Tutorial:
Synthesis of Logic Programs

Kung-Kiu Lau
Dept of Computer Science
University of Manchester
kung-kiu@cs.man.ac.uk

Geraint A Wiggins
Dept of Artificial Intelligence
University of Edinburgh
geraint@aisb.ed.ac.uk

Overview

- Introduction
  - What is Logic Program Synthesis?
  - Classification and Terminology
- Constructive Synthesis
  - Definition
  - General Description
  - Example
- Deductive Synthesis
  - Definition
  - General Description
  - Example and Brief Survey
- Conclusion

What is Logic Program Synthesis?

Specification

$$\Big|$$

Synthesis

$$\downarrow$$

(Correct) Logic Program

What is a specification?

Spec = Prog ?

Spec $\neq$ Prog ?

Spec = Prog + Types ?

e.g. Naish 92

Spec = Theory ?

Spec = Theory + (Generic) Goal ?

e.g. Lau & Ornaghi 94a

What is Logic Program Synthesis?

What kind of specification?

- Informal: natural language,
  e.g. Deville 90, Fuchs 91
- Formal:
  * first-order theory
    e.g. Clark & Tarnlund 77
    Lau & Ornaghi 94b
  * first-order logic
    e.g. Clark & Sickel 77,
    Hansson & Tärnlund 79, Hogger 81
    Lau & Prestwich 90, Kraan 93
  * (extended) Horn clauses
    e.g. Sato & Tamaki 84, Kowalski 85,
    Kanamori & Horiuchi 87,
    Komorowski 89, Fribourg 90,
    Kawamura 92,
    Proietti & Pettorossi 94
  * constructive logic
    e.g. Wiggins 92a

What is Logic Program Synthesis?

What kind of logic program?

- pure logic program
  e.g. almost everyone
- Prolog program
  e.g. Deville 90, Fribourg 90
- Gödel program
  e.g. Wiggins 92a
- reusable/modular programs
  e.g. Lau, Ornaghi & Tärnlund 94

What criteria for correctness?

- informal: intended interpretation
- formal: partial or total correctness w.r.t. given semantics

Synthesis *vs* **Transformation**

LP Synthesis = LP Transformation ?

They are intimately related...

...but there is no generally accepted distinction.

A general distinction may be:

Synthesis:
the generation of a logic program from a specification in a language richer than Horn clause logic.

Transformation:
the generation of a logic program from another logic program.

Synthesis *vs* Transformation

Two similar views:

- Eriksson, Johansson & Tärnlund 86 :
  - *Synthesis*:
    derivation of a program from a theory T of data structures and computable relations or functions
  - *Transformation*:
    derivation of a program from a program and the theory T.

- Lau & Ornaghi 94a:
  - *Synthesis*:
    getting a program P from a theory S such that $\mathsf{Ax}(\mathsf{P})$, the *if*-part of $Comp(\mathsf{P})$, is a *sub-theory* of the theory S.
  - *Transformation*:
    getting a program P_2 ¿from a given program P_1, such that $\mathsf{Ax}(\mathsf{P}_1)$ is equivalent to $\mathsf{Ax}(\mathsf{P}_2)$

A Little History

Synthesis was one of the first topics of research in the early days of LP

Early work mainly not automated, although Uppsala developed a derivation editor

For some reason, synthesis work dwindled in 80's …

… until 1991, LOPSTR (workshop on LP synthesis and transformation) started

New interest mainly in automated or semi-automated synthesis …

(… but here we cover basic techniques, not automation or implemention)

A Taxonomy of (Formal) Methods

Taxonomy based on Deville & Lau 94:

- *Constructive Synthesis*

  (Also known as the *proofs-as-programs* approach.) A conjecture specifying an input-output relation is proved, and from this proof the specified program is extracted.

- *Deductive Synthesis*

  Clauses for the specified program are deduced directly from the specification.

- *Inductive Synthesis*

  A program is induced from its partial specification. The program is a generalisation of the partial specification.

  (Inductive Logic Programming is a whole research area in its own right. See tutorial at ILPS93 by Luc De Raedt, or Muggleton & De Raedt 94.)

Constructive Synthesis (1)

- Ideas borrowed from functional program synthesis
- "Proofs-as-programs"; Bates & Constable 82
- Central notion: demonstrate that

$$\begin{aligned}\vdash \forall i_0{:}\tau_0 \ldots \forall i_n{:}\tau_n.&\\ \exists r{:}\tau.&\\ \mathsf{Spec}(i_0, \ldots, i_n, r)&\end{aligned}$$

  i.e. for all well-typed input, there exists a result

- Then, we derive a function, $\mathcal{E} : \tau_0 \times \ldots \times \tau_n \mapsto \tau$, such that

$$\begin{aligned}\vdash \forall i_0{:}\tau_0 \ldots \forall i_n{:}\tau_n.&\\ \mathsf{Spec}(i_0, \ldots, i_n, \mathcal{E}(i_0, \ldots, i_n))&\end{aligned}$$

Constructive Synthesis (2)

- The function, $\mathcal{E}$, may, in some cases, be obtained directly from a proof
- The proof that the result r exists (partly) encodes a procedure for generating r
- The encoding is complete when the proof is not of "pure existence"
- *i.e.* when the proof (not necessarily the logic) is constructive
- Classical proofs generate programs too, but they can contain "oracle" steps, where an existential witness has been shown to exist, but not given (*cf.* $\lambda\mu$ calculus of Parigot 92)

Constructive Synthesis (3)

- Fribourg 90: use proof rules of Kanamori & Seki 86's *extended execution* to generate functions expressed as logic programs

- But how can we make this work for "real" logic programs (with failure, backtracking, *etc.*)?

- View predicates as boolean-valued functions, then show

$$\begin{array}{l} \vdash \forall i_0 : \tau_0 \ldots \forall i_n : \tau_n . \\ \quad \exists b : \mathsf{bool}. \\ \quad\quad \mathit{Spec}(i_0, \ldots, i_n, b) \end{array}$$

  or demonstrate that Spec is either true or false for all input:

$$\begin{array}{l} \vdash \forall i_0 : \tau_0 \ldots \forall i_n : \tau_n . \\ \quad \mathit{Spec}(i_0, \ldots, i_n, b) \vee \neg \mathit{Spec}(i_0, \ldots, i_n, b) \end{array}$$

- Assume that we generate a program to be called in the all-ground mode

Constructive Synthesis (4)

- Kraan 92, 93 automates the first style *at the meta-level of "proof planning"*

  She plans a proof of a conjecture:

  $$\vdash \forall i_0 : \tau_0 \ldots \forall i_n : \tau_n . \mathsf{Spec}(i_0, \ldots, i_n) \leftrightarrow \mathcal{P}$$

  where $\mathcal{P}$ is a formula-valued meta-variable

  The planning process instantiates $\mathcal{P}$, (which may then be verified in FOPC)

  Basin 94 reconstructs this idea at the object level of "schematic proofs"

- There is a strong similarity with ideas in program transformation, e.g.:
  Partial Evaluation (Lloyd & Shepherdson 84);
  Compiling Control (Bruynooghe et al 89)

- Result of synthesis process is a "distillation" of the execution of the specification

Constructive Synthesis (5)

- How does the method work?

  Consider the specification

  $$\vdash \forall x\!:\!\mathbb{N}.\forall y\!:\!\mathit{list}(\mathbb{N}).(x \in y) \vee \neg(x \in y)$$

  We need a (familiar) definition for $\in$:

  $$\begin{aligned} x \in y \;\leftrightarrow\; & y = [\,] \wedge \mathit{false} \vee \\ & \exists y_0\!:\!\mathbb{N}.\exists y_1\!:\!\mathit{list}(\mathbb{N}). \\ & \quad (y = [y_0|y_1] \wedge \\ & \qquad ((x = y_0 \wedge \mathit{true}) \vee x \in y_1)) \end{aligned}$$

  The proof consists in constraining values of x and y by case-analysis (including induction)...

  ...until we are left with a conjecture whose conclusion is unconditionally true or false

  The program is then read off from the proof

Constructive Synthesis (6)

- The structure of the extracted program corresponds very closely with that of the proof:
  - Case-splits correspond with (dis)unifications and/or constraint applications
  - Inductions correspond with definitions of base and step clauses of recursive programs
  - Appeal to induction hypothesis corresponds with a recursive call
- But unfolding and rewriting steps do not appear in the extracted program

Constructive Synthesis (7)

- Example: Synthesising `inc/2`

- Synthesis conjecture:

  $\vdash \forall \mathsf{k} : \mathsf{list}(\mathbb{N}).\forall \mathsf{y} : \mathsf{list}(\mathbb{N}).\partial(\forall z : \mathbb{N}. z \in \mathsf{k} \rightarrow z \in \mathsf{l})$

  where $\partial \mathsf{P}$ means $\mathsf{P} \vee \neg \mathsf{P}$

- Strategy: induction on k
  - trivial base case
  - case-split in step case
    - induction on l
    - appeal to induction hypothesis

- After application of induction on lists to k we have

  Base case $\vdash \forall \mathsf{l} : \mathbb{N}.\partial(\forall z : \mathbb{N}. z \in [\,] \rightarrow z \in \mathsf{l})$

  Step case $\forall \mathsf{l} : \mathbb{N}.\partial(\forall z : \mathbb{N}. z \in \mathsf{k}_1 \rightarrow z \in \mathsf{l})$
  $\vdash \forall \mathsf{l} : \mathbb{N}.\partial(\forall z : \mathbb{N}. z \in [\mathsf{k}_0 | \mathsf{k}_1] \rightarrow z \in \mathsf{l})$

Constructive Synthesis (8)

- Program, after induction on k:

```
p( [], ...)       :- ...
p( [K0|K1], ...) :- ...
```

NB: the case split, $\mathsf{K} \in \{\,[\,],\,[\mathsf{K}_0\,|\,\mathsf{K}_1]\,\}$, is all that is extracted here

- Base case: $\vdash \forall \mathfrak{l}\!:\!\mathbb{N}.\partial(\forall z\!:\!\mathbb{N}.z \in [\,] \rightarrow z \in \mathfrak{l})$

  The specification formula (inside ∂) is trivially true for all $\mathfrak{l}$...

  ...so (since $\partial(\mathcal{P})$ means $\mathcal{P} \vee \neg\mathcal{P}$), choose left disjunct.

  The rest of the base case proof is trivial.

- Program, after base case proof:

```
p( [], L )       :- true.
p( [K0|K1], L ) :- ...
```

Constructive Synthesis (9)

- Step case:

$$\begin{array}{l} \forall l{:}\mathbb{N}.\partial(\forall z{:}\mathbb{N}.z \in k_1 \rightarrow z \in l) \\ \vdash \forall l{:}\mathbb{N}.\partial(\forall z{:}\mathbb{N}.z \in [k_0 | k_1] \rightarrow z \in l) \end{array}$$

unfold left $\in$ atom to give

$$\begin{array}{l} \forall l{:}\mathbb{N}.\partial(\forall z{:}\mathbb{N}.z \in k_1 \rightarrow z \in l) \\ \vdash \forall l{:}\mathbb{N}.\partial(\forall z{:}\mathbb{N}.(z = k_0 \vee z \in k_1) \rightarrow z \in l) \end{array}$$

rewrite to give

$$\begin{array}{l} \forall l{:}\mathbb{N}.\partial(\forall z{:}\mathbb{N}.z \in k_1 \rightarrow z \in l) \\ \vdash \forall l{:}\mathbb{N}.\partial(\forall z{:}\mathbb{N}.(z = k_0 \rightarrow z \in l) \wedge \\ \qquad\qquad\qquad\qquad (z \in k_1 \rightarrow z \in l)) \end{array}$$

rewrite again to give

$$\begin{array}{l} \forall l{:}\mathbb{N}.\partial(\forall z{:}\mathbb{N}.z \in k_1 \rightarrow z \in l) \\ \vdash \forall l{:}\mathbb{N}.\partial(\forall z{:}\mathbb{N}.(z = k_0 \rightarrow z \in l) \wedge \\ \qquad\qquad\qquad \forall z{:}\mathbb{N}.(z \in k_1 \rightarrow z \in l)) \end{array}$$

- None of this changes the program!

Constructive Synthesis (10)

- Introduce $\wedge$ (with some administrative rewriting) – two branches

  1. $\forall \mathtt{l}:\mathbb{N}.\partial(\forall \mathtt{z}:\mathbb{N}.z \in \mathtt{k}_1 \rightarrow z \in \mathtt{l})$
     $\vdash \forall \mathtt{l}:\mathbb{N}.\partial(\forall \mathtt{z}:\mathbb{N}.(z = \mathtt{k}_0 \rightarrow z \in \mathtt{l}))$

  2. $\forall \mathtt{l}:\mathbb{N}.\partial(\forall \mathtt{z}:\mathbb{N}.z \in \mathtt{k}_1 \rightarrow z \in \mathtt{l})$
     $\vdash \forall \mathtt{l}:\mathbb{N}.\partial(\forall \mathtt{z}:\mathbb{N}.(z \in \mathtt{k}_1 \rightarrow z \in \mathtt{l}))$

  and the program is

```
p( [], L )        :- true.
p( [K0|K1], L )   :- ... ,
                     ...
```

- Proof of case 2 is by appeal to induction hypothesis, to give program

```
p( [], L )        :- true.
p( [K0|K1], L )   :- ... ,
                     p( K1, L ).
```

Constructive Synthesis (11)

- Proof of case 1:

$$\forall l{:}\mathbb{N}.\partial(\forall z{:}\mathbb{N}.z \in k_1 \rightarrow z \in l)$$
$$\vdash \forall l{:}\mathbb{N}.\partial(\forall z{:}\mathbb{N}.(z = k_0 \rightarrow z \in l))$$

Rewrite to give

$$\vdash \forall l{:}\mathbb{N}.\partial(k_0 \in l))$$

Induction on l synthesises `member/2`, and a case split on $k_0 = head(l)$ gives us:

```
p( [], L )          :- true.
p( [K0|K1], L )     :- q( K0, L ),
                       p( K1, L ).

q( K0, [] )         :- false.
q( K0, [L0|L1] )    :- K0 = L0;
                       q( K0, L1 ).
```

Constructive Synthesis (12)

- The example shows
  - similarity with PE (PD) and Compiling Control – the extracted program is a "summary" of the unifications performed in an abstract (Prolog?) execution
  - reasoning system is much more powerful than resolution
  - more powerful steps are "compiled" into the extracted program

- Other questions:
  - What about non-ground queries? (see Wiggins 92b)
  - What about modularity? (see Wiggins 94)
  - What about correctness?

Constructive Synthesis (13)

- Correctness

  For a specification $\mathcal{S}$ expressed as a conjecture

  $$\Gamma \vdash \partial(\mathcal{S})$$

  where Γ is a hypothesis list, an extracted pure logic program $\mathcal{P}$, and a background theory Δ,

  $$\Delta, \Gamma \vdash \mathcal{S} \leftrightarrow \mathcal{P}$$

- Advantage: this is proven for the proof system, not for each synthesised program, so it only has to be done once

- Termination

  We know that the extracted programs terminate, trivially:

  - non-recursive programs terminate
  - recursive programs introduced by induction are well-founded

Deductive Synthesis – General Description

A *deductive synthesis* is the iterative construction of a program P, via a sequence

$$\mathsf{P}_1 \subseteq \mathsf{P}_2 \subseteq \cdots \subseteq \mathsf{P}_n \subseteq \mathsf{P}$$

from

– a *specification framework* $\mathcal{F}$

– a set of *goals* $\mathsf{G} = \{\mathsf{g}_i\}$

such that P is *correct* for each g_i:

$$\mathsf{P} \vdash \mathsf{g}_i \ \textit{if} \ \ \mathcal{F} \models \mathsf{g}_i \ (\textit{partial correctness})$$

$$\mathsf{P} \vdash \mathsf{g}_i \ \textit{iff} \ \ \mathcal{F} \models \mathsf{g}_i \ (\textit{total correctness})$$

(Thus, *synthesis* means *specialisation* of $\mathcal{F}$ wrt G.)

Correctness:

How to design synthesis methods that ensure correctness?

Halting Problem:

When can the synthesis process stop?

Deductive Synthesis of Logic Programs

LP provides a uniquely uniform logical framework for specification, deduction, and programs.

- *The Starting Point*:

  A *specification* $\langle \mathcal{F} \cup \mathsf{D}_0, \mathsf{G} \rangle$:

  – $\mathcal{F}$ is *specification framework*

  i.e. a set of axioms characterising the problem domain.

  Example: $\mathcal{F}$ may contain a theory of *lists* (complete with induction schemas) such as Clark & Tärnlund 77.

  – D_0 is a *definition axiom*:

  $$\forall\, (\mathsf{r}(\mathsf{x}) \leftrightarrow \mathsf{R}(\mathsf{x}))$$

  that defines r in terms of a formula R in the language of $\mathcal{F}$.

  (r is Hogger's *principal specified relation*)

  – G is a set of *goals* of the form

  $$\exists \mathsf{r}(\mathsf{t}_0), \exists \mathsf{r}(\mathsf{t}_1), \ldots$$

  ($\mathsf{r}(\mathsf{x})$ is Hogger's *most general goal*.)

Deductive Synthesis of Logic Programs

An *answer* of $\exists \mathsf{r}(\mathsf{t}) \in \mathsf{G}$ in $\mathcal{F}$ is any $\mathsf{r}(\mathsf{t}\sigma)$ such that $\mathcal{F} \cup \mathsf{D}_0 \models \mathsf{r}(\mathsf{t}\sigma)$, where σ is a substitution.)

Thus $\langle \mathcal{F} \cup \mathsf{D}_0, \mathsf{G} \rangle$ specifies the set of *answers* $\mathsf{r}(\mathsf{t}\sigma)$.

Deductive Synthesis of Logic Programs

- *The Synthesis Process*:

  Can be formalised as a sequence of the form

$$
\begin{array}{ll}
 & \langle \mathcal{F} \cup \mathsf{D}_0 \cup \mathsf{Ax}(\mathsf{P}_0), \mathsf{G} \rangle \\
\Rightarrow & \langle \mathcal{F} \cup \mathsf{D}_1 \cup \mathsf{Ax}(\mathsf{P}_1), \mathsf{G} \rangle \\
\cdots & \\
\Rightarrow & \langle \mathcal{F} \cup \mathsf{D}_n \cup \mathsf{Ax}(\mathsf{P}_n), \mathsf{G} \rangle
\end{array}
$$

  where

$\mathsf{D}_0 \subseteq \cdots \subseteq \mathsf{D}_n$	are (sets of) definition axioms for defining (new) predicates
$\mathsf{P}_0 \subseteq \cdots \subseteq \mathsf{P}_n$	are logic programs
$\mathsf{P}_0 = \{\ \}$	
$\mathsf{Ax}(\mathsf{P})$	is the *if*-part of $\mathsf{Comp}(\mathsf{P})$

  such that

$$\mathcal{F} \cup \mathsf{D}_k \cup \mathsf{Ax}(\mathsf{P}_k) \vdash \mathsf{Ax}(\mathsf{P}_{k+1}),\ 0 \leq k < n \qquad (1)$$

  (1) ensures that every program P_k is *partially correct*.

Deductive Synthesis of Logic Programs

- *The Halting Problem*:

  $\mathsf{P}_1, \ldots, \mathsf{P}_n$ are derived incrementally, i.e.

  $$\mathsf{P}_1 \subseteq \cdots \subseteq \mathsf{P}_n \,.$$

  When can we stop the synthesis process so that P_i is *totally correct*?

- *Synthesis Methods*:

  Correct methods must be devised for deriving $\mathsf{P}_1 \subseteq \cdots \subseteq \mathsf{P}_n$

  Most existing methods guarantee *partial correctness*, halt synthesis arbitrarily, then prove *total correctness*.

Example

- *Starting point*:

$$\langle \mathsf{S}_{list} \cup \mathsf{D}_{mem} \cup \mathsf{D}_{inc}, \mathsf{G} \rangle$$

where

– S_{list} is a theory of lists.

– D_{mem} is the definition axiom for *mem*:

$$\begin{aligned} &mem(\mathsf{X}, \mathsf{L}) \leftrightarrow \\ &\mathsf{L} = [\mathsf{H}|\mathsf{T}] \wedge \mathsf{X} = \mathsf{H} \vee mem(\mathsf{X}, \mathsf{T}) \end{aligned} \tag{2}$$

– D_{inc} is the definition axiom for *inc*:

$$\begin{aligned} &inc(\mathsf{L}_1, \mathsf{L}_2) \leftrightarrow \\ &\forall \mathsf{X} \,.\, (mem(\mathsf{X}, \mathsf{L}_1) \rightarrow mem(\mathsf{X}, \mathsf{L}_2)) \end{aligned}$$

– G is $\{\exists \mathsf{L}_2 .\, inc(\mathsf{L}_1, \mathsf{L}_2) \mid \mathsf{L}_1 \text{ is a ground list}\}$

The condition for partial correctness (1) here is

$$\begin{aligned} &\mathsf{S}_{list} \cup \mathsf{D}_{mem} \cup \mathsf{D}_{inc} \cup \mathsf{D}_k \cup Ax(\mathsf{P}_k) \\ &\vdash Ax(\mathsf{P}_{k+1}), \quad \text{for } 0 \leq k < n \end{aligned} \tag{3}$$

Example

- *Synthesis process*:
  (using a 'manual' synthesis method)

1. To synthesise a clause for inc, we deduce it directly from D_{inc} as follows. Since we have

$$\begin{aligned}&inc([\mathsf{H}|\mathsf{T}],\mathsf{L}_2)\\ &\leftrightarrow \forall \mathsf{X}.\,(mem(\mathsf{X},[\mathsf{H}|\mathsf{T}]) \rightarrow mem(\mathsf{X},\mathsf{L}_2))\\ &\leftrightarrow \forall \mathsf{X}.\,(\neg mem(\mathsf{X},[\mathsf{H}|\mathsf{T}]) \vee mem(\mathsf{X},\mathsf{L}_2))\end{aligned}$$

   we can deduce

$$\begin{aligned}&inc([\mathsf{H}|\mathsf{T}],\mathsf{L}_2)\\ \leftrightarrow\ &\forall \mathsf{X}.\,((\neg \mathsf{X}=\mathsf{H} \wedge \neg mem(\mathsf{X},\mathsf{T}))\vee\\ &\quad mem(\mathsf{X},\mathsf{L}_2))\end{aligned}$$

   from D_{mem} (2).

Example

Applying the distributivity law to the right-hand-side, we get

$$\begin{aligned} & inc([\mathsf{H}|\mathsf{T}], \mathsf{L}_2) \\ \leftrightarrow\ & \forall \mathsf{X}.\, ((\neg \mathsf{X} = \mathsf{H} \vee mem(\mathsf{X}, \mathsf{L}_2))\wedge \\ & (\neg mem(\mathsf{X}, \mathsf{T}) \vee mem(\mathsf{X}, \mathsf{L}_2)) \\ \leftrightarrow\ & \forall \mathsf{X}.\, (\mathsf{X} = \mathsf{H} \rightarrow mem(\mathsf{X}, \mathsf{L}_2))\wedge \\ & (mem(\mathsf{X}, \mathsf{T}) \rightarrow (mem(\mathsf{X}, \mathsf{L}_2)) \end{aligned}$$

which gives

$$\begin{aligned} & inc([\mathsf{H}|\mathsf{T}], \mathsf{L}_2) \\ & \leftrightarrow mem(\mathsf{H}, \mathsf{L}_2) \wedge inc(\mathsf{T}, \mathsf{L}_2) \end{aligned} \tag{4}$$

Example

Thus we have deduced (4) from the axioms in

$$\mathsf{S}_{list} \cup \mathsf{D}_{mem} \cup \mathsf{D}_{inc}$$

and so we can use its *if*-part, namely the clause

$$\begin{array}{l} inc([\mathsf{H}|\mathsf{T}], \mathsf{L}_2) \\ \leftarrow mem(\mathsf{H}, \mathsf{L}_2), inc(\mathsf{T}, \mathsf{L}_2) \end{array}$$

and synthesise

$$\mathsf{P}_1 = \left\{ \begin{array}{l} inc([\mathsf{H}|\mathsf{T}], \mathsf{L}_2) \\ \leftarrow mem(\mathsf{H}, \mathsf{L}_2), inc(\mathsf{T}, \mathsf{L}_2) \end{array} \right\}$$

That is we have performed the synthesis step

$$\begin{array}{rl} & \langle \mathsf{S}_{list} \cup \mathsf{D}_{mem} \cup \mathsf{D}_{inc}, \mathsf{G} \rangle \\ \Rightarrow & \langle \mathsf{S}_{list} \cup \mathsf{D}_{mem} \cup \mathsf{D}_{inc} \cup \mathsf{P}_1, \mathsf{G} \rangle \end{array}$$

in such a way that (3) is satisfied, that is

$$\mathsf{S}_{list} \cup \mathsf{D}_{mem} \cup \mathsf{D}_{inc} \vdash \mathsf{P}_1 \qquad (5)$$

thus ensuring the partial correctness of P_1.

Example

2. Similarly, we could also derive the clauses

$$\begin{aligned} inc([], []) &\leftarrow \\ inc([\mathsf{H}], [\mathsf{H}]) &\leftarrow \\ inc([], \mathsf{L}_2) &\leftarrow \end{aligned}$$

   from

$$\langle \mathsf{S}_{list} \cup \mathsf{D}_{mem} \cup \mathsf{D}_{inc} \cup \mathsf{P}_1, \mathsf{G} \rangle$$

Brief Survey of Synthesis Methods

Most existing synthesis methods

– use (first-order) logic sentences for $\mathcal{F}$

– deduce logic clauses by correct inference rules directly from these logic sentences.

- **Clark, Hansson and Tärnlund**
  - $\mathcal{F}$ is a set of *iff*-formulas (axioms for relations and data structures).
  - $\mathfrak{r}$ is defined by an *iff*-formula in terms of atoms already defined in $\mathcal{F}$.
  - Deduce logic programs from the axiomatisation either
    - by natural deduction (Hansson and Tärnlund), or
    - by symbolic execution, or re-writing of G (Clark and Tärnlund).
  - Partial correctness guaranteed, but total correctness must be proved.
  - Derivation editor implemented for Hansson and Tärnlund's method.

Brief Survey of Synthesis Methods

- **Hogger**
  - $\mathcal{F}$ is a set of *iff*-formulas.
  - r is defined by an *iff*-formula.

    $\leftarrow \mathsf{r}(\mathsf{x})$ is treated as a goal to be solved by a 'logic procedure' consisting of:

    * the *only-if* part of the definition of r
    * other relevant 'clauses' from $\mathcal{F}$ which are necessary for solving this goal

    using a resolution-like mechanism.

    The 'solution' of $\leftarrow \mathsf{r}(\mathsf{x})$ is deemed sufficient when a procedure for $\mathsf{r}(\mathsf{x})$ can be extracted from the resolution steps.

    Alternatively, he also carries out the derivation using non-resolution inference rules.
  - Partial correctness guaranteed, but total correctness must be proved.

Brief Survey of Synthesis Methods

- **Lau and Prestwich**
  - $\mathcal{F}$ is a set of *iff*-formulas.
  - r is defined by an *iff*-formula.
  - Deduction of the clause(s) for solving $\mathsf{r(x)}$ is by structured re-writing.

    Each deduction is automatically decomposed into sub-deductions.

    Sub-deductions are automatically composed into their parent deductions.
  - The user has to specify the recursion pattern in the required procedure.
  - Only partial correctness guaranteed.
  - Semi-automated synthesis system implemented.

Brief Survey of Synthesis Methods

- **Kraan**
  - Method based on *proof planning*.
  - $\mathcal{F}$ is a (typed) theory.
  - $\mathsf{r}(\mathsf{x})$ is an *iff*-formula
  - $\mathsf{r}(\mathsf{x}) \leftrightarrow \mathsf{P}(\mathsf{x})$ is used as a conjecture that the yet-to-be-synthesised program $\mathsf{P}(\mathsf{x})$ will compute $\mathsf{R}(\mathsf{x})$.

    In planning the proof of this conjecture, P's body is represented by a meta-variable.

    When the proof plan is completed, this meta-variable gets instantiated to logical formulas deduced from $\mathsf{r}(\mathsf{x}) \leftrightarrow \mathsf{R}(\mathsf{x})$.
  - Fully automated synthesis system implemented.

Brief Survey of Synthesis Methods

- **Bibel**
  - Method implemented in LOPS.
  - The specification is really an 'input-output' synthesis conjecture (as in constructive synthesis).
  - The specified program is derived by (various strategies for) re-writing formulas as well as using domain knowledge to generate relevant theorems.
  - However, we may regard the domain knowledge as $\mathcal{F}$, and LOPS can be re-cast as a deductive synthesis method for logic programs

Transformational Synthesis

Many synthesis methods are based on unfold-fold transformations of logic programs *à la* Sato & Tamaki, Pettorossi & Proietti.

Instead of the specification

$$\langle \mathcal{F} \cup \mathsf{D}_0, \mathsf{G} \rangle$$

use

$$\langle \mathsf{P} \cup \mathsf{D}_0, \mathsf{G} \rangle$$

where

– P is a logic program

– D_0 is a(n) (extended) Horn clause

- **Dayantis**

  Automatically transforms a class of extended Horn clauses into Horn clauses.

- **Kanamori and Horiuchi**

  D_0 is a *definite formula* (Horn clause extended by universally quantified implicational formulae).

  Uses Tamaki & Sato 84 for transformation.

- **Kawamura**

  Automatically transforms a class of definite formulas into Horn clauses.

Transformational Synthesis

- **Komorowski**
  Transformation based on *partial deduction* (partial evaluation of logic programs).
- **Pettorossi and Proietti**
  Synthesises programs from unfold-fold proofs.
- **Sato and Tamaki**
  Deterministic synthesis algorithms based on their first-order unfold-fold system.

Conclusion (1)

- For a more thorough survey, see Deville & Lau 94.

- Is LP synthesis worth pursuing?
  - YES!
  - It is the natural conclusion of Kowalski's Equation:

$$\mathsf{Algorithm} = \mathsf{Logic} + \mathsf{Control}$$

  - We extend the expressivity of the logic and the power of the inference system, so…
  - …specifications need not be executable in a resolution based framework, and…
  - …hard work for programs can be done once in advance, and compiled into simplified resolution-friendly programs

- However…

Conclusion (2)

- …we badly need automation, because synthesis proofs (of whatever kind) are usual very large and mostly boring

- Hard problem…

  …but we have good ways of beginning

- Techniques are becoming available to control search for programs in both constructive and deductive frameworks, e.g.

  Kraan 93b

  Proietti & Pettorossi 94

Claims

1. Logic Programming has the best potential for general software engineering

   (i.e. addressing the software crisis)

   Logic Program Synthesis is beginning to justify this claim.

2. Logic Program Synthesis is the future of Logic Programming

   – NB not excluding other LP technology

   – For example, synthesis is compatible with CLP and parallel LP

References

[Basin 94] D. Basin. Logic frameworks for logic programs. In F. Turini & L. Fribourg, editor, *Proceedings of the Fourth International Workshops on Logic Program Synthesis and Transformation and Meta-Programming in Logic*, LNCS. Springer-Verlag, 1994.

[Bibel & Hörnig 84] W. Bibel and K. M. Hörnig. LOPS – a system based on a strategical approach to program synthesis. In A. W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic program construction techniques*, chapter 3, pages 69–89. MacMillan, 1984.

[Bruynooghe *et al* 89] M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling control. *Journal of Logic Programming*, pages 135–162, 1989.

[Clark & Sickel 77] K.L. Clark and S. Sickel. Predicate logic: A calculus for deriving programs. In *Proc. IJCAI-77*, pages 419–420, 1977.

[Clark & Tärnlund 77] K.L. Clark and S.-Å. Tärnlund. A first order theory of data and programs. In *Proc. IFIP 77*, pages 939–944. North-Holland, 1977.

[Constable 82] R.L. Constable. Programs as proofs. Technical Report TR 82-532, Dept. of Computer Science, Cornell University, November 1982.

[Dayantis 87] G. Dayantis. Logic program derivation for a class of first order logic relations. In *Proc. IJCAI-87*, pages 9–14, 1987.

[Deville & Lau 94] Y. Deville and K.K. Lau. Logic program synthesis. *Journal of Logic Programming*, 1994. To appear, Special Issue on "Ten Years of Logic Prgramming".

[Deville 90] Y. Deville. *Logic Programming: Systematic Program Development.* International Series in Logic Programming. Addison-Wesley, 1990.

[Eriksson *et al* 86] A. Eriksson, A.-L. Johansson, and S.-Å. Tärnlund. Towards a derivation editor. In M. van Caneghem and D.H.D. Warren, editors, *Logic Programming and its Applications*, pages 117–126. Ablex Publ. Co., 1986.

[Fribourg 90] L. Fribourg. Extracting logic programs from proofs that use extended Prolog execution and induction. In D.H.D. Warren and P. Szeredi, editors, *Proc. 7th Int. Conf. on Logic Programming*, pages 685–699. MIT Press, 1990.

[Fuchs 92] Norbert E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, pages 323–334, September 1992.

[Hansson & Tärnlund 79] Å. Hansson and S.-Å. Tärnlund. A natural programming calculus. In *Proc. IJCAI-79*, pages 348–355, 1979.

[Hogger 81] C.J. Hogger. Derivation of logic programs. *J. ACM*, 28(2):372–392, April 1981.

[Hogger 84] C.J. Hogger. *Introduction to Logic Programming.* Academic Press, 1984.

[Kanamori & Horiuchi 87] T. Kanamori and K. Horiuchi. Construction of logic programs based on generalized unfold/fold rules. In *Proc. 4th Int. Conf. on Logic Programming, Melbourne*, pages 744–768, 1987.

[Kanamori & Seki 86] T. Kanamori and H. Seki. Verification of prolog programs using an extension of execution. In E. Shapiro, editor, *Third International Conference on Logic Programming*, LNCS 225, pages 475–489. Springer-Verlag, 1986.

[Kawamura 92] T. Kawamura. Logic program synthesis from first order logic specifications. In *Proc. Fifth Generation Computer Systems 92*, pages 463 – 472, Tokyo, 1992. Ohmsha.

[Komorowski 89] J. Komorowski. Synthesis of programs in the framework of partial deduction. Technical Report Ser. A, No 81, Dept. of Computer Science, Abo Akademi, 1989.

[Kowalski 85] R. Kowalski. The relation between logic programming and logic specification. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 11–27. Prentice-Hall, 1985.

[Kraan *et al* 93a] I. Kraan, D. Basin, and A. Bundy. Logic program synthesis via proof planning. In K.K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, pages 1–14. Springer-Verlag, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-92-244 and Edinburgh DAI Research Report 603.

[Kraan *et al* 93b] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for logic program synthesis. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming.* MIT Press, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-93-214 and Edinburgh DAI Research Report 638.

[Lau & Ornaghi 94a] K.K. Lau and M. Ornaghi. A formal view of specification, deductive synthesis and transformation of logic programs. In Y. Deville, editor, *Logic Program Synthesis and Transformation*, Workshops in Computing, pages 10–31. Springer-Verlag, 1994. Proceedings of Third International Workshop on Logic Program Synthesis and Transformation.

[Lau & Ornaghi 94b] K.K. Lau and M. Ornaghi. On specification frameworks and deductive synthesis of logic programs. 1994. To be presented at LOPSTR 94.

[Lau & Prestwich 90] K.K. Lau and S.D. Prestwich. Top-down synthesis of recursive logic procedures from first-order logic specifications. In D.H.D. Warren and P. Szeredi, editors, *Proc. 7^{th} Int. Conf. on Logic Programming*, pages 667–684. MIT Press, 1990.

[Lau *et al* 94] K.K. Lau, M. Ornaghi, and S.-Å. Tärnlund. The halting problem for deductive synthesis of logic programs. In *Proc.* 11^{th} *Int. Conf. on Logic Programming*, MIT Press, 1994.

[Lloyd & Shepherdson 87] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. Technical Report CS-87-09, Department of Computer Science, University of Bristol, 1987. Also in Journal of Logic Programming.

[Muggleton & De Raedt 94] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *J. Logic Programming*, 1994. Special Volume, 'Ten Years of Logic Programming'.

[Naish 92] L. Naish. Types and the intended meaning of logic programs. In F. Pfenning, editor, *Types in Logic Programming.* MIT Press, 1992.

[Parigot 92] M Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proceedings of LPAR '92*, pages 190–201. Springer, 1992. LNCS vol 624.

[Proietti & Pettorossi 94] M. Proietti and A. Pettorossi. Synthesis of programs from unfold/fold proofs. In Y. Deville, editor, *Logic Program Synthesis and Transformation*, Workshops in Computing, pages 141–158. Springer-Verlag, 1994. Proceedings of Third International Workshop on Logic Program Synthesis and Transformation.

[Sato & Tamaki 84] T. Sato and H. Tamaki. Transformational logic program synthesis. In *Proc. Int. Conf. on Fifth Generation Computer System, Tokyo*, pages 195–201, 1984.

[Sato & Tamaki 89] T. Sato and H. Tamaki. First order compiler: A deterministic logic program synthesis algorithm. *J. Symbolic Computation*, 8:605–627, 1989.

[Tamaki & Sato 84] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *Proc. 2nd Int. Conf. on Logic Programming, Uppsala*, pages 127–138, 1984.

[Wiggins 92a] G. A. Wiggins. Negation and control in automatically generated logic programs. In A. Pettorossi, editor, *Proceedings of META-92*. Springer Verlag, Heidelberg, 1992. LNCS Vol. 649.

[Wiggins 92b] G.A. Wiggins. Synthesis and transformation of logic programs in the Whelk proof development system. In K.R. Apt, editor, *Proc. 1992 Int. Joint Conf. and Symp. on Logic Programming*, pages 351–365. MIT Press, 1992.

[Wiggins 94] G. A. Wiggins. Whelk type theory, 1994. Presented at the Fourth International Workshop on Logic Program Synthesis and Transformation, Pisa, Italy.

[Wiggins *et al* 91] G. A. Wiggins, A. Bundy, I. Kraan, and J. Hesketh. Synthesis and transformation of logic programs through constructive, inductive proof. In K-K. Lau and T. Clement, editors, *Proceedings of LoPSTr-91*, pages 27–45. Springer Verlag, 1991. Workshops in Computing Series.