Lecture 13: Inductive Program Synthesis

Cognitive Systems II - Machine Learning

WS 2005/2006

Part III: Learning Programs and Strategies

Functional Programs, Program Schemes, Traces, Folding

Automated Programming

- AI and Software Engineering: Knowledge-based Software-Engineering
  automation/assistance for all parts of the software development process (specification acquisition, code generation, debugging, testing, verification)
- Code Generation: Automated Programming (Auto*magic* programming)
- History: Compiler languages as automated programming systems (Communcations of the ACM, 1958)
- Deductive and transformational approaches
- inductive approaches: learning from I/O examples (or traces), maybe with background knowledge, additional constraints, output or performace evaluation functions

Deductive Program Synthesis

complete, formal (not executable) specification

last(l) $\Leftarrow$ *find z such that for some y, l = y* $\circ$ *[z]*
where islist(l) and l $\neq$ *[ ]*
(Manna & Waldinger)

$$\forall\, x\, \exists\, y\, [Pre(x) \Rightarrow Post(x, y)]$$

$$\forall\, x\, [Pre(x) \Rightarrow Post(x, f(x))]$$

Inductive Program Synthesis

- incomplete specification (examples) $\hookrightarrow$ machine learning approaches going beyond the scope of classification learning
- Classical approaches: inducing functional (Lisp) programs (example-driven, analytical)
- ILP approaches: general methods, special purpose systems (e.g. Dialogs)
- Evolutionary computation (e.g. Adate)

Summers' THESYS System

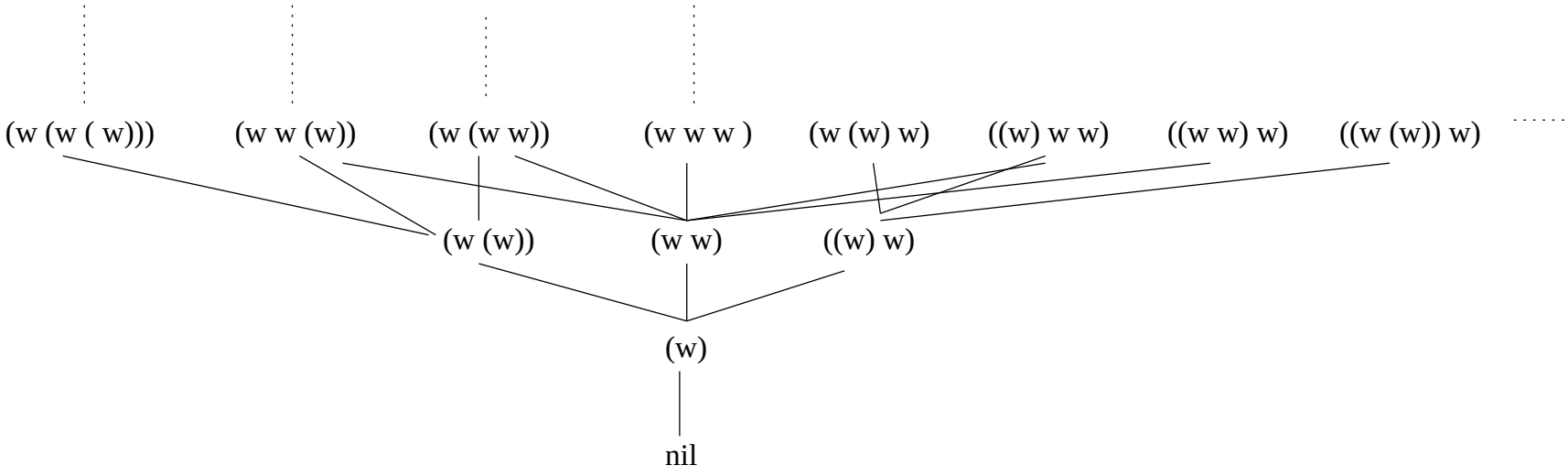
- ph.d. thesis 1976
- Two step approach:
  - Rewriting I/O examples into tests and constructive expressions
  - Recurrence detection
- Theoretical background: unfolding of recursive functions, fixed point semantics (synthesis theorem)

Linear Recursive Program Scheme

$$
\begin{array}{lllll}
f(X) & \leftarrow & p_1(X) & \rightarrow & g_1(X), \\
 & & & \ldots, & \\
 & & p_k(X) & \rightarrow & g_k(X), \\
 & & T & \rightarrow & q(f(b(X)), X)
\end{array}
$$

- base-expressions b: composed from *head* (car) and *tail* (cdr)
- cons-expressions cons over expressions
- constant *nil*
- X input part of example, $p_i(X)$ test, typically only structural characteristic *empty* (atom)
- g_i evaluates to output part of the example cons-expression or base-expression
- $q(x, y)$: combine of recursive call and base-expression over X

CPO over Lists



- originally over S-expressions (po-set, lattice)
- $cons(x, y) \leq cons(u, v)\ iff\ x \leq u\ and\ y \leq v$

Step 1: Rewriting I/O Examples

$\{nil \rightarrow nil,$
$(A) \rightarrow ((A)),$
$(A\ B) \rightarrow ((A)\ (B)),$
$(A\ B\ C) \rightarrow ((A)\ (B)\ (C))\}$

Derived Trace: ("initial program")

$$
\begin{aligned}
F_L(x) \leftarrow\ & (atom(x) \rightarrow && nil,\\
& \ atom(cdr(x)) \rightarrow && cons(x, nil),\\
& \ atom(cddr(x)) \rightarrow && cons(cons(car(x), nil),\\
& && \ cons(cdr(x), nil)),\\
& \ T \rightarrow cons(cons(car(x), nil), cons(cons(cadr(x), nil), && \\
& && \ cons(cddr(x), nil))))
\end{aligned}
$$

Step 1: Algorithm

"semitraces" g_i

$$ST(x_i, y_i) = \begin{cases} (p\ x) if\ y_i \neq\ nil\ \ and \\ y_i = p(x)\ where\ p\ is\ \ a\ \ base\ function \\ nil & if\ y_i = ni \\ (cons\ ST(x_i, head(y_i))\ ST(x_i, tail(y_i))) & otherwise \end{cases}$$

test (predicates) p_i
$PG(x_i, x_{i+1})$: find difference between two inputs
e.g.: (A), (A B) $\hookrightarrow$ tail
test: $empty(tail(X))$

Step 2: Recurrence Detection

$$
\begin{aligned}
F_L(x) \leftarrow\ & (atom(x) \rightarrow && nil,\\
& atom(cdr(x)) \rightarrow && cons(x, nil),\\
& atom(cddr(x)) \rightarrow && cons(cons(car(x), nil),\ cons(cdr(x), nil)),\\
& T \rightarrow cons(cons(car(x), nil), cons(cons(cadr(x), nil),\ cons(cddr(x), nil))))
\end{aligned}
$$

$$
\begin{aligned}
\text{unpack(x)} \leftarrow\ & (\text{atom(x)} \rightarrow \text{nil},\\
& \text{T} \rightarrow \text{u(x)})\\
\text{u(x)} \leftarrow\ & (\text{atom(cdr(x))} \rightarrow \text{cons(x, nil)},\\
& \text{T} \rightarrow \text{cons(cons(car(x), nil), u(cdr(x)))})
\end{aligned}
$$

Step 2: Pattern Matching and Folding

$F_L(x) \leftarrow$ (atom(x) $\rightarrow$ nil,

atom(cdr(x)) $\rightarrow$ cons(x, nil),

atom(cddr(x)) $\rightarrow$ cons(cons(car(x), nil), cons(cdr(x), nil)),

T $\rightarrow$ cons(cons(car(x), nil), cons(cons(cadr(x),nil),

cons(cddr(x),nil))))

Differences:

$p_2(x) = p_1(\text{cdr(x)})$

$p_3(x) = p_2(\text{cdr(x)})$

$p_4(x) = p_3(\text{cdr(x)})$

Recurrences:

$p_1(x) = \text{atom(x)}$

$p_{k+1}(x) = p_k(\text{cdr(x)})$ for $k = 1, 2, 3$

Folding cont.

$F_L(x) \leftarrow$ (atom(x) $\rightarrow$ nil,

atom(cdr(x)) $\rightarrow$ cons(x, nil),

atom(cddr(x)) $\rightarrow$ cons(cons(car(x), nil), cons(cdr(x), nil)),

T $\rightarrow$ cons(cons(car(x), nil), cons(cons(cadr(x),nil),

cons(cddr(x),nil))))

Difference:

$f_2(x) = \mathit{cons}(x, f_1(x))$

Recurrence:

$f_1(x) = \mathit{nil}$

$f_2(x) = \mathrm{cons}(x, f_1(x))$

Folding cont.

$F_L(x) \leftarrow$ (atom(x) $\rightarrow$ nil,
atom(cdr(x)) $\rightarrow$ cons(x, nil),
atom(cddr(x)) $\rightarrow$ cons(cons(car(x), nil), cons(cdr(x), nil)),
T $\rightarrow$ cons(cons(car(x), nil), cons(cons(cadr(x),nil),
cons(cddr(x),nil))))

Difference:

$f_3(x) =$ *cons(cons(car(x),nil)*, $f_2(cdr(x))$)

$f_4(x) =$ *cons(cons(car(x),nil)*, $f_3(cdr(x))$)

Recurrence:

$f_2(x) =$ *cons*$(x, f_1(x))$

$f_{k+1}(x) =$ *cons(cons(car(x),nil)*, f_k(cdr(x))) for $k = 2, 3$

Folding of Finite Terms

- If a recurrence relation is detected in finite terms
  $p_{k+1}(x) = p_k(\textit{cdr(x)})$ for $k = 1, 2, 3$
  $f_1(x) = \textit{nil}$
  $f_2(x) = \textit{cons}(x, f_1(x))$
  $f_{k+1}(x) = \textit{cons(cons(car(x),nil)}, f_k(cdr(x) =)$ for $k = 2, 3$
- they can be folded into a recursive function

  u(x) ← (atom(cdr(x)) → cons(x, nil),

  T → cons(cons(car(x), nil), u(cdr(x))))
- Relation between unfolding (i.e. evaluation of recursive function) and folding (inductive generalization)

Summers' Synthesis-Theorem (1)

- Unfolding: syntactic replacement of function call by function body with parameter substitutions
- Exploiting of the relation between recursive function and its unfolding
- Term as kth approximation of searched-for function $\mathbf{F}(x)$

leene-Sequence of Unfoldings for *unpack(*

defined for no input $U^0 \leftarrow \Omega$

leene-Sequence of Unfoldings for *unpack(x*

defined for no input $U^0 \leftarrow \Omega$

defined for empty list $U^1 \leftarrow (\text{atom(x)} \rightarrow \text{nil}, T \rightarrow \Omega)$

leene-Sequence of Unfoldings for *unpack(x*

defined for no input $U^0 \leftarrow \Omega$

defined for empty list $U^1 \leftarrow (\text{atom(x)} \rightarrow \text{nil}, T \rightarrow \Omega)$

defined for empty and one-element lists

$$U^2 \leftarrow (\text{atom(x)} \rightarrow \text{nil}, \text{atom(cdr(x))} \rightarrow \text{cons(x, nil)}, T \rightarrow \Omega)$$

leene-Sequence of Unfoldings for *unpack(x*

defined for no input $U^0 \leftarrow \Omega$

defined for empty list $U^1 \leftarrow (\text{atom(x)} \rightarrow \text{nil}, T \rightarrow \Omega)$

defined for empty and one-element lists

$$U^2 \leftarrow (\text{atom(x)} \rightarrow \text{nil}, \text{atom(cdr(x))} \rightarrow \text{cons(x, nil)}, T \rightarrow \Omega)$$

...

defined for lists of up to n elements

Summers' Synthesis-Theorem (2)

- ↪ Complete partial order over set of function approximations (Kleene sequence)
- ↪ Supremum of sequence of function approximations is searched-for function (fixed point semantics)

Summers' Synthesis-Theorem (2)

- ↪ Complete partial order over set of function approximations (Kleene sequence)
- ↪ Supremum of sequence of function approximations is searched-for function (fixed point semantics)

If a program constructed over examples is viewed as kth unfolding of an unknown recursive function
and if recurrence relations in accordance with some recursive program scheme can be detected by some search strategy
then the recursive generalization can be constructed!
(unique generalization?, characteristics of the generalization?)

IGOR: A more powerful approach

- Generalization of Summers
- More general scheme: set of all functions programs without higher-oder functions, without mutual recursion, without nested recursion

Recursive Program Schemes

Given

- a signature Σ,
- a set of *function variables* $\Phi = \{G_1, \ldots, G_n\}$ with arity $\alpha(G_i) > 0$ for all $i \in [1; n]$, and
- a set of variables X with $\Sigma \cap \Phi \cap X = \emptyset$,

then a *recursive program scheme (RPS)* $\mathcal{S}$ is a pair $\mathcal{S} = (\mathcal{G}, m)$ where

1. $\mathcal{G}$ is a set of n equations $\{G_1(x_1, \ldots, x_{\alpha(G_1)}) = t_1, \ldots, G_n(x_1, \ldots, x_{\alpha(G_n)}) = t_n\}$, where the t_i are terms with respect to the signature $\Sigma \cup \Phi$ and variables X and where
2. $m \in [1; n]$ is a number indicating the main equation.

Example

- $\Sigma = \{\text{if}, \text{empty}, \text{cons}, \text{hd}, \text{tl}, []\}$,
- $\Phi = \{G_1, G_2\}$, $G_1 = \text{lasts}, G_2 = \text{last}$,
  $\alpha(\text{lasts}) = \alpha(\text{last}) = 1$,
- $X = \{x\}$,
- $\mathcal{G} =$

$$\{\ \text{lasts}(x) = \text{if}(\text{empty}(x), [],$$
$$\text{cons}(\text{hd}(\text{last}(\text{hd}(x))), \text{lasts}(\text{tl}(x)))),$$
$$\text{last}(x) = \text{if}(\text{empty}(\text{tl}(x)), x, \text{last}(\text{tl}(x)))\ \},$$

- $m = 1$.

The Generalization Algorithm

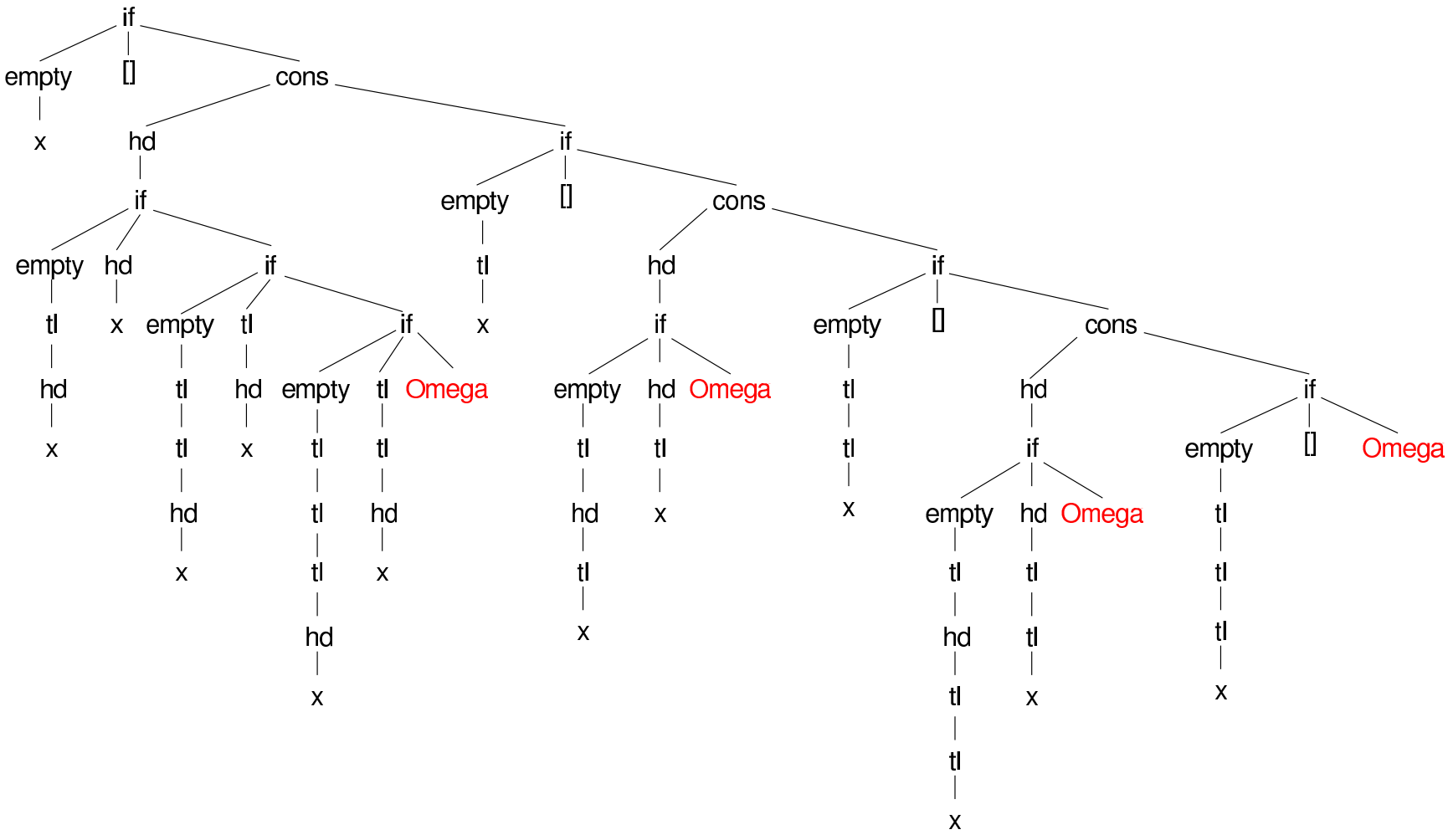
Each recursive equation is induced in three steps which are organized in a divide-and-conquer pattern:

- Divide-phase: 1. Search for a segmentation, i.e. determining recursion and subscheme positions for each equation.
- Conquer-phase: 2. Calculation of equation bodies followed by 3. calculation of substitution terms.
- Only segmentation utilizes search.
- Segmentation can be seen as search through a hypothesis space
- Correspondingly, calculation of bodies and substitution terms can be seen as *constructive* goal test. Success results in a completed RPS, failure initiates a backtrack to segmentation.

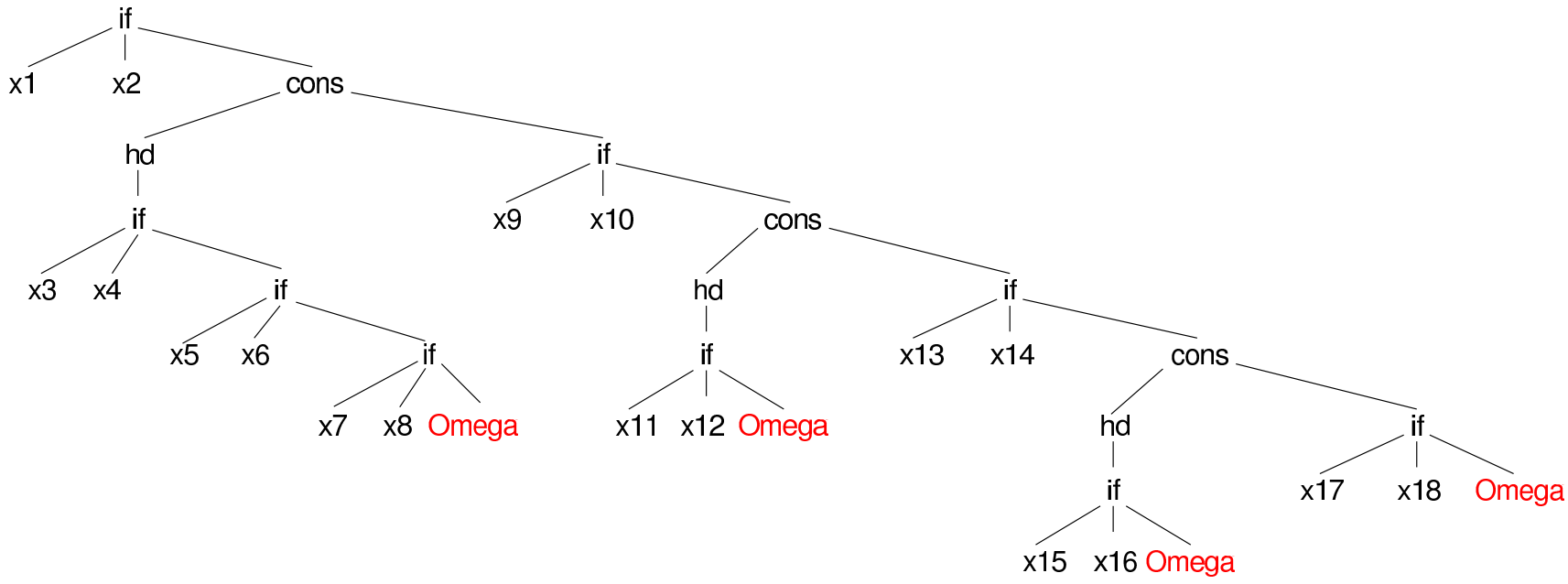
1. Step: Segmentation

- Only positions on pathes leading to some Ω come into question as recursion positions (substantial narrowing of search space).
- All Ωs not explained by recursion positions imply *subscheme positions*, i.e. calls of *further* recursive equations.
- Recursion and subscheme positions determine an equation body skeleton, which must be *equal* on each unfolding position in each initial term.
- (Greedy) search strategy for recursion positions is:
  1. Explanation of as much as possible Ωs by recursion positions.
  2. As small as possible recursion positions (i.e. as small as possible equation bodies).

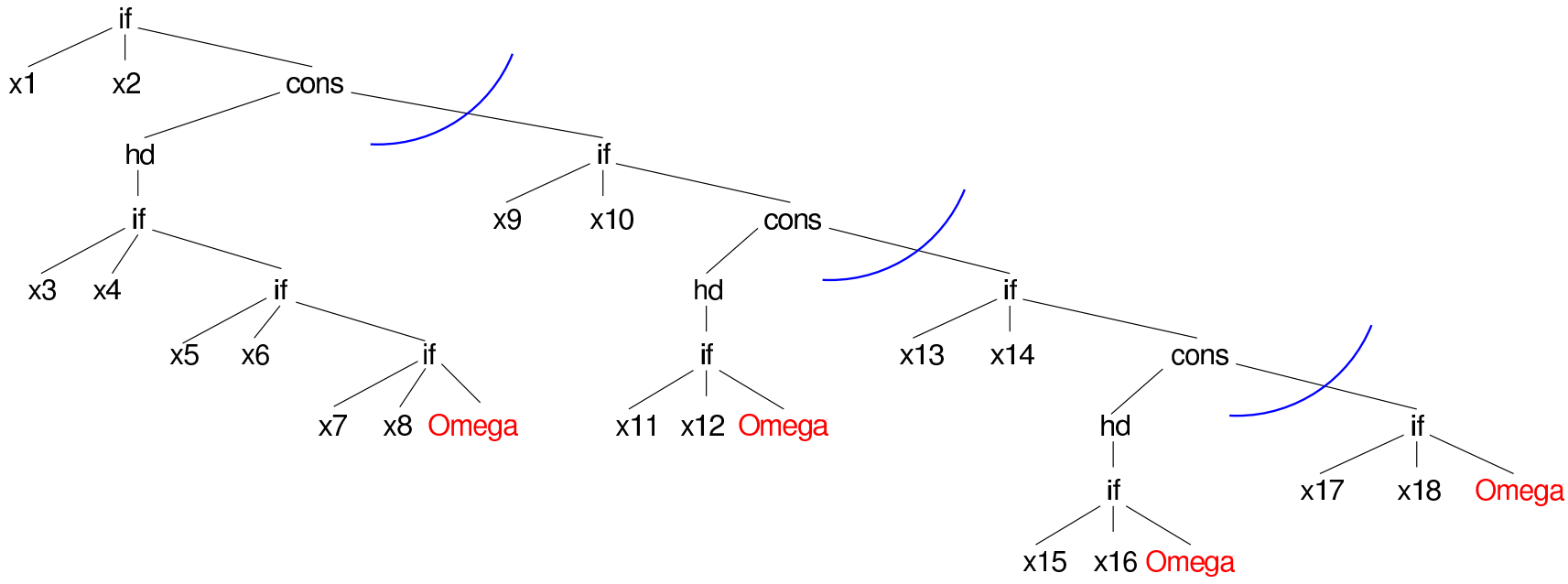
Segmentation Example



Segmentation Example

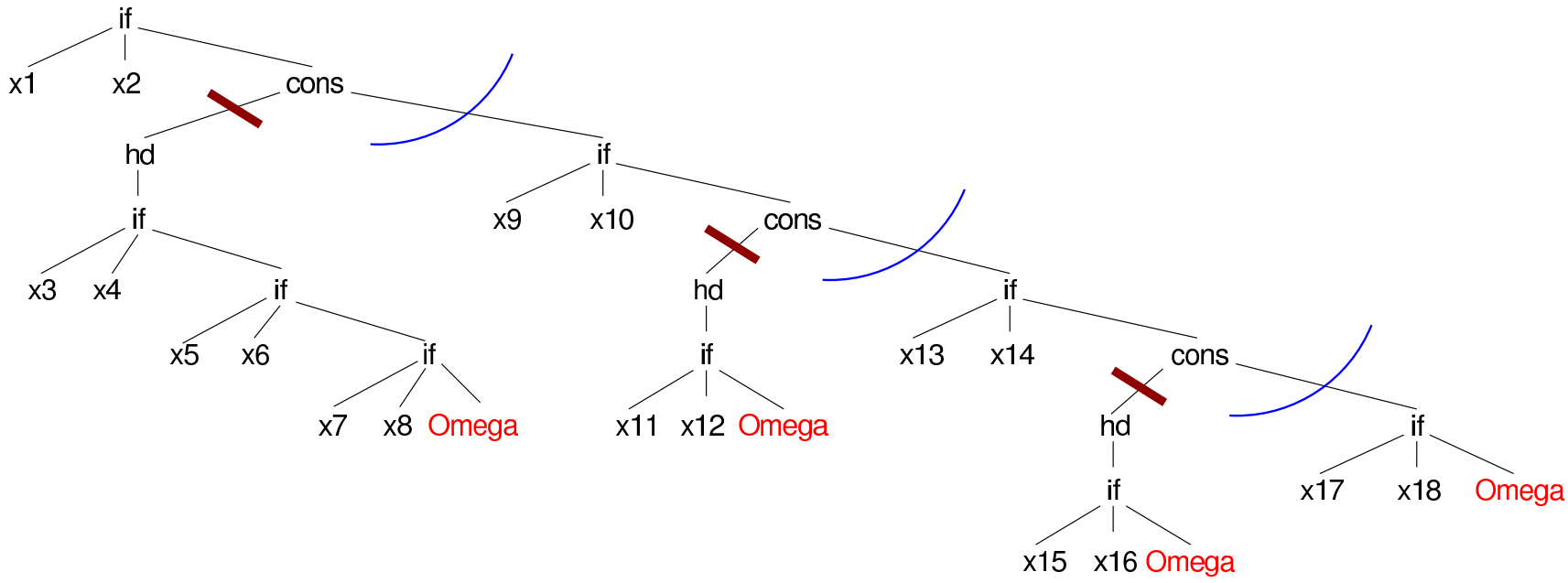


Segmentation Example



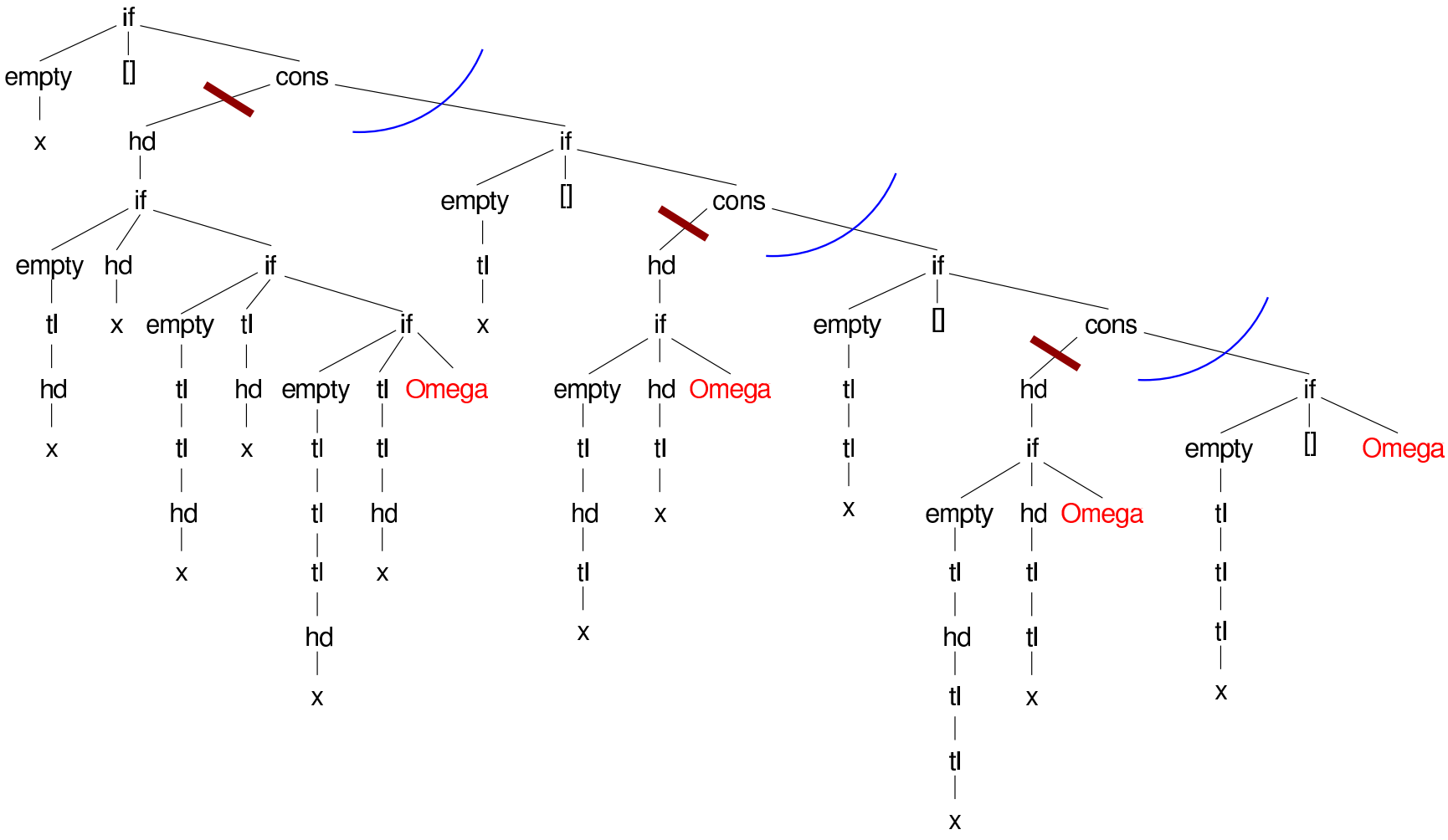
Recursion position: 32

Segmentation Example



Recursion position: 32
Subscheme position: 31

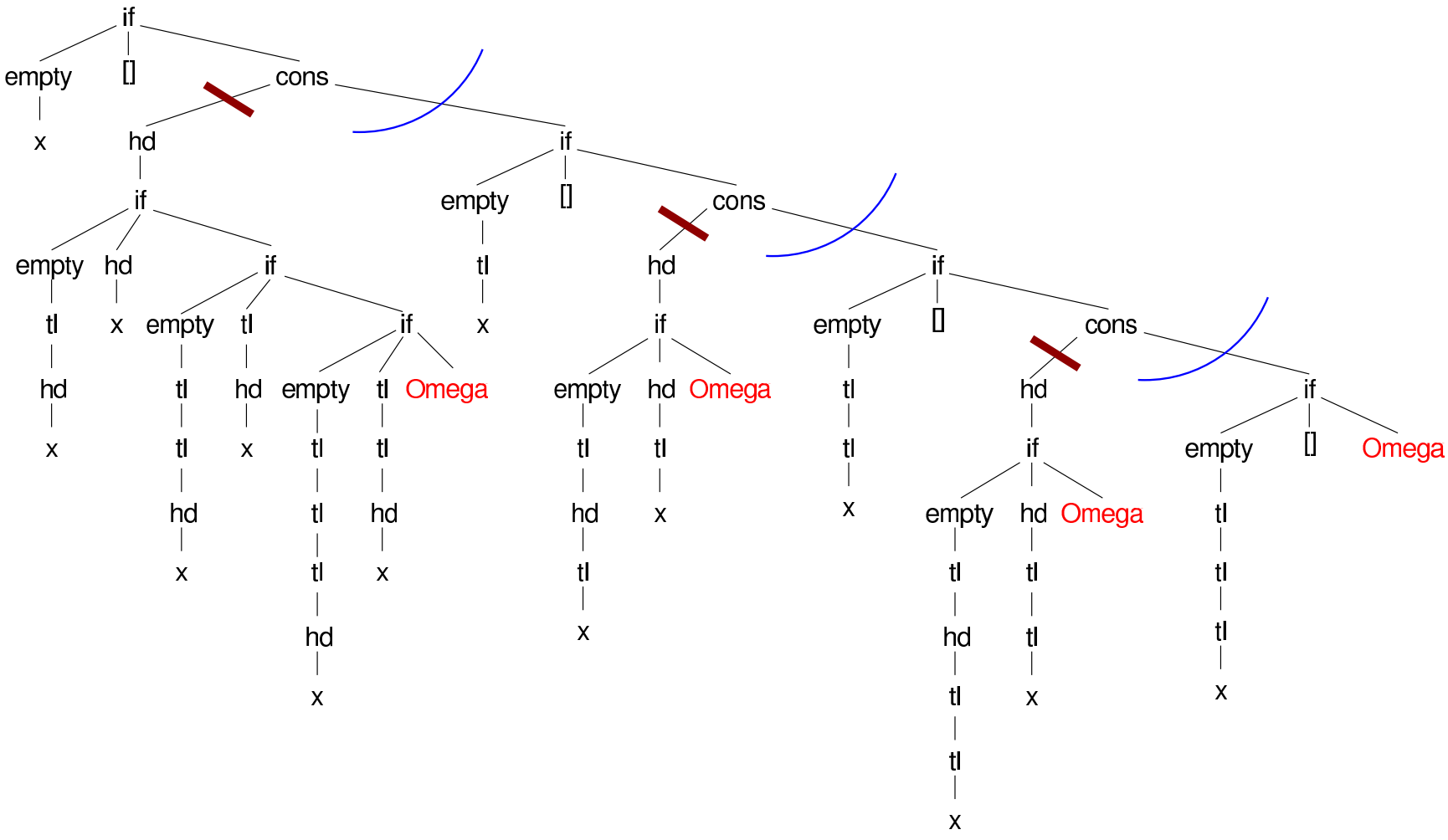
Segmentation Example



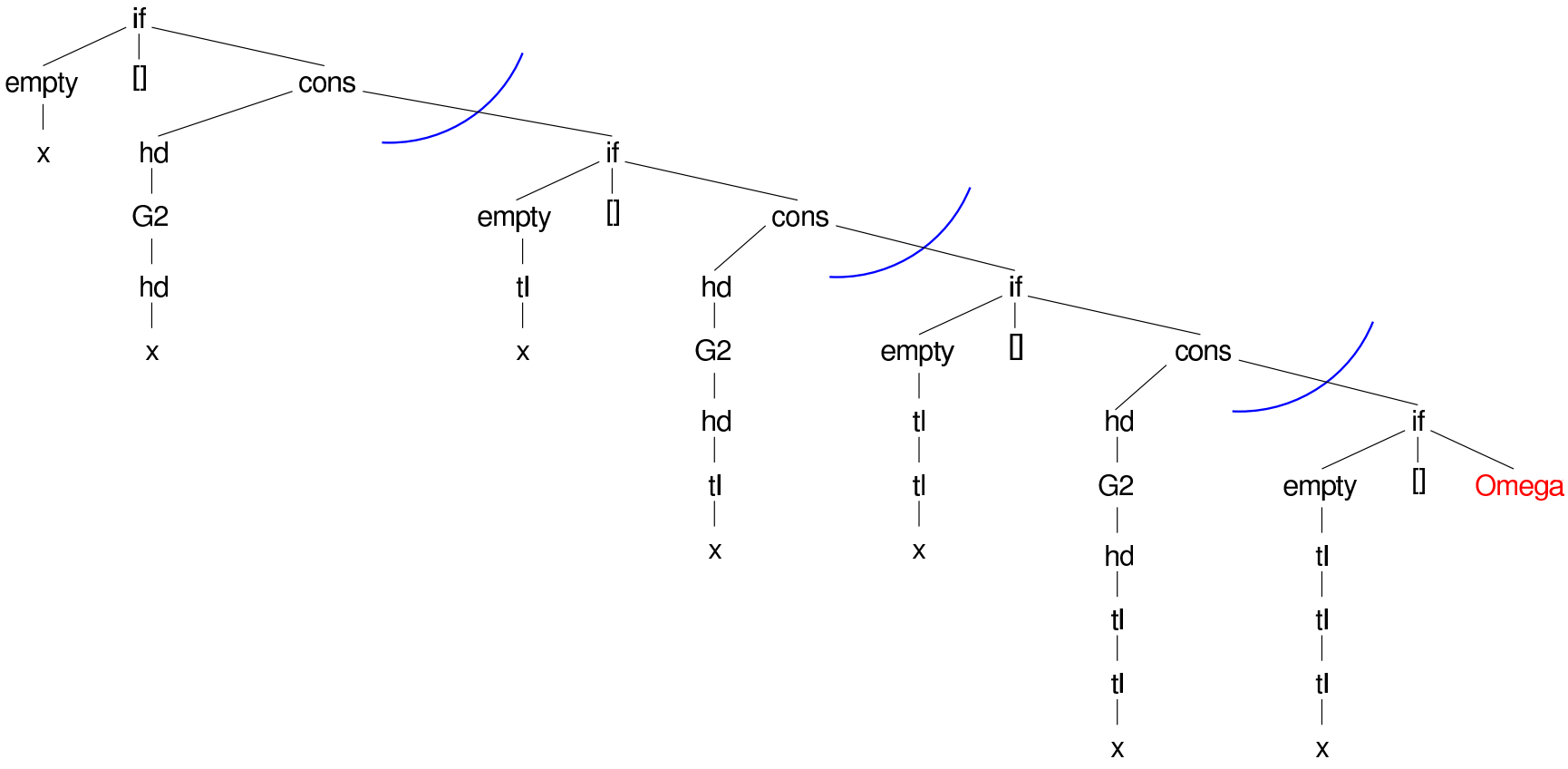
2. Step: Calculation of Equation Bodies

- Initial terms can be explained by *one* recursive equation.
- All initial terms are splitted at unfolding positions into *segments*: $\{t|_u[R \leftarrow G] \mid u \in U \cap \mathbf{pos}(t), R \subset t|_u\}$
- Segments are instantiations of the (incompl.) equation body.
- Positions which are equally labeled in each segment are assumed to belong to this equation body.
- Positions which are differently labeled in at least two segments can only belong to (different) variable instantiations.
- The (incomplete) equation body is defined as the most specific maximal pattern of all segments.

Calculation of Bodies – Example

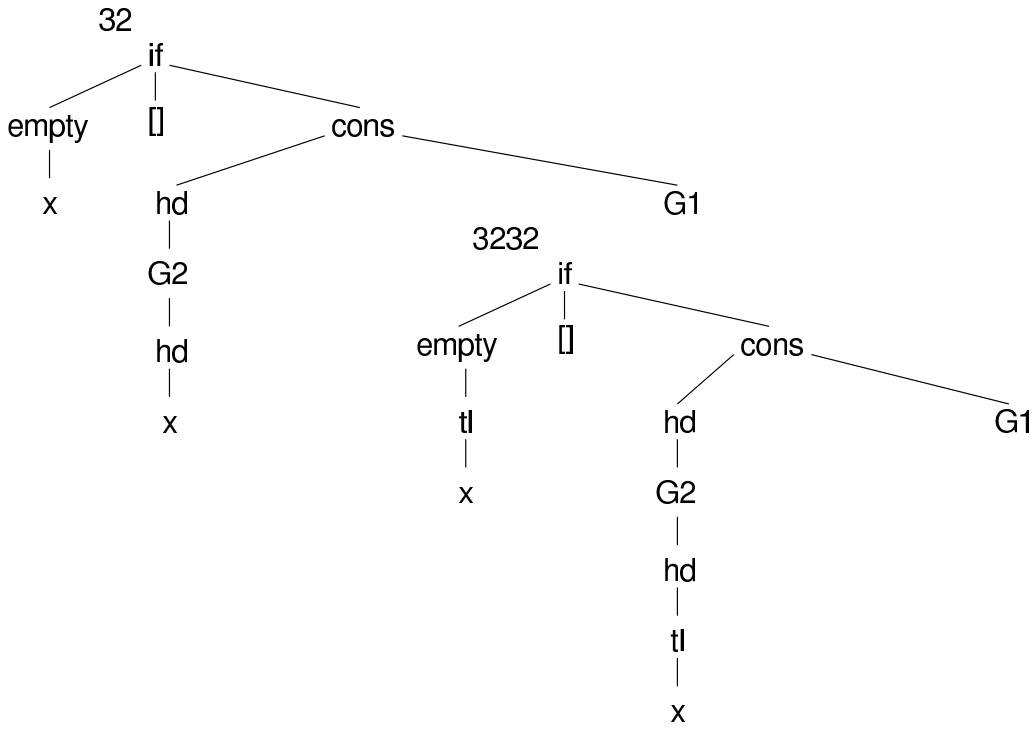


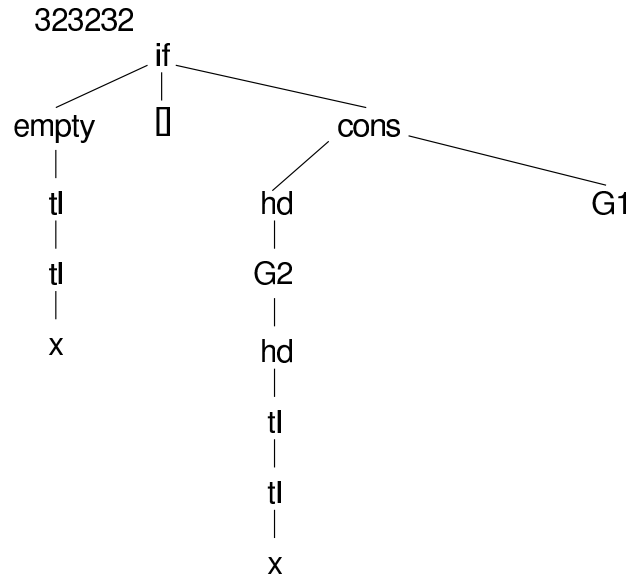
Calculation of Bodies – Example



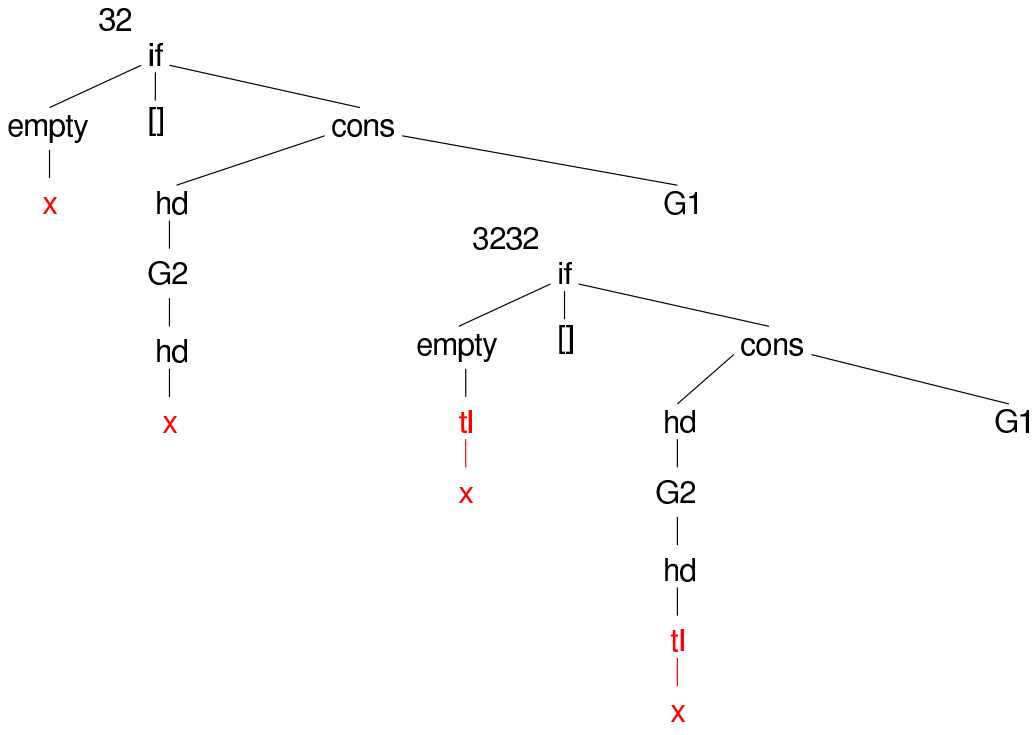
$$G_2(x) = \text{if}(\text{empty}(\text{tl}(x)), x, G_2(\text{tl}(x)))$$

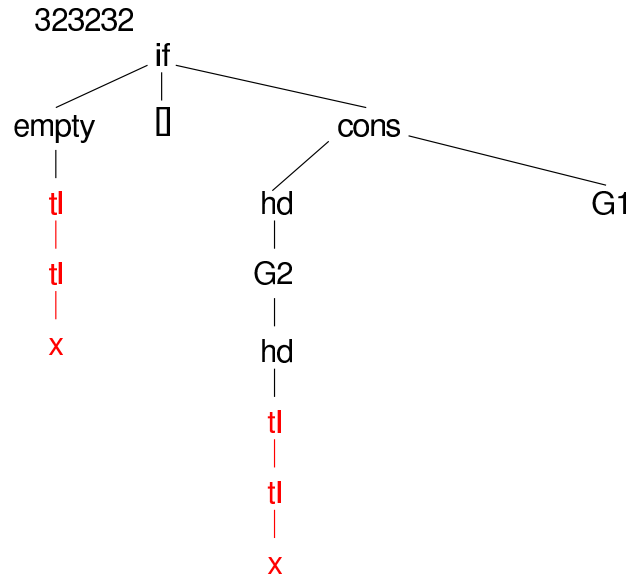
Calculation of Bodies – Example



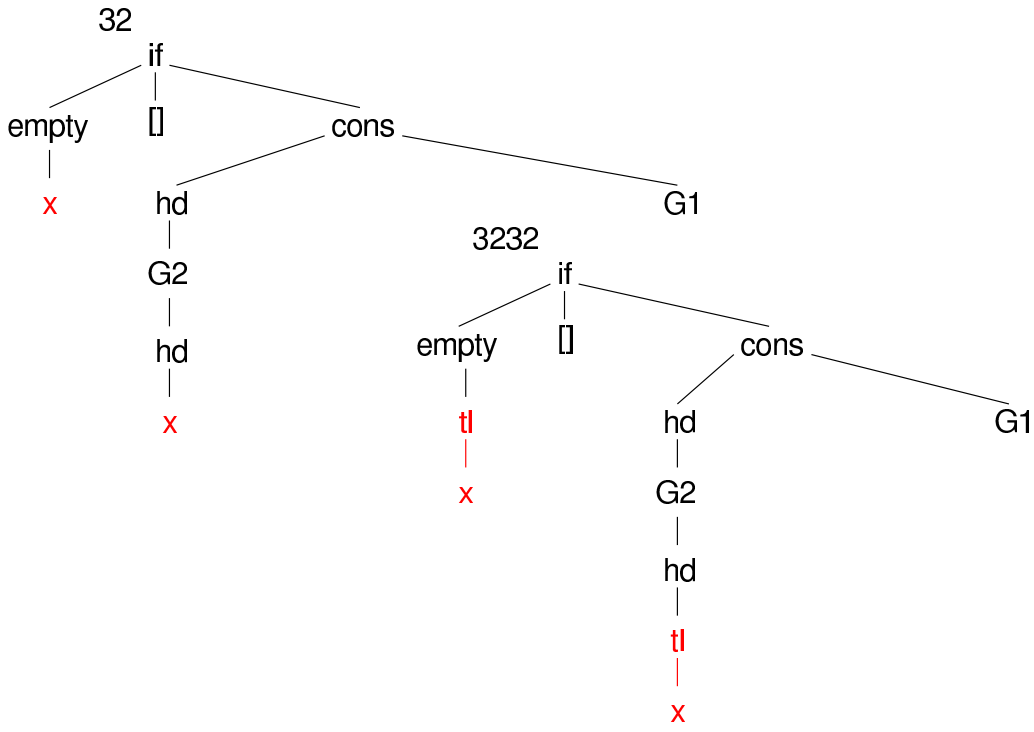


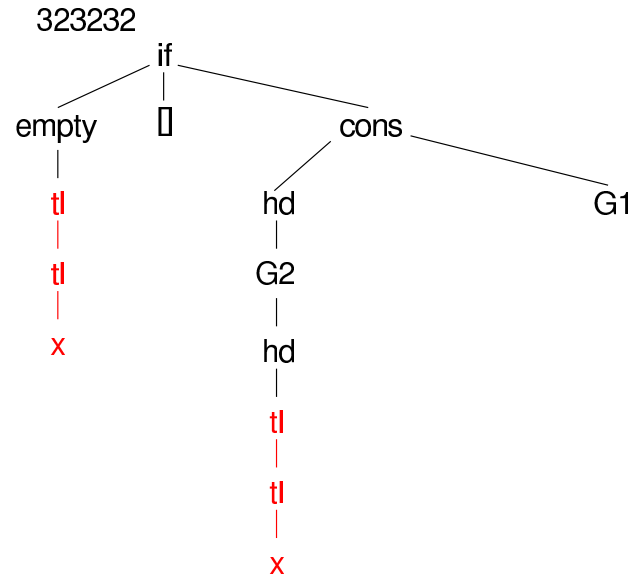
Calculation of Bodies – Example





Calculation of Bodies – Example





(Incomplete) body of equation G_1:

$$\text{if}(\text{empty}(x), [], \text{cons}(\text{hd}(G_2(\text{hd}(x))), G_1))$$

3. Step: Calculation of Substitution Terms

- As result of antiunifying the segments for determining the body, variables and their instantiations in unfoldings are given.
- Recall the inductive structure of instantiations in unfoldings:
  1. $\beta_\epsilon = \beta$
  2. $\beta_{ur} = \sigma_r \, \beta_u$, for recursion position r and unfolding position u

  By means of this characterization we can calculate the substitution terms σ_r based on the instantiations in unfoldings β_u.

3. Step: Calculation of Substitution Terms

- Special case: *Hidden variables*. vars not occuring in an (incomplete) equation body, but only in substitution terms. Thus, neither hidden variables, nor their instantiations in unfoldings are given as result from antiunifying the segments.

f(x, y, z) = if eq0(x) then y else +(x, f(p(x), z, s(y)))

Calculation of Substitution Terms continue

r denotes recursion, u denotes unfolding positions:
From $\beta_\epsilon = \beta$, $\beta_{ur} = \sigma_r\, \beta_u$ follows:

1. If $(x_i\, \sigma_r)|_v = x_j$ then for all $u \in U$ hold $(x_i \beta_{ur})|_v = x_j \beta_u$.
2. If $(x_i\, \sigma_r)|_v = f((x_i\, \sigma_r)|_{v1}, \ldots, (x_i\, \sigma_r)|_{vn}), f \in \Sigma, \alpha(f) = n$ then for all $u \in U$ hold $\mathbf{node}(x_i \beta_{ur}, v) = f$.

- Reading the implications in reverse direction yields a basic algorithm for calculating the substitution terms.
- If none of the two conclusions is given, a hidden variable x_j is hypothesized. Reading implication one in regular direction yields the unfolding instantiations of the hidden variable.
- For each infered substitution term, substitution uniqueness have to be checked.

alculation of Substitution Terms – Examp

- Only recursion position: 32, only variable: x, unfolding instantiations: $x\beta_{\epsilon} = x$, $x\beta_{32} = \mathrm{tl}(x)$, $x\beta_{3232} = \mathrm{tl}(\mathrm{tl}(x))$, searched for: $x\,\sigma_{32}$
- Starting with position ϵ:
  1. There is no u, such that holds: $x\beta_{u32} = x\beta_u$
  2. Yet for all u hold: $\mathbf{node}(x\beta_{u32}, \epsilon) = \mathrm{tl}$

  Thus $x\,\sigma_{32} = \mathrm{tl}((x\,\sigma_{32})|_1)$
- It remains position 1:
  1. For all u hold: $(x\beta_{u32})|_1 = x\beta_u$

  Thus $(x\,\sigma_{32})|_1 = x$, no remaining positions.
- Calculated substitution term: $x\,\sigma_{32} = \mathrm{tl}(x)$
- σ_{32} is substitution unique.

Putting All Parts Together

- Found recursion position: 32, resulting subscheme position: 33
- Infered equation G_2:
  $G_2(x) = if(empty(tl(x)), x, G_2(tl(x)))$
- Incomplete body of G_1:
  $if(empty(x), [], cons(hd(G_2(hd(x))), G_1))$
- Infered substitution term for G_1: $x\,\sigma_{32} = tl(x)$
- Infered equation G_1:
  $G_1(x) = if(empty(x), [], cons(hd(G_2(hd(x))), G_1(tl(x))))$
- Both equations together are the intended $lasts$-RPS.

Generation of Initial Terms from Example

- Depends on domain knowledge.
- Different approaches:
  - Explanation based on knowledge of inductive datatypes
  - AI planning
  - Genetic Programming