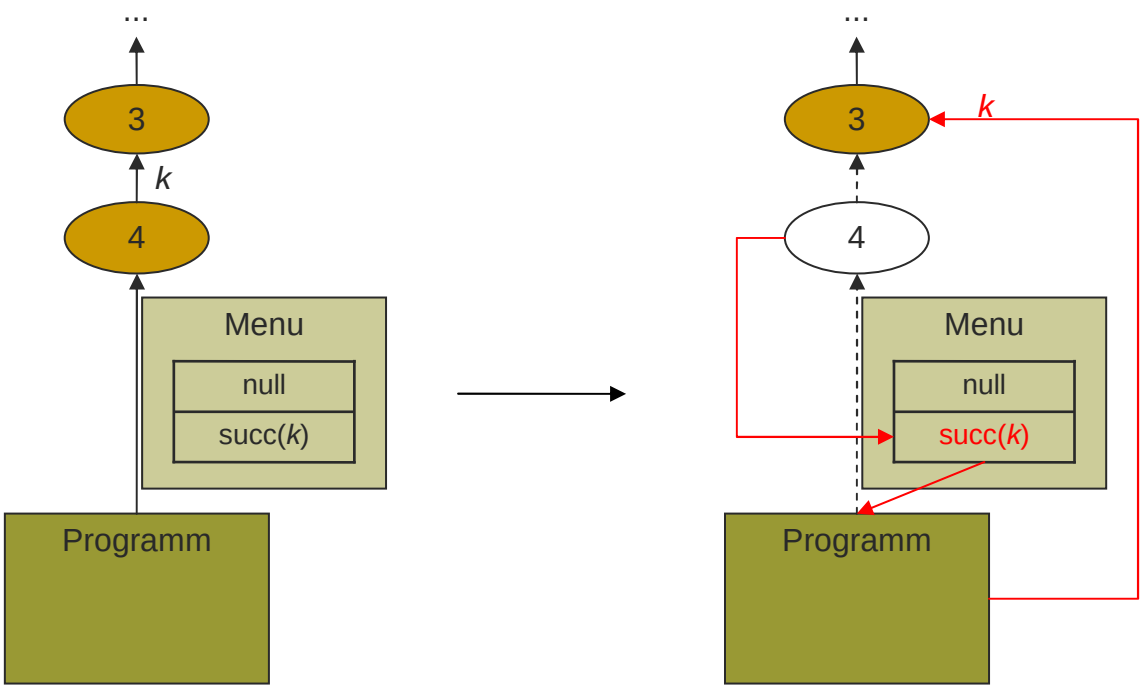

Abbildung 2 – Darstellung natürlicher Zahlen

Der entsprechende Code im π-Kalkül lässt sich jetzt sehr einfach hinschreiben (*null* und *succ* wurden hier mit n und s abgekürzt):

$$Zero := (k).k(n\,s).\overline{n}$$
$$Succ(N) := (k).\mathrm{new}\,l\,(k(n\,s).\overline{s}\,\langle l\rangle \mid N\,\langle l\rangle)$$

$Succ$ legt einen neuen „Zahlknoten“ an, ein kleiner Prozess, der die *Menuselektion* übernimmt, und legt die Vorgängerzahl N an einer dementsprechenden Lokation ab. Jetzt brauchen wir noch einen, zu $Cond$ äquivalenten, Operator zur Analyse von Zahlen:

$$Numbercases(P,F)(k) := (\mathrm{new}\,n\,s)\overline{k}\,\langle n\,s\rangle.(n.P + s.F)$$

Bei F handelt es sich um eine Abstraktion, die gerade die Lokation der Vorgänger-zahl der untersuchten Zahl erwartet.
Von $Numbercases$ verlangen wir, dass es die folgenden Korrektheitseigenschaften erfüllt:

$$(1)\; Succ(N)\langle k\rangle \mid Numbercases(P,F)\langle k\rangle \rightarrow^{*} \mathrm{new}\,l\,(N\,\langle l\rangle \mid F\,\langle l\rangle)$$
$$(2)\; Zero\langle k\rangle \mid Numbercases(P,F)\langle k\rangle \rightarrow^{*} P$$

Beide Bedingungen lassen sich wie schon in 2.1 durch einfaches Nachrechnen der Reaktionen beweisen.
Um die Lesbarkeit zu erhöhen, führen wir noch eine vereinfachte Notation von Numbercases ein:

$$Numbercases(P,F)(k) = \text{ case } k \text{ of}$$
$$Zero \quad \Rightarrow P$$
$$|\, Succ(k') \Rightarrow F\langle k' \rangle$$

Für natürliche Zahlen wären ausser einem Analyseoperator wie $Numbercases$ noch Operatoren wünschenswert, die dekrementieren oder addieren; Ersteres lässt sich wie folgt implementieren:

$$decr(km) := \text{case } k \text{ of}$$
$$Zero \quad \Rightarrow Zero\langle m \rangle$$
$$Succ(k') \Rightarrow m(n\,s).k'\langle n\,s \rangle$$

$decr$ erwartet an k eine Zahl $Succ(N)$ und legt N bei m ab, indem das Menu, dass an m geschickt wird, einfach an die Lokation N *weitergeleitet* wird.
Man würde auf den ersten Versuch hin vermuten, dass die Korrektheitsbedingung folgendermassen aussieht:

$$decr\langle km \rangle \,|\, Succ(N)\langle k \rangle \rightarrow^* N\langle m \rangle$$

Diese Bedingung gilt so jedoch nicht: $decr$ berechnet nicht genau N als Vorgänger von $Succ(N)$, sondern vielmehr eine Zahl N', die sich genauso verhält wie der N! N' macht bei Reaktion lediglich einen internen Schritt mehr, es leitet das ihm gesendete Menu an N weiter. Intuitiv arbeitet $decr$ also korrekt. An dieser Stelle benötigen wir die schwache Äquivalenz $\approx$, die wir schon für CCS kennengelernt haben. $\approx$ respektiert solche internen Schritte wie das „Menu-forwarding“ von $decr$.
Die korrekte Bedingung sieht also folgendermassen aus:

$$decr\langle km \rangle \,|\, Succ(N)\langle k \rangle \rightarrow^* \approx N\langle m \rangle$$

Für die Addition benötigen wir noch einen einfachen Operator, der Zahlen kopiert:

$$Ncopy(lm) := \text{case } l \text{ of}$$
$$Zero \quad \Rightarrow Zero\langle m \rangle$$
$$Succ(l') \Rightarrow \text{new } m'(m(n\,s).s\langle m' \rangle \,|\, Ncopy\langle l'm' \rangle)$$

$Ncopy$ erwartet bei l eine Zahl und kopiert sie nach m. Bei $Zero$ passiert nicht viel, bei einer Zahl grösser $Zero$ legt $Ncopy$ einen neuen Zahlknoten an und verbindet diesen mit einem *Platzhalter* m', an den anschliessend rekursiv die Vorgängerzahl kopiert wird. Entsprechend lässt sich die Korrektheit von $Ncopy$ definieren:

$$Ncopy\langle lm \rangle \,|\, N\langle l \rangle \rightarrow^* N\langle m \rangle$$

Da diese Bedingung eine Aussage über rekursiv definierte Datenstrukturen trifft, benötigen wir Induktion für den Beweis, strukturelle Induktion über N, um genau zu sein:

$N = Zero$:

$Ncopy\langle lm\rangle \mid N\langle l\rangle$

$= Numbercases(Zero\langle m\rangle, ...)\langle l\rangle \mid Zero\langle l\rangle$

$\rightarrow^* Zero\langle m\rangle$ (Korrektheit von *Numbercases*)

$= N\langle m\rangle$

$N = Succ(M)$:

$Ncopy\langle lm\rangle \mid N\langle l\rangle$

$= Numbercases(..., F) \mid Succ(M)\langle l\rangle$

$\rightarrow^* \text{new } l'm'(m(n\,s).s\langle m'\rangle \mid Ncopy\langle l'm'\rangle \mid M\langle l'\rangle)$ (Korrektheit von Numbercases)

$= \text{new } l'm'(m(n\,s).s\langle m'\rangle \mid M\langle m'\rangle)$ (Induktionsannahme)

$\equiv \text{new } m'(m(n\,s).s\langle m'\rangle \mid M\langle m'\rangle)$

$= Succ(M)\langle m\rangle = N\langle m\rangle$ (Definition *Succ*)

mit $F := (l').\text{new } m'(m(n\,s).s\langle m'\rangle \mid Ncopy\langle l'm'\rangle)$

Nun können wir Addition definieren:

$$\begin{aligned} plus(klm) := \text{case } k \text{ of} \\ Zero \quad &\Rightarrow Ncopy\langle lm\rangle \\ Succ(k') &\Rightarrow \text{new } m'(m(n\,s).s\langle m'\rangle \mid plus(k'l\,m')) \end{aligned}$$

Für die Korrektheit benötigen wir einen semantischen Operator „+“, der gerade die Addition zweier Zahlen darstellen soll. Offensichtlich hat + folgende Eigenschaften:

$$\begin{aligned} Zero + N &= N \\ Succ(N + M) &= Succ(N) + M \end{aligned} \quad (*)$$

plus muss somit folgende Bedingung erfüllen:

$$plus(klm) \mid K\langle k\rangle \mid L\langle l\rangle \rightarrow^* (K + L)\langle m\rangle$$

Beweisen kann man dies ebenfalls mit struktureller Induktion, hier über K, man benötigt ausserdem noch (*). Der Beweis ist analog zu dem Korrektheitsbeweis von *Ncopy*.

Mithilfe der Zahldarstellung aus diesem Abschnitt folgt die Darstellung für Listen im nun folgenden Abschnitt 2.2.2 nahezu sofort; sämtliche Operatoren, Korrektheitsbedingungen und Beweise lassen sich vollständig übersetzen auf Listen; die einzige Veränderung ist eine Erweiterung der Knoten, die jetzt noch einen Verweis auf ein Listenelement tragen.

2.2.2 Listen

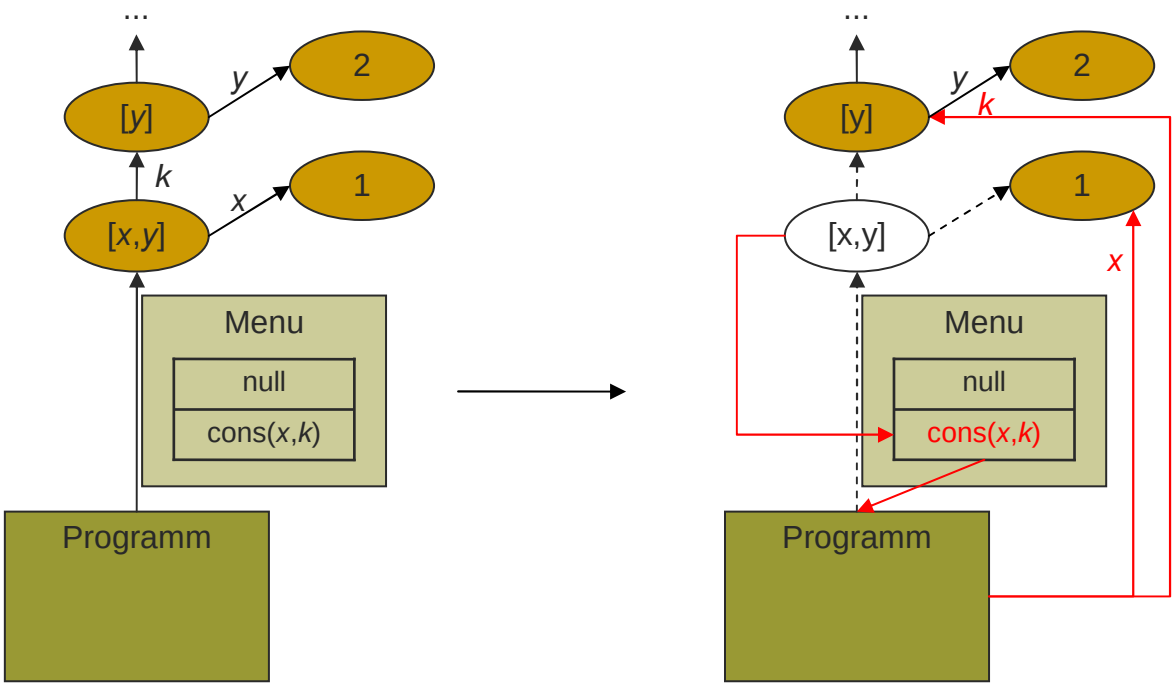

Abbildung 3: Darstellung von Listen

Wie schon am Ende von 2.2.1 erwähnt, ist der Sprung von unserer Zahldarstellung in eine Listendarstellung relativ klein. Abbildung 3 veranschaulicht die Veränderung: Jeder Knoten kennt jetzt noch einen Verweis x auf ein Listenelement. Im Fall einer Rechnung mit der Liste sendet der Knoten zusätzlich zu einem Zeiger auf die Restliste k noch diesen Verweis mit. Die Kodierung im π-Kalkül ändert sich dementsprechend:

$$Nil := (k).k(n\,c).\overline{n}$$
$$Cons(V,L) := (k).new\,v\,l\,(k(n\,c).\overline{c} < v\,l > | V < v > | L < l >)$$
$$Listcases(P,F)(k) := (new\,n\,c)\overline{k}\,\langle n\,c\rangle.(n.P + c.F)$$

Der Korrektheitsbeweis von $Listcases$ ist analog zu dem von $Numbercases$.
Sämtliche bisher eingeführten Operatoren für Zahlen lassen sich analog auf Listen übertragen, man muss lediglich den Wertzeiger v hinzufügen.

2.3 unsichere Operatoren

In 2.1/2.2 wurden verschiedene Datenstrukturen und Operationen darauf eingeführt. Die Operatoren arbeiteten dabei alle auf Kanälen, „Lokationen“ sozusagen. Dabei gibt es jedoch ein Problem:
Es spricht nichts dagegen, an einer dieser „Lokationen“ verschiedene Daten abzulegen, wie das folgende Beispiel veranschaulicht (Lokation ist daher auch kein guter Name dafür):

$$P := Ncopy\,\langle lm\rangle \,|\, 3\langle l\rangle \,|\, 4\langle l\rangle$$

P kann indeterministisch zwei verschiedene Reaktionen durchführen:

$$(1)\, P \rightarrow^{*} 3\langle m \rangle \mid 4\langle l \rangle$$
$$(2)\, P \rightarrow^{*} 3\langle l \rangle \mid 4\langle m \rangle$$

Dieses Problem lässt sich mit Restriktion (new) umgehen:

$$NMove(N) := (m).\mathrm{new}\, l\, (N\langle l \rangle \mid Ncopy\langle lm \rangle)$$

$NMove$ ist ein abgeleiteter Operator auf Zahlen; Durch die Restriktion des konkreten Kanals l sind die Reaktionen aus dem obigen Beispiel nicht mehr möglich. Auf diese Weise kann man sämtliche selbst definierten Operatoren zunächst auf der unsicheren Kanal-ebene definieren, um sie dann, nachdem Korrektheit gezeigt wurde, auf die sichere „abstrakte“ Ebene hochzuziehen.

2.4 persistente und veränderbare Daten

2.4.1 persistente Daten

Die oben eingeführten Repräsentationen von Datenstrukturen wiesen ein relevantes Defizit auf: Sie sind „flüchtig“, sie verschwinden nach einer Berechnung. Abbildung 2 und 3 veranschaulichen diesen Effekt anhand von Zahlen beziehungsweise Listen: Durch Reaktion des Programms mit der Datenstruktur wird diese zerstört.Um unsere Datenstrukturen *persistent* zu machen, benötigen wir, wie schon in 2.1 erwähnt, Replikation (Notation: persistente Daten und Operatoren werden mit einem * gekennzeichnet). Bei den Wahrheitswerten ändert sich dadurch nur wenig:

$$*True(k) := !k(t\ f).\overline{t}$$
$$*False(k) := !k(t\ f).\overline{f}$$

Um Zahlen derart anzupassen, genügt es, die Zahlknoten zu replizieren:

$$*Zero := (k).!k(n\ s).\overline{n}$$
$$*Succ(N) := (k).\mathrm{new}\, l\, (!k(n\ s).\overline{s}\langle l \rangle \mid N\langle l \rangle)$$

Somit müssen aber auch sämtliche Operationen auf Zahlen angepasst werden, die selbst wieder eine Zahl erzeugen:

$$*decr(km) := \mathrm{case}\, k\ \mathrm{of}$$
$$Zero \Rightarrow *Zero\langle m \rangle$$
$$Succ(k') \Rightarrow !m(n\ s).k'\langle n\ s \rangle$$
$$*Ncopy(lm) := \mathrm{case}\, l\ \mathrm{of}$$
$$Zero \Rightarrow *Zero\langle m \rangle$$
$$Succ(l') \Rightarrow \mathrm{new}\, m'(!m(n\ s).s\langle m' \rangle \mid *Ncopy\langle l'm' \rangle)$$

$$
\begin{aligned}
*plus(klm) := \text{case } k \text{ of} \\
&Zero \quad \Rightarrow *Ncopy\langle lm\rangle \\
&Succ(k') \Rightarrow \text{new } m'(!m(n\,s).s\langle m'\rangle \mid *plus(k'\,l\,m'))
\end{aligned}
$$

Bei *Numbercases* ändert sich also nichts. Die Änderungen bei der Repräsentation von Listen sind analog. Nun sind wir in der Lage, mit Daten zu rechnen, und wir können sie dank Replikation beliebig oft wieder verwenden.

2.4.2 persistente und veränderbare Daten

Die Einführung von Persistenz wirft die Frage nach persistenten Datenstrukturen auf, die auch veränderbar sind. Dabei wollen wir einfache Speicherzellen darstellen, die folgende Eigenschaften besitzen:

1. Die Zelle kann leer oder mit einem Wert gefüllt sein
2. Man kann die Zelle auf diesen Zustand hin testen
3. Die Zelle kann geleert oder mit einem neuen Wert gefüllt werden

Punkte 1 und 2 kann man mit Hilfe des in 2.1 vorgestellten Darstellungsmodell bereits realisieren:

$$
\begin{aligned}
Nullref(r) &:= r(n\,c).\overline{n}.Nullref\langle r\rangle \\
Ref(rv) &:= r(n\,c).\overline{c}\langle v\rangle.Ref\langle rv\rangle
\end{aligned}
$$

Für Punkte 3 müssen wir unser Modell geringfügig ändern: Bisher wurde ein *Menu* an die Datenstruktur gesendet, die daraufhin eine *Menuoption* selektiert hat. Jetzt wird dieses Protokoll erweitert: Das Programm darf auch selektieren!

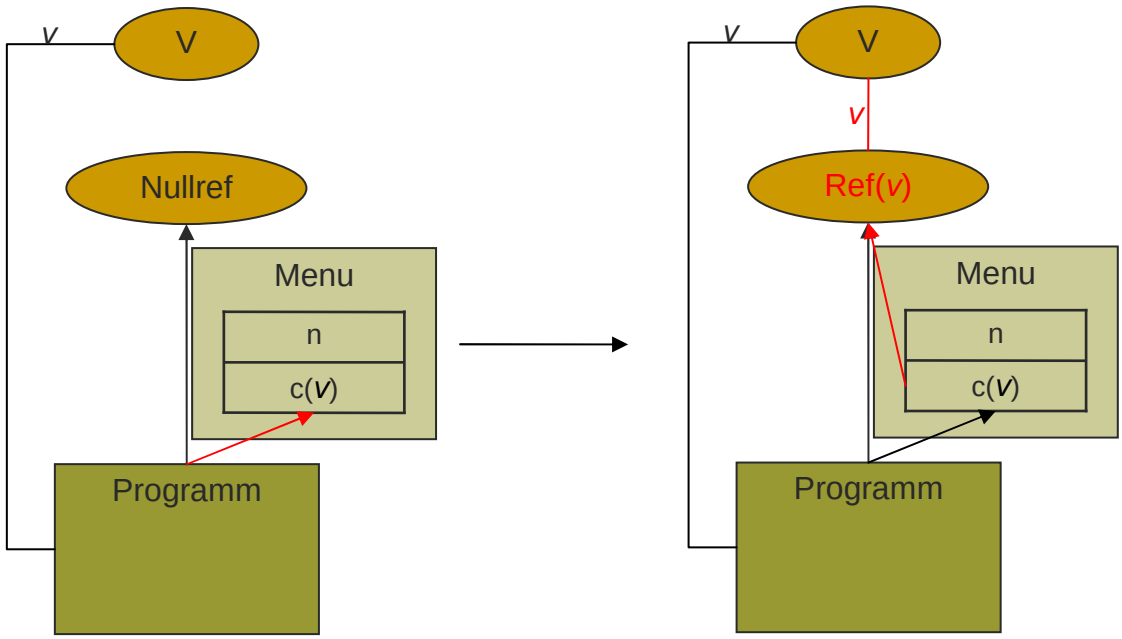

Abbildung 4 – Referenzen

Abbildung 4 veranschaulicht eine mögliche Zuweisung: Einer zunächst leeren Zelle wird ein Menu gesendet; Außerdem selektiert das *Programm* den in Zukunft gewünschten Zustand der Zelle, sie soll den Verweis *v* auf einen Wert *V* speichern. Die Zelle reagiert entsprechend und wechselt zum gefüllten Zustand Ref(*v*).
Der entsprechende Code sieht dann so aus:

$$Nullref(r) := r(n\,c).(\overline{n}.Nullref\langle r\rangle + c(v').Ref\langle rv'\rangle + n.Nullref\langle r\rangle)$$
$$Ref(rv) := r(n\,c).(\overline{c}\langle v\rangle.Ref\langle rv\rangle + c(v').Ref\langle rv'\rangle + n.Nullref\langle r\rangle)$$

Überstrichene Kanäle n und c bei der Definition von $NullreflRef$ dienen dem Zustandstest, sie entsprechen der schon bekannten *Menuselektion*. Die nicht überstrichenen Versionen erlauben einem reagierenden Programm, den Zustand der Zelle zu ändern. 2.3 entsprechend kann man eine (sichere) Speicherzelle wie folgt definieren:

$$Store(V) := (r).\mathrm{new}\,v\,(Ref\langle rv\rangle \mid V\langle v\rangle)$$

Jetzt fehlen nur noch die eigentlichen Lese- und Schreiboperationen auf den Speicherzellen:

$$Refcases(P,F) := (r).(\mathrm{new}\,n\,c)\overline{r}\langle nc\rangle.(n.P + c.F)$$
$$Nullify(P) := (r).(\mathrm{new}\,n\,c)\overline{r}\langle nc\rangle.\overline{n}.P$$
$$Assign(P) := (rv).(\mathrm{new}\,n\,c).\overline{r}\langle nc\rangle.\overline{c}\langle v\rangle.P$$

Die Schreiboperationen $Nullify$ und $Assign$ erhalten als Argument noch einen Prozess P, der eine *Continuation* darstellt, also den Code, der sequentiell **nach** Auswertung der Schreiboperation ausgeführt werden soll.
Korrektheit dieser Operationen lässt sich wieder durch einfaches Nachrechnen der Reaktionen zeigen. Die Korrektheitsbedingungen sind recht zahlreich, jedoch nicht sehr interessant, hier nur ein kleiner Ausschnitt:

$$Refcases(P,F)\langle r\rangle \mid Ref\langle rv\rangle \rightarrow^{*} F\langle rv\rangle \mid Ref\langle rv\rangle$$
$$Assign(P)\langle rv'\rangle \mid Nullref\langle r\rangle \rightarrow^{*} P \mid Ref\langle rv'\rangle$$
$$Nullify(P)\langle r\rangle \mid Ref\langle rv\rangle \rightarrow^{*} P \mid Nullref\langle r\rangle$$

3. Zusammenfassung

Daten können im π-Kalkül durch das *Menuprotokoll* dargestellt werden:

Eine Datenstruktur empfängt von dem berechnenden Programm ein *Menu* und selektiert eine *Menuoption*, eventuell unter Angabe einer *Spezifikation*.

Die *Spezifikation* kann hierbei ein Verweis auf eine Vorgängerzahl, wie in 2.2.1, oder auf einen gespeicherten Wert, wie in 2.4, sein. *Persistenz* der erzeugten Daten kann unter Verwendung von *Replikation* erreicht werden. Um persistente und veränder-

bare Datenstrukturen zu realisieren, muss dieses Protokoll geringfügig erweitert werden:

Eine Datenstruktur empfängt von dem berechnenden Programm ein *Menu*;
Die Struktur selbst oder das Programm selektieren daraufhin eine *Menuoption*,
eventuell unter Angabe einer *Spezifikation*.

Damit kein ungewollter Eingriff in eine Berechnung erfolgt, müssen Operatoren auf diesen Datenstrukturen mit Hilfe der Restriktion new „gesichert“ werden(vgl. 2.3).
Die Korrektheit eingeführter Operatoren lässt sich *meist* durch einfaches Nachrechnen der Reaktionen zeigen; Für manche Beweise benötigt man jedoch einen schwächeren Äquivalenzbegriff auf Prozessen, der interne Reaktionen zulässt (siehe [1], Kapitel 13).
Jetzt sind wir also in der Lage, im π-Kalkül einfache (nebenläufige) Programme zu schreiben. Mit der vorgestellten Repräsentation fällt es leicht, weitere Datenstrukturen wie Paare oder Bäume einzuführen.

Bibliographie

[1] Robin Milner, *Communicating and Mobile Systems: the Pi-Calculus*, Cambridge University Press, 1999