5. Future Directions

5.1. Generalisation: Schema-Guided Program Transformation

Example: Accumulator Introduction

```
r(X,Y) ←
    minimal(X),
    solve(X,Y)
```

```
r(X,Y) ←
    ¬minimal(X),
    decompose(X,H,T),
    r(T,V),
    process(H,F),
    compose(F,V,Y)
```

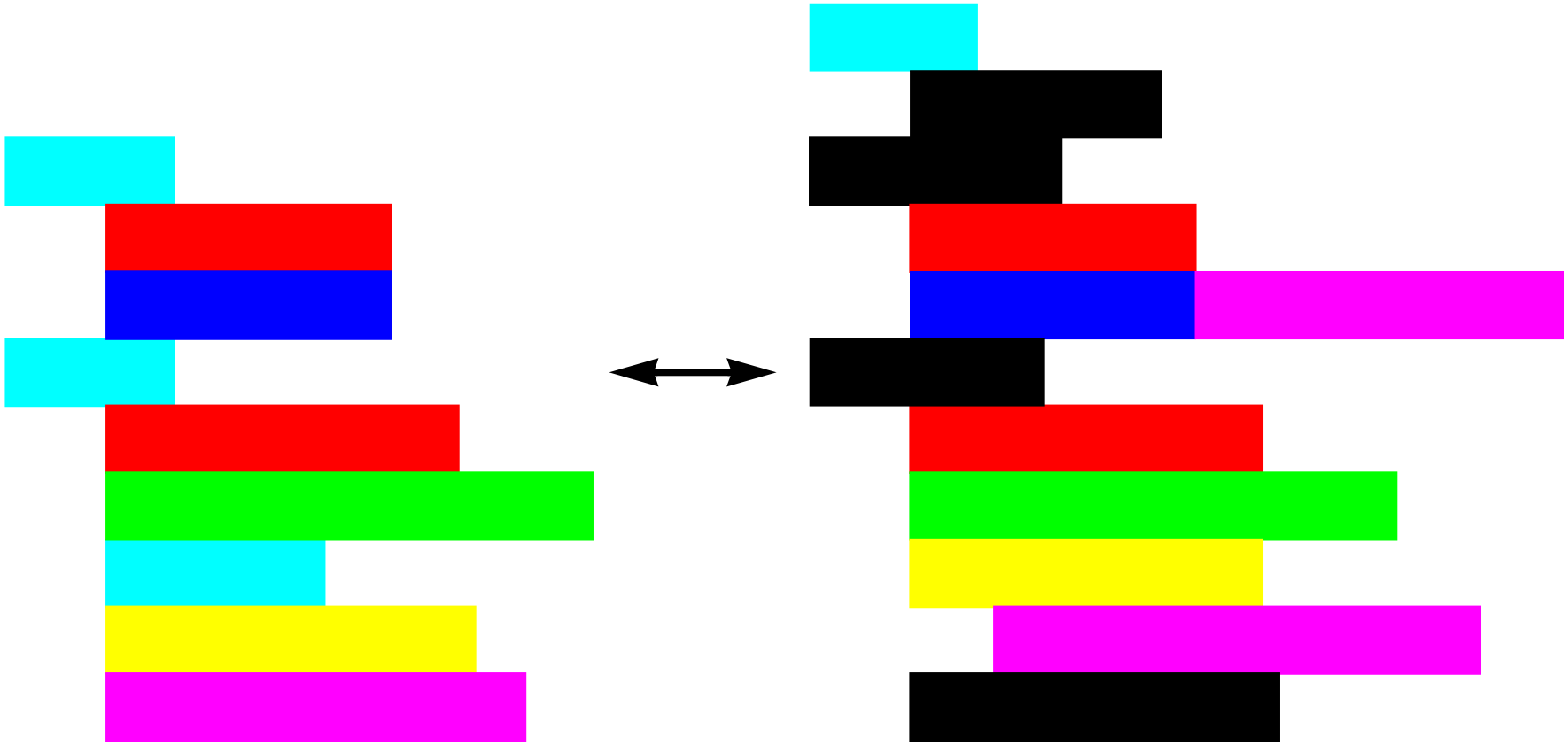
(dc_1)

```
r(X,Y) ←
    r'(X,Y,e)
r'(X,Y,A) ←
    minimal(X),
    solve(X,S), compose(A,S,Y)
```

```
r'(X,Y,A) ←
    ¬minimal(X),
    decompose(X,H,T),
    process(H,I),
        compose(A,I,NewA),
    r'(T,Y,NewA)
```

$(accu)$

Equivalence condition: The `compose` relation is associative and has `e` as identity element.



Nothing is created, nothing is lost…

Observations

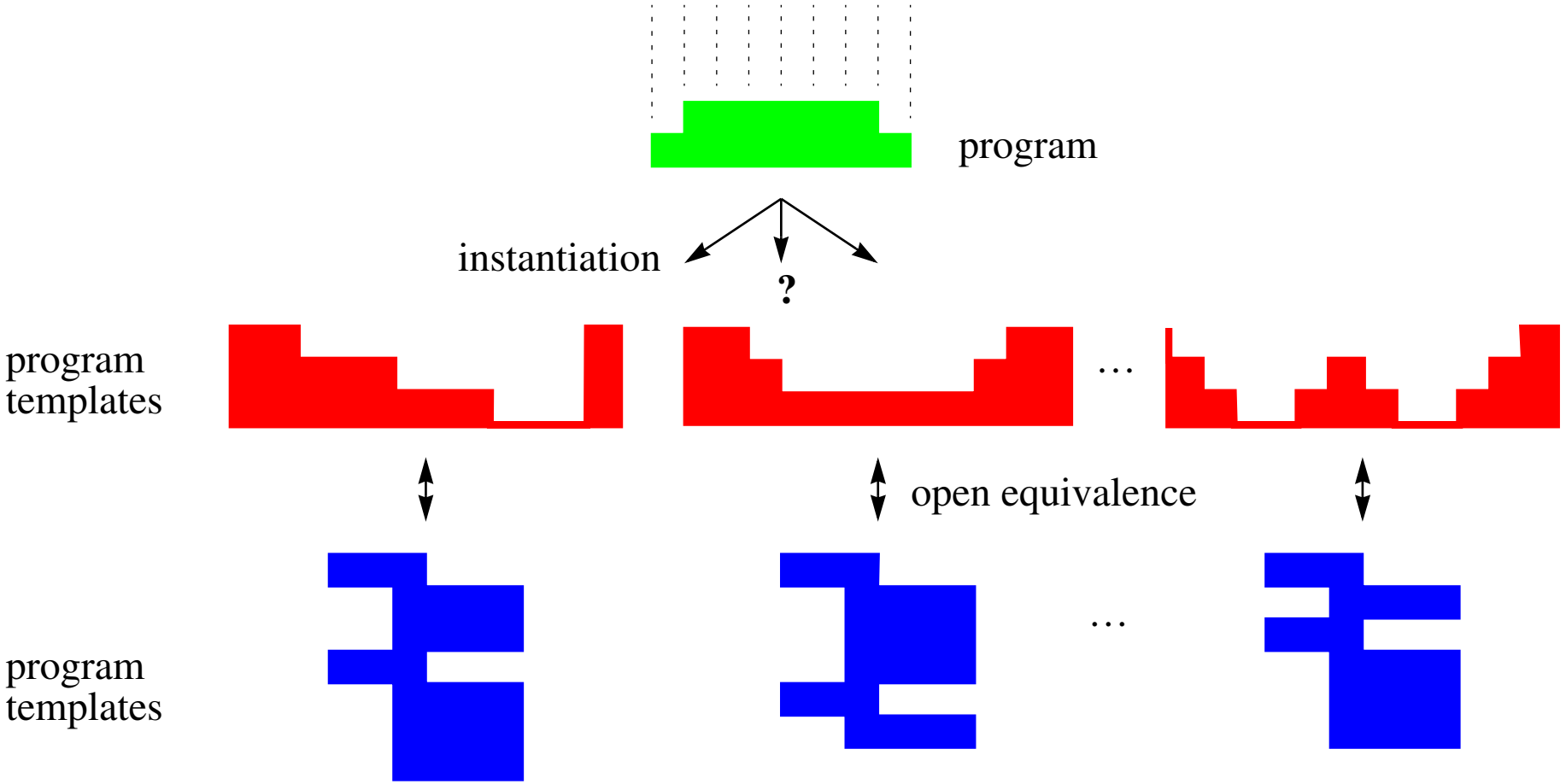
- Program template dc_1 plays the *same* role as specification template S_r in schema-guided program generation.
- The program templates dc_1 and *accu* are open in the *same* relation symbols.
- Hence: Possibility of achieving program transformation by following the *same* process as in schema-guided program generation through pre-computation (i.e., a particular case of schema-guided program generation through specification reduction)!
- The set s of open symbols for which specifications S are to be set up through 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 program template dc_1 .
  Let θ be the formulas under which a program is an instance of the program template dc_1 .
  We can thus re-use θ for R':

$$R' = \theta$$

Schema-Guided Program Transformation through Pre-Computation



Example: Transformation of the Naive Reverse Program

Specification: `reverse(L,R)` holds iff list `R` is the reverse of list `L`.

Correct program, as an instance of the program template dc_1 :

```
reverse(L,R) ←
   L=nil, R=nil
reverse(L,R) ←
   ¬L=nil,
   L=HL.TL,
   reverse(TL,TR),
   HR=HL.nil,
   append(TR,HR,R)
```

($P_{reverse}$)

Equivalently, one could add the following clauses to the template dc_1 :

```
reverse(X,Y) ← r(X,Y)
minimal(X) ← X=nil
solve(X,Y) ← X=nil, Y=nil
decompose(X,HX,TX) ← X=HX.TX
process(HX,HY) ← HY=HX.nil
compose(HY,TY,Y) ← append(TY,HY,Y)
```

($\theta_{reverse}$)

This program is *not* tail-recursive, and of $O(n^2)$ complexity, where n is the size of both lists.

Schema-transformed program, as an instance of the program template *accu* :

```
reverse(L,R) ←
   reverse'(L,R,nil)
reverse'(L,R,A) ←
   L=nil,
   S=nil, append(S,A,R)
reverse'(L,R,A) ←                                          (P'reverse)
   ¬L=nil,
   L=HL.TL,
   HA=HL.nil, append(HA,A,NewA),
   reverse'(TL,R,NewA)
```

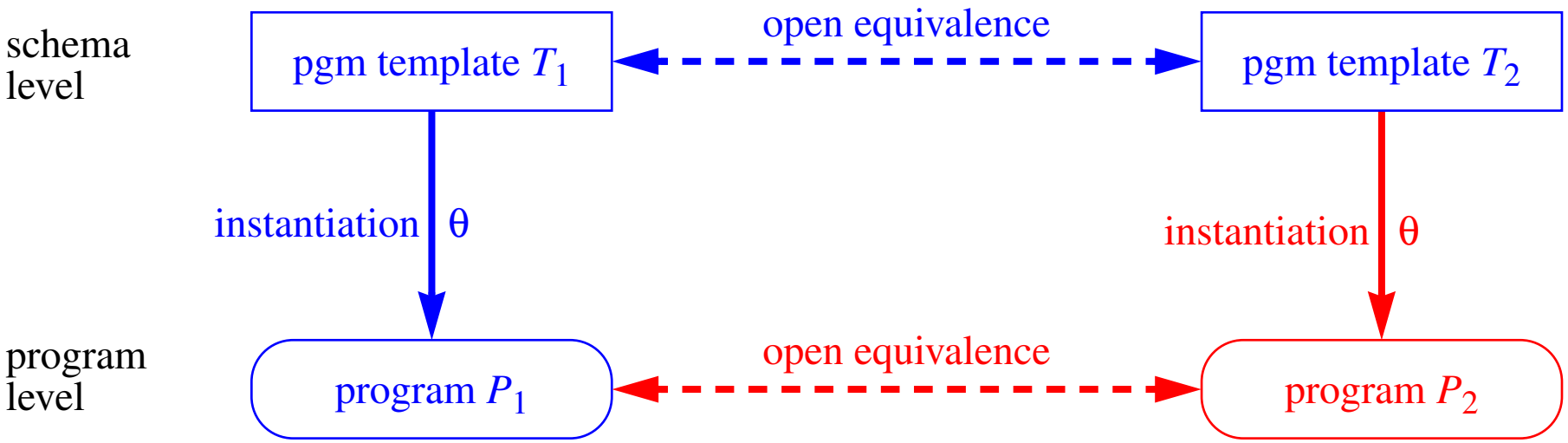
Equivalently, one could add the clauses $\theta_{reverse}$ to the template *accu* !

Partially evaluated version of $P'_{reverse}$, using program P_{append} (see page 1.5):

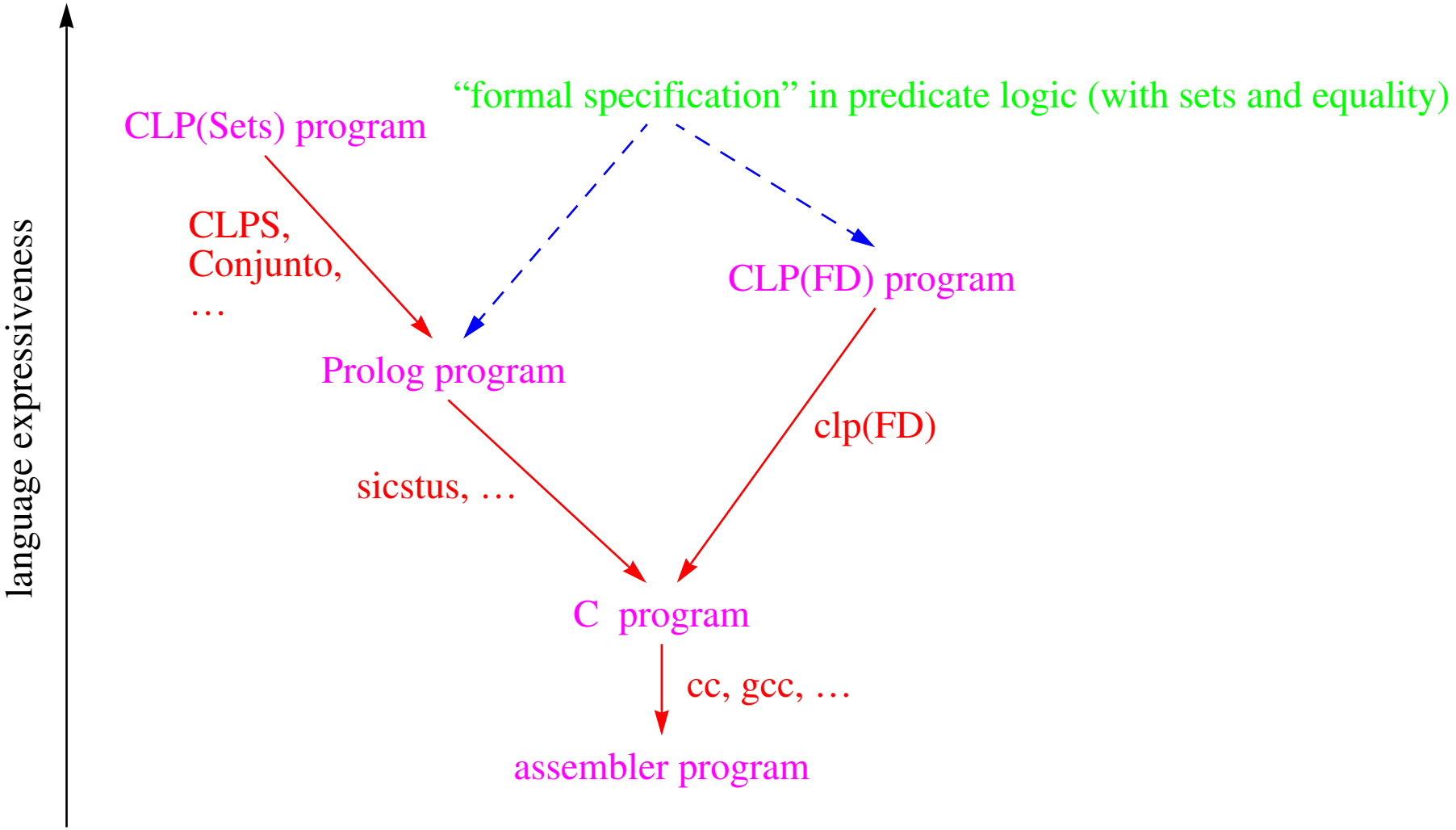
```
reverse(L,R) ← reverse'(L,R,nil)
reverse'(L,R,A) ← L=nil, R=A                               (P"reverse)
reverse'(L,R,A) ← L=HL.TL, NewA=HA.A, reverse'(TL,R,NewA)
```

This program *is* tail-recursive, and of O(n) complexity, where n is the size of both lists!

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



This diagram naturally chains up with the one for schema-guided program generation (page 3.23):
The (schema-guided) programming process is *iterative*!

By lifting the specification template S_r in a program schema framework with a template defining `r` to a privileged status, with a slot of its own thus, and
by seeing open "formal specifications" and open programs as just formulas,
whose instances are restricted by the other slots of the program schema,
one can *generalise* program schema frameworks to accommodate *both* program generation and program transformation.

Generation vs. Compilation, and "Formal Specifications" vs. Programs