Synthesizing implementations using a theorem prover

Overview:

- A little potted history
  - from 60s AI to today’s formal methods
- Some recent work on hardware and software synthesis
  - overview of a Cambridge-Utah collaboration
- Discussion, suggestions and challenges
  - idealism versus pragmatism
Synthesis by theorem proving is a very old idea

Date: Mon, 11 Feb 2008 22:09:21 -0800
From: Cordell Green <green@kestrel.edu>
To: Mike.Gordon@cl.cam.ac.uk
Subject: Synthesizing Implementations

Mike,

I saw the announcement about your upcoming talk at ttvsi on "Synthesizing Implementations Using a Theorem Prover"
Sounds great. Is this related to my ancient work on that topic?
Cordell

Incomplete potted history of deductive synthesis

- **Uniform proof**
  - “automatic programming” by resolution theorem proving
  - extract program from resolution proof of $\forall x. \exists y. R(x, y)$
  - instance of a general problem solving method (Green, QA3)

- **Deductive synthesis**
  - special deductive framework for program construction
  - first-order tableau system for both humans and machines
  - used to construct some software for NASA Cassini mission

- **Constructive proof**
  - extract program from constructive proof $p$ of $\forall x. \exists y. R(x, y)$
  - proof $p$ is a functional program satisfying $R(x, p(x))$
  - mechanised by Nuprl and Coq systems

- **Behavioural synthesis**
  - refine formal logic hardware specification to circuits
  - interactive sub-goaling + automatic point tools (scheduling)
  - commercialised (AHL Ltd)

- **Refinement**
  - special rules for wide-spectrum language
  - program construction not inside a standard logic
  - maybe most widely used in practice
Incomplete potted history of deductive synthesis

- **Uniform proof** [Slagle, Green & Yates, Waldinger]
  - “automatic programming” by resolution theorem proving
  - extract program from resolution proof of $\forall x. \exists y. R(x, y)$
  - instance of a general problem solving method (Green, QA3)

- **Deductive synthesis**
  - special deductive framework for program construction
  - first-order tableau system for both humans and machines
  - used to construct some software for NASA Cassini mission

- **Constructive proof**
  - extract program from constructive proof $p$ of $\forall x. \exists y. R(x, y)$
  - proof $p$ is a functional program satisfying $R(x, p(x))$
  - mechanised by Nuprl and Coq systems

- **Behavioural synthesis**
  - refine formal logic hardware specification to circuits
  - interactive sub-goaling + automatic point tools (scheduling)
  - commercialised (AHL Ltd)

- **Refinement**
  - special rules for wide-spectrum language
  - program construction not inside a standard logic
  - maybe most widely used in practice
Incomplete potted history of deductive synthesis

- **Uniform proof**
  - "automatic programming" by resolution theorem proving
  - extract program from resolution proof of $\forall x \exists y R(x, y)$
  - instance of a general problem solving method (Green, QA3)

- **Deductive synthesis** [Manna & Waldinger, Stickel, Lowry]
  - special deductive framework for program construction
  - first-order tableau system for both humans and machines
  - used to construct some software for NASA Cassini mission

- **Constructive proof**
  - extract program from constructive proof $p$ of $\forall x \exists y R(x, y)$
  - proof $p$ is a functional program satisfying $R(x, p(x))$
  - mechanised by Nuprl and Coq systems

- **Behavioural synthesis**
  - refine formal logic hardware specification to circuits
  - interactive sub-goaling + automatic point tools (scheduling)
  - commercialised (AHL Ltd)

- **Refinement**
  - special rules for wide-spectrum language
  - program construction not inside a standard logic
  - maybe most widely used in practice
Incomplete potted history of deductive synthesis

- Uniform proof
  - “automatic programming” by resolution theorem proving
  - extract program from resolution proof of \( \forall x. \exists y. R(x, y) \)
  - instance of a general problem solving method (Green, QA3)

- Deductive synthesis
  - special deductive framework for program construction
  - first-order tableau system for both humans and machines
  - used to construct some software for NASA Cassini mission

- Constructive proof [Constable, Huet & Coquand]
  - extract program from constructive proof \( p \) of \( \forall x. \exists y. R(x, y) \)
  - proof \( p \) is a functional program satisfying \( R(x, p(x)) \)
  - mechanised by Nuprl and Coq systems

- Behavioural synthesis
  - refine formal logic hardware specification to circuits
  - interactive sub-goaling + automatic point tools (scheduling)
  - commercialised (AHL Ltd)

- Refinement
  - special rules for wide-spectrum language
  - program construction not inside a standard logic
  - maybe most widely used in practice
Incomplete potted history of deductive synthesis

- Uniform proof
  - “automatic programming” by resolution theorem proving
  - extract program from resolution proof of $\forall x \exists y R(x, y)$
  - instance of a general problem solving method (Green, QA3)

- Deductive synthesis
  - special deductive framework for program construction
  - first-order tableau system for both humans and machines
  - used to construct some software for NASA Cassini mission

- Constructive proof
  - extract program from constructive proof $\pi$ of $\forall x \exists y R(x, y)$
  - proof $\pi$ is a functional program satisfying $R(x, \pi(x))$
  - mechanised by Nuprl and Coq systems

- Behavioural synthesis [Hanna, Fourman, Johnson]
  - refine formal logic hardware specification to circuits
  - interactive sub-goaling + automatic point tools (scheduling)
  - commercialised (AHL Ltd)

- Refinement
  - special rules for wide-spectrum language
  - program construction not inside a standard logic
  - maybe most widely used in practise
Incomplete potted history of deductive synthesis

- **Uniform proof**
  - "automatic programming" by resolution theorem proving
  - extract program from resolution proof of $\forall x \exists y \ R(x, y)$
  - instance of a general problem solving method (Green, QA3)

- **Deductive synthesis**
  - special deductive framework for program construction
  - first-order tableau system for both humans and machines
  - used to construct some software for NASA Cassini mission

- **Constructive proof**
  - extract program from constructive proof $\rho$ of $\forall x \exists y \ R(x, y)$
  - proof $\rho$ is a functional program satisfying $R(x, \rho(x))$
  - mechanised by Nuprl and Coq systems

- **Behavioural synthesis**
  - refine formal logic hardware specification to circuits
  - interactive sub-goaling + automatic point tools (scheduling)
  - commercialised (AHL Ltd)

- **Refinement**  [Bjørner & Jones, Back et al., Morgan, Abrial]
  - special rules for wide-spectrum language
  - program construction not inside a standard logic
  - maybe most widely used in practice
Incomplete potted history of deductive synthesis

- **Uniform proof** [Slagle, Green & Yates, Waldinger]
  - “automatic programming” by resolution theorem proving
  - extract program from resolution proof of $\forall x. \exists y. R(x, y)$
  - instance of a general problem solving method (Green, QA3)

- **Deductive synthesis** [Manna & Waldinger, Stickel, Lowry]
  - special deductive framework for program construction
  - first-order tableau system for both humans and machines
  - used to construct some software for NASA Cassini mission

- **Constructive proof** [Constable, Huet & Coquand]
  - extract program from constructive proof $p$ of $\forall x. \exists y. R(x, y)$
  - proof $p$ is a functional program satisfying $R(x, p(x))$
  - mechanised by Nuprl and Coq systems

- **Behavioural synthesis** [Hanna, Fourman, Johnson]
  - refine formal logic hardware specification to circuits
  - interactive sub-goaling + automatic point tools (scheduling)
  - commercialised (AHL Ltd)

- **Refinement** [Bjørner & Jones, Back et al., Morgan, Abrial]
  - special rules for wide-spectrum language
  - program construction not inside a standard logic
  - maybe most widely used in practise
Various deductive approaches differ

- Starting point (initial specification) varies
  - declarative \((\text{find } z \text{ such that } \text{perm}(l, z) \land \text{ord}(z))\)
  - special purpose ‘wide-spectrum’ notation \((Z)\)
  - recursive functions

- Target implementation varies
  - LISP code (early automatic programming)
  - imperative pseudo-code (high level or machine level)
  - circuits

- Deductive framework varies
  - raw logic (first or higher order, classical or constructive)
  - deductive synthesis tableau
  - refinement calculi

- Deduction method varies
  - automatic and uniform proof procedure (resolution/QA3)
  - automatic with user assistance (various refinement tools)
  - pencil and paper
Cambridge-Utah collaborative project

- **Starting point:**
  - function defined in higher order logic (HOL)

- **Target implementation:**
  - machine code; register transfer level (RTL) circuits

- **Deductive framework:**
  - proof rules of logic

- **Deduction method:**
  - special purpose derived inference rule (written in ML)
Proof-producing synthesis

- Start with mathematical requirements
  - non-executable higher order logic
- Derive a function $f$ meeting specification
  - function $f$ defined in HOL (TFL)
- Prove $\vdash$ Implements($imp$, $f$)
  - proof constructs $imp$
- Compiler is a specialised theorem prover
  - generated theorem certifies $imp$
- Each run generates certifying theorem
  - automates verifications that were manual
Discussion and motivation

- A few applications
  - need very high assurance of functional correctness
  - can be specified using formal mathematics

- Our example: implementing cryptographic primitives
  - mathematical specifications via elliptic curves
  - high assurance needed for certification (FIPS, CC)

- Some real world problems we avoid
  - source language with no (or intractable) formal semantics
  - verification of complicated synthesis algorithm
  - need to trust the execution of synthesizing code (e.g. C)
A ‘synthesizable subset’ of higher order logic

- Analogy: synthesizable subset of Verilog HDL
  - Verilog was originally a simulation modelling language
  - tools to compile subsets to circuits were devised later

- Synthesizable subset of HOL versus embedded language
  - no complicated language semantics
  - subset is ‘soft’ – can be expanded

- Challenge: “Tackling the awkward squad” (Peyton Jones)
  - input-output, concurrency, exceptions, partial functions, …
  - tackle by extending synthesizable subset beyond functions
ARM project (still in progress)

- Everything represented in higher order logic
- Proof-producing compiler to ARM machine code
- Proof-producing compiler to Verilog (for FPGA)
- Contributors: Fox, Hurd, Li, Myreen, Owens, Tuerk, Slind
ARM code synthesis (Slind, Li, Myreen)

What does $\vdash$ Implements($code, f$) mean?

- ARM compiler generates Hoare specifications:
  
  $$\text{Implements}(code, f) = \{ \cdots * R\, a\, x * \cdots \} \, code \{ \cdots * R\, a\, f(x) * \cdots \}$$

- Utah front-end (Slind, Li) + Cambridge back-end (Myreen)

- Hoare triple semantics uses accurate processor model
  - ARM4T instruction set architecture (Fox)
  - derived Hoare logic (Myreen)

Example:

$$\{ R\, a\, x * R\, b_\_ * S_\_ * x \neq 0 \}$$

- MOV $b, \#1$
- MUL $b, a, b$
- SUBS $a, a, \#1$
- BNE #−4

$$\{ R\, a\, 0 * R\, b\, (\text{FACTORIAL}(x)) * S_\_ \}$$
What does $\vdash \text{Implements}(\text{circuit}, f)$ mean?

- $\text{circuit}$ performs a handshake to compute $f$
- $\text{Implements}(\text{circuit}, f) = \forall \text{load imp done out}. \text{circuit}(\text{load, inp, done, out}) \Rightarrow \text{Handshake } f (\text{load, inp, done, out})$
Hardware design synthesis (Iyoda, Owens, Slind)

- Compiler generates registers + combinational logic (RTL)
- Example: Handshake(FACTORIAL)
HOL representation of circuits compared with Verilog

\[ \text{HOL} \]

A pretty-printer generates Verilog from HOL netlists

\[ \text{Verilog HDL} \]
Review of proof-producing synthesis from HOL

- From $f$ create $imp$ and prove $\vdash \text{Implements}(f, imp)$
  - $f$ if a functional program in higher order logic
  - $\text{Implements}$ defined differently for hardware and software

- Construction of implementation by various methods
  - apply mechanical refinement rules
  - run conventional algorithm then validate results post hoc

- Verification benefits
  - have verification infrastructure for free ($f$ and $imp$ in HOL)
  - correct-by-construction implementation + certificate

- Issues
  - synthesizable subset is restricted (tail-recursive TFL)
  - code for proof-producing synthesis is challenging
Combining Idealism with Pragmatism

The tension between idealism and pragmatism is as profound (almost) as that between good and evil (and just as pervasive). [Tony Hoare]

- Proof-producing synthesis from HOL is idealism!
  - jettison horrible legacy industry languages
  - replace C, C++, Verilog etc. with formal logic

- Pragmatism confronts
  - precise semantics of real languages
  - trusting execution of tools (e.g. compiler)

- Our compromise: pragmatic at bottom, idealistic above
  - what actually runs is real-world ................. pragmatism
  - accurate processor models ................... pragmatism
  - designs and implementations in logic .......... idealism
  - execution inside a theorem-prover .............. idealism
Ultra-idealistic functional programming

- Functions in HOL simpler than programs in ML or Haskell
  - pro: can reason about them directly
  - con: missing features (lazy, non-termination, exceptions)

- Maybe can define tractable subsets of existing languages
  - Scott Owens’ OCaml light is inspiring, but still complex
  - need operational semantics to relate programs to functions

- Programming with functions not functional programming
  - need a standard language for functions
  - each tool has own concrete syntax
  - start from notations in Isabelle, various HOLs, PVS, Coq
  - select a collection of constructs (abstract syntax)

- Executing functions
  - inside theorem prover
  - verified or verifying compiler
  - translation to ML (MLton) or ACL2 (Common Lisp)
Beyond functions

Compile non-function semantics for the awkward squad

- Hardware pipeline model for interrupts
- Timed temporal logic formulas for real-time
- Bus models for input-output
- Transition systems or sets of traces for concurrency
Beyond functions

Compile non-function semantics for the awkward squad

- Hardware pipeline model for interrupts
- Timed temporal logic formulas for real-time
- Bus models for input-output
- Transition systems or sets of traces for concurrency
Beyond functions

Compile non-function semantics for the awkward squad

- Hardware pipeline model for interrupts
- Timed temporal logic formulas for real-time
- Bus models for input-output
- Transition systems or sets of traces for concurrency
Multicultural logic and theorem proving

- **Good** to have a tool-independent function language?

- **Even better:** a tool-independent general logic
  - higher order logic widely used (Coq, HOL, Isabelle, PVS)
  - ACL2 has best integration with a programming language
  - interesting trade-offs between first and higher-order logics

- **Suggestion:** multi-tool projects
  - groundwork exists (ACL2 ↔ HOL ↔ Isabelle)
  - e.g. HOL probability theory + JVM implementation in ACL2

- **Longer term:** what about set-theory?
  - the standard foundation
  - includes higher order logic (in principle)
  - maybe a good framework for connecting tools?
Conclusions

- Long history of logic as a system implementation language
- Expressive general purpose logics can do everything!
- Balance ideal and real worlds

Challenges:
- tool-independent language for programming with functions
- proper logical treatment of the ‘awkward squad’
- tool-independent language for going beyond functions
- multicultural theorem-proving
THE END