Übung zur Vorlesung

Automatisierte Logik und Programmierung

Prof. Chr. Kreitz

Universität Potsdam, Theoretische Informatik — Wintersemester 2008/09

Blatt 4 — Abgabetermin: 9.12.08 nach der Übung

Aufgabe 4.1 (Typisierung)

Geben Sie, wo möglich, eine Typisierung für die folgenden Terme an.

4.1–a $\lambda t.\lambda y.\, t\, y\, y$

4.1–b $(\lambda x.\lambda y.\, x\, y)\, (\lambda z.\, z)$

4.1–c $\lambda f.\, (\lambda x.\, f\, (x\, x))\, (\lambda x.\, f\, (x\, x))$

4.1–d $\lambda y.\lambda g.\, (\lambda x.\, x^3\, y)\, (\lambda z.\, g\, z\, z)$

Welches Ergebnis würde eine Anwendung des Hindley–Milner–Algorithmus auf diese Terme liefern?

4.1-e Zeigen Sie durch Induktion, daß die Church–Numerals ($\overline{n} \equiv \lambda f.\lambda x.\, f^n\, x$) für alle $n \in \mathbb{N}$ mit $(X \to X) \to X \to X$ typisiert werden können.

Aufgabe 4.2 (Hindley–Milner Algorithmus in der Praxis)

In vielen funktionalen Programmiersprachen wie ML oder Haskell wird eine erweiterte Version des Hindley–Milner Typechecking Algorithmus eingesetzt um den Datentyp eines gegebenen Ausdrucks der Sprache zu bestimmen.

Wie müsste der Hindley–Milner Algorithmus erweitert werden, wenn neben dem einfachen Funktionenraum auch die folgenden Datenstrukturen zum Typsystem der Sprache gehören.

- Den Typ $\mathbb{B}$ der booleschen Werte zusammen mit den Elementen **T** und **F** und der Analyseoperation **if** b **then** s **else** t.
- Das Produkt $S \times T$ zweier Datentypen S und T zusammen mit dem Element $\langle s, t\rangle$ (Paarbildung) und der Analyseoperation **let** $\langle x, y\rangle = p$ **in** e.
- Den Typ $\mathbb{N}$ der natürlichen Zahlen mit den Element **0**, der Nachfolgeroperation **s**(i), den arithmetischen Ausdrücken $i+j$, i-j, i*j, i/j, i mod j, und einem induktiven Analyseoperator PR[$base$; h](i) (oft geschrieben als letrec f(x) = if x=0 then $base$ else h(x,f(x-1)) in f(i)).
- Den Typ T list der Listen über dem Datentyp T zusammen mit den Element [], der Operation t::l (t wird an den Anfang der Liste l gehängt) und einem induktiven Analyseoperator list_ind[$base$; h](l) (auch letrec f(x) = if x=[] then $base$ else let x=hd::tl in h(hd,tl,f(tl)) in f(l)).

Aufgabe 4.3 (Definitorische Erweiterung des Typsystems

Zeigen Sie, daß in der Typentheorie mit abhängigen Datentypen die folgenden Datentypen als definitorische Erweiterung erklärt werden können

4.3–a Das abhängige Produkt $x{:}S \times T[x]$ (vgl. Einheit 7, Folie 4)

4.3–b Die Summe $S+T$ (erzwungen disjunkte Vereinigung) zweier Datentypen S und T zusammen mit den Elementen inl (s) ("linksseitige" Einbettung eines $s \in S$) inr (t) und der Analyseoperation case e of inl (a) $\mapsto u$ | inr (b) $\mapsto v$.

4.3–c Ein leerer Datentyp ohne Elemente

Warum müsste ein leerer Datentyp eine Analyseoperation haben? Welchen Datentyp müsste diese sinnvollerweise haben?

Lösung 4.1 *Ziel dieser Aufgabe ist es, sich ein wenig auf das Typenkonzept und die zugehörigen Berechnungen einzustimmen. Es werden dazu einige Beispiele von niederer Schwierigkeit dargeboten:*

4.1–a λt.λy.t y y: Die Analyse von Hand ergibt $(S \to S \to T) \to S \to T$

Env	*Aktueller Term*	σ	UNIFY	*Typ*
	λt.λy.t y y			
t:X_0	λy.t y y			
t:X_0, y:X_1	t y y			
t:X_0, y:X_1	t y			
t:X_0, y:X_1	t			X_0
t:X_0, y:X_1	y			X_1
t:X_0, y:X_1	t y		X_0=$X_1 \to X_2$	X_2
t:X_0, y:X_1	y	[$X_1 \to X_2/X_0$]		X_1
t:X_0, y:X_1	t y y	[$X_1 \to X_2/X_0$]	X_2=$X_1 \to X_3$	X_3
t:X_0, y:X_1	λy.t y y	[$X_1 \to X_3$, $X_1 \to X_2/X_2$, X_0]		$X_1 \to X_3$
t:X_0, y:X_1	λt.λy.t y y	[$X_1 \to X_3$, $X_1 \to X_2/X_2$, X_0]		$(X_1 \to X_1 \to X_3) \to X_1 \to X_3$

4.1–b (λx.λy.x y) (λz.z): Die Analyse von Hand ergibt $S \to S$

Env	*Aktueller Term*	σ	UNIFY	*Typ*
	(λx.λy.x y) (λz.z)			
	λx.λy.x y			
x:X_0	λy.x y			
x:X_0, y:X_1	x y			
x:X_0, y:X_1	x			X_0
x:X_0, y:X_1	y			X_1
x:X_0, y:X_1	x y		X_0 = $X_1 \to X_2$	X_2
x:X_0, y:X_1	λy.x y	[$X_1 \to X_2/X_0$]		$X_1 \to X_2$
x:X_0, y:X_1	λx.λy.x y	[$X_1 \to X_2/X_0$]		$(X_1 \to X_2) \to X_1 \to X_2$
x:X_0, y:X_1	λz.z	[$X_1 \to X_2/X_0$]		
x:X_0, y:X_1, z:X_3	z	[$X_1 \to X_2/X_0$]		X_3
x:X_0, y:X_1, z:X_3	λz.z	[$X_1 \to X_2/X_0$]		$X_3 \to X_3$
x:X_0, y:X_1, z:X_3	(λx.λy.x y) (λz.z)	[$X_1 \to X_2/X_0$]	$= \begin{array}{l}(X_1 \to X_2) \to X_1 \to X_2 \\ (X_3 \to X_3) \to X_4\end{array}$	X_4
x:X_0, y:X_1, z:X_3	(λx.λy.x y) (λz.z)	$\left[\begin{array}{r} X_3 \to X_3/X_4 \\ X_3/X_1 \\ X_3/X_2 \\ X_3 \to X_3/X_0 \end{array}\right]$		$X_3 \to X_3$

4.1–c *Ziel dieser Teilaufgabe ist es, sich einmal klarzumachen, warum ein "Metakonstrukt" wie der Y–Kombinator nicht typisierbar sein kann.*

λf.(λx.f (x x)) (λx.f (x x))*: Wegen* x x *darf der Term nicht typisierbar sein*

Env	*Aktueller Term*	σ	UNIFY	*Typ*
	λf.(λx.f (x x)) (λx.f (x x))			
f:X_0	(λx.f (x x)) (λx.f (x x))			
f:X_0	λx.f (x x)			
f:X_0, x:X_1	f (x x)			
f:X_0, x:X_1	f			X_0
f:X_0, x:X_1	x x			
f:X_0, x:X_1	x			X_1
f:X_0, x:X_1	x			X_1
f:X_0, x:X_1	x x		X_1=$X_1 \to X_2$	X_2
f:X_0, x:X_1	f (x x)	[??/X_1]	X_0=$X_2 \to X_3$	X_3

Hier müßte ein Typ durch einen Funktionstyp ersetzt werden, in welchem er selbst vorkommt. Dies würde zu einer unendlichen Ersetzung führen, die somit nicht berechenbar ist Die Unifikation schlägt deshalb fehl. Tatsächlich wird der Kombinator typisierbar, sobald er auf einen typisierbaren Term angewendet wird, für den es einen Fixpunkt gibt.

4.1–d λy.λg. (λx. x^3 y) (λz. g z z): Die Analyse von Hand ergibt $y\in Y$, $g\in(S\to S\to T)$, $\lambda z.\,g\,z\,z\in S\to T$, $\lambda x.\,x^3\,y\in Y\to Y\to Y$, damit $S=T=Y$ und Gesamttyp $Y\to(Y\to Y\to Y)\to Y$

Env	*Aktueller Term*	σ	UNIFY	*Typ*
	$\lambda y.\lambda g.(\lambda x.\,x^3\,y)\,(\lambda z.\,g\,z\,z)$			
$y:X_0$	$\lambda g.(\lambda x.\,x^3\,y)\,(\lambda z.\,g\,z\,z)$			
$y:X_0,\ g:X_1$	$(\lambda x.\,x^3\,y)\,(\lambda z.\,g\,z\,z)$			
$y:X_0,\ g:X_1$	$\lambda x.\,x^3\,y$			
$y:X_0,\ g:X_1,\ x:X_2$	$x^3\,y$			
$y:X_0,\ g:X_1,\ x:X_2$	x			X_2
$y:X_0,\ g:X_1,\ x:X_2$	$x^2\,y$			
$y:X_0,\ g:X_1,\ x:X_2$	x			X_2
$y:X_0,\ g:X_1,\ x:X_2$	$x\,y$			
$y:X_0,\ g:X_1,\ x:X_2$	x			X_2
$y:X_0,\ g:X_1,\ x:X_2$	y			X_0
$y:X_0,\ g:X_1,\ x:X_2$	$x\,y$		$X_2=X_0\to X_3$	X_3
$y:X_0,\ g:X_1,\ x:X_2$	$x^2\,y$	$[X_0\to X_3/X_2]$	$X_0\to X_3=X_3\to X_4$	X_4
$y:X_0,\ g:X_1,\ x:X_2$	$x^3\,y$	$\begin{bmatrix} X_0\to X_3/X_2 \\ X_4/X_0 \\ X_4/X_3 \end{bmatrix}$	$X_4\to X_4=X_4\to X_5$	X_5
$y:X_0,\ g:X_1,\ x:X_2$	$\lambda x.\,x^3\,y$	$\begin{bmatrix} X_0\to X_3/X_2 \\ X_4/X_0 \\ X_4/X_3 \\ X_5/X_4 \end{bmatrix}$		$(X_5\to X_5)\to X_5$
$y:X_0,\ g:X_1,\ x:X_2$	$\lambda z.\,g\,z\,z$	$\begin{bmatrix} X_0\to X_3/X_2 \\ X_4/X_0 \\ X_4/X_3 \\ X_5/X_4 \end{bmatrix}$		
$y:X_0,\ g:X_1,\ x:X_2,\ z:X_6$	$g\,z\,z$	$\begin{bmatrix} X_0\to X_3/X_2 \\ X_4/X_0 \\ X_4/X_3 \\ X_5/X_4 \end{bmatrix}$		
$y:X_0,\ g:X_1,\ x:X_2,\ z:X_6$	$g\,z$	$\begin{bmatrix} X_0\to X_3/X_2 \\ X_4/X_0 \\ X_4/X_3 \\ X_5/X_4 \end{bmatrix}$		
$y:X_0,\ g:X_1,\ x:X_2,\ z:X_6$	g	$\begin{bmatrix} X_0\to X_3/X_2 \\ X_4/X_0 \\ X_4/X_3 \\ X_5/X_4 \end{bmatrix}$		X_1
$y:X_0,\ g:X_1,\ x:X_2,\ z:X_6$	z	$\begin{bmatrix} X_0\to X_3/X_2 \\ X_4/X_0 \\ X_4/X_3 \\ X_5/X_4 \end{bmatrix}$		X_6
$y:X_0,\ g:X_1,\ x:X_2,\ z:X_6$	$g\,z$	$\begin{bmatrix} X_0\to X_3/X_2 \\ X_4/X_0 \\ X_4/X_3 \\ X_5/X_4 \end{bmatrix}$	$X_1=X_6\to X_7$	X_7
$y:X_0,\ g:X_1,\ x:X_2,\ z:X_6$	z	$\begin{bmatrix} X_0\to X_3/X_2 \\ X_4/X_0 \\ X_4/X_3 \\ X_5/X_4 \\ X_6\to X_7/X_1 \end{bmatrix}$		X_6
$y:X_0,\ g:X_1,\ x:X_2,\ z:X_6$	$g\,z\,z$	$\begin{bmatrix} X_0\to X_3/X_2 \\ X_4/X_0 \\ X_4/X_3 \\ X_5/X_4 \\ X_6\to X_7/X_1 \end{bmatrix}$	$X_7=X_6\to X_8$	X_8
$y:X_0,\ g:X_1,\ x:X_2,\ z:X_6$	$\lambda z.\,g\,z\,z$	$\begin{bmatrix} X_0\to X_3/X_2 \\ X_4/X_0 \\ X_4/X_3 \\ X_5/X_4 \\ X_6\to X_7/X_1 \\ X_6\to X_8/X_7 \end{bmatrix}$		$X_6\to X_8$
$y:X_0,\ g:X_1,\ x:X_2,\ z:X_6$	$(\lambda x.\,x^3\,y)\,(\lambda z.\,g\,z\,z)$	$\begin{bmatrix} X_0\to X_3/X_2 \\ X_4/X_0 \\ X_4/X_3 \\ X_5/X_4 \\ X_6\to X_7/X_1 \\ X_6\to X_8/X_7 \end{bmatrix}$	$(X_5\to X_5)\to X_5 = (X_6\to X_8)\to X_9$	X_9
$y:X_0,\ g:X_1,\ x:X_2,\ z:X_6$	$(\lambda x.\,x^3\,y)\,(\lambda z.\,g\,z\,z)$	$\begin{bmatrix} X_0\to X_3/X_2 \\ X_4/X_0 \\ X_4/X_3 \\ X_5/X_4 \\ X_6\to X_7/X_1 \\ X_6\to X_8/X_7 \\ X_5/X_6 \\ X_5/X_8 \\ X_5/X_9 \end{bmatrix}$		X_5

4.1–e **Typisierung von $\overline{0}$ mit** $(\mathtt{X}\to\mathtt{X})\to\mathtt{X}\to\mathtt{X}$:

Env	*Aktueller Term*	σ	UNIFY	*Typ*
	λf.λx.x			
f:X_0	λx.x			
f:X_0, x:X_1	x			X_1
f:X_0, x:X_1	λx.x			$X_1\to X_1$
f:X_0, x:X_1	λf.λx.x			$X_0\to X_1\to X_1$

Setze nun einfach X_0=$X_1\to X_1$ sowie X_1=X.

Typisierung von $\overline{1}$ mit $(\mathtt{X}\to\mathtt{X})\to\mathtt{X}\to\mathtt{X}$:

Wurde bereits in 4.1–b gezeigt (ersetze dort X_1 und X_2 jeweils durch X).

Typisierung von $\overline{n+1}$ mit $(\mathtt{X}\to\mathtt{X})\to\mathtt{X}\to\mathtt{X}$**, falls $\overline{n}$ mit** $(\mathtt{X}\to\mathtt{X})\to\mathtt{X}\to\mathtt{X}$ **typisierbar ist**:

Sei λf.λx. f^n x vom Typ $(\mathtt{X}\to\mathtt{X})\to\mathtt{X}\to\mathtt{X}$. Dann gelten folgende Typisierungen:

1. f ist vom Typ X→X
2. x ist vom Typ X
3. f^n x ist vom Typ X

Wir typisieren nun λf.λx. f^{n+1} x $\equiv$ λf.λx. f (f^n x) unter Beachtung dieser Bedingungen:

Env	*Aktueller Term*	σ	UNIFY	*Typ*
f:X→X, x:X, f^n x:X	λf.λx.f (f^n x)			
f:X→X, x:X, f^n x:X	λx.f (f^n x)			
f:X→X, x:X, f^n x:X	f (f^n x)			
f:X→X, x:X, f^n x:X	f			X→X
f:X→X, x:X, f^n x:X	(f^n x)			X
f:X→X, x:X, f^n x:X	f (f^n x)		X→X=X→X_0	X_0
f:X→X, x:X, f^n x:X	λx.f (f^n x)	[X/X_0]		X→X
f:X→X, x:X, f^n x:X	λf.λx.f (f^n x)	[X/X_0]		(X→X)→X→X

Lösung 4.2

- **Eingabe**: geschlossener Term t
  Ausgabe: prinzipielles Typschema von t oder Fehlermeldung
- **Start**: Setze globale Substitution σ := [] und rufe TYPE-OF([], t) auf
- **Algorithmus** TYPE-OF(Env, t):

Falls t = $x \in \mathcal{V}$: suche Deklaration x:T in Env — Ausgabe: T

Falls t = $f\ u$: Setze S_1:=TYPE-OF(Env, f), S_2:=TYPE-OF(Env, u)
Wähle neues X_{i+1} und unifiziere $\sigma(S_1)$ mit $S_2 \rightarrow X_{i+1}$.
Fehlermeldung, wenn Unifikation fehlschlägt.
Sonst $\sigma := \sigma' \circ \sigma$, wobei σ' Ergebnis der Unifikation — Ausgabe: $\sigma(X_{i+1})$

Falls t = $\lambda x.u$: Wähle neues X_{i+1}

$$S_1 := \begin{cases} \texttt{TYPE-OF}(Env\cdot[x{:}X_{i+1}], u) & \text{falls } x \text{ nicht in } Env \\ \texttt{TYPE-OF}(Env\cdot[x'{:}X_{i+1}], u[x'/x]) & \text{sonst } (x' \in \mathcal{V} \text{ neu}) \end{cases}$$

Ausgabe: $\sigma(X_{i+1}) \rightarrow S_1$

Falls t = **T** oder t = **F**: — Ausgabe: $\mathbb{B}$

Falls t = **if** b **then** s **else** t:
Setze S_1:=TYPE-OF(Env, b), S_2:=TYPE-OF(Env, s), S_3:=TYPE-OF(Env, t)
Unifiziere $\sigma(S_1)$ mit $\mathbb{B}$. Fehlermeldung, wenn Unifikation fehlschlägt.
Sonst $\sigma := \sigma' \circ \sigma$, wobei σ' Ergebnis der Unifikation
Unifiziere $\sigma(S_2)$ mit $\sigma(S_3)$. Fehlermeldung, wenn Unifikation fehlschlägt.
Sonst $\sigma := \sigma' \circ \sigma$, wobei σ' Ergebnis der Unifikation — Ausgabe: $\sigma(S_3)$

Falls t = $\langle s, t\rangle$: Setze S_1:=TYPE-OF(Env, s), S_2:=TYPE-OF(Env, t) — Ausgabe: $S_1 \times S_2$

Falls t = **let** $\langle x, y\rangle = p$ **in** e: Wähle neues X_{i+1} und X_{i+2}
Setze S_1:=TYPE-OF(Env, p)
Unifiziere $\sigma(S_1)$ mit $X_{i+1} \times X_{i+2}$. Fehlermeldung, wenn Unifikation fehlschlägt.
Sonst $\sigma := \sigma' \circ \sigma$, wobei σ' Ergebnis der Unifikation

$$S_2 := \begin{cases} \texttt{TYPE-OF}(Env\cdot[x{:}X_{i+1};\ y{:}X_{i+2}], e) & \text{falls } x, y \text{ nicht in } Env \\ \texttt{TYPE-OF}(Env\cdot[x'{:}X_{i+1};\ y'{:}X_{i+2}], e[x', y'/x, y]) & \text{sonst } (x', y' \in \mathcal{V} \text{ neu}) \end{cases}$$

Ausgabe: $\sigma(S_2)$

Falls t = **0**: — Ausgabe: $\mathbb{N}$

Falls t = **s(**i**)**: Setze S_1:=TYPE-OF(Env, i)
Unifiziere $\sigma(S_1)$ mit $\mathbb{N}$. Fehlermeldung, wenn Unifikation fehlschlägt.
Sonst $\sigma := \sigma' \circ \sigma$, wobei σ' Ergebnis der Unifikation — Ausgabe: $\mathbb{N}$

Falls t = i+j oder t = i-j oder t = i*j oder t = i/j oder t = i **mod** j:
Setze S_1:=TYPE-OF(Env, i), S_2:=TYPE-OF(Env, j)
Unifiziere $\sigma(S_1)$ mit $\mathbb{N}$. Fehlermeldung, wenn Unifikation fehlschlägt.
Sonst $\sigma := \sigma' \circ \sigma$, wobei σ' Ergebnis der Unifikation
Unifiziere $\sigma(S_2)$ mit $\mathbb{N}$. Fehlermeldung, wenn Unifikation fehlschlägt.
Sonst $\sigma := \sigma' \circ \sigma$, wobei σ' Ergebnis der Unifikation — Ausgabe: $\mathbb{N}$

Falls t = **PR[**$base$; h**](**i**)**: Setze S_1:=TYPE-OF(Env, i), S_2:=TYPE-OF(Env, $base$), S_3:=TYPE-OF(Env, h)
Unifiziere $\sigma(S_1)$ mit $\mathbb{N}$. Fehlermeldung, wenn Unifikation fehlschlägt.
Sonst $\sigma := \sigma' \circ \sigma$, wobei σ' Ergebnis der Unifikation
Unifiziere $\sigma(S_3)$ mit $\sigma(\mathbb{N} \rightarrow S_2 \rightarrow S_2)$. Fehlermeldung, wenn Unifikation fehlschlägt.
Sonst $\sigma := \sigma' \circ \sigma$, wobei σ' Ergebnis der Unifikation — Ausgabe: $\sigma(S_2)$

Falls t = **[]**: Wähle neues X_{i+1} — Ausgabe: X_{i+1} list

Falls t = t**::**l:
Setze S_1:=TYPE-OF(Env, t), S_2:=TYPE-OF(Env, l)
Unifiziere $\sigma(S_2)$ mit $\sigma(S_1)$ list. Fehlermeldung, wenn Unifikation fehlschlägt.
Sonst $\sigma := \sigma' \circ \sigma$, wobei σ' Ergebnis der Unifikation — Ausgabe: $\sigma(S_2)$

Falls t = **list_ind[**$base$; h**](**l**)**:
Setze S_1:=TYPE-OF(Env, l), S_2:=TYPE-OF(Env, $base$), S_3:=TYPE-OF(Env, h)
Wähle neues X_{i+1}
Unifiziere $\sigma(S_1)$ mit X_{i+1} list. Fehlermeldung, wenn Unifikation fehlschlägt.
Sonst $\sigma := \sigma' \circ \sigma$, wobei σ' Ergebnis der Unifikation
Unifiziere $\sigma(S_3)$ mit $\sigma(X_{i+1} \rightarrow X_{i+1}$ list $\rightarrow S_2 \rightarrow S_2)$. Fehlermeldung, wenn Unifikation fehlschlägt.
Sonst $\sigma := \sigma' \circ \sigma$, wobei σ' Ergebnis der Unifikation — Ausgabe: $\sigma(S_2)$

Lösung 4.3

4.3–a Das abhängige Produkt $x{:}S{\times}T[x]$ besitzt dieselben Elemente und Analyseoperationen wie das unabhängige Produkt $S{\times}T$. Allerdings müssten diese durch etwas anderes simuliert werden, wenn $x{:}S{\times}T[x]$ simuliert wird und $S{\times}T$ ebenfalls als Spezialfall der Simulation aufgefaßt werden. Die Erweiterung der bekannten Simulation von Paaren durch λ-Terme ergibt:

$$\begin{array}{lcl} x{:}S{\times}T[x] & \equiv & \mathtt{X}{:}\mathbb{U}\rightarrow(x{:}S\rightarrow T[x]\rightarrow \mathtt{X})\rightarrow \mathtt{X} \\ \langle s,t\rangle & \equiv & \lambda \mathtt{X}.\lambda \mathtt{p}.\,\mathtt{p}\,s\,t \\ \textbf{let}\ \langle x,y\rangle = p\ \textbf{in}\ e & \equiv & p\,\mathtt{X}\,(\lambda x.\lambda y.e) \end{array}$$

Dabei ist X der Typ von e, der bestimmt werden müsste, bevor das Konstrukt instantiiert werden kann, oder als Parameter mit auftauche müsste. Wie man sieht, ist eine komplette Simulation abhängiger Produkte nicht trivial. *Das muß ich noch einmal durchdenken*

4.3–b Bei Lichte besehen geschieht durch die disjunkte Vereinigung nichts anderes, als daß man zwei Typen S und T hernimmt, die Elemente s aus S mit einem ≫*Links*≪–Sticker ($\mathsf{inl}(s)$) versieht, die t aus T mit einem ≫*Rechts*≪–Sticker ($\mathsf{inr}(t)$) versieht und anschließend alle Elemente zusammen in einen Topf ($S{+}T$) wirft. Mit $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(x)\mapsto u\ \mid\ \mathsf{inr}(y)\mapsto v$ wird dann ein e aus dem Topf $S{+}T$ herausgenommen und festgestellt ob es einen ≫*Links*≪–Sticker hat. Wenn ja, so wird dieser entfernt und das übriggebliebene Element in u für x eingesetzt. Ansonsten wird der ≫*Rechts*≪–Sticker entfernt und das übriggebliebene Element in v für y eingesetzt.

Wir brauchen also nichts weiter zur Verfügung zu stellen, als einen ≫Record≪, bestehend aus einem ≫Sticker≪ und einem Elementfeld, welches in Abhängigkeit von einem konkreten Sticker entweder dem linken oder dem rechten der beiden zu vereinigenden Typen angehört:

```
type disjoint_union_S_T = record
                            case sticker : boolean of
                              true  :  S;
                              false :  T
                            end;
```

In der Schreibweise der Typentheorie hätten wir somit:

$$\begin{array}{lcl} S{+}T & \equiv & x{:}\mathbb{B}\times(\textbf{if}\ x\ \textbf{then}\ S\ \textbf{else}\ T) \\ \mathsf{inl}(s) & \equiv & \langle \mathbf{T}, s\rangle \\ \mathsf{inr}(t) & \equiv & \langle \mathbf{F}, t\rangle \end{array}$$

$$\begin{array}{ll} & \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(x)\mapsto u\ \mid\ \mathsf{inr}(y)\mapsto v \\ \equiv & \textbf{let}\ \langle b,z\rangle = e\ \textbf{in}\ \textbf{if}\ b\ \textbf{then}\ (\lambda x.u)\ z\ \textbf{else}\ (\lambda y.v)\ z \end{array}$$

Der aufmerksame Leser bemerkt, daß die obige Simulation für das **decide***–Konstrukt durch* **spread** *und* **cond** *problematisch ist, sofern die Variable ≫b≪ bzw. ≫z≪ frei in ≫u≪ oder ≫v≪ auftritt. In diesem Fall würde in der Simulation im Gegensatz zum herkömmlichen* **decide** *eine Substitution der entsprechenden Variablen in den betroffenen Termen erfolgen. Hierfür gibt es denn zwei denkbare Auswege:*

1. *Man denkt sich möglichst exotische Namen für ≫b≪ bzw. ≫z≪ in der Hoffnung aus, daß keinem vernünftigen Menschen jemals die Idee käme, eine Variable dieses Namens zu verwenden. Beispiel:*

   ≫`cond_element_of$&_decide_simulation_%>`≪

   Das prinzipielle Problem wird so natürlich nicht behoben…

2. *Man verwendet die in NuPRL eigens für solche Fälle eingeführten* Metavariablen *für ≫b≪ bzw. ≫z≪, mit denen sich solche Konflikte generell vermeiden lassen.*

4.3–c $\mathsf{void} \equiv$ `X:`$\mathbb{U}\rightarrow$`X`

Kanonische Elemente für diesen Typ sind nicht konstruierbar: man müsste eine Funktion beschreiben können, die bei Eingabe eines beliebigen Typs ein Element dieses Typs bestimmt. Das würde nur funktionieren unter der Voraussetzung, daß jeder Typ Elemente haben muß – was in etwa der Behauptung gleichkommt, daß jede mathematische Formel einen Beweis hat, also wahr sein muß.

Die obige Definition liefert eine Analyseoperation für void mit recht seltsamen Eigenschaften. Definieren wir `any(`z`,`T`)` $\equiv$ $z\,T$, so gilt `any(`z`,`T`)` $\in$ T für jeden beliebigen Datentyp T. D.h. wir können Elemente des leeren Datentyps in Elemente eines jeden Typs tranformieren, ... wenn es denn Elemente von void gäbe. Die Angabe des Zieltyps bei der Simulation ist nur erforderlich, um der Simulation sagen zu könne, wohin denn abgebildet werden soll. Für die Eigenschaften von void ist dies nicht erforderlich. Eigentlich müsste man einfach sagen `any(`z`)` $\in T$ für alle $z \in$void und jeden Typ T.

Unabhängig von der Simulierbarkeit des leeren Datentyps sind Analyseoperationen nötig für die Bildung von `void`$\rightarrow$`T`. Dies braucht Elemente der Form λ`z.t` mit $t \in T$ wenn $z \in$void. Interessanterweise kann an dieser Stelle *jeder* beliebige Term eingesetzt werden, da die Voraussetzung $z \in$void nicht erfüllbar ist. So wäre z.B. λ`z.[]` ein Element von void$\rightarrow\mathbb{N}$, obwohl `[]` kein Element von $\mathbb{N}$ ist.