Program Equivalence from Theory to Practice

13th Panhellenic Logic Symposium

Vasileios Koutavas

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

lero
THE IRISH SOFTWARE RESEARCH CENTRE

Science Foundation Ireland sfi For what's next

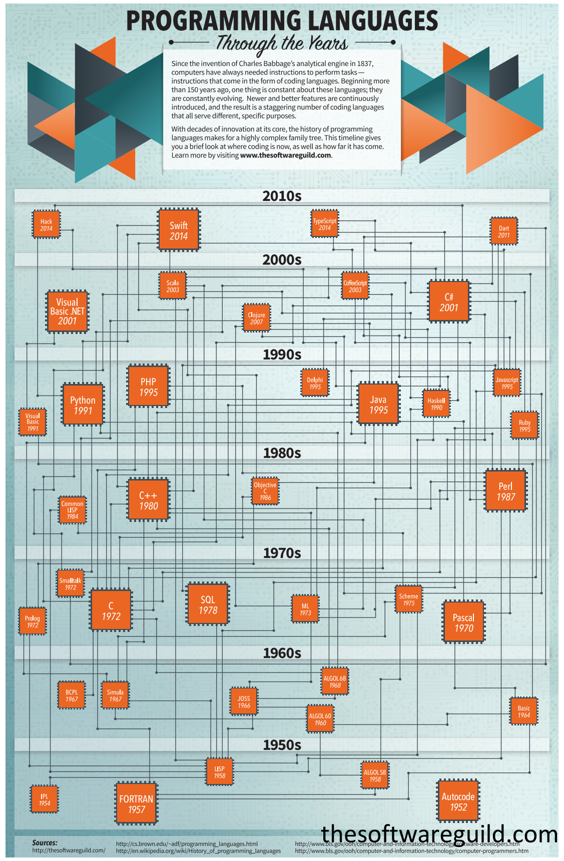
Program Equivalence

HOW DID IT COME ABOUT?

IS IT USEFUL IN PRACTICE?

WHAT ARE THE CHALLENGES?

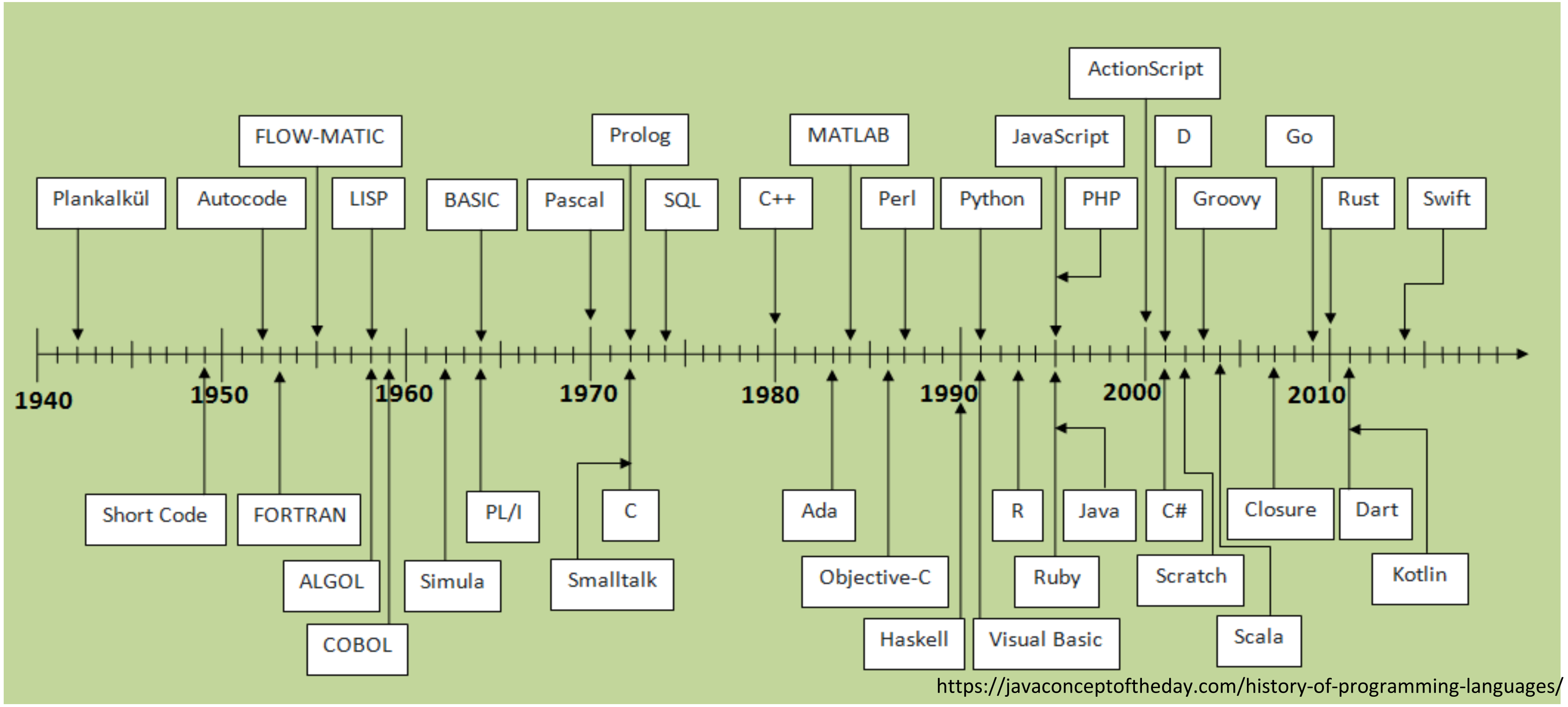
AN APPROACH TO AUTOMATION


PROGRAMMING LANGUAGES
Through the Years

Since the invention of Charles Babbage's analytical engine in 1837, computers have always needed instructions to perform tasks — instructions that come in the form of coding languages. Beginning more than 150 years ago, one thing is constant about these languages; they are constantly evolving. Newer and better features are continuously introduced, and the result is a staggering number of coding languages that all serve different, specific purposes.

With decades of innovation at its core, the history of programming languages makes for a highly complex family tree. This timeline gives you a brief look at where coding is now, as well as how far it has come. Learn more by visiting **www.thesoftwareguild.com**.

thesoftwareguild.com



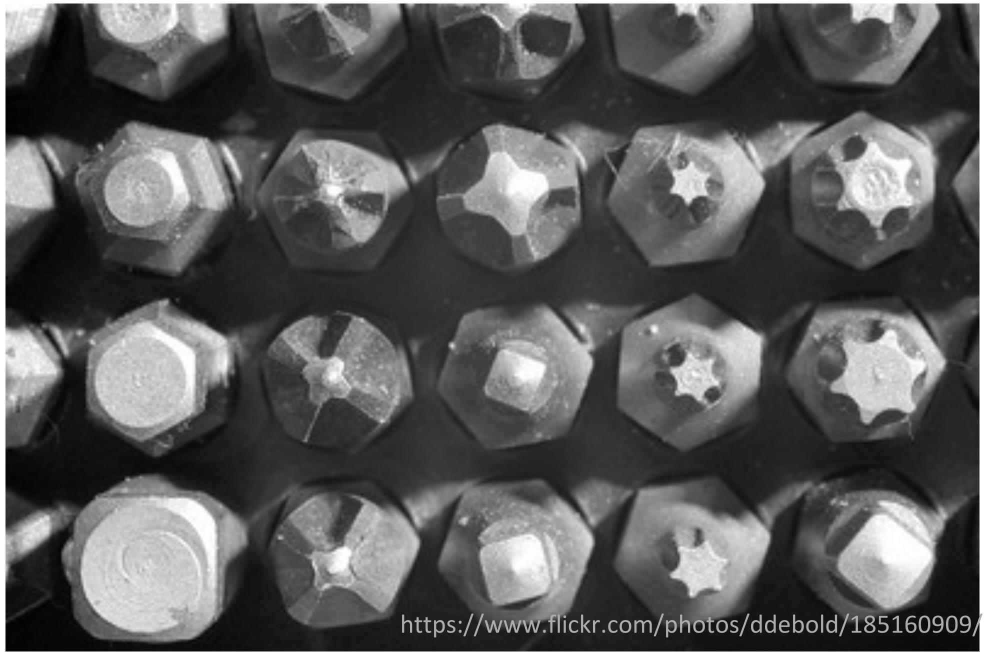
https://javaconceptoftheday.com/history-of-programming-languages/

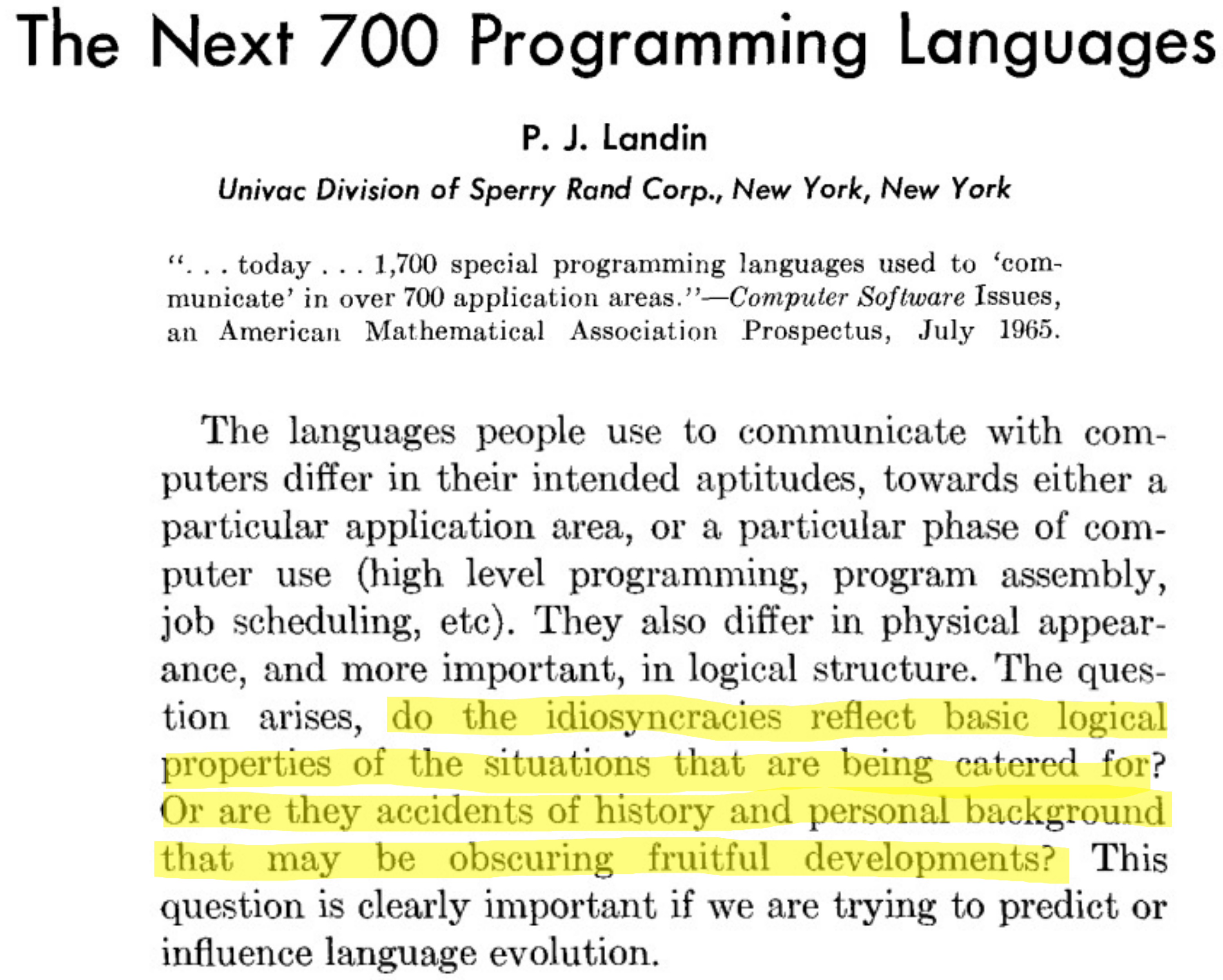
hopl.info

The right tool for the job

The difficulty of programming has become the main difficulty in the use of machines ... To make it easy, one must make coding comprehensible. This may only be done by improving the notation of programming. [Glennie 1952 – Autocode]

FORTRAN did not really grow out of some brainstorm about the beauty of programming in mathematical notation; instead, it began with recognition of a basic problem of economics: programming and debugging costs already exceeded the cost of running a program, and as computers became cheaper this imbalance would become more and more intolerable. This prosaic economic insight, plus experience with the drudgery of coding, plus an unusually lazy nature led to my continuing interest in making programming easier. [Backus - Fortran 1953]





The Next 700 Programming Languages

P. J. Landin

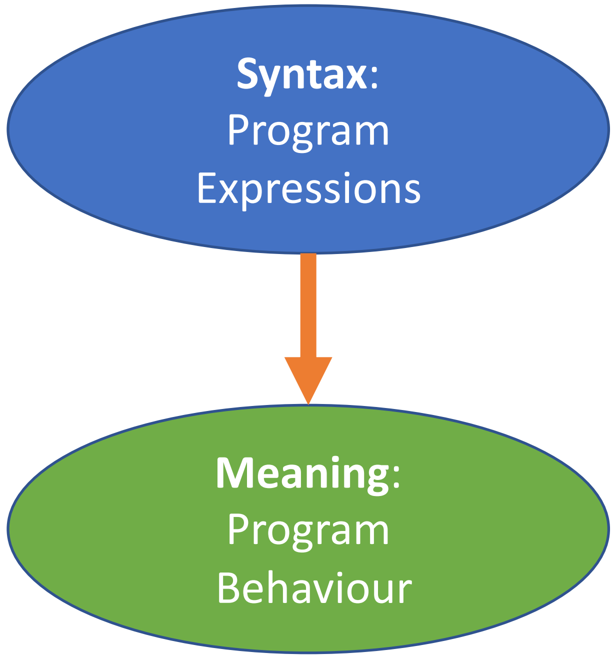
Univac Division of Sperry Rand Corp., New York, New York

". . . today . . . 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software* Issues, an American Mathematical Association Prospectus, July 1965.

The languages people use to communicate with computers differ in their intended aptitudes, towards either a particular application area, or a particular phase of computer use (high level programming, program assembly, job scheduling, etc). They also differ in physical appearance, and more important, in logical structure. The question arises, do the idiosyncracies reflect basic logical properties of the situations that are being catered for? Or are they accidents of history and personal background that may be obscuring fruitful developments? This question is clearly important if we are trying to predict or influence language evolution.

[Landin, P. J. (1966)]

Programming Language Semantics



- Create a **logical formalism** able to express ”program behaviour”
- Map program expressions of a PL into this formalism

[Morris (1946). *Signs, Language, and Behavior*]

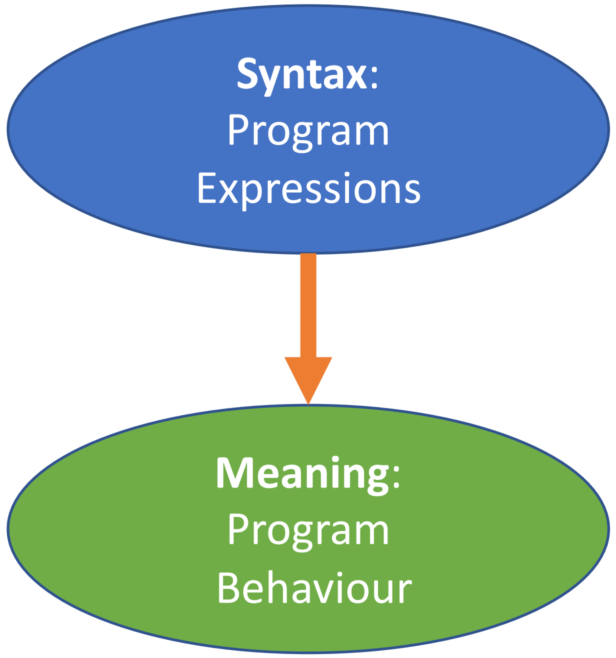
[Bakus (1959). ALGOL 58]

[Dijkstra (1961). *On the Design of Machine Independent Programming Languages]*

[Landin (1966). *A λ-Calculus Approach*]

[Plotkin (1977). *LCF considered as a programming language*]

Programming Language Semantics



Motivation:

- Improve design of practical languages
  eg: [Wilkes in (Gorn 1964)], [Backus 1964]

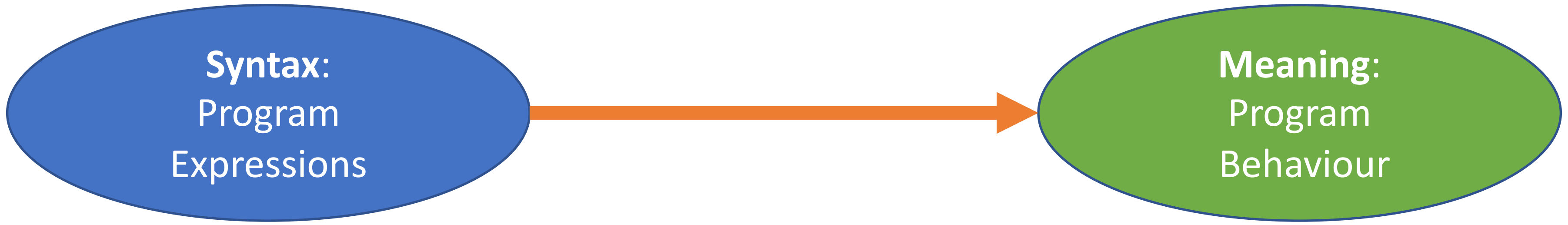
- Make sure compilers of high-level languages are correct
  eg: [Perlis in (Gorn 1964)], [Milne and Strachey 1976]

- Formal language mathematics make computing theoretically possible
  eg: [Mahoney 1988], [Milner 1993]

- Enable mathematical proofs about programs
  eg: [McCarthy 1963], [Scott 1994]

[Astarte (2019). *Formalising Meaning: a History of Programming Language Semantics*]
[Perlis (1981) and Naur (1981). *History of Programming Languages*]
[Alberts (2014). *Annals of the History of Computing.*]

Computing Science = Engineering + Mathematics
Building + Understanding



Engineering Programming Languages

- PL feature combinations
- Expressivity
- Conciseness
- Readability
- Modularity
- Linguistic style

…

Programming Language Semantics

- PL models
  - Syntax → Mathematical behaviour
- Properties
  - Functional correctness
  - Safety, Liveness, Fairness
  - Security
  - Program Equivalence
  - …
- Proof Techniques
- Verification Techniques

Emergence of Program Equivalence (1/3)

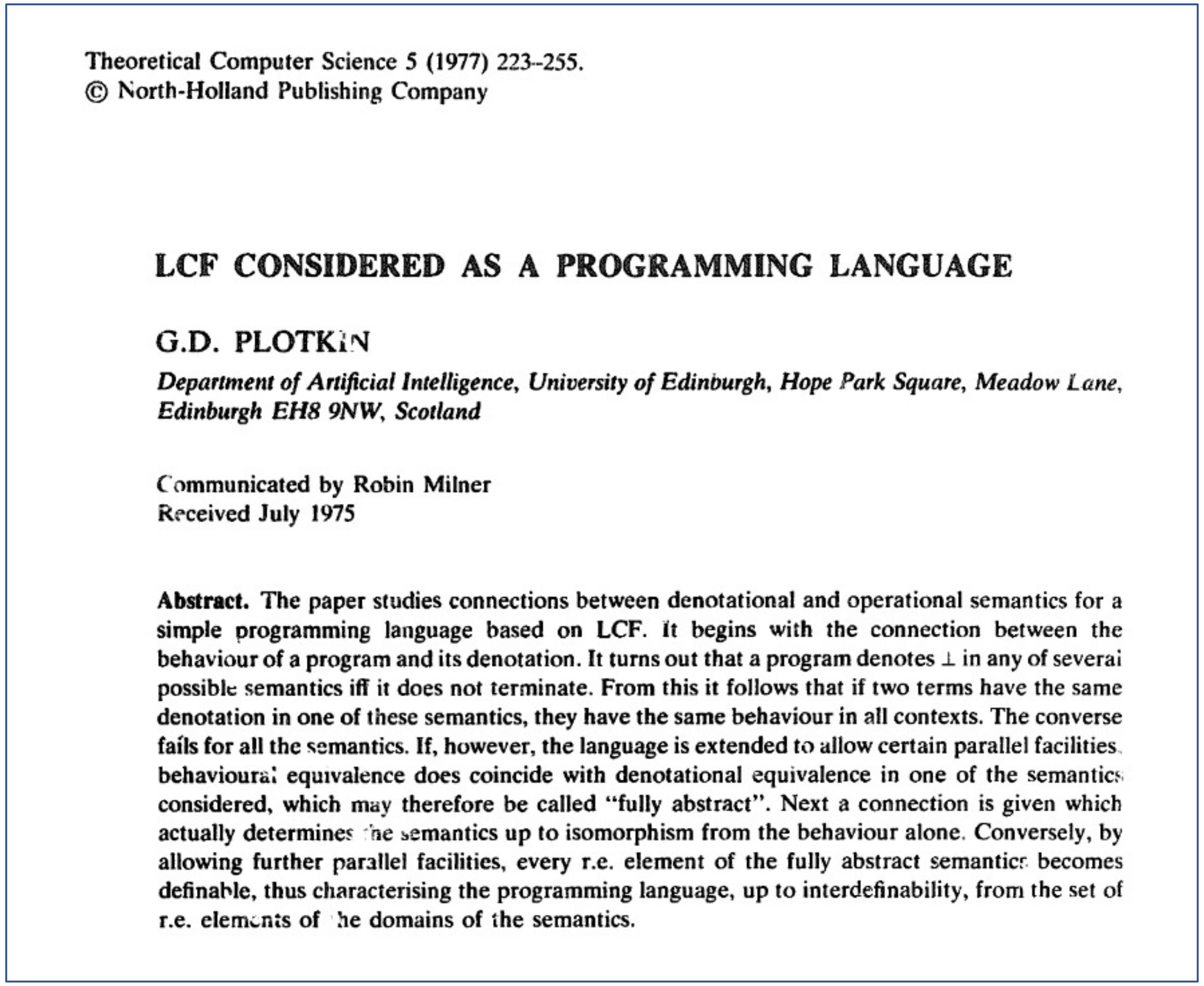
- Scott, 1969:

  The aim [of semantics] *is to develop a theory for correctness,* ***equivalence,*** *and termination, for a suitably rich language involving assignment, recursive procedures, and call by value*

- Scott, 1970-73: seminal mathematical model of the λ calculus, based of continuous functions.
  - Collaboration with Strachey
  - Birth of **Domain Theory**
  - The semantics of programs can be formed by math and logic. Proofs about programs can be made without using syntax & computations.

- Milner, Morris and Newey 1975: **Logic of Computable Functions (LCF)**
  - Theorem prover based o Scott's domain theory.
  - Enabled the verification of simple compilers.
  - Gave rise to the programming language ML.

Emergence of Program Equivalence (2/3)



Theoretical Computer Science 5 (1977) 223–255.
© North-Holland Publishing Company

LCF CONSIDERED AS A PROGRAMMING LANGUAGE

G.D. PLOTKIN
Department of Artificial Intelligence, University of Edinburgh, Hope Park Square, Meadow Lane, Edinburgh EH8 9NW, Scotland

Communicated by Robin Milner
Received July 1975

Abstract. The paper studies connections between denotational and operational semantics for a simple programming language based on LCF. It begins with the connection between the behaviour of a program and its denotation. It turns out that a program denotes ⊥ in any of several possible semantics iff it does not terminate. From this it follows that if two terms have the same denotation in one of these semantics, they have the same behaviour in all contexts. The converse fails for all the semantics. If, however, the language is extended to allow certain parallel facilities, behavioural equivalence does coincide with denotational equivalence in one of the semantics considered, which may therefore be called "fully abstract". Next a connection is given which actually determines the semantics up to isomorphism from the behaviour alone. Conversely, by allowing further parallel facilities, every r.e. element of the fully abstract semantics becomes definable, thus characterising the programming language, up to interdefinability, from the set of r.e. elements of the domains of the semantics.

Plotkin (1977) creates PCF, the basis for typed functional languages like Haskell and ML.

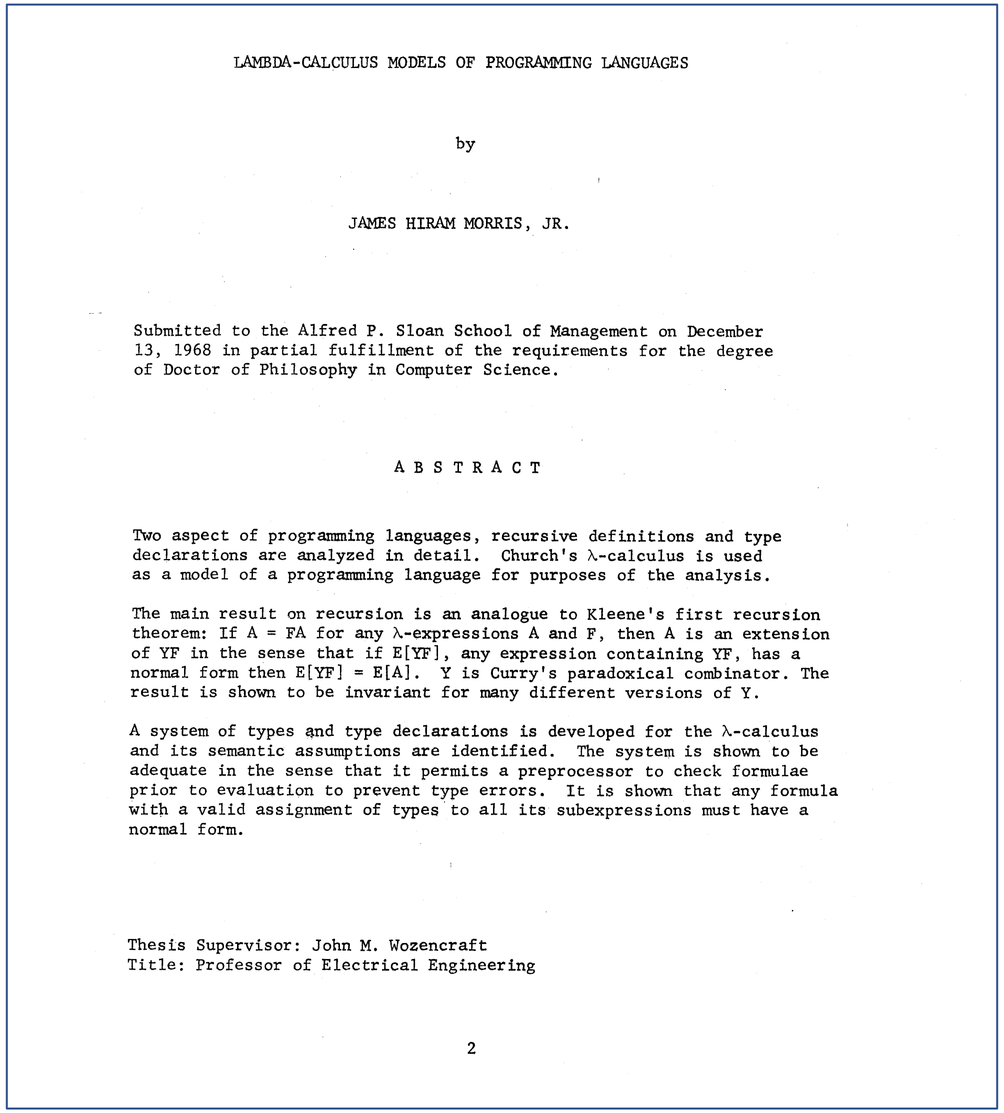
Abstract

… It turns out that a program denotes ⊥ iff it does not terminate.

From this it follows that if **two terms have the same denotation** in one of these semantics, **they have the same behaviour in all contexts. The converse fails for all the semantics…**

Emergence of Program Equivalence (3/3)

Morris 1969 PhD Thesis at MIT



LAMBDA-CALCULUS MODELS OF PROGRAMMING LANGUAGES

by

JAMES HIRAM MORRIS, JR.

Submitted to the Alfred P. Sloan School of Management on December 13, 1968 in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

A B S T R A C T

Two aspect of programming languages, recursive definitions and type declarations are analyzed in detail. Church's λ-calculus is used as a model of a programming language for purposes of the analysis.

The main result on recursion is an analogue to Kleene's first recursion theorem: If A = FA for any λ-expressions A and F, then A is an extension of YF in the sense that if E[YF], any expression containing YF, has a normal form then E[YF] = E[A]. Y is Curry's paradoxical combinator. The result is shown to be invariant for many different versions of Y.

A system of types and type declarations is developed for the λ-calculus and its semantic assumptions are identified. The system is shown to be adequate in the sense that it permits a preprocessor to check formulae prior to evaluation to prevent type errors. It is shown that any formula with a valid assignment of types to all its subexpressions must have a normal form.

Thesis Supervisor: John M. Wozencraft
Title: Professor of Electrical Engineering

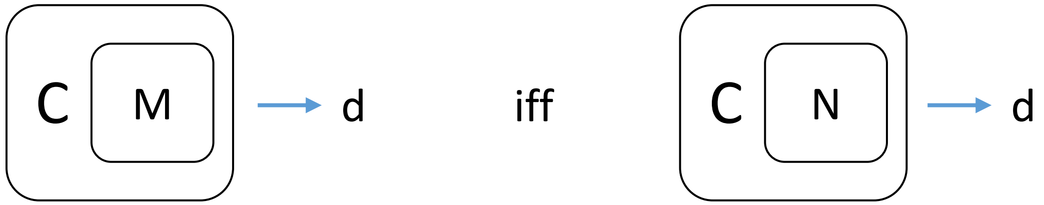
2

An Extensional Theory of **Contextual Equivalence**

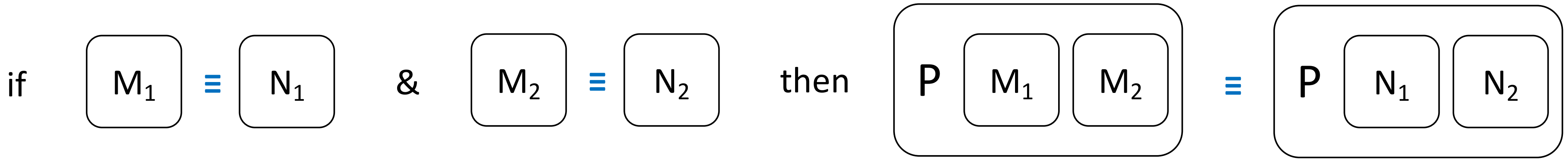
- Two terms are equivalent if they have the same behaviour inside any program context.
- Bohm Trees as meanings to λ terms
- Theorem: Two λ-terms are cxt equivalent when their Böhm trees are equal up to small transformations.
- It was later proven that Morris' and Scott's models agree: they equate the same λ-terms.

Contextual Equivalence

M ≡ N if interchangeable inside any program context C



- Captures **all functional** & some non-functional behaviour of M,N
- Supports **Compositional reasoning**:



Contextual Equivalence

M ≡ N if interchangeable inside any program context C

$\boxed{C\ \boxed{M}}$ terminates iff $\boxed{C\ \boxed{N}}$ terminates

In fact **termination** is enough

- Captures **all functional** & some non-functional behaviour of M,N
- Supports **Compositional reasoning**:

if $\boxed{M_1} \equiv \boxed{N_1}$ & $\boxed{M_2} \equiv \boxed{N_2}$ then $\boxed{P\ \boxed{M_1}\ \boxed{M_2}} \equiv \boxed{P\ \boxed{N_1}\ \boxed{N_2}}$

Contextual Equivalence becomes core in Semantics

- Scott’s and Morris’ models of the lambda calculus
  - They are “**sound**”:
    - if two λ terms are mapped to the same semantics object ➔ they are contextually equivalent.
  - But they are **not “fully abstract”** [Plotkin 1977]:
    - if two λ terms are contextually equivalent ➔ they may be distinguished by the semantics.
  - This turns out to mean: the semantics encompasses **more** than the behaviour of λ terms. This is somewhat unsatisfactory.

“parallel-or”

```
fun plfun1 f =
    if (f true _bot_) then
      if (f _bot_ true) then
        if (f false false) then
          _bot_
        else true
      else _bot_
    else _bot_
```

≡

```
fun plfun2 f =
  _bot_
```

The search continued for fully abstract models of programming languages

Contextual equivalence becomes core in PL semantics

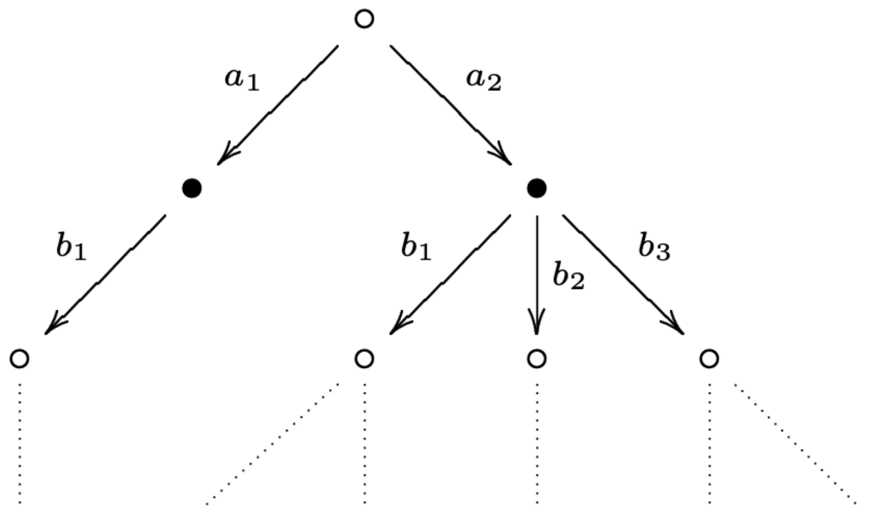
- Contextually equivalent functions:
  - no λ-calculus context can “see” a difference
- All early models of the λ-calculus distinguish the two (not fully abstract)
- If we add a “parallel or” operator in the λ calculus, then models become fully abstract
- Early models of the λ-calculus encompass exactly all λ-calculus behaviour + por

Game Semantics

[Abramsky,Jagadeesan,Malacaria’94][Hyland,Ong’95][Nickau’94]

- Mathematical objects: sequences of moves by a proponent and an opponent (plays)
- Meaning of terms = strategies of plays
- Achieved fully abstraction for the (typed) lambda calculus!*

- SIGLOG 2017 Alonzo Church award for the invention of Game Semantics



[Abramsky]

* With some caveats

Operational Semantics

- Meaning of terms = the computation steps in an abstract machine
- Early models of equivalence
- Logical Relations
- **Bisimulations** [Milner 1980’s]
- Provide effective proof techniques of equivalence

Program Equivalence

HOW DID IT COME ABOUT?

IS IT USEFUL IN PRACTICE?

WHAT ARE THE CHALLENGES?

AN APPROACH TO AUTOMATION

Is Program Equivalence Useful?

- Significant contribution to research
  - Equivalence has shaped a lot of PL semantics research (see previous slides)

- Applications areas include:

  Compiler correctness

  Verification of cryptographic protocols

  Regression verification [Strichman] [Ulbrich]

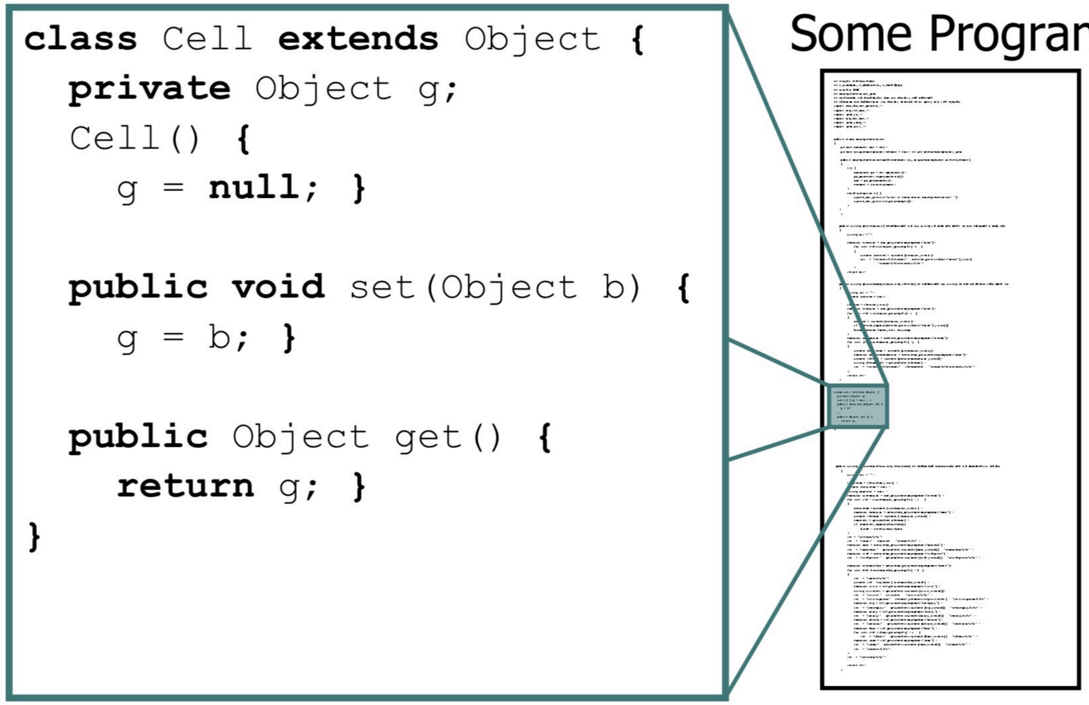
  General verification

Specification
≡
Implementation

Regression Verification

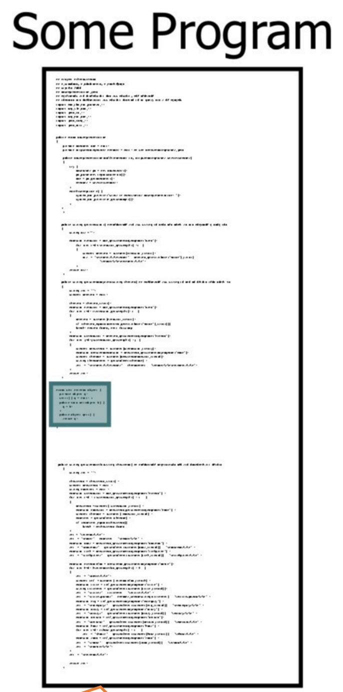
```
class Cell extends Object {
  private Object g;
  Cell() {
    g = null; }

  public void set(Object b) {
    g = b; }

  public Object get() {
    return g; }
}
```

Regression Verification



```
class Cell extends Object {
  private Object g;
  Cell() {
    g = null; }

  public void set(Object b) {
    g = b; }

  public Object get() {
    return g; }
}
```

Regression Verification



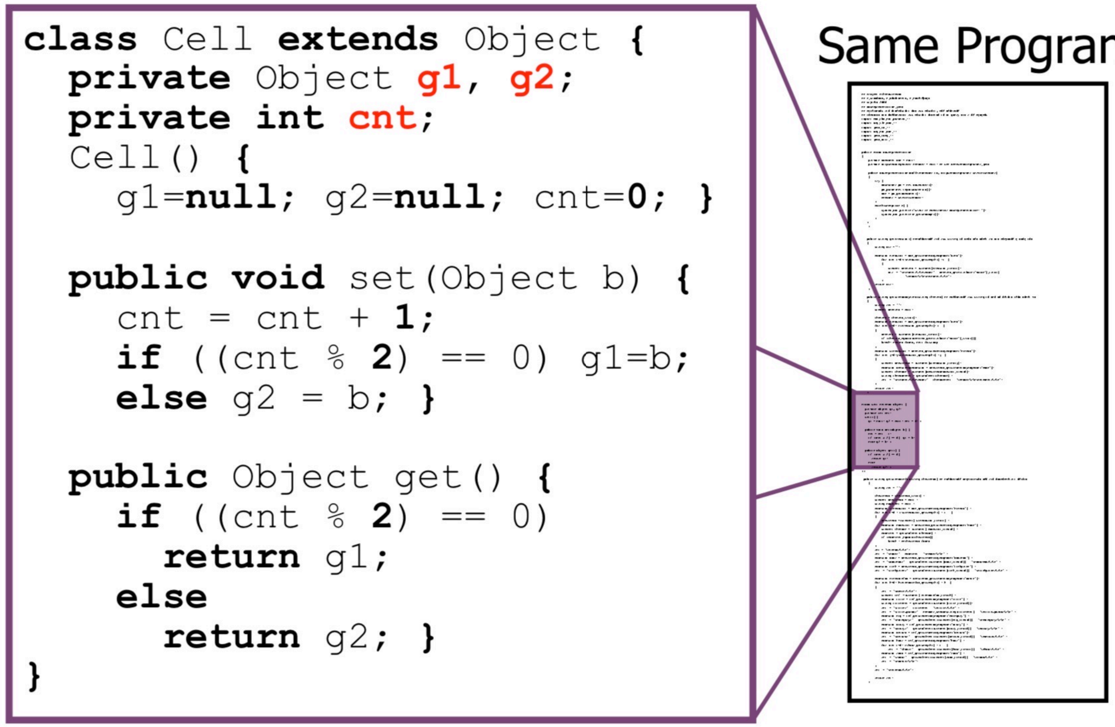
- Tried and tested
- Released to clients
- Becomes legacy

Regression Verification

```
class Cell extends Object {
  private Object g1, g2;
  private int cnt;
  Cell() {
    g1=null; g2=null; cnt=0; }

  public void set(Object b) {
    cnt = cnt + 1;
    if ((cnt % 2) == 0) g1=b;
    else g2 = b; }

  public Object get() {
    if ((cnt % 2) == 0)
      return g1;
    else
      return g2; }
}
```

Some Program

Regression Verification

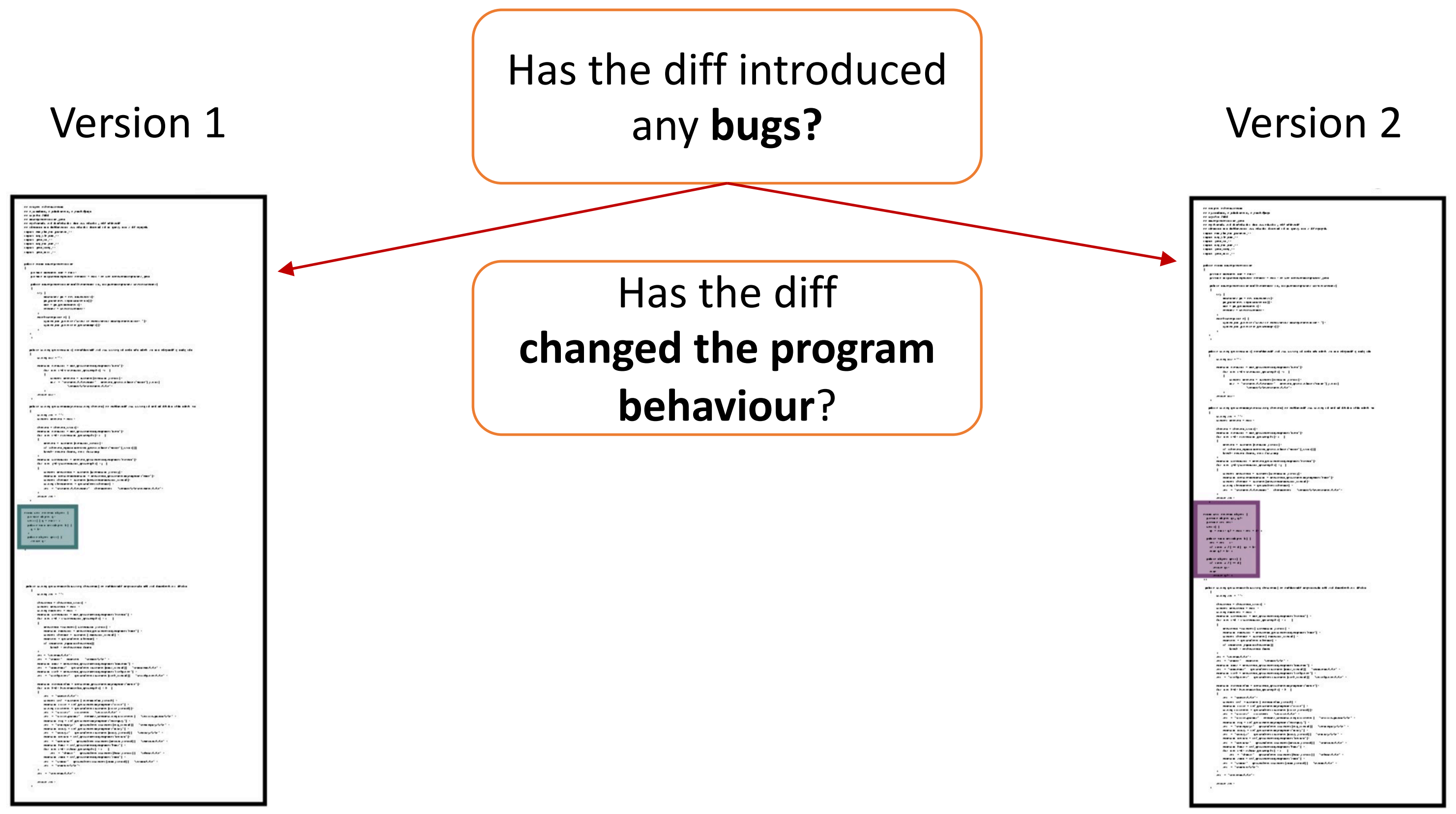


```
class Cell extends Object {
  private Object g1, g2;
  private int cnt;
  Cell() {
    g1=null; g2=null; cnt=0; }

  public void set(Object b) {
    cnt = cnt + 1;
    if ((cnt % 2) == 0) g1=b;
    else g2 = b; }

  public Object get() {
    if ((cnt % 2) == 0)
      return g1;
    else
      return g2; }
}
```

Regression Verification

Same Program

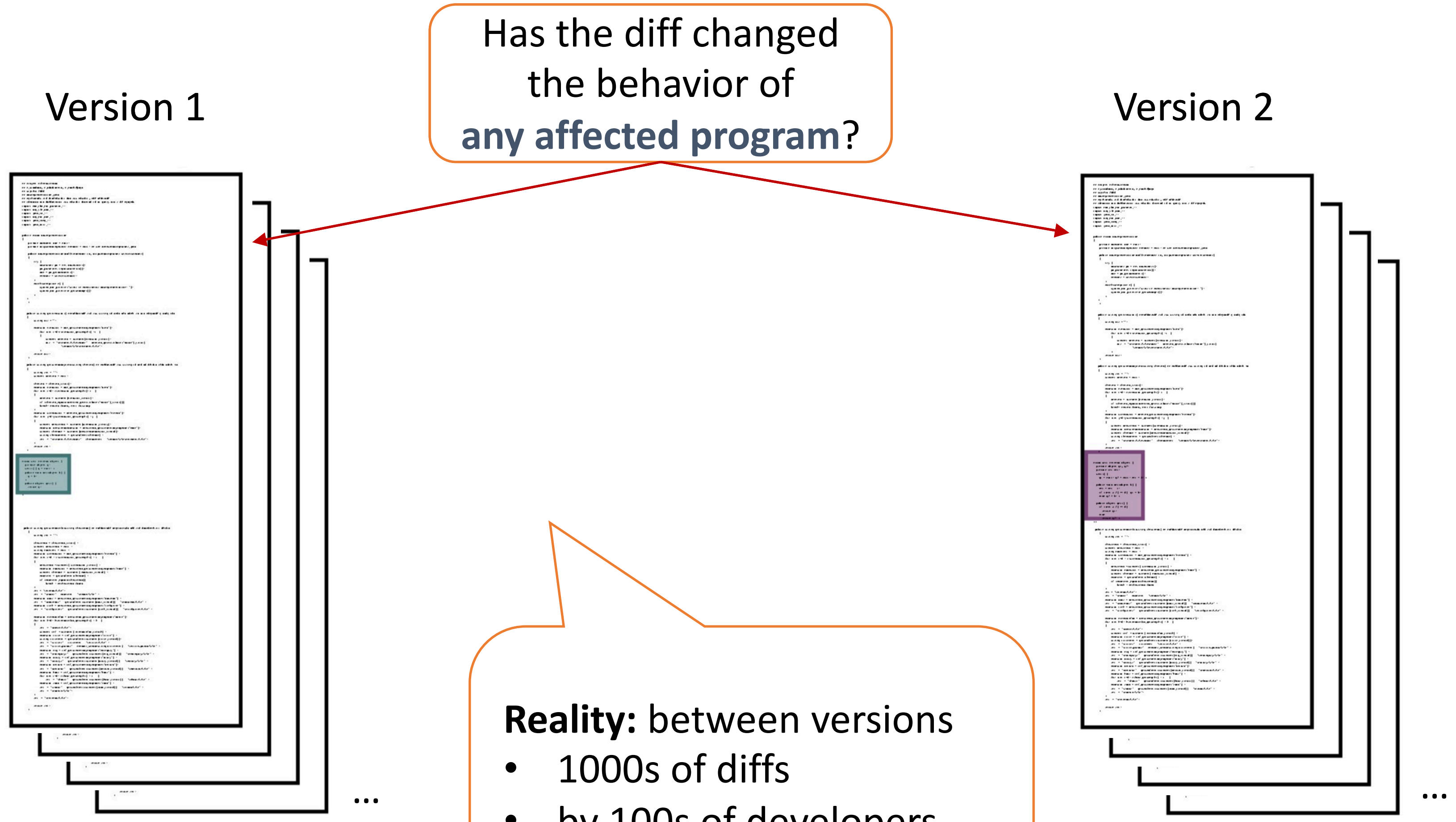
Regression Verification



Has the diff introduced any **bugs?**

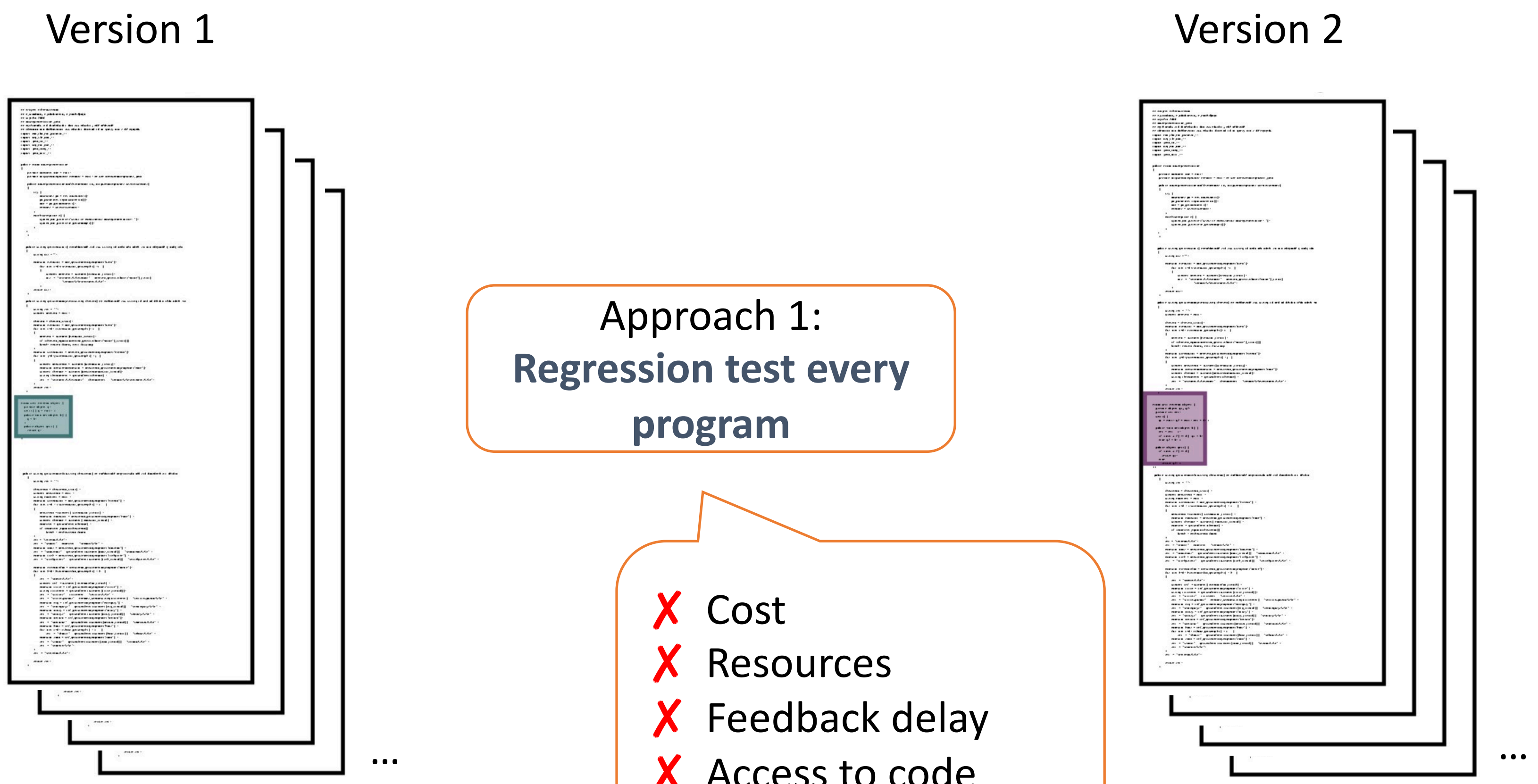
Has the diff **changed the program behaviour**?

Regression Verification


Version 1
Version 2
Has the diff changed the behavior of **any affected program**?
...
...

Reality: between versions
- 1000s of diffs
- by 100s of developers
- affecting many products

Regression Verification

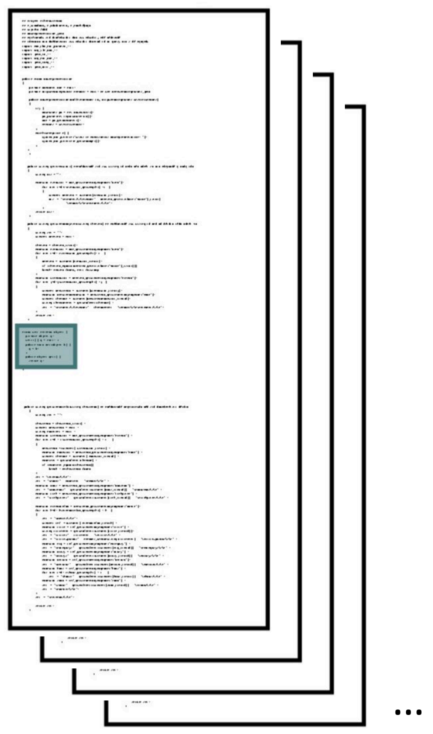


Approach 1:
Regression test every program

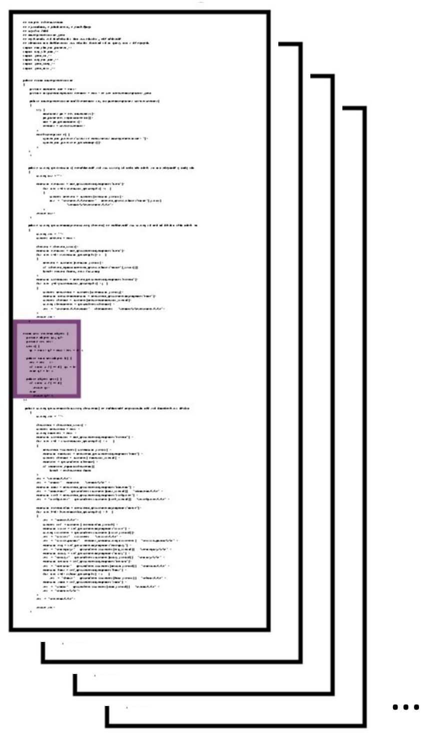
- ✗ Cost
- ✗ Resources
- ✗ Feedback delay
- ✗ Access to code

Regression Verification

Version 1



Version 2

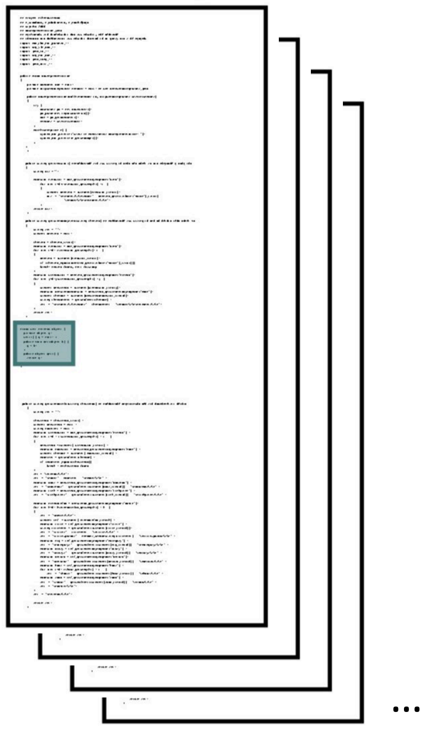


Approach 2:
Verify every program

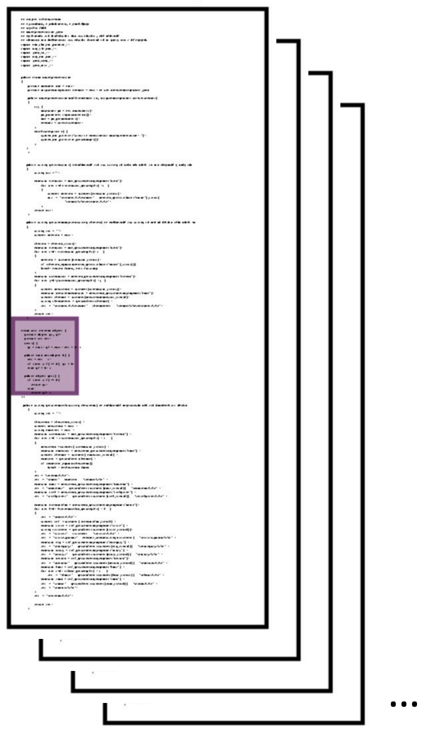
- ✗ Cost (formal model)
- ✗ Resources
- ✗ Feedback delay
- ✗ Access to code

Regression Verification

Version 1



Version 2



Approach 3:
Verify absence of
behaviour change
for every program pair

- ✗ Cost
- ✗ Resources (statespace)
- ✗ Feedback delay
- ✗ Access to code

Regression Verification

Version 1

Version 2

Approach 4:
Regression Verification
Verify absence of
<u>behaviour change for diff</u>

- **Contextual equivalence**
- **Contextual refinement**

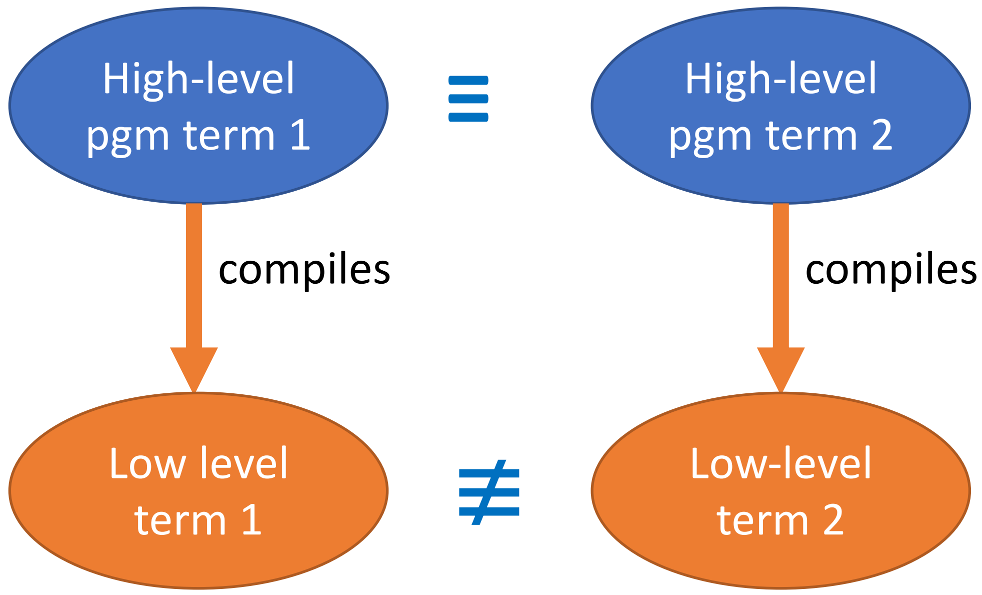
✓ Cost (no formal spec)
✓ Resources (smaller state-space)
✓ Feedback delay
✓ No need for full access to code

Compiler Correctness

[Abadi 1998] noted that full abstraction relates to security in compilers

Programmer
Reasons at high-level

Attacker may discover
vulnerabilities at low-level



Such vulnerabilities have been identified in Java and .NET [Abadi ‘98], [Kennedy 2006]

C# code

```
public class Counter {
  protected uint count = 0;
  public void Inc() { count++; }
  public virtual uint Get() { return count; }
}
public class ParityCounter : Counter {
  public override uint Get() { return count%2; }
}
```

```
class InsecureWidget {
  // No checking of argument
  public virtual bool Put(string s) {...}
}
class SecureWidget : InsecureWidget {
  // Check argument before delegating to superclass
  public override bool Put(string s)
  {  return Valid(s) ? base.Put(s) : false; }
}
```

Programmer's expectation

```
  void Bump(ParityCounter p) { p.Inc(); p.Inc(); }
≈ void Bump(ParityCounter p) { }
```

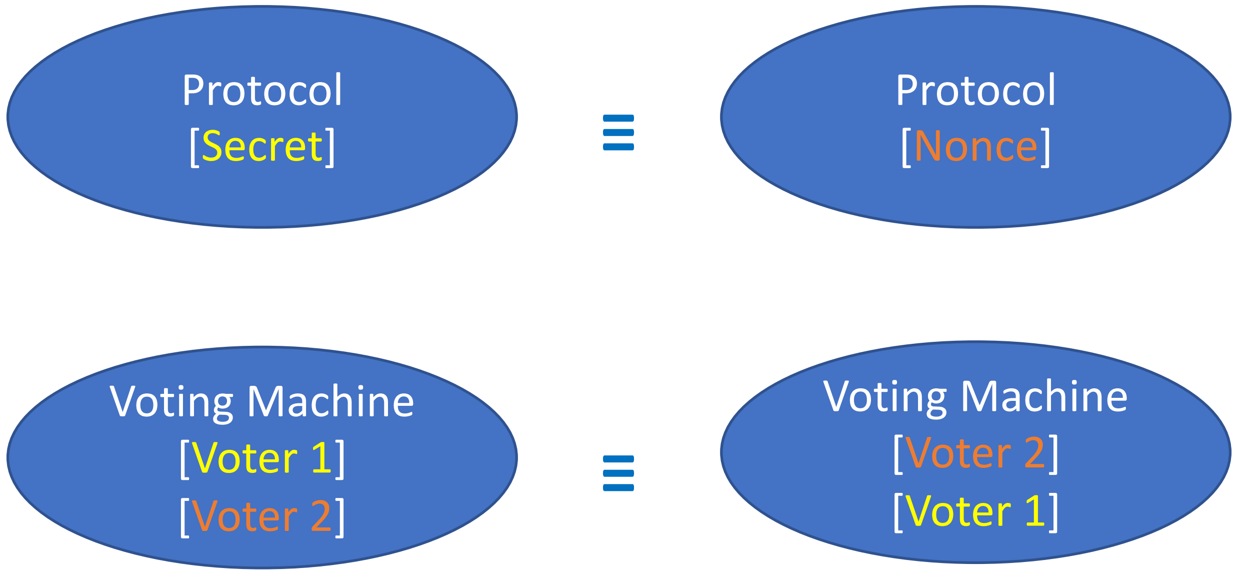
Vulnerability: compiled code in .NET breaks the expectation

```
.locals (class SecureWidget sw)
ldloc sw    ldstr "Invalid string"
call void InsecureWidget::Put(string)  // direct call
```

[Kennedy 2006]

Verification of Security protocols

- Seminal work: [Abadi, Fournet 2001] The applied pi calculus
- Secrecy, privacy, anonymity properties can be expressed as **contextual equivalence queries**
- Contexts: Dolev-Yao attackers


Protocol [Secret]
≡
Protocol [Nonce]
Voting Machine [Voter 1] [Voter 2]
≡
Voting Machine [Voter 2] [Voter 1]

Smart Contract Verification

The infamous DAO vulnerability in Etherium led to $60M worth of Ether to be stolen

```
private int balance := 100

private transfer (beneficiary, amount) :
if (balance >= amount )) then

    beneficiary.send (m);
    balance := balance - amount;

public contract (address,...) :
    ... transfer(address, 1 Eth)...
```

$\not\equiv$

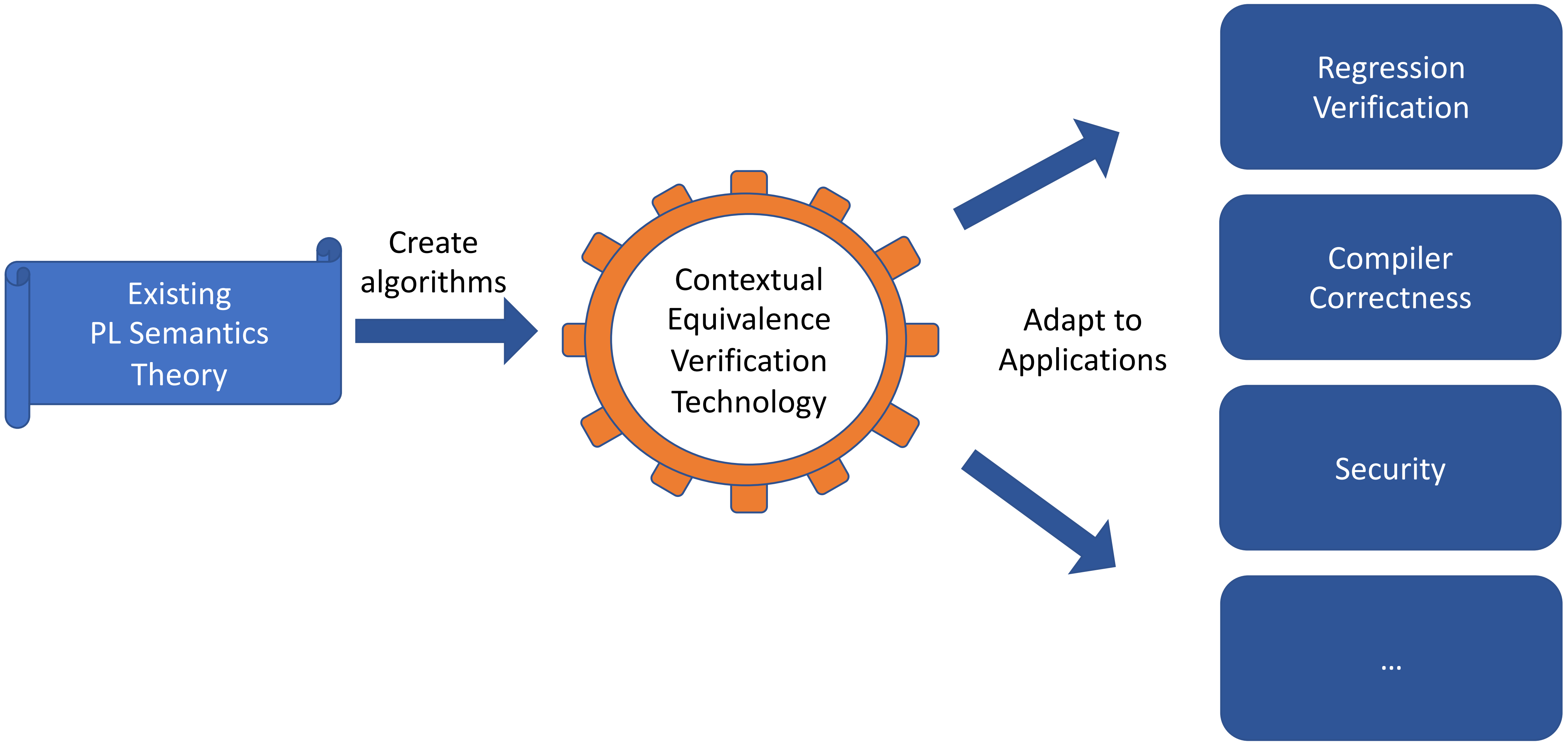
```
private int balance := 100

public transfer (beneficiary, amount) :
if (balance >= amount )) then
    old_balance := balance
    beneficiary.send (m);
    balance = balance - amount;
    assert (balance == old_balance + amount)

public contract (address,...) :
    ... transfer(address, 1 Eth)...
```

Fix:

```
balance := balance - amount;
send (m);
```

* Many similar Etherium SC vulnerabilities are being discovered each year.

Making Program Equivalence useful



Program Equivalence

HOW DID IT COME ABOUT?

IS IT USEFUL IN PRACTICE?

WHAT ARE THE CHALLENGES?

AN APPROACH TO AUTOMATION

Challenges in Creating Equivalence Verification Tech.

- Existing techniques are theoretical
  - Not made for **practical verification**
- The equivalence verification problem
  - Is **undecidable**
  - Involves **infinities**
    - integers: infinite base type domains, reasoning about arithmetic
    - Function closures: infinite higher type domains
    - Infinite interactions between program and context
  - …

Program Equivalence

HOW DID IT COME ABOUT?

IS IT USEFUL IN PRACTICE?

WHAT ARE THE CHALLENGES?

AN APPROACH TO AUTOMATION

The HOBBIT Approach

- **Undecidability:** Bounded exploration
- **Arithmetic:** symbolic execution + Z3 Theorem Prover
- **Function Closures:** Operational Game Semantics
- **Infinite interactions with context:** Bisimulation techniques
- **Released tool for ML-like language: https://github.com/LaifsV1/Hobbit**

Labelled Transition System

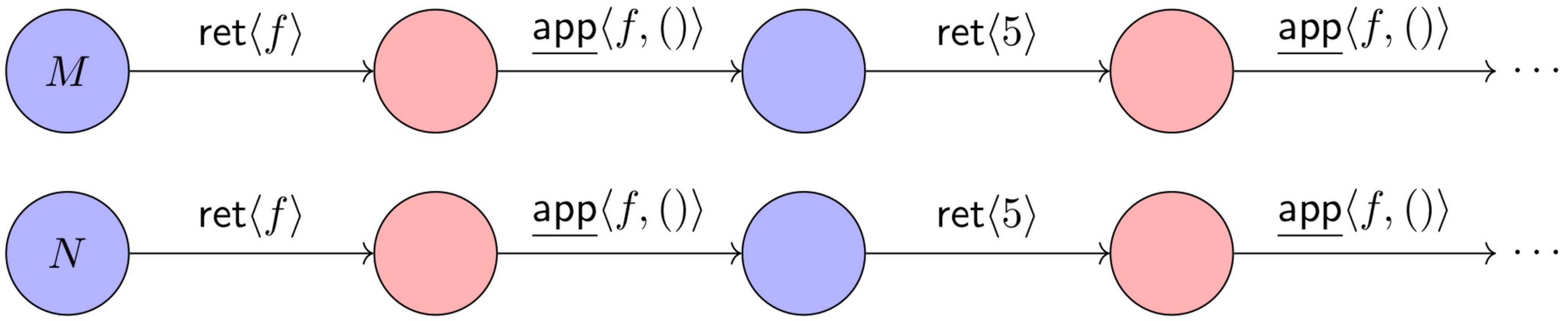
- Interface between **program** and **context**
- Two-player game
  - **Proponent**: program
  - **Opponent**: context
- Moves: Applications (**app** / **app**); Returns (**ret** / **ret**)
- Labelled Transition System: an abstract machine describing all possible interractions between **Proponent** and **Opponent**

- Benefit of Game Semantics: we only need to enumerate all opponent moves, not all opponent programs

Simple Examples

Equivalence:

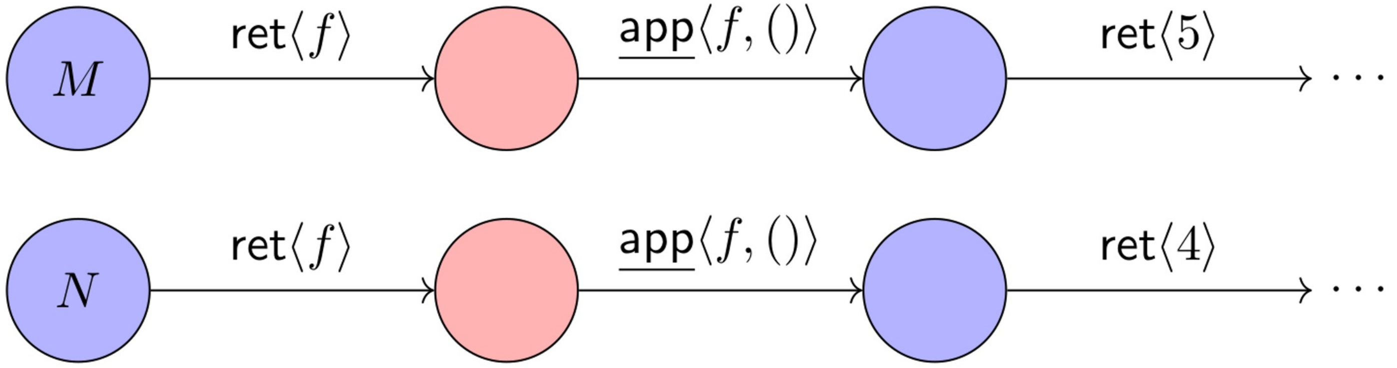
$M =$ `fun () -> (); 5` $N =$ `fun () -> 5`



M
$\mathsf{ret}\langle f\rangle$
$\underline{\mathsf{app}}\langle f, ()\rangle$
$\mathsf{ret}\langle 5\rangle$
$\underline{\mathsf{app}}\langle f, ()\rangle$
$\cdots$
N
$\mathsf{ret}\langle f\rangle$
$\underline{\mathsf{app}}\langle f, ()\rangle$
$\mathsf{ret}\langle 5\rangle$
$\underline{\mathsf{app}}\langle f, ()\rangle$
$\cdots$

Inequivalence:

$M =$ `fun () -> (); 5` $N =$ `fun () -> 4`

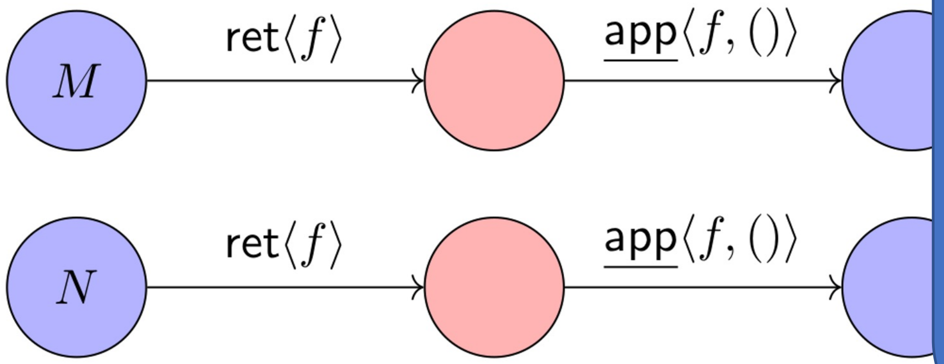


M
$\mathsf{ret}\langle f\rangle$
$\underline{\mathsf{app}}\langle f, ()\rangle$
$\mathsf{ret}\langle 5\rangle$
$\cdots$
N
$\mathsf{ret}\langle f\rangle$
$\underline{\mathsf{app}}\langle f, ()\rangle$
$\mathsf{ret}\langle 4\rangle$
$\cdots$

Simple Examples

Equivalence:

$M =$ `fun () -> (); 5`

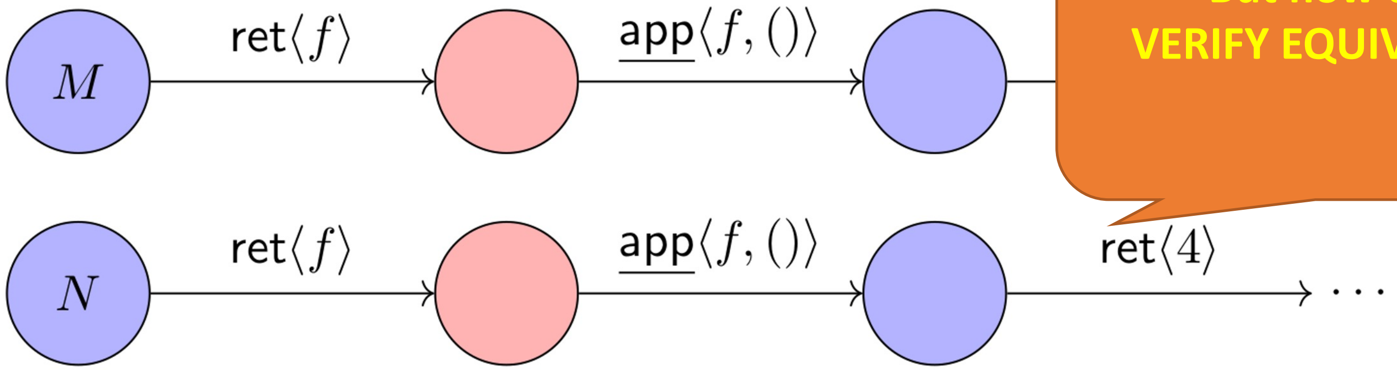


Bounded Model Checking

- Precise exploration
- Exhaustive exploration up to bound
- **Works well for verifying inequivalences**
  - Hobbit finds 68/68 inequivalences
  - 0.3sec per inequivalence avg.
  - Bound <203 app moves

Inequivalence:

$M =$ `fun () -> (); 5` $N =$ `fun`

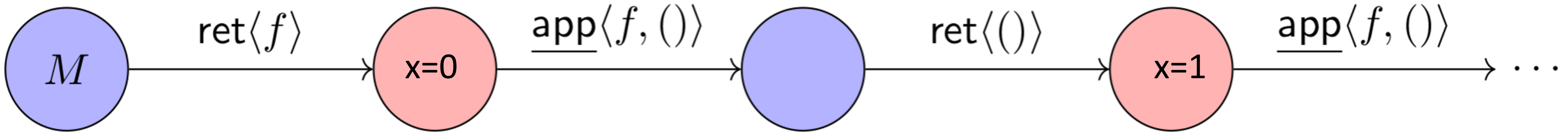


But how can we VERIFY EQUIVALENCES?

Verifying equivalence: dealing with infinities

- Opponent may **sequence** calls to the same function (inf. trace)

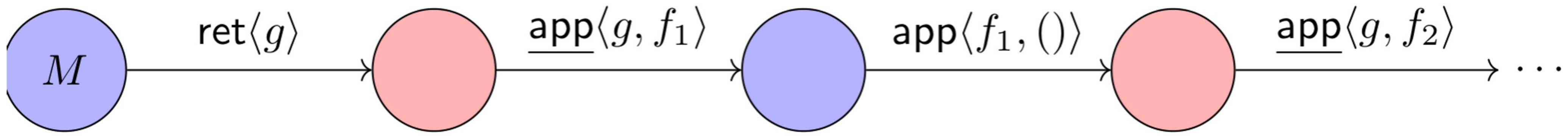
$M =$ `ref x = 0 in fun () -> x++` $N =$ `fun () -> ()`



M
ret⟨f⟩
x=0
$\underline{\text{app}}\langle f, ()\rangle$
ret⟨()⟩
x=1
$\underline{\text{app}}\langle f, ()\rangle$
. . .

- Opponent may **nest** calls to the same function (inf. trace & call stack)

$M =$ `fun f -> ref x = 0 in f (); !x` $N =$ `fun f -> f (); 0`



M
ret⟨g⟩
$\underline{\text{app}}\langle g, f_1\rangle$
app⟨f_1, ()⟩
$\underline{\text{app}}\langle g, f_2\rangle$
. . .

Finitising the exploration: loop detection

- Memoisation: loop exists if same configuration reached twice
- Normalisation: configurations identical up to permutations, α-equiv.
- Garbage Collection: configurations identical up to unused locations
  - Normalisation and GC expressed as bisimulation up-to techniques.

$M =$ `fun () -> ref x = 0 in !x` $N =$ `fun () -> 0`

Without Garbage Collection:

$$\begin{tikzcd}
M \arrow[r, "\mathsf{ret}\langle f\rangle"] & {} \arrow[r, "\underline{\mathsf{app}}\langle f, ()\rangle"] & x_1 : 0 \arrow[r, "\mathsf{ret}\langle 0\rangle"] & x_1 : 0 \arrow[r, "\underline{\mathsf{app}}\langle f, ()\rangle"] & \begin{array}{c} x_1 : 0 \\ x_2 : 0 \end{array} \arrow[r, "\mathsf{ret}\langle 0\rangle"] & \cdots
\end{tikzcd}$$

With Garbage Collection:

$$\begin{tikzcd}
M \arrow[r, "\mathsf{ret}\langle f\rangle"] & s_1 \arrow[r, "\underline{\mathsf{app}}\langle f, ()\rangle"] & x_1 : 0 \arrow[r, "\mathsf{ret}\langle 0\rangle"] & s_1 \arrow[r, "/" marking] & {}
\end{tikzcd}$$

Bisimulation up-to techniques

Bisimulation:

$C_1 \approx C_2$ iff

- $C_1 \overset{\alpha}{\Rightarrow} C_1'$ implies $\exists C_2'.\ C_2 \overset{\alpha}{\Rightarrow} C_2'$ and $C_1' \approx C_2'$
- Vice-versa

Bisimulation up to normalization and GC:

$C_1 \approx C_2$ iff

- $C_1 \overset{\alpha}{\Rightarrow} C_1'$ implies $\exists C_2'.\ C_2 \overset{\alpha}{\Rightarrow} C_2'$ and $\mathrm{norm}(\mathrm{gc}(C_1')) \approx \mathrm{norm}(\mathrm{gc}(C_2'))$
- Vice-versa

[Theory of bisimulation enhancements by Pous, Sangiorgi]

Bisimulation up-to techniques

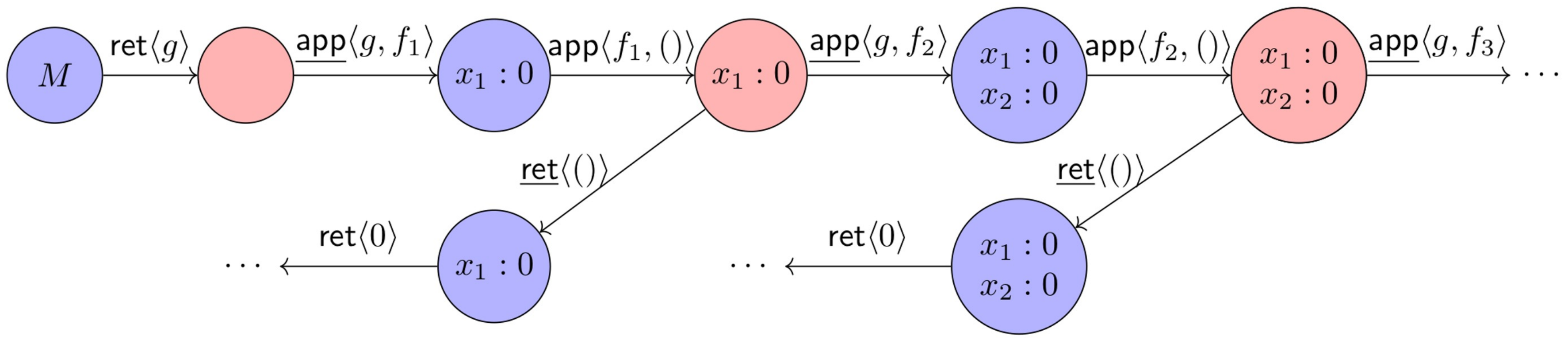
- Many existing up-to techniques more useful in manual proofs but **hard to implement** (e.g. up to context)
- LTS with **symbolic higher-order arguments** an implementable alternative to some advanced up-to techniques (e.g. up to context)
- Some standard up-to techniques (GC, normalisation) still important.
- BUT **more techniques needed** to address infinite LTSs in many examples
  - In our work we invent **3 new up-to techniques**

Up to Separation

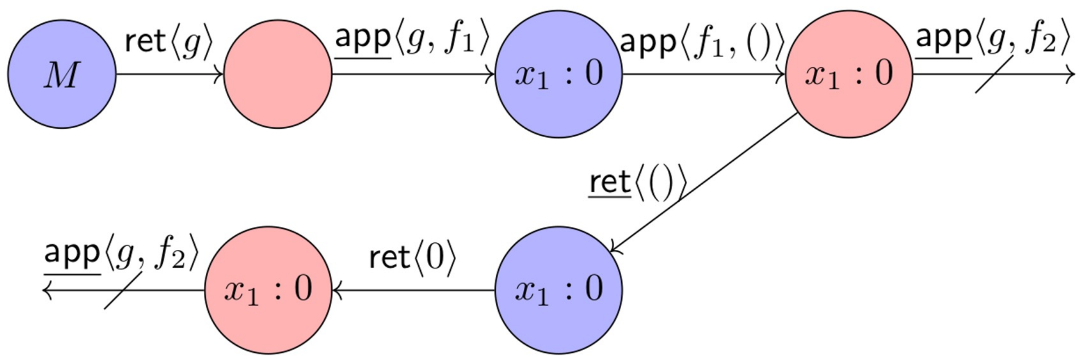
- Intuition: function calls that explore different parts of the state can be explored independently
- Corollary: calls to functions without shared state **need not be sequenced by opponent**

$M =$ `fun f -> ref x = 0 in f (); !x` $N =$ `fun f -> f (); 0`

Without Separation:



With Separation:


M
ret$\langle g\rangle$
app$\langle g, f_1\rangle$
$x_1 : 0$
app$\langle f_1, ()\rangle$
$x_1 : 0$
app$\langle g, f_2\rangle$
ret$\langle()\rangle$
$x_1 : 0$
ret$\langle 0\rangle$
$x_1 : 0$
app$\langle g, f_2\rangle$

Up to Separation

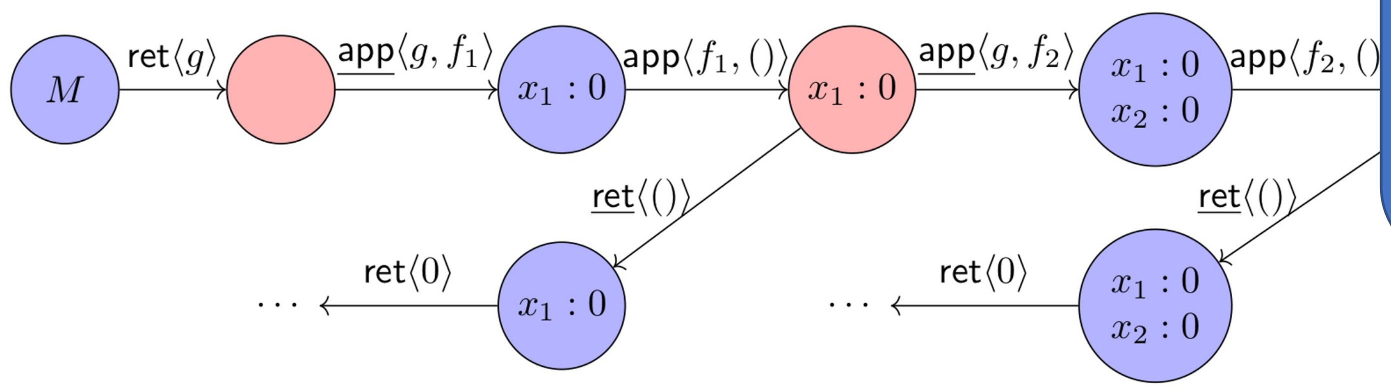
- Intuition: function calls that explore different parts of the state can be explored independently
- Corollary: calls to functions without shared state **need not be sequenced by opponent**

$M =$ `fun f -> ref x = 0 in f (); !x` $N =$ `fun f`

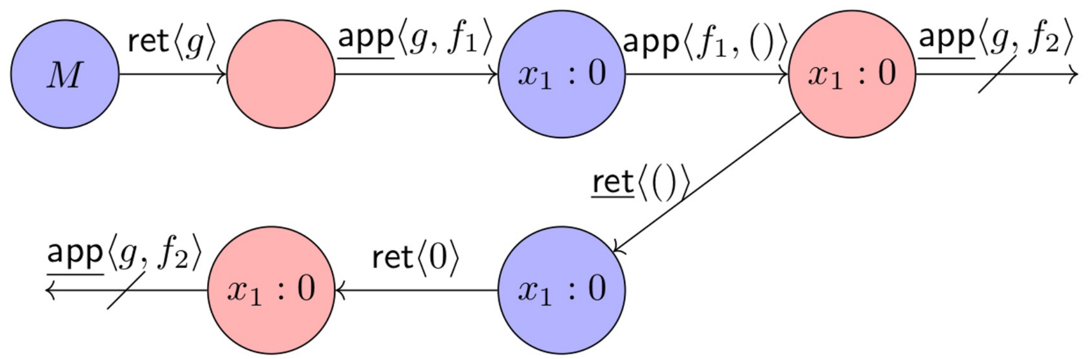
In many examples **removes infinite traces** due to:

- **Sequencing** function calls
- **Nesting** function calls

Without Separation:



With Separation:


M
ret$\langle g\rangle$
app$\langle g, f_1\rangle$
$x_1 : 0$
app$\langle f_1, ()\rangle$
$x_1 : 0$
app$\langle g, f_2\rangle$
ret$\langle()\rangle$
$x_1 : 0$
ret$\langle 0\rangle$
$x_1 : 0$
app$\langle g, f_2\rangle$

Easily **implementable:**

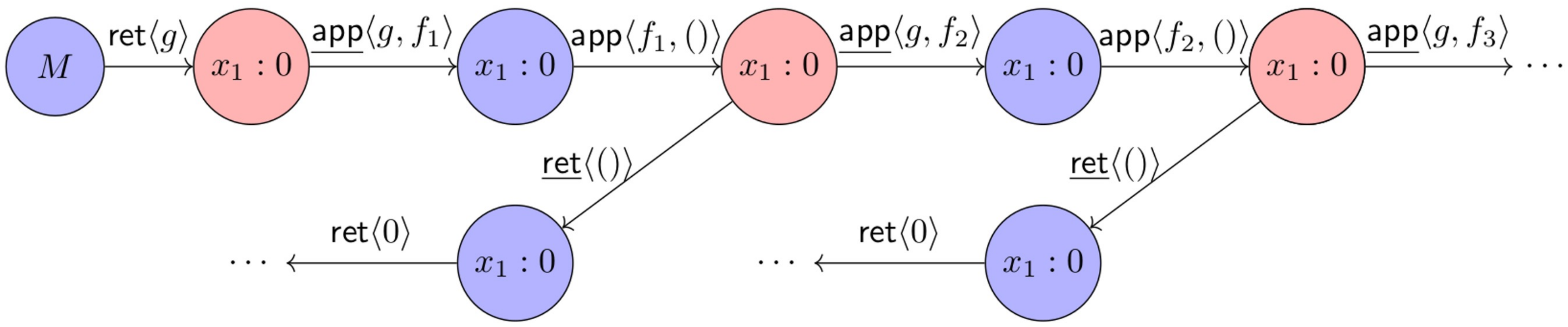
- Separate functions based on shared memory footprint

Up to Reentry

- Intuition: skip nested calls if functions do not observably modify shared state

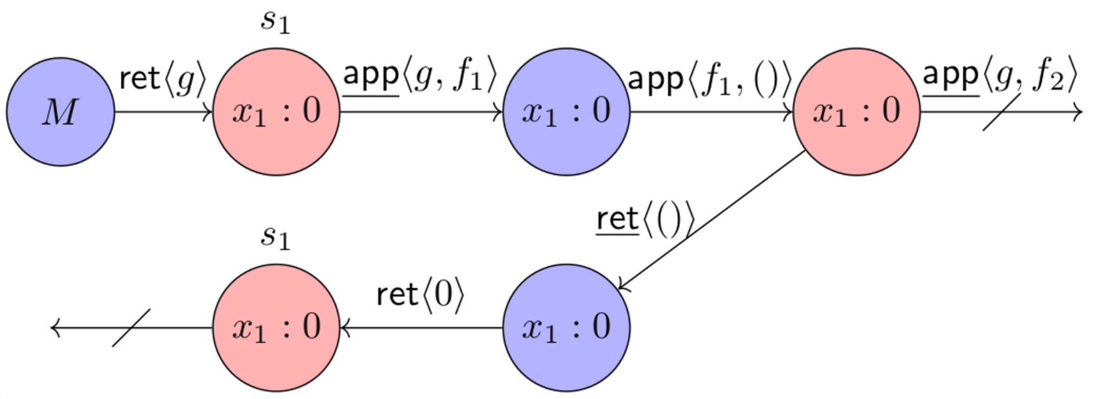
$M =$ **ref** `x = 0` **in fun** `f -> f (); !x` $N =$ **fun** `f -> f (); 0`

Without Up to Re-entry:



M
$\mathsf{ret}\langle g\rangle$
$x_1 : 0$
$\underline{\mathsf{app}}\langle g, f_1\rangle$
$\mathsf{app}\langle f_1, ()\rangle$
$\underline{\mathsf{app}}\langle g, f_2\rangle$
$\mathsf{app}\langle f_2, ()\rangle$
$\underline{\mathsf{app}}\langle g, f_3\rangle$
$\underline{\mathsf{ret}}\langle ()\rangle$
$\mathsf{ret}\langle 0\rangle$
$\cdots$

With Up to Re-entry:

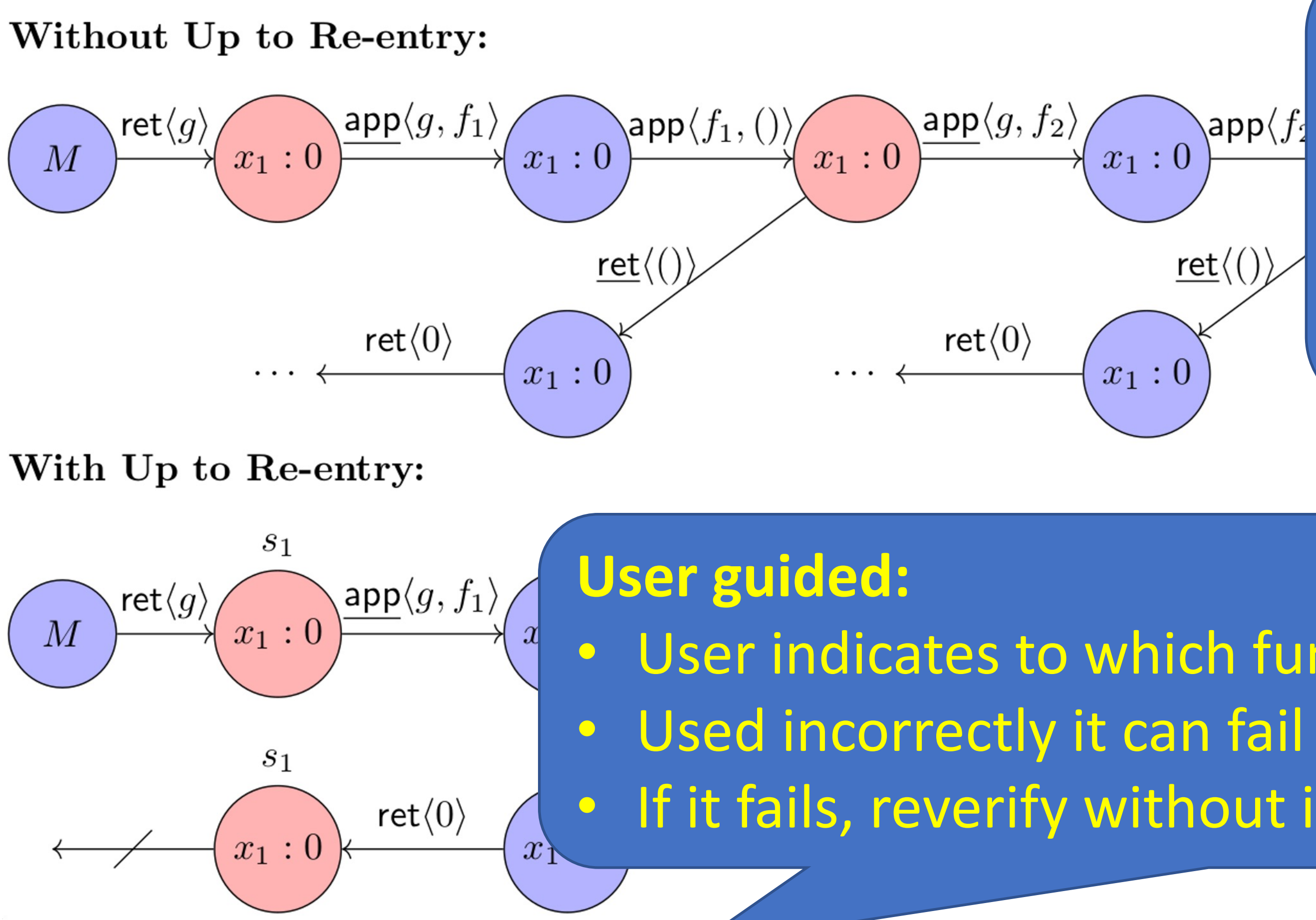


s_1
M
$\mathsf{ret}\langle g\rangle$
$x_1 : 0$
$\underline{\mathsf{app}}\langle g, f_1\rangle$
$\mathsf{app}\langle f_1, ()\rangle$
$\underline{\mathsf{app}}\langle g, f_2\rangle$
$\underline{\mathsf{ret}}\langle ()\rangle$
$\mathsf{ret}\langle 0\rangle$

Up to Reentry

- Intuition: skip nested calls if functions do not observably modify shared state

$M =$ `ref x = 0 in fun f -> f (); !x` $N =$ `fun f -> f (); 0`



Removes infinite traces due to **nesting** function calls in even more examples

User guided:
- User indicates to which functions to apply
- Used incorrectly it can fail in equivalences
- If it fails, reverify without it to weed out false-negatives

Up to State Invariants

- Intuition: use predicates over symbolic values to **abstract concrete shared state**

Without Abstraction:

$M =$ `ref x = 0 in fun () -> x++; !x > 0`
$N =$ `fun () -> true`

$$\begin{tikzcd}
M \arrow[r, "\mathsf{ret}\langle f \rangle"] & x = 0 \arrow[r, "\underline{\mathsf{app}}\langle f, () \rangle"] & x : 0 \arrow[r, "\mathsf{ret}\langle true \rangle"] & x : 1 \arrow[r, "\underline{\mathsf{app}}\langle f, () \rangle"] & x : 1 \arrow[r, "\mathsf{ret}\langle true \rangle"] & x : 2 \arrow[r, "\underline{\mathsf{app}}\langle f, () \rangle"] & \cdots
\end{tikzcd}$$

With Abstraction Invariant: $\exists w.(!x = w) \wedge (w \geq 0)$

$M =$ `ref x = 0 in fun () { w | x as w | w >= 0 } -> x++; !x > 0`
$N =$ `fun () -> true`

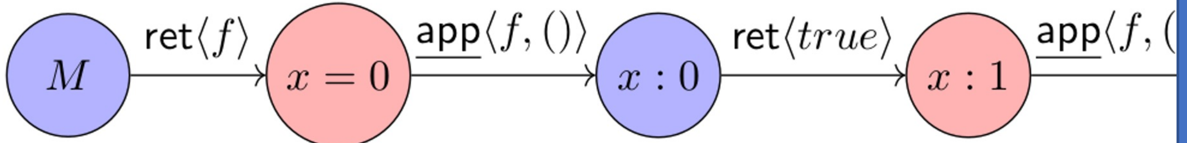
$$\begin{tikzcd}
 & & w \geq 0 & w \geq 0 & w \geq 0 & \\
M \arrow[r, "\mathsf{ret}\langle f \rangle"] & x : 0 \arrow[r, "\underline{\mathsf{app}}\langle f, () \rangle"] & x : w \arrow[r, "\mathsf{ret}\langle true \rangle"] & x : w \arrow[r, "\underline{\mathsf{app}}\langle f, () \rangle"] & x : w \arrow[r, "/" marking] & {} \\
 & & s_1 & & s_1 &
\end{tikzcd}$$

Up to State Invariants

- Intuition: use predicates over symbolic values to **abstract concrete shared state**

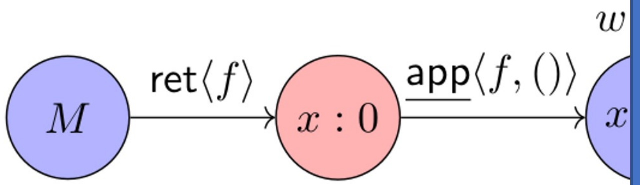
Without Abstraction:

$M =$ **ref** x = 0 **in fun** () -> x++; !x > 0
$N =$ **fun** () -> true



With Abstraction Invariant: $\exists w.(!x = w) \wedge (w \geq 0)$

$M =$ **ref** x = 0 **in fun** () { w | x **as** w | w >= 0 } -> x++; !x > 0
$N =$ **fun** () -> true



- **Removes infinite traces** due to **state change** in many examples
- Can describe **relational invariants**

User guided:

- User provides invariants (no inference atm)
- Used incorrectly it can fail in equivalences
- If it fails, reverify without it to weed out false-negatives

HOBBIT: Higher Order Bounded BIsimulation Tool

All techniques implemented in HOBBIT.

https://github.com/LaifsV1/Hobbit

- Only reports true positive, true negative, bound exhausted

What can we verify with these techniques?

- 68/68 inequivalences [20sec]
- 72/105 equivalences [5.6sec]
  - Only 32 [1623sec] **without up to separation**
  - Only 57 [178sec] **without up to reentry**
  - Only 47 [178sec] **without up to invariants**
  - **Only 3 [2098sec] without any up-to techniques!**

HOBBIT: Higher Order Bounded BIsimulation Tool

- We can verify all [Meyer-Sieber ‘88] example equivalences
  - Benchmark examples for equivalence proof techniques

```
M = let loc_eq loc1loc2 = [...] in
    fun q -> ref x = 0 in
             let locx = (fun () -> !x) , (fun v -> x := v) in
             let almostadd_2 locz {w | x as w | w mod 2 == 0} =
               if loc_eq (locx,locz) then x := 1 else x := !x + 2
             in q almostadd_2; if !x mod 2 = 0 then _bot_ else ()

N = fun q -> _bot_
```

- Many but not all example equivalences from the literature
  - Well-bracketed examples [Jaber – SyTeCi]
  - Internal recursion examples [Ulbrich et. al – Reve] [Strichman et. al – RVT]

Future Work

- Handle all example equivalences from the literature for this deterministic programming language
- Extend the approach to non-deterministic and concurrent languages, purely

  functional (λ calculus)
- Apply to real-world settings such as regression verification

New theory and verification techniques required!

Thank you!