Scrap your boilerplate: Prologically

Ralf Lämmel
Universität Koblenz-Landau
Important abbreviations

Please write them down!

☐ SYB = Scrap Your Boilerplate

☐ SingYB = Scrap[Ing]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: ....
SingYB
= Way of dealing w/ large bananas

- Scrap your boilerplate
- Scrap more boilerplate
- Scrap your boilerplate with class
- Scrap your nameplate
- Scrap++

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

Related papers
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([

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

```xml
<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*

```haskell

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

Generic function

Traversal scheme

“Make transformation”

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 boilerplate: Prologically

Ralf Lämmel
Universität Koblenz-Landau

Ode to Prolog

+ + +
Cutting salaries in Prolog \textit{w/o boilerplate}

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

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

Prolog

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

cutSalary(X,Y) :- everywhere(mkT(cutSalary_),X,Y).
% where
cutSalary_ (salary(S1),salary(S2)) :- S2 is S1 / 2.
Let’s look at the details ...

```prolog
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.
```

```
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

In case you haven’t noticed: SingYB involves higher-order functions.
Doing it w/o \texttt{mkT}

\texttt{cutSalary(X,Y) :- everywhere(mkT(cutSalary_),X,Y).}

\texttt{cutSalary_ (salary(S1),salary(S2)) :- S2 is S1 / 2.}

\texttt{cutSalary(X,Y) :- everywhere(cutSalary_,X,Y).}

\texttt{cutSalary_ (X,Y) :-}
\texttt{X = salary(S1) ->}
\texttt{( S2 is S1 / 2, Y = salary(S2) )}
\texttt{; 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).
Doing it w/o everywhere

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

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*

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

getSalary_(salary(S),S).

Generic function

Traversal scheme

"Make query"

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).

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

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

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

Map over immediate subterms; collect results in a plain list

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

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

Derivation of \( gmapT \)

\[
gmapT(T,X,Y) :-
\]
\[
gfoldl(gmapT_(T),id,X,Y).
\]

\[
gmapT_(T,C,X,Z) :-
\]
\[
\text{apply(T,[X,Y]),}
\]
\[
\text{pass(C,[Y],Z).}
\]

% 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.

There are 16 options.
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).
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]).
```

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.
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

\[
\text{fire}(X,Y) :- \text{everywhere(mkT(fire_),X,Y)}. \\
\text{fire_}(\text{manager}(\_, \text{salary}(S))\_,\_) :- S > 99999.
\]

\[
\text{cutSalary}(X,Y) :- \text{everywhere(mkT(cutSalary_),X,Y)}. \\
\text{cutSalary_}(\text{salary}(S1),\text{salary}(S2)) :- \text{\(\neg\) var}(S1), S2 \text{ is } S1 \div 2.
\]

\[
\text{logvars}(X,Y) :- \text{everything(varunion,mkQ([],logvars_),X,Y)}. \\
\text{logvars_}(X,[X]) :- \text{var}(X).
\]

... except that this doesn’t work yet: \text{gmap? aborts}.
Traversing non-ground terms
-- Transformations --

\+ ready

```
gmapT(T,X,Y) :-
  var(X) -> Y = X
; ( 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 --

Think of the following law: \( gmapT \ id = \ id \)
Traversing non-ground terms

-- Queries --

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

gmapQ(Q,X,Y) :-
  X =.. [_|Kids],
  map(Q,Kids,Y).
Traversing non-ground terms

-- Queries --

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

\[
\text{incSalary}(X,Y) \leftarrow \text{everywhere}(\text{mkT}(\text{incSalary}_),X,Y).
\]

\[
\text{incSalary}_(\text{salary}(S1),\text{salary}(S2)) \leftarrow \text{add}(S1,1,S2).
\]

\[
\text{add}(X,Y,Z) \leftarrow Z \text{ is } X + Y.
\]
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
   ).
Suppose $X$ is free and $Y$ is ground.

What's the meaning of $\text{incSalary}(X,Y)$?

- Abort for original $\text{gmapT}$.
- Identity for logvars-enhanced $\text{gmapT}$. 

\[
\text{incSalary}(X,Y) :\text{- everywhere}(\text{mkT}(\text{incSalary}_\_),X,Y).
\]

\[
\text{incSalary}_\_(\text{salary}(S1),\text{salary}(S2)) :\text{- add}(S1,1,S2).
\]
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_  

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

\[
everywhere_\mathbf{(T, X, Z)} :-
apply(T, [X, Y]),
gmapT(everywhere_\mathbf{(T)}, Y, Z).
\]

Alas, incSalary(X, Y) is still (!) the identity function because \( T \) prematurely commits to the identity function and gmapT’s backward instantiation comes too late.
everywhere must be bidirectional by itself

\[\text{everywhere}(T,X,Z) \leftarrow\]
\[\text{gmapT}(\text{everywhere}(T),X,Y),\]
\[\text{apply}(T,[Y,Z]).\]

\[\text{everywhere}(T,X,Z) \leftarrow\]
\[\text{Apply} = \text{apply}(T,[Y,Z]),\]
\[\text{Recurse} = \text{gmapT}(\text{everywhere}(T),X,Y),\]
\[(\text{var}(X), \text{\(+\text{var}(Z)\rightarrow})\]
\[\text{Apply, Recurse}\]
\[;\text{Recurse, Apply}).\]
Stop the madness!

Proposal for new SYB efforts
(Author(s) T.B.D.)

- SYB constraInItIfully
- More applications of SingYB bidirectionally
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?

```prolog
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)`

```prolog
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)).
```

Applications of `id/2` were unfolded.

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

- company(A) :- map(A).
- topdept(A) :- alias(A).
- manager(A) :- alias(A).
- subunit(A) :- alias(A).
- salary(A) :- number(A).
- name(A) :- atom(A).

- dept(A) :- name(A), manager(A), map(A).
- person(A) :- name(A), salary(A).

Types are programs.
These programs are algebraic signatures.
+ lists
+ aliases
Sort dependency graph

- company
  - dept
    - name
    - person
      - subunit
    - salary
Sorts \( \setminus + \) reaching salary
Cutting salaries with generating boilerplate.

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

:- completeT(cutSalary,company).

Type-specific case as usual

Generate rest of predicate

Commit to root sort for type-driven generation
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.
I ❤ Prolog
That’s it more or less.
SYB research topics

- Language design of traversal control
SYB research topics cont’d

- Type-driven traversal optimization
SYB research topics cont’d

- Fusion-like traversal optimization
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

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)
Applications, applications, applications
Model transformations
Modern compilers
... ?
Thanks!

Questions / comments?