1. Introduction

Why Program Generation? What Is Program Generation?

Example: Constraint Satisfaction Problems (CSP)

Optimisation problems: Find appropriate *values* for some *variables* within their *domains*, subject to some *constraints*, so that some *cost function* on these variables takes an optimal value.

Examples:

- Production Planning, subject to demand and resource availability, so that profit is maximised.
- Air Traffic Control, subject to safety protocols, so that flight times are minimised.
- Transportation Scheduling, subject to initial and final location of the goods and the transportation resources, so that delivery time and fuel expenses are minimised.

Decision problems: Particular case where there is no cost function.

Such problems can be expressed as *constraint programs* (CP) and solved using *constraint solvers*.

Example: Constraint Program for the **Map Colouring** Problem

Specification: Find a colouring M of the regions R of a map (with adjacency set A) using C colours. Formally:

$$\forall \langle R, C, A\rangle : \mathtt{set}(\mathtt{term}) \times \mathtt{nat} \times \mathtt{set}(R \times R).\ \forall M : \mathtt{set}(R \times 1..C).$$
$$\mathtt{colour}(\langle R, C, A\rangle, M) \leftrightarrow \forall \langle R_1, C_1\rangle \in M.\ \forall \langle R_2, C_2\rangle \in M.\ \langle R_1, R_2\rangle \in A \rightarrow C_1 \neq C_2$$

Correct program in CLP(FD) (a language for constraints over finite domains, hosted by Prolog):

```
colour(⟨R,C,A⟩,M) ←
    D=⟨R,[]⟩,
    post(⟨C,A⟩,D,M),
    labeling([ffc,step,up,all],M)
post(_,D,M) ←
    D=⟨[],M⟩
post(⟨C,A⟩,D,M) ←
    D=⟨[R1|R],M'⟩,
    C1 in 1..C,
    constrain(⟨R1,C1⟩,M',A),
    D'=⟨R,[⟨R1,C1⟩|M']⟩,
    post(⟨C,A⟩,D',M)
```

```
constrain(_,M,_) ←
    M=[]
constrain(δ,M,A) ←
    M=[⟨R2,C2⟩|M'],
    δ=⟨R1,C1⟩,
    ⟨R1,R2⟩∈A → C1≠C2,
    constrain(δ,M',A)
```

Problems

– *Long programming time*, because effective constraint programming is very difficult.

– *Long solving time*, because many of these problems are in NP.

Remedies

+ Quest for *more expressive* and *more declarative* specification/programming *languages*, with traditional algebraic notations (such as Σ, π) and rich datatypes (such as sets, sequences, enumerations), enabling a more natural expression of the constraints.

+ Addition of a *search procedure* and/or *redundant constraints*, so as to accelerate the default solution enumeration (or: labelling).

Trade-Off: The specification/programming language must still be *"executable"* (though it need not be computationally complete), and its programs should *execute quickly* (and finitely).

Program Generation: The production of correct (and efficient) and reusable programs from higher-level specifications/programs.

Classical Approaches to (Deductive) Program Generation

By Transformations/Refinements/...

$$\text{Formal Specification} = D_1 \Rightarrow D_2 \Rightarrow \ldots \Rightarrow D_{n-1} \Rightarrow D_n = \text{Program}$$

By Constructive Proof

Formal Specification ➡ Constructive Proof ➡ Program

(Common) Problems

– When to stop?

– Enormous search space.

– Bizarre programs.

– Does it really scale up?

Remedy: Inject more human knowledge about programming, via the notion of *program schema* !

Example: Transformational Generation of a Program for **List Membership**

Specification: `member(L,E)` holds iff term `E` is a member of list `L`.
Formally:

$$\forall \texttt{L:list.}\ \forall \texttt{E:term.}$$
$$\texttt{member(L,E)} \leftrightarrow \exists \texttt{P,S:list.}\ \underline{\texttt{append}}\texttt{(P,E.S,L)} \qquad (S_{member})$$

where `append(A,B,AB)` holds iff list `AB` is the concatenation of list `B` to the end of list `A`.
Formally, as *background knowledge*, and implemented as a logic program:

```
append(A,B,AB) ← A=nil, AB=B
append(A,B,AB) ← A=H.C, append(C,B,CB), AB=H.CB        (P_append)
```

Transformational Program Generation

- *Rewrite* the formal specification S_{member} as a logic program:

```
member(L,E) ← append(P,E.S,L)
```

- *Unfold* the `append` literal using P_{append}:

```
member(L,E) ← P=nil, L=E.S
member(L,E) ← P=H.C, append(C,E.S,D), L=H.D
```

- *Fold* the `append` literal using S_{member}:

  ```
  member(L,E) ← P=nil, L=E.S
  member(L,E) ← P=H.C, member(D,E), L=H.D
  ```

- *Rewrite* (eliminate useless literals, and reorder the literals), yielding a *correct program*:

  ```
  member(L,E) ← L=E.S                          (P_member)
  member(L,E) ← L=H.D, member(D,E)
  ```

Strategic Issues

- When and how to apply what transformation rule (*unfold*, *fold*, *definition*, *rewrite*) to what sub-expression?
- Usually, *unfold** + *rewrite** + *fold* + *rewrite** is sufficient, but when to stop unfolding?

Classical Approach to Program Schemas

A *program schema* should capture all the programs of a family of "similar" programs.

Example: the **Divide-and-Conquer** family

Program Template

```
r(X,Y) ←
   minimal(X),
   solve(X,Y)
r(X,Y) ←
   ¬minimal(X),
   decompose(X,V,X1,X2),
   r(X1,Y1), r(X2,Y2),
   compose(V,Y1,Y2,Y)
```

```
procedure r(X:𝒳,var Y:𝒴);
begin
 if minimal(X)
  then solve(X,Y)
 else
  begin
   decompose(X,V,X1,X2);
   r(X1,Y1); r(X2,Y2);
   compose(V,Y1,Y2,Y)
  end
end
```

(*dc*)

The relation symbols `minimal`, `solve`, `decompose`, `compose` are *open* in *dc*, whereas relation symbol `r` is *closed* in *dc*.

Example: Program for **Number-List Sorting**

Specification: `sort(L,SL)` holds iff list `SL` is a non-decreasingly ordered permutation of list `L`. Formally:

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

Correct program, as an *instance* of program template *dc* :

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

Equivalently, one could add the following clauses to the (open) program *dc* :

```
sort(X,Y) ← r(X,Y)
minimal(X) ← X=nil
solve(X,Y) ← X=nil, Y=nil                           (θ_sort)
decompose(X,V,X1,X2) ← X=V.T, partition(T,V,X1,X2)
compose(V,Y1,Y2,Y) ← append(Y1,V.Y2,Y)
```

Problems

– *Every* program is an instance of the *dc* program template!

For example, the following non-divide-and-conquer (open) program:

```
sort(L,SL) ← perm(L,SL), ord(SL)
```

is an instance of (open) program *dc* through addition of the following clauses:

```
minimal(X) ← true
solve(X,Y) ← perm(X,Y), ord(Y)
decompose(X,V,X1,X2) ← true
compose(V,Y1,Y2,Y) ← true
```

– Such program templates *cannot* actually *guide* program generation,
as they just prescribe some data-flow and control-flow for instances.

– Such program templates *only implicitly* assume a language and domain knowledge.

Remedy

Semantics has to be added to a program template in order to make a program schema,
to *prevent* some programs from being undesired instances,
to help *guide* the generation of *correct* programs, and
to *explicitly* define the underlying *language* and *domain knowledge*.