Scrap your boilerplate: Prologically

Ralf Lämmel
Universität Koblenz-Landau


+

+


Important abbreviations

Please write them down!

- SYB = Scrap Your Boilerplate
- SingYB = Scrap[p]ing Your Boilerplate

Quiz: Do you know functional programming with bananas?

- ☐ Functional Programming with Bananas …
- ☐ Bananas in Space: …
- ☐ Boxes go bananas: …
- ☐ Dealing with large bananas
- ☐ Banana Algebra: ….

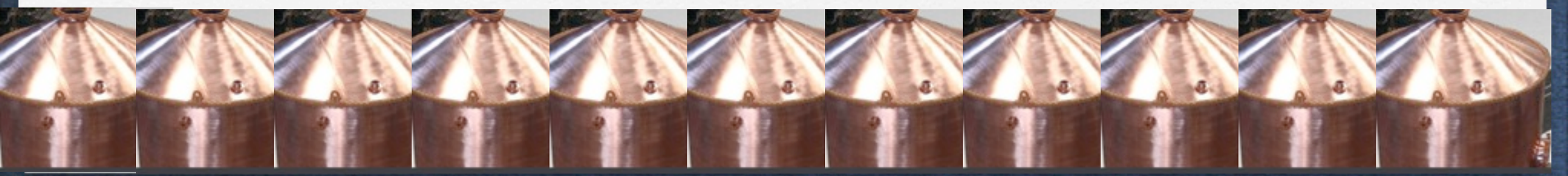
Related papers

SingYB

= Way of dealing w/ large bananas

Related papers

- Scrap your boilerplate
- Scrap your boilerplate systematically
- Scrap more boilerplate
- "Scrap Your Boilerplate" Reloaded
- Scrap your boilerplate with class
- "Scrap Your Boilerplate" Revolutions
- Scrap your nameplate
- Scrap your boilerplate with XPath
- Scrap++
- Uniform boilerplate and list processing



Semi-formal definition of SYB

SYB is a programming technique for ... totaling, increasing, cutting salaries in a company -- either for all employees or for managers only.

```
<company>
   <department deptno="1">
      <label>Leadership department</label>
      <manager>
         <name>Merlin</name>
         <salary>9999</salary>
      </manager>
      <member>
         <department deptno="11">
            <label>Planning department</label>
            <manager>
               <name>Douglas</name>
               <salary>4200</salary>
            </manager>
            <member>
               <employee>
                  <name>Peter</name>
                  <salary>4241</salary>
               </employee>
            </member>
         </department>
      </member>

      . . .
```

MSFT goes Prolog

```
company([

  topdept(
    name('Human Resources'),
    manager(
      name('Lisa'),
      salary(123456)),
    []),

topdept(
  name('Development'),
  manager(
    name('Anders'),
    salary(43210)),
    [
  subdept(
    name('Visual Basic'),
    manager(
      name('Amanda'),
      salary(8888)),
      []),
  subdept(
    name('Visual C#'),
    manager(
      name('Erik'),
      salary(4444)),
      []) ])])
```

Cutting salaries in Prolog ***with boilerplate (code)***

```
cutSalary(company(L1),company(L2)) :-
  map(cutSalary,L1,L2).
cutSalary(topdept(N0,M1,L1),topdept(N0,M2,L2)) :-
  cutSalary(M1,M2),
  map(cutSalary,L1,L2).
cutSalary(manager(N0,S1),manager(N0,S2)) :-
  cutSalary(S1,S2).
cutSalary(subdept(N0,M1,L1),subdept(N0,M2,L2)) :-
  cutSalary(M1,M2),
  map(cutSalary,L1,L2).
cutSalary(employee(N0,S1),employee(N0,S2)) :-
  cutSalary(S1,S2).
cutSalary(salary(S1),salary(S2)) :-
  S2 is S1 / 2.
```

Cutting salaries in XSLT ***w/o boilerplate***

```
<xsl:stylesheet>

  <xsl:template match="salary">
    <xsl:copy>
      <xsl:value-of select=". div 2"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

Type-specific template

Generic default

Recursion into kids

Cutting salaries in Haskell ***w/o boilerplate***

Generic function

Traversal scheme

"Make transformation"

```
cutSalary = everywhere (mkT cutSalary')
 where
  cutSalary' (Salary x) = Salary (x `div` 2)
```

Type-specific function

Take home message

REMEMBER

- One can SYB in Prolog.
- SingYB in Prolog is different.

REMEMBER

- Prolog helps understanding SYB.
- ... is a sandbox for advancing SYB.

Scrap your b… plate:
R… cally

Ode to Prolog

…alf Lämmel
…ersität Koblenz-Landau


+

+


Cutting salaries in Prolog ***w/o boilerplate***

Generic function

Traversal scheme

"Make transformation"

```
cutSalary(X,Y) :- everywhere(mkT(cutSalary_),X,Y).

cutSalary_(salary(S1),salary(S2)) :- S2 is S1 / 2.
```

Type-specific function

Haskell

```
cutSalary = everywhere (mkT cutSalary')
 where
  cutSalary' (Salary x) = Salary (x `div` 2)
```

vs.

Seriously: how different is SingYB in Prolog from doing it in Haskell?

Prolog

```
cutSalary(X,Y) :- everywhere(mkT(cutSalary_),X,Y).
% where
cutSalary_(salary(S1),salary(S2)) :- S2 is S1 / 2.
```

Let's look at the details ...

```
cutSalary(X,Y) :- everywhere(mkT(cutSalary_),X,Y).

cutSalary_(salary(S1),salary(S2)) :- S2 is S1 / 2.
```

- What's ...
  - **mkT** ("make transformation")?
  - **everywhere** (traversal scheme)?

“Make transformation”

```
mkT(T,X,Y) :-
  apply(T,[X,Y]) ->
      true
    ; Y = X.
```

In case you haven’t noticed: SingYB involves higher-order functions.

or

```
mkT(T,X,Y) :- choice(T,id,X,Y).

choice(F,G,X,Y) :-
  apply(F,[X,Y]) ->
      true
    ; apply(G,[X,Y]).

id(X,X).
```

- more combinatorial
- slightly more point-free

Doing it w/o **mkT**

w/

```
cutSalary(X,Y) :- everywhere(mkT(cutSalary_),X,Y).

cutSalary_(salary(S1),salary(S2)) :- S2 is S1 / 2.
```

w/o

```
cutSalary(X,Y) :- everywhere(cutSalary_,X,Y).

cutSalary_(X,Y) :-
  X = salary(S1) ->
      ( S2 is S1 / 2, Y = salary(S2) )
    ; Y = X.
```

- less compositional
- less point-free
- less reusable

everywhere (traversal scheme)

```
everywhere(T,X,Z) :-
  gmapT(everywhere(T),X,Y),
  apply(T,[Y,Z]).

gmapT(T,X,Y) :-
  X =.. [C|Kids1],
  map(T,Kids1,Kids2),
  Y =.. [C|Kids2].

map(_,[],[]).
map(F,[X|Xs],[Y|Ys]) :-
  apply(F,[X,Y]),
  map(F,Xs,Ys).
```

Think of everywhere as a “deep” map.

Map over immediate subtems; preserve outermost constructor

Good old “map” for lists

Doing it w/o **everywhere**

w/

```
cutSalary(X,Y) :- everywhere(mkT(cutSalary_),X,Y).

cutSalary_(salary(S1),salary(S2)) :- S2 is S1 / 2.
```

w/o

```
cutSalary(X,Y) :-
  X = salary(S1) ->
      ( S2 is S1 / 2, Y = salary(S2) )
    ; gmapT(cutSalary,X,Y).
```

- termination less obvious
- behavior less obvious
- problem-specific parts non-reusable

Totaling salaries in Prolog ***w/o boilerplate***

Generic function

Traversal scheme

"Make query"

```
getSalary(X,S) :- everything(add,mkQ(0,getSalary_),X,S).

getSalary_(salary(S),S).
```

Type-specific function

Combine intermediate results

everything (traversal scheme)

Think of everything as a "deep" fold (crush).

```
everything(F,Q,X,Z) :-
  gmapQ(everything(F,Q),X,Y),
  apply(Q,[X,R]),
  foldl(F,R,Y,Z).
```

- F: Binary operation
- Q: Extract value
- X: Input term
- Z: Aggregated result

```
gmapQ(Q,X,Y) :-
  X =.. [_|Kids],
  map(Q,Kids,Y).
```

Map over immediate subtems; collect results in a plain list

```
foldl(_,Z,[],Z).
foldl(F,Z,[X|Xs],Y0) :-
  apply(F,[Z,X,Y1]),
  foldl(F,Y1,Xs,Y0).
```

Good old "left-associative fold" for lists

The 1 M$ of SYB

- There are several one-layer traversal combinators:
  - **gmapT**
  - **gmapQ**
  - ... (many more)
- Can they be derived from a single combinator?
- Can that combinator be typeful (in principle)?

One primitive does it all: fold over kids

The traversal primitive

```
gfoldl(F,Z,X,R) :-
  X =.. [C|Kids],
  apply(Z,[C,CR]),
  foldl(F,CR,Kids,R).
```

Derivation of **gmapT**

```
gmapT(T,X,Y) :-
  gfoldl(gmapT_(T),id,X,Y).

gmapT_(T,C,X,Z) :-
  apply(T,[X,Y]),
  pass(C,[Y],Z).
```

```
% Pass argument
pass(T1,Xs,T2) :-
  T1 =.. [C|Ts1],
  append(Ts1,Xs,Ts2),
  T2 =.. [C|Ts2].
```

Direct style for comparison

```
gmapT(T,X,Y) :-
  X =.. [C|Kids1],
  map(T,Kids1,Kids2),
  Y =.. [C|Kids2].
```

Exercise: derive gmapQ from gfoldl.

Haskell's gfoldl for comparison

aka the "headache" combinator

```
-- Type shown only! Definition is a code generator!

gfoldl :: (Data a)
  => (∀ d b. (Data d) => c (d -> b) -> d -> c b)
  -> (∀ g. g -> c g)
  -> a
  -> c a
```

Polymorphic function to process kids one by one

Polymorphic function to process constructor

Type of input term

Result type dependent on type of input term

Prological inquiry into SYB

- Backtracking traversal
- \+ ground traversal
- Bidirectional traversal
- Optimized traversal

Backtracking traversal

Consider this recession-inspired scenario:

- You are CFO.
- You must cut some salaries.
- You don't want to upset everyone.
- Fuzzy constraints:
  - "A will be quiet, if B is also paid less."

The CFO needs a GUI to see all combinations of optional cuts.

```
company([

  topdept(
    name('Human Resources'),
    manager(
      name('Lisa'),
      salary(123456)),
    []),
```

```
topdept(
  name('Development'),
  manager(
    name('Anders'),
    salary(43210)),
    [
  subdept(
    name('Visual Basic'),
    manager(
      name('Amanda'),
      salary(8888)),
      []),
  subdept(
    name('Visual C#'),
    manager(
      name('Erik'),
      salary(4444)),
      []) ])])
```

There are 16 options.

1st attempt

Why does the original encoding not backtrack?

```
cutSalary(X,Y) :-
  everywhere(mkT(cutSalary_),X,Y).

cutSalary_(salary(S1),salary(S2)) :-
  S2 is S1 / 2.
```

```
mkT(T,X,Y) :- choice(T,id,X,Y).

choice(F,G,X,Y) :-
  apply(F,[X,Y]) ->
      true
    ; apply(G,[X,Y]).

id(X,X).
```

mkT is asymmetric (left-biased).

2nd attempt

What about adding an extra *default* for salaries?

```
cutSalary(X,Y) :-
  everywhere(mkT(cutSalary_),X,Y).

cutSalary_(salary(S1),salary(S2)) :- S2 is S1 / 2.
cutSalary_(salary(S),salary(S)).
```

Alas, there is still no backtracking because mkT is deterministic in its first argument.

3rd attempt

Use nondeterministic choice hence.

```
cutSalary(X,Y) :-
  everywhere(;(cutSalary_,id),X,Y).

cutSalary_(salary(S1),salary(S2)) :-
  S2 is S1 / 2.

;(F,G,X,Y) :-
    apply(F,[X,Y])
  ; apply(G,[X,Y]).
```

Noncompositional

16 options.
Alas, "id" serves two purposes now:
fallback for salaries; fallback for other types.

4th attempt

Equip `mkT` with an *applicability condition*.

```
cutSalary(X,Y) :-
  everywhere(mkT(isSalary,cutSalary_),X,Y).

cutSalary_(salary(S1),salary(S2)) :- S2 is S1 / 2.
cutSalary_(salary(S),salary(S)).

isSalary(salary(_)).

mkT(AC,T,X,Y) :-
  apply(AC,[X]) ->
      apply(T,[X,Y])
    ; Y = X.
```

This sort of choice is left-biased but not deterministic.

Something Else

\+ ground traversal

Consider another recession-inspired scenario:

- ☐ Fire the more expensive managers
- ☐ Represent open positions as log var
- ☐ Keep on cutting other salaries
- ☐ Look up list of open positions
- ☐ Assign new managers

Fire, cut, collect open positions

```
fire(X,Y) :- everywhere(mkT(fire_),X,Y).
fire_(manager(_, salary(S)),_) :- S > 99999.
```

```
cutSalary(X,Y) :- everywhere(mkT(cutSalary_),X,Y).
cutSalary_(salary(S1),salary(S2)) :- \+ var(S1), S2 is S1 / 2.
```

```
logvars(X,Y) :- everything(varunion,mkQ([],logvars_),X,Y).
logvars_(X,[X]) :- var(X).
```

... except that this doesn't work yet: `gmap`? aborts.

Traversing non-ground terms
-- Transformations --

\+ ready

```
gmapT(T,X,Y) :-
   X =.. [C|Kids1],
   map(T,Kids1,Kids2),
   Y =.. [C|Kids2].
```

ready

```
gmapT(T,X,Y) :-
   var(X) -> Y = X
 ; ( X =.. [C|Kids1],
     map(T,Kids1,Kids2),
     Y =.. [C|Kids2] ).
```

Traversing non-ground terms
-- Transformations --

disfavored semantics

```
gmapT(T,X,Y) :-
  var(X) -> true
  ; ( X =.. [C|Kids1],
      map(T,Kids1,Kids2),
      Y =.. [C|Kids2] ).
```

favored semantics

```
gmapT(T,X,Y) :-
  var(X) -> Y = X
  ; ( X =.. [C|Kids1],
      map(T,Kids1,Kids2),
      Y =.. [C|Kids2] ).
```

Think of the following law: **`gmapT id = id`**

Traversing non-ground terms
-- Queries --

\+ ready

```
gmapQ(Q,X,Y) :-
  X =.. [_|Kids],
  map(Q,Kids,Y).
```

ready

```
gmapQ(Q,X,Y) :-
  var(X) -> Y = []
 ; ( X =.. [_|Kids],
     map(Q,Kids,Y) ).
```

Traversing non-ground terms
-- Queries --

disfavored semantics

```
gmapQ(Q,X,Y) :-
  var(X) -> true
  ; ( X =.. [_|Kids],
      map(Q,Kids,Y) ).
```

favored semantics

```
gmapQ(Q,X,Y) :-
  var(X) -> Y = []
  ; ( X =.. [_|Kids],
      map(Q,Kids,Y) ).
```

Who else should instantiate “Y”?
At least, the traversal is over - no matter what.

Something Else

Bidirectional traversal

Consider this recession-inspired Prolog teaser:

- We can increase salaries everywhere.
- We can think of a multi-mode add/3.
- Hence we can do subtraction too.
- Hence we can decrease salaries too.

Increasing salaries is easy

```
incSalary(X,Y) :- everywhere(mkT(incSalary_),X,Y).

incSalary_(salary(S1),salary(S2)) :- add(S1,1,S2).

add(X,Y,Z) :- Z is X + Y.
```

multi-mode add/3

```
add(X,Y,Z) :-
  ( \+ var(X), \+ var(Y), Z is X + Y
  ; \+ var(X), \+ var(Z), Y is Z - X
  ; \+ var(Y), \+ var(Z), X is Z - Y
  ).
```

Gedankenexperiment

```
incSalary(X,Y) :- everywhere(mkT(incSalary_),X,Y).
incSalary_(salary(S1),salary(S2)) :- add(S1,1,S2).
```

- Suppose X is free and Y is ground.
- What's the meaning of **incSalary(X,Y)**?
  - ***Abort*** for original **gmapT**.
  - ***Identity*** for logvars-enhanced **gmapT**.

In need of a bidirectional gmapT

```
gmapT(T,X,Y) :-
    var(X), var(Y),
    Y = X
  ;
    \+ var(X),
    X =.. [C|Kids1],
    map(T,Kids1,Kids2),
    Y =.. [C|Kids2]
  ;
    var(X), \+ var(Y),
    Y =.. [C|Kids2],
    map(T,Kids1,Kids2),
    X =.. [C|Kids1].
```

Alas, incSalary(X,Y) is **still** the identity function because gmapT gets to see ***two*** ground terms during bottom-up traversal (because argument T hadn't yet a chance to instantiate).

```
everywhere(T,X,Z) :-
  gmapT(everywhere(T),X,Y),
  apply(T,[Y,Z]).
```

everwhere_

```
everywhere(T,X,Z) :-
  gmapT(everywhere(T),X,Y),
  apply(T,[Y,Z]).
```

```
everywhere_(T,X,Z) :-
  apply(T,[X,Y]),
  gmapT(everywhere_(T),Y,Z).
```

Alas, incSalary(X,Y) is **still (!)** the identity function because T prematurely commits to the identity function and gmapT's backward instantiation comes to late.

everywhere must be bidirectional *by itself*

\+ *ready*

```
everywhere(T,X,Z) :-
  gmapT(everywhere(T),X,Y),
  apply(T,[Y,Z]).
```

ready

```
everywhere(T,X,Z) :-
  Apply = apply(T,[Y,Z]),
  Recurse = gmapT(everywhere(T),X,Y),
  ( var(X), \+ var(Z) ->
       Apply, Recurse
     ; Recurse, Apply ).
```

Stop the madness!

Proposal for new SYB efforts

(Author(s) T.B.D.)

- SYB constrain[t]fully
- More applications of SingYB bidirectionally

Something Else

Optimized traversal

Q: SYB - does it scale?

A: It depends on the precise model.

Let's optimize through a generative model.

Remember the boilerplate code?

```
cutSalary(company(L1),company(L2)) :-
  map(cutSalary,L1,L2).
cutSalary(topdept(N0,M1,L1),topdept(N0,M2,L2)) :-
  cutSalary(M1,M2),
  map(cutSalary,L1,L2).
cutSalary(manager(N0,S1),manager(N0,S2)) :-
  cutSalary(S1,S2).
cutSalary(subdept(N0,M1,L1),subdept(N0,M2,L2)) :-
  cutSalary(M1,M2),
  map(cutSalary,L1,L2).
cutSalary(employee(N0,S1),employee(N0,S2)) :-
  cutSalary(S1,S2).
cutSalary(salary(S1),salary(S2)) :-
  S2 is S1 / 2.
```

This is clever! We do not traverse into **names**.

everywhere describes a ***full*** traversal

```
cutSalary(company(L1),company(L2)) :-
  map(cutSalary,L1,L2).
cutSalary(topdept(N1,M1,L1),topdept(N1,M2,L2)) :-
  cutSalary(N1,N2),
  cutSalary(M1,M2),
  map(cutSalary,L1,L2).
cutSalary(manager(N1,S1),manager(N1,S2)) :-
  cutSalary(N1,N2),
  cutSalary(S1,S2).
cutSalary(subdept(N1,M1,L1),subdept(N1,M2,L2)) :-
  cutSalary(N1,N2),
  cutSalary(M1,M2),
  map(cutSalary,L1,L2).
cutSalary(employee(N1,S1),employee(N1,S2)) :-
  cutSalary(N1,N2),
  cutSalary(S1,S2).
cutSalary(salary(S1),salary(S2)) :-
  S2 is S1 / 2.
cutSalary(name(N0),name(N0)).
```

Think of **everywhere(id)**

```
nowhere(company(L1),company(L2)) :-
   nowhere(nowhere,L1,L2).
nowhere(topdept(N1,M1,L1),topdept(N1,M2,L2)) :-
   nowhere(N1,N2),
   nowhere(M1,M2),
   map(nowhere,L1,L2).
nowhere(manager(N1,S1),manager(N1,S2)) :-
   nowhere(N1,N2),
   nowhere(S1,S2).
nowhere(subdept(N1,M1,L1),subdept(N1,M2,L2)) :-
   nowhere(N1,N2),
   nowhere(M1,M2),
   map(nowhere,L1,L2).
nowhere(employee(N1,S1),employee(N1,S2)) :-
   nowhere(N1,N2),
   nowhere(S1,S2).
nowhere(salary(S0),salary(S0)).
nowhere(name(N0),name(N0)).
```

Let's generate optimized code from type definitions and customize it with the type-specific cases -- all to be done by **logic meta-programming**.

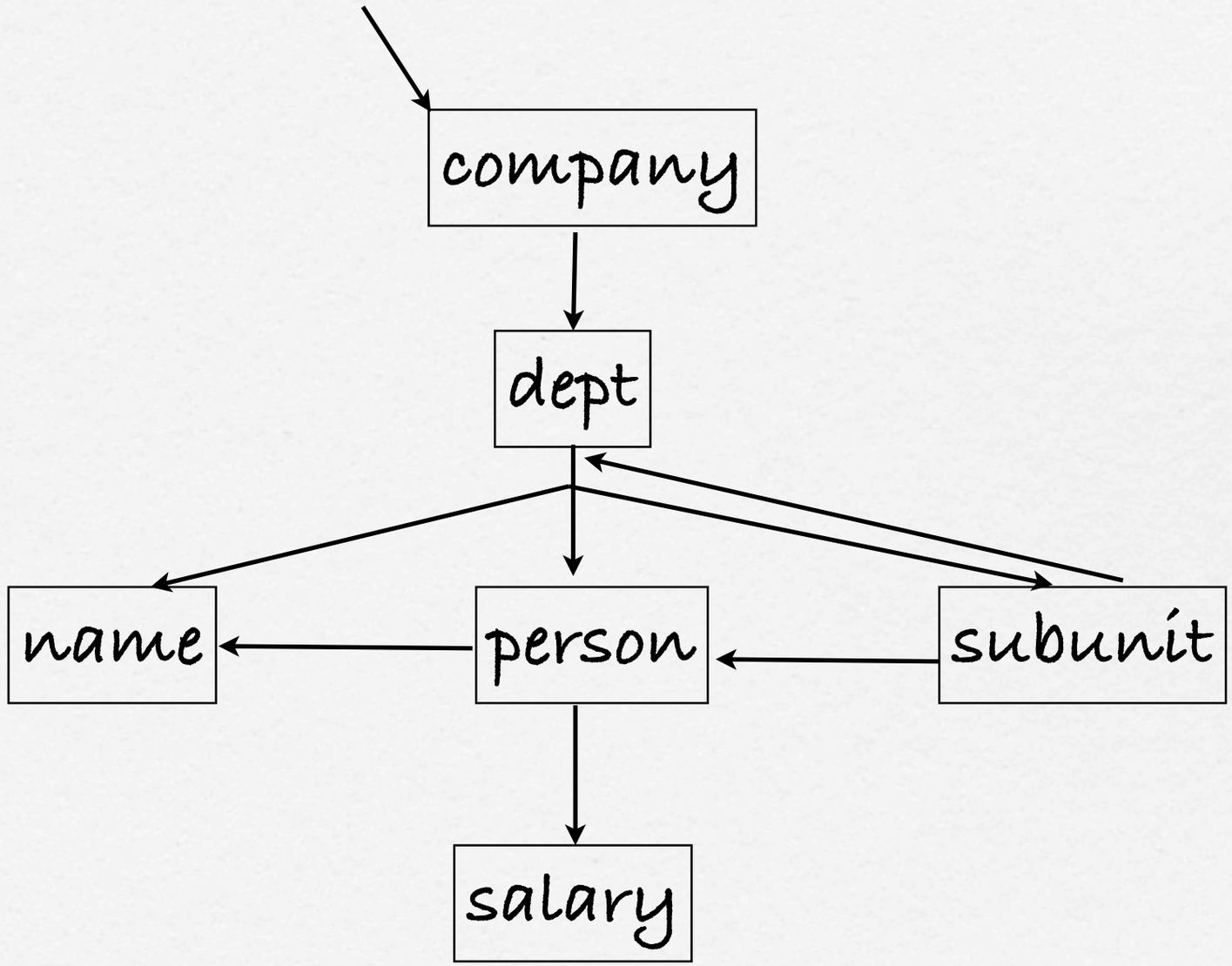
Applications of `id/2` were unfolded.

Type definitions for companies

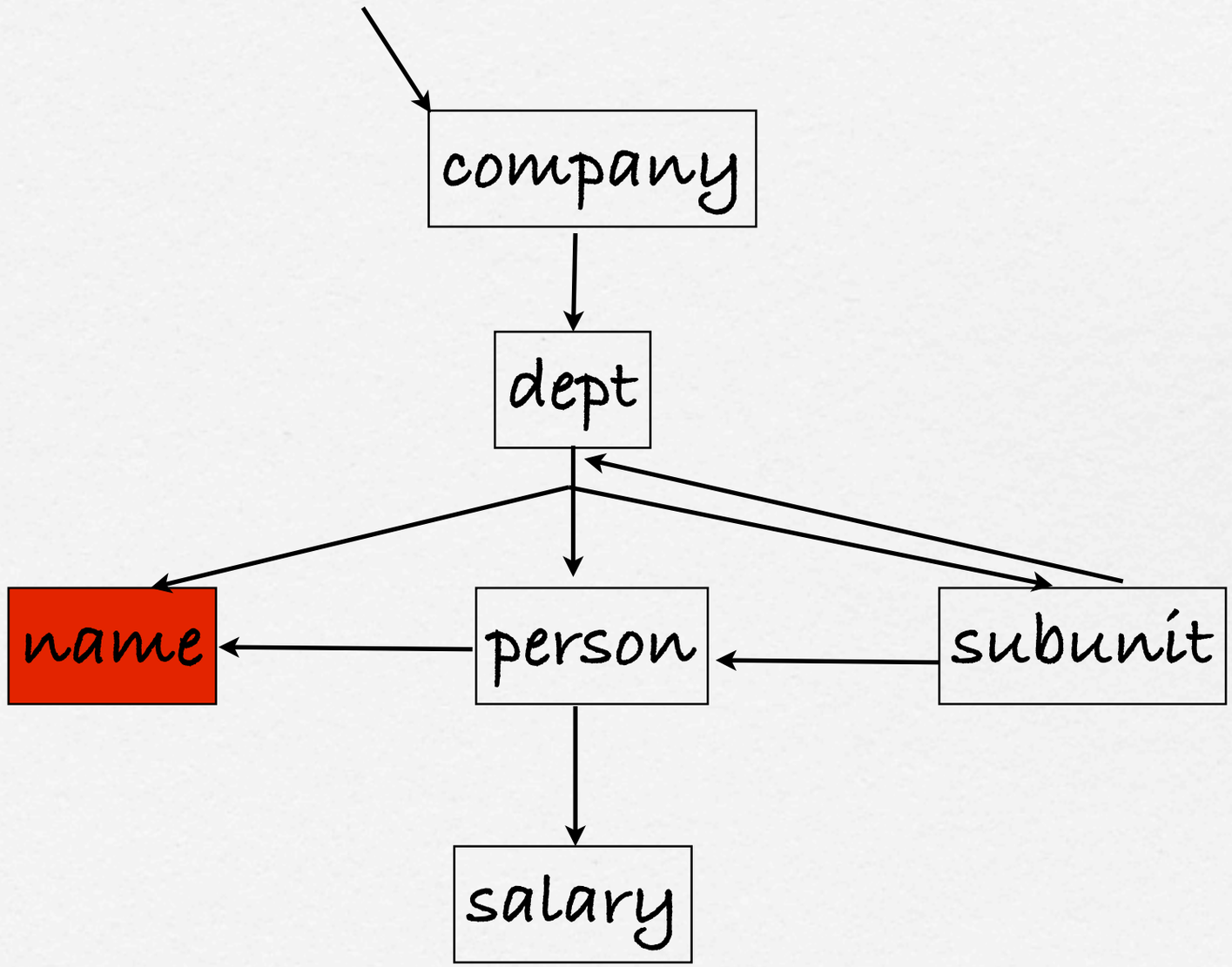
```
company(company(L))      :- map(topdept,L).
topdept(topdept(N,M,L)) :- alias(dept(N,M,L)).
manager(manager(N,S))    :- alias(person(N,S)).
subunit(subdept(N,M,L)) :- alias(dept(N,M,L)).
subunit(employee(N,S))   :- alias(person(N,S)).
salary(salary(N))        :- number(N).
name(name(A))            :- atom(A).

dept(N,M,L) :- name(N), manager(M), map(subunit,L).
person(N,S) :- name(N), salary(S).
```

- Types are programs.
- These programs are algebraic signatures.
  - + lists
  - + aliases

Sort dependency graph



Sorts \+ reaching *salary*



Cutting salaries
w/ generating boilerplate

Type-specific case as usual

```
cutSalary(salary(S1),salary(S2)) :- S2 is S1 / 2.

:- completeT(cutSalary,company).
```

Generate rest of predicate

Commit to root sort for type-driven generation

everywhere on the low road

```
completeT(Name,Sort) :-
  clauses(Name/2,ClausesO),
  abolish(Name/2),
  skippableSorts(ClausesO,Sort,Skip),
  generateT(Skip,Name,Sort,ClausesG),
  override(ClausesG,ClausesO,Clauses),
  map(assert,Clauses),
  compile_predicates([Name/2]).
```

Retrieve type-specific cases for term-level meta-programming.

Remove those type-specific cases because the ultimate predicate will be generated.

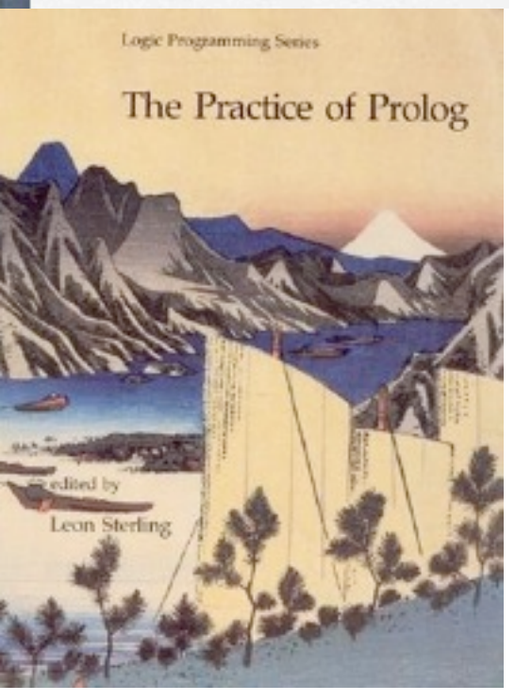
Determine all sorts that can be skipped when encountered during traversal.

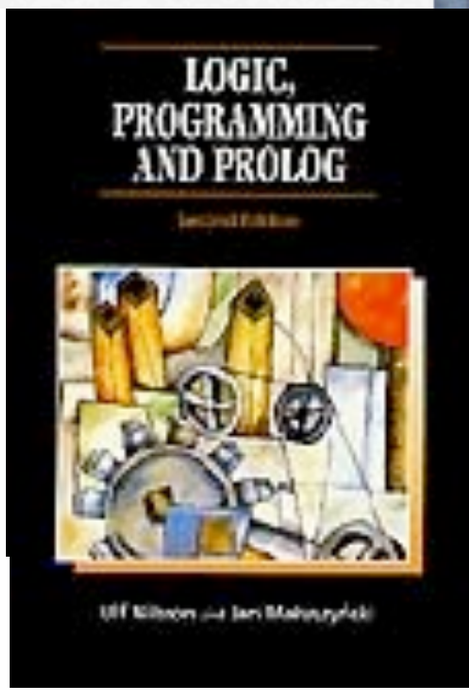
Generate traversal clauses at the term level for all reachable, unskippable sorts.

Override generated clauses (still at the term level) by type-specific cases.

Assert the resulting program.

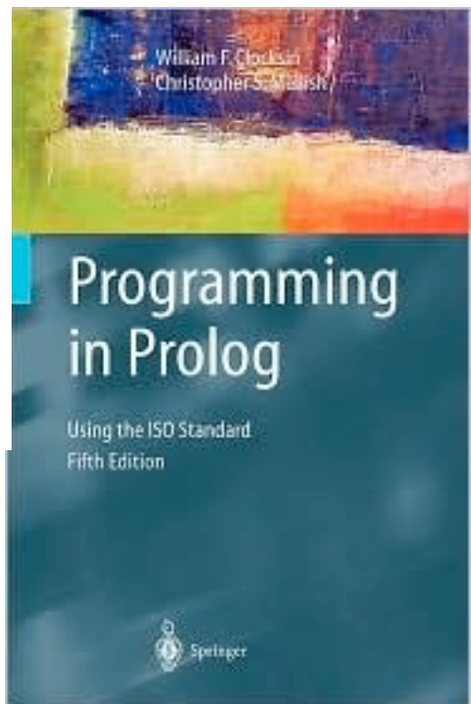
Compile the computed program.


Logic Programming Series
The Practice of Prolog
edited by
Leon Sterling


LOGIC,
PROGRAMMING
AND PROLOG

Prolog


The Art of Prolog


Programming
in Prolog
Using the ISO Standard
Fifth Edition
Springer

That’s it more or less.

SYB research topics

- Language design of traversal control

  (see Victor L. Winter and Mahadevan Subramaniam: The transient combinator, higher-order strategies, and the distributed data problem. Science of Computer Programming, 2004)

SYB research topics cont'd

- Type-driven traversal optimization

  (see Karl J. Lieberherr, Boaz Patt-Shamir, Doug Orleans: Traversals of object structures: Specification and Efficient Implementation. TOPLAS, 2004)

SYB research topics cont'd

- Fusion-like traversal optimization

  (see Patricia Johann and Eelco Visser: Strategies for Fusing Logic and Control via Local, Application-Specific Transformations. Technical Report, University of Utrecht, 2003)

SYB research topics cont'd

- Scrap your objectplate

(see Gavin M. Bierman, Erik Meijer, Wolfram Schulte: The Essence of Data Access in Comega. ECOOP 2005)

SYB research topics cont'd

- Attribute grammars and traversal

  (see Lennart C. L. Kats, Anthony M. Sloane, Eelco Visser: Decorated Attribute Grammars: Attribute Evaluation Meets Strategic Programming. CC 2009)

SYB research topics cont'd

- Termination analysis

  (see Markus Kaiser and Ralf Lämmel: An Isabelle/HOL-based model of Stratego-like traversal strategies. PPDP 2009)

SYB research topics cont'd

- Sweet spots in generic programming

  (e.g., Ralf Hinze, Johan Jeuring, Andres Löh: Type-indexed data types. Science of Computer Programming, 2004)

SYB research topics cont'd

- Applications, applications, applications
- Model transformations
- Modern compilers
- … ?


GAME OVER
Thanks!
Questions / comments?