PROPERTIES OF A PROGRAM'S RUNTIME STATE SPACE

Neil D. Jones (Professor emeritus, University of Copenhagen)

August 12, 2010

ESSLLI 2010 Copenhagen

OUTLINE, PROGRESS

1. • Programs
2. Models in First Order Logic (. . . and programs. . .)
3. But . . . Aren't all nontrivial questions about real programs undecidable?
4. Subrecursive programming languages (life without CONS)
5. One-sided program analyses (approximating reachable states)
6. Termination analysis (approximating state transitions)
7. Conclusions

This talk is not encyclopedic. Main focus: areas close to my own research.

PROGRAMS

A program is a syntactic object. We typically call it $\mathtt{p}$, $\mathtt{q}$, ...

Its purpose is to realise some computational intent.

Semantics $[\![\mathtt{p}]\!]$ of program $\mathtt{p}$: the computation(s) specified by $\mathtt{p}$.

What are computations? Many variants...

HOW computations can take place:

- deterministic versus nondeterministic;
- finite versus continuing/infinite;
- local versus global;
- concurrent; synchronous or asynchronous;
- quantum, ...

WHAT the purpose of running $\mathtt{p}$ may be: very many possibilities.

For this talk, program $\mathtt{p}$'s purpose is to compute an an input-output function (possibly partial: $\mathtt{p}$ may not terminate on all inputs)

PROGRAM ANALYSIS OVERVIEW

What can we say about a program's behavior based on its syntax alone?

Practical needs: dangerous applications, reliability, performance, ...

- The practice of compilers:
  - Compiler construction, including code optimisation phases
  - Compiler correctness: what is it, how to define?

Relevant theory:

- Theory of computability: Turing machines, Turing completeness, the Halting Problem, Rice's theorem, the Rogers Isomorphism theorem, ...
- Programming language semantics
- Abstract interpretation, or program flow analysis
- Computational Complexity theory
  - Effect of programming style (functional, imperative, logic programming, process algebra ...) on what can be expressed by programs

(a Whorfian hypothesis?)

OUTLINE, PROGRESS

1. Programs
2. • Models in First Order Logic (. . . and programs. . .)
3. But . . . Aren't all nontrivial questions about real programs undecidable?
4. Subrecursive programming languages (life without CONS)
5. One-sided program analyses (approximating reachable states)
6. Termination analysis (approximating state transitions)
7. Conclusions

MODELS IN FIRST ORDER LOGIC

- Near the beginning: Tarski 1936

  The concept of truth in formalized languages

- Heinrich Scholz 1952:

  asked for a characterization of spectra, i.e., sets of natural numbers that are the cardinalities of finite models of first order sentences.

- Scholz probably expected something like:

  Spectra are the smallest class of sets of natural numbers that contain . . . and are closed under the operations . . .

- Surprise!

  The first solution was a characterisation by complexity classes (Jones & Selman 1972).

- Finite model theory was then developed much further, e.g., by Fagin, Gurevich, Immerman, . . .)

ANALOGY: FIRST ORDER LOGIC AND PROGRAMS

- A first order formula ϕ is a bit like a program $\mathtt{p}$
- A model $\mathcal{M}$ of FOL formula ϕ is analogous to the runtime state space of a program $\mathtt{p}$.
- Viewed as a "program", ϕ has only boolean logic $(\wedge, \vee, \neg)$ and simple iteration $(\forall, \exists)$ with no accumulator
- Dimension of interest: the cardinality of model $\mathcal{M}$.

The Spectrum of a FOL formula ϕ is

$$\text{SPECTRUM}(\phi) = \{n \in \mathbb{N} \mid \phi \text{ has a model } \mathcal{M} \text{ of cardinality } n\}$$

Example:

- If $\phi =$ the axioms for Boolean Algebra
- then

$$\text{SPECTRUM}(\phi) = \{2^m \mid m \geq 0\}$$

by Stone's representation theorem

An answer to Scholz' problem:

$I \subseteq \mathbb{N}$ IS A SPECTRUM

IF AND ONLY IF

$I \in$ NEXPTIME

(Viewpoint: regard I as a set of numbers written as bit strings)

Proof both ways by programming:

- $\Rightarrow$: given ϕ, find a nondeterministic program p to answer the question

  "is $n \in \text{SPECTRUM}(\phi)$?"

  (also: show that p runs in exponential time)

- $\Leftarrow$: show that

  - for any nondeterministic program p that runs in exponential time

  - there exists a first order formula ϕ that simulates the running of p

OUTLINE, PROGRESS

1. Programs
2. Models in First Order Logic (. . . and programs. . .)
3. • But . . . Aren't all nontrivial questions about real programs undecidable?
4. Subrecursive programming languages (life without CONS)
5. One-sided program analyses (approximating reachable states)
6. Termination analysis (approximating state transitions)
7. Conclusions

BUT . . . AREN'T ALL NONTRIVIAL QUESTIONS ABOUT REAL PROGRAMS UNDECIDABLE?

The 1930s golden age: Turing/Church/Post/Kleene/. . . :

- They devised many formulations of the class of all computable functions. Superficially quite different.
- All turned out to be equivalent (any one could simulate any other)
- Even stronger: Hartley Rogers showed that all programming languages are isomorphic.

Alan Turing: halting problem is undecidable (by mechanical computation).

WHAT: Given, a program `p`, and an input `d` to run it on

TO DECIDE: Will `p` eventually terminate its computation on `d` ?

And yet more:

Rice's Theorem: any nontrivial question about program behavior is undecidable.

(shown by reduction from the halting problem)

HOW TO AVOID THIS DILEMMA ?
(or, is computer science hopeless?)

Programs are ubiquitous – we can't live without them! What to do?

One way: Sacrifice Turing completeness and use subrecursive languages, e.g.,

- finite models of first-order or temporal logic or
- ICC (implicit computational complexity, e.g., Dal Lago and Martini here)
- Strongly normalising programming languages, e.g., constructive type theory, System F,. . .

Another way: Do "one-sided" analyses of a Turing-complete language

- Older: practical compiler work since the 1950s. Program flow analysis is widely used in optimising compilers

  (Fortran, Algol, C, Java, Haskell,. . .)
- More semantically based program analyses: abstract interpretation (Cousot, Nielson, Hankin, Jones, Muchnick, . . .)

OUTLINE, PROGRESS

1. Programs
2. Models in First Order Logic (. . . and programs. . .)
3. But . . . Aren't all nontrivial questions about real programs undecidable?
4. • Subrecursive programming languages (life without CONS)
5. One-sided program analyses (approximating reachable states)
6. Termination analysis (approximating state transitions)
7. Conclusions

SUBRECURSIVE PROGRAMMING LANGUAGES

Approach:

study function classes that are smaller than the class of all computable partial functions.

Early works in this direction (1930s – 1958):

- **Primitive recursive functions (Kurt Gödel, Rósza Péter)**
- **Gödel's System T (much larger, still total and computable)**
- **The Grzegorczyk hierarchy inside the primitive recursive functions**

$$\varepsilon^0 \subsetneq \varepsilon^1 \subsetneq \varepsilon^2 \subsetneq \varepsilon^3 \subsetneq \ldots \subseteq PrimRec$$

Alas, even the small class $\varepsilon^3 = Elementary$ is too large to be of practical relevance to computer scientists! (And the bigger classes are much bigger...)

PROBLEM CLASSES SMALL ENOUGH TO BE OF PRACTICAL RELEVANCE TO COMPUTER SCIENTISTS

- P, or PTIME: **decision problems solvable by programs**

  in time bounded by a polynomial function (of the length of the input)

- NP, or NPTIME: **ditto, but**

  a program/algorithm may be nondeterministic, i.e., it may "guess"

- LOGSPACE: **a class smaller than P**
- EXPTIME: **a larger class, getting near to infeasibility**
- NEXPTIME: **still larger, a nondeterministic version of** EXPTIME

Complexity classes:

$$\text{LOGSPACE} \subseteq \text{PTIME} \subseteq \text{NPTIME} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \ldots$$

(all are proper subsets of Grzegorczyk's "elementary" class ε^3)

A SMALL SUBRECURSIVE LANGUAGE: "READ-ONLY" PROGRAMS

Data Structures: Booleans, lists of booleans (and functions, for higher-order programs). 0-order = booleans or lists of Booleans.

Read-only programs: "Life without CONS"

- no constructors or memory allocation or $x + 1$ allowed;
- only $x - 1$ or $hd(x)$,...)

Data types: 0-order, 1-order, 2-order, ... [finite orders only]

Control Structures: some choices

- Primitive recursion (FOR-loops only, called "folds" in functional programming), eg `f(x) = for y := 1 to x do {x := x+y}; return x`
- Tail recursion (WHILE-loops; they seem a bit more powerful),
- General recursion: nested function calls, eg `f(x,y) = g(x,h(y))`
- Calls to higher-order functions, eg `double(f,x) = f(f(x))`

TYPICAL RESULTS

- Gurevich: 0-order primitive recursive read-only programs can decide
  all and only the problems in LOGSPACE.
- Jones: 0-order general recursive read-only programs can decide
  all and only the problems in PTIME.
- Gurevich: 1-order primitive recursive read-only programs can decide
  all and only the problems in PTIME.

Conclusion from the first two: (for 0-order data)

general recursion is stronger than primitive recursion
if and only if
PTIME properly includes LOGSPACE

Alas, this doesn't answer the expressibility question
"is general recursion stronger than primitive recursion ?".

BUT: it shows it equivalent to another very hard question!
A question in family with: is $P = NP$?

DECISION POWER, BY SEVERAL PROGRAM CONTROLS AND SEVERAL DATA ORDERS

Programs	Data Order 0	Data Order 1	Data Order 2	...	LIMIT
Read-write	REC. ENUM	REC. ENUM	REC. ENUM	...	REC. ENUMERABLE
Primitive recursive (foldr)	PRIM.REC.	PRIM^{1}REC.	PRIM^{2}REC.	...	**System T**
General rec. (RO)	PTIME	EXPTIME	EXP^{2}TIME	...	ELEMENTARY $= \varepsilon^3$
Tail recursive RO	LOGSPACE	PSPACE	EXPSPACE	...	ELEMENTARY $= \varepsilon^3$
Primitive rec. RO	LOGSPACE	PTIME	PSPACE	...	ELEMENTARY $= \varepsilon^3$

Top half notation:

- RECURSIVELY ENUMERABLE = **all problems solvable by a Turing machine (that only halts on "yes" answers)**
- PRIMITIVE RECURSIVE = **usual Gödel-style, including successor $x + 1$**
- **System T =** PRIMITIVE RECURSIVE **of any finite order.** **Huge!**

DECISION POWER, BY SEVERAL PROGRAM KINDS AND SEVERAL DATA ORDERS

Programs	Data Order 0	Data Order 1	Data Order 2	...	LIMIT
Read-write	REC. ENUM	REC. ENUM	REC. ENUM	...	REC. ENUMERABLE
Primitive recursive (foldr)	PRIM.REC.	$\text{PRIM}^1\text{REC.}$	$\text{PRIM}^2\text{REC.}$	...	**System T**
General rec. (RO)	PTIME	EXPTIME	EXP^2TIME	...	ELEMENTARY $= \varepsilon^3$
Tail recursive RO	LOGSPACE	PSPACE	EXPSPACE	...	ELEMENTARY $= \varepsilon^3$
Primitive rec. RO	LOGSPACE	PTIME	PSPACE	...	ELEMENTARY $= \varepsilon^3$

Bottom half notation:

Read-only (RO) programs that have no successor $x + 1$.

- **Complexity classes:**

$$\text{LOGSPACE} \subseteq \text{PTIME} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \ldots$$

- **Grzegorczyk's class** ELEMENTARY $= \varepsilon^3 = \bigcup_{k=0}^{\infty} \text{EXP}^k\text{TIME}$. **Pretty big!**

READ-ONLY PROGRAMS, DIFFERENT DATA ORDERS

Programs	Data Order 0	Data Order 1	Data Order 2	...	LIMIT
General rec. (RO)	PTIME	EXPTIME	EXP^2TIME	...	ELEMENTARY $= \varepsilon^3$
Tail recursive RO	LOGSPACE	PSPACE	EXPSPACE	...	ELEMENTARY $= \varepsilon^3$
Primitive rec. RO	LOGSPACE	PTIME	PSPACE	...	ELEMENTARY $= \varepsilon^3$

(proven by: Gödel, Gurevich, Goerdt, Seidl, Jones.)

- **For read-only programs, data order 0:** Is general recursion more powerful than tail recursion or primitive rec. ? **Equivalent by table:**

  Is PTIME $\supsetneq$ LOGSPACE **?**

  A long-standing open problem!

- **For read-only programs, data order 1:** general recursion IS more powerful than primitive recursion **since**

  $$\text{EXPTIME} \supsetneq \text{PTIME}$$

  A problem with deriving programs from proofs?

  Reasoning: induction proofs lead to primitive recursion.

OUTLINE, PROGRESS

1. Programs
2. Models in First Order Logic (. . . and programs. . .)
3. But . . . Aren't all nontrivial questions about real programs undecidable?
4. Subrecursive programming languages (life without CONS)
5. • One-sided program analyses (approximating reachable states)
6. Termination analysis (approximating state transitions)
7. Conclusions

ONE-SIDED PROGRAM ANALYSES (FLOW ANALYSIS, ABSTRACT INTERPRETATION)

Semantic and logical aspects of a computational practice.

- Practice: decades of practical experience in writing compilers
  (though correctness is rarely addressed by compiler hackers!)
- Data flow analysis: informal, pragmatic, ad hoc methods from the 1950s.
- Engineering methodology: program analysis by fix-point computations.

Theory: Semantics-based program analysis

- Formally based in program semantics.
  Cousot, Hankin, Jones, Muchnick, Nielson, many others.
- Research since 1970's under the name of Abstract Interpretation
- January 2008 conference in San Francisco:
  "30 Years of Abstract Interpretation."

TOWARDS UNDERSTANDING THE PROBLEM

Consider the code elimination transformation

$$[\texttt{x := a}]^{\ell} \Rightarrow [\texttt{skip}]^{\ell}$$

(It sounds trivial, but it's significant in practice!)

Semantic reasons that make it valid: **(control flow and data flow)**

1. ℓ **is unreachable: $\exists$ no control flow from the program's start to** $[\texttt{x := a}]^{\ell}$
2. ℓ **is dead: $\exists$ no control flow from** $[\texttt{x := a}]^{\ell}$ **to the program's end. E.g.,**
   - **The program will definitely loop after point ℓ. Or**
   - **The program will definitely abort execution after point ℓ.**
3. $\texttt{x}$ **is dead at ℓ: the value of** $\texttt{x}$ **is never used again.**
4. $\texttt{x}$ **is already equal to** $\texttt{a}$ **(if control ever gets to ℓ)**
5. $\texttt{a}$ **is an uninitialised variable: the value of** $\texttt{x}$ **is completely undependable**

ALAS, MOST OF THESE REASONS ARE AS UNDECIDABLE AS THE HALTING PROBLEM (!)

Remark: many (all!) of the above program behavior properties are **undecidable** (if you insist on exact answers).

Proof Rice's Theorem from Computability Theory.

So what do we do?

An answer: allow "one-sided" errors (in practice of **program analysis** and theory of **abstract interpretation**).

- Find **safe** descriptions of program behavior. Meaning of safety:
  - if **the analysis says that a program has a certain behavior**
    (e.g., that x is dead at point ℓ),
  - then it **definitely has that behavior** in all computations.
- Allow the analysis to be imprecise, i.e., **"one-sided"** :
  the analysis can answer "don't know" even when the property is true

  (this is the trick to gain decidablility; it can be misused)

"ONE-SIDED" REASONING TO DISCOVER PROGRAM PROPERTIES

"Program-point-centric" analysis: approximate the control flow or data flow at each program point ℓ.

The flow properties at a program point ℓ are determined by

- the time dimension:
  - the computational past (of computations that get as far as ℓ); or
  - the computational future (of computations after ℓ)
- the path modality:
  - a property of all computation paths from (or to) ℓ, or by
  - a property of at least one computation path from (or to) ℓ

Does this look familiar? The practitioners' methods in effect achieve

- applied temporal logic
- on finite models

WHAT AND HOW

What: program transformation to improve efficiency

- Based on program flow analysis
- Must be correct. Semantic question: what does this mean?

  Study object: the space of all run-time states
- Important: efficiency, complex hardware, human limits, etc
- Semantically subtle

How: several steps in program optimisation. First: program analysis.

- Choose a data flow lattice to describe program properties:

  one-sided finite descriptions of run-time state sets
- Build a system of data flow equations from the program:

  time dimension = future/past, modality = may/must.
- Solve the system of data flow equations

Then transform the program, usually to optimise it

STATE-BASED PROGRAM ANALYSIS

An example: for every program point ℓ, over-approximate

$$Reach(\ell) = \{\sigma \mid \text{some initial state } (\ell_0, \sigma_0) \text{ can reach } (\ell, \sigma)\}$$

Program analysis

- **must be finitely (and feasibly!) computable**
- **is computed uniformly for all points ℓ in the given program.**
- **is a mass act: applied automatically to any input program**

  (in contrast to: one-program-at-a-time verification)
- **Adjacent program points will have properties that are related, e.g., by classic flow equations of dataflow analysis for compiler construction.**

An analogy: heat flow equations as in Physics.

(ALTHOUGH. . . heat flows 2-ways, but program flows are asymmetric.)

OUTLINE, PROGRESS

1. Programs
2. Models in First Order Logic (. . . and programs. . .)
3. But . . . Aren't all nontrivial questions about real programs undecidable?
4. Subrecursive programming languages (life without CONS)
5. One-sided program analyses (approximating reachable states)
6. • Termination analysis (approximating state transitions)
7. Conclusions

TERMINATION ANALYSIS

Aim: prove that program p will terminate on all possible runs.

This is

- not a property of the states (ℓ, σ) reachable at program point ℓ, but
- rather a property of the state transitions from a program point ℓ to its successors

Definition p terminates iff there does not exist an infinite transition sequence

$$(\ell_0, \sigma_0) \rightarrow (\ell_1, \sigma_1) \rightarrow \ldots \rightarrow (\ell_i, \sigma_i) \rightarrow \ldots$$

Flow analysis approach: over-approximate, for each program point ℓ, ℓ', the set

$$(\ell, \sigma) \rightarrow (\ell', \sigma')$$

What is new here?

Focus not on "one-state-at-a-time" (as for flow analysis). Focus is now on relations between states, between one state and its successor.

SIZE-CHANGE TERMINATION ANALYSIS

Size-Change Termination criterion:

- Methods, viewpoint: from automata theory and graph theory
- Size-change Termination can serve as a simple common core of many termination analyses
- Challenge: reduce need for human creativity, e.g., designing and finding lexicographic orders, polynomial interpretations, etc.
- Early motivation from partial evaluation: a binding-time analysis sufficient to guarantee that program specialization will stop

A one-sided analysis:

- Any size-change terminating program is terminating
- Some terminating programs may not be size-change terminating

. . . so we're not solving the halting problem!

OVERVIEW OF METHODS, RESULTS

1. **Assume:**
   - **Set** $Value$ **has a well-founded order** $>$
   - **Size changes known from program operations, e.g.,**

     `cons(X,Y) > X,` `cons(X,Y) > Y,` `X > head(X)`

2. **For each transition** ℓ **in program** p**, obtain a "safe" size-change graph** G_ℓ
3. **Program** p **is size-change terminating if . . .**
   - **(a property of the graphs** G_ℓ **that implies)**
   - **∃ no infinite transition sequence in any computation.**
4. **Algorithms to test the graph property:**

   (a) Compute closure of a set of graphs **or, an alternative:**

   (b) Operations on Büchi automata (test $\subseteq$**);**
5. **Upper and lower complexity bounds: both** PSPACE.

APPROACH TO TERMINATION ANALYSIS

1. **Consider all traces (finite or infinite transition sequences) that might occur in actual computations**
2. **Identify as Bad traces: all those of infinite length**
3. **Identify as Dead traces: all those that are impossible because they would cause infinite descent**

The Size-change principle: program `p` terminates if

Every infinite transition sequence would cause an infinite descent in at least one value

In other words: every "bad" trace is a "dead" trace.

Decide using regular (finite-state) approximations to Bad, Dead.

SIZE-CHANGE GRAPHS

Consider a program transition

```
f(x1,..,xi,..xn)     = ...[ℓ:] g(e1,..,ej,..em)...
g(y1,...,yj,...,ym)  = ...
```

Size-change graph G_ℓ for the transition ℓ : f $\longrightarrow$ g has a labeled arc

- **from** f **parameter** xi **to** g **parameter** yj
  if xi **is used to compute** ej**.**
- **Arc label: how is** xi **value related to** ej **value?**

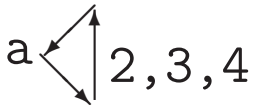
G_ℓ must safely approximate size relations:

- xi $\overset{\downarrow}{\longrightarrow}$ yj **implies** xi**'s value is greater than** ej**'s new value**
- xi $\overset{\mp}{\longrightarrow}$ yj**: same, but "greater than or equal"**

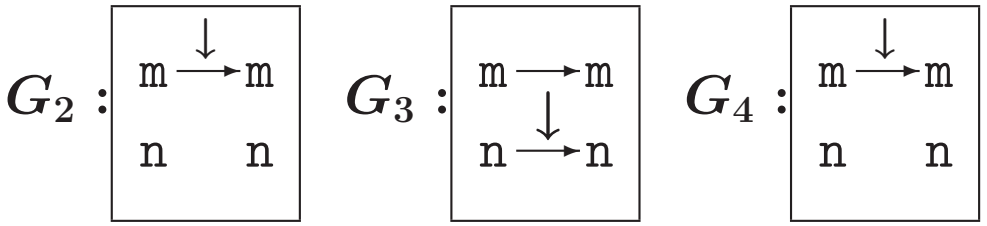
ACKERMANN: A NONLINEAR PROGRAM

```
a(m,n) = if m=0 then n+1 else
         if n=0 then [2]:a(m-1,1) else
            [4]:a(m-1, [3]:a(m,n-1))
```

Transition graph

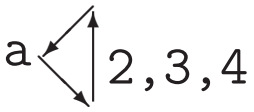


Size-change graphs

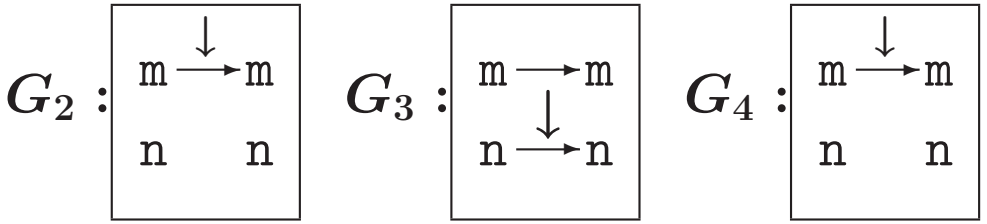


SIZE-CHANGE TERMINATION REASONING:

Transition graph



Size-change graphs



Consider any infinite transition sequence:

$$\pi \in (2 + 3 + 4)^{\omega}$$

- **If $\pi = \ldots 3^{\omega}$, then n descends infinitely.**
- **Otherwise π has infinitely many 2's or 4's; so m descends infinitely.**

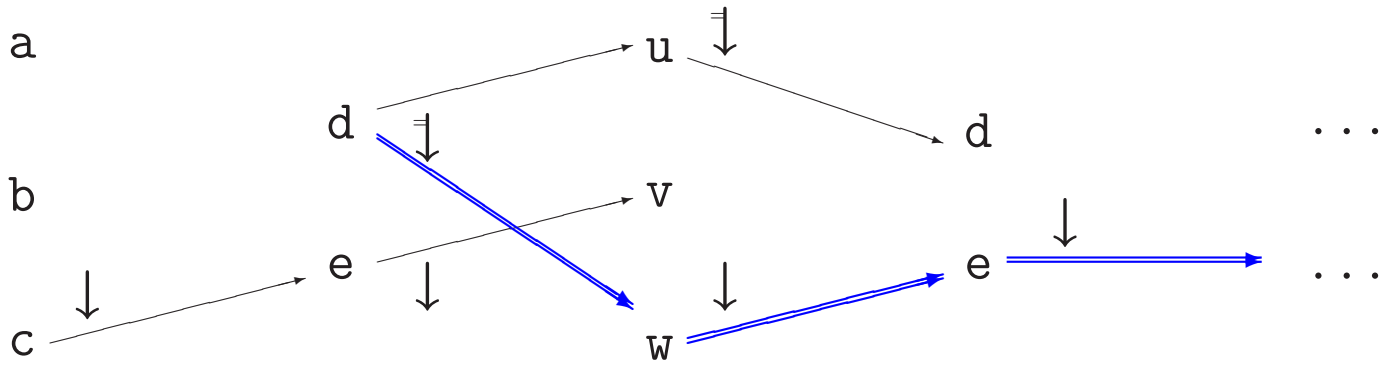
There is no obvious bound on length of a descending chain.

(No big surprise, since Ackermann's function isn't primitive recursive!)

THE MULTIPATH OF A TRANSITION SEQUENCE

Definition: The **multipath** $\mathcal{M}(\pi)$ of transition sequence $\pi = \tau_1\tau_2\ldots$ is:

$\mathcal{M}(\tau_1\tau_2\ldots) =$ concatenation of G_{τ_1}, G_{τ_2},...



$$\mathcal{M}(\tau_1\tau_2\ldots) = \boxed{G_{\tau_1} : \mathtt{f} \to \mathtt{g}}\boxed{G_{\tau_2} : \mathtt{g} \to \mathtt{h}}\boxed{G_{\tau_3} : \mathtt{h} \to \mathtt{g}}\boxed{G_{\tau_4} : \mathtt{g} \to \mathtt{k}}\ \ldots$$

Definition:

1. A **thread** in multipath $\mathcal{M}(cs)$ is a connected sequence of labeled arcs.
2. The thread is of **infinite descent** if its arc label sequence contains infinitely many $\downarrow$.

THE SIZE-CHANGE TERMINATION CRITERION

Recall: **Program** p **is size-change terminating if** $\mathrm{Bad} \subseteq \mathrm{Dead}$, **where**

$$\mathrm{Bad} = \{\pi \in C^\omega \mid \pi \text{ follows } \mathrm{p}\text{'s flow chart } \}$$

$$\mathrm{Dead} = \{\pi \in C^\omega \mid \mathcal{M}(\pi) \text{ has a thread with infinite } \downarrow\}$$

Algorithms:

1. **Graph calculation used in practice:**

   Compute and test the closure of the set of size-change graphs.

   Theorem p **is size-change terminating iff every idempotent graph in the closure has an arc** $\mathrm{x} \overset{\downarrow}{\longrightarrow} \mathrm{x}$.

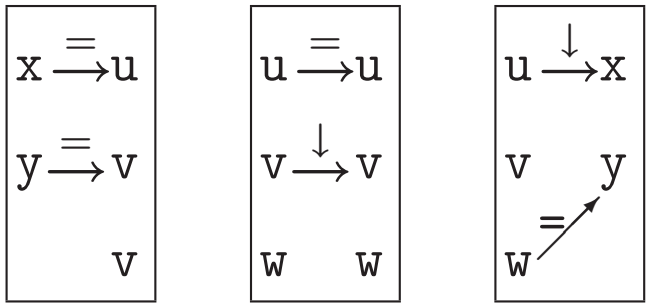
2. **Asymptotically better, practicality as yet unclear:**

   Büchi automaton algorithms (Sasha, Vardi, Fogarty).

TERMINATION BY GRAPH CALCULATION

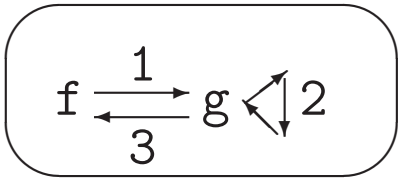
```
f(x,y)   = if x = 0 then y else 1: g(x,y,0)
g(u,v,w) = if w = 0 then 3: f(u-1,w) else 2: g(u,v-1,w+2)
```

Size-change graphs $\mathcal{G}$

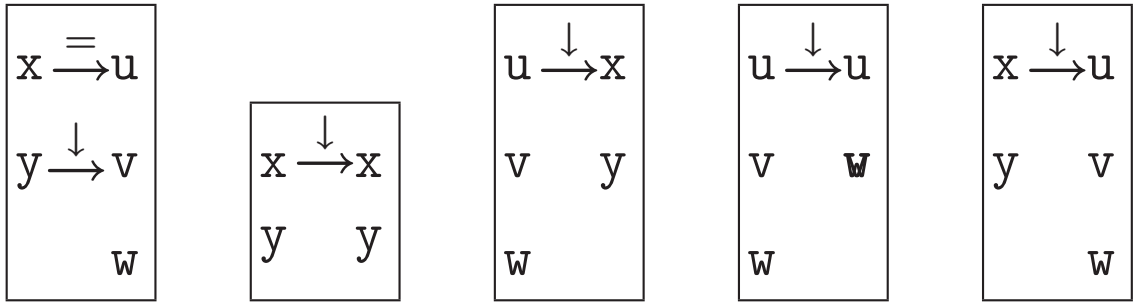


G_1 $*G_2$ G_3

Transition graph



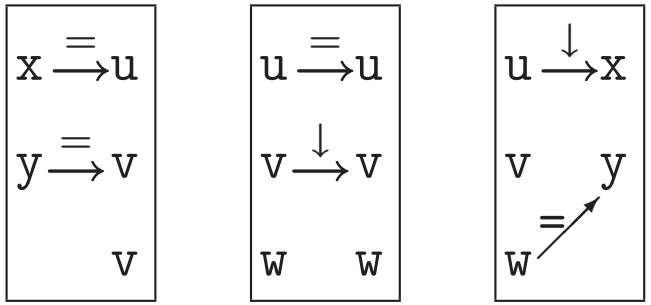
Closure: $\{G_1, G_2, G_3, G_{12}, G_{13}, G_{23}, G_{31}, G_{131}\}$ **(all reachable data flows)**



G_{12} $*G_{13}$ G_{23} G_{31} G_{131}

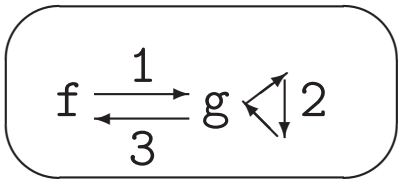
TERMINATION BY GRAPH CALCULATION II

Size-change graphs $\mathcal{G}$



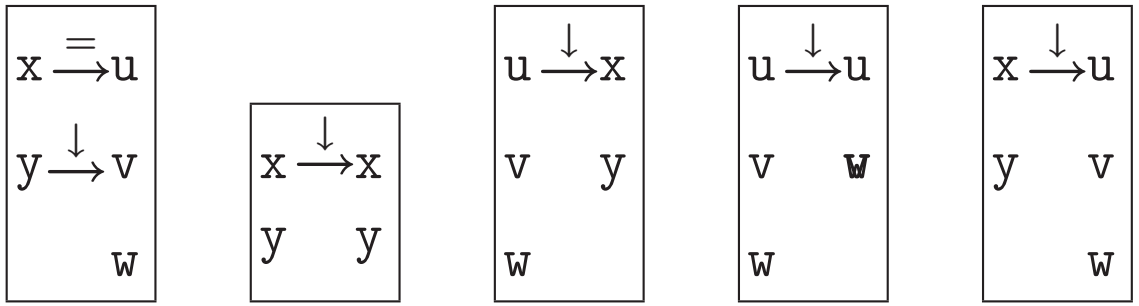
G_1 $*G_2$ G_3

Transition graph



$G_2 = G_2; G_2$, so
G_2 is idempotent.

Closure: $\{G_1, G_2, G_3, G_{12}, G_{13}, G_{23}, G_{31}, G_{131}\}$ **(all reachable data flows)**



G_{12} $*G_{13}$ G_{23} G_{31} G_{131}

G_{13} is idempotent

Idempotent graphs G_2, G_{13} have decreasing variables, so no infinite traces are possible. **Therefore program p terminates on all inputs.**

MORE ABOUT TERMINATION ANALYSIS

- **Ramsey's theorem** is the key to prove correctness of the closure method.
- **Worst-case behavior:** Size-change termination is complete for PSPACE.

Related work (far from all...)

1. Early functional: Abel and Altenkirch
2. Damien Sereni's 2006 Ph.D. thesis: higher-order functions
3. Term rewriting: Giesl, Arts, many others
4. Logic programming: Codish, Lindenstrauss, Plümer, Sagiv, Taboch, ...
5. Chin Soon Lee's 2002 Ph.D. thesis:
   - Application to partial evaluation
   - Program analysis in PTIME (weaker but strong enough in practice)

OUTLINE, PROGRESS

1. Programs
2. Models in First Order Logic (. . . and programs. . .)
3. But . . . Aren't all nontrivial questions about real programs undecidable?
4. Subrecursive programming languages (life without CONS)
5. One-sided program analyses (flow analysis, abstract interpretation)
6. Termination analysis (approximating state transitions)
7. • Conclusions

CONCLUSIONS

Overviewed in this talk:

- **A model in FOL resembles a program's runtime state space**
- **Relate unsolved problems to each other, e.g.:**

$$\text{RECURSION} \supsetneq \text{ITERATION} \text{ if and only if } \text{PTIME} \supsetneq \text{LOGSPACE}$$

- **Connections between practice (compilers' flow analysis) and theory (abstract interpretation, model checking)**
- **The halting problem can be dealt with, if not completely solved**

More generally:

- **Properties of a program's runtime state space are fascinating**
- **There are many connections, some unexpected, with theoretical and applied Computer Science, and Logic too (spectra, temporal logic, . . .)**
- **Don't give up, even on undecidable problems!**

5 RELEVANT PAPERS

References

[1] Neil D. Jones. The expressive power of higher-order types or, life without cons. J. Funct. Program., 11(1):5–94, 2001.

[2] Neil D. Jones and Alan L. Selman. Turing machines and the spectra of first-order formulas with equality. ACM Symp. Theory of Computing, pp.157–167, 1972. Journal of Symbolic Logic, 39:139–150, 1974.

[3] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In Handbook of Logic in Computer Science. Oxford University Press, 1994. 527–629.

[4] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Compiler optimization correctness by temporal logic. Higher-Order and Symbolic Computation, 17(3):173–206, 2004.

[5] Chin Soon Lee, Neil D. Jones, Amir M. Ben-Amram. The size-change principle for program termination. ACM POPL, pages 81–92, 2001.