3. Schema-Guided Program Generation

3.1. The General Case

Reminder: Formal Conditional Specifications

Find a solution `Y` for input/problem `X` that satisfies some input condition $\mathtt{i}_r$,
such that some output condition $\mathtt{o}_r$ is satisfied.
Formally:

$$\forall\, \mathtt{X}{:}\mathcal{X}.\ \ \forall\, \mathtt{Y}{:}\mathcal{Y}.$$
$$\mathtt{i}_r\mathtt{(X)} \ \rightarrow\ (\ \mathtt{r(X,Y)} \ \leftrightarrow\ \mathtt{o}_r\mathtt{(X,Y)}\ ) \qquad (S_r)$$

The relation symbols $\mathtt{i}_r$ and $\mathtt{o}_r$ are *open* in S_r, whereas relation symbol `r` is *closed* in S_r.

Example: `sort(L,SL)` holds iff list `SL` is a non-decreasingly ordered permutation of list `L`.
Formally, in $\mathcal{LIST}$(`Elems`, ➤):

$$\forall\, \mathtt{L{:}list(Elems)}.\ \ \forall\, \mathtt{SL{:}list(Elems)}.$$
$$\mathtt{sort(L,SL)} \leftrightarrow \mathtt{perm(L,SL)} \wedge \mathtt{ord(SL)} \qquad (S_{sort})$$

A Divide-and-Conquer Program Schema, Called $\mathcal{DC}$

Template

$$
\begin{array}{l}
\texttt{r(X,Y)} \leftarrow \\
\quad \texttt{minimal(X)}, \\
\quad \texttt{solve(X,Y)} \\
\texttt{r(X,Y)} \leftarrow \\
\quad \neg\,\texttt{minimal(X)}, \\
\quad \texttt{decompose(X,V,}\texttt{X}_1\texttt{,}\texttt{X}_2\texttt{)}, \\
\quad \texttt{r(}\texttt{X}_1\texttt{,}\texttt{Y}_1\texttt{)},\ \texttt{r(}\texttt{X}_2\texttt{,}\texttt{Y}_2\texttt{)}, \\
\quad \texttt{compose(V,}\texttt{Y}_1\texttt{,}\texttt{Y}_2\texttt{,Y)}
\end{array}
\qquad (dc)
$$

Constraints $\texttt{i}_{\texttt{dec}}\texttt{(X)} \wedge \texttt{o}_{\texttt{dec}}\texttt{(X,V,}\texttt{X}_1\texttt{,}\texttt{X}_2\texttt{)} \rightarrow \texttt{i}_{\texttt{r}}\texttt{(}\texttt{X}_1\texttt{)} \wedge \texttt{X}_1 \prec \texttt{X} \wedge \texttt{i}_{\texttt{r}}\texttt{(}\texttt{X}_2\texttt{)} \wedge \texttt{X}_2 \prec \texttt{X} \qquad (C_1)$

Specifications

$$\texttt{i}_{\texttt{r}}\texttt{(X)} \rightarrow (\ \texttt{r(X,Y)} \leftrightarrow \texttt{o}_{\texttt{r}}\texttt{(X,Y)}\ ) \qquad (S_r)$$

$$\texttt{i}_{\texttt{r}}\texttt{(X)} \rightarrow (\ \texttt{minimal(X)} \leftrightarrow \neg\,\texttt{i}_{\texttt{dec}}\texttt{(X)}\ ) \qquad (S_{min})$$

$$\texttt{i}_{\texttt{r}}\texttt{(X)} \wedge \neg\,\texttt{i}_{\texttt{dec}}\texttt{(X)} \rightarrow (\ \texttt{solve(X,Y)} \leftrightarrow \texttt{o}_{\texttt{r}}\texttt{(X,Y)}\ ) \qquad (S_{solve})$$

$$\texttt{i}_{\texttt{dec}}\texttt{(X)} \rightarrow (\ \texttt{decompose(X,V,}\texttt{X}_1\texttt{,}\texttt{X}_2\texttt{)} \leftrightarrow \texttt{o}_{\texttt{dec}}\texttt{(X,V,}\texttt{X}_1\texttt{,}\texttt{X}_2\texttt{)}\ ) \qquad (S_{dec})$$

$$\texttt{o}_{\texttt{dec}}\texttt{(X,V,}\texttt{X}_1\texttt{,}\texttt{X}_2\texttt{)} \wedge \texttt{o}_{\texttt{r}}\texttt{(}\texttt{X}_1\texttt{,}\texttt{Y}_1\texttt{)} \wedge \texttt{o}_{\texttt{r}}\texttt{(}\texttt{X}_2\texttt{,}\texttt{Y}_2\texttt{)} \rightarrow (\ \texttt{compose(V,}\texttt{Y}_1\texttt{,}\texttt{Y}_2\texttt{,Y)} \leftrightarrow \texttt{o}_{\texttt{r}}\texttt{(X,Y)}\ ) \qquad (S_{comp})$$

Observations

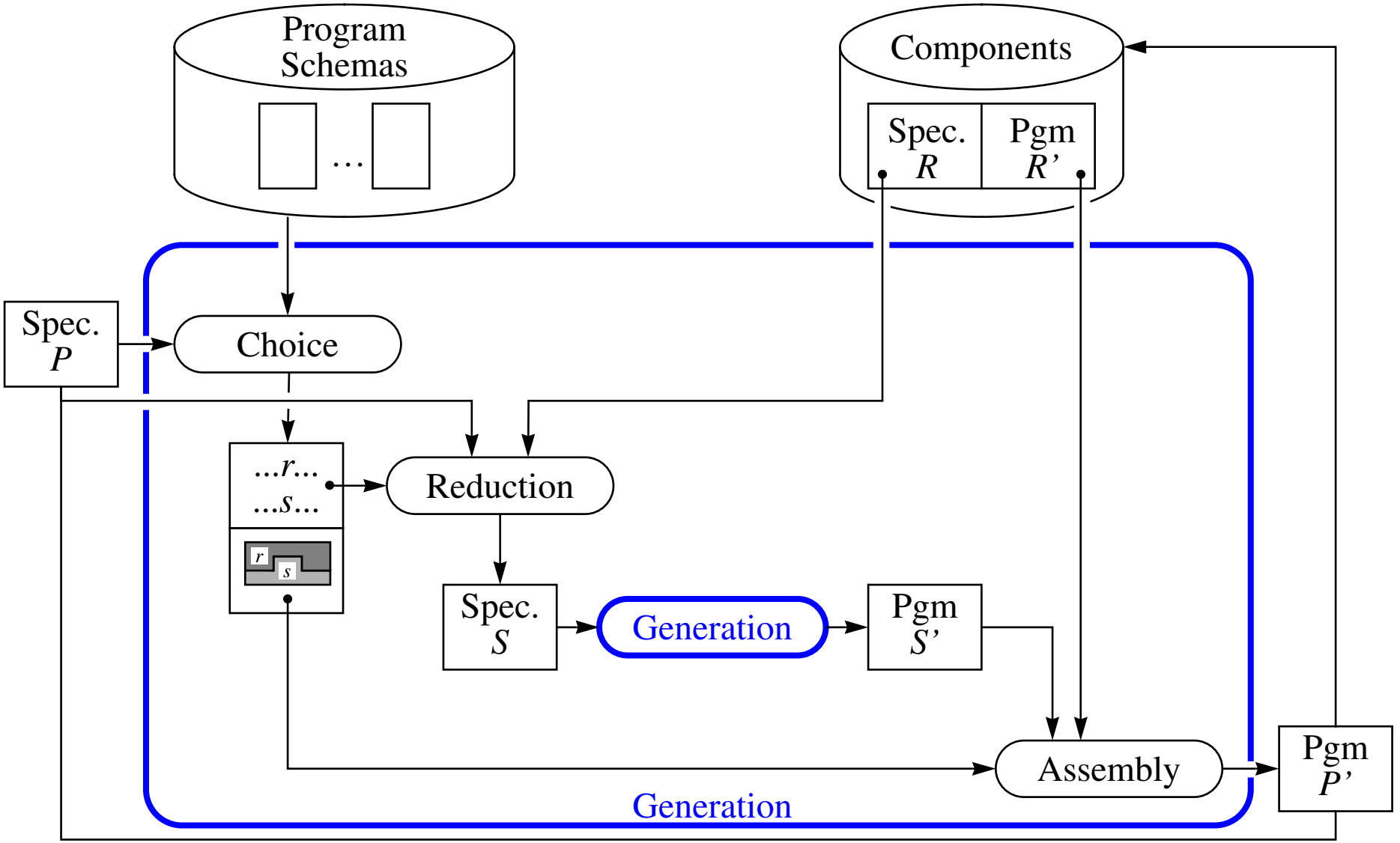
- *All* constraints and specifications of $\mathcal{DC}$ are expressed in terms of well-founded order $\prec$ and the input/output conditions $\mathtt{i}_{\mathtt{r}}$, $\mathtt{o}_{\mathtt{r}}$, $\mathtt{i}_{\mathtt{dec}}$, $\mathtt{o}_{\mathtt{dec}}$ of relations `r` and `decompose`.

- The input/output conditions $\mathtt{i}_{\mathtt{r}}$ and $\mathtt{o}_{\mathtt{r}}$ of relation `r` are of course *known*!

- Hence, if $\prec$ and `decompose`, that is S_{dec} (i.e., $\mathtt{i}_{\mathtt{dec}}$ and $\mathtt{o}_{\mathtt{dec}}$) and P_{dec}, are *re-used*, such that the constraints are satisfied,
  then the (open) specifications S_{min}, S_{solve}, S_{comp} become fully instantiated (or: closed),
  so that new program generation processes can be started from them,
  yielding programs P_{min}, P_{solve}, P_{comp},
  and that the generated program P_r is the union of dc, P_{min}, P_{solve}, P_{dec}, P_{comp}.

- In other words, specification S_r is then *reduced* into specifications S_{min}, S_{solve}, S_{comp}.
  This reflects the divide-and-conquer *generation strategy* that `decompose` is re-used first.
  An *alternative* strategy is to start by re-using `compose` instead,
  which leads to a different divide-and-conquer schema,
  with different constraints and different specifications, but with the same template.

Schema-Guided Program Generation through Specification Reduction



Definition of Schema-Guided Program Generation

Recursive *specification reduction* process,
followed by a recursive *program assembly* process:

- Base Case: Program reuse.
- Step Case: Specification reduction into new specifications.

Benefits of Schema-Guided Program Generation

- Program generation *can bottom out in re-use*, rather than only in the primitives of the language!
- Generation of steadfast (open) programs,
  in open frameworks,
  i.e., of *a-priori-correctly-reusable components*.
- *Reduced program search space*, because of off-line, paper-and-pencil pre-computations:

  + Pre-computed data-flow and control-flow (by the template).

  + Pre-computed interactions among the relations (by the constraints and specifications).

Issues of Schema-Guided Program Generation

- Generation is *backtrackable* at all choice-points: Many programs from a single specification.
- Choice *heuristics*: See Section 4.

Example: Generation of a List Sorting Program

In the open framework $\mathcal{LIST}$(`Elems`, ➤), consider the following (open) *formal specification*:

$$\forall \texttt{L,SL:list(Elems)}.$$

$$\texttt{sort(L,SL)} \leftrightarrow \texttt{perm(L,SL)} \wedge \texttt{ord(SL)} \qquad (S_{sort})$$

Program generation can proceed as follows:

- Assume re-use fails.
- Assume program schema $\mathcal{DC}$ is chosen.
- Assume $\angle$ is chosen as $<$, where $A \angle B$ means that list A has fewer elements than list B. Formally:

$$\forall \texttt{A,B:list(Elems)}.\ \texttt{A} \angle \texttt{B} \leftrightarrow \texttt{l(A)} < \texttt{l(B)}$$

- Assume `part` is chosen as `decompose`, because it satisfies the constraints, where `part(L,P,A,B)` holds iff `P` is the first element of list `L`, list `A` has the remaining elements of `L` that are smaller (under ➤) than `P`, and list `B` has the remaining elements of `L` that are not smaller (under ➤) than `P`. Formally:

$$\forall \texttt{L,A,B:list(Elems)}.\ \forall \texttt{P:Elems}.$$

$$\neg \texttt{L} = \texttt{nil} \rightarrow \qquad (S_{part})$$

$$(\ \texttt{part(L,P,A,B)} \leftrightarrow \texttt{L} = \texttt{P.T} \wedge \texttt{perm(A|B,T)} \wedge \texttt{A} \subset \texttt{P} \wedge \texttt{B} \supset \texttt{P}\ )$$

Consider that the following axioms:

$$\texttt{L} \subset \texttt{E} \leftrightarrow \forall \texttt{X:Elems.}\ \texttt{mem(X,L)} \rightarrow \texttt{E} \blacktriangleright \texttt{X}$$

$$\texttt{L} \supset \texttt{E} \leftrightarrow \forall \texttt{X:Elems.}\ \texttt{mem(X,L)} \rightarrow \neg\texttt{E} \blacktriangleright \texttt{X}$$

are part of $\mathcal{LIST}$(Elems, ➤).

- A known program P_{part} for part(L,P,A,B) is re-used for decompose:

```
decompose(X,V,X1,X2) ← part(X,V,X1,X2)
part(L,P,A,B) ← L=P.T, partition(T,P,A,B)                    (P_part)
partition(L,P,A,B) ← L=nil, A=nil, B=nil
partition(L,P,A,B) ← L=H.T, P ➤H, partition(T,P,TA,B), A=H.TA
partition(L,P,A,B) ← L=H.T, ¬P ➤H, partition(T,P,A,TB), B=H.TB
```

- Using the specifications of $\mathcal{DC}$, the following specifications can now be set up mechanically:

$$\forall\, \texttt{L:list(Elems).}\ \texttt{minimal(L)} \leftrightarrow \neg\neg\texttt{L=nil}$$

which simplifies into:

$$\forall\, \texttt{L:list(Elems).}\ \texttt{minimal(L)} \leftrightarrow \texttt{L=nil} \qquad (S_1)$$

and:

$$\forall \mathtt{L,S:list(Elems)}.$$

$$\mathtt{solve(L,S)} \leftrightarrow \mathtt{true} \wedge \neg\neg \mathtt{L = nil} \wedge \mathtt{perm(L,S)} \wedge \mathtt{ord(S)}$$

which simplifies into:

$$\forall \mathtt{L,S:list(Elems)}.\ \mathtt{solve(L,S)} \leftrightarrow \mathtt{L = nil} \wedge \mathtt{S = nil} \qquad (S_2)$$

and:

$$\forall \mathtt{E:Elems}.\ \forall \mathtt{C,D,S:list(Elems)}.$$

$$\mathtt{compose(E,C,D,S)} \leftrightarrow \exists \mathtt{L,T,A,B:list(Elems)}.$$

$$\mathtt{L = E.T} \wedge \mathtt{perm(A|B,T)} \wedge \mathtt{A} \subset \mathtt{E} \wedge \mathtt{B} \supset \mathtt{E} \wedge \mathtt{true} \wedge \mathtt{true} \wedge \mathtt{A} \angle \mathtt{L} \wedge \mathtt{B} \angle \mathtt{L}$$

$$\wedge\ \mathtt{perm(A,C)} \wedge \mathtt{ord(C)} \wedge \mathtt{perm(B,D)} \wedge \mathtt{ord(D)} \wedge \mathtt{perm(L,S)} \wedge \mathtt{ord(S)}$$

which simplifies into:

$$\forall \mathtt{E:Elems}.\ \forall \mathtt{C,D,S:list(Elems)}.\ \mathtt{compose(E,C,D,S)} \leftrightarrow \mathtt{S = C|(E.D)} \qquad (S_3)$$

- Recursive calls to program generation for specifications S_1, S_2, S_3 yield programs P_1, P_2, P_3:

```
minimal(X) ← X=nil                                          (P1)
solve(X,Y) ← X=nil, Y=nil                                   (P2)
compose(V,Y1,Y2,Y) ← append(Y1,V.Y2,Y)                      (P3)
```

- Assembling the programs P_{part}, P_1, P_2, P_3 with template *dc* gives the program:

```
sort(X,Y) ← minimal(X), solve(X,Y)
sort(X,Y) ← ¬minimal(X), decompose(X,V,X1,X2),
              sort(X1,Y1), sort(X2,Y2), compose(V,Y1,Y2,Y)
minimal(X) ← X=nil
solve(X,Y) ← X=nil, Y=nil
decompose(X,V,X1,X2) ← part(X,V,X1,X2)
compose(V,Y1,Y2,Y) ← append(Y1,V.Y2,Y)
part(L,P,A,B) ← L=P.T, partition(T,P,A,B)
partition(L,P,A,B) ← L=nil, A=nil, B=nil
partition(L,P,A,B) ← L=H.T, P➤H, partition(T,P,TA,B), A=H.TA
partition(L,P,A,B) ← L=H.T, ¬P➤H, partition(T,P,A,TB), B=H.TB
```

which can be partially evaluated into the final, still open, *steadfast program* :

```
sort(L,SL) ← L=nil, SL=nil
sort(L,SL) ← ¬L=nil, L=P.T, partition(T,P,A,B),
                sort(A,SA), sort(B,SB), append(SA,P.SB,SL)    (P_sort)
partition(L,P,A,B) ← L=nil, A=nil, B=nil
partition(L,P,A,B) ← L=H.T, P ➤H, partition(T,P,TA,B), A=H.TA
partition(L,P,A,B) ← L=H.T, ¬P ➤H, partition(T,P,A,TB), B=H.TB
```

This is a quick-sort program.

Alternative Program Generations (through backtracking)

- Assume the choice of head/tail decomposition as `decompose`:
  The result will be an insertion-sort program.

- Assume the choice of a divide-and-conquer schema where the initial re-use of `compose` is hardwired into the constraints and specifications:
  The choice of `append` as `compose` will then again lead to a quick-sort program.

3.2. A Particular Case, Studied via a Global Search Program Schema, Called $\mathcal{GS}$

Template

```
r(X,Y) ←                          r'(X,D,Y') ←
   initialise(X,D),                  extract(X,D,Y')
   r'(X,D,Y'),                    r'(X,D,Y') ←                    (gs)
   generate(Y',Y,X)                  split(D,X,D',δ),
                                      constrain(δ,D,X),
                                      r'(X,D',Y')
```

Informal Constraints and Specifications

`initialise(X,D)` iff `D` is the descriptor of the initial space of candidate solutions to problem `X`.

`extract(X,D,Y')` iff the variablised internal representation `Y'` of a solution to problem `X` is directly extracted from descriptor `D`.

`split(D,X,D',δ)` iff descriptor `D'` describes a subspace of descriptor `D` wrt problem `X`, such that `D'` is obtained by adding δ to `D`.

`constrain(δ,D,X)` iff adding δ to descriptor `D` leads to a descriptor defining a sub-space of `D` that may contain correct solutions to problem `X`.

`generate(Y',Y,X)` iff correct solution `Y` to problem `X` corresponds to the internal representation `Y'` of a solution that is generated from the constraint store (representing `X`).

Formal Constraints and Specifications

Let $\mathcal{X}$ and $\mathcal{Y}$ be the types of `X` and `Y`.

Let $\mathcal{Y}'$ be the type of internal representations of solutions.

Let $\mathcal{D}$ be the type of search space descriptors (such as `D`, `D'`).

Let Δ be the type of the elements (such as δ) of the partial solutions stored in descriptors.

Formal (open) *specifications* of the open relations of the *gs* template:

$$\texttt{initialise(X,D)} \leftrightarrow o_{\texttt{init}}\texttt{(X,D)} \qquad (S_{init})$$

$$\texttt{extract(X,D,Y')} \leftrightarrow o_{\texttt{extr}}\texttt{(X,D,Y')} \qquad (S_{extr})$$

$$\texttt{split(D,X,D',}\delta\texttt{)} \leftrightarrow o_{\texttt{split}}\texttt{(D,X,D',}\delta\texttt{)} \qquad (S_{split})$$

$$\texttt{constrain(}\delta\texttt{,D,X)} \leftrightarrow o_{\texttt{constr}}\texttt{(}\delta\texttt{,D,X)} \qquad (S_{constr})$$

$$\texttt{generate(Y',Y,X)} \leftrightarrow o_{\texttt{gen}}\texttt{(Y',Y,X)} \qquad (S_{gen})$$

where the 5 new symbols (the output conditions) must satisfy the 4 following constraints.

Formal *constraints*:

- Relations `generate` and `r` are equivalent when the constraint store is set up:

$$o_{\texttt{r}}\texttt{(X,Y)} \leftrightarrow \exists \texttt{Y'}:\mathcal{Y}'.\ o_{\texttt{gen}}\texttt{(Y',Y,X)} \qquad (C_0)$$

- All correct solutions `Y` to problem `X` are in the initial space for `X`:

$$\mathtt{o_r(X,Y)} \rightarrow \exists \mathtt{D}:\mathcal{D}.\ \exists \mathtt{Y'}:\mathcal{Y}'.$$
$$\mathtt{o_{init}(X,D)} \wedge \mathtt{o_{int2ext}(Y',Y)} \wedge \mathtt{satisfies(Y',D)} \qquad (C_1)$$

  where `satisfies(Y',D)` iff the internal representation `Y'` of a solution is in the space described by descriptor `D`, i.e., iff `Y'` can be extracted after a finite number of splits of `D`:

$$\mathtt{satisfies(Y',D)} \leftrightarrow$$
$$\exists \mathtt{k:int}.\ \exists \mathtt{D'}:\mathcal{D}.\ \exists \delta:\Delta.\ \mathtt{split^k(D,X,D',\delta)} \wedge \mathtt{o_{extr}(X,D',Y')}$$

  where:

$$\mathtt{split^0(D,X,D',\delta)} \leftrightarrow \mathtt{D = D'}$$

  and, for all `k:int`:

$$\mathtt{split^{k+1}(D,X,D',\delta)} \leftrightarrow$$
$$\exists \mathtt{D''}:\mathcal{D}.\ \exists \delta':\Delta.\ \mathtt{o_{split}(D,X,D'',\delta')} \wedge \mathtt{split^k(D'',X,D',\delta)} \qquad (C_2)$$

- Given a descriptor `D` and a solution `Y` to problem `X`, if splitting `D` into descriptor `D'` makes descriptor `D'` still contain the solution `Y`, then a constraint must be posted:

$$\mathtt{o_r(X,Y)} \wedge \mathtt{o_{int2ext}(Y',Y)} \wedge \mathtt{o_{split}(D,X,D',\delta)} \wedge \mathtt{satisfies(Y',D')}$$
$$\rightarrow \mathtt{o_{constr}(\delta,D,X)} \qquad (C_3)$$

  Contrapositive: If `constrain` fails, then the space described by `D'` contains no solution to `X`.

Observations

- The open relations of template *gs* have *"less universal" instances* than those of template *dc*.
  Hence: Difficulty of establishing useful bases of re-usable components for them.

- Schema $\mathcal{GS}$ has *"more complicated" constraints* than schema $\mathcal{DC}$.
  Hence: Difficulty of reducing specifications after the re-use step.

- There are *"not that many" families* of constraint satisfaction problems, e.g.:

  + *Assignment Problems*: Find a mapping between two given sets.

  + *Permutation Problems*: Find a sequence that is a permutation of the elements of a given set.

  + *Binary Search Problems*: Find an integer within a given interval.

  + *k-Subset Problems*: Find *k* subsets of a given set.

  + *Sequence Problems*: Find a sequence (of given or bounded size) over a given set.

  Hence: Possibility of pre-computing suitable instances for *all* open relations of template *gs*, such that the constraints are satisfied, for each such family!

Example: Formal Specifications of **Subset Decision Problems**

Find a subset `S` of a given finite set `T`, such that `S` satisfies a constraint `g` and any two different elements of `S` satisfy a constraint `p`.
Formally:

$$\forall\, \texttt{T:set(int)}.\ \forall\, \texttt{S:set(T)}.$$
$$\texttt{subset(T,S)} \leftrightarrow \texttt{g(S)} \wedge \forall\, \texttt{I,J:S}.\ \texttt{I} \neq \texttt{J} \rightarrow \texttt{p(I,J,T)} \qquad (S_{subset})$$

The specification S_{subset} is an *instance* of open specification (or: specification template) S_r, and has itself instances:

Example: Find a clique `C` of an undirected graph $\langle \texttt{V,A} \rangle$, so that the sum of its labels exceeds 22.
Formally, in $\mathcal{SET}(\texttt{int})$:

$$\forall\, \langle \texttt{V,A} \rangle\texttt{:set(int)} \times \texttt{set(V} \times \texttt{V)}.\ \forall\, \texttt{C:set(V)}.$$
$$\texttt{clique22}(\langle \texttt{V,A} \rangle, \texttt{C}) \leftrightarrow \qquad (S_{clique22})$$
$$\underline{\texttt{sum}}\texttt{(C,S)} \wedge \texttt{S} > 22 \wedge \forall\, \texttt{M,N:C}.\ \texttt{M} \neq \texttt{N} \rightarrow \langle \texttt{M,N} \rangle \in \texttt{A}$$

Example: Subset Decision Programs

```
subset(X,Y) ←
   initialise(X,D),
   subset'(X,D,Y'),
   generate(Y',Y,X)

initialise(X,D) ←
   D=⟨X,[],[]⟩
extract(_,D,Y') ←
   D=⟨{},V,W⟩,
   Y'=⟨V,W⟩, g(Y')
split(D,X,D',δ) ←
   D=⟨{A}∪T,V,W⟩,
   B in {true,false},
   δ=⟨A,B⟩,
   D'=⟨T,[A|V],[B|W]⟩
```

```
subset'(X,D,Y') ←
   extract(X,D,Y')
subset'(X,D,Y') ←
   split(D,X,D',δ),
   constrain(δ,D,X),
   subset'(X,D',Y')
constrain(_,D,_) ←
   D=⟨_,[],[]⟩
constrain(δ,D,X) ←                    (gs_subset)
   δ=⟨A,B⟩,
   D=⟨_,[E|V],[F|W]⟩,
   B∧F → p(A,E,X),
   constrain(δ,⟨_,V,W⟩,X)
generate(Y',Y,_) ←
   Y'=⟨_,W⟩, labeling([],W)
   int2ext(Y',Y)
```

… + set library with programs for ∪, ∩, ∈, sum, int2ext, …

Observations

- We can construct a program schema $\mathcal{GS}_{subset}$ with template gs_{subset} and *no* constraints.
- The specification S_{subset} and the program gs_{subset} are open in the *same* symbols!
- The set s of open symbols for which specifications S are to be set up through specification reduction can thus be *empty* , and *no* programs S' need to be generated:

$$s = S = S' = \varnothing$$

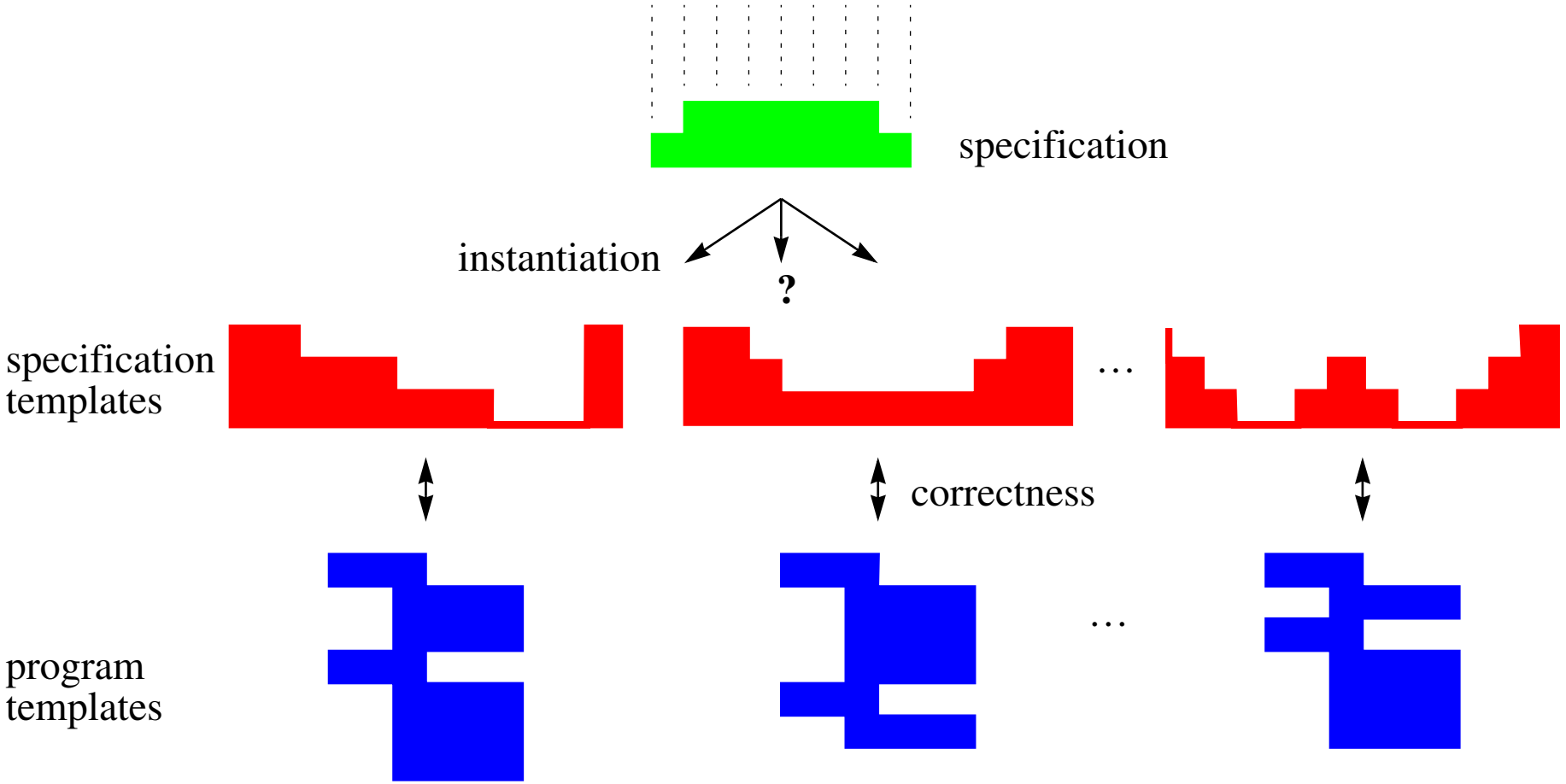
- The set r of open symbols for which programs R' are to be re-used can thus be the set of *all* open symbols in the open specification S_{subset}.
  Let θ be the formulas under which a specification is an instance of the open specification S_{subset}.
  We can thus re-use θ for R':

$$R' = \theta$$

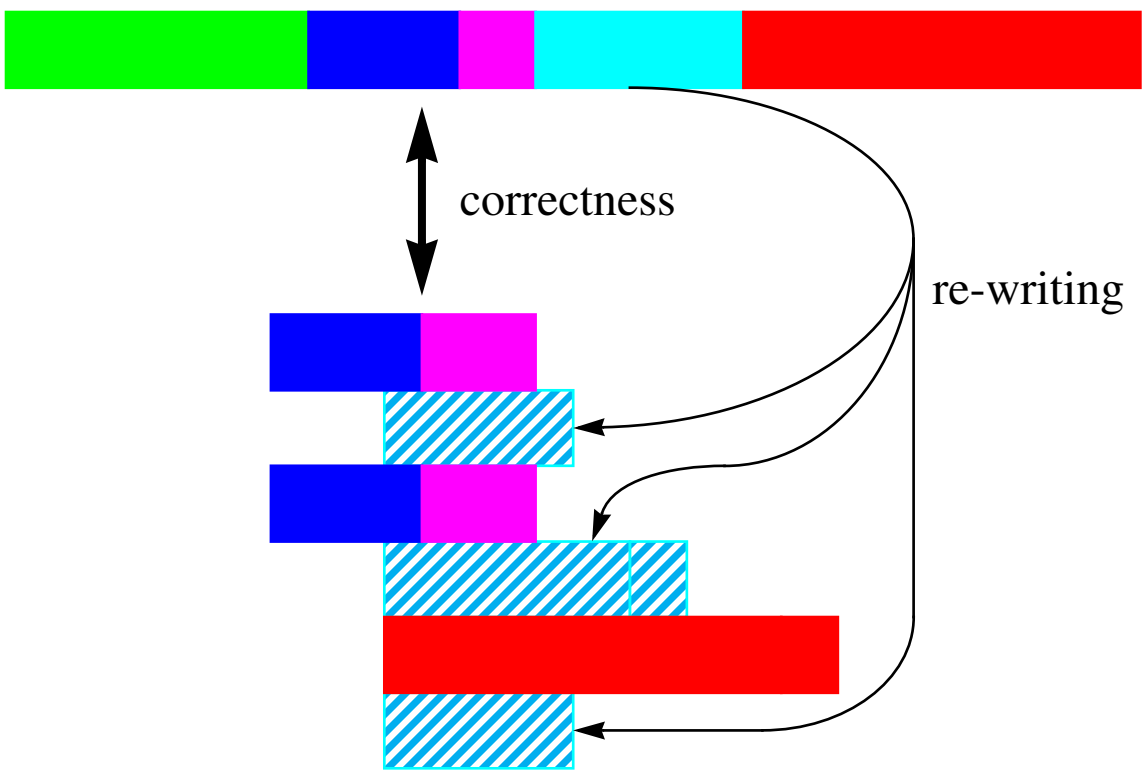
Example: $S_{clique22} = S_{subset}\,\theta_{clique22}$, with:

$$\begin{array}{l} \texttt{subset(X,Y)} \leftrightarrow \texttt{clique22(X,Y)} \\ \texttt{g(X)} \leftrightarrow \exists \texttt{Y:int.}\ \underline{\texttt{sum}}\texttt{(X,Y)} \wedge \texttt{Y} > 22 \\ \texttt{p(X,Y,Z)} \leftrightarrow \texttt{Z} = \langle\, \texttt{_,S}\, \rangle \wedge \langle\, \texttt{X,Y}\, \rangle \in \texttt{S} \end{array} \qquad (\theta_{clique22})$$

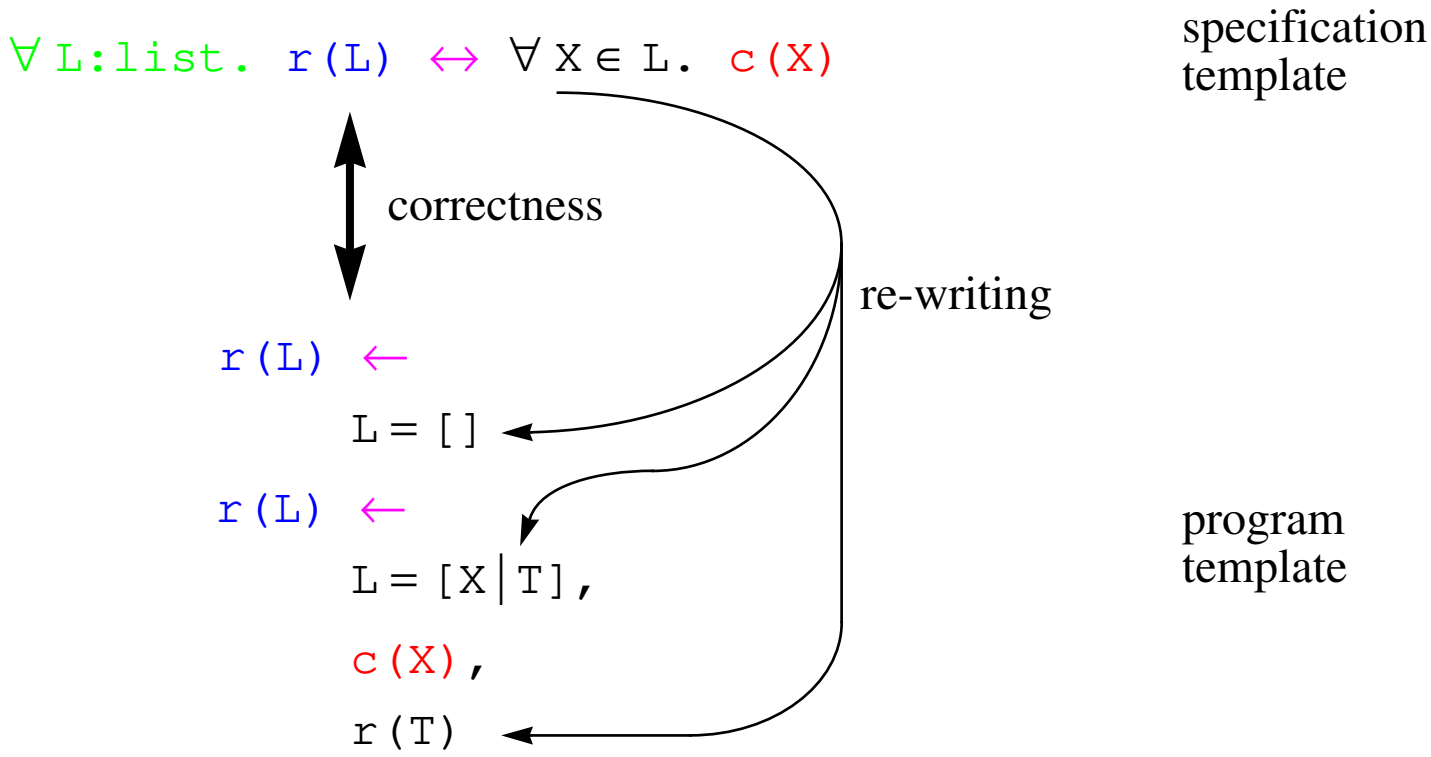
Schema-Guided Program Generation through Pre-Computation



Example:


specification
template
correctness
re-writing
program
template

Example:


∀ L:list. r(L) ↔ ∀ X ∈ L. c(X)
specification template
correctness
re-writing
r(L) ←
L = []
r(L) ←
L = [X | T],
c(X),
r(T)
program template

Example: Subset Decision Problem Specifications and Programs

$$\forall\, T{:}set(int).\ \forall\, S{:}set(T).$$
$$subset(T,S) \leftrightarrow g(S) \wedge \forall\, I,J{:}S.\ I \neq J \rightarrow p(I,J,T) \qquad (S_{subset})$$

```
subset(X,Y) ←
   initialise(X,D),
   subset'(X,D,Y'),
   generate(Y',Y,X)

initialise(X,D) ←
   D=⟨X,[],[]⟩
extract(_,D,Y') ←
   D=⟨{},V,W⟩,
   Y'=⟨V,W⟩, g(Y')
split(D,X,D',δ) ←
   D=⟨{A}∪T,V,W⟩,
   B in {true,false},
   δ=⟨A,B⟩,
   D'=⟨T,[A|V],[B|W]⟩
```

```
subset'(X,D,Y') ←
   extract(X,D,Y')
subset'(X,D,Y') ←
   split(D,X,D',δ),
   constrain(δ,D,X),
   subset'(X,D',Y')
constrain(_,D,_) ←
   D=⟨_,[],[]⟩                    (gs_subset)
constrain(δ,D,X) ←
   δ=⟨A,B⟩,
   D=⟨_,[E|V],[F|W]⟩,
   B∧F → p(A,E,X),
   constrain(δ,⟨_,V,W⟩,X)
generate(Y',Y,_) ←
   Y'=⟨_,W⟩, labeling([],W)
   int2ext(Y',Y)
```

Example: Generation of a Program for the `clique22` Problem

Knowing that:

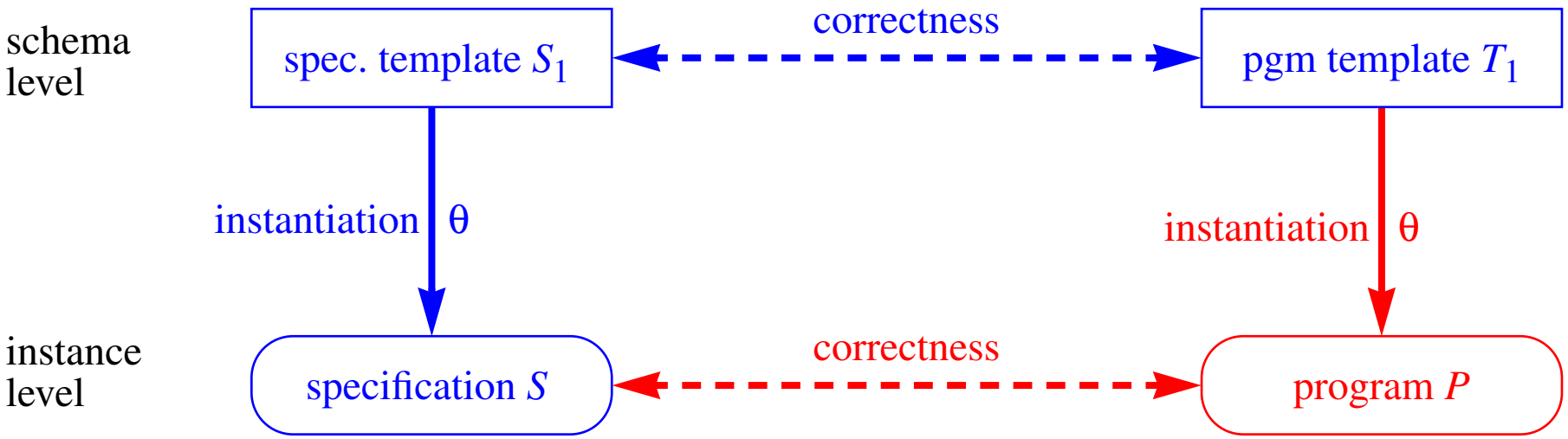
$$S_{clique22} = S_{subset}\ \theta_{clique22}$$

with:

$$\texttt{subset(X,Y)} \leftrightarrow \texttt{clique22(X,Y)}$$
$$\texttt{g(X)} \leftrightarrow \exists \texttt{Y:int.}\ \underline{\texttt{sum}}\texttt{(X,Y)} \wedge \texttt{Y} > 22 \qquad (\theta_{clique22})$$
$$\texttt{p(X,Y,Z)} \leftrightarrow \texttt{Z} = \langle \texttt{_,S} \rangle \wedge \langle \texttt{X,Y} \rangle \in \texttt{S}$$

program generation can establish:

$$P_{clique22} = gs_{subset}\ \theta_{clique22}$$

after rewriting $\theta_{clique22}$ as a logic program.

Alternative View of Schema-Guided Program Generation through Pre-Computation