Struktur des Vortrags

Wiederholung

- Funktional-logische Programmierung
- Residuierung, Needed Narrowing
- Die Sprache Curry

Harry

- Core Sprache
- Narrowing, Backtracking
- Residuierung, Task-Verwaltung
- Ausblick

Funktional-logische Programmierung

- Funktionale Programmierung
  - (Higher-order-)Funktionen und Gleichungen
  - Geschachtelte Ausdrücke
  - Vollständig deterministische Reduktion
  - Lazyness
- Logische Programmierung
  - Prädikate und logische Formeln
  - Logische Variable, damit u. a. Rechnen mit unvollständigen Datenstrukturen
  - Suche nach Lösungen, Nichtdeterminismus

$\Rightarrow$ Funktional-logische Programmierung

  - Ausdrucksstärke der logischen Programmierung kombiniert mit dem effizienten Auswertungsprinzip und den Strukturierungsmöglichkeiten der funktionalen Programmierung

Residuierung

Verwendet in Funlog, Le Fun, Life, NUE-Prolog, Escher, Oz, . . .

Behält die Idee der deterministischen Reduktion von Funktionen bei: Funktionsaufrufe mit Variablen suspendieren, bis diese Variablen gebunden sind. Nichtdeterminismus durch Prädikate und/oder Disjunktionen, Nebenläufigkeit explizit, z. B. durch Konjunktion

```
add Z     y -> y
add (S x) y -> S (add x y)

nat Z
nat (S x) <= nat x
```

Auszuwertender Ausdruck: `add w Z = S Z` $\wedge$ `nat w`

`add w Z = S Z` $\wedge$ `nat w`
`add Z Z = S Z` $\wedge$ `w = Z` *(nichtdet.)*
`Z = S Z` $\wedge$ `w = Z`
fail

`add w Z = S Z` $\wedge$ `nat w`
`add (S Z) Z = S Z` $\wedge$ `w = S Z` *(nichtdet.)*
`S (add Z Z) = S Z` $\wedge$ `w = S Z`
`add Z Z = Z` $\wedge$ `w = S Z`
`Z = Z` $\wedge$ `w = S Z`
`w = S Z`
Berechnete Antwort: $\{\mathtt{w} \mapsto \mathtt{S\ Z}\}$

Residuierung (Forts.)

Leider unvollständig:

```
append []     ys -> ys
append (x:xs) ys -> x : append xs ys
```

Der Versuch, das letzte Element `y` der Liste `[Z,Z]` wie folgt zu berechnen, führt bei Residuierung nicht zum Erfolg:

```
append xs [y] = [Z,Z]
```

```
rev [] []
rev xs (y:ys) <= append l [y] = xs /\ rev l ys
```

Für den Ausdruck

```
rev [Z, S Z, S (S Z)] r
```

existiert unendliche Ableitung (immer wieder 2. Regel für `rev`), da `append`-Terme nie hinreichend instanziiert werden. Narrowing (s. u.) hat dagegen einen endlichen Suchraum.

Einfach: Einbau externer Funktionen

Narrowing

Verwendet in EQLOG, SLOG, L-LEAF, ALF, Babel, ...

I. allg. nichtdeterministisches Instanziieren von Variablen im Funktionsaufruf

Definition: t läßt sich an der Stelle p mit der Regel $l \to r$ zu t' narrowen (Schreibweise: $t \leadsto_{[p, l \to r, \sigma]} t'$), falls gilt:

- Sei p eine nicht-variable Position in t (d. h. $t|_p \notin \mathit{Vars}$)
- $l \to r$ ist eine frische Variante einer Regel
- σ ist mgu von $t|_p$ und l
- $t' = \sigma(t[r]_p)$

Beispiele:

$$\begin{array}{rrcl} & \underline{\texttt{add w Z}} & = & \texttt{S Z} \\ \leadsto_{[1, add\ Z\ g \to g, \{w \mapsto Z\}]} & \texttt{Z} & = & \texttt{S Z} \end{array}$$

$$\begin{array}{rrcl} & \underline{\texttt{add w Z}} & = & \texttt{S Z} \\ \leadsto_{[1, add\ (S\ g)\ h \to S\ (add\ g\ h), \{w \mapsto S\ g\}]} & \texttt{S (add g Z)} & = & \texttt{S Z} \end{array}$$

Schreibvereinfachung, z. B. im letzen Fall:

$$\underline{\texttt{add w Z}} = \texttt{S Z} \quad \leadsto_{\{w \mapsto S\ g\}} \quad \texttt{S (add g Z)} = \texttt{S Z}$$

Nachteil: Großer Suchraum (Auswahl von Termposition und Regel)

Eager Narrowing

Eager (= leftmost-innermost) Narrowing terminiert nicht immer, wenn es möglich ist:

```
take Z     xs              = []
take (S n) []              = []
take (S n) (x:xs)          = x : take n xs

from n = n : from (S n)
```

Problematisch z. B. `take (S Z) (from Z).`

Lazy Narrowing

- Narrowing-Schritte nur an äußersten Positionen
- Auswertung von Argumenten nur, falls notwendig

Damit terminiert obiges Beispiel. Weiteres Beispiel mit Variablen im auszuwertendem Ausdruck:

$$
\begin{array}{ll}
 & \texttt{take v } \underline{\texttt{(from w)}} \texttt{ = [Z]} \\
\rightsquigarrow_{\{\}} & \underline{\texttt{take v (w : from (S w))}} \texttt{ = [Z]} \\
\rightsquigarrow_{\{v \mapsto S\ n\}} & \underline{\texttt{w : take n (from (S w)) = [Z]}} \\
\rightsquigarrow_{\{w \mapsto Z\}} & \underline{\texttt{take n (from (S w))}} \texttt{ = []} \\
\rightsquigarrow_{\{n \mapsto Z\}} & \texttt{[] = []}
\end{array}
$$

Berechnete Antwort: $\{ \texttt{v} \mapsto \texttt{S Z}, \texttt{w} \mapsto \texttt{Z} \}$

Lazy Narrowing (Forts.)

Lazy Narrowing macht jedoch u. U. unnötige Schritte:

```
leq Z     y     -> True
leq (S x) Z     -> False
leq (S x) (S y) -> leq x y
```

$$\begin{array}{ll} & \underline{\texttt{leq u (add u v)}} \\ \rightsquigarrow_{\{u \mapsto Z\}} & \texttt{True} \end{array}$$

$$\begin{array}{ll} & \texttt{leq u } \underline{\texttt{(add u v)}} \\ \rightsquigarrow_{\{u \mapsto Z\}} & \underline{\texttt{leq Z v}} \\ \rightsquigarrow_{\{\}} & \texttt{True} \end{array}$$

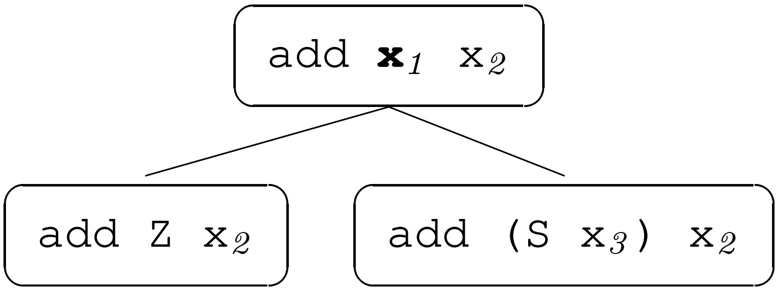
$$\begin{array}{ll} & \texttt{leq u } \underline{\texttt{(add u v)}} \\ \rightsquigarrow_{\{u \mapsto S\ g\}} & \underline{\texttt{leq (S Z) (S (add g v))}} \\ \rightsquigarrow_{\{\}} & \underline{\texttt{leq Z (add g v)}} \\ \rightsquigarrow_{\{\}} & \texttt{True} \end{array}$$

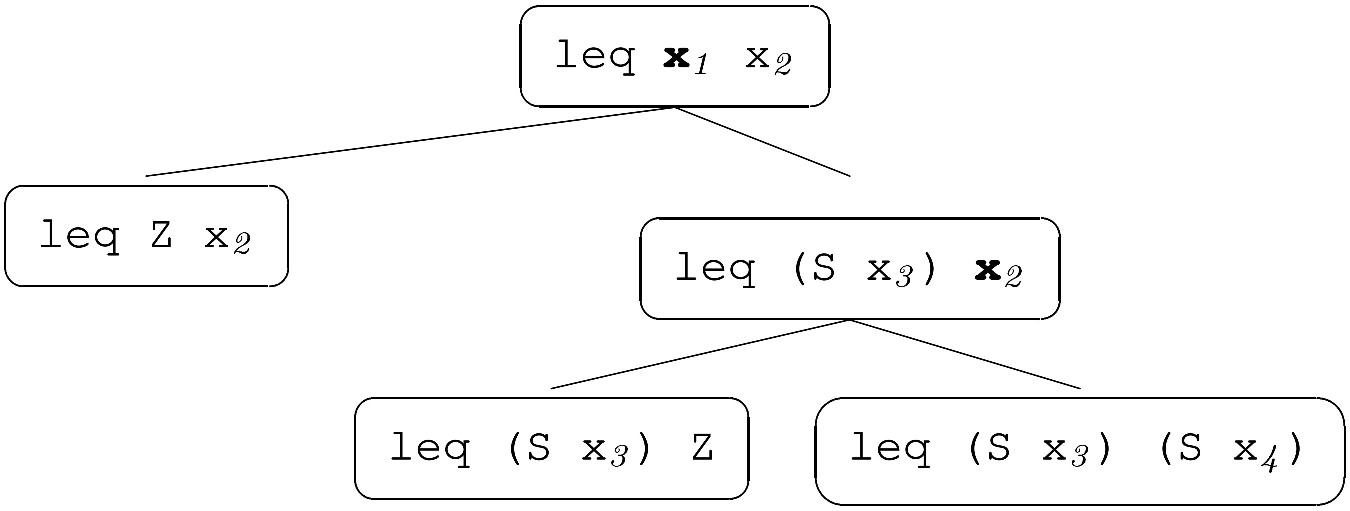
Problem: Reihenfolge Regelauswahl/Instanziierung

Needed Narrowing

Idee: Partitionierung der Musterpositionen in solche, die *immer* benötigt werden und solche, die (evtl.) nicht benötigt werden.

Darstellung der Strategie durch *Definitional Trees.*





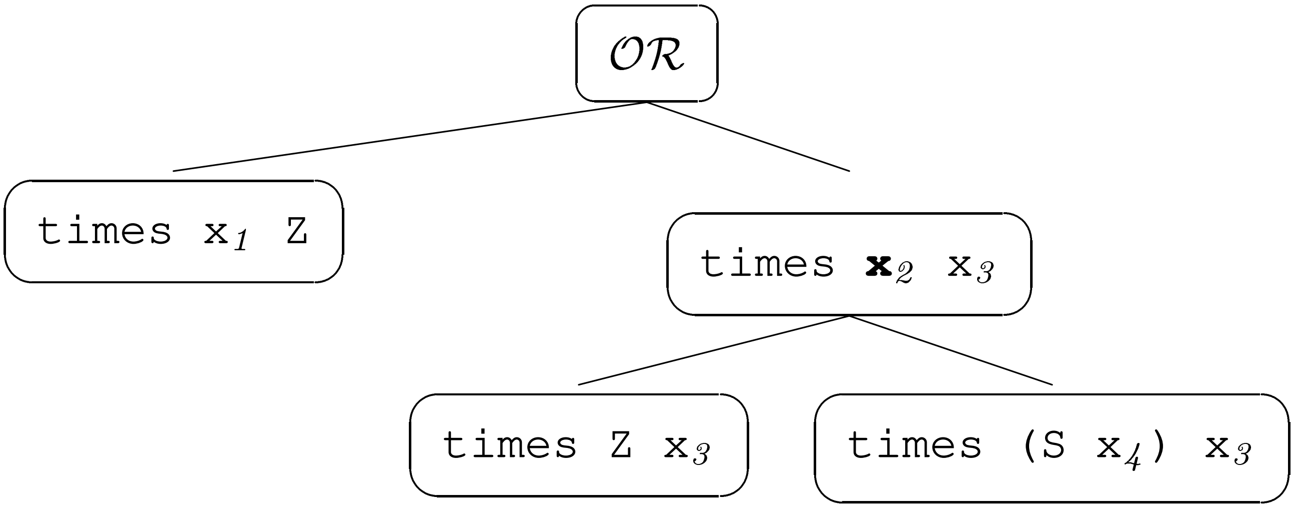
Optimalität von Needed Narrowing:

- Keine unnötigen Schritte
- Kürzeste Ableitungen (falls Term-Sharing gemacht wird)
- Unabhängigkeit der Lösungen: Sind σ und σ' zwei Lösungen, so gilt für alle Variablen x: $\sigma(x)$ und $\sigma'(x)$ sind nicht unifizierbar.

OR-Knoten

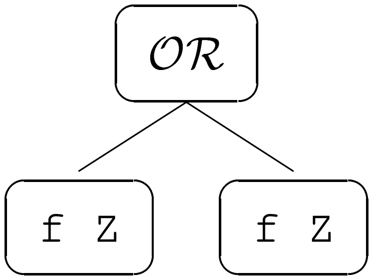
Überlappende Regeln

```
times Z     y = Z
times x     Z = Z
times (S x) y = add y (times x y)
```



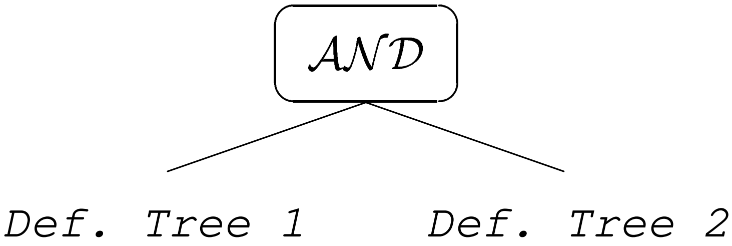
Nichtdeterministische Funktionen

```
f Z = Z
f Z = S Z
```



Nebenläufigkeit

Durch Einführung eines weiteren Knotentyps auch nebenläufige Auswertungsstrategien spezifizierbar:



Versuche zunächst gemäß des ersten Unterbaums auszuwerten. Falls diese Berechnung suspendiert, führe einen Berechnungsschritt gemäß des zweiten Unterbaums durch. Wird dadurch eine Variable gebunden, die für die Suspension des ersten Unterbaums verantwortlich war, wird die Berechnung im ersten Unterbaums fortgeführt: *Interleaving*

Curry

Disjunktive Ausdrücke

Was erwartet man als Rückgabe?

- Bei funktionalen Sprachen: Wert
- Bei logischen Sprachen: Substitution

$\Rightarrow$ Ein *Antwortausdruck* ist ein Paar $\sigma \| e$, bestehend aus einer Substitution σ und einem Ausdruck e. Ist σ die Identität, so schreibt man einfach e statt $\{\} \| e$.

Enthält e keine Variablen mehr, so nennt man $\sigma \| e$ *gelöst.*

Ein *disjunktiver Ausdruck* ist eine Multimenge von Antworausdrücken: $\{\sigma_1 \| e_1, \ldots, \sigma_n \| e_n\}$, geschrieben als $\sigma_1 e_1 \mid \ldots \mid \sigma_n e_n$.

Beispiel

```
default narrow

f :: Int -> Int
f 1 = 11
f 2 = 22
```

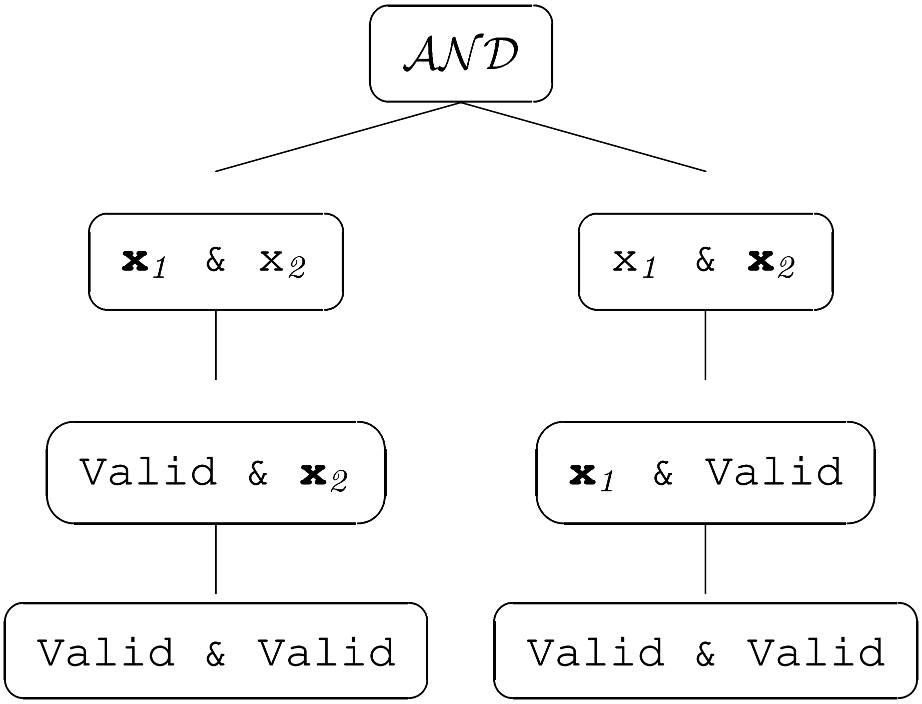
Der Anfrageausdruck `f x` liefert den disjunktiven Ausdruck
$\{x \mapsto 1\}11 \mid \{x \mapsto 2\}22$

Constraints

Curry bietet als Typ einfache Gleichheitsconstraints (`=:=`) und eine *nebenläufige* Konjunktion (`&`) von Constraints.

```
data Constraint = Valid

(&) :: Constraint -> Constraint -> Constraint
Valid & Valid = Valid
```



Constraints (Forts.)

Constraints können als Bedingungen in Gleichungen auftreten:

```
member :: a -> [a] -> Bool
member x ys | append l1 (x:l2) =:= ys = True
```

Constraints sind first-class-citizens:

```
conj :: [Constraint] -> Constraint
conj []     = Valid
conj (c:cs) = c & conj cs
```

oder für Higher-Order-Liebhaber: `:-)`

```
conj = foldr (&) Valid
```

Damit sind nebenläufige Kontrollstrukturen selbst programmierbar.

Weitere Features von Curry

- Modulsystem
- Kontrolle der Suche
- Erweiterung um Higher-Order-Unifikation

Interpretationsphasen

Quelltext

$\downarrow$

$\boxed{\text{Parser}}$

$\downarrow$

Reader-Syntax

$\downarrow$

$\boxed{\text{Mangler 1}}$

$\downarrow$

Typechecker-Syntax

$\downarrow$

$\boxed{\text{Typechecker}}$

$\downarrow$

Typechecker-Syntax

$\downarrow$

$\boxed{\text{Mangler 2}}$

$\downarrow$

Core-Syntax

$\downarrow$

$\boxed{\text{Interpreter}}$

$\downarrow$

Ausgabe

Abstrakte Syntax der Core-Sprache

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Function} & = & \mathit{Function}\ \mathit{Name}\ [\mathit{Name}]\ \mathit{Expr} \\
\\
\textbf{data}\ \mathit{Expr} & = & \mathit{Lambda}\ [\mathit{Name}]\ \mathit{Expr} \\
 & | & \mathit{FreeDecl}\ \mathit{Name}\ \mathit{Expr} \\
 & | & \mathit{Let}\ \mathit{LocalDef}\ \mathit{Expr} \\
 & | & \mathit{Letrec}\ [\mathit{LocalDef}]\ \mathit{Expr} \\
 & | & \mathit{Appl}\ \mathit{Expr}\ \mathit{Expr} \\
 & | & \mathit{Var}\ \mathit{Name} \\
 & | & \mathit{Constructor}\ \mathit{ConstrTag}\ [\mathit{Expr}] \\
 & | & \mathit{Literal}\ \mathit{Lit} \\
 & | & \mathit{Builtin}\ \mathit{BuiltinTag}\ \mathit{Arity} \\
 & | & \mathit{Case}\ \mathit{Expr}\ [\mathit{Clause}] \\
\\
\textbf{data}\ \mathit{LocalDef} & = & \mathit{VarBinding}\ \mathit{Name}\ \mathit{Expr} \\
 & | & \mathit{FuncBinding}\ \mathit{Function} \\
\\
\textbf{data}\ \mathit{BuiltinTag} & = & \mathit{BuiltinTagGuard} \\
 & | & \mathit{BuiltinTagUnify} \\
 & | & \mathit{BuiltinTagOr} \\
 & | & \mathit{BuiltinTagFail} \\
 & | & \mathit{BuiltinTagIfFree} \\
 & | & \mathit{BuiltinTagSuspendOn} \\
 & | & \mathit{BuiltinTagAnd} \\
\\
\textbf{data}\ \mathit{Clause} & = & \mathit{Clause}\ \mathit{ConstrTag}\ [\mathit{Name}]\ \mathit{Expr}
\end{array}
$$

Built-ins

- `guard` *constraint expr*: Wenn *constraint* erf¸llt ist, wird *expr* zur¸ckgegeben, andernfalls scheitert die Auswertung.
- `unify` $expr_1$ $expr_2$: Unifiziert die zwei Ausdr¸cke.
- `or` $expr_1$ $expr_2$: Gibt die Vereinigung der Antworten der zwei Ausdr¸cke zur¸ck (selbe Wirkung wie ein Or-Knoten in einem Definitional Tree).
- `fail`: Scheitert explizit (d. h. backtrackt).
- `ifFree` *name* $expr_1$ $expr_2$: Wenn die Variable, die durch *name* angegeben wird, frei ist, wird $expr_1$ zurückgegeben, andernfalls $expr_2$.
- `suspendOn` *name expr*: Wenn die durch *name* angegebene Variable frei ist, suspendiert die Auswertung. Sobald sie einen Wert erhält, wird *expr* zurückgegeben.
- `and` $constraint_1$ $constraint_2$: Löst gleichzeitig die beiden Constraints (selbe Wirkung wie eine Konjunktion von Constraints).

Case-Anweisung

Die Case-Anweisung wird in drei Schritten eingef¸hrt:

- Einf¸hrung eines theoretischen, annotierten Case.
- Einf¸hrung der Core-Case-Anweisung.
- Transformation des annotierten in das Core-Case.

Annotiertes Case

annotCase	::=	*caseRigid* | *caseFlex*
caseRigid	::=	`caseR` *var* `of` {*branch*}+
caseFlex	::=	`caseF` *var* `of` {*branch*}+
branch	::=	*constructor* {*var*} `->` *expression*

Beispiel: Annotiertes Case

Programm:

$$\begin{array}{lcl}
\textbf{data}\ Nat & = & Z \mid S\ Nat \\
leq\ ::\ Nat \to Nat \to Bool & & \\
leq\ Z\ \ \ \ \ \ \ n & = & True \\
leq\ (S\ m)\ Z & = & False \\
leq\ (S\ m)\ (S\ n) & = & leq\ m\ n
\end{array}$$

Darstellung der Needed-Narrowing-Auswertung mit annotiertem Case:

$$\begin{array}{l}
leq\ x_1\ x_2 \ \to\ \textbf{caseF}\ x_1\ \textbf{of} \\
\qquad\qquad\qquad Z \quad\ \to\ True \\
\qquad\qquad\qquad S\ x_3 \ \to\ \textbf{caseF}\ x_2\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad Z \quad\ \to\ False \\
\qquad\qquad\qquad\qquad\qquad S\ x_4 \ \to\ leq\ x_3\ x_4
\end{array}$$

Vom annotiertem zum Core-Case

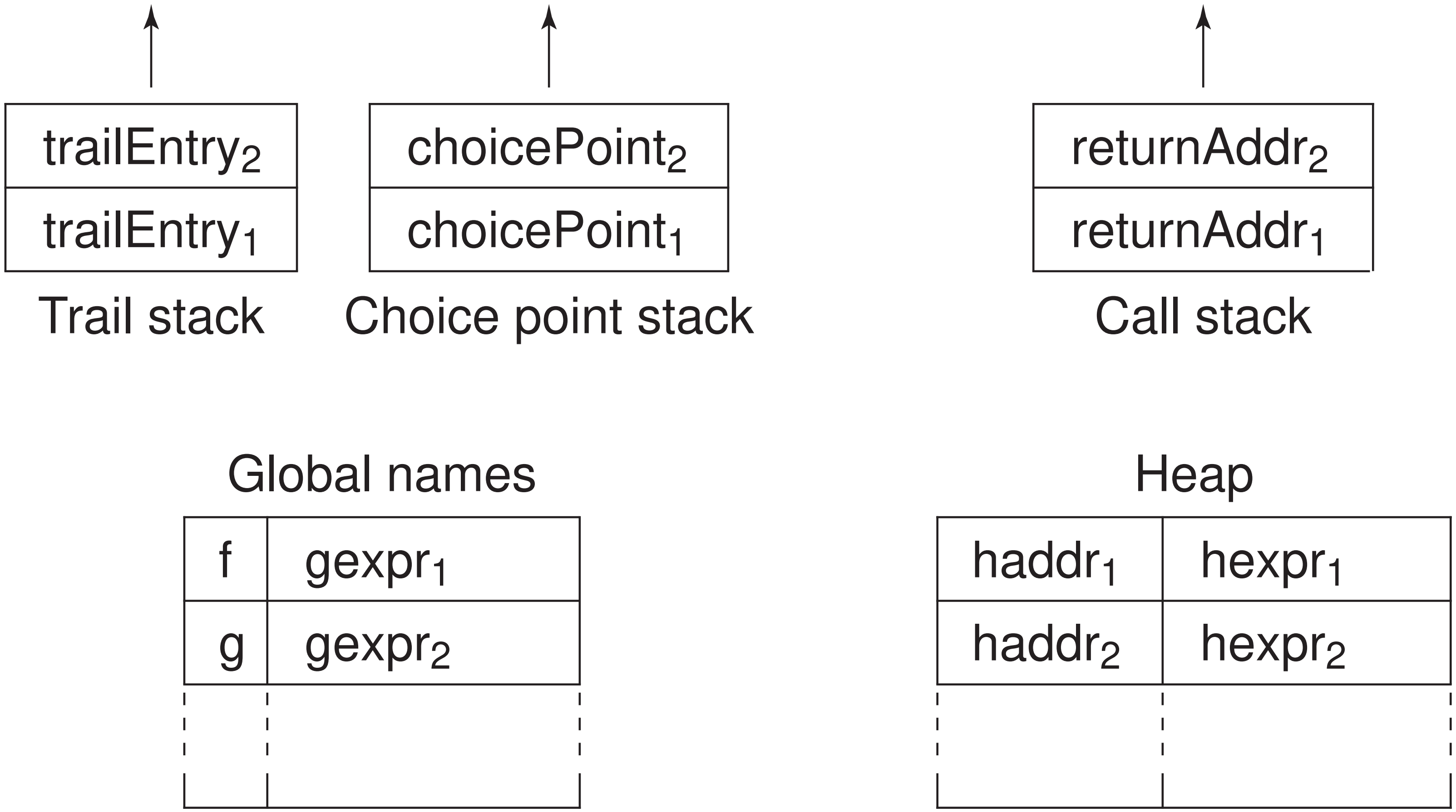
Transformation von caseR:

$$\begin{array}{l} \textbf{caseR}\ v\ \textbf{of} \\ \quad A\ x\ y\ \rightarrow 11 \\ \quad B\ z\quad\ \rightarrow 22 \end{array} \Longrightarrow \begin{array}{l} \textbf{suspendOn}\ v \\ \qquad\qquad \begin{array}{l} \textbf{case}\ v\ \textbf{of} \\ \quad A\ x\ y\ \rightarrow 11 \\ \quad B\ z\quad\ \rightarrow 22 \end{array} \end{array}$$

Transformation von caseF:

$$\begin{array}{l} \textbf{caseF}\ v\ \textbf{of} \\ \quad A\ x\ y\ \rightarrow 11 \\ \quad B\ z\quad\ \rightarrow 22 \end{array} \Longrightarrow \begin{array}{l} \textbf{ifFree}\ v\ (\textbf{or}\ (\textbf{guard}\ (v\ =:=\ A\ x\ y)\ 11) \\ \qquad\qquad\quad (\textbf{guard}\ (v\ =:=\ B\ z)\quad 22)) \\ \qquad\quad \begin{array}{l} \textbf{case}\ v\ \textbf{of} \\ \quad A\ x\ y\ \rightarrow 11 \\ \quad B\ z\quad\ \rightarrow 22 \end{array} \end{array}$$

Auswertung: Narrowing



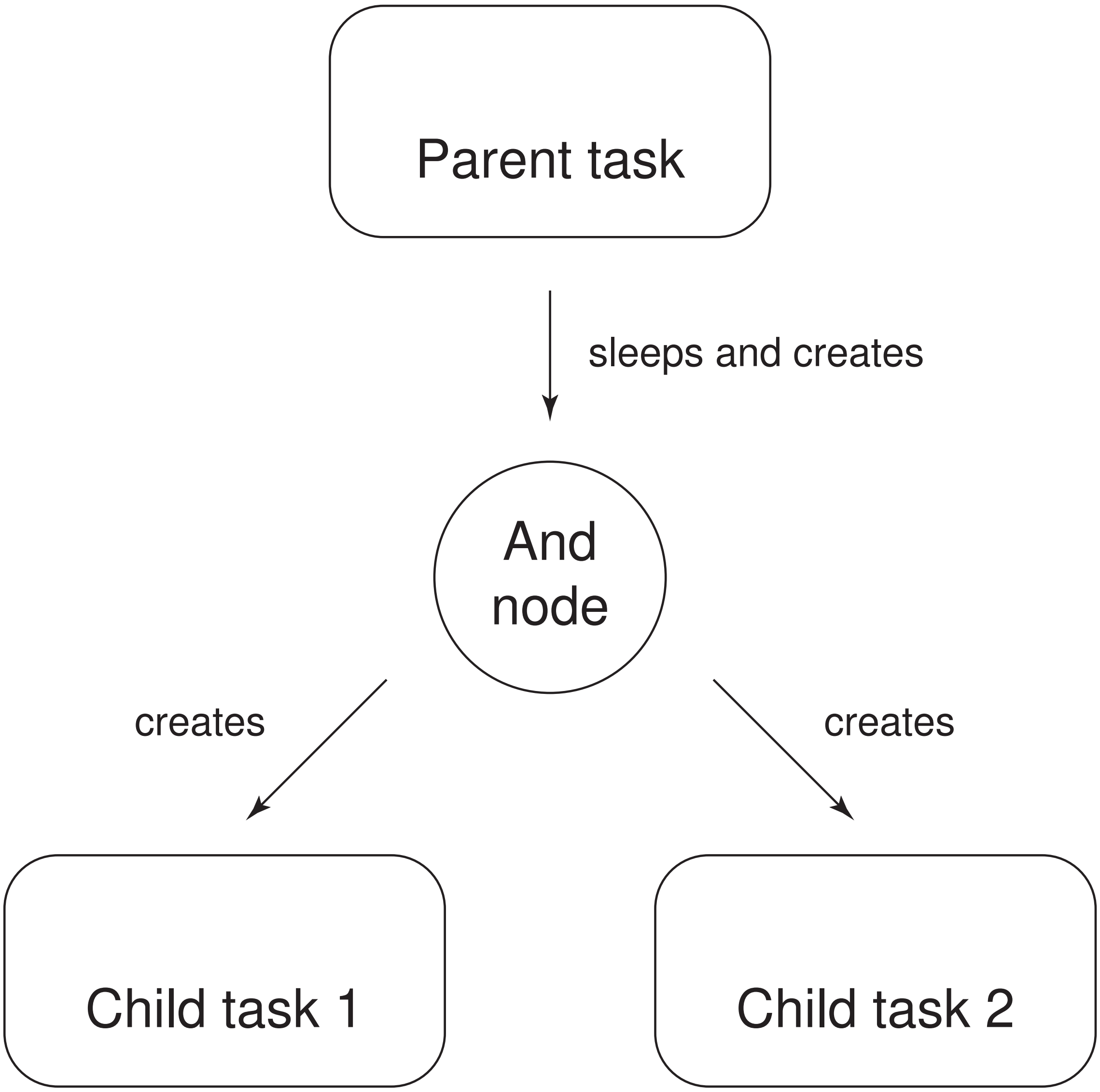
Die virtuelle Maschine

Backtracking

Ein Choice-Point enthält:

- Eine Liste der Alternativen (als Adressen)
- Einen Trail-Zeiger
- Den gesamten Call-Stack

Auswertung: Residuierung

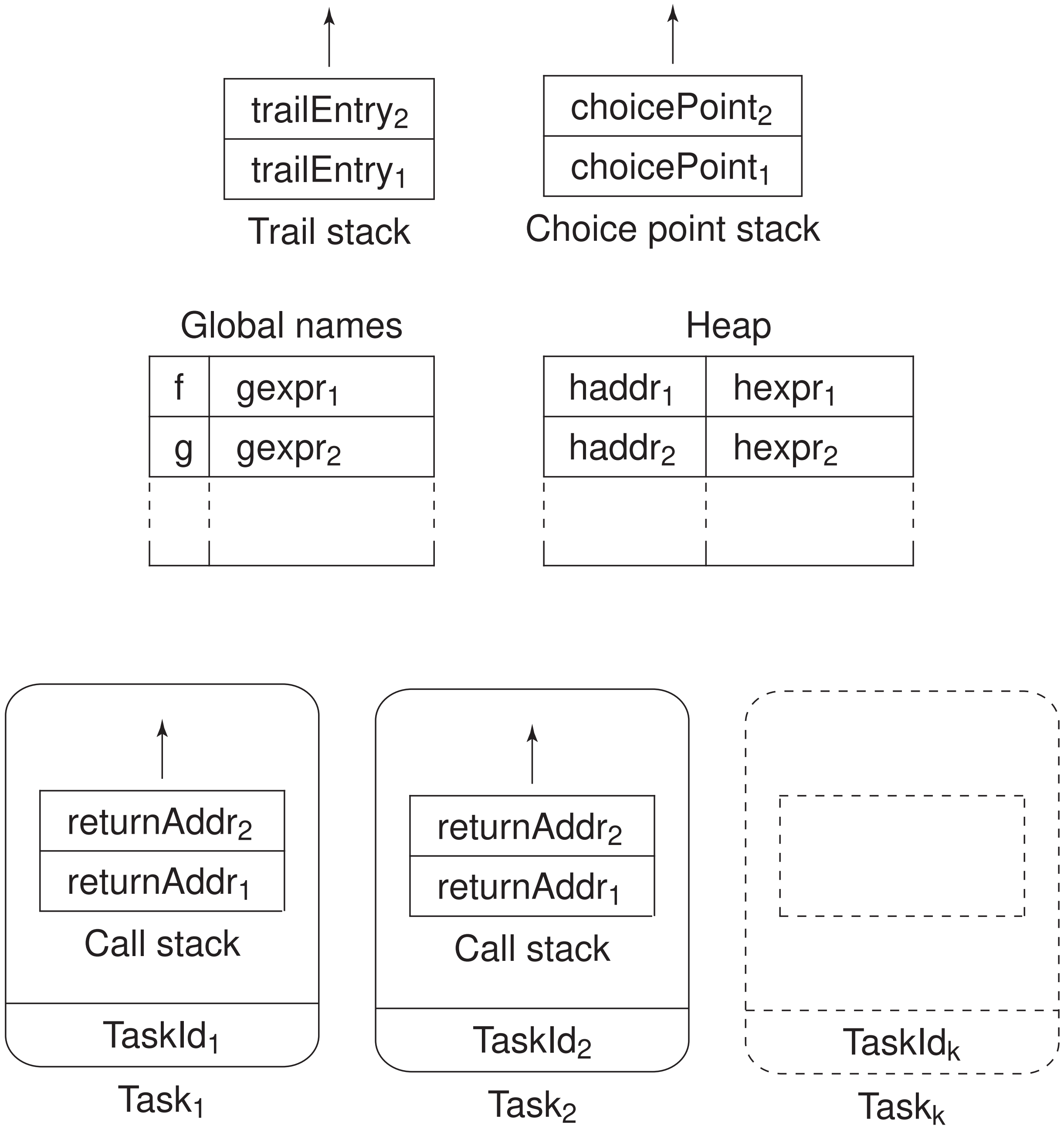


Auswertung einer Konjunktion von Constraints (c_1 & c_2)

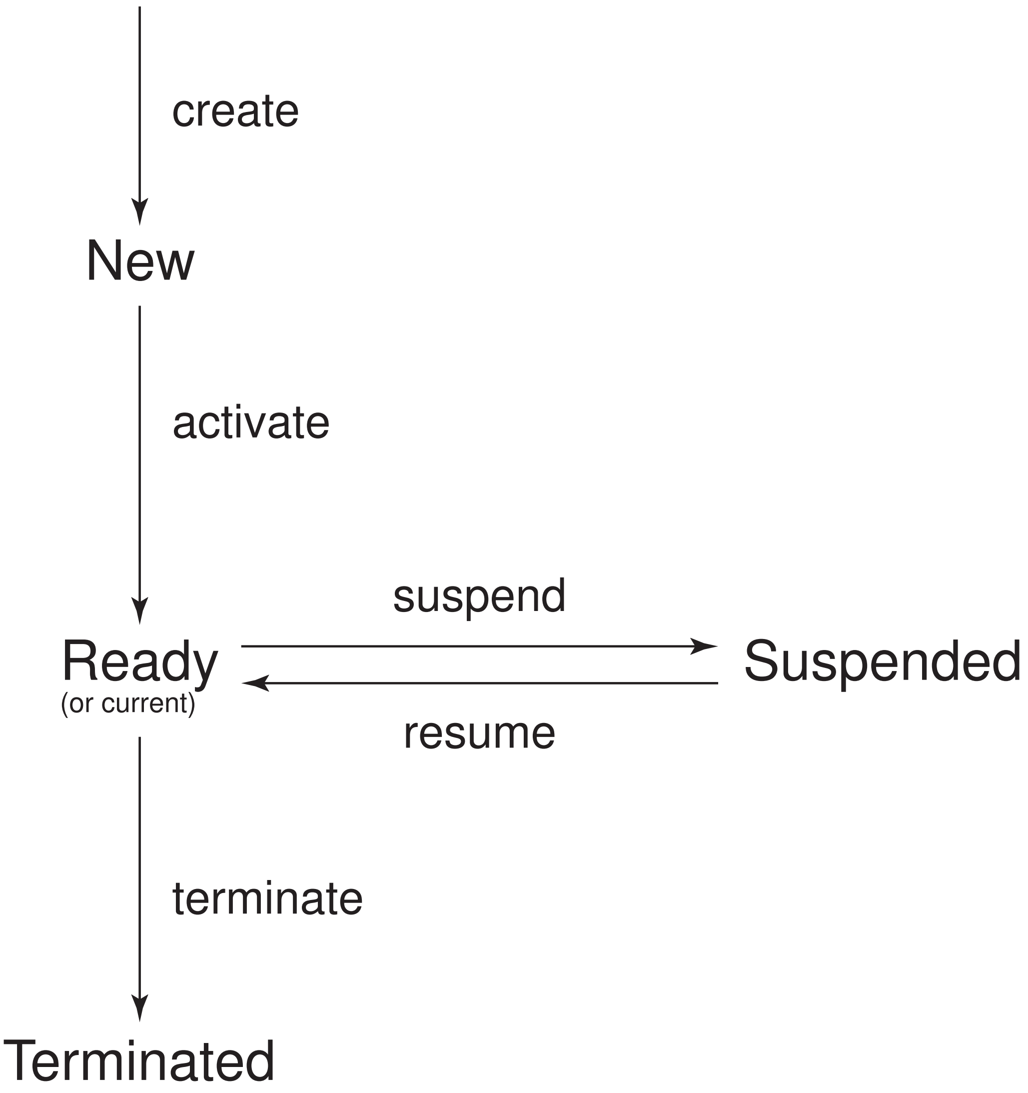
$\mathbf{data}\ \mathit{Task} \ = \ \mathit{Task}\ \mathit{TaskId}\ \mathit{CallStack}$

$\mathbf{data}\ \mathit{AndNode} \ = \ \mathit{AndNode}\ \mathit{AndNodeId}\ \mathit{TaskId}\ \mathit{Int}$

Abstrakte Maschine für Residuierung



Task-Zustände



Backtracking bei mehreren Tasks

Neu in einem Choice-Point: ID seines Tasks.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{TrailEntry} & = & \mathit{THeap}\ \mathit{HeapPtr}\ \mathit{Expr} \\
 & | & \mathit{TTaskCreation}\ \mathit{TaskId} \\
 & | & \mathit{TTaskActivation}\ \mathit{TaskId}\ \mathit{TaskQueueEntry} \\
 & | & \mathit{TAndAllocation}\ \mathit{AndNodeId} \\
 & | & \mathit{TAndDisposal}\ \mathit{AndNode} \\
 & | & \mathit{TSuspensionCount}\ \mathit{AndNodeId}\ \mathit{ChildCount} \\
 & | & \mathit{TSuspension}\ \mathit{TaskId} \\
 & | & \mathit{TResumption}\ \mathit{TaskId}\ \mathit{TaskQueueEntry} \\
 & | & \mathit{TSuspListAddition}\ \mathit{HeapPtr}\ \mathit{TaskId} \\
 & | & \mathit{TBindFreeVar}\ \mathit{HeapPtr}\ [\mathit{TaskId}] \\
 & | & \mathit{TTaskTermination}\ \mathit{TaskId}
\end{array}
$$

Ausblick

- Verbesserung der Entwicklungsumgebung: Debugging etc.
- Andere Auswertungs-Ansätze: Echte Parallelität, Oder-Parallelität.
- Operationale Semantik des Curry-Report nicht gut geeignet für Implementierungen, aber siehe Sacher-Diplomarbeit.