Logic Program Transformation through Generalization Schemata [Extended Abstract]

Pierre Flener
Department of Computer Engineering and Information Science
Bilkent University
TR–06533 Bilkent, Ankara, Turkey
Email: pf@cs.bilkent.edu.tr

Yves Deville
Department of Computing Science and Engineering
Université Catholique de Louvain
B–1348 Louvain-la-Neuve, Belgium
Email: yde@info.ucl.ac.be

1 Introduction

Programs can be classified according to their construction methodologies, such as divide-and-conquer, top-down decomposition, global search, and so on, or any composition thereof. Informally, a *program schema* [2] is a template program with a fixed control and data flow, but without specific indications about the actual parameters or the actual computations, except that they must satisfy certain constraints. A program schema thus abstracts a whole family of particular programs that can be obtained by instantiating its place-holders to particular parameters or computations, using the specification, the program synthesized so far, and the constraints of the schema. It is therefore interesting to guide program construction by a schema that captures the essence of some methodology. This reflects the conjecture that experienced programmers actually instantiate schemata when programming, which schemata are summaries of their past programming experience.

Moreover, in contrast to traditional programming methodologies where program transformation sequentially follows program construction and is merely based on the syntax of the constructed program, these two phases can actually be interleaved in (semi-)automated program synthesis, based on information generated and exploited during the program construction process.

It would therefore be interesting to pre-compile transformation techniques at the schema-level: a *transformation schema* is a pair $\langle T_1, T_2 \rangle$ of program schemata, such that T_2 is a transformation of T_1, according to some transformation technique. Transformation at the program-level then reduces to (a) selecting a transformation schema $\langle T_1, T_2 \rangle$ such that T_1 covers the given program under some substitution σ, and (b) applying σ to T_2. This is similar to the work of Fuchs *et al.* [4] [also see this volume], except that they have a preliminary abstraction phase where the covering schema of the given program is discovered, whereas we here assume that this program was synthesized in a schema-guided fashion, so that the covering schema is already known.

It sometimes becomes "difficult," if not impossible, to follow the divide-and-conquer methodology. One can then generalize the initial specification and construct a (recursive) program from the generalized specification as well as express the initial problem as a particular case of the generalized one. Paradoxically, the new construction then becomes "easier," if not possible in the first place. As an additional and beneficial side-effect, programs for generalized problems are often more efficient, because they feature tail recursion and/or because the complexity could be reduced through loop merging..

2 Program Transformation by Structural Generalization

In *structural generalization* [1], one generalizes the structure (type) of some parameter. For instance, in *tupling generalization*, an integer parameter would be generalized into an integer-list parameter, and the intended relation would be generalized accordingly.

Example 1: Let `flat(B,F)` hold iff list `F` contains the elements of binary tree `B` as they are visited by a prefix traversal of `B`. We also say that `F` is the prefix representation of `B`. A corresponding "naive" divide-and-conquer logic program could be:

```
flat(void,[])
flat(btree(L,E,R),F) ← flat(L,U), flat(R,V),
   H=[E], append(U,V,I), append(H,I,F)
```

Tupling generalization yields: `flats(Bs,F)` holds iff list `F` is the concatenation of the prefix representations of the elements of binary tree list `Bs`. We obtain the program:

```
flat(B,F) ← flats([B],F)
flats([],[])
flats([void|Bs],F) ← flats(Bs,F)
flats([btree(L,E,R)|Bs],[E|TF]) ← flats([L,R|Bs],TF)
```

In contrast to the "naive" version above, this program has a linear time complexity, a better space complexity, and it can be made tail-recursive in the mode (`in`, `out`). ♦

Result 1: Suppose the specification of the initial problem is (where $\mathcal{R}$ is the intended relation, and $\mathcal{T}_1$, $\mathcal{T}_2$ are the types of its parameters):

`R(X,Y)` $\Leftrightarrow$ $\mathcal{R}$ `[X,Y], where X` $\in \mathcal{T}_1$ `and Y` $\in \mathcal{T}_2$,

and that the initial program is covered by the following divide-and-conquer schema:

```
R(X,Y) ← Minimal(X), Solve(X,Y)                          (Schema 1)
R(X,Y) ← NonMinimal(X), Decompose(X,HX,TX1,TX2),
   R(TX1,TY1), R(TX2,TY2),
   Process(HX,HY), Compose(TY1,TY2,I), Compose(HY,I,Y)
```

where `Compose/3` is associative and has some left/right-identity element `e`.

The *eureka*, that is, the formal specification of the tupling-generalized problem is [3]:

`Rs(Xs,Y)` $\Leftrightarrow$ `(Xs=[]` $\wedge$ `Y=e)`
$\vee$ `(Xs=[X1,X2,...,Xn]` $\wedge$ `R(Xi,Yi)` $\wedge$ $I_1 = Y_1 \wedge$ **Compose**$(I_{i-1}, Y_i, I_i) \wedge Y = I_n$),
`where X` $\in$ `list of` $\mathcal{T}_1$ `and Y` $\in \mathcal{T}_2$.

and the new logic program is the corresponding instance of the following schema:

```
R(X,Y) ← Rs([X],Y)                                       (Schema 2)
Rs(Xs,Y) ← Xs=[], Y=e
Rs(Xs,Y) ← Xs=[X|TXs], Minimal(X),
   Rs(TXs,TY), Solve(X,HY), Compose(HY,TY,Y)
Rs(Xs,Y) ← Xs=[X|TXs], NonMinimal(X), Decompose(X,HX,TX1,TX2),
   Rs([TX1,TX2|TXs],TY), Process(HX,HY), Compose(HY,TY,Y)
```

If `Solve(X,Y)` converts `X` into a constant "size" `Y` and/or if `Process(HX,HY)` converts `HX` into a constant "size" `HY`, then some partial evaluation can be done, which often results in the disappearance of calls to `Compose/3`, if not in the possibility of making the recursive calls iterative ones. ♦

The pair ⟨`Schema 1, Schema 2`⟩ thus constitutes a first generalization schema. More generalization schemata must be pre-compiled, for different numbers of heads and tails of `X`, and for each ordering of composition of `Y` from `HY` and the **`TY`** [3].

3 Program Transformation by Descending Generalization

In *computational generalization* [1], one generalizes a state of computation in terms of "what has already been done" and "what remains to be done." If information about what has already been done is not needed, then it is called *descending generalization.*

Example 2: The "naive" logic program for the well-known `reverse(L,R)` is:

```
reverse([],[])
reverse([HL|TL],R) ← reverse(TL,TR), HR=[HL], append(TR,HR,R)
```

Descending generalization yields: `reverseDesc(L,R,A)` holds iff list `R` is the concatenation of list `A` to the end of the reverse of list `L`. We obtain the program:

```
reverse(L,R) ← reverseDesc(L,R,[])
reverseDesc([],R,R)
reverseDesc([HL|TL],R,A) ← reverseDesc(TL,R,[HL|A])
```

In contrast to the "naive" version above, this program has a linear time complexity, a better space complexity, and it can be made tail-recursive in the mode (`in, out`). ♦

Result 2: Suppose the initial problem exhibits a functional dependency from parameter `X` to parameter `Y`, and that the initial program is covered by the following schema:

```
R(X,Y) ← Minimal(X), Solve(X,Y)                          (Schema 3)
R(X,Y) ← NonMinimal(X), Decompose(X,HX,TX),
   R(TX,TY), Process(HX,HY), Compose(HY,TY,Y)
```

where `Compose/3` is associative with left-identity element `e`.

The *eureka* can be mechanically extracted [1] [3] from the initial program:

```
R-desc(X,Y,A) ⇔ ∃S R(X,S) ∧ Compose(A,S,Y)
```

and the new logic program is the corresponding instance of the following schema:

```
R(X,Y) ← R-desc(X,Y,e)                                   (Schema 4)
R-desc(X,Y,A) ← Minimal(X), Solve(X,S), Compose(A,S,Y)
R-desc(X,Y,A) ← NonMinimal(X), Decompose(X,HX,TX),
   Process(HX,HI), Compose(A,HI,NewA), R-desc(TX,Y,NewA)
```

If `Process(HX,HY)` converts `HX` into a constant "size" `HY` and/or if the intended relation behind `R/2` maps the minimal form of parameter `X` into `e`, and `e` is also a right-identity element of `Compose/3`, then some partial evaluation can be done, which usually results in the disappearance of calls to `Compose/3`, if not in the possibility of making the recursive calls iterative ones. ♦

The pair ⟨`Schema 3`, `Schema 4`⟩ thus constitutes another generalization schema.

4 Conclusion

Both generalization techniques are very suitable for mechanical transformation: all operators of the generalized programs are operators of the initial programs. Given a divide-and-conquer program, a mere inspection of the properties of its solving, processing, and composition operators thus allows the detection of which kinds of generalization are possible, and to which optimizations they would lead. The *eureka* discoveries are compiled away, and the transformations can be completely automated.

References

[1] Y. Deville. *Logic Programming: Systematic Program Development.* Addison-Wesley, 1990.

[2] P. Flener. *Logic Program Synthesis from Incomplete Information.* Kluwer, 1995.

[3] P. Flener and Y. Deville. *Logic Program Transformation through Generalization Schemata.* TR BU-CEIS-95xx, Bilkent University, Ankara (Turkey), 1995.

[4] N.E. Fuchs and M.P.J. Fromherz. Schema-based transformations of logic programs. In Clement and Lau (eds), *Proc. of LOPSTR'91*, pp. 111–125. Springer-Verlag, 1992.