Korrektheit von Programmen beweisen mit Coq

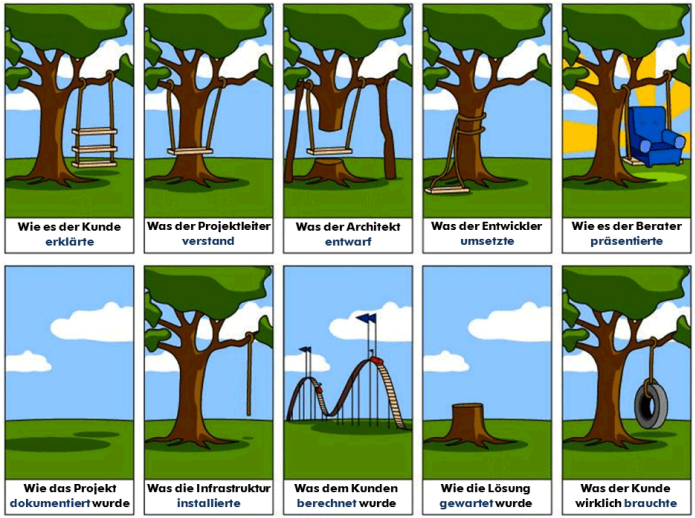
Peter Hrenka

Linux Tag Tübingen 2016

11. Juni 2016

Über mich

- Linux Anwender seit 1995
- Studium Informatik und Mathematik in Tübingen
- Sofwareentwickler `C++`, `python`, OpenGL
- regelmäßig auf OpenSource Konferenzen anzutreffen
- Programmiersprachenjunkie


Wie es der Kunde erklärte
Was der Projektleiter verstand
Was der Architekt entwarf
Was der Entwickler umsetzte
Wie es der Berater präsentierte
Wie das Projekt dokumentiert wurde
Was die Infrastruktur installierte
Was dem Kunden berechnet wurde
Wie die Lösung gewartet wurde
Was der Kunde wirklich brauchte

“Jedes nicht-triviale Programm hat Fehler”

“Jedes nicht-triviale Programm hat Fehler”

Warum?

"Jedes nicht-triviale Programm hat Fehler"

Warum?

1. Fehlendes Verständnis des Problems

"Jedes nicht-triviale Programm hat Fehler"

Warum?

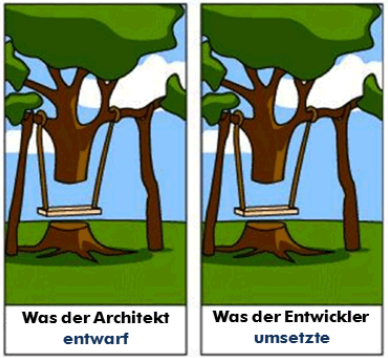
1. Fehlendes Verständnis des Problems
2. Fehlerhafte "Spezifikation"

"Jedes nicht-triviale Programm hat Fehler"

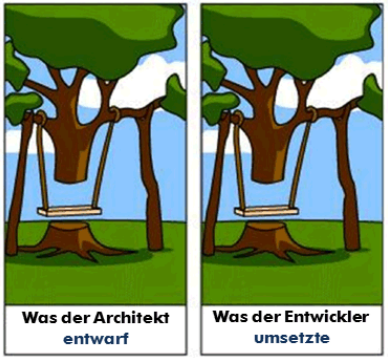
Warum?

1. Fehlendes Verständnis des Problems
2. Fehlerhafte "Spezifikation"
3. Implementierung nicht korrekt

Unser Tagesziel:



Unser Tagesziel:



Rest unverändert...

Wie kann man die Korrektheit (bzgl. der Spezifikation) sicherstellen?

- Testen
- Pair Programming
- Code Reviews
- Bug Bounties
- Open Source
- Statische Analyse
- Audit

Wie kann man die Korrektheit (bzgl. der Spezifikation) sicherstellen?

- Testen
- Pair Programming
- Code Reviews
- Bug Bounties
- Open Source
- Statische Analyse
- Audit

Reicht das?

Fallbeispiel: Timsort

- Verbreiteter Sortieralgorithmus aus der Praxis (Python, Java, Android)
- relativ kompliziert
- 2015: Formale Verfikation fehlgeschlagen und Bug gefunden!
- `proving-android-java-and-python-sorting-algorithm-is-broken`
- Verwendetes Tool KeY: `http://www.key-project.org/`
- Zähneknischende Korrektur

Verifiziertes Betriebssystem:

- Microkernel, Devices laufen im Userspace
- Implementiert in C
- Läuft auf ARM, x86
- GPL
- Formaler Beweis mit Beweisassistenzsystem `Isabelle/HOL`
  - Keine Pufferüberläufe, Null-Pointer-Zugriffe oder use-after-free
  - Code erfüllt die Spezifikation, Sicherheitsaspekte
  - Binärcode ebenfalls verifiziert (Compiler hat keine Fehler gemacht)

Wie soll so ein Beweis funktionieren?

Wie soll so ein Beweis funktionieren?

Ist mein Sortieralgorithmus korrekt?

Wie soll so ein Beweis funktionieren?

Ist mein Sortieralgorithmus korrekt?

Welches Beweisassistenzsystem nehmen?

Coq

- Beweisassistenzsystem, in Entwicklung sei 1984
- “coq”: französisch “Hahn”
- entwickelt u.a. am INRIA
- implementiert in `OCaml`
- verwendet “dependent types”
- interaktiv, eigene IDE `coqide` oder `Proof General` für Emacs
- online (via JS transpiler) `https://x80.org/rhino-coq/`
- LGPL

Mathematik mit Coq

- Vier-Farben-Satz, 2005
- Satz von Feit-Thomson (Gruppentheorie), 2012
- Univalent Foundations, Homotopy Type Theory (HoTT), ca. 2012

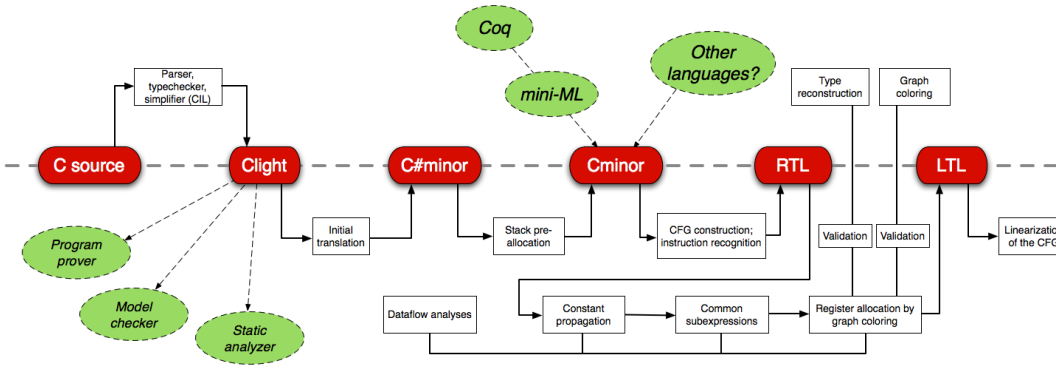
Mathematik mit Coq

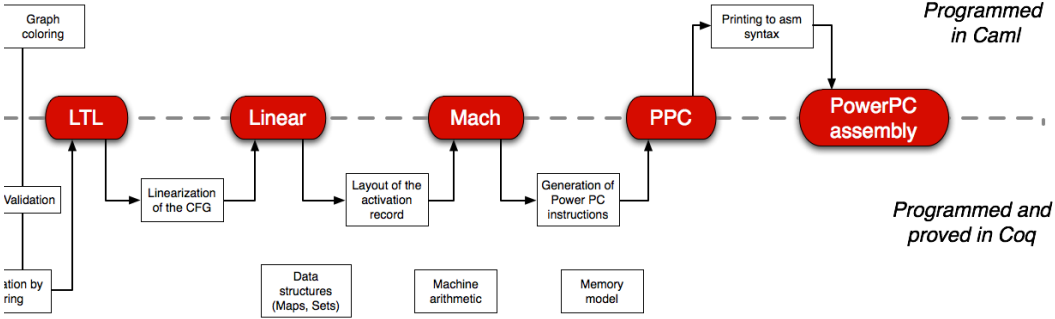
- Vier-Farben-Satz, 2005
- Satz von Feit-Thomson (Gruppentheorie), 2012
- Univalent Foundations, Homotopy Type Theory (HoTT), ca. 2012

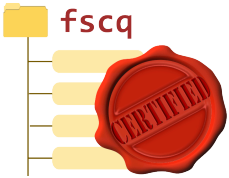
$\rightarrow$ scheint für Mathematik brauchbar, wie sieht es mit Informatik aus?

CompCert

- "Compilers you can *formally* trust"
- große Teilmenge von ISO C90 / ANSI C, verträglich mit MISRA-C 2004
- generiert effizienten Code für PowerPC, ARM and x86
  "about 90% of the performance of GCC version 4 at optimization level 1"
- implementiert in `OCaml` und `coq`
- geleitet von Xavier Leroy (LinuxThreads)
- nicht-freie Lizenz, aber einige Teile unter GPL und BSD


Coq
Other languages?
mini-ML
Parser, typechecker, simplifier (CIL)
Type reconstruction
Graph coloring
C source
Clight
C#minor
Cminor
RTL
LTL
Program prover
Model checker
Static analyzer
Initial translation
Stack pre-allocation
CFG construction; instruction recognition
Validation
Validation
Linearizatio of the CFG
Dataflow analyses
Constant propagation
Common subexpressions
Register allocation by graph coloring


Graph coloring
LTL
Linear
Mach
PPC
Printing to asm syntax
PowerPC assembly
Programmed in Caml
Validation
ation by ring
Linearization of the CFG
Layout of the activation record
Generation of Power PC instructions
Programmed and proved in Coq
Data structures (Maps, Sets)
Machine arithmetic
Memory model



- "A Formally Certified Crash-proof File System"
- Nachweis, daß bei Absturz zu beliebigem Zeitpunkt keine Daten verloren gehen
- MIT 2015, u.A. Adam Chlipala
- implementiert und verfiziert in `coq`
- extrahierbar nach `ocaml`, `Haskell` oder `go`
- verwenbar unter Linux mit `fuse`

Demo

Demo hat gezeigt

- Aussagenlogik ist leicht
- Arithmetik ist fummelig
- Coq = schlechtestes Taschenrechnerprogramm der Welt
- Geniale Notationsfunktionalität
- Sätze sind Typen, Programme sind Beweise (Curry-Howard-Isomorphismus)
- Induktion geht nicht nur mit $n \in \mathbb{N}$
- Automatisierung hilft

Dokumentation

- gutes Tutorial zum Durcharbeiten: Software Foundations
  `http://www.cis.upenn.edu/~bcpierce/sf/current/index.html`
- Älteres Buch: Interactive Theorem Proving and Program Development
- Online-Buch (Fortgeschritten): Certified Programming with dependent types:
  `http://adam.chlipala.net/cpdt/`
- Offizielle Doku:
  `https://coq.inria.fr/documentation`

Probleme

- Offizielle Dokumentation (höchstens) zum Nachschlagen geeignet
- Standardbibliothek mathematiklastig, wenig Datenstrukturen
- Unidirektione Arbeitsweise
  - `coq` $\longrightarrow$ `OCaml`
- Kein Export nach `C`, `C++`, $\langle$*mainstream Sprache*$\rangle$

Projekte

- Floats for Coq: formalisiere Fließkommazahlen
  `http://flocq.gforge.inria.fr/`
- JSCert: Coq specification of ECMAScript 5
  `https://github.com/jscert/jscert`
- The C11 standard formalized in Coq
  `http://robbertkrebbers.nl/thesis.html`
- RustBelt: Logical Foundations for the Future of Safe Systems Programming
  `http://plv.mpi-sws.org/rustbelt/`

Projekte

Why3

- Platform für Programmverfikation von Programmen in WhyML
- Anbindungen für andere Programmiersprachen existieren
  - Frama-C für C
  - SPARK für Ada
- Automatische Solver: Alt-Ergo, CVC3/4, Z3, uvm.
- Manuelle Solver: coq, PVS und Isabelle/HOL

Projekte

- Erweitere Untermenge von `Ada`
- kann Vor- und Nachbedingungen teilweise zur Compilezeit prüfen
- kann coq-Code exportieren, um schwere Fälle manuell zu beweisen
- Pro- und GPL Editionen

Vielen Dank!

Fragen?