# 4. Towards Automation

In this section, we consider automation issues. We argue that we need domain-specific heuristic knowledge in order to automate schema selection and application. We illustrate the approach with an example of schema-based generation, and identify some useful heuristics.

The schemas must be designed in order to make the automation practicable.

# Introduction

- We want to automate schema *choice* and *application*.
  - Global search?
  - Divide and conquer?
  - Which parameter is the input?
  - How many subproblems will there be?
  - How can we make sure the generated programs satisfy their specifications?
  - Reuse or generate?
- This means given a specification, finding a “good” applicable schema, and applying it.
- Why? – obvious!

- How?
  - Choice: sort considerations
    matching with specification template
    heuristics.
  - Application: management of subtasks
    automation of *appropriate* proof obligations.

- We do not attempt (at the moment) to automate:
  - derivation of schemas. Prefer to do this offline with pencil and paper.
  - the entire deductive synthesis process using only a theorem prover and our bare teeth.

- We must make compromises in order to make the process tractable.

# Framework operations

- Do we have the technology to manipulate schema frameworks?
  - Parameterisation: frameworks may take formal parameters (sorts, relations etc, for example $\mathcal{DC}(\mathsf{I}, \mathsf{O}, I_r, O_r, I_{dec}, O_{dec}, \prec)$).
    - Automation no problem.
  - DECLN: declare sorts of relation and function symbols in framework.
    - Automation may be nontrivial if the frameworks have complex sorts.
  - IMPORT: import other frameworks.
    - When a framework is imported, all its (public) symbols and axioms become available in the importing framework.
  - Solving subproblems.
    - Subproblems inherit the axioms, sorts, heuristics etc of their ancestors.

# Reducing the theorem proving burden

- Do difficult proofs offline.
  - This is an important but necessary break from the ideal of deductive synthesis and of proof automation.
- Prefer to reuse programs when possible.
- Prefer large number of specific schemas to small number of very general ones
  - we will have to organise them well.
- Use the strongest proof technology:
  - decision procedures
  - avoid massive search (e.g. resolution theorem proving) unless we have a well-tuned theory from which we expect a proof
  - employ heuristics to distinguish between when we can apply a particular schema and when it is a good idea.

# Heuristics

- There are many kinds of heuristics. For example:
  - Mode analysis, complexity analysis, statistical analysis of what has worked before, lookahead.
- We cannot do without heuristics if we want to have an automatic system.
- The heuristics need to examine criteria such as the availability of rewrite rules, the presence or absence of particular subterms, the complexity of functions.
- Heuristics are therefore necessarily at least second order.
- Clearly many heuristics are domain-specific, so we associate them with the (domain-specific) schemas.
- We may need heuristics to decide when we reach an executable specification, whether to try to reuse a previously stored program, continue generation from the specification, or translate the specification into a program in the target programming language.

# An example – using $\mathcal{DC}$

This is the hard subgoal

- Axiom part of single recursive call version of $\mathcal{DC}$:

$$\forall x : TX, y : TY . i_r(x) \rightarrow (r(x, y) \leftrightarrow o_r(x, y)) \qquad \text{(iffspec)}$$

$$\forall x : TX . i_r(x) \rightarrow (minimal(x) \leftrightarrow \neg i_{dec}(x)) \qquad \text{(minspec)}$$

$$\forall x : TX, y : TY . i_r(x) \wedge \neg i_{dec}(x) \rightarrow (solve(x, y) \leftrightarrow o_r(x, y)) \qquad \text{(solvespec)}$$

$$\forall x : TX, h : TH, t : TX . i_{dec}(x) \rightarrow (decompose(x, h, t) \leftrightarrow o_{dec}(x, h, t)) \qquad \text{(decspec)}$$

$$\forall x, t : TX, h : TH, v : TY . o_r(t, v) \rightarrow o_{dec}(x, h, t) \rightarrow (compose(h, v, y) \leftrightarrow o_r(x, y)) \qquad \text{(compspec)}$$

$$\forall x : TX . i_{dec}(x) \rightarrow \exists h : TH, t : TX . o_{dec}(x, h, t) \qquad \text{(decnontriv)}$$

$$\forall x, t : TX, h : TH . i_{dec}(x) \wedge o_{dec}(x, h, t) \rightarrow i_r(t) \wedge t \prec x \qquad \text{(decwf)}$$

$$well\_founded\_order(\prec) \qquad \text{(wfo)}$$

# Adding heuristics to the framework

- The schema framework packaged with its heuristics in a structure like a proof planning method:

Schema $\mathcal{DC}$

| | |
|---|---|
| spec pattern | **iffspec** |
| applicability heuristics | **library**($decompose(TX, TH, TX)$)<br>**annotate**(compspec, $Ann$)<br>**no_blocked_wave_fronts(**$Ann$**)** |
| output pattern | **minspec, solvespec, decwf, wfo, …** |
| schema framework | $\mathcal{DC}$ framework verbatim |

# Testing schema applicability

- When generating from a specification, we first (higher-order) match the specification with the specification template of the schema.
- We then try to satisfy the heuristics. This may reuse implementations (from a library) for some of the open relations.
- If the heuristics are satisfied, we set up the subgoals. We divide them into proof subgoals and generation subgoals (they are treated differently).
- We recurse on the subgoals. If they are all successfully dealt with, then we compose any programs from the subgoals with the program template from the applied schema to get the final program.
- If any stage of the process fails, we backtrack and try alternatives.

# Example – matching

- Specification:

$$\forall l : list(nat), n : nat.len(l) > 0 \rightarrow$$
$$min(l, n) \leftrightarrow n \in l \wedge (\forall y : nat. y \in l \rightarrow y \geq n)$$

- Try schema DC. Specification template:

$$\forall x : TX, y : TY.i_r(x) \rightarrow (r(x, y) \leftrightarrow o_r(x, y))$$

- The first task is matching. We use higher-order substitutions – λClam is an appropriate basis for implementation. Here:

$$[l / x,\ list(nat) / TX, n / y, \quad nat / TY, \lambda x.len(x) > 0 / i_r$$
$$\lambda x, y.min(x, y) / r, \lambda l, x. x \in l \wedge (\forall y : nat. y \in l \rightarrow y \geq x) / o_r]$$

# Example – choosing *decompose* and $\prec$

- So far, this schema would apply to anything.
- We now use some heuristics to choose an appropriate *decompose* and $\prec$.
- We use a heuristic from the *proof plan for induction* – we pick *decompose* and $\prec$ which will allow development of a *compose* operator to proceed. We can use the same heuristics to choose *compose*.
- The induction heuristic picks the candidate induction principle which is most likely to allow subsequent proof to proceed.
- Analogously, we look for the choice of *decompose* and $\prec$ which will allow generation to proceed.
- We use the same (relational) rippling (structure-preserving rewriting) heuristic.

# Example – choosing *decompose* and $\prec$

- The heuristics try each available *decompose* in turn, and annotate **compspec** accordingly. The heuristics succeed if the annotated goal contains no blocked wave fronts. The following choice has that property:

$$
\begin{aligned}
&decompose(l,h,t) \leftrightarrow l = h \bullet t \\
&i_{dec}(l) \leftrightarrow len(l) > 1 \\
&o_{dec}(l,h,t) \leftrightarrow l = h \bullet t \\
&x \prec y \leftrightarrow len(x) < len(y)
\end{aligned}
$$

- We assume these are all stored in a library.
- This choice allows the subsequent generation strategy to proceed at least one step.
- This choice reuses a program for *decompose*. The program here is merely a translation of its specification, but this need not be the case.

# Example – subgoals

- We can now set up the subgoals by applying the substitutions we have derived so far to the remaining axioms in the schema template. If subsequent generation fails, we can backtrack and pick a different *decompose* or $\prec$.
- The subgoals are:

$$\forall l : list(nat) . len(l) > 0 \rightarrow (minimal(l) \leftrightarrow \neg(len(l) > 1))$$
$$\forall l : list(nat), x : nat. len(l) > 0 \wedge \neg(len(l) > 1) \rightarrow (solve(l, x) \leftrightarrow o_r(l, x))$$
$$\forall l, t : list(nat), h, x : nat . o_r(t, v) \rightarrow l = h \bullet t \rightarrow (compose(h, v, x) \leftrightarrow o_r(l, x))$$

- The subgoals are fed back into the generator.

# Treating the subgoals

- Executable programs are derived from the first two subgoals using the following schema (a single argument version is needed for *minimal*):

Schema *simplifypre*

$$\forall l{:}TL, x{:}TX\ .\Phi(l) \rightarrow (r(l,x) \leftrightarrow p(l,x))$$

$$[\text{executable}(\forall l{:}TL.\Phi(l), \forall l{:}TL.\Psi(l)),$$

$$\text{executable}(\forall l : TL, x{:}TX\ .\Psi(l) \rightarrow p(l,x), \forall l : TL, x{:}TX\ .\Psi(l) \rightarrow q(l,x))\,]$$

$$r(l,x) \leftarrow \Psi(l), q(l,x)$$

- where **executable** succeeds iff its second argument is in the executable fragment of the logic, and is a simplification of the first argument. The domain theory for performing the simplication here comes from the $\mathcal{LIST}$ framework.
- In the case of **solvespec**, we get:

$$\Psi(l) \leftrightarrow \exists a : nat.\, l = a \bullet nil$$

$$q(l,x) \leftrightarrow x = a$$

# Generating from the *compose* subgoal

- The last subgoal is a specification of *compose*.
- Since it is of the form $o_r(x, y) \rightarrow \ldots \rightarrow o_r(x', y')$ we apply a schema *rippling* (adapted from the stepcase proof planning method), which uses rippling to remove $o_r$ and thereby simplify the specification of *compose*.
- The proof uses relational rippling, fertilisation, and a case split. It is quite hard but within the range of current proof planning technology.
- The result can be translated into a program for *compose*:

$$\forall h, v, y : nat.compose(h, v, y) \leftrightarrow ((h < v \wedge y = h) \vee (h \geq v \wedge y = v))$$

- The generation is now complete, and we form the union of the program templates with the other generated programs.

# Generated program

- This is the final program.

$$min(x,y) \leftarrow minimal(x), solve(x,y).$$

$$min(x,y) \leftarrow \neg minimal(x), decompose(x,h,t), min(t,v), compose(h,v,y).$$

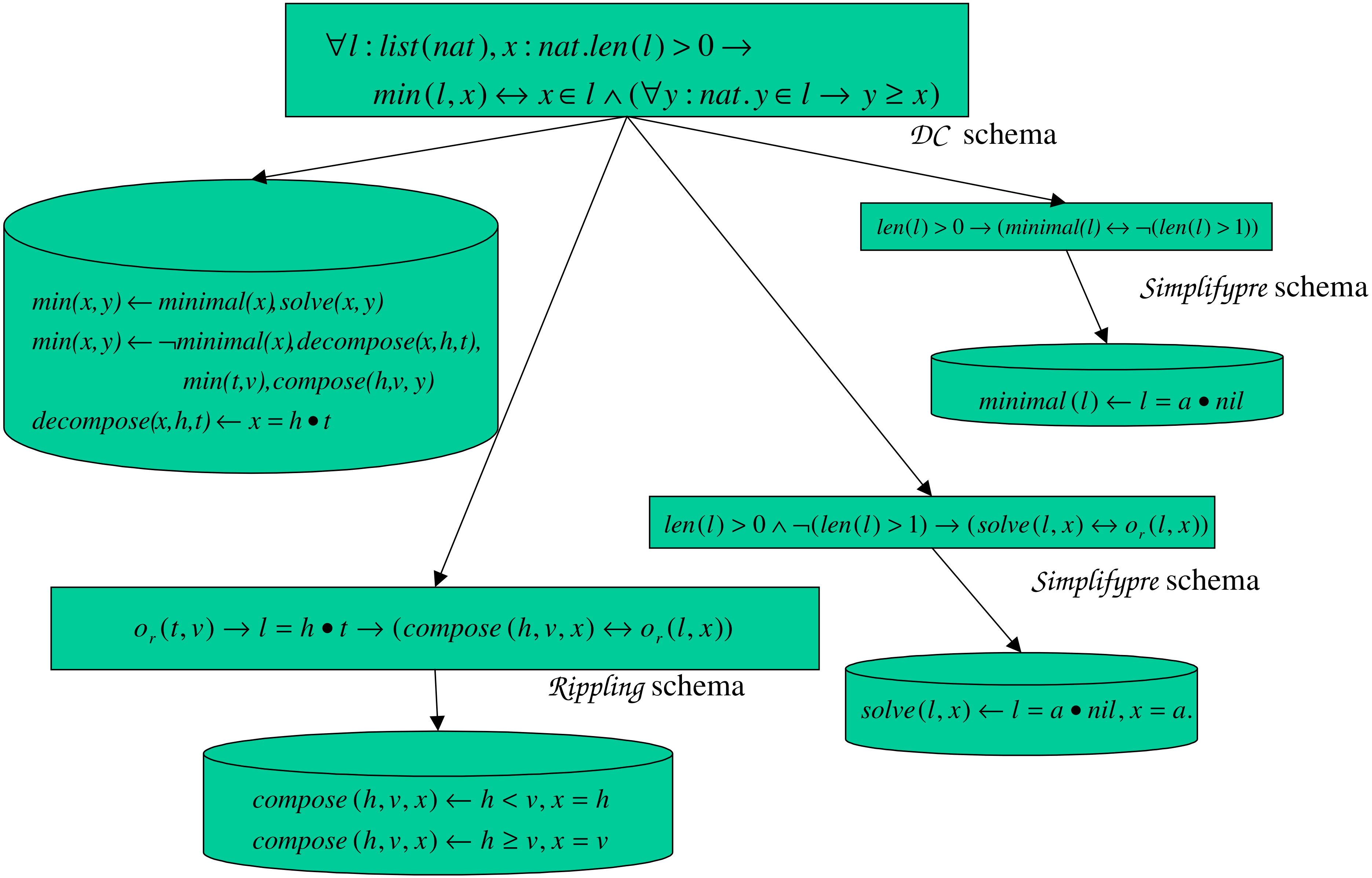
$$decompose(x,h,t) \leftarrow x = h \bullet t.$$

$$minimal(x) \leftarrow x = a \bullet nil.$$

$$solve(l,y) \leftarrow l = a \bullet nil, y = a.$$

$$compose(h,v,y) \leftarrow h < v, y = h.$$

$$compose(h,v,y) \leftarrow h \geq v, y = v.$$



$\forall l : list(nat), x : nat . len(l) > 0 \rightarrow min(l, x) \leftrightarrow x \in l \wedge (\forall y : nat . y \in l \rightarrow y \geq x)$
$\mathcal{DC}$ schema
$min(x,y) \leftarrow minimal(x), solve(x,y)$
$min(x,y) \leftarrow \neg minimal(x), decompose(x,h,t), min(t,v), compose(h,v,y)$
$decompose(x,h,t) \leftarrow x = h \bullet t$
$len(l) > 0 \rightarrow (minimal(l) \leftrightarrow \neg(len(l) > 1))$
$\mathcal{Simplifypre}$ schema
$minimal(l) \leftarrow l = a \bullet nil$
$len(l) > 0 \wedge \neg(len(l) > 1) \rightarrow (solve(l,x) \leftrightarrow o_r(l,x))$
$\mathcal{Simplifypre}$ schema
$o_r(t,v) \rightarrow l = h \bullet t \rightarrow (compose(h,v,x) \leftrightarrow o_r(l,x))$
$\mathcal{Rippling}$ schema
$solve(l,x) \leftarrow l = a \bullet nil, x = a.$
$compose(h,v,x) \leftarrow h < v, x = h$
$compose(h,v,x) \leftarrow h \geq v, x = v$

# What happened?

- We used:
  - higher-order matching between specifications and specification templates,
  - a prestored library of *decompose* and $\prec$ relations,
  - a heuristic adapted from the proof plan for induction to choose a *decompose* which allowed subsequent generation to proceed.
- The generation subgoals were satisfied by simplification and rippling schemas.
- The schemas performed some proof, using domain axioms from the $\mathcal{LIST}$ framework.
- Each of the schema applications yielded a program. We assembled the programs into a single correct program which implemented the initial specification.

# Organising domain knowledge

- Successful deductive synthesis systems, e.g. Amphion, rely on a heavily tuned domain theory to make the proof process work.
- Domain axioms are included in schemas. Tuning of the domain theory for us means careful choice of axioms in the schemas. This is a good place to do it, because schemas represent domain knowledge.
- Schemas can be seen as a way of organising theory (domain axioms and programs) and meta-theory (heuristics). If we do it right, then the (meta-)theory inherited by subproblems is just sufficient to allow them to be solved.
- This is important because previous work has demonstrated that having a good domain theory is crucial to the success of deductive synthesis.

# More about heuristics

- It may be necessary to sometimes employ specialised inference procedures, e.g. AC unification. We can allow schemas to do this too because this is also a kind of domain knowledge.
- Heuristics represent valuable and hard-to-come-by knowledge about how to program. It is important to represent them declaratively, and share them within the community.

# Summary

- We want to automate the selection and application of schemas.
- The schemas must be carefully designed, and sufficiently specific to yield proof/generation subgoals which are within the range of current proof/generation technology.
- In particular, we take some proof offline to pencil and paper.
- We attach heuristics to the schemas.
- The heuristics select (from a library) appropriate implementations for some of the specifications in the schema, and set up subgoals.
- Heuristics from proof by induction are useful for some subproblems.
- Schemas play an important role in organising the domain (meta)theory.