Scrap Your Boilerplate—Prologically!

PPDP'09 Invited Talk

Ralf Lämmel

Software Languages Team, Universität Koblenz-Landau

Abstract

"Scrap Your Boilerplate" (SYB) is an established style of generic functional programming. The present paper reconstructs SYB within the Prolog language with the help of the univ operator and higher-order logic programming techniques. We pay attention to the particularities of Prolog. For instance, we deal with traversal of non-ground terms. We also develop an alternative model of SYB-like traversal based on metaprogramming. This generative, type-driven model is also amenable to type-driven optimization.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features; D.1.1 [*PROGRAMMING TECHNIQUES*]: Applicative (Functional) Programming; D.1.6 [*PROGRAMMING TECHNIQUES*]: Logic Programming; D.2.13 [*SOFTWARE ENGINEERING*]: Reusable Software

General Terms Design, Experimentation, Languages

Keywords Scrap Your Boilerplate, Haskell, Prolog, Stratego

1. Prologue

Consider the following Prolog term, which represents the organizational structure of an illustrative company in terms of hierarchical departments with managers and employees:

```
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)), [])])])
```

Now suppose that we need to total all salaries of the company. Also, we need to cut all salaries in half. The following Prolog session illustrates these operations:

```
?- Com1 = ... the sample company ...
|  getSalary(Com1,S1),
|  cutSalary(Com1,Com2),
|  getSalary(Com2,S2).
S1 = 179998,
S2 = 89999.
```

PPDP'09, September 7–9, 2009, Coimbra, Portugal.
Copyright © 2009 ACM 978-1-60558-568-0/09/09...$10.00

```
getSalary(company(L),S) :-
  mapreduce(getSalary,add,0,L,S).
getSalary(topdept(_,M,L),S0) :-
  getSalary(M,S1),
  mapreduce(getSalary,add,0,L,S2),
  add(S1,S2,S0).
getSalary(manager(_,S1),S2) :-
  getSalary(S1,S2).
getSalary(subdept(_,M,L),S0) :-
  getSalary(M,S1),
  mapreduce(getSalary,add,0,L,S2),
  add(S1,S2,S0).
getSalary(employee(_,S1),S2) :-
  getSalary(S1,S2).
getSalary(salary(S),S).

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.
```

Figure 1. Boilerplate for totaling salaries and cutting them in half

Fig. 1 shows the routine implementation of `getSalary/2` and `cutSalary/2`.[1] These predicates simply recurse into the terms, and they use list-processing predicates `map/3` and `mapreduce/5` to deal with the lists of subunits (i.e., employees vs. sub-departments) that each department can have. (We will recall such higher-order techniques in §2.) Writing such boilerplate code is laborious and error-prone—especially when the underlying data model involves yet more types and constructors.

The "Scrap Your Boilerplate" style of generic functional programming (SYB [Lämmel and Peyton Jones 2003]) and the strategic programming style (c.f., Stratego [Visser et al. 1998, Bravenboer et al. 2008])[2] allow us to scrap such boilerplate. We adopt SYB in this paper. (Once we abstract from the specific setups of Haskell or Stratego, once we focus on only simple traversal idioms, as we do in this paper, these two styles are not too different.)

With SYB aboard, we focus on problem-specific cases:

[1] A code distribution for this paper is available from its website `http://www.uni-koblenz.de/~laemmel/OdeToProlog/`. We use SWI-Prolog 5.6.62 (without fundamentally relying on any nonstandard features though).

[2] `http://www.program-transformation.org/Transform/TheEssenceOfStrategicProgramming`

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

These cases define what to do when salary terms are hit. The rest of the traversals for totaling or cutting salaries is taken care of by traversal schemes. SYB can be done (easily) in Prolog, too:

```
?- company1 = ... the sample company ...
|  everything(add,mkQ(0,getSalary),Com1,S1),
|  everywhere(mkT(cutSalary),Com1,Com2),
|  everything(add,mkQ(0,getSalary),Com2,S2).
S1 = 179998,
S2 = 89999.
```

Here, `everything/4` is a traversal scheme that performs a query ('a deep map-reduce'), and `everywhere/3` is a traversal scheme for a transformation ('a deep map'). We instantiate `everything/4` to apply `add/3` (addition for numbers) for reduction and to attempt `getSalary/2` at each node, but to return '0' if `getSalary/2` does not fit; c.f., '`mkQ(0,getSalary)`'. We instantiate `everywhere/3` to attempt `cutSalary/2` at each node, but to preserve the node if `cutSalary/2` does not fit; c.f., '`mkT(cutSalary)`'.

Road-map

The remaining sections develop SYB support for Prolog. Our development relies on higher-order logic programming techniques which we recall in §2. The development has two major parts: a 'high road' (§3) and a 'low road' (§4) reminiscent of [Naish and Sterling 2000]; the former supports SYB in terms of higher-order predicates; the latter uses generative programming (metaprogramming) instead. On the high road, we make some specific efforts to seamlessly integrate SYB with Prolog—as far as backtracking or non-ground terms are concerned. On the low road, we attempt a simple form of type-driven traversal optimization. The paper is concluded in §5.

2. Higher-order logic programming primer

We recall higher-order logic programming in Prolog [Warren 1982, Cheng et al. 1990, Nilsson and Hamfelt 1995, Naish 1996, Naish and Sterling 2000].[3] We begin with a regular (first-order) goal that calls a predicate `add/3` right away. Thus:

```
?- add(23,19,X).
X = 42.
```

In this paper, we usually classify positions of predicates (rather implicitly) to serve for arguments or results. `add/3` has two arguments and one result. By convention, arguments go first, while results (if any) go last. Obviously, some predicates may be flexible enough to be used in multiple directions—thereby blurring the separation of arguments and results. For instance, `add/3` could be defined such that it can do subtraction, too. Such different directions will not be relevant (say, exploited) for predicates defined in this paper (except for a short discussion in §3.6).

We can use terms to represent goals—applications of predicate symbols in particular. Hence, we can use term variables to *abstract* over such applications. We can 'call' (say, 'execute' or 'prove') such terms. Thus:

```
?- Term = add(23,19,Result), call(Term).
Term = add(23, 19, 42),
Result = 42.
```

[3] In this paper, we limit ourselves to plain Prolog; we do not leverage hybrid forms of functional/logic programming languages, such as Lambda-PROLOG [Nadathur and Miller 1988] or Curry [Hanus 1997]. We do not use any form of static typing either. In particular, we do not exploit any directional types (modi) or algebraic types [Mycroft and O'Keefe 1984, Boye and Maluszynski 1997].

```
map(_,[]).
map(P,[X|Xs]) :- apply(P,[X]), map(P,Xs).

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

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

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

filter(_,[],[]).
  filter(P,[X|Xs],Ys1) :-
  filter(P,Xs,Ys2),
  ( apply(P,[X]) -> Ys1 = [X|Ys2] ; Ys1 = Ys2 ).

mapreduce(F,G,Z,X,Y) :-
  foldl(mapreduce_(F,G),Z,X,Y).
mapreduce_(F,G,X1,X2,Y) :-
  apply(F,[X2,X3]), apply(G,[X1,X3,Y]).
```

Figure 2. Higher-order list-processing predicates

(`call(Term)` may also just be written as `Term`.) Now it should be obvious that we can represent *incomplete* applications of predicate symbols as terms, too—because we can always append positions to a term and call it eventually. Let us assume a helper predicate `pass/3` to this end. Thus:

```
?- Inc = add(1), pass(Inc,[41,Result],Goal), Goal.
Inc = add(1),
Result = 42,
Goal = add(1, 41, 42).
```

The composition of `pass/3` and `call/1` is called `apply/2`:

```
?- Inc = add(1), apply(Inc,[41,Result]).
Inc = add(1),
Result = 42.
```

This is the fundamental functionality for higher-order logic programming. Here we note that `pass/3` and `apply/2` can be defined in terms `=../2` and `call/1`:

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

The definition of the argument-passing predicate uses Prolog's univ operator (c.f., '`=..`') to obtain access to subterms of a term (here: the positions of an incomplete application of a predicate symbol). Conceptually, the univ operator models a relation between a term, its functor and its immediate subterms.

On top of `apply/2` and friends, Fig. 2 defines fundamental higher-order predicates for list processing: `map(P,Xs)` applies predicate `P` to each element of `Xs`; `map(F,Xs,Ys)` applies `F` to each element of `Xs` and collects the results in `Ys`; `foldl(F,Z,Xs,Y)` folds over `Xs` in a left-associative manner while starting from `Z` and combining intermediate results by `F`; `foldr/4` is the right-associative companion of `foldl/4`; `filter(P,Xs,Ys)` obtains the subsequence `Ys` from `Xs` with all the elements that satisfy `P`; `mapreduce(F,G,Z,X,Y)` composes mapping and reduction (folding).[4]

[4] In this paper, we generally apply the convention that a predicate symbol ends on one or many underscore characters to mean that it is a helper of another predicate. We should prefer the use of binding blocks, say closures.

As an illustration, the following sessions starts from the list [1,2,3,4]; a filter for odd numbers is applied; the remaining numbers are incremented by 1; finally, the list is totaled. Thus:

```
odd(X) :- 1 is X mod 2.
inc(X,Y) :- add(X,1,Y).
add(X,Y,Z) :- Z is X + Y.

?- L1 = [1,2,3,4],
|  filter(odd,L1,L2),
|  map(inc,L2,L3),
|  foldr(add,0,L3,R).
L1 = [1, 2, 3, 4],
L2 = [1, 3],
L3 = [2, 4],
R = 6.
```

3. The high road to SYB

We define SYB's basic combinators for one-layer traversal and common traversal schemes. We discuss challenges related to backtracking and non-ground terms.

3.1 Generic traversal

SYB (and strategic programming for that matter) is based on a few one-layer traversal combinators from which recursive traversals and traversal schemes can be derived. By one-layer traversal we mean that an argument function (predicate) is applied only to the immediate subterms of a given term. The most important one-layer traversal combinators are gmapT/3 for transformations (c.f., T) and gmapQ/3 for list-producing queries (c.f., Q). For instance, we can define the traversal schemes everywhere/3 and everything/4 as follows.

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

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

Prolog is particularly suited to express a reference semantics for one-layer traversal combinators—as long as we do not expect to obtain a statically typed model easily. Both combinators take apart arbitrary terms by the univ operator and process the immediate subterms by a plain list map:

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

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

gmapT/3 recomposes a term with the original constructor whereas gmapQ/3 simply returns a list of intermediate results. In the definitions, univ deserves all the credit. Colloquially speaking, =../2 embodies Prolog's reflection API. (For comparison, think of Java's java.lang.reflect package.)

3.2 Traversal customization

In the introductory example of §1, we used mkT/3 and mkQ/4 to construct appropriate arguments for the traversal schemes as a means to customize the ultimate traversal. The idea is that the problem-specific parts of a traversal typically are limited to specific types (such as salaries in the running example), but these parts need to be generalized ('made generic') before they can be used by a traversal scheme. Hence, mkT/3 and mkQ/4 compose a type-specific predicate with a generic default.

In Haskell, this process is truly type-driven, whereas in Prolog, the process is success/failure-driven (just like in strategic programming with Stratego). A common default for transformations is the identity function, i.e., the argument term is returned unchanged. A common default for queries is a constant function, i.e., a designated 'zero' (algebraically speaking) is returned. Thus:

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

The use of 'if-then-else' (c.f., '->') expresses that a type-specific case is attempted first, and only if it fails, the generic default applies. Arguably, this style of composition makes it difficult to separate failure due to inapplicability vs. failure in the sense of a missing pre-/post-condition somewhere along the execution of the type-specific case. We can easily provide variations that carry an extra argument AC (for applicability condition) to 'unbacktrackably' commit to the type-specific case on the grounds of that condition.

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

The applicability condition could be a type test. For instance, we could test for the presence of a salary, but let the type-specific case fail if we find a salary that is too small to be cut any further. The applicability condition and the type-specific case are to be instantiated as follows:

```
isSalary(salary(_)).
cutSalary(salary(S1),salary(S2)) :- S1 > 1, S2 is S1 / 2.
```

3.3 Backtracking traversal

In our experience, the use of backtracking in generic traversal seems to be very limited (say, local), but let us look into the feature interaction between SYB and general backtracking anyway. Suppose, we are interested in all possible combinations of *optionally* cutting salaries. Our sample company involves 4 salaries. Hence, we should find 16 different result companies. We may use the following type-specific case.

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

If we apply everywhere/3 and if we now tried to find all solutions, no backtracking would be noticeable. (There is one solution.) This deterministic semantics is a result of using ->/2 in mkT/2. The non-determinism of the condition is cut off. The applicability condition-based variations of the mk predicates can be fruitfully used here. We commit to the type-specific case first and then allow for backtracking within that case. Now we count 16 solutions.

```
?- Com1 = ... the sample company ...
|  findall(
|     Com2,
|     everywhere(mkT(isSalary,cutSalary),Com1,Com2),
|     Companies),
|  length(Companies,Len).
Len = 16.
...
```

3.4 SYB's headache combinator

One of the mind-boggling SYB results is that apparently any thinkable basic traversal combinator (that processes the immediate subterms of a term) can be expressed in terms of a cunning fold operator gfoldl on terms. In pseudo-code (and functional style), an application of gfoldl to a term c(t1,...,tn) would be expanded as follows:

```
    gfoldl f z c(t1,...,tn) = (z c) ‘f‘ t1 ··· ‘f‘ tn
```

In Haskell's SYB implementation of the Glasgow Haskell Compiler (GHC), `gfoldl` is supported by 'code generation'. All other one-layer traversal combinators (such as `gmapT` and `gmapQ`) are defined in terms of `gfoldl`. Let us look at the Haskell type of `gfoldl`:

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

The arguments are similar to those of `foldl/4` for lists, i.e., the first argument models a binary operator to combine an intermediate result and another element (i.e., a subterm of type `d`); the second argument models a unary operator to be applied to the mere term constructor so that an initial value is obtained for folding. The type-constructor argument `c` models the dependency between the argument type of the fold and its result. Conceptually, `c` is the (type-level) identity function in the case of `gmapT`, and a (type-level) constant function in the case of `gmapQ`.

Again, if we do not mind the lack of static typing, we can easily implement `gfoldl` in Prolog once and for all. The definition immediately clarifies the above explanation.

```
gfoldl(F,Z1,X,R) :-
  X =.. [C|Kids1],
  apply(Z1,[C,CR]),
  foldl(F,CR,Kids1,R).
```

Haskell's `gfoldl`-based definitions of one-layer traversal combinators are pretty complicated because of the involved degree of polymorphism combined with Haskell's lack of type-level lambdas. Of course, in Prolog the `gfoldl`-based definitions are straigthforward. For instance:

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

(`(=)` denotes the polymorphic identity function.)

3.5 Traversal of non-ground terms

So far we have silently assumed to traverse *ground* terms—as this is the only option anyhow in the case of the original Haskell incarnation of SYB. In Prolog though, we may want to deal with non-ground terms seamlessly. For instance, it should be possible to instantiate `everything` so that we can retrieve all free variables from a given term. Thus, the plan is to define a function `freevars` with the following behavior:

```
?- freevars(f(X,g(X,h(Y,Z))),R).
R = [X, Y, Z]
```

However, the current definition of `gmapQ/3` (which is used by `everything`) is not fit for this purpose. The issue is that `gmapQ` tries to deconstruct terms, and its application of the univ operator will abort execution when facing a free variable. This is easy to resolve though; there is no real choice here, in fact: we have to view free variables as constant terms that cannot be visited any further by one-layer traversal. We improve `gmapQ` as of §3.1 accordingly.

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

Here we use Prolog's meta-predicate `var/1` for testing a term to be a free variable. Traversal schemes such as `everything` immediately benefit from the generalization of `gmapQ`. The scheme for collecting free variables can now be defined as follows:

```
freevars(X,Y) :-
  everything(
    varunion,
    mkQ([],guard(var,singleton)),
    X,Y).
```

These helpers are used: `singleton/2` returns an argument term as a singleton list. `guard/4` guards a unary function by a predicate (i.e., the function is applied only if the predicate succeeds). `varunion/3` takes the left-biased union of two lists that are supposed to be sets of free variables. (This is implemented very inefficiently below. An accumulator should be used instead.) `varmember/2` is a free variable-aware variation on the standard `member/2` predicate.

```
singleton(X,[X]).

guard(P,F,X,Y) :- apply(P,[X]), apply(F,[X,Y]).

varunion(X,[],X).
varunion(X0,[H|T],Y) :-
  ( varmember(H,X0) -> varunion(X0,T,Y)
  ; ( append(X0,[H],X1), varunion(X1,T,Y) )).

varmember(E,[H|T]) :-
  var(E), var(H), ( E == H; varmember(E,T) ).
```

3.6 Backward traversal

We can also further generalize traversal to deal with backward traversal by which we mean that the argument position is not bound while the result position is bound instead. In fact, we only know of a useful definition of backward *transformation* (as opposed to backward query). Consider the following candidate for a traversal parameter:

```
incSalary(salary(S1),salary(S2)) :- add(S1,1,S2).
```

Now consider a goal `everything(mkT(incSalary),X,Y)`. If `X` is bound (to, say, a ground company term) before the traversal's execution, then the regular (forward) model of traversal applies. Hence, `Y` will be bound to a company term with *incremented* salaries after execution. If instead `X` is free and `Y` is bound before execution, then backward transformation would mean that `X` should be bound to a term with *decremented* salaries after execution. This semantics can be achieved; see the code distribution. Both `gmapT/3` and `everywhere/3` (and `add/3`) have to be made aware of directions to this end.

4. The low road to SYB

Given Prolog's ease with metaprogramming, we should also walk the low road of SYB where higher-order traversal schemes are replaced by metaprograms that produce first-order traversals. Those generated traversals are convenient to understand SYB in yet another way. Also, the generated code is efficient in that it uses no higher-order style (except for `map/3` and friends). The idea of traversal generation is inspired by similar work in the rewriting community [van den Brand et al. 2000]. It turns out that the generated code is also amenable to simple but effective type-driven optimizations that cannot be achieved for the higher-order schemes. This optimization effort is inspired by techniques of adaptive programming [Palsberg et al. 1997, Lieberherr et al. 2004].

4.1 Algebraic signatures

Ultimately, we want to generate the code of Fig. 1. To this end, we need an explicit definition of the signature of terms that we expect to traverse. We represent this signature as regular (though structurally constrained) predicates whose extension is the actual set of terms of interest:

```
transform(company(L1),company(L2)) :-
  map(transform,L1,L2).
transform(topdept(N1,M1,L1),topdept(N2,M2,L2)) :-
  transform(N1,N2),
  transform(M1,M2),
  map(transform,L1,L2).
transform(manager(N1,S1),manager(N2,S2)) :-
  transform(N1,N2),
  transform(S1,S2).
transform(subdept(N1,M1,L1),subdept(N2,M2,L2)) :-
  transform(N1,N2),
  transform(M1,M2),
  map(transform,L1,L2).
transform(employee(N1,S1),employee(N2,S2)) :-
  transform(N1,N2),
  transform(S1,S2).
transform(salary(Number),salary(Number)).
transform(name(Atom),name(Atom)).
```

Figure 3. Generated, type-driven, deep identity transformation

-- One term constructor per clause

```
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).
```

-- Type aliases

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

`atom/1` and `number/1` are built-in 'simple' types. The predicate `alias/1` serves as an indication that its argument predicate is a type alias ('macro'). Operationally, `alias/1` is just `call/1`. We will use algebraic signatures to drive traversal generation, but, much simpler, we can use them for run-time type checking, too:

```
?- Com1 = ... the sample company ...
|  company(Com1).
true.
```

4.2 Traversal generation

The algebraic signatures can be directly facilitated to generate traversal programs. For instance, Fig. 3 shows a generated traversal program from which we derive `cutSalary/2` eventually. As the generated code stands, it does not yet involve any problem-specific parts; it covers generically all types (constructors) that are defined by the algebraic signature. It is easy to see that, operationally, the generated predicate would just perform a deep identity mapping over its argument term.

We present the metaprogram for traversal generation below. The generation algorithm is parametrized by a `Name` for the generated predicate and by the root `Sort` for data to be traversed (such as `company` in our example). The algorithm determines all `Sorts` that are reachable from `Sort` (i.e., sorts of possible subterms), and all sorts to `Skip` during code generation (i.e., `simple` types).

```
generateT(Name,Sort,Cs) :-
  findall(X,simple(X),Skip),
  generateT(Skip,Name,Sort,Cs).

generateT(Skip,Name,Sort,Cs) :-
  reachable_sorts(Sort,Sorts),
  map(generateT_(Skip,Name),Sorts,Css),
  concat(Css,Cs).
```

If a sort should not be skipped, then we look up all clauses that define the sort; that is done by `sort_to_clauses/2`. All those clauses are processed one by one. Thus:

```
generateT_(Skip,Name,Sort,Cs0) :-
  member(Sort,Skip) ->
      Cs0 = []
    ; sort_to_clauses(Sort,Cs1),
      map(generateT__(Skip,Name),Cs1,Cs0).
```

Each clause of the signature has to be taken apart to retrieve the constructor symbol `C`, the variables `Vars1` for the subterms, and the literals `Lits` that specify the types of the subterms. The `Head` of the generated clause uses the term pattern at hand on both the argument and the result position; see the uses of univ below. The body of the clause is obtained by iteration over the literals where `list_to_and/2` turns the resulting list into a conjunction. Thus:

```
generateT__(Skip,Name,Clause,(Head:-Body)) :-
  sort_clause(Clause,_Sort,C,Vars1,Lits),
  Head =.. [Name,X,Y],
  map(generateT___(Skip,Name),Vars1,Lits,Vars2,Gss),
  concat(Gss,Gs),
  list_to_and(Gs,Body),
  X =.. [C|Vars1],
  Y =.. [C|Vars2].
```

The literals from the type clauses are mapped to literals of the traversal clauses. We use univ again to compose recursive calls from appropriate variables and the always same `Name`. We have to cover a number of cases depending on whether we face a sort to be skipped and whether the literal is perhaps an invocation of `map/2` for the case of a list-typed constructor argument. The latter variation is not covered below:

```
generateT___(Skip,Name,Var1,Lit,Var2,Gs) :-
    Lit =.. [Sort,Var1],
    ( member(Sort,Skip) -> Var2 = Var1, Gs = []
    ; G =.. [Name,Var1,Var2], Gs = [G] ).
```

4.3 Traversal customization

The generated traversal must be customized to incorporate problem-specific cases. We reuse the high road's traversal parameters. Thus:

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

The derivation of a customized, generated transformation is taken care of by a metaprogram `completeT/2` which combines all the steps of generation, customization as well as actual assertion of derived clauses to the Prolog fact base and compilation. Thus:

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

`clauses/2` retrieves all clauses for a predicate indicator from the fact base. `abolish/2` removes all clauses for a predicated indicator. `override/2` is a metaprogramming predicate which overrides clauses in one set by clauses from another set (both represented at the term level). Overriding is based on the term pattern of the assumed argument position of the predicate at hand—details are omitted for brevity. (Of course, there is similar metaprogramming support for queries.)

4.4 Optimized traversal

The (possibly hand-written) code of Fig. 1 is actually clever enough to cut off some parts of the traversal that cannot be affected by the transformation or contribute to the query result. That is, the code does not traverse into names (of persons and departments) because no salary terms are possibly contained in name terms. This insight can actually be implemented as an optimizing transformation. For each Candidate sort (among those reachable from the root sort), we determine whether it is 'skippable':

```
skippableSort(Overridden,Candidate) :-
  reachable_sorts(Candidate,Reachable),
  intersection(Reachable,Overridden,[]).
```

That is, given the set of sorts that are Overridden by the problem-specific cases, we can check for any given sort Candidate whether any of the overridden sorts can be reached from it. In our example, the sort name will be discovered as skippable. Overridden is determined by a simple type inference.

```
inferSort(Reachable,Clause,Sort) :-
  clause_to_case(Clause,Term1),
  functor(Term1,C,Arity),
  findall(S, (
       member(S,Reachable),
       sort_to_clauses(S,Clauses),
       member((Head:-_),Clauses),
       Head =.. [_,Term2],
       functor(Term2,C,Arity) ),
    [Sort]).
```

That is, we use clause_to_case/2 in order to extract the argument-term pattern Term1 and with it a specific constructor C from the (overriding) Clause at hand. Then, we query the fact base for the algebraic signatures to find a (unique) clause for C.

5. Epilogue

The high road to SYB in Prolog was long overdue. SYB is practically useful—as we can confirm on the grounds of applications in language processing. The high road also clearly demonstrates the relative fitness of the Prolog language for functional programming-like coding style—even though languages like Curry and Lambda-Prolog are superior in this respect—if one is willing to leave the grounds of (nearly) Standard Prolog.

The low road to SYB suggests important directions for future work. We have used a layman's approach to metaprogramming—without any use of staging or hygienic generation. The question is how to set up a more reliable framework in Prolog (using input from [Taha and Sheard 1997, Jones and Glenstrup 2002, Calcagno et al. 2003, Smaragdakis 2004]) without though harming the seductive simplicity of our present approach. We would like to execute the high road's traversal schemes as generators on the low road and be able to show the correctness of any optimizations done. There is prior work on optimized traversal [Johann and Visser 2003, Cunha and Visser 2007] that should be integrated.

Acknowledgments

I am very grateful for collaboration and exchange on SYB and strategic programming-related subjects with Karl Lieberherr, Simon Peyton Jones, Eelco Visser and Joost Visser over the last few years. I am also grateful for conversations with Pablo Nogueira and others members of the Babel Research Group in Madrid during a working visit this spring.

References

Johan Boye and Jan Maluszynski. Directional Types and the Annotation Method. *Journal of Logic Programming*, 33(3):179–220, 1997.

M.G.J van den Brand, Alex Sellink, and Chris Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 36(2-3):209–266, 2000.

Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Generative Programming and Component Engineering, 2nd International Conference, GPCE 2003, Proceedings*, volume 2830 of *LNCS*, pages 57–76. Springer, 2003.

M. H. M. Cheng, M. H. van Emden, and B. E. Richards. On Warren's method for functional programming in logic. In *Logic Programming, Proceedings of the Seventh International Conference*, pages 546–560. MIT Press, 1990.

Alcino Cunha and Joost Visser. Transformation of structure-shy programs: applied to xpath queries and strategic functions. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 11–20. ACM, 2007.

Michael Hanus. A unified computation model for functional and logic programming. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 80–93. ACM, 1997.

Patricia Johann and Eelco Visser. Strategies for Fusing Logic and Control via Local, Application-Specific Transformations. Technical Report UU-CS-2003-050, Utrecht University, 2003.

Neil D. Jones and Arne J. Glenstrup. Program generation, termination, and binding-time analysis. In *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Proceedings*, volume 2487 of *LNCS*, pages 1–31. Springer, 2002.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 26–37. ACM, 2003.

Karl J. Lieberherr, Boaz Patt-Shamir, and Doug Orleans. Traversals of object structures: Specification and Efficient Implementation. *ACM TOPLAS*, 26(2):370–412, 2004.

Alan Mycroft and Richard A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.

G. Nadathur and D. Miller. An Overview of Lambda-PROLOG. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, pages 810–827. MIT-press, 1988.

Lee Naish. Higher-order logic programming in Prolog. Technical report, Department of Computer Science, University of Melbourne, 1996. Technical Report 96/2.

Lee Naish and Leon Sterling. Stepwise Enhancement and Higher-Order Programming in Prolog. *Journal of Functional and Logic Programming*, 2000(4), 2000.

Jørgen Fischer Nilsson and Andreas Hamfelt. Constructing logic programs with higher-order predicates. In *1995 Joint Conference on Declarative Programming, GULP-PRODE'95, Proceedings*, pages 307–312, 1995.

Jens Palsberg, Boaz Patt-Shamir, and Karl J. Lieberherr. A New Approach to Compiling Adaptive Programs. *Science of Computer Programming*, 29(3):303–326, 1997.

Yannis Smaragdakis. Invited talk: program generators and the tools to make them. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 7–8. ACM, 2004.

Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 203–217. ACM, 1997.

Eelco Visser, Zine el Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 13–26. ACM, 1998.

David H.D. Warren. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd. Chicester, England, 1982.