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
--------------------------------------------------------------
```

- Green, C. Application of Theorem Proving to Problem Solving, Technical Note 4. AI Center, SRI International, 333 Ravenswood Ave, Menlo Park, CA 94025, Mar 1969.

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 practise

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 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 [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 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 [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 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 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
  - 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 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 [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

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

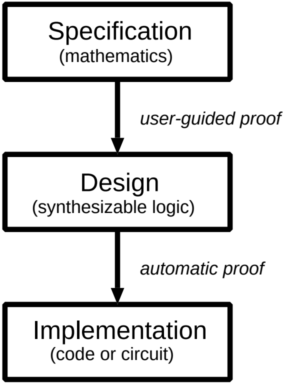
Various deductive approaches differ

- Starting point (initial specification) varies
  - declarative ($\mathit{find\ z\ such\ that\ perm}(l, z) \wedge \mathit{ord}(z)$)
  - 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 $\vdash \mathrm{Implements}(imp, f)$
  - proof constructs *imp*
- Compiler is a specialised theorem prover
  - generated theorem certifies *imp*
- Each run generates certifying theorem
  - automates verifications that were manual

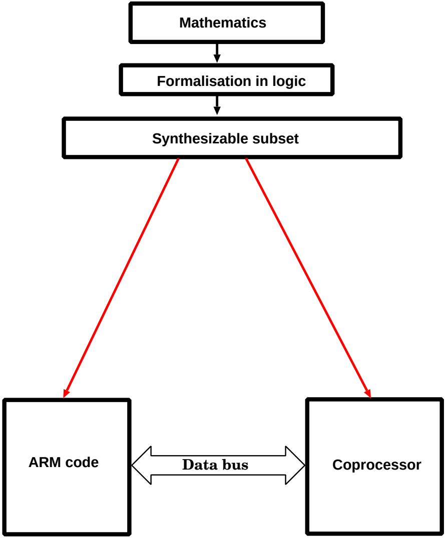
Discussion and motivation

- 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)

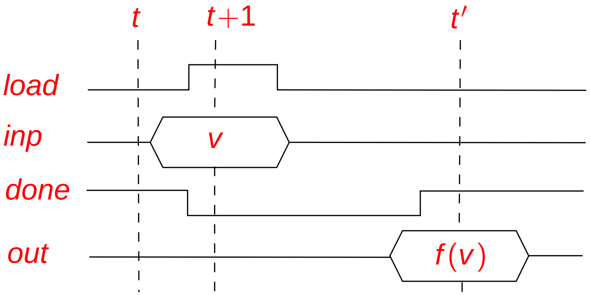
- What does $\vdash$ Implements$(code, f)$ mean?
  - ARM compiler generates Hoare specifications:
  - Implements$(code, f) =$
    $\{\cdots * R\ a\ x\ * \cdots\}\ code\ \{\cdots * R\ a\ f(x)\ * \cdots\}$
- Utah front-end (Slind, Li) + Cambridge back-end (Myreen)
- Hoare triple semantics uses accurate processor model
  - ARM4T instruction set architecture (Fox)
  - derived Hoare logic (Myreen)
- Example: $\{R\ a\ x * R\ b\ \_ * S\ \_\ * x{\neq}0\}$

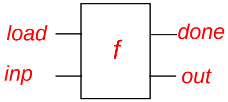
```
MOV b,#1
MUL b,a,b
SUBS a,a,#1
BNE #-4
```

$\{R\ a\ 0 * R\ b\ (\textsc{factorial}(x)) * S\ \_\}$

Hardware meaning of a HOL function

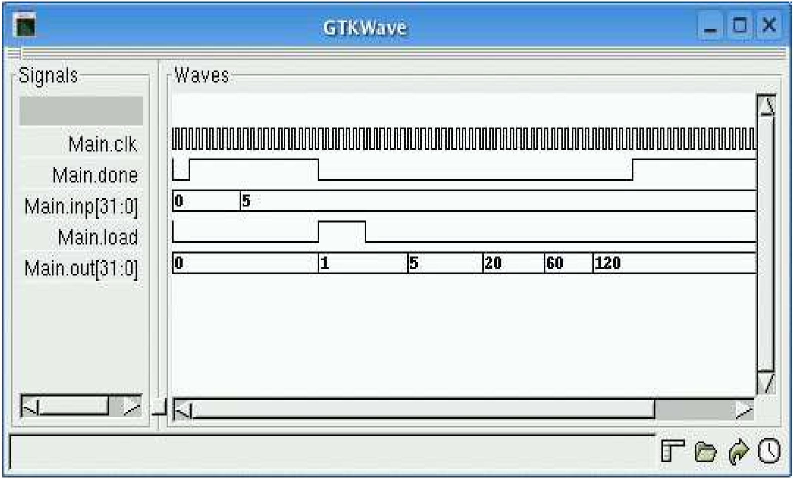
- What does $\vdash$ Implements($circuit$, f) mean?
  - $circuit$ performs a handshake to compute f
  - $\mathrm{Implements}(circuit, f) =$
    $\forall load\ imp\ done\ out.$
    $circuit(load, inp, done, out)$
    $\Rightarrow \mathrm{Handshake}\ f\ (load, inp, done, out)$





Hardware design synthesis (Iyoda, Owens, Slind)

- Compiler generates registers + combinational logic (RTL)
- Example: Handshake(FACTORIAL)



HOL representation of circuits compared with Verilog

```
|− (?v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 v15 v16 v17 v18
     v19 v20 v21 v22 v23 v24 v25 v26 v27 v28 v29 v30 v31 v32 v33 v34
     v35 v36 v37 v38 v39 v40 v41 v42 v43 v44 v45 v46 v47 v48 v49 v50
     v51 v52 v53 v54 v55 v56 v57 v58 v59 v60 v61 v62 v63 v64 v65 v66
     v67 v68 v69 v70 v71 v72 v73 v74 v75 v76 v77 v78 v79 v80 v81 v82
     v83 v84 v85 v86 v87 v88 v89 v90 v91 v92 v93 v94 v95 v96 v97 v98
     v99 v100.
    CONSTANT 1 v0 /\ DELT (load,v20) /\ NOT (v20,v19) /\
    AND (v19,load,v18) /\ DEL (done,v17) /\ AND (v18,v17,v16) /\
    OR (v16,v15,v11) /\ DELT (v14,v22) /\ NOT (v22,v21) /\
    AND (v21,v14,v15) /\ MUX (v15,v13,inp,v4) /\
    MUX (v15,v12,v0,v3) /\ DFF (v4,v11,v10) /\ DFF (v3,v11,v9) /\
    DELT (v11,v26) /\ NOT (v26,v25) /\ AND (v25,v11,v24) /\
    NOT (v24,v8) /\ CONSTANT 0 v27 /\ EQ (v4,v27,v23) /\
    DEL (v23,v7) /\ DELT (v8,v31) /\ NOT (v31,v30) /\
    AND (v30,v8,v29) /\ AND (v29,v7,v6) /\ NOT (v7,v28) /\
    AND (v28,v29,v5) /\ DELT (v6,v34) /\ NOT (v34,v33) /\
    AND (v33,v6,v32) /\ NOT (v32,v2) /\ DEL (v10,v1) /\
    DEL (v9,out) /\ DELT (v5,v43) /\ NOT (v43,v42) /\
    AND (v42,v5,v41) /\ DEL (v14,v40) /\ AND (v41,v40,v39) /\
    DELT (v39,v47) /\ NOT (v47,v46) /\ AND (v46,v39,v45) /\
    NOT (v45,v38) /\ CONSTANT 1 v48 /\ SUB (v10,v48,v44) /\
    DEL (v44,v36) /\ CONSTANT 0 v49 /\ DELT (v39,v73) /\
     ...
    AND (v52,v59,v97) /\ AND (v97,v95,v37) /\ DFF (v36,v38,v13) /\
    DFF (v35,v37,v12) /\ AND (v38,v37,v14) /\ DEL (v14,v99) /\
    AND (v14,v99,v98) /\ AND (v2,v8,v100) /\ AND (v100,v98,done)) ==>
  Handshake FACTORIAL (load,inp,done,out) : thm
```

HOL

A pretty-printer generates Verilog from HOL netlists

```
// Definition of module FACT
module FACT (clk,load,inp,done,out);
 input clk,load;
 input [31:0] inp;
 output done;
 output [31:0] out;
 wire clk,done;

 wire [31:0] v0;
 wire [31:0] v1;
 wire [0:0] v2;
 wire [31:0] v3;
 wire [31:0] v4;
 ...

/* CONSTANT (1 :num) (v0 :num -> num) */
CONSTANT   CONSTANT_0 (v0);
  defparam CONSTANT_0.size  = 31;
  defparam CONSTANT_0.value = 1;

/* DtypeT ((clk :num -> bool),(load :num -> bool),(v20 :num -> bool))
DtypeT       DtypeT_0 (clk,load,v20);
...
```

Verilog HDL

Review of proof-producing synthesis from HOL

- From f create imp and prove $\vdash \mathrm{Implements}(f, imp)$
  - f if a functional program in higher order logic
  - 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

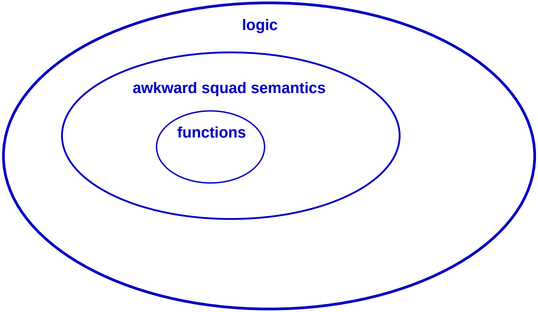
Compile non-function semantics for the awkward squad

- 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

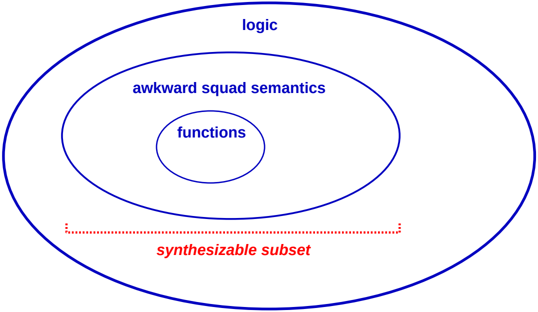
- Hardware pipeline model for interrupts
- Timed temporal logic formulas for real-time
- Bus models for input-output
- Transition systems or sets of traces for concurrency



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 $\leftrightarrow$ HOL $\leftrightarrow$ 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