Correctness, Compositionality, Concurrency
Facets of formal program analysis

Martin Steffen

Autumn 2017

Program analysis

"process of automatically analyzing the behavior of computer programs"

- all kinds of beneficial effects. . .

Program analysis

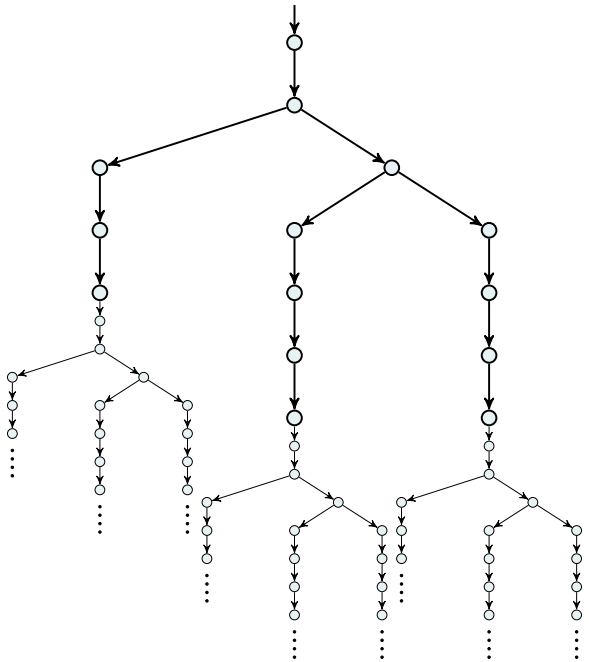
"process of automatically analyzing the behavior of computer programs"

- all kinds of beneficial effects. . .

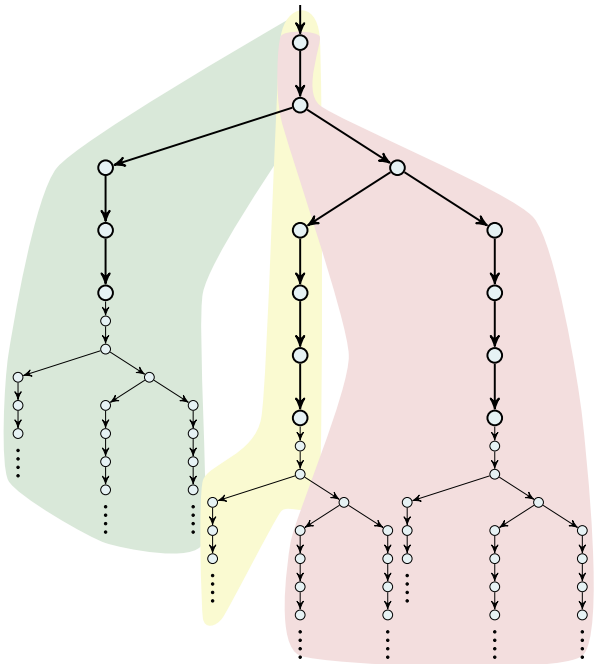
Rice's theorem

All non-trivial, semantic properties of programs are undecidable.

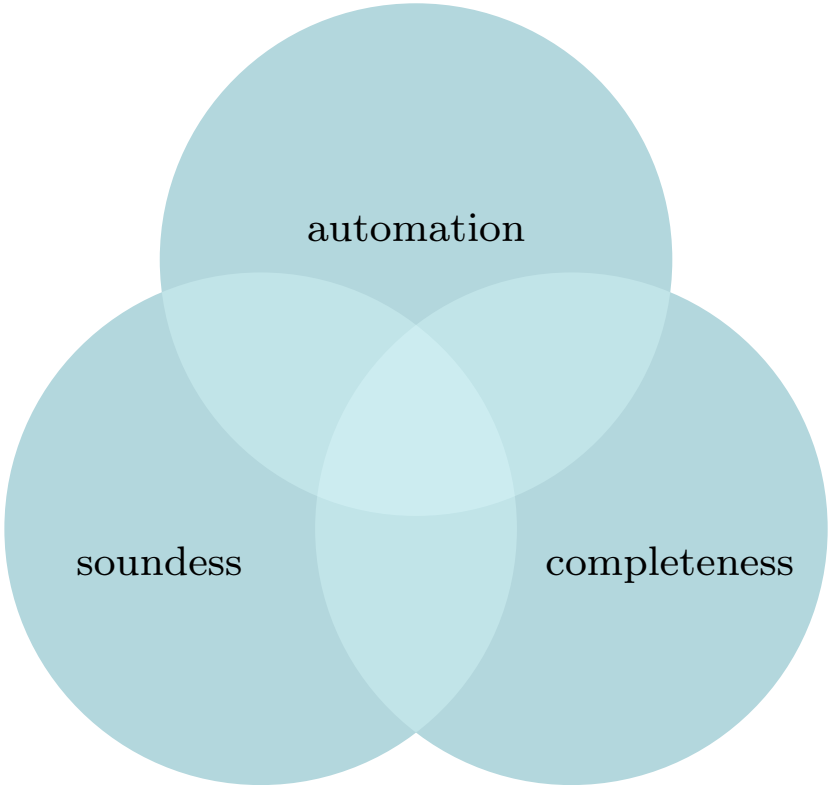
The good, the bad, and the ones we can't figure out



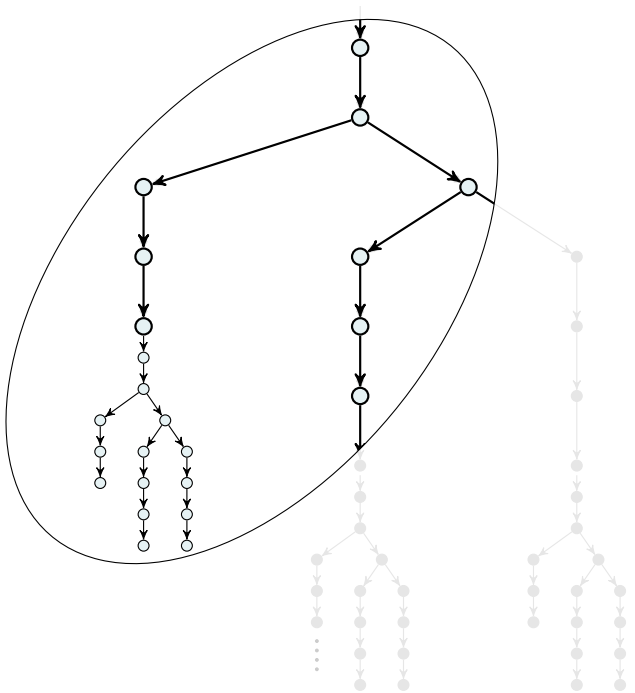
The good, the bad, and the ones we can't figure out

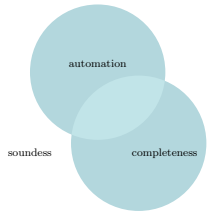


Two out of three ain't bad



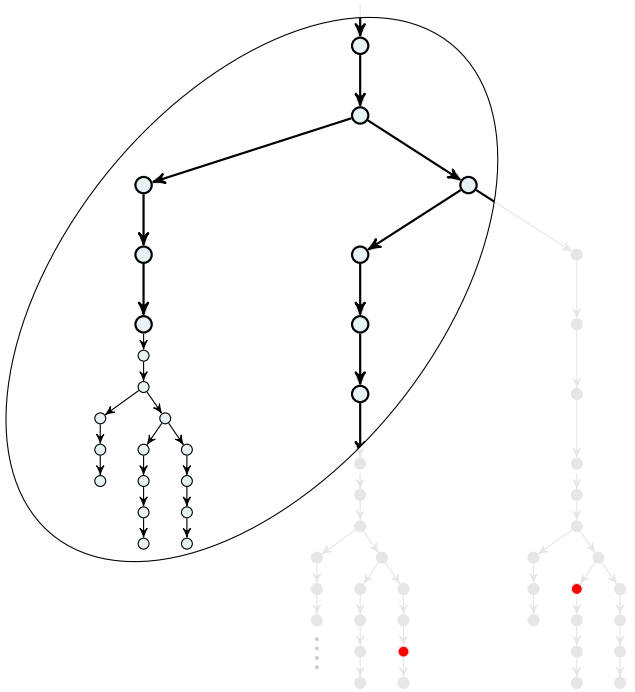
Sacrifice soundness / underapproximation

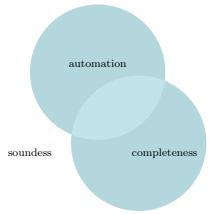




- testing (and “testing can be formal too”)
- run-time verification

Sacrifice soundness / underapproximation



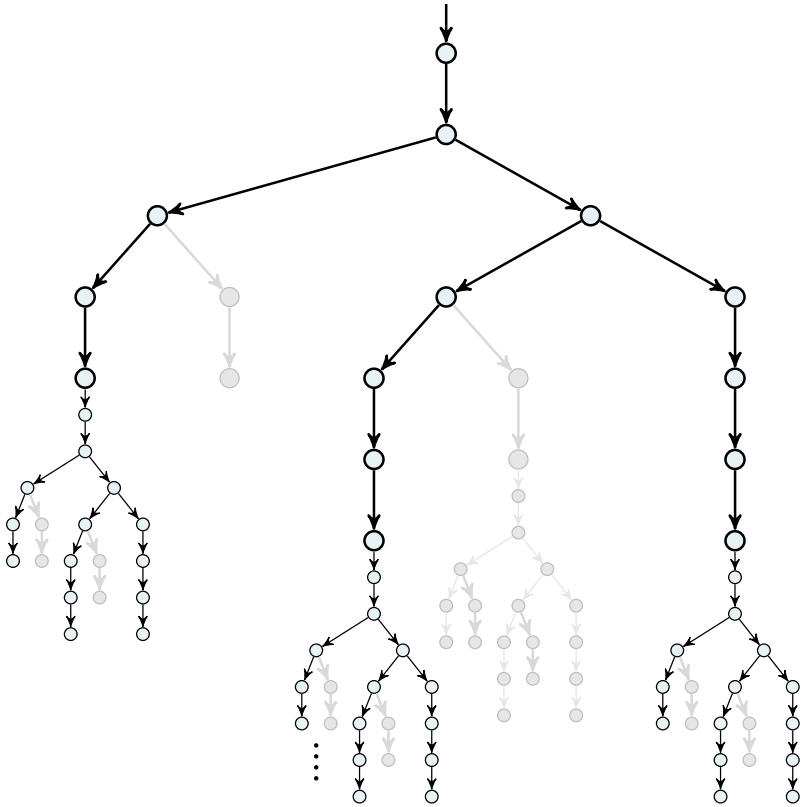


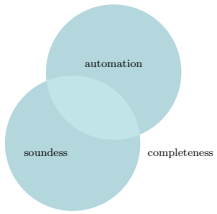
- testing (and “testing can be formal too”)
- run-time verification

Of course: one can do better than just that...

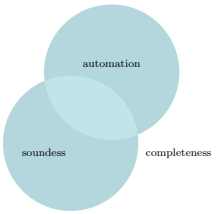
- avoiding redundant exploration (POR)
- prioritizing
  - preemption bounding
- symbolic execution
- combination with (other) static analysis

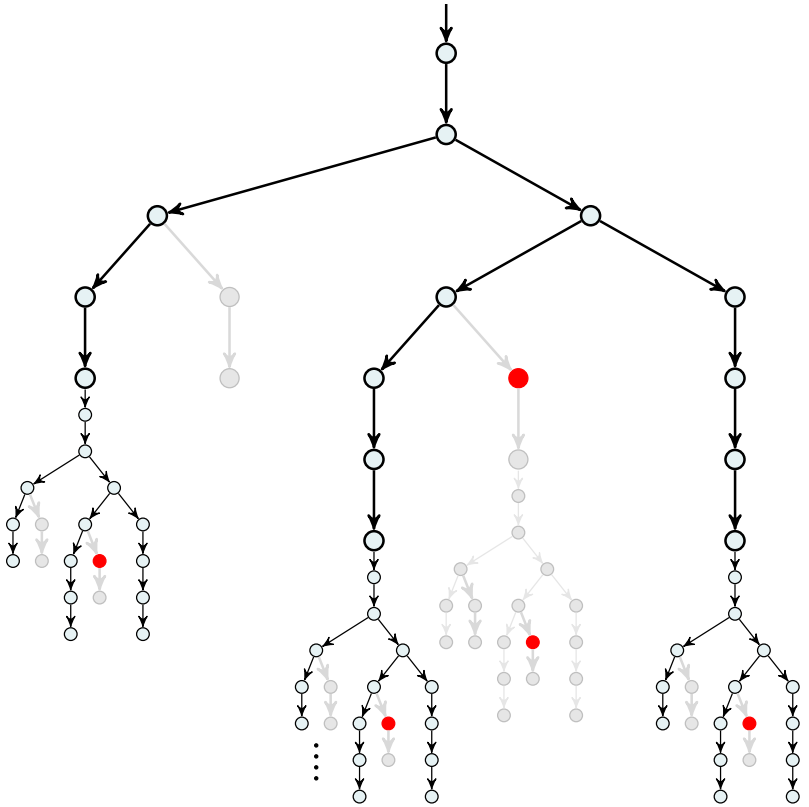
Giving up on completeness





Giving up on completeness



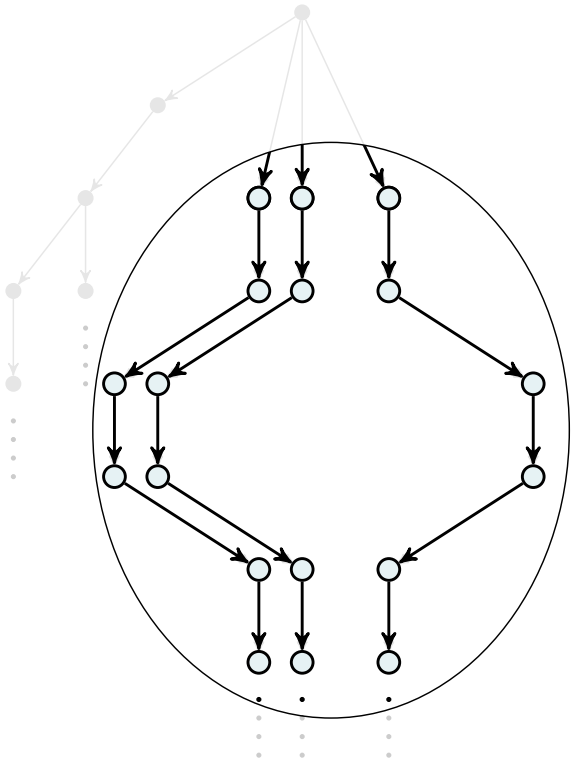


false positives

But how?

```
if x=y then skip else x:=x+1
```

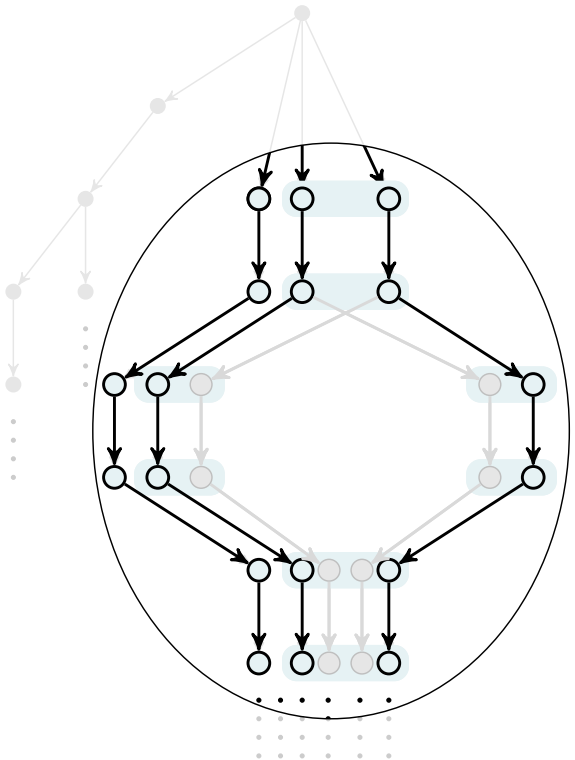
Assume $(x, y) \in \{(0,0), (1,1), (0,1)\}$



But how?

```
if x=y then skip else x:=x+1
```

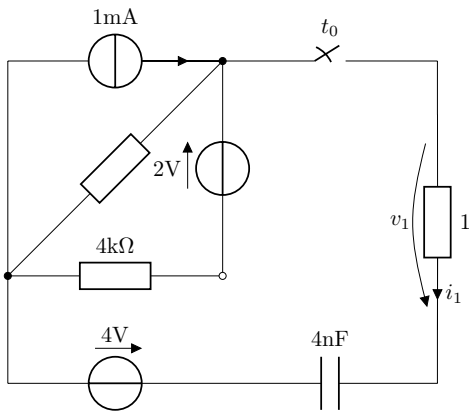
Assume $(x, y) \in \{(0,0), \{(1,1),(0,1)\}\}$

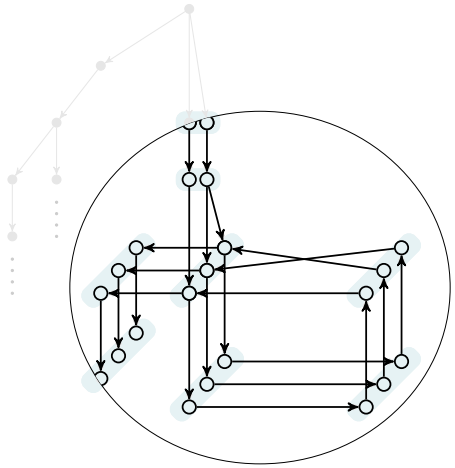


Abstraction

- abstraction ⇒ overapproximation ⇒ false positives
- lumping different “elements” together (values, program points . . .)
- *symbolic* representation
- abstract interpretation
- data flow analysis

Data flow analysis





- abstraction: “Ampere” . . .
- circuit laws (Kirchhoff)
- *stationary* solution

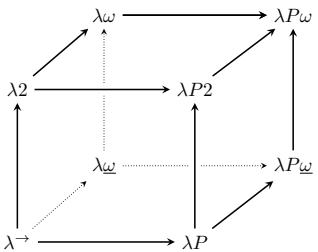
- abstraction: *sets* of values . . .
- *data flow constraints*, transfer functions
- fixpoint (μ or ν)

What's a type?

What's a type?

A **union** is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

What's a type?



- directive for memory allocation
- an object in a category? homotopy?
- an *abstraction?*
- formula in a HO constructive logic?

Types, flows, and effects

$$t : T$$

$$t : \texttt{Int}$$

What?

if t terminates, `Int`-value

Types, flows, and effects

$$t : T^{++}$$

What?	From where?
if t terminates, `Int`-value	data flow

Types, flows, and effects

$$t : T^{++} :: \varphi$$

What?	From where?	During?
if t terminates, `Int`-value	data flow	effects, *while* executing

- all of it: more or less approximative

Type systems & type checking

$$\Gamma \vdash t : T$$

- derivation systems

$$\dfrac{\dfrac{\dfrac{A, B \vdash A \qquad A, B \vdash B}{A, B \vdash A \wedge B}}{A \vdash B \to A \wedge B}}{\vdash A \to B \to A \wedge B}$$

Type systems & type checking

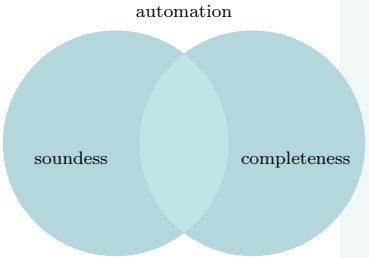
$$\Gamma \vdash t : T$$

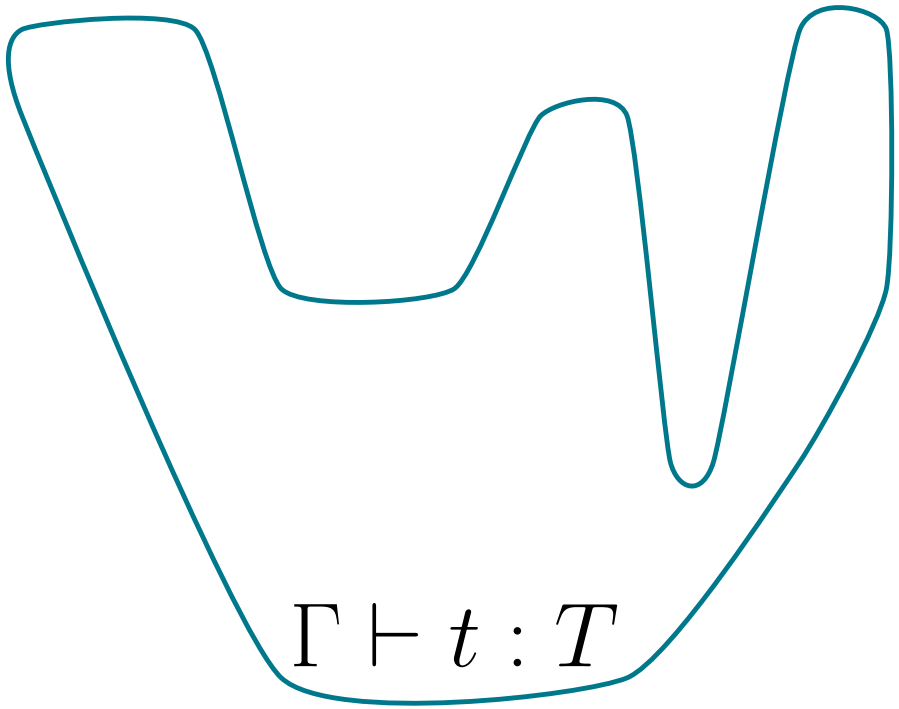
- derivation systems

$$\dfrac{\dfrac{\dfrac{x_1{:}T_1, x_2{:}T_2 \vdash x_1 : T_1 \qquad x_1{:}T_1, x_2{:}T_2 \vdash x_2 : T_2}{x_1{:}T_1, x_2{:}T_2 \vdash (x_1, x_2) : T_1 \times T_2}}{x_1{:}T_1 \vdash \lambda x_2{:}T_2.(x_1, x_2) : T_2 \to T_1 \times T_2}}{\vdash \lambda x_1{:}T_1.\lambda x_2{:}T_2.(x_1, x_2) : T_1 \to T_2 \to T_1 \times T_2}$$

Type checking

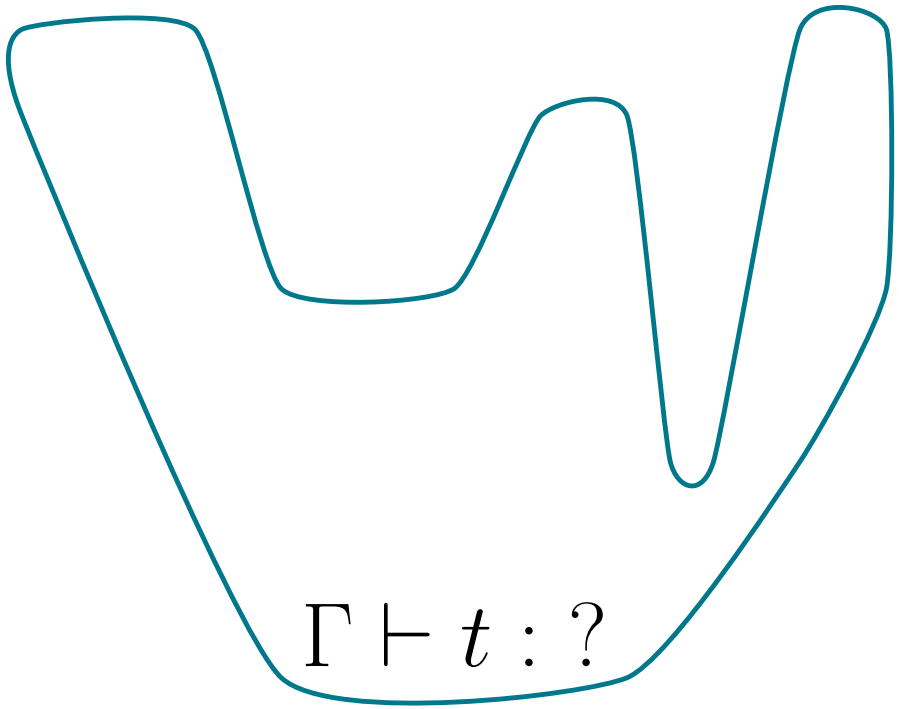
$$\Gamma \vdash t : T$$





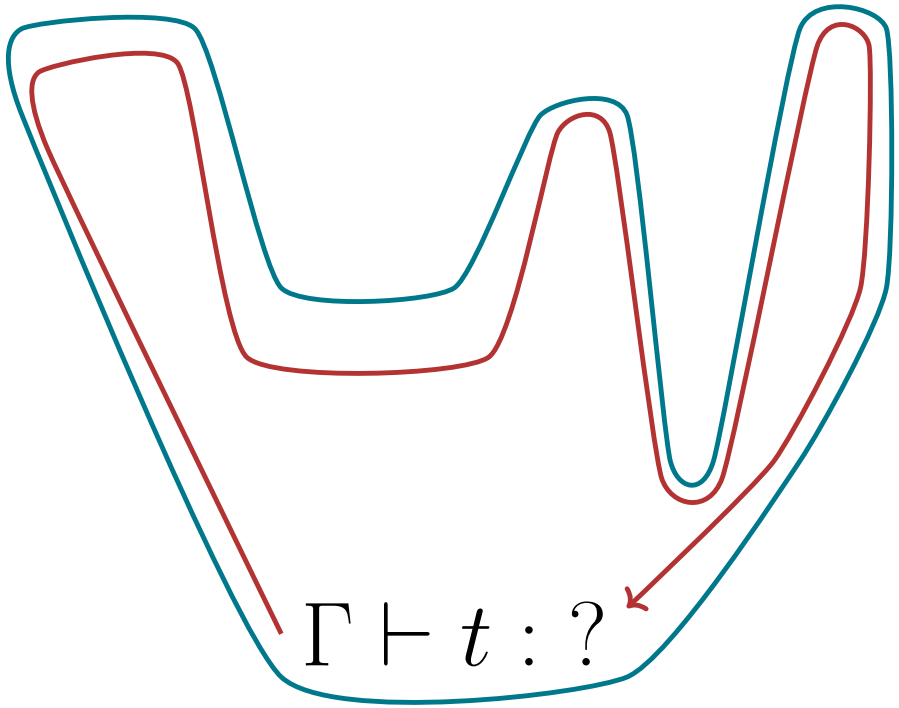
Type inference

$$\Gamma \vdash t : ?$$

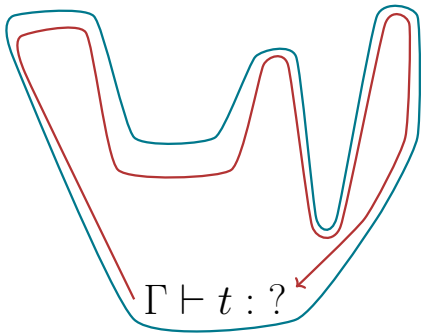


Type inference

$$\Gamma \vdash t : ?$$

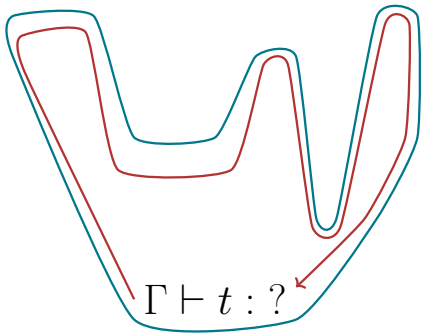


Type inference

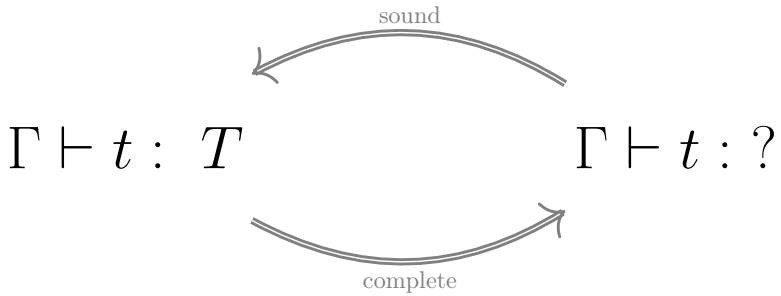


- *unification*
- decidability?
- cf. synthesized and inherited attributes

Type inference



- *unification*
- decidability?
- cf. synthesized and inherited attributes



And for flows and effects?

$$\hat{\Gamma} \vdash t : \hat{T} :: \varphi$$

And for flows and effects?

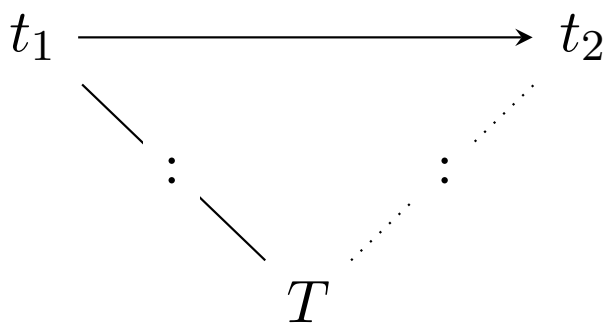
$$C;\ \hat{\Gamma} \vdash t : \hat{T} :: \varphi$$

- adding constraints
- for flows: *simple* constraints

Correctness

Milner's dictum ("type safety" / "static typing")

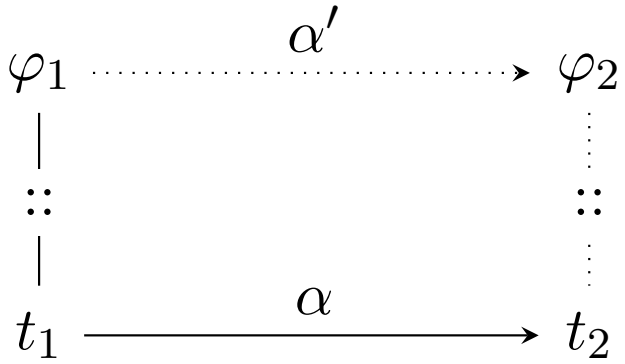
Well-typed programs cannot go wrong!



Correctness

Milner's dictum ("type safety" / "static typing")

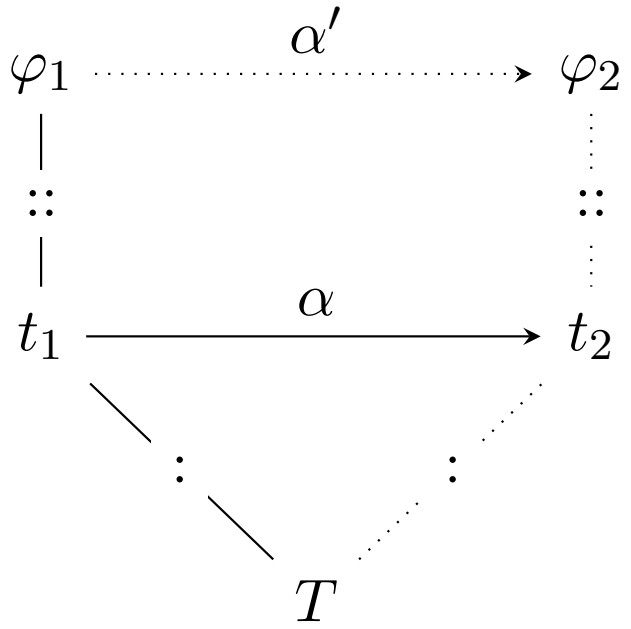
Well-typed programs cannot go wrong!



Correctness

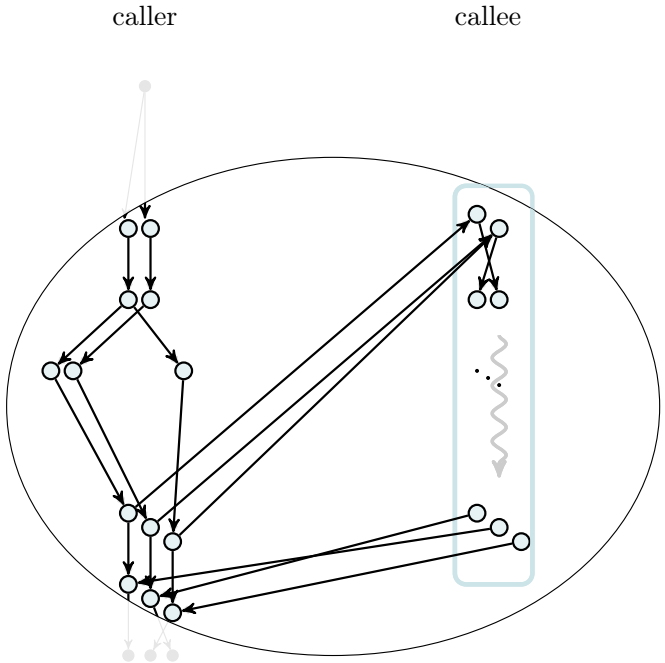
Milner's dictum ("type safety" / "static typing")

Well-typed programs cannot go wrong!



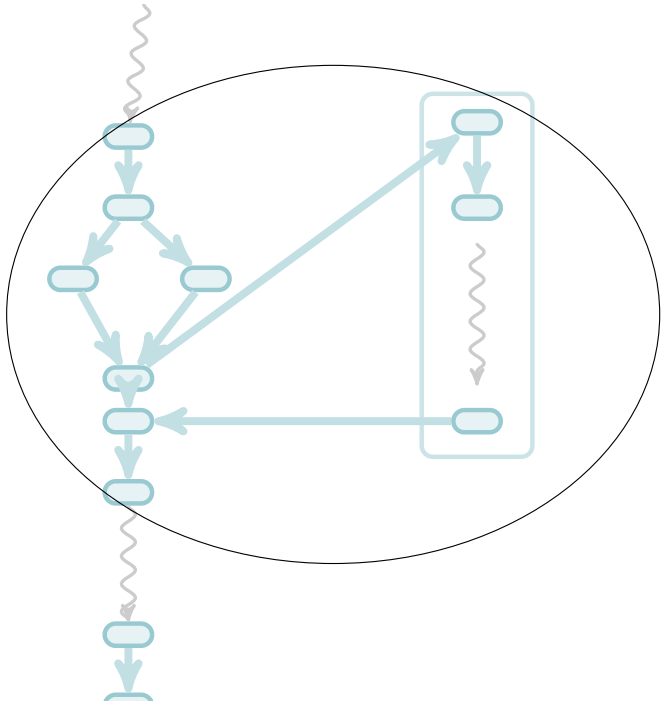
Context-sensitivity

- treat function calls “properly” (= dependent on call-site)



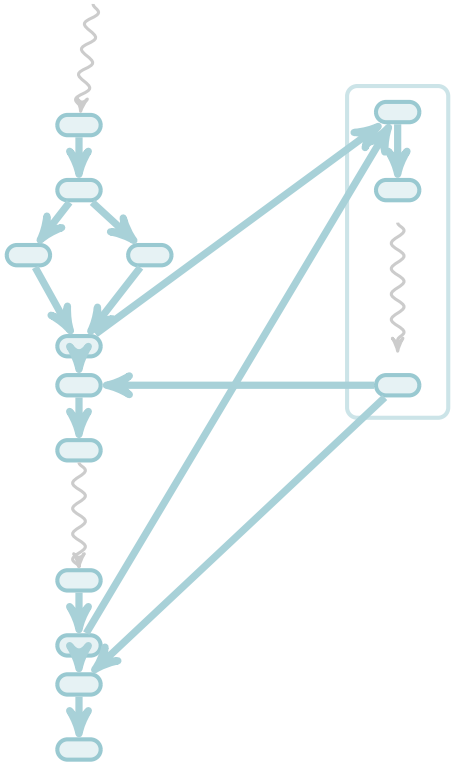
Context-sensitivity

- treat function calls “properly” (= dependent on call-site)



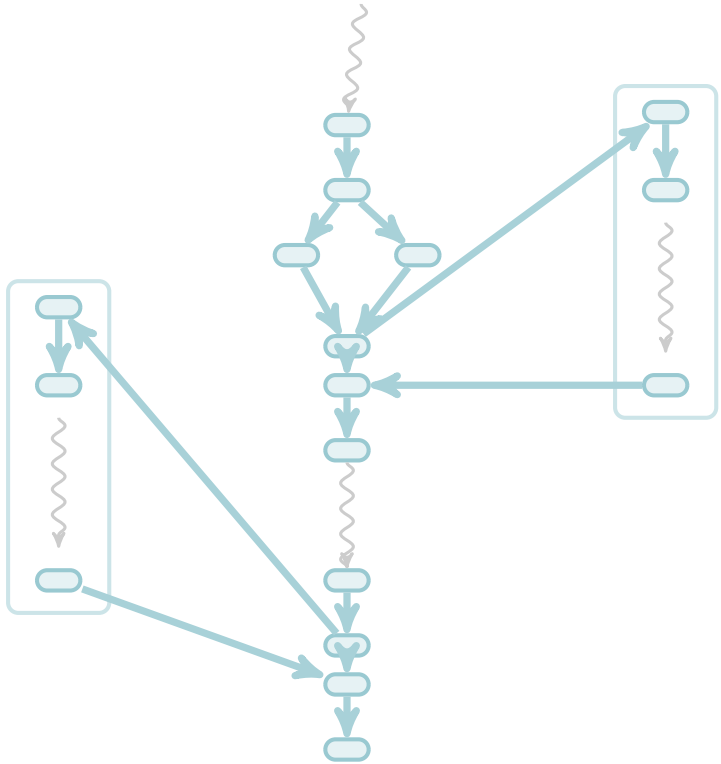
Context-sensitivity

- treat function calls “properly” (= dependent on call-site)



Context-sensitivity

- treat function calls “properly” (= dependent on call-site)



Compositional accounts and "polymophism"

- context-sensitive:function analysis distinguishing different call-sites

constrained universally quantified types

$$\forall Y \quad .\hat{T}$$

cf.

- *type schemes* in ML-polymorphmism and
- bounded quantification $\forall Y \leq T_1.T_2$ in $F_{\leq}$ ...

Compositional accounts and "polymophism"

- context-sensitive:function analysis distinguishing different call-sites

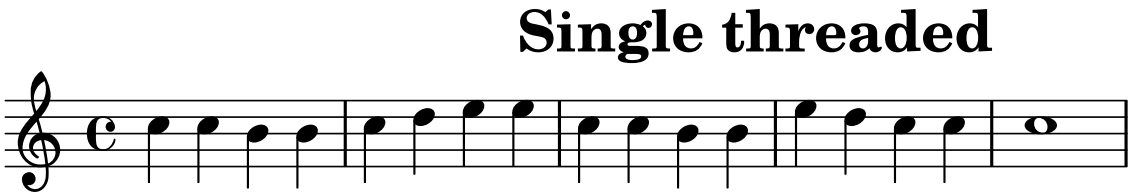
constrained universally quantified types

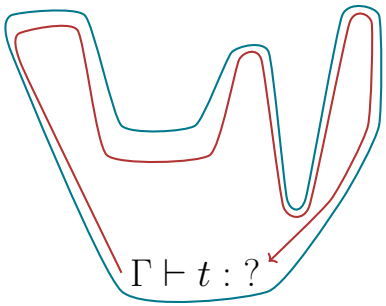
$$\forall Y : C.\hat{T}$$

cf.

- *type schemes* in ML-polymorphmism and
- bounded quantification $\forall Y \leq T_1.T_2$ in $F_{\leq}$ ...

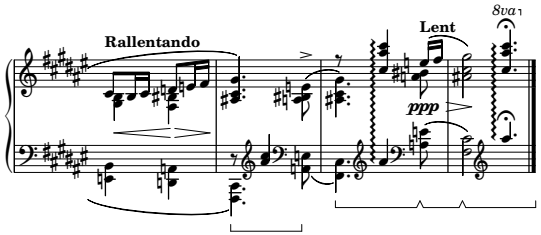
Program analysis





- types: basically *single threaded*
- structural analysis of *program code*

Concurrency and analysis



New kinds of errors

- races
- deadlocks
- starvation
- ...

Analyses get more hairy

- reproducability ("Heisenbugs")
- interference vs. isolation
- state space explosion problem

Illustration: Deadlock analysis

- given: multi-threaded calculus
- types and effects for lock interaction

$Y ::= \varrho \mid X$	type-level variables
$r ::= \varrho \mid \{\pi\} \mid r \sqcup r$	lock/label sets
$\hat{T} ::= B \mid \mathtt{L}^r \mid \hat{T} \xrightarrow{\varphi} \hat{T}$	types
$\hat{\sigma} ::= \forall \vec{Y}{:}C.\ \hat{T} \xrightarrow{\varphi} \hat{T} \mid \hat{T}$	type schemes
$C ::= \emptyset \mid \varrho \sqsupseteq r, C \mid \epsilon \sqsupseteq \varphi, C$	simple constraints

Deadlocks (2)

- two level approach

  local type and effect system

  global state exploration (à la model checking)

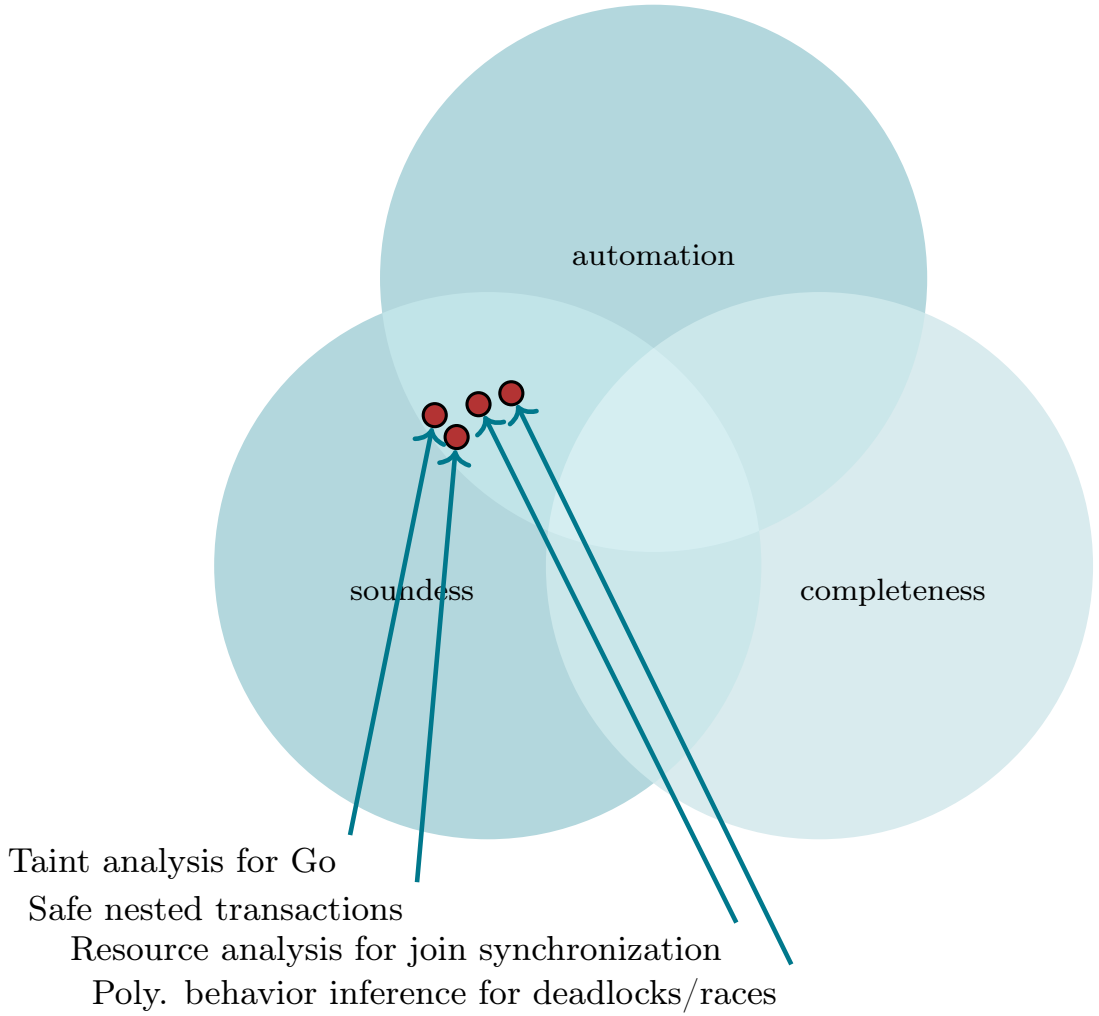
- flows: tracing lock instances (rudimentary *alias* analysis)

Processes as effects

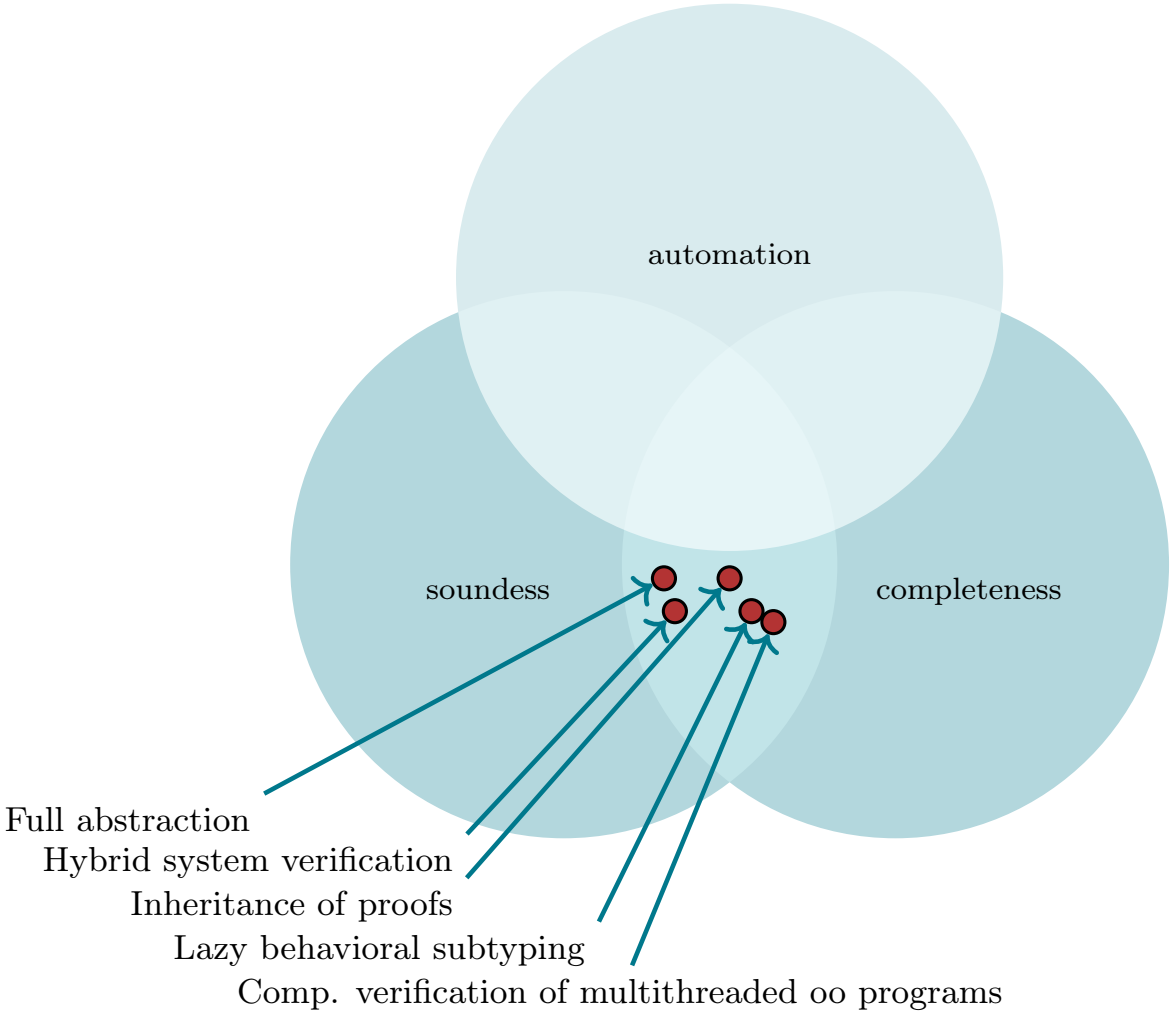
Abstracting approximative lock interaction of one thread into an"abstract process" (as in process algebra)

- correctness: deadlock-sensitive simulation
- further abstractions: bounded call stack

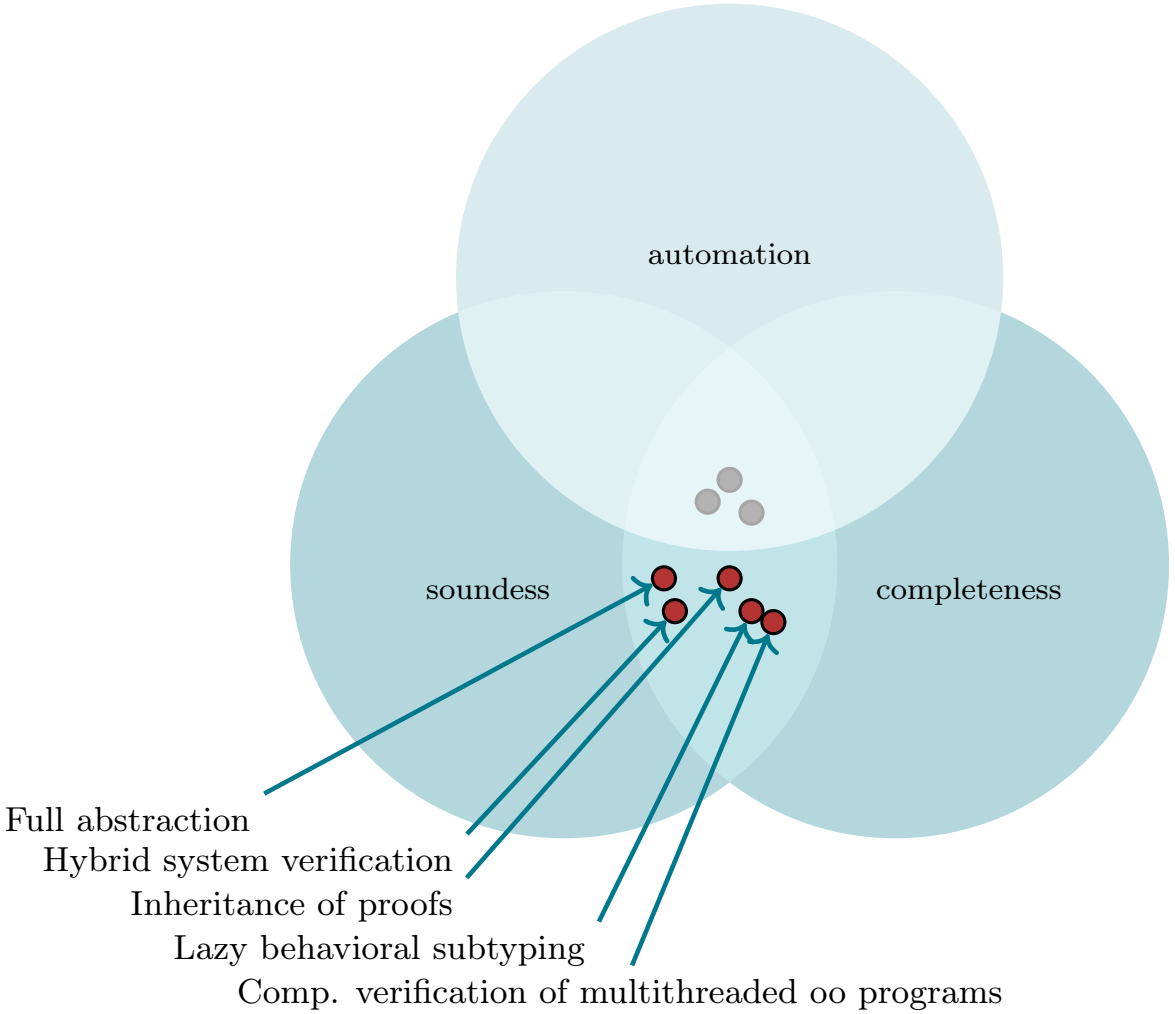
Type & effect systems



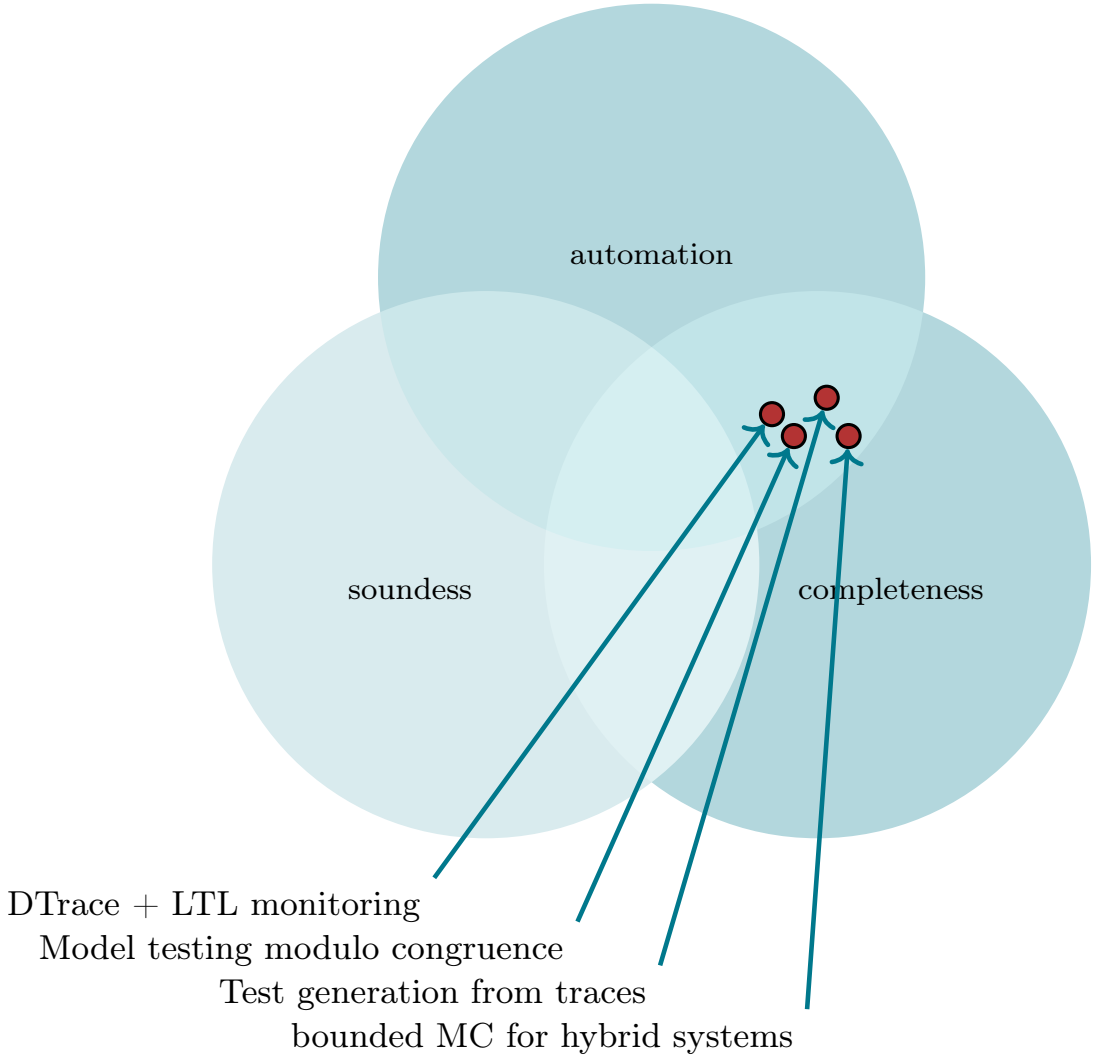
Proof systems for program verification



Proof systems for program verification



Run-time veryification, testing, BMC



Miscellaneous

- railway design verification (with datalog)
- full abstraction
- protocol verification (model checking, SDL) and engineering
- parametric model checking (rewriting theory, transducers)
- timed ambients & π-calculus (semantics, assumption-commitement type system for resources)
- semantics of weak memory models
- Petri-net semantics for concurrent actor language

Kudos

Slides were done with emacs *org-mode*, LaTeX, TikZ (and a bit Lilypond).

References I

Bibliography

[1] Ábrahám, E., de Boer, F. S., de Roever, W.-P., and Steffen, M. (2003). Inductive proof-outlines for monitors in Java. In Najm, E., Nestmann, U., and Stevens, P., editors, *FMOODS '03*, volume 2884 of *Lecture Notes in Computer Science*, pages 155–169 (15 pages). Springer Verlag. A longer version appeared as technical report TR-ST-03-1, April 2003.

[2] Ábrahám-Mumm, E., de Boer, F. S., de Roever, W.-P., and Steffen, M. (2002a). A tool-supported proof system for monitors in Java. In Bonsangue, M. M., de Boer, F. S., de Roever, W.-P., and Graf, S., editors, *FMCO 2002*, volume 2852 of *Lecture Notes in Computer Science*, pages 1–32 (33 pages). Springer Verlag.

[3] Ábrahám-Mumm, E., de Boer, F. S., de Roever, W.-P., and Steffen, M. (2002b). Verification for Java's reentrant multithreading concept. In Nielsen, M. and Engberg, U. H., editors, *Proceedings of FoSSaCS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 4–20 (17 pages). Springer Verlag. A longer version, including the proofs for soundness and completeness, appeared as Technical Report TR-ST-02-1, March 2002.

[4] Bodden, E., Pun, K. I., Steffen, M., Stolz, V., and Wickert, A.-K. (2016). Information flow analysis for Go. In Margaria, T. and Steffen, B., editors, *7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'16)*, volume 9952 of *Lecture Notes in Computer Science*, pages 431–445. Springer Verlag.

[5] de Boer, F. S., Bonsangue, M. M., Grüner, A., and Steffen, M. (2008). Automated test driver generation for Java components. In *Netherland's Testing Day "TestDag'08", Delft*. White Paper.

[6] de Boer, F. S., Bonsangue, M. M., Grüner, A., and Steffen, M. (2009). Java test driver generation from object-oriented interaction traces. *Electronic Notes in Theoretical Computer Science*, 243:33–47 (15 pages). Special issue for the Proceedings of the 2nd International Workshop on Harnessing Theories for Tool Support in Software TTSS'08, ICTAC 2008 satellite workshop, 30. August 2008, Istambul, Turkey.

[7] Dovland, J., Johnsen, E. B., Owe, O., and Steffen, M. (2008). Lazy behavioral subtyping. In Cuellar, J., Maibaum, T., and Sere, K., editors, *Proceedings of the 15th International Symposium on Formal Methods (FM'08)*, volume 5014 of *Lecture Notes in Computer Science*, pages 52–67 (16 pages). Springer Verlag.

References II

[8] Dovland, J., Johnsen, E. B., Owe, O., and Steffen, M. (2009). Incremental reasoning for multiple inheritance. In *Proceedings of the 7th International Conference on integrated Formal Methods (iFM'09), Düsseldorf, Germany, 16 – 19 February, 2009*, volume 5423 of *Lecture Notes in Computer Science*, pages 215–230. Springer Verlag.

[9] Dovland, J., Johnsen, E. B., Owe, O., and Steffen, M. (2010). Lazy behavioral subtyping. *Journal of Logic and Algebraic Programming*, 79(7):578–607. 30 pages. Article in Press, Preprint available at doi:10.1016/j.jlap.2010.07.008. Presented at the 20th Nordic Workshop on Programming Theory, NWPT '08, Tallinn. A shorter conference version appeared in the Proceedings of the 15th International Symposium on Formal Methods (FM'08), LNCS 5014.

[10] Dovland, J., Johnsen, E. B., Owe, O., and Steffen, M. (2011). Incremental reasoning with lazy behavioral subtyping for multiple inheritance. *Science of Computer Programming*, 76:915–941.

[11] Gaudel, M.-C. (1995). Testing can be formal, too. In Mosses, P. D., Nielsen, M., and Schwarzbach, M. I., editors, *Proceedings of TAPSOFT '95*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer Verlag.

[12] Hofmann, M., Naraschewski, W., Steffen, M., and Stroup, T. (1998). Inheritance of proofs. *Theory and Practice of Object Systems (Tapos), Special Issue on the Third Workshop on Foundations of Object-Oriented Languages (FOOL 3, LICS and Federated Logic Conferences Workshop), July 1996*, 4(1):51–69 (19 pages). An extended version appeared as Interner Bericht, Universität Erlangen-Nürnberg, IMMDVII-5/96.

[13] Johnsen, E. B., Mai Thuong Tran, T., Owe, O., and Steffen, M. (2012a). Safe locking for multi-threaded Java. In *Proceedings of the International Conference on Foundations of Software Engineering (Theory and Practice) (FSEN'11)*, volume 7141 of *Lecture Notes in Computer Science*, pages 158–173. Springer Verlag.

[14] Johnsen, E. B., Mai Thuong Tran, T., Owe, O., and Steffen, M. (2012b). Safe locking for multi-threaded Java with exceptions. *Journal of Logic and Algebraic Programming, special issue of selected contributions to NWPT'10*. Available online 3. March 2012.

[15] Mai Thuong Tran, T., Owe, O., and Steffen, M. (2010). Safe typing for transactional vs. lock-based concurrency in multi-threaded Java. In Pham, S. B., Hoang, T.-H., McKay, B., and Hirota, K., editors, *Proceedings of the Second International Conference on Knowledge and Systems Engineering, KSE 2010*, pages 188 – 193. IEEE Computer Society.

References III

[16] Mai Thuong Tran, T. and Steffen, M. (2010). Safe commits for Transactional Featherweight Java. In Méry, D. and Merz, S., editors, *Proceedings of the 8th International Conference on Integrated Formal Methods (iFM 2010)*, volume 6396 of *Lecture Notes in Computer Science*, pages 290–304. Springer Verlag. An earlier and longer version has appeared as UiO, Dept. of Informatics Technical Report 392, Oct. 2009.

[17] Mai Thuong Tran, T., Steffen, M., and Truong, H. (2013). Compositional static analysis for implicit join synchronization in a transactional setting. In Eleftherakis, G., Hinchey, M., and Holcombe, M., editors, *Proceedings of SEFM'13*, volume 8137 of *Lecture Notes in Computer Science*, pages 212–228. Springer Verlag.

[18] Owe, O., Steffen, M., and Torjusen, A. (2010). Model testing asynchronously communicating objects using modulo AC rewriting. In *Proceedings of the 6th Workshop on Model-Based Testing MBT'10 (ETAPS Satellite Workshop)*, pages 68–84. Elsevier Science Publishers. Electronic Notes in Theoretical Computer Science ENTCS, Volume 264, Issue 3.

[19] Pun, K. I., Steffen, M., and Stolz, V. (2012). Deadlock checking by a behavioral effect system for lock handling. *Journal of Logic and Algebraic Programming*, 81(3):331–354. A preliminary version was published as University of Oslo, Dept. of Computer Science Technical Report 404, March 2011.

[20] Pun, K. I., Steffen, M., and Stolz, V. (2013). Deadlock checking by data race detection. In *Proceedings of the 5th IPM International Conference on Fundamentals of Software Engineering (FSEN'13)*, volume 8161 of *Lecture Notes in Computer Science*, pages 34–50. Springer Verlag.

[21] Pun, K. I., Steffen, M., and Stolz, V. (2014a). Behaviour inference for deadlock checking. In *Proceeding of the 8th International Symposium on Theoretical Aspects of Software Engineering (TASE'14)*, pages 106–113. IEEE.

[22] Pun, K. I., Steffen, M., and Stolz, V. (2014b). Deadlock checking by data race detection. *Journal of Logic and Algebraic Methods in Programming*. Available online 13 August 2014, http://dx.doi.org/10.1016/j.jlamp.2014.07.003. A preliminary version was published as University of Oslo, Dept. of Computer Science Technical Report 421, October 2012, and a shorter version in the proceedings of FSEN'13.

References IV

[23] Pun, K. I., Steffen, M., and Stolz, V. (2014c). Effect-polymorphic behaviour inference for deadlock checking. In Giannakopoulou, D. and Salaün, G., editors, *Proceedings of SEFM'14*, volume 8702 of *Lecture Notes in Computer Science*, pages 50–64. Springer Verlag. A longer version is available (under the title "Lock-Polymorphic Behaviour Inference for Deadlock Checking") as UiO, Dept. of Informatics Technical Report 436, Sep. 2013.

[24] Pun, K. I., Steffen, M., and Stolz, V. (2016a). Effect-polymorphic behaviour inference for deadlock checking. *Journal of Logic and Algebraic Methods in Programming*, 85(6). A longer version is available (under the title "Lock-Polymorphic Behaviour Inference for Deadlock Checking") as UiO, Dept. of Informatics Technical Report 436, Sep. 2013.

[25] Pun, K. I., Steffen, M., Stolz, V., Wickert, A.-K., Bodden, E., and Eichberg, M. (2016b). Don't let data Go astray: A context-sensitive taint analysis for concurrent programs in Go (extended abstract). In *Proceedings of the 24nd Nordic Workshop on Programming Theory (NWPT'16)*.

[26] Pun, K. I., Steffen, M., Stolz, V., Wickert, A.-K., Bodden, E., and Eichberg, M. (2017). Gotcha: Static taint analysis for Go. *Journal of Logic and Algebraic Methods in Programming*. Under Review for a special issue with selected papers from NWPT'16.

[27] Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *American Mathematical Society*, 74:358–366.

[28] Rosenberg, C. M., Steffen, M., and Stolz, V. (2016). Leveraging DTrace for runtime verification. In Sánchez, C., editor, *Proceedings of the 16th International Conference on Runtime Verification, RV 2016, Madrid, Spain, September 23-30, 2016*, volume 10012 of *Lecture Notes in Computer Science*, pages 318–332. Springer Verlag.