

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.

Regular-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

$\llbracket \Gamma \vdash M:\theta \rrbracket : \text{Term} \rightarrow \text{Regular Language}$

Helpful notation

$$\llbracket \Gamma \vdash E:\text{int} \rrbracket = \sum_n q \cdot (\Gamma \vdash E:\text{int})_n \cdot n$$

$$\llbracket \Gamma \vdash C:\text{com} \rrbracket = \text{run} \cdot (\Gamma \vdash C:\text{com}) \cdot \text{done}$$

$$\begin{aligned} \llbracket \Gamma \vdash V:\text{var} \rrbracket &= \sum_n \text{read} \cdot (\Gamma \vdash V:\text{var})_n^{\text{read}} \cdot n \\ &\quad + \sum_n \text{write}(n) \cdot (\Gamma \vdash V:\text{var})_n^{\text{write}} \cdot \text{ok} \end{aligned}$$

RL interpretation

Language constants:

$$\llbracket \text{skip:com} \rrbracket = \text{run} \cdot \text{done} \quad \llbracket n:\text{int} \rrbracket = q \cdot n \quad \llbracket \text{loop:com} \rrbracket = \emptyset$$

Imperative features:

$$\llbracket C; C' \rrbracket = \llbracket C \rrbracket \cdot \llbracket C' \rrbracket$$

$$\llbracket \text{if } B \text{ then } C \text{ else } C' \rrbracket = \llbracket B:\text{bool} \rrbracket_{tt} \cdot \llbracket C \rrbracket + \llbracket B:\text{bool} \rrbracket_{ff} \cdot \llbracket C' \rrbracket$$

$$\llbracket \text{while } B \text{ do } C \rrbracket = \left(\llbracket B \rrbracket_{tt} \cdot \llbracket C \rrbracket \right)^* \cdot \llbracket B \rrbracket_{ff}$$

A simple example

$\Gamma \vdash \text{while true do } C \equiv \text{loop}$

$$\begin{aligned} \llbracket \text{while true do } C \rrbracket &= \left(\llbracket \text{true} \rrbracket_{tt} \cdot \llbracket C \rrbracket \right)^* \cdot \llbracket \text{true} \rrbracket_{ff} \\ &= \left(\epsilon \cdot \llbracket C \rrbracket \right)^* \cdot \emptyset \\ &= \emptyset = \llbracket \text{loop} \rrbracket \end{aligned}$$

Because: $\llbracket \text{true} \rrbracket_{tt} = \epsilon$, $\llbracket \text{true} \rrbracket_{ff} = \emptyset$.

Free identifiers

$$\llbracket c:\text{com} \vdash c:\text{com} \rrbracket = \text{run} \cdot \text{run}^c \cdot \text{done}^c \cdot \text{done}$$

$$\begin{aligned} \llbracket f:\text{com} \rightarrow \text{com} \vdash f:\text{com} \rightarrow \text{com} \rrbracket \\ = \text{run} \cdot \text{run}^f \cdot (\text{run}^{f1} \cdot \text{run}^1 \cdot \text{done}^1 \cdot \text{done}^{f1})^* \cdot \text{done}^f \cdot \text{done} \end{aligned}$$

E.g.

$$\llbracket f(\text{skip}) \rrbracket = \text{run} \cdot \text{run}^f \cdot (\text{run}^{f1} \cdot \text{done}^{f1})^* \cdot \text{done}^f \cdot \text{done}$$

Variables

Assignment:

$$\langle V := E \rangle = \sum_n \langle E \rangle_n \cdot \langle V \rangle_n^{\text{write}}$$

Dereferencing (reading):

$$\langle !V : \text{int} \rangle_n = \langle V : \text{var} \rangle_n^{\text{read}}$$

Obs: no causal relation between read and write actions:

let v **be** $y := !y + 1; y$ **in** \dots

Local (block) variables

Only local variables are guaranteed to be well behaved:

$$\dots write^x(n) \cdot ok^x \dots read^x \cdot n^x \dots read^x \cdot n^x \dots$$

Interpretation of block variables:

$$\llbracket \Gamma \vdash \text{int } x \text{ in } C \rrbracket = \left(\llbracket \Gamma, x:\text{var} \vdash C \rrbracket \cap \widetilde{\gamma}^x \right) \upharpoonright \mathcal{A}^x$$

Where $\gamma^x = \left(\sum_n write^x(n) \cdot ok^x \cdot (read^x \cdot n^x)^* \right)^*$

- \mathcal{A}^x the alphabet of moves not tagged by x ,
- $\widetilde{\gamma}^x = \gamma^x \amalg (\mathcal{A}^x)^*$ (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.

$$\begin{aligned} (c:\text{com} \vdash \text{int } x \text{ in } c) &= ((c:\text{com}, x:\text{var} \vdash c) \cap \widetilde{\gamma}^x) \uparrow \mathcal{A}^x \\ &= (run^c \cdot done^c \cap \widetilde{\gamma}^x) \uparrow \mathcal{A}^x \\ &= run^c \cdot done^c \\ &= (c:\text{com} \vdash c) \end{aligned}$$

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

$$\phi ::= \forall x : \sigma. \phi \mid \exists x : \sigma. \phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \neg \phi \mid M \equiv N$$

We can encode polynomials: $\ulcorner - \urcorner : \mathbb{N}[X] \rightarrow IA$.

$$\ulcorner X \urcorner = x : \text{comm} \rightarrow \text{comm}$$

$$\ulcorner 0 \urcorner = \lambda c. \text{skip}$$

$$\ulcorner \text{succ}(n) \urcorner = \lambda c. (\ulcorner n \urcorner)c; c$$

$$\ulcorner m + n \urcorner = \lambda c. (\ulcorner n \urcorner)c; (\ulcorner m \urcorner)c$$

$$\ulcorner m \times n \urcorner = \lambda c. (\ulcorner n \urcorner)((\ulcorner m \urcorner)c)$$

Equational theory of IA is undecidable (cont'd)

Hilbert's 10th problem corresponds to proving

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

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”

$$\Psi c : \underbrace{\{v = v_0\} c \{v = v_0 + 1\}}_{S_{\text{proc}}} \bullet \underbrace{\{v = z\} c; c \{v = z + 2\}}_{S_{\text{prog}}}$$

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 \dots

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

$$\nabla x : \sigma.S$$

E.g. $\nabla x : \text{exp}.x = x$ is always true.

Semantics: similar to block variables $(\dots \cap \gamma_{\sigma}^x) \upharpoonright \mathcal{A}^x$

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.

$$\nabla y. \nabla x / y \bullet x + x = 2 \times x \quad \text{is true}$$

$$\nabla y. \nabla x / y \bullet x + y = y + x \quad \text{is not necessarily true}$$

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, \dots, n_k)

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

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

$\Psi_{x,y}$: there are as many x with $P(x)$ as there are y with $Q(y)$

strict hierarchy (Hella et.al. 96)

$(1) < (1, 1) < \dots < (2) < (2, 1) < (2, 1, 1) \dots < (2, 2) < \dots < (3) \dots$

Proof theory of generalised quantifiers

substructural logics—substitution subject to extra conditions

$$\frac{\Psi \vec{x}_1 : P_1 \dots \vec{x}_k : P_k \bullet S \quad P_i(\vec{M}_i)}{S[\vec{x}_i / \vec{M}_i]} \quad M_i \# S, M_i \# M_j, i \neq j$$

we call $\#$ non-interference

work done by Alechina & Lambalgen (96)

Example

$$\frac{\Psi c : \{v = v_0\} c \{v = v_0 + 1\} \bullet \{v = z\} c; c \{v = z + 2\} \quad \{v = v_0\} v := v + 1 \{v = v_0 + 1\}}{\{v = z\} v = v + 1; v = v + 1; \{v = z + 2\}}$$

Or

$$\frac{\Psi c : \{v = v_0\} c \{v = v_0 + 1\} \bullet \{v = z\} c; c \{v = z + 2\} \quad \{v = v_0\} v := v + 1; k := k + 1 \{v = v_0 + 1\}}{\{v = z\} v = v + 1; k := k + 1; v = v + 1; k := k + 1 \{v = z + 2\}}$$

But not

$$\frac{\Psi c : \{v = v_0\} c \{v = v_0 + 1\} \bullet \{v = z\} c; c \{v = z + 2\} \quad \{v = v_0\} v := v + 1; z := z + 1 \{v = v_0 + 1\}}{\{v = z\} v = v + 1; z := z + 1; v = v + 1; z := z + 1 \{v = z + 2\}}$$

Non-interference

$C \# S$

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

$$\left(\Psi_c : \nabla e_0. \{e = e_0\} \overset{\nabla e/c}{c} \{e = e_0 + 1\} \right) \bullet \nabla z. \{e = z\} c; c \{e = z + 2\}$$

Elimination must be *simultaneous*

$$\nabla v. \nabla (!v)$$

$$\nabla v. \nabla e_0. \{!v = e_0\} v := !v + 1 \{!v = e_0 + 1\}$$

$$\nabla v. \nabla z. \{!v = z\} v := !v + 1; v := !v + 1 \{!v = z + 2\}$$

Highlights of the logical system

- effects and stability quantifiers for ground and first order types
- quantifiers of type $(1, 1, \dots, 1)$
- regular-language semantics, decidable
- quantifiers discharged through substitution
- also, Hoare axioms

Drawback

“algebraic” specifications involve quantifiers of higher type:

$$\left(\begin{array}{c} \nabla x / do, undo \\ \nabla y. \{x = y\} do; undo \{x = y\} \\ \nabla y. \{x = y\} undo; do \{x = y\} \end{array} \right) \bullet \dots$$

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”