Mobile Ambients

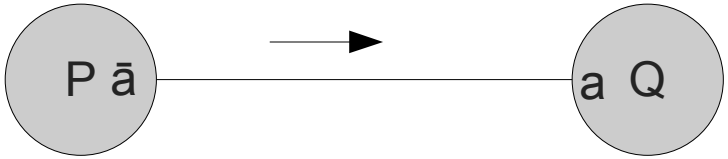
05.06.2007

Formalismen

- Man kann auch Agenten formal behandeln
- Arbeiten von Luca Cardelli: Mobile Ambients
  - baut auf dem π-Kalkül auf
  - baut seinerseits auf dem λ-Kalkül auf

Prozesskalkül

- π-Kalkül als Erweiterung des Lambda-Kalküls
- Komponenten
  - Prozesse
  - Kanäle, Ports
  - Namen



P und Q sind mit einem gerichteten Kanal a verbunden

P ≡ ā5.P'
Q ≡ a(x).Q'

P gibt „5“ über Port ā aus und verhält sich dann wie P'.

Q wartet auf einen Wert (Namen) auf dem Port a, setzt x auf diesen Wert und verhält sich dann wie Q' (wobei jedes x in Q' den neuen Wert angenommen hat)

Sinn

- Beweisbare Aussagen über das Verhalten konkurrierender Prozesse (→Agenten)
- Verteilte Berechnungen modellieren
- Formalismus:

P + Q:	P oder Q wird ausgeführt (auch mit Summenzeichen geschrieben)
0:	Leerer Prozess (tut nichts)
P | Q:	P und Q werden parallel ausgeführt
$\bar{a}x.P$:	Gib x über Port ā aus und mache als P weiter
a(x).P:	Nimm einen Namen z über Port a an und mache als P{z/x} weiter (z für x) x ist durch a(x) eine „gebundene Variable“ von P
$\tau.P$:	Keine Interaktion (Stille), dann wie P
(a)P:	Restriktion: keine Aktionen über a oder ā (nach außen)
[a=b]P:	Wenn a=b, dann wie P, sonst wie **0**
$A(y_1,...,y_n)$:	Agent

Agent

- $A(x_1,...,x_n)$ ist ein spezieller Prozess P mit
  - x_1, ..., x_n: einzige ungebundene Variablen (=freie Variablen) in P
- Dann verhält sich $A(y_1, ... , y_n)$ wie $P\{y_1/x_1, ..., y_n/x_n\}$
  - d.h. alle „formalen“ Parameter werden durch die „aktuellen“ Parameter belegt

Einfache Beispiele

$\bar{u}a.P' \mid u(z).Q' \mid R \xrightarrow{\tau} P' \mid Q'\{a/z\} \mid R$

Übergabe eines Kanals an einen anderen Prozess

Ambient-Kalkül

- π-Kalkül repräsentiert Mobilität von Software und Hardware nicht angemessen
  - lediglich Übermittlung von Namen zwischen parallelen Prozessen
- Ambient-Kalkül
  - Parallele Prozesse wie π-Kalkül
  - außerdem Umgebungen, die betreten und verlassen werden können: Ambients

Ambient-Kalkül

- Umgebungen (Ambients) können verschiedene Lokationen repräsentieren
- Prozesse können ihre Umgebung verlassen oder eine Umgebung betreten
- Sicherheit ist repräsentiert als Steuerung des Zugangs zu Umgebungen
  - keine kryptografischen Verfahren im Kalkül
  - keine Zugangssteuerungslisten (ACL)
- Kalkül wertet Interaktionen zwischen Prozessen in gemeinsamen Umgebungen aus

Primitive

n	Namen
P,Q ::=	Prozesse
$(\nu n)P$	Restriktion und neuer Name n in P („ny-n“)
0	Inaktivität
P | Q	Komposition (Parallelität)
!P	Replikation (!P := P | !P)
n[P]	Umgebung (Ambient) n
M.P	Aktion
M ::=	Berechtigungen
in n	darf n betreten
out n	darf n verlassen
open n	darf n öffnen

Erklärungen

Freie Namen (fn)

$fn(\mathbf{0}) := \varnothing$

$fn(n[P]) := \{n\} \cup fn(P)$

$fn(P \mid Q) := fn(P) \cup fn(Q)$

$fn(\text{in } n) := \{n\}$

$fn(\text{out } n) := \{n\}$

$fn(\text{open } n) := \{n\}$

$fn(M.P) := fn(M) \cup fn(P)$

$fn(!P) := fn(P)$

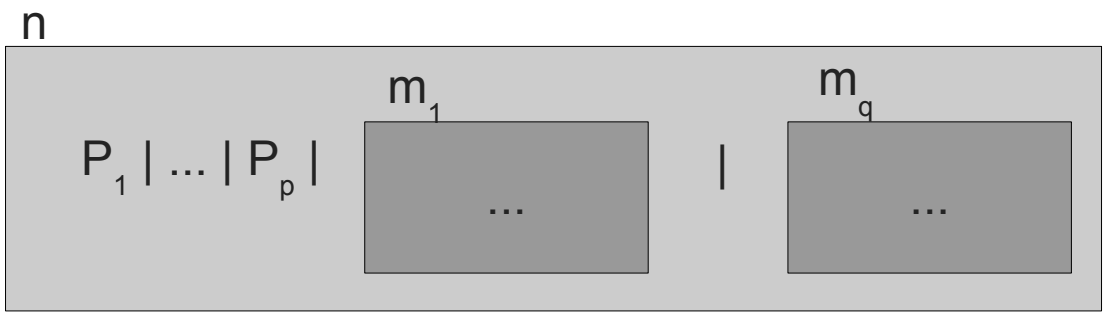
$fn((\nu n)P) := fn(P) \setminus \{n\}$

Konventionen

$P\{n \leftarrow m\}$	ersetze alle n durch m in P
$M\{n \leftarrow m\}$	ersetze alle n durch m in M
(...)	Klammerung
$(\nu n)P \mid Q$	$((\nu n)P) \mid Q$
$!P \mid Q$	$(!P) \mid Q$
$M.P \mid Q$	$(M.P) \mid Q$
$(\nu n_1 \ldots n_m)P$	$:= (\nu n_1)\ldots(\nu n_m)P$
$n[\,]$	$:= n[\mathbf{0}]$
M	$:= M.\mathbf{0}$

Ambients

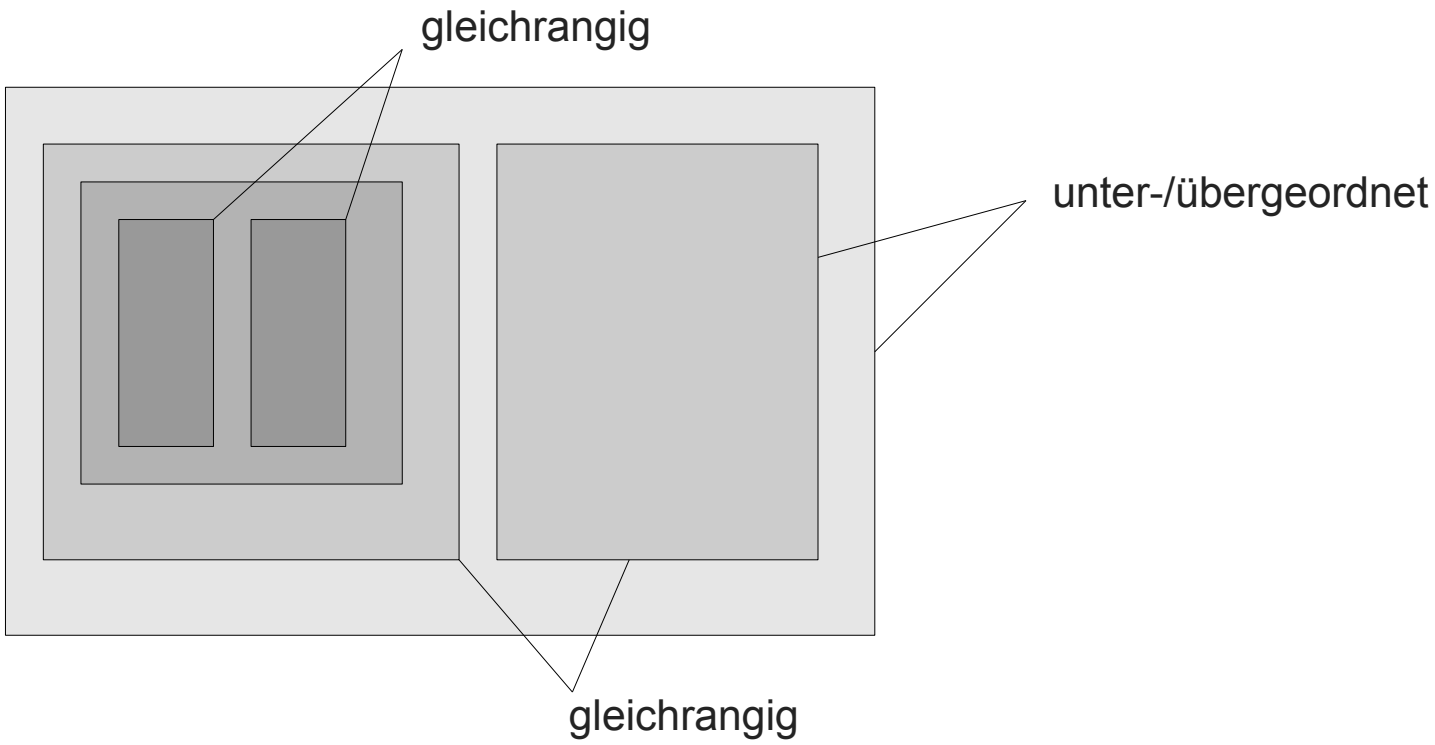
- n[P]
  - Umgebung hat den Namen n
  - In ihr läuft der Prozess P
  - P kann aus mehreren Teilen komponiert sein
  - Umgebungen stellen auch Prozesse dar, also Schachtelung möglich
  - Namen müssen nicht eindeutig sein (siehe z.B. !n[P])



Typische Umgebung

Hierarchie

- Umgebungen können beliebig geschachtelt werden
  - Hierarchie



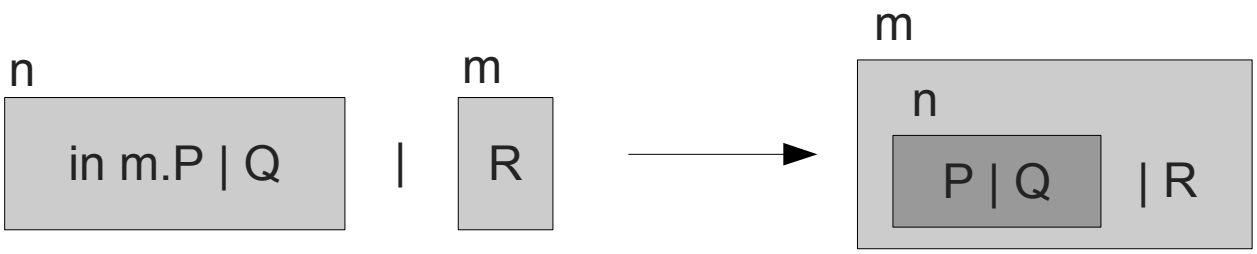
Aktionen

- Aktionen stehen zu Beginn eines Ausdrucks
  - Verschieben der Umgebungen in der Hierarchie
    - sensible Operationen, erfordern explizite Berechtigung
- Gesamtausdruck ist wieder ein Prozess
  - M.P ist ein Prozess
  - P ist der Prozess, der nach der Ausübung der durch M reglementierten Aktion in M.P verbleibt (→Abarbeitung von links nach rechts)
- Reduktion eines Ausdrucks
  - Abarbeiten der Aktionen
  - zeigt eine möglichen Verarbeitungssequenz

Ambients betreten

- Berechtigung „in“

in m.P : Die Umgebung n von Prozess P betritt die Umgebung m, welche gleichrangig neben n steht



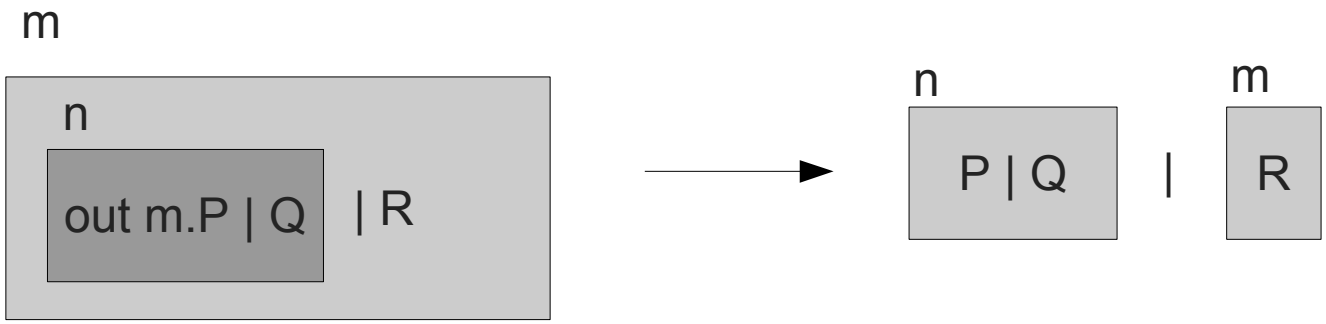
Formal: **n[in m.P | Q] | m[R] → m[n[P | Q] | R]**

Gibt es kein m, so wird gewartet; gibt es mehrere m, so wird ein beliebiges genommen

Ambients verlassen

- Berechtigung „out“

  out m.P : Die Umgebung n von Prozess out n.P verlässt die Umgebung m, und steht dann gleichrangig neben n



Formal: **m[n[out m.P | Q] | R] → n[P | Q] | m[R]**

Gibt es keine Elternumgebung m, so wird gewartet.

Ambients auflösen

- Berechtigung „open“

  open m.P : Eine Umgebung m, die gleichrangig neben open m.P steht, wird aufgelöst und das Innere gleichrangig angeordnet

Formal: **open m.P | m[Q] → P | Q**

Gibt es keine gleichrangige Umgebung m, so wird gewartet. Gibt es mehrere, wird irgendeine genommen.

Ausführung dieser Operationen

- Sensible Operationen
  - in m, out m, open m sind von m erteilte Berechtigungen
  - Aus in m, out m und open m kann der Name m nicht erschlossen werden
    - nur symbolische Bezeichnung, kein tatsächlicher Text!
    - Repräsentiert ein Sicherheitsmerkmal

Anwendung des Kalküls

- Fragen
  - Sind zwei gegebene Beschreibungen einander äquivalent?
  - Verhalten sich zwei Prozesse in bestimmten Umgebungen gleich?
  - Praktisch: Wie rechnet man das Kalkül?
    - Umformungsregeln

Strukturelle Kongruenz

- „≡“: Äquivalenz modulo syntaktischer Umordnung

$P \equiv P$
$P \equiv Q \Rightarrow Q \equiv P$
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$

$P \equiv Q \Rightarrow (\nu n)P \equiv (\nu n)Q$
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$
$P \equiv Q \Rightarrow !P \equiv !Q$
$P \equiv Q \Rightarrow n[P] \equiv n[Q]$
$P \equiv Q \Rightarrow M.P \equiv M.Q$

$P \mid Q \equiv Q \mid P$
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
$!P \equiv P \mid !P$
$(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$
$(\nu n)(P|Q) \equiv P \mid (\nu n)Q$, wenn $n \notin fn(P)$
$(\nu n)(m[P]) \equiv m[(\nu n)P]$, wenn $m \neq n$
$P \mid \mathbf{0} \equiv P$
$(\nu n)\mathbf{0} \equiv \mathbf{0}$
$!\mathbf{0} \equiv \mathbf{0}$

Gebundene Namen dürfen stets (konsistent) ersetzt werden

$(\nu n)P = (\nu m)P\{n \leftarrow m\}$, wenn $m \notin fn(P)$

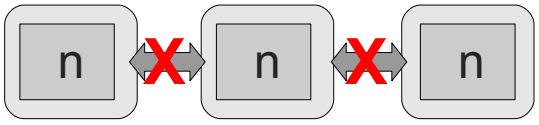
Achtung: Hier steht „=“ als Ausdruck der Identität, nicht nur als Kongruenz

Strukturelle Kongruenz

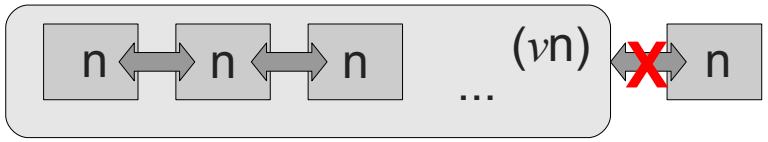
- Auswirkung
  - Man darf nun die Terme umsortieren und die miteinander reagierenden Teile optisch zusammenbringen
- Man beachte

$!(\nu n)P \not\equiv (\nu n)!P$ (Replikation erzeugt neue Namen)

$n[P] \mid n[Q] \not\equiv n[P \mid Q]$ (Verschiedene Umgebungen können gleiche Namen haben)



$!(\nu n)P$



$(\nu n)!P$

Reduktionsregeln

- Reduktion „→“

$$
\begin{array}{lcl}
n[\text{in } m.P \mid Q] \mid m[R] & \rightarrow & m[n[P \mid Q] \mid R] \\
m[n[\text{out } m.P \mid Q\,] \mid R\,] & \rightarrow & n[P \mid Q] \mid m[R] \\
\text{open } n.P \mid n[Q] & \rightarrow & P \mid Q
\end{array}
$$

$$
\begin{array}{lcl}
P \rightarrow Q & \Rightarrow & (\nu n)P \rightarrow (\nu n)Q \\
P \rightarrow Q & \Rightarrow & n[P] \rightarrow n[Q] \\
P \rightarrow Q & \Rightarrow & P \mid R \rightarrow Q \mid R
\end{array}
$$

$$
P' \equiv P,\ Q' \equiv Q,\ P \rightarrow Q \ \Rightarrow\ P' \rightarrow Q'
$$

$\rightarrow^*$ ist die reflexiv-transitive Hülle von $\rightarrow$

Kontextuelle Äquivalenz

- Allgemein: Wann sind Prozesse äquivalent?
  - Wenn sich sie in derselben Umgebung gleich verhalten (zumindest bezüglich dessen, was man von außen wahrnehmen kann)
- Ambients: Wann sind Prozesse äquivalent?
  - Der Prozess reduziert sich schrittweise
  - Der Kontext kann dafür sorgen, dass die Reduktion sich fortsetzt oder dass weitere Bestandteile übrig bleiben
  - Man betrachte die Ambients auf höchster Ebene
  - Gleiches Verhalten: die gleichen Ambients werden dort sichtbar

Kontextuelle Äquivalenz

- Definition
  - P „zeigt eine Umgebung n“ (P↓n), wenn der Prozess P eine Umgebung n auf oberster Hierarchieebene aufweist

    $P\downarrow n \quad := \quad P \equiv (\nu\, m_1 \ldots m_i)\,(n[P'] \mid P'') \qquad \text{wobei } n \notin \{m_1, \ldots, m_i\}$

  - P zeigt *schließlich* eine Umgebung n (P⇓n), wenn gilt:
    $P \rightarrow^* Q$ und $Q\downarrow n$

Kontextuelle Äquivalenz

- Sei C(x) ein Prozess mit 0 oder mehr Freistellen (x), C(P) entsprechend der Prozess nach Einsetzen von P in x
  - in P freie Namen können dabei gebunden werden
- **P ≃ Q** (P kontextuell äquivalent zu Q) : ⇔
  ∀n,C(): C(P)⇓n ⇔ C(Q)⇓n
- Kurzschreibweise: P $\rightarrow^* \simeq$ Q,
  wenn es R gibt, sodass P $\rightarrow^*$ R und R ≃ Q
- Satz: $(\nu n)n[P] \simeq \mathbf{0}$, wenn $n \notin fn(P)$
  - Damit lassen sich inaktive Ambients entsorgen (Garbage collect)

Beispiele im Ambient-Kalkül

- Repräsentation einer Sperre

  acquire n.P := open n.P
  release n.P := n[ ] | P

  Damit ist ein „Handshake“ repräsentierbar:

  acquire n.release m.P | release n.acquire m.Q $\rightarrow^*$ P | Q

Authentifikation mobiler Agenten

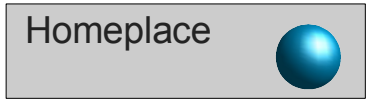
- Prozesse auf oberster Ebene eines Ambients können als „privilegiert“ angesehen werden
- Szenario
  - Privilegierter Prozess wechselt seine Umgebung und möchte wieder in dieselbe Umgebung auf gleicher Ebene zurückkehren

```
Homeplace[
    (νpass)(open pass |
       Agent [
          out Homeplace.in Homeplace.pass[out Agent.open Agent.P]
       ]
    )
]
```

„pass“ als geheimes Kennwort, das nur die Stelle und der Agent kennt
P = Agentenverhalten

Authentifikation mobiler Agenten

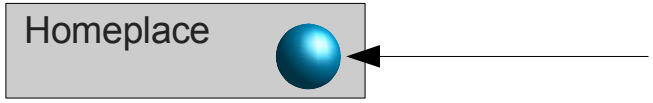
Homeplace[(νpass)(open pass | Agent [out Homeplace.in Homeplace.pass[out Agent.open Agent.P]])]
≡ (νpass)Homeplace[open pass | Agent [out Homeplace.in Homeplace.pass[out Agent.open Agent.P]]]



→ (νpass)(Homeplace[open pass] | Agent [in Homeplace.pass[out Agent.open Agent.P]])

Agent und Homeplace sind gleichrangig geworden

→ (νpass)(Homeplace[open pass | Agent [pass[out Agent.open Agent.P]])



Agent ist zurückgekehrt

→ (νpass)(Homeplace[open pass | pass[open Agent.P] | Agent [ ])
→ (νpass)(Homeplace[**0** | open Agent.P | Agent [ ])
→ (νpass)(Homeplace[**0** | P | **0**])

≡ Homeplace[P]

Firewalls im Ambient-Kalkül

- Firewall behält ihr Geheimnis (w)
- Agent durchkreuzt die Firewall mittels dreier Kennwörter k, k', k''
- Firewall schleust den Agenten durch

Firewall := (νw) w[k[out w.in k'.in w] | open k'.open k''.P]
Agent := k'[open k.k''[Q]]

Name w bleibt in der Firewall

Agent | Firewall $\rightarrow^*$ (νw)w[Q | P]

Das Agentenverhalten Q ist nun im geschützten Bereich w, ohne dass der Agent w kennen musste.
Damit ist beweisbar, dass der Agent k, k', k'' gekannt hat

Firewalls

- Man kann zeigen:
  $(\nu\ k\ k'\ k'')(\text{Agent} \mid \text{Firewall}) \simeq (\nu w)w[\ Q \mid P\ ]$
- Folgerung
  - Da kontextuelle Äquivalenz definitionsgemäß in allen Kontexten gilt, gilt sie insbesondere in jenen, in denen Angreifer auftauchen!
  - Damit ist diese Firewall beweisbar sicher.
  - Voraussetzung: Der Angreifer kennt k, k', k'' nicht
    - ausgedrückt durch die Restriktion $(\nu\ k\ k'\ k'')$

Aktive und passive Mobilität

- Ambient-Kalkül sieht aktive (subjektive) Mobilität vor
  - Umgebung initiiert selbst die Migration: in und out
  - Beispiel: n[in m.P|Q] | m[R] → m[n[P|Q]|R]
    - n bewegt sich selbst in m hinein
- Alternativvorschlag: Passive (objektive) Mobilität
  - mv in m.P | m[R] → m[P|R]
  - m[mv out m.P | R] → P | m[R]
  - Einfachere Definition, jedoch nur auf passiven Umgebungen agierend
    - P macht keinen Fortschritt, nur „mv in/out m“ wird „verbraucht“

Aktive und passive Mobilität

- Problem: Umgebungen können „gefangen“ werden
  - Toplevel-Agenten einer Umgebung sorgen für die Bewegung
  - Bei subjektiver Mobilität kann man keinen „Bewegungsagenten“ unmittelbar injizieren (steckt immer in einer eigenen Umgebung!)
  - Bei objektiver Mobilität kann man es:

entrap m := $(\nu k)(k[\ ] \mid mv \text{ in } m.\text{in } k.\mathbf{0})$

Voraussetzung:
Man muss „mv in m“ kennen!

entrap m | m[P] $\rightarrow^*$ (νk) k[m[P]]

m[P] ist in k gefangen und kann es durch keine Operation mehr verlassen

→ Gegenargument, solche Bewegungsprimitive zuzulassen

Steuerung objektiver Mobilität

- Objektive Mobilität mit Zugangssteuerung
  - Realisierung ohne Hinzufügen von Primitiven

allow n := !open n
mv in n.P := (νk) k[in n.enter[out k.open k.P]]
mv out n.P := (νk) k[out n.exit[out k.open k.P]]

enter/exit:
spezielle Namen, nach Konvention gewählt

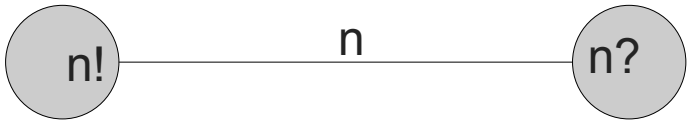
$n^{\downarrow}$[P] := n[P | allow enter] erlaubt mv in
$n^{\uparrow}$[P] := n[P] | allow exit erlaubt mv out
$n^{\uparrow\downarrow}$[P] := n[P | allow enter] | allow exit erlaubt mv in und out

- Statt enter/exit spezifischen „Schlüssel“ verwenden

$mv\ in_k\ n.P := k[in\ n.P]$ entsprechend „allow k“ verwenden
$mv\ out_k\ n.P := k[out\ n.P]$

Synchronisation auf benannten Kanälen

- $n^{\uparrow\downarrow}[]$ als Kanal interpretiert
- Zwei komplementäre Ports
  - n? : Eingangsport
  - n! : Ausgangsport



- n?.P (n!.P) synchronisiert auf n? (n!) und fährt als P fort
- Repräsentation im Ambient-Kalkül

```
n?.P    := mv in n.acquire rd.release wr.mv out n.P
n!.P    := mv in n.release rd.acquire wr.mv out n.P
```

Weitere Beispiele

- Auswahl
  - $n \Rightarrow P + m \Rightarrow Q$

$$:= (\nu\ p\ q\ r)\ (\ p[\text{in } n.\text{out } n.q[\text{out } p.\text{open } r.P]] \mid p[\text{in } m.\text{out } m.q[\text{out } p.\text{open } r.Q]] \mid \text{open } q \mid r[]\ )$$

$$(n \Rightarrow P + m \Rightarrow Q) \mid n[S] \rightarrow^* \simeq P \mid n[S]$$
$$(n \Rightarrow P + m \Rightarrow Q) \mid m[S] \rightarrow^* \simeq Q \mid m[S]$$

Weitere Beispiele

- Umbenennung einer Umgebung
  - n be m.P := m[out n.open n.P] | in m
- Feststellen, ob eine Umgebung vorhanden ist
  - see n.P := (ν r s)(r[in n.out n.r be s.P] | open s)
  - P wird nur aktiv, wenn parallel eine n-Umgebung vorliegt
  - n darf sich nicht wegbewegen
- Iteration
  - rec $(m_1)P_1$ ... $(m_p)P_p$ in Q := (ν m_1...m_p)(!open m_1.P_1 | ... | !open m_p.P_p | Q)
  - Sobald eine Umgebung m_i in Q gezeigt wird, wird diese Umgebung in Q durch das entsprechende P_i ersetzt
    rec $(m_1)P_1$ ... $(m_p)P_p$ in $m_i[] \rightarrow^*$ rec $(m_1)P_1$ ... $(m_p)P_p$ in P_i

Zahlen

- Auch Zahlen sind durch Umgebungen repräsentierbar
  - $\underline{0}$:= zero[]
    $\underline{i+1}$:= succ[open op | $\underline{i}$]
  - Idee: Natürliche Zahl als Kette ineinander geschachtelter succ-Umgebungen
- Operationen
  - ifzero P Q := zero $\Rightarrow$ P + succ $\Rightarrow$ Q
    Probe: $\underline{0}$ | ifzero P Q $\rightarrow^* \simeq \underline{0}$ | P
    $\underline{i+1}$ | ifzero P Q $\rightarrow^* \simeq \underline{i+1}$ | Q

Operatoren

- inc.P := ifzero (inczero.P) (incsucc.P)
- inczero.P := open zero.(1 | P)
- incsucc.P := (ν p q)(p[succ[open op]] | open q.open p.P |
  op[in succ.in p.in succ.(q[out succ.out succ.out p] | open op)])
- dec.P := (νp)(op[in succ.p[out succ]] | open p.open succ.P)
- Erklärung
  - incsucc erhöht jede Zahl ungleich 0 durch Einschachtelung in succ
  - dec vermindert die Schachtelungstiefe
- i | inc.P →*≃ i+1 | P
  i+1 | dec.P →*≃ i | P

Hiermit kann jede arithmetische Operation formell abgeleitet werden.

Weitere Fähigkeiten

- Ambient-Kalkül kann Turingmaschine repräsentieren
  - Band ist eine Schachtelung von Umgebungen
  - Binäre Zustände durch zwei Namen t (true) und f (false)
  - Bewegung = tiefer oder höher in der Schachtelung der Zellen auf dem Band
  - Zustandsmaschine definierbar durch Umgebungen
    - Namen wählbar, etwa „Zustand1“, „Zustand2“, „Zustand3“...
- Damit ist das Kalkül „Turing-vollständig“, kann also im Prinzip alles Berechenbare darstellen

Kommunikation

- Bislang nur Mobilität gezeigt
- Modellierung von Kommunikation
  - wichtig für Kompatibilitätsbeweise
  - wichtig zum Repräsentieren anderer Kalküle wie Lambda oder Pi
  - daraus möglicherweise formaler Beweis der Typverträglichkeit!
- Kommunikation sollte Berechtigungssystem nicht beeinflussen

Zusätzliche Primitive

P,Q := ...		bisherige Mobilitätsprimitive
	(x).P	Eingabeaktion
	⟨M⟩	Ausgabeaktion (asynchron)
M:= ...		bisherige Berechtigungen
	x	Variable
	n	Name
	ε	Null
	M.M'	Pfad

Freie Namen und freie Variablen

$fn(M[P]) := fn(M) \cup fn(P)$
$fn((x).P) := fn(P)$
$fn(\langle M \rangle) := fn(M)$
$fn(x) := \varnothing$
$fn(n) := \{n\}$
$fn(\varepsilon) := \varnothing$
$fn(M.M') := fn(M) \cup fn(M')$

$P\{x \leftarrow M\}$ Ersetzen jedes Vorkommens der freien Variablen x durch M in P

$M\{x \leftarrow M'\}$ Ersetzen jedes Vorkommens der freien Variablen x durch M' in M

(x).P | Q = ((x).P) | Q

$fv((\nu n)P) := fv(P)$
$fv(\mathbf{0}) := \varnothing$
$fv(P \mid Q) := fv(P) \cup fv(Q)$
$fv(!P) := fv(P)$
$fv(M[P]) := fv(M) \cup fv(P)$
$fv(M.P) := fv(M) \cup fv(P)$
$fv((x).P) := fv(P) \setminus \{x\}$
$fv(\langle M \rangle) := fv(M)$
$fv(x) := \{x\}$
$fv(n) := \varnothing$
$fv(\text{in } M) := fv(M)$
$fv(\text{out } M) := fv(M)$
$fv(\text{open } M) := fv(M)$
$fv(\varepsilon) := \varnothing$
$fv(M.M') := fv(M) \cup fv(M')$

Kommunizierte Daten

- Daten, die kommuniziert werden können:
  - Namen
  - Berechtigungen
- In der Regel sollten Namen nicht kommuniziert werden, sondern Berechtigungen
- Pfad = Kette von Berechtigungen
- Namen als Berechtigung: Recht, eine Umgebung dieses Namens zu erzeugen
- Zwei Bindungsarten: ν und (...) (Eingabe)
- Namen: m, n, p, q; Variablen: w, x, y, z

Kommunikation mit Umgebungen

- Eingabeoperation (bindet Variable)
- Ausgabeoperation (asynchron, wartet nicht auf Entgegennahme)
- Reduktion bei der Entgegennahme
- Vorsicht: Syntaktische Anomalie
  - ((x).x.P) | $\langle n \rangle \rightarrow$ n.P: was soll das sein?
  - Vermischung von Bewegungsberechtigungen und Namen im Kalkül über das Variablenkonzept
  - Geeignetes Typsystem kann solche Fehlzuweisungen verhindern

Semantik der Kommunikation

- Strukturelle Kongruenz: Zusätzliche Vereinbarungen

$P \equiv Q \Rightarrow M[P] \equiv M[Q]$
$P \equiv Q \Rightarrow (x).P \equiv (x).Q$
$\varepsilon.P \equiv P$
$(M.M').P \equiv M.M'.P$

$(x).P = (y)P\{x \leftarrow y\} \quad y \notin fv(P)$

Umbenennung: Identität

- Reduktion

$(x).P \mid \langle M \rangle \rightarrow P\{x \leftarrow M\}$

- Kontextuelle Äquivalenz

$P \simeq Q \quad \Leftrightarrow \quad \forall n, C() \text{ mit } fv(C(P)) = fv(C(Q)) = \emptyset: C(P) \Downarrow n \Leftrightarrow C(Q) \Downarrow n$

Beispiel

- Zellen
  - „cell c w“ speichert den Wert w (Berechtigung) an der Lokation c
  - Zelle gibt Inhalt nur einmal aus
    - daher muss get die Zelle mit dem altem Wert $\langle w/x \rangle$ „auffrischen“

cell c w := $c^{\uparrow\downarrow}[\langle w \rangle]$
get c (x).P := mv in c.(x).($\langle x \rangle$ | mv out c.P)
set c $\langle w \rangle$.P := mv in c.(x).($\langle w \rangle$ | mv out c.P)

get-and-set c (x) $\langle w \rangle$.P := mv in c.(x).($\langle w \rangle$ | mv out c.P)

atomische (untrennbare) Ausführung von get und set

Beispiele

- Datensätze

record r(f_1=v_1, ... f_n=v_n) := $r^{\uparrow\downarrow}$[cell f_1 v_1 | ... cell f_n v_n]

getr r f (x).P := mv in r.get f (x).mv out r.P
setr r f ⟨v⟩.P := mv in r.set f ⟨v⟩.mv out r.P

Datensätze können prinzipiell auch andere Datensätze als Werte tragen

Beispiele

- Pakete
  - Verschicken eines Prozesses an andere Lokationen

packet pkt := pkt[!(x).x | !open route]
route pkt with P to M := route[in pkt.⟨M⟩ | P]
forward pkt to M := route pkt with **0** to M

P darf natürlich nicht mit dem Routing interferieren
M kann mehrere Berechtigungen beinhalten (Pfad!)

Entfernte Kommunikation

- Kommunikation zwischen Umgebungen

$@M\langle a\rangle := io[M.\langle a\rangle]$ Entfernte Ausgabe

$@M(x)M^{-1}.P := (\nu n)\ (io[M.(x).n[M^{-1}.P]] \mid open\ n)$ Entfernte Eingabe

Zu beachten:
M^{-1} ist der Rückwärtspfad zu M
M erlaubt Eingabe/Ausgabe durch !open io

io ist ein Bote (Messenger)!

Man möchte vielleicht P im Messenger nicht mitschleppen:

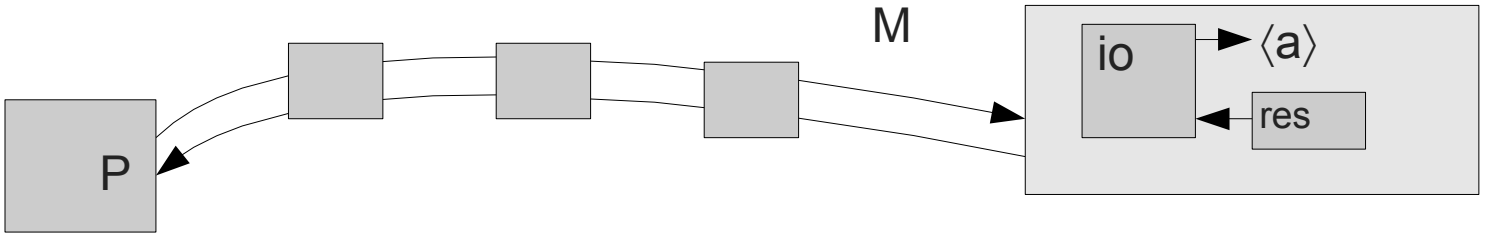
$@M(x)M^{-1}.P := (\nu n)\ (io[M.(x).n[M^{-1}.\langle x\rangle]] \mid open\ n) \mid (x).P$

Fernprozeduraufruf (RPC)

$@M\ arg\langle a\rangle\ res(x)\ M^{-1}.P :=$

$(\nu n)\ (io[M.(\langle a\rangle |\ open\ res.(x).n[M^{-1}.\langle x\rangle])] \mid open\ n) \mid (x).P$

res beinhaltet das Ergebnis, welches der entfernte Kontext auf die Übermittlung von ⟨a⟩ produziert.



Dies ist im Wesentlichen die Implementierung einer synchronen Kommunikation (RPC) über zwei asynchrone Kommunikationsvorgänge (⟨a⟩ und ⟨x⟩).

Polyadische, typisierte Kommunikation

- Bislang noch monadisch (nur ein Argument) und typlos
- Erweiterung der Definitionen

P,Q ::=	...	
	$(\nu n{:}W)P$	Restriktion
	$(n_1{:}W_1, ..., n_k{:}W_k).P$	Eingabe
	$\langle M_1, ..., M_k \rangle$	Asynchrone Ausgabe

$$fn((\nu n{:}W)P) := fn(P) \setminus \{n\}$$
$$fn((n_1{:}W_1, ..., n_k{:}W_k).P) := fn(P) \setminus \{n_1, ..., n_k\}$$
$$fn(\langle M_1, ..., M_k \rangle) := fn(M_1) \cup ... \cup fn(M_k)$$

Cardelli unterscheidet in [2] nicht zwischen Variablen und Namen und macht Namen durch Input prinzipiell ersetzbar.

Weitere Regeln

$P \equiv P$
$P \equiv Q \Rightarrow Q \equiv P$
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$

$P \equiv Q \Rightarrow (\nu n{:}T)P \equiv (\nu n{:}T)Q$
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$
$P \equiv Q \Rightarrow !P \equiv !Q$
$P \equiv Q \Rightarrow M[P] \equiv M[Q]$
$P \equiv Q \Rightarrow M.P \equiv M.Q$
$P \equiv Q \Rightarrow (n_1{:}T_1, \ldots, n_k{:}T_k).P \equiv (n_1{:}T_1, \ldots, n_k{:}T_k).Q$

$P \mid Q \equiv Q \mid P$
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
$!P \equiv P \mid !P$
$(\nu n{:}T)(\nu m{:}U)P \equiv (\nu m{:}U)(\nu n{:}T)P$, wenn $m \neq n$
$(\nu n{:}T)(P|Q) \equiv P \mid (\nu n{:}T)Q$, wenn $n \notin fn(P)$
$(\nu n{:}T)(m[P]) \equiv m[(\nu n{:}T)P]$, wenn $m \neq n$
$P \mid \mathbf{0} \equiv P$
$(\nu n{:}Amb[T])\mathbf{0} \equiv \mathbf{0}$
$!\mathbf{0} \equiv \mathbf{0}$
$\varepsilon.P \equiv P$
$(M.M').P \equiv M.M'.P$

Strukturelle Kongruenz

...
$(n_1{:}W_1, \ldots, n_k{:}W_k).P \mid \langle M_1, \ldots, M_k \rangle \rightarrow P\{n_1 \leftarrow M_1, \ldots, n_k \leftarrow M_k\}$
$P \rightarrow Q \Rightarrow (\nu n{:}W)P \rightarrow (\nu n{:}W)Q$

Reduktion

Typen

- Der Sender kann unabhängig von irgendeinem Empfänger beliebige Daten absetzen
- Definiere Nachrichtentypen, um den Austausch zu steuern

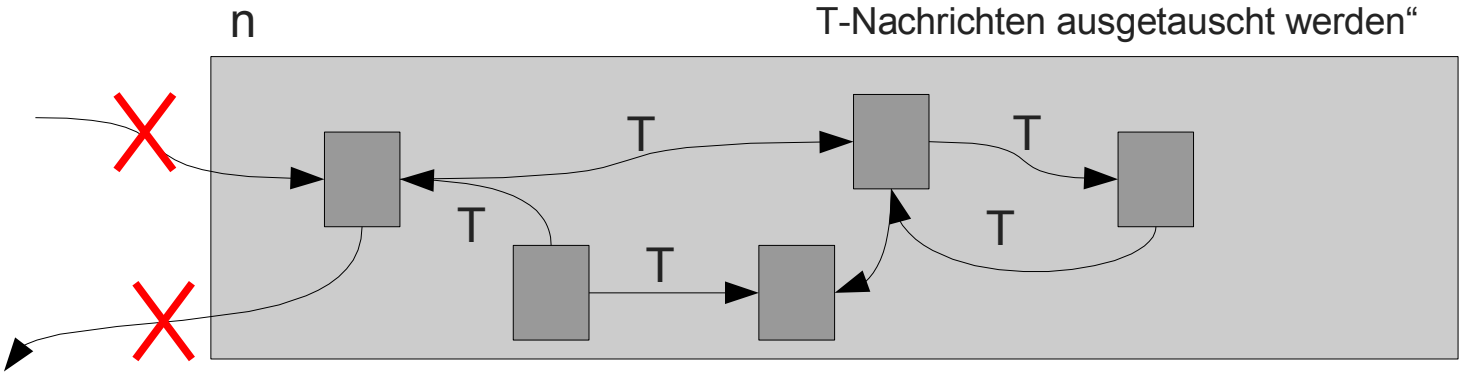
W::=		Nachrichtentypen
	Amb[T]	Ambientnamen, welche den Nachrichten-austausch des Typs T erlauben
	Cap[T]	Berechtigung, die zum Austausch von T-Nachrichten führt

T::=		Austauschtypen
	Shh	Kein Austausch
	$W_1 \times ... \times W_k$	Tupelaustausch

Typen

n:Amb[T]

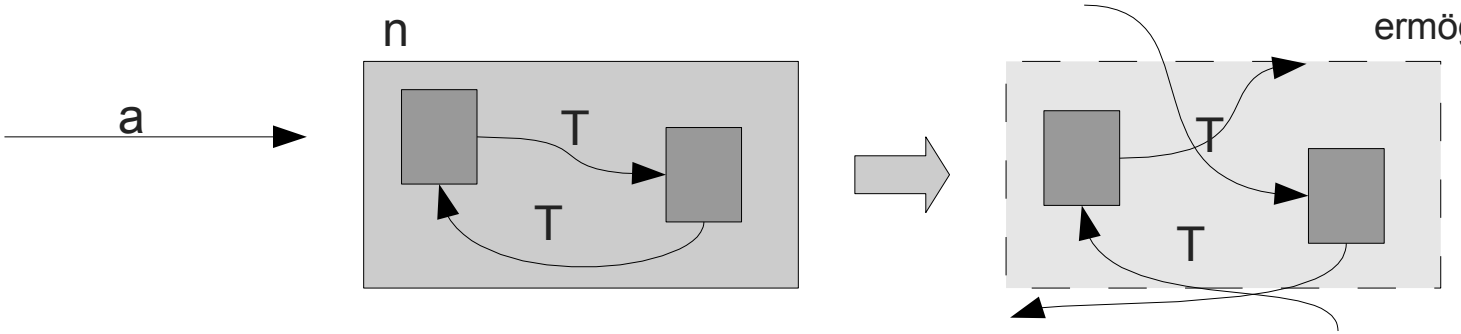
„n ist vom Typ Amb[T]“
„n ist der Name eines Ambients, in dem T-Nachrichten ausgetauscht werden“



Nachrichten
können Ambient-Grenzen nicht
überqueren

a:Cap[T]

a ist der Name einer Berechtigung, welche T-Nachrichtenaustausch ermöglicht



Typen

- **1** ist das leere Tupel (nur Synchronisation; k=0)
- Jeder Nachrichtentyp kann auch ein Austauschtyp sein (k=1)
- Beispiele
  - Amb[Shh]: eine ruhige Umgebung
  - Cap[Shh]: eine „harmlose“ Berechtigung (löst keine Nachrichten aus)
  - Amb[**1**]: Synchronisationsumgebung
  - Amb[Cap[Shh]]: eine Umgebung, die den Austausch von harmlosen Berechtigungen erlaubt
  - Cap[Amb[Shh]]: eine Berechtigung, die den Austausch von Namen stiller Umgebungen auslöst

Typisierung von Prozessen

- Typ eines Prozesses = Typ der ausgetauschten Nachrichten
  - $M:W \Rightarrow \langle M \rangle:W$
    - Ist M eine Nachricht vom (Nachrichten-)Typ W, dann ist $\langle M \rangle$ ein Ausgabeprozess des (Austausch-)Typs W
  - $M:W \Rightarrow (x:W).P:W$
    - Nachrichtenaustausch bedeutet auch die Annahme von Nachrichten, also ist (x:W).P vom Typ W

Typsystem für mobile Ambients

- Typregeln; kurz erläutert
  - Typausdruck hat die Form X:Y, liest sich als „X hat den Typ Y“
  - X kann eine Nachricht sein, dann ist Y ein Nachrichtentyp
  - X kann ein Prozess sein, dann ist Y ein Austauschtyp
  - Typumgebung (environment) sammelt gültige Typausdrücke
- Schreibweise

$$\frac{A\ B\ \ldots\ X}{Z}$$ Wenn A, B, ..., X gelten, dann möge auch Z gelten

$E \vdash \Diamond$	Gültige („wohlgeformte“) Umgebung
$E \vdash M:W$	M:W ist ein gültiger Nachrichtentypausdruck
$E \vdash P:T$	P:T ist ein gültiger Austauschtypausdruck

Typregeln

$$\frac{}{\varnothing \vdash \Diamond}$$

Die leere Typumgebung ist immer gültig

$$\frac{E \vdash \Diamond \quad n \notin dom(E)}{E, n:W \vdash \Diamond}$$

Die Typumgebung besteht aus Ausdrücken der Form n:W (wobei n nur einmal auftauchen darf)

$$\frac{E', n:W, E'' \vdash \Diamond}{E', n:W, E'' \vdash n:W}$$

Jeder Ausdruck in der Umgebung zählt als gültiger Typausdruck

$$\frac{E \vdash \Diamond}{E \vdash \varepsilon:Cap[T]}$$

Der Nullpfad kann eines beliebigen Austausch-typs sein (d.h. jeder Ausdruck dieser Form ist gültig).

$$\frac{E \vdash M:Cap[T] \quad E \vdash M':Cap[T]}{E \vdash M.M':Cap[T]}$$

Austauschtyp eines Pfads. Alle Bestandteile müssen desselben Typs sein.

Typregeln

$$\frac{E \vdash M:Amb[S]}{E \vdash in\ M:Cap[T]}$$

$$\frac{E \vdash M:Amb[S]}{E \vdash out\ M:Cap[T]}$$

$$\frac{E \vdash M:Amb[T]}{E \vdash open\ M:Cap[T]}$$

Die in-Berechtigung kann nicht bewirken, dass Nachrichten das Ambient verlassen oder betreten, also kann sie jedes Typs sein.

Die out-Berechtigung kann nicht bewirken, dass Nachrichten das Ambient verlassen oder betreten, also kann sie jedes Typs sein.

Die open-Berechtigung kann bewirken, dass Nachrichten das Ambient verlassen oder betreten, also muss sie den Austauschtyp der im Ambient zulässigen Nachrichten haben.

(Bei in und out bleiben Ambients erhalten und wechseln lediglich ihre Anordnung in der Hierarchie; open hingegen löst Ambients auf)

Typregeln

$$\frac{E \vdash M:Cap[T] \quad E \vdash P:T}{E \vdash M.P:T}$$

Eine Aktion M muss den Austauschtyp T haben, wenn sie auf einem Prozess des Typs T operiert. Das Resultat ist dann wieder vom Typ T.

$$\frac{E \vdash M:Amb[T] \quad E \vdash P:T}{E \vdash M[P]:S}$$

Ambients bilden nur dann gültige Ausdrücke, wenn der Nachrichtentyp dem Prozesstyp entspricht. Aber da Nachrichten das Ambient weder betreten noch verlassen, darf das gesamte Gebilde jeden beliebigen Typ aufweisen.

$$\frac{E, n:Amb[T] \vdash P:S}{E \vdash (\nu n:Amb[T])P:S}$$

Die Restriktion ändert den Typ des Prozesses nicht.

Typregeln

$$\frac{E \vdash \Diamond}{E \vdash \mathbf{0}:T}$$

Der inaktive Prozess kann jeden Typ haben (und kann daher nie inkompatibel sein)

$$\frac{E \vdash P:T \quad E \vdash Q:T}{E \vdash P \mid Q:T}$$

Man kann nur typgleiche Prozesse parallel laufen lassen; der Gesamttyp bleibt derselbe.

$$\frac{E \vdash P:T}{E \vdash !P:T}$$

Der Typ bleibt unter Replikation erhalten.

$$\frac{E, n_1:W_1, ..., n_k:W_k \vdash P:W_1 \times ... \times W_k}{E \vdash (n_1:W_1, ..., n_k:W_k).P:W_1 \times ... \times W_k}$$

Ein Prozess, der den Typ $W_1 \times ... \times W_k$ aufweist, muss bei seinen Eingabevariablen dieselben Typen in derselben Reihenfolge aufweisen. Der Typ bleibt gleich.

$$\frac{E \vdash M_1:W_1, ..., E \vdash M_k:W_k}{E \vdash \langle M_1, ..., M_k \rangle:W_1 \times ... \times W_k}$$

Der Ausgabeprozess $\langle M_1, ..., M_k \rangle$ weist als Austauschtyp das Tupel der Typen aller enthaltenen Elemente auf.

Typregeln

- Typregeln geben exakt an, welche Terme bezüglich des Typsystems gültig sind
  - Was aus den Regeln nicht ableitbar ist, ist ungültig!
- Beispiele für gültige Ausdrücke

$\varnothing \vdash !(\nu n{:}Amb[Shh])\langle n\rangle : Amb[Shh]$

Prozess, der Namen von stillen Ambients ausgibt

$\varnothing, n{:}Amb[T], m{:}Amb[S] \vdash in\ n.open\ m : Cap[S]$

Man beachte, dass wieder „in“ nichts am Typ des Pfads ändern kann, wohl aber „open“

Typsystem

- Korrektheit des Typsystems

  Wenn $E \vdash P:U$ und $P \rightarrow Q$, dann gilt $E \vdash Q:U$ (ohne Beweis)

- Das Typsystem ist also mit den Reduktionen verträglich
  - Aus korrekt typisierten Ausdrücken entstehen keine falsch typisierten Ausdrücke
- Ausdrücke können als ungültig erkannt werden

Beispiele:

in n[P]

$(\nu n{:}Amb[T])n.P$

$\langle in\ (in\ n) \rangle$

Syntaktisch möglich, aber keine gültigen Ausdrücke bezüglich des Typsystems; und laut obigem Satz auch nicht infolge von Reduktion zu erhalten

Anwendung

- Typisierte Datensätze
  - Jeder Datensatz r besteht aus Zellen c_i
  - Struktur: r[... | c_i^{buf}[⟨M_i⟩ | !open c_i^{ip}] | ...]
    - c_i^{buf}: Wertcontainer
    - c_i^{ip}: Eingangspakete
    - M_i: Zellinhalte
  - Operationen
    - record: Erzeugen eines neuen (leeren) Datensatzes
    - add: Zelle mit Inhalt einem Datensatz hinzufügen
    - get: Lesen des Inhalts einer Zelle und Belegen einer Variablen
    - set: Setzen des Inhalts einer Zelle

Typisierte Datensätze

◂record r▸ := r[ ]

◂add r c M▸ := c^{buf}[!open c^{ip} | in r.⟨M⟩]

◂get r c (x:W).P_S▸ := (νop:Amb[S])(open op.op[ ] | c^{ip}[in r.in c^{buf}.(x:W).
(⟨x⟩ | op[out c^{buf}.out r.open op.◂P▸])])

◂set r c ⟨M_W⟩.P_S▸ := (νop:Amb[S])(open op.op[ ] | c^{ip}[in r.in c^{buf}.(x:W).
(⟨M⟩ | op[out c^{buf}.out r.open op.◂P▸])])

P_S = Prozess des Typs S (Kurzschreibweise)
M_W = Nachricht des Typs W (Kurzschreibweise)

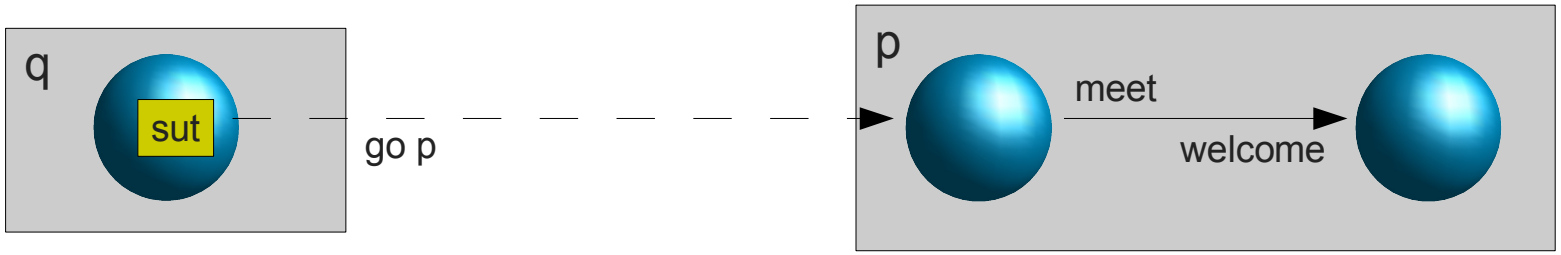
Field[W]: Typ der Namen eines Datensatzfeldes, das W beinhaltet

Typisierung von Agenten

- Hier am Beispiel Telescript
  - nur ein Ausschnitt: „Telestrip'd“
- Zunächst die Syntax von Telestrip'd in einer Tabelle
- Semantik durch Übersetzung in das Ambient-Kalkül
- Die hierarchische Struktur von Stellen, Agenten, Daten wird durch das Ambient-Kalkül optimal bewahrt

Telestrip'd

- Suitcase (Tasche) *sut*
  - Beinhaltet Folder (Ordner), die der Agent mit sich nimmt
  - Eine spezielle Zelle heißt „at“ und beinhaltet den Namen der aktuellen Stelle
    - at ist vom Typ Amb[Amb[Shh]]



- Verwendung von typisierten Datensätzen

Syntax von Telestrip'd

W::=Agent[W_1, ..., W_k]	Agententypen (k≥0)
Net ::=	Netzwerk
noplace	Keine Stelle
place p[Arena]	Stelle namens p
Net | Net	Weitere Stellen
Arena ::=	In einer Stelle
empty	Niemand hier
agent(n:W)[Code]	Agent mit neuem Namen n
Arena | Arena	Weitere Agenten
Code ::=	Agentencode
stop	Stop
go p.Code	Migriere zu p und mache dort weiter
spawn(n':W)[Code'].Code	Erzeuge an dieser Stelle einen neuen Agenten n'
welcome(n_1:W_1, ..., n_k:W_k).Code	Akzeptiere Daten von einem lokalen Agenten
meet n⟨n_1, ..., n_k⟩.Code	Ausgabe an Agent n
folder n n'.Code	Neuer Ordner n mit Inhalt n' in der Tasche
get n(x:W).Code	Hole Inhalt von Ordner n aus der Tasche
set n⟨n'⟩.Code	Setze Inhalt von Ordner n als n' in der Tasche
...	Weiteres weggelassen

Semantik von Telestrip'd

$\blacktriangleleft Agent[W_1,...,W_k]\blacktriangleright := Amb[\blacktriangleleft W_1\blacktriangleright \times ... \times \blacktriangleleft W_k\blacktriangleright]$

$\blacktriangleleft Net\blacktriangleright$: Shh

$\blacktriangleleft Arena\blacktriangleright_p$: Shh mit p:Amb[Shh]

$\blacktriangleleft Code\blacktriangleright_m$: $\blacktriangleleft W_1\blacktriangleright \times ... \times \blacktriangleleft W_k\blacktriangleright$ mit m:$\blacktriangleleft Agent[W_1,...,W_k]\blacktriangleright$

$\blacktriangleleft noplace\blacktriangleright$: **0**

$\blacktriangleleft place\ p[Arena]\blacktriangleright = p[\blacktriangleleft Arena\blacktriangleright_p]$ mit p:Amb[Shh]

$\blacktriangleleft Net \mid Net\blacktriangleright := \blacktriangleleft Net\blacktriangleright \mid \blacktriangleleft Net\blacktriangleright$

$\blacktriangleleft empty\blacktriangleright_p := \mathbf{0}$

$\blacktriangleleft agent(n:Agent[W_1,...,W_k])[Code]\blacktriangleright_p := (\nu n:\blacktriangleleft Agent[W_1,...,W_k]\blacktriangleright)$
$n[record\ sut \mid add\ sut\ at\ p \mid \blacktriangleleft Code\blacktriangleright_n]$

Semantik von Telestrip'd

$\blacktriangleleft \text{Arena} \mid \text{Arena} \blacktriangleright_p := \blacktriangleleft \text{Arena} \blacktriangleright_p \mid \blacktriangleleft \text{Arena} \blacktriangleright_p$

$\blacktriangleleft \text{stop} \blacktriangleright_m := \mathbf{0}$

$\blacktriangleleft \text{go p.Code} \blacktriangleright_m := \text{get sut at(q:Amb[Shh]).set sut at}\langle p \rangle\text{.out q.in p.}\blacktriangleleft \text{Code} \blacktriangleright_m$

$\blacktriangleleft \text{spawn (n':Agent}[W_1,\ldots,W_k]\text{)[Code'].Code} \blacktriangleright_m :=$
$\text{get sut at(p:Amb[Shh]).}(\nu n', u{:}\blacktriangleleft \text{Agent}[W_1,\ldots,W_k] \blacktriangleright)$
$(n'[\text{record sut} \mid \text{add sut at p} \mid \text{out m.open n.}\blacktriangleleft \text{Code'} \blacktriangleright_{n'}]$
$\mid \text{open u.}\blacktriangleleft \text{Code} \blacktriangleright_m$
$\mid (\nu t{:}\text{Amb[Shh]})\ t[\text{out m.in n'.out n'.}(u[\text{out t.in n'}] \mid u[\text{out t.in m}])])$

Semantik von Telestrip'd

$\blacktriangleleft$meet $n\langle n1_{W1}, ..., nk_{Wk}\rangle$.Code$\blacktriangleright_m$:= (νmsg:$\blacktriangleleft$Agent[W1,...,Wk]$\blacktriangleright$)
msg[out m.in n.n[out msg.open msg.$\langle$n1,...,nk$\rangle$]] | $\blacktriangleleft$Code$\blacktriangleright_m$

$\blacktriangleleft$welcome(n_1:W_1, ..., n_k:W_k).Code$\blacktriangleright_m$:= open m | (n_1:$\blacktriangleleft W_1\blacktriangleright$, ..., n_k:$\blacktriangleleft W_k\blacktriangleright$).$\blacktriangleleft$Code$\blacktriangleright_m$

$\blacktriangleleft$folder n n'$_W$.Code$\blacktriangleright_m$:= (νn:Field[$\blacktriangleleft$W$\blacktriangleright$])(add sut n n' | $\blacktriangleleft$Code$\blacktriangleright_m$)

$\blacktriangleleft$get n(x:W).Code$\blacktriangleright_m$:= get sut n(x:$\blacktriangleleft$W$\blacktriangleright$).$\blacktriangleleft$Code$\blacktriangleright_m$

$\blacktriangleleft$set n$\langle$n'$\rangle$.Code$\blacktriangleright_m$:= set sut n$\langle$n'$\rangle$.$\blacktriangleleft$Code$\blacktriangleright_m$

...

Beispiel

```
place Zuhause [
    agent(Alice:Agent[String]) [
        go Treffpunkt.
        meet Bob ⟨Alice, "Hallo", "Ich bin Alice"⟩.
        welcome (n1:String).
        go Zuhause.
        stop
    ]
]
|
place Treffpunkt [
    agent(Bob:Agent[Agent[String],String,String])[
        welcome (visitor:Agent[String], n1:String, n2:String).
        meet visitor ⟨"Willkommen"⟩.
        stop
    ]
]
```

„String“ sei hier als Datentyp im Ambient-Kalkül (bzw. in Telestrip'd) formuliert vorhanden

Wohltypisierter Ausdruck (kann durch geeigneten Typprüfer algorithmisch festgestellt werden.

Typisierte Agentensprache

- Resultat
  - Einfache Sprache zur Erstellung von Agentenanwendungen
  - Formal spezifizierte Semantik
  - Typisierte Sprache

Agententyp in Telestrip'd:

Agent[W_1, ..., W_k] ist der Typ der Namen von Agenten, welche eine Kommunikation der Form $W_1 \times ... \times W_k$ akzeptieren.

Das Typsystem garantiert den fehlerfreien Nachrichtenaustausch insbesondere unter Berücksichtigung der Mobilität.

Literatur

[1] Luca Cardelli, Andrew Gordon: Mobile Ambients; Theoretical Computer Science 240 (2000):177-213

[2] Luca Cardelli, Andrew Gordon: Types for Mobile Ambients, Lecture Notes In Computer Science; Vol. 1378, S. 140 – 155, ISBN:3-540-64300-1

Online:
http://research.microsoft.com/Users/luca/Papers/MobileAmbients.A4.pdf
http://research.microsoft.com/Users/luca/Papers/Types%20for%20Mobile%20Ambients.A4.pdf