Program Verification and the Church-Rosser Theorem

Peter Vincent Homeier

National Security Agency

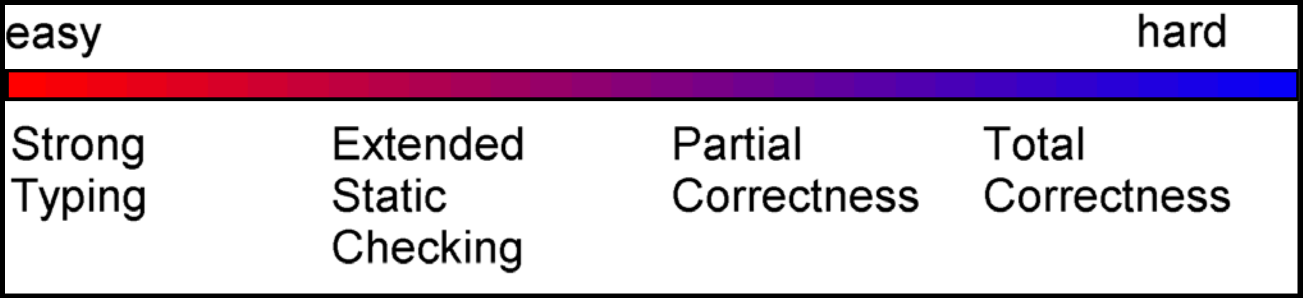
homeier@saul.cis.upenn.edu

The Need for Practical Verification

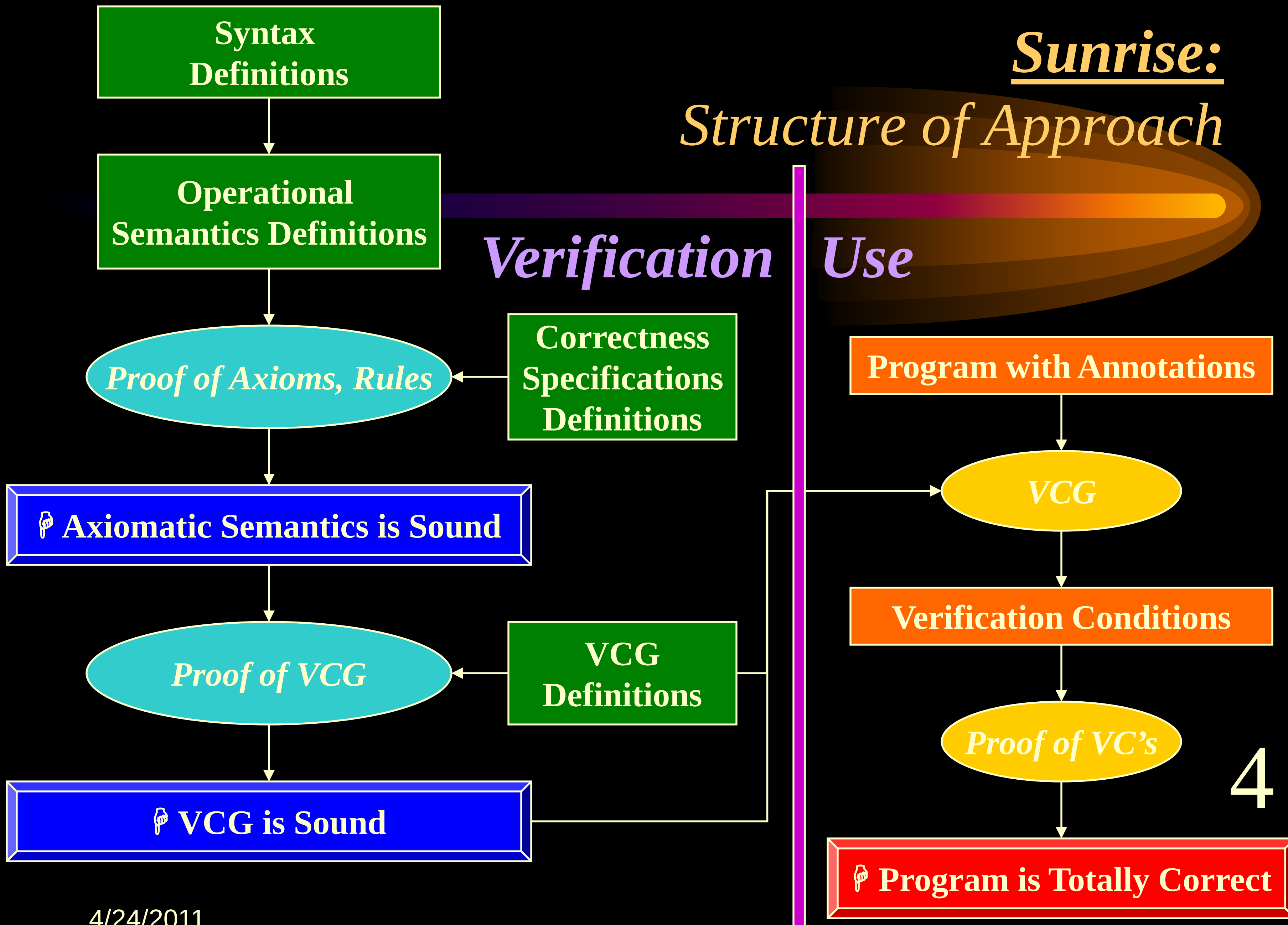
- Reliability is critical for some applications
- For qualitatively superior reliability, verification is necessary
- For credible proofs, mechanical verification is necessary
- Goal is a tool to support human construction of software designs and code that are proven consistent with specs
- Desired result is code verified to perform as specified

Prior Related Work

- **Sunrise**, total correctness for small imperative language, like subset of Pascal including mutually rec. procedures
- **Bali**, formalizes aspects of Java in Isabelle/HOL, including dynamic binding, exceptions, side-effects
- **Extended Static Checking (ESC)**, super-lint for Java, checks array bounds, nil dereference, synchronization



Sunrise:
Structure of Approach



Process and Advantages of Verification

- Programmer iteratively writes design/code with annotations about intended behavior; reveals flaws
- Tool automatically resolves most of verification, resorting to programmer for remaining issues
- Many common programming errors prevented absolutely
- Verification implies significantly higher reliability
- Eases but does not replace testing; Only part of a wider high-confidence software engineering methodology

Foundations of Semantics of Languages

- Most such previous VCG tools were not formally verified
  - … hence proofs of programs were suspect!
- Need formal proof of soundness of VCG tool
- … based on *formal semantics* of the programming language
- Lambda Calculus is a prototypical programming language
- A laboratory for examining general language issues
- … including the nontrivial Church-Rosser property

Prior Proofs of Church-Rosser Theorem

- Shankar, 1988, Boyer-Moore (nqthm), name-carrying syntax
- Huet, 1994, Coq, de Bruijn syntax
- Rasmussen, 1995, Isabelle-ZF, de Bruijn syntax
- Vestergaard/Brotherston, 2001, Isabelle-HOL, name-carrying syntax

Raw Lambda Calculus Syntax

λ-calculus syntax:

variables (var): $x, y, z, \ldots$

terms (term$_1$): Λ_1 ::= var | $\Lambda_1\ \Lambda_1$ | λvar.Λ_1

(variable, application, abstraction)

substitutions (subst$_1$): Σ_1 ::= [ ] | (var := Λ_1) :: Σ_1

(nil, cons of (var, term) pair) - a *simultaneous* substitution

Typical meta-variables of types: term: t, u, M, N, L subst: s var set: r

```
val _ = Hol_datatype
        ` term1 = Var1 of var
                | App1 of term1 => term1
                | Lam1 of var => term1 ` ;
```

Hol98 automatically proves term 1) structural induction, 2) function existence, 3) cases, 4) constructors distinctiveness, and 5) constructors one-to-one

Functions on Raw Lambda Calculus Syntax

Functions on λ-calculus syntax:

HEIGHT$_1$:	$\Lambda \rightarrow$ num	Height of term, var is 0, else 1+components
FV$_1$:	$\Lambda \rightarrow$ var set	Set of free variables of term
_❐$_1{}^v$_:	var $\rightarrow \Sigma \rightarrow \Lambda$	Application of a substitution to a variable
_❐$_1$_:	$\Lambda \rightarrow \Sigma \rightarrow \Lambda$	***Proper*** application of a substitution to a term

HEIGHT$_1$ and FV$_1$ are defined by primitive recursion on the structure of terms

❐$_1{}^v$ is defined by list recursion on the structure of the substitution

❐$_1$ is defined by primitive recursion on the structure of terms, making use of the simultaneous substitution to add new bindings to properly avoid capture.

Substitution

Definition of substitution: (Complete)

$x \,❐_1\, s \quad = \quad x \,❐_1^{v}\, s$

$(t\;\; u) \,❐_1\, s \quad = \quad (t \,❐_1^{v}\, s)\; (u \,❐_1^{v}\, s)$

$(\lambda x.\, t) \,❐_1\, s \quad = \quad$ **let** x' = variant x (FVsubst$_1$ s (FV$_1$ $t - \{x\}$)) **in**

$\qquad \lambda x'.\, (t \,❐_1\, ((x := x') :: s))$

where

FVsubst$_1$ $s\ r$ $= \bigcup$ (image (FV$_1$ $\square$ SUB$_1$ s) r)

SUB$_1$ $s\ x$ $= \ x \,❐_1^{v}\, s$

"Naïve" substitution is easy and simple but NOT CORRECT:

$(\lambda x.\, t) \,❐_1\, s \quad = \quad \lambda x.\, (t \,❐_1\, s)$

Constructors One-to-One Property

Almost right, but constructors one-to-one property says that

$$(\lambda x_1.\ t_1 = \lambda x_2.\ t_2) \Leftrightarrow (x_1 = x_2) \wedge (t_1 = t_2)$$

But we want, for example, $\lambda x.\ x = \lambda y.\ y$. Just which name is used for the variable should be immaterial, as long as names are changed consistently.

This one-to-one property is too discriminating. We want to create a variant of this calculus to blur such distinctions.

The exact blurring we wish is called *alpha-equivalence*.

Alpha-Equivalence

- Church represented as *semantic reduction*: $t \rightarrow_{\alpha} t'$
- More modern approach (Barendreght, Abadi/Cardelli, …) is to *identify* equivalent terms at *syntactic* level
- Alpha-equivalence: relation on terms; e.g., $\lambda x.\ x \equiv_{\alpha} \lambda y.\ y$.
- Design issue: How to define $\equiv_{\alpha}$?
  - Others used substitution ($\square_1$); is it deceptively complex?
  - We used *contextual alpha-equivalence*, where the contexts are lists of variables denoting bindings present

Real Lambda Calculus

- Real lambda calculus formed as quotient of raw lambda calculus by alpha-equivalence:

$$\Lambda \;=\; \Lambda_1 \,/\equiv_\alpha$$

- New type “term” made by new HOL package for quotients
- Produces two mapping functions between term and term1:

$$\lfloor \_ \rfloor : \Lambda_1 \to \Lambda \qquad\qquad \lceil \_ \rceil : \Lambda \to \Lambda_1$$

✌$a.\ \lfloor \lceil a \rceil \rfloor = a \qquad \wedge \qquad$ ✌$r\ r'.\ \ r\ \equiv_\alpha r' \ \Leftrightarrow\ (\lfloor r \rfloor = \lfloor r' \rfloor)$

- Term constructor functions redefined in Λ using map fns

E.g., Lam $x\ t = \lfloor \mathrm{Lam}_1\ x\ \lceil t \rceil \rfloor$, which is $\lambda x.\ t = \lfloor \lambda x. \lceil t \rceil \rfloor$

Recreating Function Definitions in the Real Lambda Calculus

- Functions are defined first in Λ_1 and then recreated in Λ
- BUT, not *every* function definable in Λ_1 can be recreated!
- Functions must respect alpha-equivalence, e.g.,

$$t_1 \equiv_\alpha t_2 \;\Rightarrow\; \mathsf{FV}_1\, t_1 = \mathsf{FV}_1\, t_2$$

$$t_1 \equiv_\alpha t_2 \;\wedge\; s_1 \equiv_\alpha^{\text{subst}} s_2 \;\Rightarrow\; (t_1 \square_1 s_1) \equiv_\alpha (t_2 \square_1 s_2)$$

- 1) Prove function respects alpha-equivalence (arb. complex)
- 2) Define new function using $\lfloor\_\rfloor$ and $\lceil\_\rceil$
- 3) Prove as theorem in Λ the same form as definition in Λ_1

Recreated Properties in the Real Lambda Calculus

- Now we have the one-to-one property

$(\lambda x_1.t_1 = \lambda x_2.t_2) \Leftrightarrow (t_1$ ❐ $[x_1 := x_2] = t_2) \wedge (t_2$ ❐ $[x_2 := x_1] = t_1)$

- All other properties and definitions of Λ_1 are recreated in Λ, *except* for function existence
- More general term height induction principle:

☞ ✌*P*. (✌*x*. *P x*) ∧
(✌*t u*. *P t* ∧ *P u* ⇒ *P* (*t u*)) ∧
(✌*t*. (✌*t'*. HEIGHT *t* = HEIGHT *t'* ⇒ *P t'*) ⇒ ✌*x*. *P* (λ*x*. *t*))
⇒
(✌*t*. *P t*)

Barendregt Variable Convention (BVC)

- Barendregt's *Lambda Calculus: It's Syntax and Semantics*
- The BVC states that in any proof, one can assume that all bound variables are different from all free variables
- Then substitution is simple (naïve), and proofs are elegant
- Controversial; some have suggested the BVC is incomplete
- We have found a mechanization within the security of HOL that (partially) justifies the BVC —
  A new HOL tactic to shift abstractions away from capture, used along with height-based induction

Semantics of Reduction in Lambda Calculus

- Define β as relation on terms such that for all $M, N \in \Lambda$,

$$\beta\ ((\lambda x.\ M)\ N)\ \ (M \ \square\ [x := N])$$

- A relation R on Λ is *compatible* (with the operations) if for all $M, M', Z \in \Lambda$, $x \in$ var,

$$R\ M\ M' \Rightarrow R(Z\ M)(Z\ M') \wedge R(M\ Z)(M'\ Z) \wedge R(\lambda x.M)(\lambda x.M')$$

- Given relation R, R induces reduction relations:

- $\to_R$ one step R-reduction compatible closure of R
- $\twoheadrightarrow_R$ R-reduction reflexive, transitive closure of $\to_R$
- $=_R$ R-equality equivalence relation generated by $\twoheadrightarrow_R$

Diamond Property and Church-Rosser Property

- ⇘ satisfies diamond property (⇘ ☺ ◊) if
  $\forall M\ M_1\ M_2.\ M ⇘ M_1 \wedge M ⇘ M_2 \Rightarrow\ \exists M_3.\ M_1 ⇘ M_3 \wedge M_2 ⇘ M_3$

$$\begin{tikzcd}
 & M \arrow[dl] \arrow[dr] & \\
M_1 \arrow[dr, dashed] & & M_2 \arrow[dl, dashed] \\
 & M_3 &
\end{tikzcd}$$

- *R* is Church-Rosser if ⇗$_R$ ☺ ◊ ; want to prove ⇗$_\beta$ ☺ ◊

The Church-Rosser Theorem

- Original by Church-Rosser (1936); Schroer (1965) 627 pgs
- Greatly simplified proof found by Martin-Löf (1972), based on ideas of Tait
- Elegant presentation by Barendregt (1981) using the BVC
- Define parallel reduction (☆) inductively by the rules

$$\frac{}{M \mathbin{☆} M} \qquad\qquad \frac{M \mathbin{☆} M' \quad , \quad N \mathbin{☆} N'}{M\ N \mathbin{☆} M'\ N'}$$

$$\frac{M \mathbin{☆} M'}{\lambda x.\ M \mathbin{☆} \lambda x.\ M'} \qquad\qquad \frac{M \mathbin{☆} M' \quad , \quad N \mathbin{☆} N' \quad .}{(\lambda x.\ M)\ N \mathbin{☆} M' ❒ [x := N']}$$

Proof of the Church-Rosser Theorem

- **Theorem:** For all relations ⬂, ⬂ ☺ ◊ $\Rightarrow$ ⬂* ☺ ◊
- **Theorem:** ☆ satisfies the diamond property (☆ ☺ ◊)
- **Theorem:** ⬀$_{\beta}$ is the transitive closure of ☆ (⬀$_{\beta}$ = ☆*)

- **Theorem:** β is Church-Rosser.

  Proof: by definition of Church-Rosser and above theorems

HOL Proof of the Church-Rosser Theorem

- 6 main HOL theories (+ 2 auxiliary)
- 3 new types
- 73 new definitions
- 302 theorems proved
- 0 new axioms (secure, conservative extension of HOL)
- 22,252 lines of Standard ML code (including comments)

All theory scripts and associated code, including the new quotient library and mutual recursion tools, are available at

http://www.cis.upenn.edu/~hol/lamcr/

Conclusions

- Separation of concerns is simpler: alpha-equivalence and beta-reduction analyzed in two distinct layers.
- Creating the real lambda calculus as a quotient relied on the proof that substitution respected alpha-equivalence. This proof for a complete substitution function is new.
- We have justified the controversial BVC for this CR proof.
- As the lambda calculus is an archetype of programming languages, this proof is a prototype for general foundations.
- *Soli Deo Gloria.*