Synthesis and formal specification

Alan Smaill
Feb 11th 2003

Synthesis and theorem proving

- Use to (generate and) verify proof obligations
  - cf work in Isabelle, HOL, etc
- Use to generate programs
- Use to guide design choice (less work)

Precedents

Some work we should take into account.

- Simply typed lambda calculus
  Take the type of desired program, e.g.
  \[(a \rightarrow (b \rightarrow c)) \rightarrow (a \rightarrow b) \rightarrow (b \rightarrow c)\]

Then we can find corresponding program automatically, if one exists
(Curry-Howard and decidability of corresponding propositional logic).


This aims to provide a formal basis and an implementation to subsume various systems developed at Kestrel (eg KIDS, DTRE).

- Category theoretic formulation, involving two sorts of composition.
- Proof theoretic ideas preferred.
- Graphical Interface, recording design process.
- Uses (extended) resolution prover for proof obligations.
Aims

- Provide a common conceptual basis for several existing systems.
- Support software systems development: provide "a scaffolding which takes care of the mundane details, thus letting the developer identify and focus on the creative part."
- Represent explicitly the structure of specifications, refinements and program modules.

Specifications

These are finite presentation of theory in (classical) higher-order sorted logic (some ideas borrowed from type theories). Booleans, quantification and polymorphic equality are built in. Take finite products and co-products as primitive, and function sorts. Fairly close to EML (and logic used for EU Specification project).

Using proof-planning

We can use proof planning ideas to guide synthesis in this style. We use a version in HO logic that gives powerful unification, allowing declarative theory of specification refinement, and of heuristic guidance.

A key idea here: we can separate compile-time and run-time inference via the use of EML functors:

- Functor carries both type info (checkable by ML compiler), and axiomatic info. This can be verified when the functor is designed, giving us a decomposition operator guaranteed to be sound.
- The applicability of the functor involves inference too; if we are careful, we could make this tractable.

Example

signature BASE1 = (* pure SML *)
  sig
  type t1
  type t2
  val f : t1 -> t2
end;
signature OUTSIG1 =
  sig
  type a
  type b
  val f1 : a * a -> b * b
  val f2 : a * b -> b * a
end
Also give functor signatures:

fun sig Func1 ( structure S1 : BASE1 ) = OUTSIG1

Suppose we have a target:

signature TARGET =
    sig
    type tar1
    type tar2
    val tfun2 : tar1 * tar2 -> tar2 * tar1
    val tfun1 : tar1 * tar1 -> tar2 * tar2
end

Two sorts of steps needed: renaming functor:

functor renameOut1ToTarget ( structure 01 : OUTSIG1 )
    : TARGET =
    struct
        type tar1 = 01.a
        type tar2 = 01.b
        val tfun1 = 01.f1
        val tfun2 = 01.f2
    end

And recognise we have library object, suitably parametrised: generate declarations:

structure Base1 : BASE1 = Base1();
structure Out1 : OUTSIG1 = Func1 ( structure S1 = Base1 )
structure T : TARGET =
    renameOut1ToTarget ( structure 01 = Out1 );

Adding axioms

Axioms can appear throughout the signatures. Functors now carry proof obligations:

if argument structures satisfy their signature axioms, then result signature satisfies axioms.

For inductively defined data-structures, these proofs typically involve induction.
Synthesis, control

Top-down synthesis will solve target with library module, if possible; otherwise match with functor result signature and backchain. This matching implicitly invokes inference.

The choice of decomposition functor to use can be subject to proof-planning considerations (as the pre-condition to method).

Conclusions, questions

- Can relate proof planning closely to this style of synthesis.
- Need experience in decomposition
- Top-down approach in natural (but others can be supported)
- What about distributed synthesis?
- Non-declarative properties??