Vorlesung Ausgewählte Kapitel aus dem Übersetzerbau

Sabine Glesner

SS 2002

Universität Karlsruhe

Foliensatz 11:

Compiler Verification

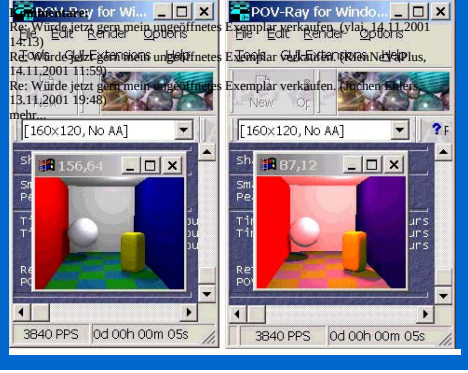
Overview

- Motivation & Definition of Compiler Correctness
- State of the Art
- Program Checking in Compiler Frontends
- Program Checking in Optimizing Compilers
- Practical Experiences
- Verified Program Checking as an Independent Verification Method
- Conclusions & Future Work

Problematische Optimierungen ließe sich lieber verzichten, denn auch ohne *-ipo* erzeugt Intels Compiler sehr flotten Code und das nicht nur für Intel-Prozessoren: So erzielte Bram Stolk auf seinem Notebook mit Transmeta-Prozessor 28 Prozent Beschleunigung gegenüber Kompilaten, die er mit der GNU Compiler Collection erstellte. (*Yue Shi Lai*) ((hes/c't)

Rotstich durch Fehler in Intels C++-Compiler

[Version zum Drucken] [Per E-Mail versenden] [<< Vorige] [Nächste >>]



Re: Würde jetzt gern mein ungeöffnetes Exemplar verkaufen. (ylai, 14.11.2001 14:13)
Re: Würde jetzt gern mein ungeöffnetes Exemplar verkaufen. (Jochen Liedtke, 14.11.2001 11:59)
Re: Würde jetzt gern mein ungeöffnetes Exemplar verkaufen. (Jochen Liedtke, 13.11.2001 19:48)
mehr...

Translation Correctness

- Principle Demand:
  - Transformations need to preserve semantics
- More precisely:
  - Preserve observable behavior of programs
- Proof Technique:
  - Compositionality of programming languages
- But:
  Compiler as well as translated program may produce no result due to resource limitations!

Correctness on Three Levels

- Translation Correctness:
  - Is the specification itself correct?
- Compiler Correctness:
  - Is the specification implemented correctly?
- Implementation Correctness:
  - Is the compiler translated correctly into machine code?

Resource Limitations

What about a compiler that never produces any target program but only says for each source program:

No result due to resource limitations.

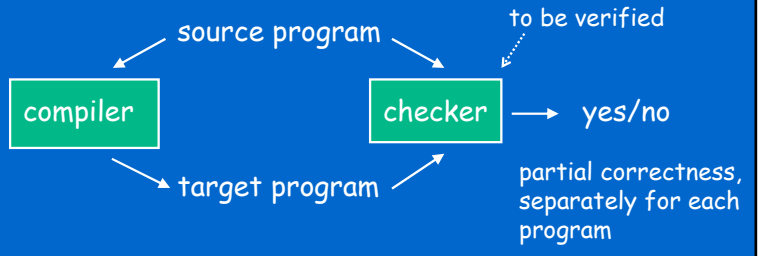
Such a compiler is correct.

Translation Correctness

- Principle Demand:
  - Transformations need to preserve semantics
- More precisely:
  - Preserve observable behavior of programs
- Proof Technique:
  - Compositionality of programming languages
- Correct Translations:
  - preserve observable behavior of programs
  - up to resource limitations

Checking as Proof Principle

Instead of verifying a compiler, verify its result.



Program checking goes back to Blum, Kannan, Wasserman *("Programs that check their work").*

Implementation Correctness

- Checker in higher programming language
  - need for an initial correct compiler to translate checker
  - does not need to be optimizing
- Work by Verifix partners in Kiel and Ulm: Dold, Goerigk, v.Henke, Hoffmann, Langmaack, Vialard
- Talk on April 12, ´02 by Dold and Vialard in the Graduiertenkolleg "Logik in der Informatik"

Related Work

- Translation verification: Zimmermann/Gaul:
  - proof that a refining translation preserves semantics
- Compiler validation:
  - Pnueli: translation validation for restricted programs, in recent work: for some phases of optimizing compilers
  - Necula: proof-carrying code, gcc-validation
  - Gaul: checker for compiler frontends
- Initial correct compiler: Verifix in Ulm/Kiel

Correctness on Three Levels

- Translation Correctness:
  - Is the specification itself correct?
- Compiler Correctness:
  - Is the specification implemented correctly?
- Implementation Correctness:
  - Is the compiler translated correctly into machine code?

Requirements

- Compiler Technology Today: Generators
  - Compilers not hand-written
  - Compilers generated by generators
  - Input for generators: specifications
- Theoretical Foundations:
  - finite automata, context-free grammars, attribute grammars, natural semantics, program analysis, term-rewrite & graph-rewrite systems, scheduling, ...
  - all are implemented in generators
- Use as much of existing technology as possible

Frontend Tasks

Each result is uniquely determined.

Checking Lexical Analysis

Work by Thilo Gaul

- Task of lexical analysis:
  - combines subsequent input characters into tokens, eliminates meaningless characters (whitespaces, ...)
  - described by finite automata, together with rule of longest pattern and priorities to avoid indeterminism
  - unique result
- Task of checker for lexical analysis:
  - check if input character sequence can be mapped onto token sequence
  - if mapping is not deterministic, use conflict table can be generated given finite automaton specification

Checking Syntax Analysis

Work by Thilo Gaul

- Task of syntax analysis:
  - computes context-free properties of programs
  - expressed by deterministic context-free grammars
  - computes syntax tree by using productions of context-free grammar
- Task of checker for syntax analysis:
  - check if syntax tree is a valid derivation
  - check if token sequence is the result of this derivation
  - top-down recursive process along the syntax tree

Checking Semantic Analysis

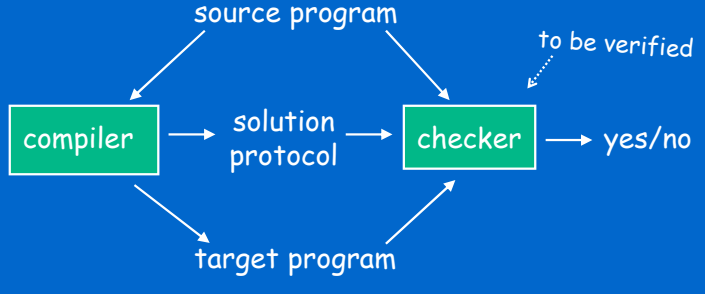
- Task of semantic analysis:
  - computes context-sensitive properties of programs
  - expressed by attributes, associated with program nodes
  - described by local rules, associated with productions of context-free grammar, together with conditions (e.g. attribution rules or natural semantics rules)
- Task of checker for semantic analysis:
  - check consistency of attribution
  - check validity of conditions
  - only local checks

Work by Thilo Gaul

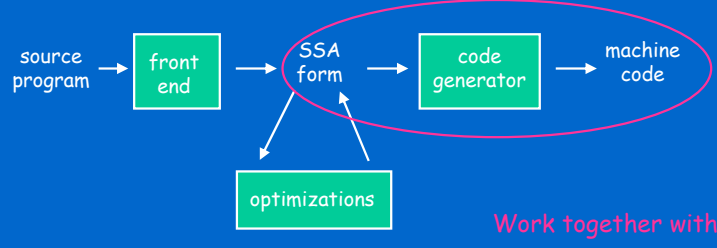
Common Characteristics

- Refining Transformations
  - solution is unique (backtracking might be necessary during computation)
  - solution can be checked without knowing about internal decisions of the compiler
- In Contrast: Optimizing Transformations
  - huge solution space with cost functions
  - only correctness must be checked, not optimality
  - use internal knowledge about the compiler decisions to restrict solution space

Scenario



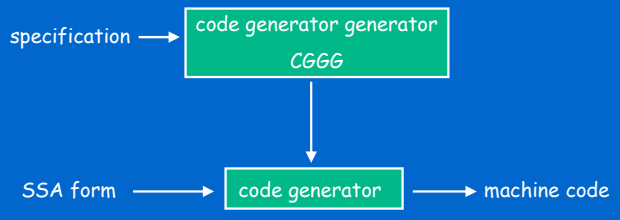
Backend Tasks



Work together with Jan Olaf Blech

SSA: static single assignment representations

Backend Tasks in Detail

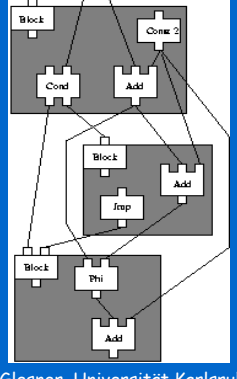


Static Single Assignment

- internal program representation in compiler for handling program analyses and code generation
- explicit representation of def-use-chains
- directed graph of elementary operations such that each variable is assigned exactly once in a program
- overlay of control and data flow
- easy and elegant only for local variables
- memory accesses give additional complexity
- Trapp: explicit dependency graphs as refined SSA

Static Single Assignment

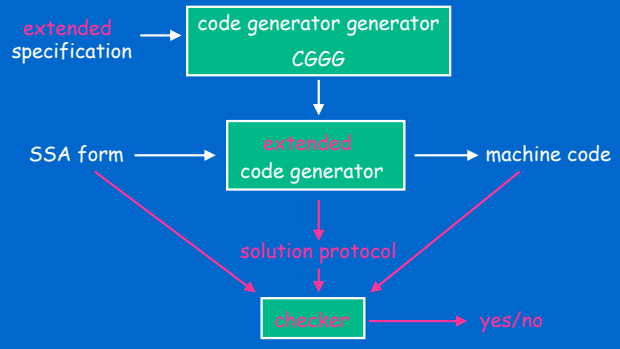
- implemented in FIRM
- Example:
  a:=a+2; if(…) {a:=a+2;} b:=a+2
- five kinds of nodes:
  - data nodes, control flow nodes, block nodes, memory nodes, Phi-nodes
- can be generated using abstract syntax tree
- Work by Liekweg, Lindenmaier, Boesler



Code Generation

- Rewrite systems: well-established in code generation
- Here: bottom-up rewrite systems
  (Pelegri-Llopart 1988, Nymeyer/Katoen, 1997)
- Implemented in our system *CGGG* (code generator generator based on graphs), Boesler 1998
- *CGGG*: Graph rewriting, A*-search to get optimal solution
- Code generator generator in our work: *CGGG*

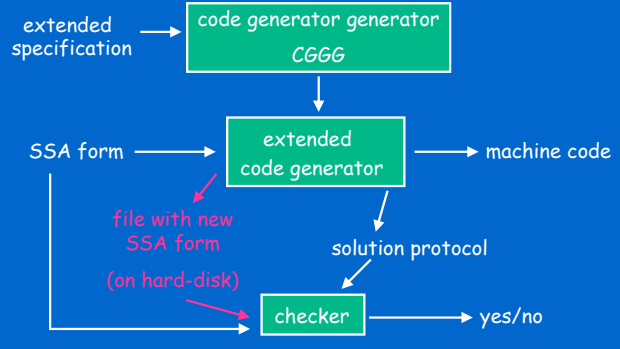
Principle of Backend Checking



Backend Checking: Problems in Practice

- Data structures only available in main memory
  - tight interlocking and communication between checker and code generator necessary
- In particular: we need node identifiers to recompute result of code generator
  - can be output by special option
- Code generator transforms SSA graphs into its own representation form
  - get new node identities (code inspection of code generator)
  - show equivalence of source and new SSA form (proof on paper)
  - check if transformation has been done correctly (checker task)

Principle of Backend Checking



Backend Checking: Further Problems

- Compiler generator specification not fully declarative
  - common trait among diverse compiler generation tools (e.g. Cocktail, ELI, lex, yacc, ...)
  - specs contain source code which is placed directly into generated compiler component
- Levels of correctness not fully separated
  - Specification and implementation correctness cannot be considered independently
- Solution:
  - do code and specification inspection
  - postulate pre-and postconditions, invariants, assertions ("postulates")
  - prove postulates (subject to current and future work)

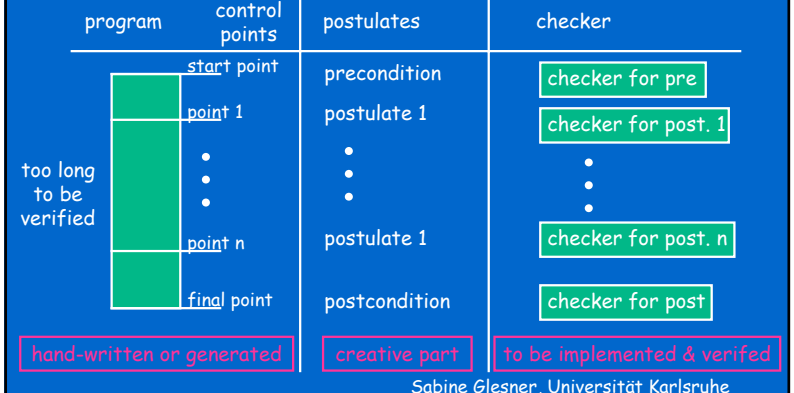
Backend Checking: Characteristics

- Result not uniquely determined
  - optimization problem
  - solution protocol as additional checker input to restrict solution space
- Data structures only available in main memory
  - compiler phase and its checker must work together
- Compiler generator specifications not fully declarative
  - find, prove and check:
    - postulates (assertions about program states), in particular:
    - relations (relations among consecutive program states)

What about Generalization?

- Compiler technology outside of compiler construction:
  - XML processing
  - adapting data formats in general
  - design patterns in software engineering technology
  - software maintenance
  - software components and adaptation
  - meta programming
  - ...
- Compiler technology as a core methodology for automatically handling all kinds of programs and data
- Checking could help in many verification problems!

Checking as Verification Method



Checking as Verification Method

- Code inspection: often necessary to find postulates
  - advantage if code is available
  - use this advantage!
- Assertions, as e.g. in Eiffel, checked at run time
- "Check by contract" with verified checkers
- Especially suited for search and optimization problems
  - in particular suited for NP-problems
  - NP property: result can be checked in poly time

Conclusions

- Correctness of compilers on three abstraction levels
- Frontend checking: frontend treated as black box
  - uniquely determined results
- Backend Checking: checker needs solution protocol
  - optimization problem
- Verified program checking
  - independent verification method
  - combines methods such as code inspection, design by contract, program verification
  - applicable not only to compiler construction but many other software engineering areas

Future Work

- Complete generic backend checker
  - separate specification and code generator correctness in particular: find suitable abstractions
  - verify backend checker
- Establish correctness on specification level
  - prove optimizations to be correct
  - problem: optimizations may change control and data flow
  - formalism for semantics: ASMs or natural semantics?
- Apply verified program checking to software verification problems:
  - software components
  - software maintenance, ...

Thank you!