Compositional Software Model Checking

Dan R. Ghica
Oxford University Computing Laboratory

October 18, 2002
Outline of talk

- program verification issues
- the semantic challenge — programming languages
- the logical challenge — specification logics
- specialised quantifiers — a model-checking friendly logic
Program verification I — inferential

- programming logics:
  - Hoare logic; specification logic (Reynolds); dynamic logic (Harel)

- common features:
  - ‘exogenous’; compositional; sound; excruciating;
  - (partially) automated with proof checkers/assistants
  - can prove (partial) correctness; cannot find bugs
Program verification II — semantic

- temporal logics (many)

- common features:
  - ‘endogenous’; non-compositional; unsound (‘semantic gap’); fun
  - algorithmic semantics
  - fully automated with model checkers (Slam, Bandera)
  - cannot prove correctness; can find bugs
Program verification desiderata

- unify the two approaches: ‘compositional model checking’

- inference rules
  - correctness, compositionality

- algorithmic semantics of program and logic
  - automated model-checking, find counterexamples

- fewer inferential steps
The semantic challenge

- an algorithmic semantics of programming languages with procedures

- program (fragment) code $\Rightarrow$ finite model

- difficult problem
Games semantics

- Hyland & Ong and (independently) Abramsky, Jagadeesan & Malacaria and (independently) Nickau (93–94):
  
  Full abstraction for PCF

- Abramsky & McCusker (96–97):
  
  Full abstraction of Idealised Algol

- many more important results followed

A complex combinatorial account of programming language semantics.
Observation: for certain languages, if higher-order functions and recursion are omitted then much of the game-theoretic formal apparatus can be also omitted

- second-order Idealised Algol can be modeled using regular languages (Ghica & McCusker 2000)

- second-order reduced ML can be modeled using regular languages (Ghica 01)

- third-order Idealised Algol can be modeled using deterministic context-free languages (Ong 02)

A practicable algorithmic model.
The programming language IA

Data sets:
  booleans, bounded integers

Ground types:
  variables, expressions, commands;

Imperative features:
  assignment, branching, iteration, local variables;

Functional features:
  first-order functions (uniformly on all types), call-by-name;
Semantic valuations

\[ [\Gamma \vdash M:\theta] : \text{Term} \rightarrow \text{Regular Language} \]

Helpful notation

\[ [\Gamma \vdash E:\text{int}] = \sum_n q \cdot (\Gamma \vdash E:\text{int})_n \cdot n \]

\[ [\Gamma \vdash C:\text{com}] = \text{run} \cdot (\Gamma \vdash C:\text{com}) \cdot \text{done} \]

\[ [\Gamma \vdash V:\text{var}] = \sum_n \text{read} \cdot (\Gamma \vdash V:\text{var})^\text{read}_n \cdot n \\
+ \sum_n \text{write}(n) \cdot (\Gamma \vdash V:\text{var})^\text{write}_n \cdot \text{ok} \]
**RL interpretation**

Language constants:

\[
\begin{align*}
\text{[skip:com]} &= \text{run} \cdot \text{done} \\
\text{[n:int]} &= q \cdot n \\
\text{[loop:com]} &= \emptyset
\end{align*}
\]

Imperative features:

\[
\begin{align*}
\{C; C'\} &= \{C\} \cdot \{C'\} \\
\{\text{if } B \text{ then } C \text{ else } C'\} &= \{B:\text{bool}\}_{tt} \cdot \{C\} + \{B:\text{bool}\}_{ff} \cdot \{C'\} \\
\{\text{while } B \text{ do } C\} &= (\{B\}_{tt} \cdot \{C\})^* \cdot \{B\}_{ff}
\end{align*}
\]
A simple example

Γ ⊦ while true do C ≡ loop

(while true do C) = ((true) tt · (C))∗ · (true) ff

= (ε · (C))∗ · ∅
= ∅ = (loop)

Because: (true) tt = ε, (true) ff = ∅.
Free identifiers

\[
[c : \text{com} \vdash c : \text{com}] = \text{run} \cdot \text{run}^c \cdot \text{done}^c \cdot \text{done}
\]

\[
[f : \text{com} \rightarrow \text{com} \vdash f : \text{com} \rightarrow \text{com}] \\
= \text{run} \cdot \text{run}^f \cdot (\text{run}^{f_1} \cdot \text{run}^1 \cdot \text{done}^1 \cdot \text{done}^{f_1})^* \cdot \text{done}^f \cdot \text{done}
\]

E.g.

\[
[f(\text{skip})] = \text{run} \cdot \text{run}^f \cdot (\text{run}^{f_1} \cdot \text{done}^{f_1})^* \cdot \text{done}^f \cdot \text{done}
\]
Variables

Assignment:

\[(V := E) = \sum_n (E)_n \cdot (V)_{n}^{\text{write}}\]

Dereferencing (reading):

\[(!V : \text{int})_n = (V : \text{var})_{n}^{\text{read}}\]

\textbf{Obs:} no causal relation between read and write actions:

\[
\text{let } v \text{ be } y := !y + 1; y \text{ in } \cdots
\]
Local (block) variables

Only local variables are guaranteed to be well behaved:

\[ \cdots \text{write}^x(n) \cdot \text{ok}^x \cdots \text{read}^x \cdot n^x \cdots \text{read}^x \cdot n^x \cdots \]

Interpretation of block variables:

\[
(\Gamma \vdash \text{int } x \text{ in } C) = (\Gamma, x: \text{var} \vdash C) \cap \gamma^x \upharpoonright A^x
\]

Where \( \gamma^x = (\sum_n \text{write}^x(n) \cdot \text{ok}^x \cdot (\text{read}^x \cdot n^x)^*)^* \)

- \( A^x \) the alphabet of moves not tagged by \( x \),
- \( \gamma^x = \gamma^x \uplus (A^x)^* \) (shuffle)
- \( \upharpoonright \) is restriction.
An example

\[ c : \text{com} \vdash \text{int } x \text{ in } c \equiv c \]

First proved using possible-worlds-style functor categories.

\[
\langle c : \text{com} \vdash \text{int } x \text{ in } c \rangle = \left( \langle c : \text{com}, x : \text{var} \vdash c \rangle \cap \gamma^x \right) \upharpoonright A^x
\]
\[
= \left( \text{run}^c \cdot \text{done}^c \cap \gamma^x \right) \upharpoonright A^x
\]
\[
= \text{run}^c \cdot \text{done}^c
\]
\[
= \langle c : \text{com} \vdash c \rangle
\]
The semantic gap is bridged.

- algorithmic regular-language based model
- sound and complete (fully abstract)
- decidable (for most properties)
And now for something completely different...

...specification logic.
Common specification idiom

In Hoare-logic

\[ \{ P \} \ C \ \{ Q \} \]

In (Reynolds) spec. logic, with non-local procedures

\[ \forall p : \sigma \bullet S_{\text{proc}}(p) \rightarrow S_{\text{prog}}(p) \]

E.g.

\[ \forall c : \text{comm} \bullet \{ v = v_0 \} \ c \ \{ v = v_0 + 1 \} \rightarrow \{ v = z \} \ c; c \ \{ v = z + 2 \} \]
A problem

Specification logic is undecidable

- boolean, recursion-free, first-order fragment

Questionable suitability for model checking
Equational theory of IA is undecidable

\[ \phi ::= \forall x : \sigma.\phi \mid \exists x : \sigma.\phi \mid \phi \land \phi \mid \phi \lor \phi \mid \phi \rightarrow \phi \mid \neg \phi \mid M \equiv N \]

We can encode polynomials: \( \gamma^{-1} : \mathbb{N}[X] \rightarrow IA \).

\[
\begin{align*}
\gamma^{-1} X & = x : \text{comm} \rightarrow \text{comm} \\
\gamma^{-1} 0 & = \lambda c. \text{skip} \\
\gamma^{-1} \text{succ}(n) & = \lambda c. (\gamma^{-1} n) c; c \\
\gamma^{-1} m + n & = \lambda c. (\gamma^{-1} n) c; (\gamma^{-1} m) c \\
\gamma^{-1} m \times n & = \lambda c. (\gamma^{-1} n)(((\gamma^{-1} m)) c)
\end{align*}
\]
Equational theory of IA is undecidable (cont’d)

Hilbert’s 10th problem corresponds to proving

\[ \exists \vec{x} : \text{comm} \rightarrow \text{comm.} \left[ P_1 \equiv P_2 \right], \quad P_i \in \mathbb{N}[X]. \]

Spec. logic of IA is also undecidable

\[ \left( \forall p : \text{assert.} \{ \text{true} \} c_1 \{p\} \leftrightarrow \{ \text{true} \} c_2 \{p\} \right) \leftrightarrow c_1 \equiv c_2. \]
A solution

Common idiom:

\[ \forall p : \sigma \bullet S_{\text{proc}}(p) \rightarrow S_{\text{prog}}(p) \]

Relativised quantifiers (syntactic):

\[ \forall p : S_{\text{proc}} \bullet S_{\text{prog}}(p) \]

Specialised quantifiers (semantic):

\[ \Psi_p : S_{\text{proc}} \bullet S_{\text{prog}}(p) \]
Example: “effects quantifier”

\[ \Psi_c : \{v = v_0\} \cdot \{v = v_0 + 1\} \cdot \{v = z\} \cdot \{v = z + 2\} \]

All the variables above are global, distinct.
Another useful quantifier: “stability quantifier”

CBN+computational side-effects breaks arithmetic: \( x \neq x \)

Context: let \( x \) be \( v := !v + 1; !v \) in \( \cdots \)

Obs: this is in general a difficult problem (Boehm 82)

\[ \nabla x : \sigma.S \]

E.g. \( \nabla x : \exp.x = x \) is always true.

Semantics: similar to block variables \( (\cdots \cap \gamma^x_\sigma) \upharpoonright A^x \)
A more generalised stability quantifier

\[ \nabla x / \vec{y} : \sigma . S \]

Interpretation: \( x \) is stable but the actions of identifiers in \( \vec{y} \) may change its value.

E.g.

\[ \nabla y . \nabla x / y \bullet x + x = 2 \times x \] is true
\[ \nabla y . \nabla x / y \bullet x + y = y + x \] is not necessarily true
Our example, this time formally

\[ \nabla v/c. \Psi c : \nabla v_0. \{ v = v_0 \} c \{ v = v_0 + 1 \} \bullet \nabla z. \{ v = z \} c; c \{ v = z + 2 \} \]

- no need to restrict to global variables

- more abstract formulations

\[ \nabla e/c. \Psi c : \nabla e_0. \{ e = e_0 \} c \{ e = e_0 + 1 \} \bullet \nabla z. \{ e = z \} c; c \{ e = z + 2 \} \]
Why does it work? — a discussion

• no true universal quantifier in the logic
  – no quantification over languages
  – (no quantification over opponent behaviours)

• specialised quantifiers are tamer
  – regular-language interpretation
  – (encode a strategy for the opponent)
A bit of quantifier theory

introduced by Mostowski (57) and extended by Lindstroem (66);
called “generalised” quantifiers;
extend expressiveness without increasing order

- “for uncountably many”
- “for all connected graphs”

applications to descriptive complexity theory
What is a generalised quantifier?

any collection of structures closed under isomorphism

type of a generalised quantifier \((n_1, n_2, \ldots, n_k)\)

\(n_i\) identifiers are bound in the \(i^{th}\) formula

e.g. of type \((1, 1)\) quantifier:

\[ \Psi x, y : \text{there are as many } x \text{ with } P(x) \text{ as there are } y \text{ with } Q(y) \]

strict hierarchy (Hella et.al. 96)

\((1) < (1, 1) < \cdots < (2) < (2, 1) < (2, 1, 1) \cdots < (2, 2) < \cdots < (3) \cdots \)
Proof theory of generalised quantifiers

substructural logics—substitution subject to extra conditions

\[
\Psi \bar{x}_1 : P_1 \ldots \bar{x}_k : P_k \bullet S \quad P_i(\bar{M}_i) \\
M_i \# S, \ M_i \# M_j, \ i \neq j
\]

\[
S[\bar{x}_i / \bar{M}_i]
\]

we call \# non-interference

work done by Alechina & Lambalgen (96)
Example

\[ \Psi_c : \{ v = v_0 \} c \{ v = v_0 + 1 \} \bullet \{ v = z \} c; c \{ v = z + 2 \} \]
\[ \{ v = v_0 \} v := v + 1 \{ v = v_0 + 1 \} \]
\[ \{ v = z \} v = v + 1; v = v + 1; \{ v = z + 2 \} \]

Or

\[ \Psi_c : \{ v = v_0 \} c \{ v = v_0 + 1 \} \bullet \{ v = z \} c; c \{ v = z + 2 \} \]
\[ \{ v = v_0 \} v := v + 1; k := k + 1 \{ v = v_0 + 1 \} \]
\[ \{ v = z \} v = v + 1; k := k + 1; v = v + 1; k := k + 1 \{ v = z + 2 \} \]

But not

\[ \Psi_c : \{ v = v_0 \} c \{ v = v_0 + 1 \} \bullet \{ v = z \} c; c \{ v = z + 2 \} \]
\[ \{ v = v_0 \} v := v + 1; z := z + 1 \{ v = v_0 + 1 \} \]
\[ \{ v = z \} v = v + 1; z := z + 1; v = v + 1; z := z + 1 \{ v = z + 2 \} \]
Non-interference

$C \not\in S$

$C$ does not write to the ‘protected’ variables of $S$ and vice versa.
Higher-type quantifiers and substitution

In the previous example we actually have a type (1,1) quantifier

\[
\left( \Psi_c : \nabla e_0. \{ e = e_0 \} \ x \{ e = e_0 + 1 \} \right) \bullet \nabla z. \{ e = z \} \ y \{ e = z + 2 \}
\]

Elimination must be \textit{simultaneous}

\[
\nabla v. \nabla (!v)
\]
\[
\nabla v. \nabla e_0. \{ !v = e_0 \} \ y := !v + 1 \ { !v = e_0 + 1 }\]

\[
\nabla v. \nabla z. \{ !v = z \} \ y := !v + 1; \ y := !v + 1 \ { !v = z + 2 }\]
Highlights of the logical system

- effects and stability quantifiers for ground and first order types

- quantifiers of type \((1,1,\cdots,1)\)

- regular-language semantics, decidable

- quantifiers discharged through substitution

- also, Hoare axioms
"algebraic" specifications involve quantifiers of higher type:

\[
\left( \nabla x, \text{do, undo} \\nabla y, \{x = y\}, \text{do; undo \{x = y\}} \\nabla y, \{x = y\}, \text{undo; do \{x = y\}} \right) \cdot \ldots .
\]

has type (1, 2), does not (?) have a RL semantics.
Conclusion

• RL semantics, inference — compositional model checking

• non-interference conditions — syntactic or semantic

• challenging programming language — semantic and logic

• not very complicated (?) — avoid “low level” axioms

• some drawbacks

• no verification tool yet
Related work

Interface automata, de Alfaro & Henzinger (2001)

- emphasis on the automata model
- the discharge of the interface — “refinement”