Generalized Generalization Generalizers

Extended Abstract

Halime Büyükyıldız and Pierre Flener

Department of Computer Engineering and Information Science
Faculty of Engineering, Bilkent University, 06533, Bilkent, Ankara, Turkey
Email: {halime, pf}@cs.bilkent.edu.tr

1 Introduction

Schema-guided program construction was studied in logic programming, especially for Prolog programs [4, 5, 10, 14]. Using schemas for logic program transformation was first studied in [9] and extended in [15]. Schema-guided program transformation was also studied in [7, 11]. If the transformation schema embodies some generalization technique, then it is called *generalization schema* (or: generalization generalizer, as in the title of this paper). This paper results from the research that began by investigating the suggestions in [7]. The contributions of this research are:

- pre-compilation of more generalized generalization schemas (tupling and descending), since the schemas in [7] were restricted to special families of divide-and-conquer programs;
- pre-compilation of new generalization schemas that we call *simultaneous-tupling-and-descending generalization schemas*;
- validation of the generalization schemas, based on the notions of correctness of a program, steadfastness of a program in a set of specifications, and equivalence of two programs;
- performance tests for evaluation of the generalization schemas.

Throughout the paper, the word program (resp. procedure) is used to mean typed definite program (resp. procedure). The definitions for correctness of a program, steadfastness of a program in a set of specifications, and equivalence of two programs, which is used in proving the transformation schemas, are given in Appendix A. The *specification* of a predicate R is written in the format:

$$\forall X : \mathcal{T}_X, \forall Y : \mathcal{T}_Y.\ \ \mathcal{I}_R(X) \Rightarrow [R(X,Y) \Leftrightarrow \mathcal{O}_R(X,Y)]$$

where $\mathcal{I}_R(X)$ denotes the *input condition* that must be fulfilled before the execution of the procedure, and $\mathcal{O}_R(X,Y)$ denotes the *output condition* that will be fulfilled after the execution.

1.1 Program Schemas

The notion of program schema was also used in [4, 5, 9, 10, 11, 15], but here we have an additional component, which makes our definition [6, 7, 8] of program schemas different from their definitions.

Definition 1 A *program schema* contains a *template* program with a fixed data flow, but without specific indications about the actual computations, except that they must satisfy certain (*steadfastness*) *constraints*, which are the second component of a schema.

A program schema thus abstracts a whole family of particular programs that can be obtained by instantiating the place-holders of its template to particular computations, using the specification and the program synthesized so far, so that the constraints of the schema are satisfied [6].

To illustrate our purpose, the example below is used as the initial problem throughout the paper.

Example 1 Let $flat(B,F)$ hold iff list F is the infix representation of binary tree B, where *infix representation* means the list representation of the infix traversal of the tree. The constant *void* is used to represent the empty binary tree, and the compound term *bt(L,E,R)* is used to represent a binary tree of root E, left subtree L, and right subtree R.

For the problem above, two different programs and the two program schemas that these programs belong to are explained in the remainder of this section. These program schemas abstract sub-families of divide-and-conquer programs. The divide-and-conquer (DC) program schemas explained in this section are restricted, for pedagogical reasons only, to binary predicates with X as the induction parameter and Y as the result parameter. Another restriction in the schemas is that when X is non-minimal, then X is decomposed into *one* head HX and *two* tails TX_1 and TX_2, so that Y is composed from one head HY (which is the result of processing HX) and two tails TY_1 and TY_2 (which are the results of recursively calling the predicate R with TX_1 and TX_2, respectively) by infix composition (i.e. Y is composed by putting HY between TY_1 and TY_2). For the sake of better understandability, the program template and the program that is an instantiation of that template are given side by side in the figures.

```
R(X,Y) ←                                      flat(B,F) ←
        Minimal(X),                                   B = void,
        Solve(X,Y)                                    F = []
R(X,Y) ←                                      flat(B,F) ←
        NonMinimal(X),                                B = bt(_,_,_),
        Decompose(X,HX,TX1,TX2),                      B = bt(L,E,R),
        R(TX1,TY1),R(TX2,TY2),                        flat(L,FL),flat(R,FR),
        I0 = e,Compose(I0,TY1,I1),                    I0 = [],append(I0,FL,I1),
        Process(HX,HY),Compose(I1,HY,I2),             HF = [E],append(I1,HF,I2),
        Compose(I2,TY2,I3),Y = I3                     append(I2,FR,I3),F = I3
```

Figure 1: Template $DCLR$ and Program 1

```
R(X,Y) ←                                      flat(B,F) ←
        Minimal(X),                                   B = void,
        Solve(X,Y)                                    F = []
R(X,Y) ←                                      flat(B,F) ←
        NonMinimal(X),                                B = bt(_,_,_),
        Decompose(X,HX,TX1,TX2),                      B = bt(L,E,R),
        R(TX1,TY1),R(TX2,TY2),                        flat(L,FL),flat(R,FR),
        I3 = e,Compose(TY2,I3,I2),                    I3 = [],append(FR,I3,I2),
        Process(HX,HY),Compose(HY,I2,I1),             HF = [E],append(HF,I2,I1),
        Compose(TY1,I1,I0),Y = I0                     append(FL,I1,I0),F = I0
```

Figure 2: Template $DCRL$ and Program 2

The constraints on these schemas (i.e. their semantics) are shown in a companion paper [8]. If we denote the functional version of the *Compose* predicate with $\oplus$, then the composition of Y in template $DCLR$ by *left-to-right* (LR) *composition ordering* can be written as $Y = ((e \oplus TY_1) \oplus HY) \oplus TY_2$. The composition of Y in $DCRL$ by *right-to-left* (RL) *composition ordering* can be written as $Y = TY_1 \oplus (HY \oplus (TY_2 \oplus e))$.

Since *append*, which is *Compose* in our *flat* example, is associative and has [] as the identity element, Programs 1 and 2 are equivalent. This shows that the problem families that the two program schemas abstract have an intersection family (resulting in equivalent programs for the problem), if *Compose* satisfies the constraints above.

More generalized schemas for any number of heads and tails, and any composition place of the head in the result parameter (infix composition above is only one possibility) are listed elsewhere [3].

1.2 Schema-Guided Program Transformation

In schema-guided transformation, transformation techniques are pre-compiled at the schema-level.

Definition 2 A *transformation schema* encoding a transformation technique is a triple $\langle S_1, S_2, C\rangle$, where S_1 and S_2 are program schemas and C is a set of (*applicability*) *conditions*. Schema S_2 is expressed using all the predicates of S_1, and the steadfastness constraints on S_2 are thus a superset of those of S_1.

Thus, a schema-guided program transformation system will have a collection of transformation schemas and the transformation of a program P_1 to a program P_2 reduces to: first selecting a transformation schema $\langle S_1, S_2, C\rangle$, such that P_1 is an instance of S_1 under some form of higher-order substitution σ, then the conditions $C\sigma$ must be verified, and finally P_2 is computed by applying σ to the schema S_2. If P_1 is synthesized in a schema-guided fashion (i.e. if σ is known), then transformation can be automated.

Generalization is used to transform a possibly inefficient program into a more efficient one, because the generalization process may provoke a complexity reduction by loop merging and because the output program may be (semi-)tail-recursive (which can be further transformed into an iterative program by an optimizing interpreter). The problem generalization techniques that are used in this paper are explained in detail in [4], but using these techniques for synthesizing and/or transforming a program in a schema-guided fashion was first proposed in [4, 5], and then extended in [7].

Given a program, the generalization process works as follows: first the specification of the initial program is generalized, then a recursive program for the generalized specification is synthesized, and finally a non-recursive program for the initial problem can be written, since the initial problem is a particular case of the generalized one. The two generalization approaches used here are:

1. *Structural generalization*: The intended relation is generalized by generalizing the structure (or: type) of a parameter. If a problem dealing with a term is generalized to a problem dealing with a *list* of terms, then this generalization is called *tupling generalization.*

2. *Computational generalization*: The intended relation is generalized so as to express the general state of a computation in terms of what has been done and what remains to be done. *Ascending* and *descending* generalizations are two particular cases of computational generalization, where in ascending generalization, information about what has already been done is also needed, but in descending generalization the information about what remains to be done is enough.

If schema S_2 is obtained by any method of generalization described above, the transformation schema is called a *generalization schema.*

In the remainder of this paper, we explain in detail how automation of program transformation is achieved by tupling, descending, and simultaneous-tupling-and-descending generalization, with the help of the *flat* example, in Sections 2, 3, and 4, respectively. In Section 5, we discuss, by using the results of performance tests, how a prototype transformation system choose one of the transformation schemas, if the applicability conditions of more than one transformation schema are satisfied by the input program. Finally, in Section 6, we conclude.

2 Program Transformation by Tupling Generalization

Let us generalize the initial specification of our example by using tupling generalization:

$flat\_t(Bs, F)$ iff F is the concatenation of the infix representations of the elements in binary tree list Bs.

The specification above can be constructed by instantiating the following specification $S_{R\_tupling}$:

$$\forall Xs : \textit{list of } \mathcal{T}_X, \forall Y : \mathcal{T}_Y.\ (\forall X : \mathcal{T}_X.\ X \in Xs \Rightarrow \mathcal{I}_R(X)) \Rightarrow [R\_tupling(Xs, Y) \Leftrightarrow (Xs = [\,] \wedge Y = e)$$
$$\vee (Xs = [X_1, X_2, \ldots, X_n] \wedge \ \bigwedge_{i=1}^{n} \mathcal{O}_R(X_i, Y_i) \ \wedge I_1 = Y_1 \wedge \ \bigwedge_{i=2}^{n} \mathcal{O}_C(I_{i-1}, Y_i, I_i) \ \wedge Y = I_n)]$$

where $\mathcal{O}_C$ is the output condition of *Compose*, and $\mathcal{O}_R$ is the output condition of R, and $n \geq 1$.

The tupling generalization schemas (one for each input divide-and-conquer program schema given in Section 1.1) (the proof of TG_1 is given in Appendix B) are:

TG_1 : $\langle$ *DCLR*, *TG*, C_{t1} $\rangle$ where
C_{t1} : - *Compose* is associative
- *Compose* has e as the left and right identity element, where e appears in *DCLR*
- $\forall X : \mathcal{T}_X.\ \mathcal{I}_R(X) \wedge Minimal(X) \Rightarrow \mathcal{O}_R(X, e)$
- $\forall X : \mathcal{T}_X.\ \mathcal{I}_R(X) \Rightarrow [\neg Minimal(X) \Leftrightarrow NonMinimal(X)]$

TG_2 : $\langle$ *DCRL*, *TG*, C_{t2} $\rangle$ where
C_{t2} : - *Compose* is associative
- *Compose* has e as the left and right identity element, where e appears in *DCRL*
- $\forall X : \mathcal{T}_X.\ \mathcal{I}_R(X) \wedge Minimal(X) \Rightarrow \mathcal{O}_R(X, e)$
- $\forall X : \mathcal{T}_X.\ \mathcal{I}_R(X) \Rightarrow [\neg Minimal(X) \Leftrightarrow NonMinimal(X)]$

where the shared template *TG* and the corresponding program for *flat* are in Figure 3.

Note that the predicates in *TG* include *all* of those of *DCLR* (resp. *DCRL*), plus *R_tupling*. The steadfastness constraints of *TG* are thus those of *DCLR* (resp. *DCRL*), plus $S_{R\_tupling}$. The tupling generalization schemas above give rise to full automation of the transformation process. However, this is not the end of an effective transformation, since the transformed program can be simplified by further detection of properties of *Solve*, *Process*, and *Compose*. If *Solve*(X, Y) converts X into a constant 'size' Y, and *Process*(HX, HY) converts HX into a constant 'size' HY, and partial evaluation

```
$R(X,Y) \leftarrow$
        $R\_tupling([X],Y)$
$R\_tupling(Xs,Y) \leftarrow$
        $Xs = [\,], Y = e$
$R\_tupling(Xs,Y) \leftarrow$
        $Xs = [X|TXs]$,
        $Minimal(X)$,
        $R\_tupling(TXs,TY)$,
        $Solve(X,HY), Compose(HY,TY,Y)$
$R\_tupling(Xs,Y) \leftarrow$
        $Xs = [X|TXs]$,
        $NonMinimal(X)$,
        $Decompose(X,HX,TX_1,TX_2)$,
        $Minimal(TX_1)$,
        $R\_tupling([T_2|TXs],TY)$,
        $Solve(TX_1,TY_1)$,
        $I_0 = e, Compose(I_0,TY_1,I_1)$,
        $Process(HX,HHY), Compose(I_1,HHY,I_2)$,
        $HY = I_2, Compose(HY,TY,Y)$
$R\_tupling(Xs,Y) \leftarrow$
        $Xs = [X|TXs]$,
        $NonMinimal(X)$,
        $Decompose(X,HX,TX_1,TX_2)$,
        $NonMinimal(TX_1), Minimal(U)$,
        $Decompose(N,HX,U,TX_2)$,
        $R\_tupling([TX_1,N|TXs],Y)$
```

```
$flat(B,F) \leftarrow$
        $flat\_t([B],F)$
$flat\_t(Bs,F) \leftarrow$
        $Bs = [\,], F = [\,]$
$flat\_t(Bs,F) \leftarrow$
        $Bs = [B|TBs]$,
        $B = void$,
        $flat\_t(TBs,TF)$,
        $HF = [\,], append(HF,TF,F)$
$flat\_t(Bs,F) \leftarrow$
        $Bs = [B|TBs]$,
        $B = bt(\_,\_,\_)$,
        $B = bt(L,E,R)$,
        $L = void$,
        $flat\_t([R|TBs],TF)$,
        $FL = [\,]$,
        $I_0 = [\,], append(I_0,FL,I_1)$,
        $HHF = [E], append(I_1,HHF,I_2)$,
        $HF = I_2, append(HF,TF,F)$
$flat\_t(Bs,F) \leftarrow$
        $Bs = [B|TBs]$,
        $B = bt(\_,\_,\_)$,
        $B = bt(L,E,R)$,
        $L \neq void, U = void$,
        $N = bt(U,E,R)$,
        $flat\_t([L,N|TBs],F)$
```

Figure 3: Template TG and Program 3

of $Compose(HX,TY,Y)$ is possible when the 'size' of HX is constant, then Program 3 can be further simplified. The conjunction $HF = [\,], append(HF,TF,F)$ in the second clause of $flat\_t$ can be transformed into $F = TF$, which can be further unfolded into the head of that clause. The conjunction $FL = [\,], I_0 = [\,], append(I_0,FL,I_1), HHF = [E], append(I_1,HHF,I_2), HF = I_2, append(HF,TF,F)$ in the third clause of $flat\_t$ can be transformed into $F = [E|TF]$, which can also be unfolded into the head of that clause. After doing the simplifications above and further doing syntactic transformations on the transformed program, such as "compiling" $U = void$ into $N = bt(U,E,R)$ in the last clause of $flat\_t$, the $flat\_t$ procedure is semi-tail-recursive. The resulting $flat$ program is:

$$\begin{aligned}
flat(B,F) &\leftarrow \quad flat\_t([B],F)\\
flat\_t([\,],[\,]) &\leftarrow\\
flat\_t([void|TBs],F) &\leftarrow \quad flat\_t(TBs,F)\\
flat\_t([bt(void,E,R)|TBs],[E|TF]) &\leftarrow \quad flat\_t([R|TBs],TF)\\
flat\_t([bt(L,E,R)|TBs],F) &\leftarrow \quad L \neq void, flat\_t([L,bt(void,E,R)|TBs],F)
\end{aligned}$$

3 Program Transformation by Descending Generalization

In [7], schema-guided program transformation by descending generalization is explained in detail and a specific generalization schema is given for an example. In this section, descending generalization schemas are given and we explain how we eliminate the eureka finding step by using the schemas only. Eureka finding [7] is actually finding the specification of the descendingly generalized problem. Descending generalization can also be called *accumulation strategy* (in functional programming [2], also used in logic programming [13]), since it introduces an accumulator parameter, which is progressively extended to the final result. This can also be seen as transformation towards *difference-structure* manipulation.

Since the conditions of each descending generalization schema are different, the process of choosing the appropriate generalization schema for the input divide-and-conquer program is done only by checking the conditions, then the eureka comes for free. Four descending generalization schemas (two for each divide-and-conquer program schema) are given. The first two descending generalization schemas are:

DG_1 : $\langle$ $DCLR$, $DGLR$, C_{d1} $\rangle$ where

```
R(X,Y) ←                                      flat(B,F) ←
        R_descending_1(X,Y,e)                         flat_d_1(B,F,[])
R_descending_1(X,Y,A) ←                        flat_d_1(B,F,A) ←
        Minimal(X),                                   B = void,
        Solve(X,S), Compose(A,S,Y)                    S = [], append(A,S,F)
R_descending_1(X,Y,A) ←                        flat_d_1(B,F,A) ←
        NonMinimal(X),                                B = bt(_,_,_),
        Decompose(X,HX,TX_1,TX_2),                    B = bt(L,E,R),
        Compose(A,e,A_0),                             append(A,[],A_0),
        R_descending_1(TX_1,A_1,A_0),                 flat_d_1(L,A_1,A_0),
        Process(HX,HY), Compose(A_1,HY,A_2),          HF = [E], append(A_1,HF,A_2),
        R_descending_1(TX_2,A_3,A_2), Y = A_3         flat_d_1(R,A_3,A_2), F = A_3
```

Figure 4: Template *DGLR* and Program 4

C_{d1} : - *Compose* is associative
- *Compose* has e as the left identity element, where e appears in *DCLR*

DG_4 : ⟨ *DCRL*, *DGLR*, C_{d4} ⟩ where
C_{d4} : - *Compose* is associative
- *Compose* has e as the left identity element, where e appears in *DCRL*

They have the *same* formal specification (i.e. eureka) for the output program *DGLR*:

$$\forall X:\mathcal{T}_X, \forall Y,A:\mathcal{T}_Y.\ \mathcal{I}_R(X) \Rightarrow [R\_descending_1(X,Y,A) \ \Leftrightarrow\ \exists S:\mathcal{T}_Y.\ \mathcal{O}_R(X,S) \wedge \mathcal{O}_C(A,S,Y)]$$

where $\mathcal{O}_C$ is the output condition of *Compose*, and $\mathcal{O}_R$ is the output condition of R.

If we apply DG_1 or DG_4 to our *flat* example (since the conditions of both schemas are satisfied by Programs 1 and 2), then the descending generalization of the initial specification is:

$flat\_d_1(B,F,A)$ iff list F is the concatenation of list A and the infix representation of binary tree B.

The transformed program and the template it belongs to (i.e. *DGLR*) are in Figure 4.

The other two descending generalization schemas (the proof of DG_2 is given in Appendix C) are:

DG_2 : ⟨ *DCLR*, *DGRL*, C_{d2} ⟩ where
C_{d2} : - *Compose* is associative
- *Compose* has e as the right identity element, where e appears in *DCLR*

DG_3 : ⟨ *DCRL*, *DGRL*, C_{d3} ⟩ where
C_{d3} : - *Compose* is associative
- *Compose* has e as the right identity element, where e appears in *DCRL*

They have the *same* formal specification (i.e. eureka) for the output program *DGRL*:

$$\forall X:\mathcal{T}_X, \forall Y,A:\mathcal{T}_Y.\ \mathcal{I}_R(X) \Rightarrow [R\_descending_2(X,Y,A) \ \Leftrightarrow\ \exists S:\mathcal{T}_Y.\ \mathcal{O}_R(X,S) \wedge \mathcal{O}_C(S,A,Y)]$$

where $\mathcal{O}_C$ is the output condition of *Compose*, and $\mathcal{O}_R$ is the output condition of R.

If we apply DG_2 or DG_3 to our *flat* example (since the conditions of both schemas are satisfied by Programs 1 and 2), then the descending generalization of the initial specification is:

$flat\_d_2(B,F,A)$ iff list F is the concatenation of the infix representation of binary tree B and list A.

The transformed program and the template it belongs to (i.e. *DGRL*) are in Figure 5.

The reason why we call the descendingly generalized program schemas '*DGLR*' and '*DGRL*' is similar to the reason why we call the divide-and-conquer program schemas *DCLR* and *DCRL*, respectively. In descending generalization, the composition ordering for extending the accumulator parameter in template *DGLR* is from *left-to-right* and the composition ordering for extending the accumulator parameter in template *DGRL* is from *right-to-left*.

Since *append*, which is *Compose* of Program 1, has [] as left and right identity element, both conditions of descending generalization schemas DG_1 and DG_2 are satisfied by Program 1. Similarly, both conditions of descending generalization schemas DG_3 and DG_4 are satisfied by Program 2. Syntactic simplifications may be done on Programs 4 and 5, but it can be observed that Program 5 is more efficient than Program 4,

$$\begin{array}{l}
R(X,Y) \leftarrow \\
\qquad R\_descending_2(X,Y,e) \\
R\_descending_2(X,Y,A) \leftarrow \\
\qquad Minimal(X), \\
\qquad Solve(X,S), Compose(S,A,Y) \\
R\_descending_2(X,Y,A) \leftarrow \\
\qquad NonMinimal(X), \\
\qquad Decompose(X,HX,TX_1,TX_2), \\
\qquad A = A_3, R\_descending_2(TX_2,A_2,A_3), \\
\qquad Process(HX,HY), Compose(HY,A_2,A_1), \\
\qquad R\_descending_2(TX_1,A_0,A_1), \\
\qquad Compose(e,A_0,Y)
\end{array}$$

$$\begin{array}{l}
flat(B,F) \leftarrow \\
\qquad flat\_d_2(B,F,[\,]) \\
flat\_d_2(B,F,A) \leftarrow \\
\qquad B = void, \\
\qquad S = [\,], append(S,A,F) \\
flat\_d_2(B,F,A) \leftarrow \\
\qquad B = bt(\_,\_,\_), \\
\qquad B = bt(L,E,R), \\
\qquad A = A_3, flat\_d_2(R,A_2,A_3), \\
\qquad HF = [E], append(HF,A_2,A_1), \\
\qquad flat\_d_2(L,A_0,A_1), \\
\qquad append([\,],A_0,F)
\end{array}$$

Figure 5: Template *DGRL* and Program 5

since we can also eliminate *append* by partial evaluation during simplification. For example, Program 5 may be simplified into:

$$\begin{array}{rl}
flat(B,F) \leftarrow & flat\_d_2(B,F,[\,]) \\
flat\_d_2(void,A,A) \leftarrow & \\
flat\_d_2(bt(L,E,R),F,A) \leftarrow & flat\_d_2(R,NA,A), flat\_d_2(L,F,[E|NA])
\end{array}$$

Our next plan is to extend the descending generalization schemas by adding the "simplifiability" conditions to the schemas, since we want to achieve the objective that the system chooses in one step the transformation that yields the most efficient output program for input problems like *flat*, where the conditions of each schema for the same input program are satisfied.

4 Program Transformation by Simultaneous-Tupling-and-Descending Generalization

While working on constructing possible generalized generalization schemas for different input program schemas, we also tried to apply descending generalization to a tupling generalized problem, and vice versa. The generalization schemas that we explain in this section are the results of this work. We call them simultaneous-tupling-and-descending generalization schemas, although the reader may notice by looking at the specification of the generalized problem that the process may also be thought of as applying descending generalization to a tupling generalized problem.

Like we did in Section 3 for descending generalization, four simultaneous-tupling-and-descending generalization schemas are given. The first two are:

TDG_1 : $\langle$ *DCLR*, *TDGLR*, C_{tdlr} $\rangle$ where
C_{tdlr} : - *Compose* is associative
- *Compose* has e as the left and right identity element, where e appears in *DCLR*
- $\forall X : \mathcal{T}_X.\ \mathcal{I}_R(X) \wedge Minimal(X) \Rightarrow \mathcal{O}_R(X,e)$
- $\forall X : \mathcal{T}_X.\ \mathcal{I}_R(X) \Rightarrow [\neg Minimal(X) \Leftrightarrow NonMinimal(X)]$

TDG_4 : $\langle$ *DCRL*, *TDGLR*, C_{tdrl} $\rangle$ where
C_{tdrl} : - *Compose* is associative
- *Compose* has e as the left and right identity element, where e appears in *DCRL*
- $\forall X : \mathcal{T}_X.\ \mathcal{I}_R(X) \wedge Minimal(X) \Rightarrow \mathcal{O}_R(X,e)$
- $\forall X : \mathcal{T}_X.\ \mathcal{I}_R(X) \Rightarrow [\neg Minimal(X) \Leftrightarrow NonMinimal(X)]$

If we apply TDG_1 or TDG_4 to our *flat* example (since the conditions of both schemas are satisfied by Programs 1 and 2), then the simultaneous-tupling-and-descending generalization of the initial specification is:

$flat\_td_1(Bs,F,A)$ iff list F is the concatenation of list A and the infix representations of the elements in binary tree list Bs.

The transformed program and the template it belongs to (i.e. *TDGLR*) are in Figure 6.

The other two simultaneous-tupling-and-descending generalization schemas are: TDG_2 : $\langle$ *DCLR*, *TDGRL*, C_{tdlr} $\rangle$ and TDG_3 : $\langle$ *DCRL*, *TDGRL*, C_{tdrl} $\rangle$, where C_{tdlr} and C_{tdrl} are as above.

```
$R(X,Y) \leftarrow$
        $R\_td_1([X],Y,e)$
$R\_td_1(Xs,Y,A) \leftarrow$
        $Xs = [\,], Y = A$
$R\_td_1(Xs,Y,A) \leftarrow$
        $Xs = [X|TXs]$,
        $Minimal(X), Solve(X,HY)$,
        $A = A_0, Compose(A_0,HY,A_1)$,
        $R\_td_1(TXs,A_2,A_1), Y = A_2$
$R\_td_1(Xs,Y,A) \leftarrow$
        $Xs = [X|TXs]$,
        $NonMinimal(X)$,
        $Decompose(X,HX,TX_1,TX_2)$,
        $Minimal(TX_1), Solve(TX_1,TY_1)$,
        $A = A_0, Compose(A_0,TY_1,A_1)$,
        $Process(HX,HY), Compose(A_1,HY,A_2)$,
        $R\_td_1([TX_2|TXs],A_3,A_2), Y = A_3$
$R\_td_1(Xs,Y,A) \leftarrow$
        $Xs = [X|TXs]$,
        $NonMinimal(X)$,
        $Decompose(X,HX,TX_1,TX_2)$,
        $NonMinimal(TX_1), Minimal(U)$,
        $Decompose(N,HX,U,TX_2)$,
        $R\_td_1([TX_1,N|TXs],Y,A)$
```

```
$flat(B,F) \leftarrow$
        $flat\_td_1([B],F,[\,])$
$flat\_td_1(Bs,F,A) \leftarrow$
        $Bs = [\,], F = A$
$flat\_td_1(Bs,F,A) \leftarrow$
        $Bs = [B|TBs]$,
        $B = void, HF = [\,]$,
        $A = A_0, append(A_0,HF,A_1)$,
        $flat\_td_1(TBs,A_2,A_1), F = A_2$
$flat\_td_1(Bs,F,A) \leftarrow$
        $Bs = [B|TBs]$,
        $B = bt(\_,\_,\_)$,
        $B = bt(L,E,R)$,
        $L = void, FL = [\,]$,
        $A = A_0, append(A_0,FL,A_1)$,
        $HF = [E], append(A_1,HF,A_2)$,
        $flat\_td_1([R|TBs],A_3,A_2), F = A_3$
$flat\_td_1(Bs,F,A) \leftarrow$
        $Bs = [B|TBs]$,
        $B = bt(\_,\_,\_)$,
        $B = bt(L,E,R)$,
        $L \neq void, U = void$,
        $N = bt(U,E,R)$,
        $flat\_td_1([L,N|TBs],F,A)$
```

Figure 6: Template $TDGLR$ and Program 6

If we apply TDG_2 or TDG_3 to our *flat* example (since the conditions of both schemas are satisfied by Programs 1 and 2), then the simultaneous-tupling-and-descending generalization of the initial specification is:

$flat\_td_2(Bs,F,A)$ iff list F is the concatenation of the infix representations of the elements in binary tree list Bs and list A.

The transformed program and the template it belongs to (i.e $TDGRL$) are in Figure 7.

We call the two simultaneous-tupling-and-descending generalization program schemas '$TDGLR$' and '$TDGRL$' by using the same reasoning we made for calling the descending generalization program schemas $DGLR$ and $DGRL$, respectively.

If we use properties of *append*, the generalized programs can be further simplified.

5 Evaluation of Generalization Schemas

We now evaluate the generalization schemas using performance tests done on the input and output programs of each generalization schema. The programs are executed and tested using Mercury 0.6 on a SPARCstation 4. Since the programs are really short, the predicates were called 1,000 times to achieve meaningful timing results. In Table 1, the results of the performance tests for five selected problems are shown, where each column heading represents the schema to which the program written for the problem of that row belongs. The timing results are normalized wrt the DCLR column.

problems	DCLR	DCRL	TG	DGLR	DGRL	TDGLR	TDGRL
Prefix *flat*	1.00	0.88	0.50	3.38	0.25	3.38	0.50
Infix *flat*	1.00	0.76	0.65	3.06	0.18	3.35	0.59
Postfix *flat*	1.00	0.80	0.45	2.70	0.15	2.64	0.50
reverse	1.00	1.00	0.15	0.99	0.02	1.02	0.15
quicksort	1.00	0.93	1.03	2.33	0.76	2.60	1.07

Table 1: Performance Tests Results

By looking at Table 1, we can say that the most time efficient programs for our example test problems are the ones that belong to the $DGRL$ program schema. It is so because the examples we chose are such that

```
$R(X, Y) \leftarrow$
        $R\_td_2([X], Y, e)$
$R\_td_2(Xs, Y, A) \leftarrow$
        $Xs = [\,], Y = A$
$R\_td_2(Xs, Y, A) \leftarrow$
        $Xs = [X|TXs]$,
        $Minimal(X), Solve(X, HY)$,
        $A = A_2, R\_td_2(TXs, A_1, A_2)$,
        $Compose(HY, A_1, A_0), Y = A_0$
$R\_td_2(Xs, Y, A) \leftarrow$
        $Xs = [X|TXs]$,
        $NonMinimal(X)$,
        $Decompose(X, HX, TX_1, TX_2)$,
        $Minimal(TX_1), Solve(TX_1, TY_1)$,
        $A = A_3, R\_td_2([TX_2|TXs], A_2, A_3)$,
        $Process(HX, HY), Compose(HY, A_2, A_1)$,
        $Compose(TY_1, A_1, A_0), Y = A_0$
$R\_td_2(Xs, Y, A) \leftarrow$
        $Xs = [X|TXs]$,
        $NonMinimal(X)$,
        $Decompose(X, HX, TX_1, TX_2)$,
        $NonMinimal(TX_1), Minimal(U)$,
        $Decompose(N, HX, U, TX_2)$,
        $R\_td_2([TX_1, N|TXs], Y, A)$
```

```
$flat(B, F) \leftarrow$
        $flat\_td_2([B], F, [\,])$
$flat\_td_2(Bs, F, A) \leftarrow$
        $Bs = [\,], F = A$
$flat\_td_2(Bs, F, A) \leftarrow$
        $Bs = [B|TBs]$,
        $B = void, HF = [\,]$,
        $A = A_2, flat\_td_2(TBs, A_2, A_3)$,
        $append(HF, A_1, A_0), F = A_0$
$flat\_td_2(Bs, F, A) \leftarrow$
        $Bs = [B|TBs]$,
        $B = bt(\_, \_, \_)$,
        $B = bt(L, E, R)$,
        $L = void, FL = [\,]$,
        $A = A_3, flat\_td_2([R|TBs], A_2, A_3)$,
        $HF = [E], append(HF, A_2, A_1)$,
        $append(FL, A_1, A_0), F = A_0$
$flat\_td_2(Bs, F, A) \leftarrow$
        $Bs = [B|TBs]$,
        $B = bt(\_, \_, \_)$,
        $B = bt(L, E, R)$,
        $L \neq void, U = void$,
        $N = bt(U, E, R)$,
        $flat\_td_2([L, N|TBs], F, A)$
```

Figure 7: Template *TDGRL* and Program 7

the best transformation schemas (which also give rise to the best simplifications) are DG_2 and DG_3 for the *DCLR* and *DCRL* input programs, respectively, since *Compose* is *append* for all the programs. The reason above is also the reason why the *DCRL* programs are more efficient than the *DCLR* programs. For all programs with left-to-right composition ordering or accumulator extension, we lose efficiency, because the programs can't be further simplified and we can't eliminate *append*. In the descending and simultaneous-tupling-and-descending generalization schemas, this was difficult, since the percentage of the total running time of the program used by *Compose* is very high, because of the increase in the 'size' of the input parameter. Except for *quicksort*, for all the examples, the *TG* and *TDGRL* programs are more efficient than the initial *DCLR* and *DCRL* programs, which supports our claim that we can also gain efficiency by tupling and simultaneous-tupling-and-descending generalizations. The reason why we lose efficiency in tupling generalization of *quicksort* is that we increase the calls to *Decompose*, which is *partition* in *quicksort* and is a very costly operation, although we eliminate *Compose*. If we try to compare simultaneous-tupling-and-descending generalization schemas with the other generalization schemas, we may think that we do not gain anything. The reader has to remember that this generalization can also be thought of as applying a descending generalization to a tupling generalized problem, and for infix *flat*, we gain some efficiency over tupling generalization. The results also show that, except for *quicksort*, the *TDGRL* programs are more efficient than the input divide-and-conquer programs.

6 Conclusion and Future Work

Program generalization can be automated by using generic generalization schemas, since we fully automate the eureka discovery (i.e. finding the specification of the generalized problem). Further detection of properties of the operators in the input template helps us to simplify the program resulting from transformation, which can be done by constructing global plans for each generalization schema. Our next aim is to develop a prototype transformation system supporting this approach.

References

[1] T. Batu. *Schema-Guided Transformations of Logic Algorithms.* Senior Project Report, Bilkent University, Department of Computer Science, 1996.

[2] R.S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.

[3] H. Büyükyıldız. *Generic Program Transformation Schemata.* Technical Report, in preparation. Bilkent University, Department of Computer Science, 1997.

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

[5] Y. Deville and J. Burnay. Generalization and program Schemata: a step towards computer-aided construction of logic programs. In: E.L. Lusk and R.A. Overbeek (eds), *Proc. of NACLP'89*, pp. 409–425. The MIT Press, 1989.

[6] P. Flener. *Logic Program Schemata: Synthesis and Analysis.* Technical Report BU-CEIS-9502. Bilkent University, Department of Computer Science, 1995.

[7] P. Flener and Y. Deville. Logic program transformation through generalization schemata. Extended abstract in: M. Proietti (ed), *Proc. of LOPSTR'95*, pp. 171–173. LNCS 1048. Springer-Verlag, 1996. Full version in: M. Proietti (ed), *Pre-proc. of LOPSTR'95.*

[8] P. Flener, K.-K. Lau, and M. Ornaghi. *On Correct Program Schemas.* Also submitted to *LOPSTR'97.*

[9] N.E. Fuchs and M.P.J. Fromherz. Schema-based transformation of logic programs. In: T. Clement and K.-K. Lau (eds), *Proc. of LOPSTR'91*, pp. 111–125. Springer Verlag, 1992.

[10] T.S. Gegg-Harrison. Representing logic program schemata in λProlog. In: L. Sterling (ed), *Proc. of ICLP'95*, pp. 467–481. The MIT Press, 1995.

[11] T.S. Gegg-Harrison. Extensible logic program schemata. In: J. Gallagher (ed), *Proc. of LOPSTR'96.* LNCS 1207. Springer-Verlag, 1997.

[12] K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. *Steadfast Logic Programs.* Submitted.

[13] A. Pettorossi and M. Proietti. Transformation of logic programs: foundations and techniques. *Journal of Logic Programming*, 19(20):261–320, 1994.

[14] L.S. Sterling and M. Kirschenbaum. Applying techniques to skeletons. In: J.-M. Jacquet (ed), *Constructing Logic Programs*, pp. 127–140, John Wiley, 1993.

[15] W.W. Vasconcelos and N.E. Fuchs. An opportunistic approach for logic program analysis and optimisation using enhanced schema-based transformations. In: M. Proietti (ed), *Proc. of LOPSTR'95*, pp. 174–188. LNCS 1048. Springer-Verlag, 1996.

A Correctness and Equivalence Criteria

We will give the correctness and equivalence criteria that we use to prove our generalization schemas. The definitions for correctness of a program and steadfastness of a program in a set of specifications are similar to the definitions in [8, 12]. Our definitions are all model-theoretic.

We give again the format of the *specification* S_R of a predicate R:

$$\forall X : \mathcal{T}_X, \forall Y : \mathcal{T}_Y.\ \mathcal{I}_R(X) \Rightarrow [R(X,Y) \Leftrightarrow \mathcal{O}_R(X,Y)]$$

where $\mathcal{I}_R(X)$ denotes the *input condition* that must be fulfilled before the execution of the procedure, and $\mathcal{O}_R(X,Y)$ denotes the *output condition* that will be fulfilled after the execution.

An *open program* is a program where some of the predicates appearing in the clause bodies are not appearing in any heads of clauses, and these predicates are called *undefined* predicates. If all the predicates appearing in the program are *defined*, then the program is called a *closed program.* We do not consider mutually recursive programs.

Definition 3 (Correctness of a Closed Program)
Let P be a closed program for relation R. We say that P is (*totally*) *correct* wrt its specification S_R iff, for any ground term t of $\mathcal{T}_X$ such that $\mathcal{I}_R(t)$ holds, the following condition holds: $P \models R(t,u)$ iff $\mathcal{O}_R(t,u)$, for every ground term u of $\mathcal{T}_Y$.

If we replace 'iff' by 'implies' in the condition above, then P is said to be *partially correct* wrt S_R, and if we replace 'iff' by 'if', then P is said to be *complete* wrt S_R.

This kind of correctness is not entirely satisfactory, for two reasons. First, it defines the correctness of P in terms of the procedures for the predicates in its clause bodies, rather than in terms of their specifications. Second, P must be a closed program, even though it might be desirable to discuss the correctness of P without having to fully implement it. So, the abstraction achieved through the introduction (and specification) of the predicates in its clause bodies is wasted. This leads us to the notion of steadfastness, which we only define here for the most interesting case, namely where *all* predicates occurring in the clause bodies are also known by their specifications.

Definition 4 (Steadfastness of an Open Program in a Set of Specifications)
Let:

- P be an open program for relation R
- $q_1, \ldots, q_m$ be all the undefined predicate names appearing in P
- $S_1, \ldots, S_m$ be the specifications of $q_1, \ldots, q_m$.

We say that P is *steadfast* wrt its specification S_R in $\{S_1, \ldots, S_m\}$ iff, for any closed program P_S such that

- P_S is correct wrt each specification S_j $(1 \leq j \leq m)$
- P_S contains no occurrences of the predicates defined in P,

it is the case that $P \cup P_S$ is correct wrt S_R.

Steadfastness has the following interesting property:
Property:
Let:

- P be an open program for a relation R
- $p_1, \ldots, p_t$ be all the defined predicate names appearing in P (including R thus)
- $q_1, \ldots, q_m$ be all the undefined predicate names appearing in P
- $S_1, \ldots, S_m$ be the specifications of $q_1, \ldots, q_m$.

We have that, for $t \geq 2$, P is steadfast wrt S_R in $\{S_1, \ldots, S_m\}$ if every P_i $(1 \leq i \leq t)$ is steadfast wrt the specification of p_i in the set of the specifications of all undefined predicates in P_i, where P_i is a program for p_i, such that $P = \bigcup_{i=1}^{t} P_i$.

For program equivalence, we do not require the two programs to have the same models, because this would not make much sense in a program generalization setting, where the transformed program features predicates that were not in the initially given program. That is why our program equivalence criterion establishes equivalence wrt the specification of a common predicate (usually the root of their call-hierarchies).

Definition 5 (Equivalence of Two Open Programs)
Let P and Q be two open programs for a predicate R. We say that P is *equivalent to* Q wrt the specification S_R iff the following two conditions hold:

(a) P is steadfast wrt S_R in $\{S_1, \ldots, S_m\}$, where $S_1, \ldots, S_m$ are the specifications of $p_1, \ldots, p_m$, which are all the undefined predicate names appearing in P

(b) Q is steadfast wrt S_R in $\{S'_1, \ldots, S'_t\}$, where $S'_1, \ldots, S'_t$ are the specifications of $q_1, \ldots, q_t$, which are all the undefined predicate names appearing in Q.

Since the 'is equivalent to' relation is symmetric, we also say that P and Q are *equivalent* wrt S_R.

B Proof of TG_1

As a reminder, we give again the generalization schema TG_1.

TG_1 : $\langle$ $DCLR$, TG, C_{t1} $\rangle$ where
C_{t1} : (1) $Compose$ is associative
(2) $Compose$ has e as the left and right identity element, where e appears in $DCLR$
(3) $\forall X : \mathcal{T}_X.\ \mathcal{I}_R(X) \wedge Minimal(X) \Rightarrow \mathcal{O}_R(X, e)$
(4) $\forall X : \mathcal{T}_X.\ \mathcal{I}_R(X) \Rightarrow [\neg Minimal(X) \Leftrightarrow NonMinimal(X)]$

and templates $DCLR$ and TG are in Figure 8.

```
$R(X, Y) \leftarrow$
        $Minimal(X),$
        $Solve(X, Y)$
$R(X, Y) \leftarrow$
        $NonMinimal(X),$
        $Decompose(X, HX, TX_1, TX_2),$
        $R(TX_1, TY_1), R(TX_2, TY_2),$
        $I_0 = e, Compose(I_0, TY_1, I_1),$
        $Process(HX, HY), Compose(I_1, HY, I_2),$
        $Compose(I_2, TY_2, I_3), Y = I_3$
```

```
$R(X, Y) \leftarrow$
        $R\_tupling([X], Y)$
$R\_tupling(Xs, Y) \leftarrow$
        $Xs = [\,], Y = e$
$R\_tupling(Xs, Y) \leftarrow$
        $Xs = [X|TXs],$
        $Minimal(X),$
        $R\_tupling(TXs, TY),$
        $Solve(X, HY), Compose(HY, TY, Y)$
$R\_tupling(Xs, Y) \leftarrow$
        $Xs = [X|TXs],$
        $NonMinimal(X),$
        $Decompose(X, HX, TX_1, TX_2),$
        $Minimal(TX_1),$
        $R\_tupling([TX_2|TXs], TY),$
        $Solve(TX_1, TY_1),$
        $I_0 = e, Compose(I_0, TY_1, I_1),$
        $Process(HX, HHY), Compose(I_1, HHY, I_2),$
        $HY = I_2, Compose(HY, TY, Y)$
$R\_tupling(Xs, Y) \leftarrow$
        $Xs = [X|TXs],$
        $NonMinimal(X),$
        $Decompose(X, HX, TX_1, TX_2),$
        $NonMinimal(TX_1),$
        $Minimal(U),$
        $Decompose(N, HX, U, TX_2),$
        $R\_tupling([TX_1, N|TXs], Y)$
```

Figure 8: Templates $DCLR$ and TG

The specification S_R of predicate R is:

$$\forall X : \mathcal{T}_X, \forall Y : \mathcal{T}_Y.\ \mathcal{I}_R(X) \Rightarrow [R(X,Y) \Leftrightarrow \mathcal{O}_R(X,Y)]$$

The specification $S_{R\_tupling}$ of predicate $R\_tupling$ is:

$$\begin{array}{l}\forall Xs : \textit{list of}\ \mathcal{T}_X, \forall Y : \mathcal{T}_Y.\ (\forall X : \mathcal{T}_X.\ X \in Xs \Rightarrow \mathcal{I}_R(X)) \Rightarrow [R\_tupling(Xs,Y) \Leftrightarrow \\ (Xs = [\,] \wedge Y = e) \\ \vee (Xs = [X_1, \ldots, X_q] \wedge \ \bigwedge_{i=1}^{q} \mathcal{O}_R(X_i, Y_i) \ \wedge I_1 = Y_1 \wedge \ \bigwedge_{i=2}^{q} \mathcal{O}_C(I_{i-1}, Y_i, I_i) \ \wedge Y = I_q)]\end{array}$$

where $\mathcal{O}_C$ is the output condition of *Compose*, $\mathcal{O}_R$ is the output condition of R, and $q \geq 1$.

To prove the generalization schema TG_1, we have to prove that templates $DCLR$ and TG are *equivalent* wrt S_R. By Definition 5, this holds iff the following two conditions hold:

(a) $DCLR$ is steadfast wrt S_R in $\mathcal{S} = \{S_{Minimal}, S_{NonMinimal}, S_{Solve}, S_{Decompose}, S_{Compose}\}$, where $S_{Minimal}$, $S_{NonMinimal}$, S_{Solve}, $S_{Decompose}$, $S_{Compose}$ are the specifications of $Minimal$, $NonMinimal$, *Solve*, *Decompose*, *Compose*, which are all the undefined predicate names appearing in $DCLR$.

(b) TG is also steadfast wrt S_R in $\mathcal{S}$.

Note that the sets $\{S_1, \ldots, S_m\}$ and $\{S'_1, \ldots, S'_t\}$ in Definition 5 are equal when Q is obtained by generalization of P.

In program transformation, we assume that the input program, here template $DCLR$, is steadfast wrt S_R in $\mathcal{S}$, so condition (a) always holds.

To prove equivalence, we have to prove condition (b). We will use the property of steadfastness: TG is steadfast wrt S_R in $\mathcal{S}$ if $P_{R\_tupling}$ is steadfast wrt $S_{R\_tupling}$ in $\mathcal{S}$, where $P_{R\_tupling}$ is the procedure for $R\_tupling$, and P_R is steadfast wrt S_R in $\{S_{R\_tupling}\}$, where P_R is the procedure for R.

To prove that $P_{R\_tupling}$ is steadfast wrt $S_{R\_tupling}$ in $\mathcal{S}$, we do a constructive forward proof that we begin with $S_{R\_tupling}$ and from which we try to obtain $P_{R\_tupling}$.

If we separate the cases of $q \geq 1$ by $q = 1 \vee q \geq 2$, then $S_{R\_tupling}$ becomes:

$$\begin{array}{l}\forall Xs : \textit{list of}\ \mathcal{T}_X, \forall Y : \mathcal{T}_Y.\ (\forall X : \mathcal{T}_X.\ X \in Xs \Rightarrow \mathcal{I}_R(X)) \Rightarrow [R\_tupling(Xs,Y) \Leftrightarrow \\ (Xs = [\,] \wedge Y = e) \\ \vee (Xs = [X_1] \wedge \mathcal{O}_R(X_1, Y_1) \wedge Y_1 = I_1 \wedge Y = I_1) \\ \vee (Xs = [X_1, X_2, \ldots, X_q] \wedge \ \bigwedge_{i=1}^{q} \mathcal{O}_R(X_i, Y_i) \ \wedge I_1 = Y_1 \wedge \ \bigwedge_{i=2}^{q} \mathcal{O}_C(I_{i-1}, Y_i, I_i) \ \wedge Y = I_q)]\end{array}$$

where $q \geq 2$.

By using applicability conditions (1) and (2):

$$\begin{array}{l}\forall Xs : \textit{list of}\ \mathcal{T}_X, \forall Y : \mathcal{T}_Y.\ (\forall X : \mathcal{T}_X.\ X \in Xs \Rightarrow \mathcal{I}_R(X)) \Rightarrow [R\_tupling(Xs,Y) \Leftrightarrow \\ (Xs = [\,] \wedge Y = e) \\ \vee (Xs = [X_1|TXs] \wedge TXs = [\,] \wedge \mathcal{O}_R(X_1, Y_1) \wedge Y_1 = I_1 \wedge TY = e \wedge \mathcal{O}_C(I_1, TY, Y)) \\ \vee (Xs = [X_1|TXs] \wedge TXs = [X_2, \ldots, X_q] \wedge \ \bigwedge_{i=1}^{q} \mathcal{O}_R(X_i, Y_i) \ \wedge Y_1 = I_1 \wedge Y_2 = I_2 \wedge \\ \bigwedge_{i=3}^{q} \mathcal{O}_C(I_{i-1}, Y_i, I_i) \ \wedge TY = I_q \wedge \mathcal{O}_C(I_1, TY, Y))]\end{array}$$

where $q \geq 2$.

By folding using $S_{R\_tupling}$, and renaming:

$$\begin{array}{l}\forall Xs : \textit{list of}\ \mathcal{T}_X, \forall Y : \mathcal{T}_Y.\ (\forall X : \mathcal{T}_X.\ X \in Xs \Rightarrow \mathcal{I}_R(X)) \Rightarrow [R\_tupling(Xs,Y) \Leftrightarrow \\ (Xs = [\,] \wedge Y = e) \\ \vee (Xs = [X|TXs] \wedge \mathcal{O}_R(X, HY) \wedge R\_tupling(TXs, TY) \wedge \mathcal{O}_C(HY, TY, Y))]\end{array}$$

By taking the 'decompletion':

$$\begin{array}{ll}\textit{clause}\ 1: & R\_tupling(Xs,Y) \leftarrow \\ & \quad Xs = [\,], Y = e \\ \textit{clause}\ 2: & R\_tupling(Xs,Y) \leftarrow \\ & \quad Xs = [X|TXs], R(X, HY), \\ & \quad R\_tupling(TXs, TY), Compose(HY, TY, Y)\end{array}$$

By unfolding clause 2 wrt $R(X, HY)$ using $DCLR$, and using the assumption that $DCLR$ is steadfast wrt S_R in $\mathcal{S}$:

$$\begin{array}{ll}
\mathit{clause}\ 3: & \mathit{R\_tupling}(Xs, Y) \leftarrow \\
& \quad Xs = [X|TXs], \\
& \quad \mathit{Minimal}(X), \\
& \quad \mathit{R\_tupling}(TXs, TY), \\
& \quad \mathit{Solve}(X, HY), \mathit{Compose}(HY, TY, Y) \\
\mathit{clause}\ 4: & \mathit{R\_tupling}(Xs, Y) \leftarrow \\
& \quad Xs = [X|TXs], \\
& \quad \mathit{NonMinimal}(X), \mathit{Decompose}(X, HX, TX_1, TX_2), \\
& \quad R(TX_1, TY_1), R(TX_2, TY_2), \\
& \quad I_0 = e, \mathit{Compose}(I_0, TY_1, I_1), \\
& \quad \mathit{Process}(HX, HHY), \mathit{Compose}(I_1, HHY, I_2), \\
& \quad \mathit{Compose}(I_2, TY_2, I_3), HY = I_3, \\
& \quad \mathit{R\_tupling}(TXs, TY), \mathit{Compose}(HY, TY, Y)
\end{array}$$

By introducing $\mathit{Minimal}(TX_1) \vee \mathit{NonMinimal}(TX_1)$ in clause 4, using applicability condition (4):

$$\begin{array}{ll}
\mathit{clause}\ 5: & \mathit{R\_tupling}(Xs, Y) \leftarrow \\
& \quad Xs = [X|TXs], \\
& \quad \mathit{NonMinimal}(X), \mathit{Decompose}(X, HX, TX_1, TX_2), \\
& \quad \mathit{Minimal}(TX_1), R(TX_1, TY_1), R(TX_2, TY_2), \\
& \quad I_0 = e, \mathit{Compose}(I_0, TY_1, I_1), \\
& \quad \mathit{Process}(HX, HHY), \mathit{Compose}(I_1, HHY, I_2), \\
& \quad \mathit{Compose}(I_2, TY_2, I_3), HY = I_3, \\
& \quad \mathit{R\_tupling}(TXs, TY), \mathit{Compose}(HY, TY, Y) \\
\mathit{clause}\ 6: & \mathit{R\_tupling}(Xs, Y) \leftarrow \\
& \quad Xs = [X|TXs], \\
& \quad \mathit{NonMinimal}(X), \mathit{Decompose}(X, HX, TX_1, TX_2), \\
& \quad \mathit{NonMinimal}(TX_1), R(TX_1, TY_1), R(TX_2, TY_2), \\
& \quad I_0 = e, \mathit{Compose}(I_0, TY_1, I_1), \\
& \quad \mathit{Process}(HX, HHY), \mathit{Compose}(I_1, HHY, I_2), \\
& \quad \mathit{Compose}(I_2, TY_2, I_3), HY = I_3, \\
& \quad \mathit{R\_tupling}(TXs, TY), \mathit{Compose}(HY, TY, Y)
\end{array}$$

By unfolding clause 5 wrt $R(TX_1, TY_1)$ using $DCLR$, and simplifying using condition (4):

$$\begin{array}{ll}
\mathit{clause}\ 7: & \mathit{R\_tupling}(Xs, Y) \leftarrow \\
& \quad Xs = [X|TXs], \\
& \quad \mathit{NonMinimal}(X), \mathit{Decompose}(X, HX, TX_1, TX_2), \\
& \quad \mathit{Minimal}(TX_1), \mathit{Minimal}(TX_1), \mathit{Solve}(TX_1, TY_1), \\
& \quad R(TX_2, TY_2), \\
& \quad I_0 = e, \mathit{Compose}(I_0, TY_1, I_1), \\
& \quad \mathit{Process}(HX, HHY), \mathit{Compose}(I_1, HHY, I_2), \\
& \quad \mathit{Compose}(I_2, TY_2, I_3), HY = I_3, \\
& \quad \mathit{R\_tupling}(TXs, TY), \mathit{Compose}(HY, TY, Y)
\end{array}$$

By deleting one of the $\mathit{Minimal}(TX_1)$ atoms in clause 7:

$$\begin{array}{ll}
\mathit{clause}\ 8: & \mathit{R\_tupling}(Xs, Y) \leftarrow \\
& \quad Xs = [X|TXs], \\
& \quad \mathit{NonMinimal}(X), \mathit{Decompose}(X, HX, TX_1, TX_2), \\
& \quad \mathit{Minimal}(TX_1), \mathit{Solve}(TX_1, TY_1), \\
& \quad R(TX_2, TY_2), \\
& \quad I_0 = e, \mathit{Compose}(I_0, TY_1, I_1), \\
& \quad \mathit{Process}(HX, HHY), \mathit{Compose}(I_1, HHY, I_2), \\
& \quad \mathit{Compose}(I_2, TY_2, I_3), HY = I_3, \\
& \quad \mathit{R\_tupling}(TXs, TY), \mathit{Compose}(HY, TY, Y)
\end{array}$$

By rewriting clause 8 using applicability condition (1):

$$\begin{aligned}
&\textit{clause } 9: \quad R\_tupling(Xs,Y) \leftarrow \\
&\qquad Xs = [X|TXs], \\
&\qquad NonMinimal(X), Decompose(X,HX,TX_1,TX_2), \\
&\qquad Minimal(TX_1), Solve(TX_1,TY_1), \\
&\qquad R(TX_2,TY_2), \\
&\qquad I_0 = e, Compose(I_0,TY_1,I_1), \\
&\qquad Process(HX,HHY), Compose(I_1,HHY,I_2), HY = I_2, \\
&\qquad R\_tupling(TXs,TTY), Compose(TY_2,TTY,TY), \\
&\qquad Compose(HY,TY,Y)
\end{aligned}$$

By folding clause 9 using clauses 1 and 2:

$$\begin{aligned}
&\textit{clause } 10: \quad R\_tupling(Xs,Y) \leftarrow \\
&\qquad Xs = [X|TXs], \\
&\qquad NonMinimal(X), Decompose(X,HX,TX_1,TX_2), \\
&\qquad Minimal(TX_1), Solve(TX_1,TY_1), \\
&\qquad R\_tupling([TX_2|TXs],TY), \\
&\qquad I_0 = e, Compose(I_0,TY_1,I_1), \\
&\qquad Process(HX,HHY), Compose(I_1,HHY,I_2), HY = I_2, \\
&\qquad Compose(HY,TY,Y)
\end{aligned}$$

By introducing atom $Minimal(U)$ (with new, i.e. existentially quantified, variable U) in clause 6:

$$\begin{aligned}
&\textit{clause } 11: \quad R\_tupling(Xs,Y) \leftarrow \\
&\qquad Xs = [X|TXs], \\
&\qquad NonMinimal(X), Decompose(X,HX,TX_1,TX_2), \\
&\qquad NonMinimal(TX_1), Minimal(U), \\
&\qquad R(TX_1,TY_1), R(TX_2,TY_2), \\
&\qquad I_0 = e, Compose(I_0,TY_1,I_1), \\
&\qquad Process(HX,HHY), Compose(I_1,HHY,I_2), \\
&\qquad Compose(I_2,TY_2,I_3), HY = I_3, \\
&\qquad R\_tupling(TXs,TY), Compose(HY,TY,Y)
\end{aligned}$$

By using applicability condition (3):

$$\begin{aligned}
&\textit{clause } 12: \quad R\_tupling(Xs,Y) \leftarrow \\
&\qquad Xs = [X|TXs], \\
&\qquad NonMinimal(X), Decompose(X,HX,TX_1,TX_2), \\
&\qquad NonMinimal(TX_1), Minimal(U), R(U,e), \\
&\qquad R(TX_1,TY_1), R(TX_2,TY_2), \\
&\qquad I_0 = e, Compose(I_0,TY_1,I_1), \\
&\qquad Process(HX,HHY), Compose(I_1,HHY,I_2), \\
&\qquad Compose(I_2,TY_2,I_3), HY = I_3, \\
&\qquad R\_tupling(TXs,TY), Compose(HY,TY,Y)
\end{aligned}$$

By using applicability condition (2):

$$\begin{aligned}
&\textit{clause } 13: \quad R\_tupling(Xs,Y) \leftarrow \\
&\qquad Xs = [X|TXs], \\
&\qquad NonMinimal(X), Decompose(X,HX,TX_1,TX_2), \\
&\qquad NonMinimal(TX_1), Minimal(U), R(U,e), \\
&\qquad R(TX_1,TY_1), R(TX_2,TY_2), \\
&\qquad I_0 = e, Compose(I_0,TY_1,I_1), \\
&\qquad Compose(I_1,e,I_4), \\
&\qquad Process(HX,HHY), Compose(I_4,HHY,I_2), \\
&\qquad Compose(I_2,TY_2,I_3), HY = I_3, \\
&\qquad R\_tupling(TXs,TY), Compose(HY,TY,Y)
\end{aligned}$$

By using applicability conditions (1) and (2):

$$\begin{array}{ll}
\mathit{clause}\ 14: & \mathit{R\_tupling}(Xs, Y) \leftarrow \\
& \quad Xs = [X|TXs], \\
& \quad NonMinimal(X), \mathit{Decompose}(X, HX, TX_1, TX_2), \\
& \quad NonMinimal(TX_1), \\
& \quad Minimal(U), R(U, e), \\
& \quad R(TX_1, TY_1), R(TX_2, TY_2), \\
& \quad I_0 = e, Compose(I_0, e, I_1), \\
& \quad Process(HX, HHY), Compose(I_1, HHY, I_2), \\
& \quad Compose(I_2, TY_2, I_3), HY = I_3, \\
& \quad \mathit{R\_tupling}(TXs, TY), Compose(HY, TY, TI), \\
& \quad Compose(TY_1, TI, Y)
\end{array}$$

By introducing a new, i.e. existentially quantified, variable YU in place of some occurrences of e:

$$\begin{array}{ll}
\mathit{clause}\ 15: & \mathit{R\_tupling}(Xs, Y) \leftarrow \\
& \quad Xs = [X|TXs], \\
& \quad NonMinimal(X), \mathit{Decompose}(X, HX, TX_1, TX_2), \\
& \quad NonMinimal(TX_1), \\
& \quad Minimal(U), R(U, YU), \\
& \quad R(TX_1, TY_1), R(TX_2, TY_2), \\
& \quad I_0 = e, Compose(I_0, YU, I_1), \\
& \quad Process(HX, HHY), Compose(I_1, HHY, I_2), \\
& \quad Compose(I_2, TY_2, I_3), HY = I_3, \\
& \quad \mathit{R\_tupling}(TXs, TY), Compose(HY, TY, TI), \\
& \quad Compose(TY_1, TI, Y)
\end{array}$$

By introducing $NonMinimal(N)$ and $\mathit{Decompose}(N, HX, U, TX_2)$, since $\exists N : \mathcal{T}_X . NonMinimal(N) \wedge \mathit{Decompose}(N, HX, U, TX_2)$ always holds (because N is existentially quantified):

$$\begin{array}{ll}
\mathit{clause}\ 16: & \mathit{R\_tupling}(Xs, Y) \leftarrow \\
& \quad Xs = [X|TXs], \\
& \quad NonMinimal(X), \mathit{Decompose}(X, HX, TX_1, TX_2), \\
& \quad NonMinimal(TX_1), \\
& \quad Minimal(U), R(U, YU), \\
& \quad NonMinimal(N), \mathit{Decompose}(N, HX, U, TX_2), \\
& \quad R(TX_1, TY_1), R(TX_2, TY_2), \\
& \quad I_0 = e, Compose(I_0, YU, I_1), \\
& \quad Process(HX, HHY), Compose(I_1, HHY, I_2), \\
& \quad Compose(I_2, TY_2, I_3), HY = I_3, \\
& \quad \mathit{R\_tupling}(TXs, TY), Compose(HY, TY, TI), \\
& \quad Compose(TY_1, TI, Y)
\end{array}$$

By duplicating goal $\mathit{Decompose}(N, HX, U, TX_2)$:

$$\begin{array}{ll}
\mathit{clause}\ 17: & \mathit{R\_tupling}(Xs, Y) \leftarrow \\
& \quad Xs = [X|TXs], \\
& \quad NonMinimal(X), \mathit{Decompose}(X, HX, TX_1, TX_2), \\
& \quad NonMinimal(TX_1), \\
& \quad Minimal(U), R(U, YU), \\
& \quad NonMinimal(N), \mathit{Decompose}(N, HX, U, TX_2), \\
& \quad \mathit{Decompose}(N, HX, U, TX_2), \\
& \quad R(TX_1, TY_1), R(TX_2, TY_2), \\
& \quad I_0 = e, Compose(I_0, YU, I_1), \\
& \quad Process(HX, HHY), Compose(I_1, HHY, I_2), \\
& \quad Compose(I_2, TY_2, I_3), HY = I_3, \\
& \quad \mathit{R\_tupling}(TXs, TY), Compose(HY, TY, TI), \\
& \quad Compose(TY_1, TI, Y)
\end{array}$$

By folding clause 17 using $DCLR$:

$$\begin{aligned}
\textit{clause } 18: \quad & R\_tupling(Xs, Y) \leftarrow \\
& \quad Xs = [X|TXs], \\
& \quad NonMinimal(X), Decompose(X, HX, TX_1, TX_2), \\
& \quad NonMinimal(TX_1), \\
& \quad Minimal(U), \\
& \quad Decompose(N, HX, U, TX_2), \\
& \quad R(TX_1, TY_1), R(N, HY), \\
& \quad R\_tupling(TXs, TY), Compose(HY, TY, TI), \\
& \quad Compose(TY_1, TI, Y)
\end{aligned}$$

By folding clause 18 using clauses 1 and 2:

$$\begin{aligned}
\textit{clause } 19: \quad & R\_tupling(Xs, Y) \leftarrow \\
& \quad Xs = [X|TXs], \\
& \quad NonMinimal(X), Decompose(X, HX, TX_1, TX_2), \\
& \quad NonMinimal(TX_1), \\
& \quad Minimal(U), \\
& \quad Decompose(N, HX, U, TX_2), \\
& \quad R(TX_1, TY_1), \\
& \quad R\_tupling([N|TXs], TI), \\
& \quad Compose(TY_1, TI, Y)
\end{aligned}$$

By folding clause 19 using clauses 1 and 2:

$$\begin{aligned}
\textit{clause } 20: \quad & R\_tupling(Xs, Y) \leftarrow \\
& \quad Xs = [X|TXs], \\
& \quad NonMinimal(X), Decompose(X, HX, TX_1, TX_2), \\
& \quad NonMinimal(TX_1), \\
& \quad Minimal(U), \\
& \quad Decompose(N, HX, U, TX_2), \\
& \quad R\_tupling([TX_1, N|TXs], Y)
\end{aligned}$$

Clauses 1, 3, 10, and 20 are the clauses of $P_{R\_tupling}$. Therefore $P_{R\_tupling}$ is steadfast wrt $S_{R\_tupling}$ in $\mathcal{S}$.

To prove that P_R is steadfast wrt S_R in $\{S_{R\_tupling}\}$, we do a backward proof that we begin with P_R in TG and from which we try to obtain S_R.

The procedure P_R for R in TG is:

$$R(X, Y) \leftarrow \quad R\_tupling([X], Y)$$

By taking the 'completion':

$$\forall X : \mathcal{T}_X, \forall Y : \mathcal{T}_Y.\ \mathcal{I}_R(X) \Rightarrow [R(X, Y) \Leftrightarrow R\_tupling([X], Y)]$$

By unfolding the 'completion' above wrt $R\_tupling([X], Y)$ using $S_{R\_tupling}$:

$$\forall X : \mathcal{T}_X, \forall Y : \mathcal{T}_Y.\ \mathcal{I}_R(X) \Rightarrow [R(X, Y) \Leftrightarrow \exists Y_1, I_1 : \mathcal{T}_Y.\ \mathcal{O}_R(X, Y_1) \wedge I_1 = Y_1 \wedge Y = I_1]$$

By simplification:

$$\forall X : \mathcal{T}_X, \forall Y : \mathcal{T}_Y.\ \mathcal{I}_R(X)) \Rightarrow [R(X, Y) \Leftrightarrow \mathcal{O}_R(X, Y)]$$

We obtain S_R, so P_R is steadfast wrt S_R in $\{S_{R\_tupling}\}$.

Therefore, TG is also steadfast wrt S_R in $\mathcal{S}$.

C Proof of DG_2

As a reminder, we give again the generalization schema DG_2.

DG_2 : $\langle$ $DCLR$, $DGRL$, C_{d2} $\rangle$ where
 C_{d2} : (1) *Compose* is associative
  (2) *Compose* has e as the right identity element, where e appears in $DCLR$

and templates $DCLR$ and $DGRL$ are in Figure 9.

$$\begin{array}{l}
R(X,Y) \leftarrow \\
\quad Minimal(X), \\
\quad Solve(X,Y) \\
R(X,Y) \leftarrow \\
\quad NonMinimal(X), \\
\quad Decompose(X,HX,TX_1,TX_2), \\
\quad R(TX_1,TY_1), R(TX_2,TY_2), \\
\quad I_0 = e, Compose(I_0,TY_1,I_1), \\
\quad Process(HX,HY), Compose(I_1,HY,I_2), \\
\quad Compose(I_2,TY_2,I_3), Y = I_3
\end{array}$$

$$\begin{array}{l}
R(X,Y) \leftarrow \\
\quad R\_descending_2(X,Y,e) \\
R\_\ descending_2(X,Y,A) \leftarrow \\
\quad Minimal(X), \\
\quad Solve(X,S), Compose(S,A,Y) \\
R\_\ descending_2(X,Y,A) \leftarrow \\
\quad NonMinimal(X), \\
\quad Decompose(X,HX,TX_1,TX_2), \\
\quad A = A_3, R\_descending_2(TX_2,A_2,A_3), \\
\quad Process(HX,HY), Compose(HY,A_2,A_1), \\
\quad R\_descending_2(TX_1,A_0,A_1), \\
\quad Compose(e,A_0,Y)
\end{array}$$

Figure 9: Templates $DCLR$ and $DGRL$

The specification S_R of predicate R is:

$$\forall X : \mathcal{T}_X, \forall Y : \mathcal{T}_Y.\ \mathcal{I}_R(X) \Rightarrow [R(X,Y) \Leftrightarrow \mathcal{O}_R(X,Y)]$$

The specification $S_{R\_descending_2}$ of predicate $R\_descending_2$ is:

$$\forall X : \mathcal{T}_X, \forall Y, A : \mathcal{T}_Y.\ \mathcal{I}_R(X) \Rightarrow [R\_descending_2(X,Y,A)\ \Leftrightarrow\ \exists S : \mathcal{T}_Y.\ \mathcal{O}_R(X,S) \wedge \mathcal{O}_C(S,A,Y)]$$

where $\mathcal{O}_C$ is the output condition of *Compose* and $\mathcal{O}_R$ is the output condition of R.

To prove the generalization schema DG_2, we have to prove that templates $DCLR$ and $DGRL$ are *equivalent* wrt S_R. By Definition 5, this holds iff the following two conditions hold:

(a) $DCLR$ is steadfast wrt S_R in $\mathcal{S} = \{S_{Minimal}, S_{NonMinimal}, S_{Solve}, S_{Decompose}, S_{Compose}\}$, where $S_{Minimal}$, $S_{NonMinimal}$, S_{Solve}, $S_{Decompose}$, $S_{Compose}$ are the specifications of *Minimal*, *NonMinimal*, *Solve*, *Decompose*, *Compose*, which are all the undefined predicate names appearing in $DCLR$.

(b) $DGRL$ is also steadfast wrt S_R in $\mathcal{S}$.

In program transformation, we assume that the input program, here template $DCLR$, is steadfast wrt S_R in $\mathcal{S}$, so condition (a) always holds.

To prove equivalence, we have to prove condition (b). We will use the property of steadfastness: $DGRL$ is steadfast wrt S_R in $\mathcal{S}$ if $P_{R\_descending_2}$ is steadfast wrt $S_{R\_descending_2}$ in $\mathcal{S}$, where $P_{R\_descending_2}$ is the procedure for $R\_descending_2$, and P_R is steadfast wrt S_R in $\{S_{R\_descending_2}\}$, where P_R is the procedure for R.

To prove that $P_{R\_descending_2}$ is steadfast wrt $S_{R\_descending_2}$ in $\mathcal{S}$, we do a constructive forward proof that we begin with $S_{R\_descending_2}$ and from which we try to obtain $P_{R\_descending_2}$.

By taking the 'decompletion' of $S_{R\_descending_2}$:

$$clause\ 1 : \quad R\_descending_2(X,Y,A) \leftarrow R(X,S), Compose(S,A,Y)$$

By unfolding clause 1 wrt $R(X,S)$ using $DCLR$, and using the assumption that $DCLR$ is steadfast wrt S_R in $\mathcal{S}$:

$$
\begin{array}{ll}
\mathit{clause}\ 2: & R\_descending_2(X,Y,A) \leftarrow \\
& \quad Minimal(X), \\
& \quad Solve(X,S), Compose(S,A,Y) \\
\mathit{clause}\ 3: & R\_descending_2(X,Y,A) \leftarrow \\
& \quad NonMinimal(X), Decompose(X,HX,TX_1,TX_2), \\
& \quad R(TX_1,TS_1), R(TX_2,TS_2), \\
& \quad I_0 = e, Compose(I_0,TS_1,I_1), \\
& \quad Process(HX,HS), Compose(I_1,HS,I_2), \\
& \quad Compose(I_2,TS_2,I_3), S = I_3, \\
& \quad Compose(S,A,Y)
\end{array}
$$

By using applicability condition (1) on clause 3:

$$
\begin{array}{ll}
\mathit{clause}\ 4: & R\_descending_2(X,Y,A) \leftarrow \\
& \quad NonMinimal(X), Decompose(X,HX,TX_1,TX_2), \\
& \quad R(TX_1,TS_1), R(TX_2,TS_2), \\
& \quad Compose(TS_2,A,A_2), \\
& \quad Process(HX,HY), Compose(HY,A_2,A_1), \\
& \quad Compose(TS_1,A_1,A_0), \\
& \quad Compose(e,A_0,Y)
\end{array}
$$

By twice folding clause 4 using clause 1:

$$
\begin{array}{ll}
\mathit{clause}\ 5: & R\_descending_2(X,Y,A) \leftarrow \\
& \quad NonMinimal(X), Decompose(X,HX,TX_1,TX_2), \\
& \quad R\_descending_2(TX_2,A_2,A), \\
& \quad Process(HX,HY), Compose(HY,A_2,A_1), \\
& \quad R\_descending_2(TX_1,A_0,A_1), \\
& \quad Compose(e,A_0,Y)
\end{array}
$$

By introducing a new, i.e. existentially quantified, variable A_3:

$$
\begin{array}{ll}
\mathit{clause}\ 6: & R\_descending_2(X,Y,A) \leftarrow \\
& \quad NonMinimal(X), Decompose(X,HX,TX_1,TX_2), \\
& \quad A = A_3, R\_descending_2(TX_2,A_2,A_3), \\
& \quad Process(HX,HY), Compose(HY,A_2,A_1), \\
& \quad R\_descending_2(TX_1,A_0,A_1), \\
& \quad Compose(e,A_0,Y)
\end{array}
$$

Clauses 2 and 6 are the clauses of $P_{R\_descending_2}$. Therefore $P_{R\_descending_2}$ is steadfast wrt $S_{R\_descending_2}$ in $\mathcal{S}$.

To prove that P_R is steadfast wrt S_R in $\{S_{R\_descending_2}\}$, we do a backward proof that we begin with P_R in $DGRL$ and from which we try to obtain S_R.
The procedure P_R for R in $DGRL$ is:

$$R(X,Y) \leftarrow \quad R\_descending_2(X,Y,e)$$

By taking the 'completion':

$$\forall X : \mathcal{T}_X, \forall Y : \mathcal{T}_Y.\ \mathcal{I}_R(X) \Rightarrow [R(X,Y) \Leftrightarrow R\_descending_2(X,Y,e)]$$

By unfolding the 'completion' above wrt $R\_descending_2(X,Y,e)$ using $S_{R\_descending_2}$:

$$\forall X : \mathcal{T}_X, \forall Y : \mathcal{T}_Y.\ \mathcal{I}_R(X) \Rightarrow [R(X,Y) \ \Leftrightarrow\ \exists S : \mathcal{T}_Y.\ \mathcal{O}_R(X,S) \wedge \mathcal{O}_C(S,e,Y)]$$

By using applicability condition (2):

$$\forall X : \mathcal{T}_X, \forall Y : \mathcal{T}_Y.\ \mathcal{I}_R(X) \Rightarrow [R(X,Y) \ \Leftrightarrow\ \exists S : \mathcal{T}_Y.\ \mathcal{O}_R(X,S) \wedge S = Y]$$

By simplification:

$$\forall X : \mathcal{T}_X, \forall Y : \mathcal{T}_Y.\ \mathcal{I}_R(X) \Rightarrow [R(X,Y) \ \Leftrightarrow\ \mathcal{O}_R(X,Y)]$$

We obtain S_R, so P_R is steadfast wrt S_R in $\{S_{R\_descending_2}\}$.

Therefore, $DGRL$ is also steadfast wrt S_R in $\mathcal{S}$.