Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

# APA
# Administration, Introduction, Motivation

Jurriaan Hage
e-mail: jur@cs.uu.nl
homepage: http://www.cs.uu.nl/people/jur/

Department of Information and Computing Sciences, Universiteit Utrecht

April 20, 2012

# 1. Administration

# Organizational information



- Did you register in Osiris?
  - If not, then do so
- APA course home page: http://www.cs.uu.nl/wiki/Apa/
- It has the current state of the schedule
- Also, check it out for grading information, links, assignments, deadlines and so on
- Please, do this *before* asking me questions
- Feedback about the course always appreciated

# Material for the course



- Formerly, I used Nielson, Nielson and Hankin. (Section 1.1-1.5, Section 2.1-2.5, Section 3.1,3.3-3.5, Section 4.2-4.4 and Section 5.1-5.5)
- The book is still recommended, and many of the slides are based upon it
- Use the slides as a guide
- The website will have links to more reading material
- Particularly work we did ourselves

# What is expected of you?



- Study the slides, study the papers
- Discuss, discuss
- Do the assignments/project(s)
  - Feel free to contribute ideas.
- Possibly, a short oral examination at the end of the course
- Give feedback about the course!

# 2. Introduction

# What is program analysis?



Program analysis
=
deriving information about the behaviour of computer programs.

# Why do program analysis?



- optimization
  - dead code removal, strict evaluation, avoiding run-time type checks
- validation
  - type checking, security analysis, soft typing
- comprehension
  - maintainability monitoring, reverse engineering, architecture detection

# Static or dynamic



- Dynamic: testing, run-time instrumentation, profiling
- Very precise for observed executions
  - Not the subject of this course

# Static or dynamic



- Dynamic: testing, run-time instrumentation, profiling
- Very precise for observed executions
  - Not the subject of this course
- Static: analysis on the inputs of the compilation
- Often as part of a compiler
- Even for programs with infinite executions, compilation should terminate.
- Analysis must be valid for all executions.

# Static or dynamic



- Dynamic: testing, run-time instrumentation, profiling
- Very precise for observed executions
  - Not the subject of this course
- Static: analysis on the inputs of the compilation
- Often as part of a compiler
- Even for programs with infinite executions, compilation should terminate.
- Analysis must be valid for all executions.
- The two forms can complement each other.

# Optimization



- Optimizations are silenty applied by a compiler,
- based on information discovered during program analysis.
- Optimizing analysis should never lead to failure to compile.
- Information should be valid for all executions.
- We must be able to trust the results of analysis.

# Optimization



- Optimizations are silenty applied by a compiler,
- based on information discovered during program analysis.
- Optimizing analysis should never lead to failure to compile.
- Information should be valid for all executions.
- We must be able to trust the results of analysis.
- Program analysis must be sound (safe) with respect to the language semantics.
  - The analyzer may only err on the safe side
- So prove it.
- Case study: uniqueness typing.
  - Something marked as unique, but used twice, may have been GC'ed away.

# Validation



- Verify that a program is type correct
- Verify that a highly secure value does not end up in a lowly secure variable
- Some programs will fail to compile
- This raises the issue of feedback
- Case study: type inferencing/checking, pattern match analysis, security analysis

# Comprehension



- Software analysis is often coined as the term here.
- Analysis need not be sound, need not be complete.
- Validation not by proof, but empirical validation.
- Metrics are a typical example:
  - McCabe's Cyclomatic Complexity.
  - The higher the value, the more complex the code
  - Above 50 implies unmaintainable.
- Typically, you can always find examples where metrics do not predict well, but they work very well in practice.
- Cheap to compute.

# The setting



Typically,

- a compiler validates a program and generates code.
- For any program, it has to do this in finite time.
- Running the program for all possible inputs is out of the question.
- Halting Problem is undecidable.
  - Decide for any given program and given input whether the program will terminate for that input.
- Every behavioural property of programs is undecidable.
  - Rice's Theorem
- What can we do?

# Possible solutions



Verify properties by

- Program verification: verify properties by using (interactive) proof tools.
- Model checking: exhaustively test the property for all reachable states.
- Automatic program analysis: allow (safe) approximate answers, but keep it automatic and efficient.
- We consider the latter possibility and hope our solutions are not too approximate to be of use.

These three areas do overlap in many ways.

# Two dimensions of complexity



- Properties of the language:
  - parametric polymorphism
  - higher-order, higher-ranked, polymorphic recursion
  - subtyping
  - by-value (strict) or by-need (lazy) evaluation
  - strictness and other annotations,
- More complex implies more flexibility for programmer.
- Properties of the analysis:
  - subtyping, subeffecting, or poisoning
  - monovariant, polyvariant, higher-ranked
  - flow-sensitive versus flow-insensitive
  - minimal or most general (Holdermans and Hage)
  - whole program or modular
- More complex implies more precision and more expensive.

# Make note



- Program analysis is not always restricted to programming languages.
- Can be applied in other places as well:
  - FIRST and FOLLOW for parsing LL(k) languages.
- Admittedly, general recursion/while loops provide most of the essential complications
- Still, even SQL can profit from optimizations.

# Program properties



- In dependently typed programming and contract checking, static properties are encoded in the language itself.
  - Programmer-driven static analysis
- In static analysis we tend to not leave this to the programmer.
- The truth is probably somewhere in the middle.
- Contracts and dependently typed programming establish only properties of values, not of the computations.
- Static analysis often addresses issues relating to how something is computed.

# Safe and sound



- Strong typing (Haskell, Java,...)
  - Programs are guaranteed not to go wrong.
  - Intended optimization: avoiding run-time checks and validation
  - Conservative: sometimes disallows programs that would go right.
- Soft typing (on languages like Scheme, Perl, Ruby, PHP, Python)
  - Allow all programs that might go right.
  - Intended optimization: avoiding run-time checks, some validation
  - Liberal: some programs may go wrong.
  - Add run-time checks/generate warnings
- It all depends on how you will use the analysis results.

# Some example analyses

- Dead-code elimination
- Strictness analysis in lazy functional languages
  - Which arguments to a function will always be evaluated at some point?
- Liveness of variables
  - which variables may still be used?
- Available expressions
  - eliminating double computations
- Dynamic dispatch problem
  - dead code with functions being first class citizens

# More examples



- Shape analysis
  - for avoiding garbage collect
- Binding-time analysis
  - partial evaluation
- Uncaught exceptions in Java
- Weak circularity test in attribute grammars
- Escape analysis
  - What does not escape may be allocated on the stack instead of the heap.
- Binding-time analysis
  - What can be evaluated at compile-time.
- And many, many more...

# Topic overview I



- Dataflow analysis of While language
- Monotone frameworks
- Relation to Abstract Interpretation
- Literature: Chapter 2 of Nielson, Nielson and Hankin
- Project I: soft typing dynamic languages, slicing, dataflow analysis, ...

# Topic overview II



- Type inferencing
- Type and effect systems
  - control-flow analysis
  - binding-time analysis
  - usage analysis
  - minimal typing derivations
  - strictness analysis
- Meta-theory: soundness, conservative extensions
- Algorithmic issues.
- Project II: pattern match analysis, security analysis, usage, ...

# Additional subjects



- Type error feedback for Haskell
  - Scripting the type inferencer
- Security analysis
- Plagiarism detection
- Slicing (Amir Saeidi)
- Software analysis/code querying