C++ ist typsicher? Garantiert!

Daniel Wasserrab[1] Tobias Nipkow[2] Gregor Snelting[1] Frank Tip[3]

[1]Universität Passau

[2]Technische Universität München

[3]IBM T. J. Watson Research Center

D. Wasserrab, T. Nipkow, G. Snelting, F. Tip:
An Operational Semantics and Type Safety Proof for Multiple Inheritance in C++.
In Proc. OOPSLA 2006, pp. 345–362. Portland, OR, October 2006

Inhalt

1. Motivation
   - Bedeutung von Typsicherheit
   - Mehrfachvererbung in C++
   - CoreC++
2. Beispiele
3. Semantik und Typsystem von CoreC++
   - Formalismen
   - Typsicherheitsbeweis
4. Anwendbarkeit in der Softwaretechnik
   - Ausführbarkeit
   - Softwaresicherheit
5. Zusammenfassung

Softwaretechnische Bedeutung von Typsicherheit

Typsicherheit bedeutende Errungenschaft der Softwaretechnik:

- kein unkontrolliertes Abstürzen mit Laufzeitfehlern (z.B. in C++: kein `segmentation fault`)
- verständlicher zu lesen und zu warten
- effizienter compilierbar

Typsicherheit einer Sprache erhöht Vertrauen in Qualität und Sicherheit der in ihr implementierten Anwendungen

Softwaretechnische Bedeutung von Typsicherheit

Typsicherheit bedeutende Errungenschaft der Softwaretechnik:

- kein unkontrolliertes Abstürzen mit Laufzeitfehlern (z.B. in C++: kein `segmentation fault`)
- verständlicher zu lesen und zu warten
- effizienter compilierbar

Typsicherheit einer Sprache erhöht Vertrauen in Qualität und Sicherheit der in ihr implementierten Anwendungen

Typsicherheit für reale Sprachen

erste Typsicherheitsbeweise für akademische Spielzeugsprachen schon vor Jahrzehnten (von Hand)

Problem für reale Sprachen: Komplexität des Typsystems
$\Longrightarrow$ maschinelle Unterstützung unverzichtbar!
deshalb nötig: mächtigere Beweissysteme

Vor 10 Jahren Typsicherheit einer formalen Semantik von Java, Beweis in Isabelle/HOL (Nipkow et al.)

Darauf aufbauend unser Beweis für Vererbungsmechanismus in C++

Typsicherheit für reale Sprachen

erste Typsicherheitsbeweise für akademische Spielzeugsprachen schon vor Jahrzehnten (von Hand)

Problem für reale Sprachen: Komplexität des Typsystems
$\Longrightarrow$ maschinelle Unterstützung unverzichtbar!
deshalb nötig: mächtigere Beweissysteme

Vor 10 Jahren Typsicherheit einer formalen Semantik von Java, Beweis in Isabelle/HOL (Nipkow et al.)

Darauf aufbauend unser Beweis für Vererbungsmechanismus in C++

Typsicherheit für reale Sprachen

erste Typsicherheitsbeweise für akademische Spielzeugsprachen schon vor Jahrzehnten (von Hand)

Problem für reale Sprachen: Komplexität des Typsystems
$\Longrightarrow$ maschinelle Unterstützung unverzichtbar!
deshalb nötig: mächtigere Beweissysteme

Vor 10 Jahren Typsicherheit einer formalen Semantik von Java, Beweis in Isabelle/HOL (Nipkow et al.)

Darauf aufbauend unser Beweis für Vererbungsmechanismus in C++

Typsicherheit für reale Sprachen

erste Typsicherheitsbeweise für akademische Spielzeugsprachen schon vor Jahrzehnten (von Hand)

Problem für reale Sprachen: Komplexität des Typsystems
$\Longrightarrow$ maschinelle Unterstützung unverzichtbar!
deshalb nötig: mächtigere Beweissysteme

Vor 10 Jahren Typsicherheit einer formalen Semantik von Java, Beweis in Isabelle/HOL (Nipkow et al.)

Darauf aufbauend unser Beweis für Vererbungsmechanismus in C++

Unsere Ziele

Vorliegende Semantik mit Typsicherheitsbeweis soll

- Vertrauen in C++ erhöhen
- Leistungsfähigkeit von Theorembeweiser und formalen Semantiken demonstrieren
- Grundlage für Beweise von Sicherheitsanalysen bilden

Unsere Ziele

Vorliegende Semantik mit Typsicherheitsbeweis soll

- Vertrauen in C++ erhöhen
- Leistungsfähigkeit von Theorembeweiser und formalen Semantiken demonstrieren
- Grundlage für Beweise von Sicherheitsanalysen bilden

Unsere Ziele

Vorliegende Semantik mit Typsicherheitsbeweis soll

- Vertrauen in C++ erhöhen
- Leistungsfähigkeit von Theorembeweiser und formalen Semantiken demonstrieren
- Grundlage für Beweise von Sicherheitsanalysen bilden

Unsere Ziele

Vorliegende Semantik mit Typsicherheitsbeweis soll

- Vertrauen in C++ erhöhen
- Leistungsfähigkeit von Theorembeweiser und formalen Semantiken demonstrieren
- Grundlage für Beweise von Sicherheitsanalysen bilden

Mehrfachvererbung in C++

Kurze Zusammenfassung:

- zwei Vererbungsarten: *virtuelle* und *nichtvirtuelle*
- in einem Objekt besitzt jede Oberklasse ein oder mehrere Subobjekte (je nach Art der Vererbung)
- Rossie und Friedman entwickelten Subobjektalgebra (keine Semantik!)

CoreC++

ausführbare	C++ Programme gegen die Formalisierung testen
Formalisierung	alle existierenden Spezifikationen benutzen Implementationsdetails (z.B. v-tables) oder informelle Beschreibungen
von (Kern-)C++	Rossie/Friedman: keine Behandlung von Casts, dynamische Bindung nicht echtes C++ andere Ansätze ignorieren Mehrfachvererbung
und seiner	Kern von C++ (ohne Zeigerarithmetik) wurde als
Typsicherheit	typsicher angesehen (Rossie,Friedman,Wand '96)
mittels eines	nun Theorembeweiser mächtig (und benutzbar) genug
Theorembeweisers	für komplexe Semantik und Typsystem

CoreC++

ausführbare	C++ Programme gegen die Formalisierung testen
Formalisierung	alle existierenden Spezifikationen benutzen Implementationsdetails (z.B. v-tables) oder informelle Beschreibungen
von (Kern-)C++	Rossie/Friedman: keine Behandlung von Casts, dynamische Bindung nicht echtes C++ andere Ansätze ignorieren Mehrfachvererbung
und seiner	Kern von C++ (ohne Zeigerarithmetik) wurde als
Typsicherheit	typsicher angesehen (Rossie,Friedman,Wand '96)
mittels eines	nun Theorembeweiser mächtig (und benutzbar) genug
Theorembeweisers	für komplexe Semantik und Typsystem

CoreC++

ausführbare	C++ Programme gegen die Formalisierung testen
Formalisierung	alle existierenden Spezifikationen benutzen Implementationsdetails (z.B. v-tables) oder informelle Beschreibungen
von (Kern-)C++	Rossie/Friedman: keine Behandlung von Casts, dynamische Bindung nicht echtes C++ andere Ansätze ignorieren Mehrfachvererbung
und seiner	Kern von C++ (ohne Zeigerarithmetik) wurde als
Typsicherheit	typsicher angesehen (Rossie,Friedman,Wand '96)
mittels eines	nun Theorembeweiser mächtig (und benutzbar) genug
Theorembeweisers	für komplexe Semantik und Typsystem

CoreC++

ausführbare	C++ Programme gegen die Formalisierung testen
Formalisierung	alle existierenden Spezifikationen benutzen Implementationsdetails (z.B. v-tables) oder informelle Beschreibungen
von (Kern-)C++	Rossie/Friedman: keine Behandlung von Casts, dynamische Bindung nicht echtes C++ andere Ansätze ignorieren Mehrfachvererbung
und seiner	Kern von C++ (ohne Zeigerarithmetik) wurde als
Typsicherheit	typsicher angesehen (Rossie,Friedman,Wand ’96)
mittels eines	nun Theorembeweiser mächtig (und benutzbar) genug
Theorembeweisers	für komplexe Semantik und Typsystem

CoreC++

ausführbare	C++ Programme gegen die Formalisierung testen
Formalisierung	alle existierenden Spezifikationen benutzen Implementationsdetails (z.B. v-tables) oder informelle Beschreibungen
von (Kern-)C++	Rossie/Friedman: keine Behandlung von Casts, dynamische Bindung nicht echtes C++ andere Ansätze ignorieren Mehrfachvererbung
und seiner	Kern von C++ (ohne Zeigerarithmetik) wurde als
Typsicherheit	typsicher angesehen (Rossie,Friedman,Wand '96)
mittels eines	nun Theorembeweiser mächtig (und benutzbar) genug
Theorembeweisers	für komplexe Semantik und Typsystem

CoreC++

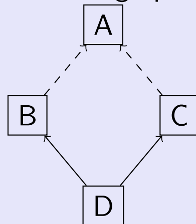
ausführbare	C++ Programme gegen die Formalisierung testen
Formalisierung	alle existierenden Spezifikationen benutzen Implementationsdetails (z.B. v-tables) oder informelle Beschreibungen
von (Kern-)C++	Rossie/Friedman: keine Behandlung von Casts, dynamische Bindung nicht echtes C++ andere Ansätze ignorieren Mehrfachvererbung
und seiner	Kern von C++ (ohne Zeigerarithmetik) wurde als
Typsicherheit	typsicher angesehen (Rossie,Friedman,Wand '96)
mittels eines	nun Theorembeweiser mächtig (und benutzbar) genug
Theorembeweisers	für komplexe Semantik und Typsystem

CoreC++!

Codebeispiele

Dynamische Bindung

```
class A {virtual f();};
class B : virtual A { };
class C : virtual A {virtual f();};
class D : B, C { };
...
B* b = new D();
b->f();
```

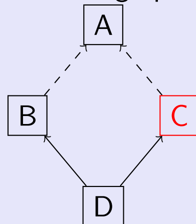


Welches $f()$ wird aufgerufen?

Codebeispiele

Dynamische Bindung

```
class A {virtual f();};
class B : virtual A { };
class C : virtual A {virtual f();};
class D : B, C { };
...
B* b = new D();
b->f();
```

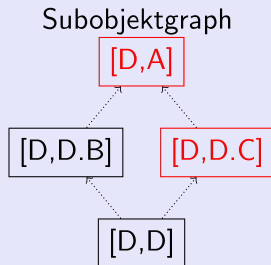


Welches $f()$ wird aufgerufen?
Methodenaufruf aufgelöst zu C::$f()$, weder Ober- noch Unterklasse von B

Codebeispiele

Dynamische Bindung

```
class A {virtual f();};
class B : virtual A { };
class C : virtual A {virtual f();};
class D : B, C { };
...
B* b = new D();
b->f();
```



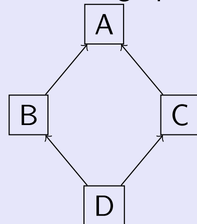
Welches $f()$ wird aufgerufen?
Methodenaufruf aufgelöst zu `C`::$f()$, weder Ober- noch Unterklasse von `B`

Lösung: `D` (dynamische Klasse) sieht "kleinste" Definition von $f()$
in seinem `C`-Subobjekt (dominiert das `A`-Subobjekt)

Codebeispiele

Von virtueller zu nichtvirtueller Vererbung

```
class A {virtual f();};
class B : A { };
class C : A {virtual f();};
class D : B, C { };
...
B* b = new D();
b->f();
```

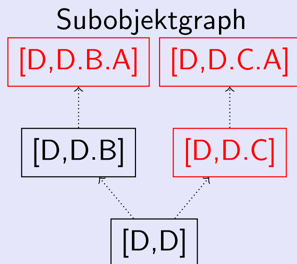


Welches $f()$ wird aufgerufen?

Codebeispiele

Von virtueller zu nichtvirtueller Vererbung

```
class A {virtual f();};
class B : A { };
class C : A {virtual f();};
class D : B, C { };
...
B* b = new D();
b->f();
```



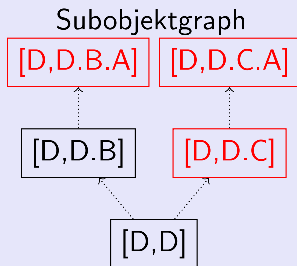
Welches $f()$ wird aufgerufen?

`D` (dyn. Klasse): zwei `A`-Subobjekte und ein `C`-Subobjekt mit Def. von $f()$

Codebeispiele

Von virtueller zu nichtvirtueller Vererbung

```
class A {virtual f();};
class B : A { };
class C : A {virtual f();};
class D : B, C { };
...
B* b = new D();
b->f();
```



Welches $f()$ wird aufgerufen?

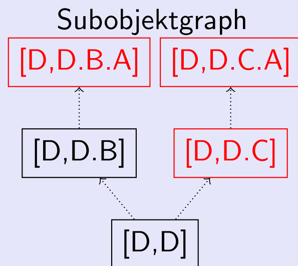
D (dyn. Klasse): zwei A-Subobjekte und ein C-Subobjekt mit Def. von $f()$

dynamisch mehrdeutig!

Codebeispiele

Von virtueller zu nichtvirtueller Vererbung

```
class A {virtual f();};
class B : A { };
class C : A {virtual f();};
class D : B, C { };
...
B* b = new D();
b->f();
```



Welches $f()$ wird aufgerufen?

D (dyn. Klasse): zwei A-Subobjekte und ein C-Subobjekt mit Def. von $f()$

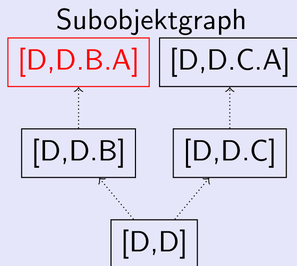
dynamisch mehrdeutig!

aber C++ verwendet *statische Klasse* B für Eindeutigkeit

Codebeispiele

Von virtueller zu nichtvirtueller Vererbung

```
class A {virtual f();};
class B : A { };
class C : A {virtual f();};
class D : B, C { };
...
B* b = new D();
b->f();
```



Welches $f()$ wird aufgerufen?

`D` (dyn. Klasse): zwei `A`-Subobjekte und ein `C`-Subobjekt mit Def. von $f()$

dynamisch mehrdeutig!

aber C++ verwendet *statische Klasse* `B` für Eindeutigkeit

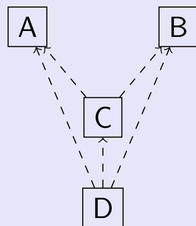
Methodenaufruf aufgelöst zu $f()$ in `A`-Subobjekt innerhalb `B`-Subobjekts

Codebeispiele

Und was macht der Compiler?

```
class A { int x; };
class B { int x; };
class C : virtual A, virtual B
  { int x; };
class D : virtual A, virtual B,
  virtual C { };
...
D* d = new D();
d->x = 42;
```

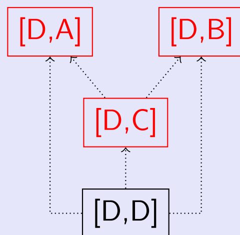
Klassengraph



Codebeispiele

Und was macht der Compiler?

```
class A { int x; };
class B { int x; };
class C : virtual A, virtual B
  { int x; };
class D : virtual A, virtual B,
  virtual C { };
...
D* d = new D();
d->x = 42;
```

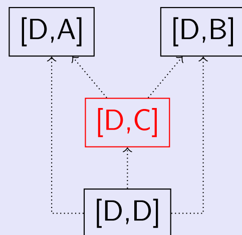


g++ weist Attributszuweisung als mehrdeutig zurück
(Kandidaten: `A`- and `B`-Subobjekt)

Codebeispiele

Und was macht der Compiler?

```
class A { int x; };
class B { int x; };
class C : virtual A, virtual B
  { int x; };
class D : virtual A, virtual B,
  virtual C { };
...
D* d = new D();
d->x = 42;
```



g++ weist Attributszuweisung als mehrdeutig zurück (Kandidaten: A- and B-Subobjekt)

Intel Compiler weist korrektem x im C-Subobjekt zu

Formales Modell

Programm: Klassenhierarchie

Programmanweisung:

- Objekterzeugung
- Literal
- sequentielle Komposition
- statischer/dynamischer Cast
- Variablenzugriff/-zuweisung
- Attributzugriff/-zuweisung
- Blöcke mit lokalen Variablen
- Methodenaufruf
- binäre Operatoren
- `if-then-else`
- `while`

Ausnahmen: `OutOfMemory`, `ClassCast`, `NullPointer`

Formales Modell

Programm: Klassenhierarchie

Programmanweisung:

- Objekterzeugung
- Literal
- sequentielle Komposition
- statischer/dynamischer Cast
- Variablenzugriff/-zuweisung
- Attributzugriff/-zuweisung
- Blöcke mit lokalen Variablen
- Methodenaufruf
- binäre Operatoren
- `if-then-else`
- `while`

Ausnahmen: `OutOfMemory`, `ClassCast`, `NullPointer`

Werte:

- Primitive (`boolean` und `int`)
- Zeiger auf Objekte (auf dem Heap)

Zeiger: Tupel aus Heapadresse und Subobjektidentifikator (`this`-pointer, bestimmt statische Klasse von gecasteten Objekten)

Typsystem

Typen: Boolean, Integer, Null, Class, Void
Benötigt *Typumgebung*, bildet Variablennamen auf Typen ab
Beispiel:

Statischer Up Cast:

$$\frac{P,E \vdash e :: \mathit{Class}\ D \qquad \mathit{is\text{-}class}\ P\ C \qquad P \vdash \mathsf{path}\ D\ \mathsf{to}\ C\ \mathsf{unique}}{P,E \vdash \mathtt{stat_cast}\ C\ e :: \mathit{Class}\ C}$$

$P \vdash \mathsf{path}\ D\ \mathsf{to}\ C\ \mathsf{unique} \equiv$ Klasse D hat genau ein(en Pfad zu) C Subobjekt

Typsystem

Typen: Boolean, Integer, Null, Class, Void
Benötigt *Typumgebung*, bildet Variablennamen auf Typen ab
Beispiel:

Statischer Up Cast:

$$\frac{P,E \vdash e :: \mathit{Class}\ D \qquad \mathit{is\text{-}class}\ P\ C \qquad P \vdash \mathit{path}\ D\ \mathit{to}\ C\ \mathit{unique}}{P,E \vdash \mathtt{stat_cast}\ C\ e :: \mathit{Class}\ C}$$

$P \vdash \mathit{path}\ D\ \mathit{to}\ C\ \mathit{unique} \equiv$ Klasse D hat genau ein(en Pfad zu) C Subobjekt

Methodenaufruf:

$$\frac{P,E \vdash e :: \mathit{Class}\ C \qquad P \vdash C\ \mathit{has\ least}\ M = (Ts,\ T,\ m)\ \mathit{via}\ Cs \qquad P,E \vdash es\ [::]\ Ts' \qquad P \vdash Ts'\ [\leq]\ Ts}{P,E \vdash e.M(es) :: T}$$

$P \vdash C\ \mathit{has\ least}\ M = (Ts,\ T,\ m)\ \mathit{via}\ Cs \equiv$ kleinste Definition von M für C

Semantik

Semantik: Formelle Beschreibung des Programmverhalten

Operationelle Semantik:

- simuliert Programmausführung
- beschreibt Transitionen auf einer Zustandsmaschine
- "in Programm P, werden Anweisung e und Zustand s zu Restanweisung e' und Zustand s' ausgewertet"

Zustand: Tupel von Heap und Speicher für lokale Variablen

Beispiele

Statischer Up Cast:

$$\frac{P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\,(a,\ Cs),s_1\rangle \qquad P \vdash \textsf{\textit{path}}\ last\ Cs\ \textsf{\textit{to}}\ C\ \textsf{\textit{via}}\ Cs' \qquad Ds = Cs\ @_p\ Cs'}{P,E \vdash \langle \texttt{stat_cast}\ C\ e,s_0\rangle \Rightarrow \langle ref\,(a,\ Ds),s_1\rangle}$$

$P \vdash \textsf{\textit{path}}\ D\ \textsf{\textit{to}}\ C\ \textsf{\textit{via}}\ Cs'$ $\equiv$ Klasse D erreicht ein C Subobjekt mittels Pfad Cs'
Pfad entspricht Subobjekt, die letzte Klasse ist statische Klasse des Objekts

Beispiele

Statischer Up Cast:

$$\frac{P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\,(a,\ Cs),s_1\rangle \qquad P \vdash \textsf{path}\ last\ Cs\ \textsf{to}\ C\ \textsf{via}\ Cs' \qquad Ds = Cs\ @_p\ Cs'}{P,E \vdash \langle \texttt{stat_cast}\ C\ e,s_0\rangle \Rightarrow \langle ref\,(a,\ Ds),s_1\rangle}$$

$P \vdash \textsf{path}\ D\ \textsf{to}\ C\ \textsf{via}\ Cs'$ $\equiv$ Klasse D erreicht ein C Subobjekt mittels Pfad Cs'
Pfad entspricht Subobjekt, die letzte Klasse ist statische Klasse des Objekts

Methodenaufruf:

$$\frac{\begin{array}{c} P,E \vdash \langle e,s_0\rangle \Rightarrow \langle ref\,(a,\ Cs),s_1\rangle \qquad P,E \vdash \langle ps,s_1\rangle\ [\Rightarrow]\ \langle map\ \texttt{Val}\ vs,(h_2,\ l_2)\rangle \\ h_2\ a = \lfloor(C,\ _)\rfloor \qquad P \vdash last\ Cs\ \textsf{has least}\ M = (_,\ T',\ _,\ _)\ \textsf{via}\ Ds \\ P \vdash (C,\ Cs\ @_p\ Ds)\ \textsf{selects}\ M = (Ts,\ T,\ pns,\ body)\ \textsf{via}\ Cs' \qquad |vs| = |pns| \\ P \vdash Ts\ \textsf{Casts}\ vs\ \textsf{to}\ vs' \qquad {l_2}' = [this \mapsto Ref\,(a,\ Cs'),\ pns\ [\mapsto]\ vs'] \\ new\text{-}body = (\textsf{case}\ T'\ \textsf{of}\ Class\ D \Rightarrow \texttt{stat_cast}\ D\ body \mid \text{-} \Rightarrow body) \\ P,E(this \mapsto Class\ (last\ Cs'),\ pns\ [\mapsto]\ Ts) \vdash \langle new\text{-}body,(h_2,\ {l_2}')\rangle \Rightarrow \langle e',(h_3,\ l_3)\rangle \end{array}}{P,E \vdash \langle e.M(ps),s_0\rangle \Rightarrow \langle e',(h_3,\ l_2)\rangle}$$

Typsicherheit

Typsicherheit: Ausführung einer Programmanweisung e des Typs T in Zustand s resultiert in

- entweder *vollständig ausgewerteter Anweisung* e' des Typs $T' \leq T$
- oder *kontrollierter Ausnahme*

"No untrapped errors may occur!" (Cardelli '04)

Typsicherheitsbeweis

Standard-Beweistechnik: (Wright, Felleisen '94)

Progress: "Semantik kann nicht steckenbleiben"

Preservation: "Auswertung einer wohlgetypten Anweisung resultiert in weiterer wohlgetypter Anweisung mit gleichem oder kleineren Typ"

Beweisinvariante als Laufzeit-Typsystem formuliert
(Drossopoulou, Eisenbach '97)

Typsicherheitsbeweis

Standard-Beweistechnik: (Wright, Felleisen '94)

Progress: "Semantik kann nicht steckenbleiben"

Preservation: "Auswertung einer wohlgetypten Anweisung resultiert in weiterer wohlgetypter Anweisung mit gleichem oder kleineren Typ"

Beweisinvariante als Laufzeit-Typsystem formuliert (Drossopoulou, Eisenbach '97)

Das (formale) Typsicherheitstheorem:

If wf-C-prog P and $P,E \vdash s \surd$ and $P,E \vdash e :: T$ and
$\mathcal{D}\ e\ \lfloor \text{dom}\ (\text{lcl}\ s) \rfloor$ and $P,E \vdash \langle e,s \rangle \to^* \langle e',s' \rangle$ and
$\nexists e''\ s''.\ P,E \vdash \langle e',s' \rangle \to \langle e'',s'' \rangle$ then
$(\exists v.\ e' = \texttt{Val}\ v \land P,\text{hp}\ s' \vdash v :\leq T) \lor$
$(\exists r.\ e' = \text{Throw}\ r \land \text{the-addr}\ (\text{Ref}\ r) \in \text{dom}\ (\text{hp}\ s'))$.

Isabelle / HOL

emacs: TypeSafe.thy

File Edit View Cmds Tools Options Buffers Proof-General X-Symbol Isabelle Help

State Context Retract Undo Next Use Goto Find Command Stop Restart Info Help

```
corollary TypeSafety:
assumes wf:"wf_C_prog P" and sconf:"P,E ⊢ s √"
  and e:"P,E ⊢ e :: T" and D:"\<D> e ⌊dom(lcl s→
  and step:"P,E ⊢ ⟨e,s⟩ →* ⟨e',s'⟩"
  and d:"¬(∃e'' s''. P,E ⊢ ⟨e',s'⟩ → ⟨e'',s''⟩)"
shows "(∃v. e' = Val v ∧ P,hp s' ⊢ v :≤ T) ∨
      (∃r. e' = Throw r ∧
         the_addr (Ref r) ∈ dom(hp s'))"
proof -
  from sconf e wf have wf_c:"P,E,s ⊢ e : T √"
    by(fastsimp intro:WT_implies_WTrt simp:wf_c→
  with wf step have tc:"P,E,(hp s') ⊢ e' :NT T"
    by(rule Subject_reductions)
  from step_preserves_sconf[OF wf step
    wte[THEN WT_implies_WTrt] sconf] wf
  have sconf':"P,E ⊢ s' √" by simp
```

u--**-XEmacs: TypeSafe.thy (Isar script XS:i

emacs: *goals*

```
proof (prove): step 3

fixed variables: P, E, s, e, T, e',
  s'
prems:
  wf_C_prog P
  P,E ⊢ s √
  P,E ⊢ e :: T
  \<D> e ⌊dom (lcl s)⌋
  P,E ⊢ ⟨e,s⟩ →* ⟨e',s'⟩
  ¬ (∃e'' s''.
        P,E ⊢ ⟨e',s'⟩ → ⟨e'',s''⟩)

using this:
  P,E ⊢ s √
  P,E ⊢ e :: T
```

emacs: *response*

```
have wf_c: P,E,s ⊢ e : T √
```

Isabelle: generischer Theorembeweiser, *Isar*: verständlich Beweisskripte

HOL: *H*igher *O*rder *L*ogic (First order logic + Arithmetik)

Beweisen *interaktiver Prozess*, Benutzer gibt allgemeine Beweisstruktur vor
System stellt Taktiken z.B. für Termersetzung oder logisches Kalkül bereit

Ausführbarkeit

Isabelle enthält *Codegenerator*, erstellt ML Dateien aus Beweisen
$\longrightarrow$ ML Interpreter für Semantik and Typsystem

Um C++ Dateien gegen die Formalisierung zu prüfen:

- C++ Programm darf nur CoreC++ Programmanweisungen verwenden
- Programm als CoreC++ Repräsentation in ML umschreiben
- dafür existiert ein Parser

Semantik formalisiert korrekt dynamisches Binden (vgl. Bsp. 1 und 2)
Semantik formalisiert korrekt Subobjekt-Domination (vgl. Bsp. 3)

Softwaresicherheit

- Software-Sicherheitsanalyse heute unverzichtbar!
- Algorithmen für verschiedenste Anwendungen existieren
- meist ohne oder bestenfalls mit manuellem Korrektheitsbeweis

Softwaresicherheit

- Software-Sicherheitsanalyse heute unverzichtbar!
- Algorithmen für verschiedenste Anwendungen existieren
- meist ohne oder bestenfalls mit manuellem Korrektheitsbeweis

Quis custodiet ipsos custodes?

Kann ich sicher sein, dass der Algorithmus das tut, was er behauptet?

Softwaresicherheit

- Software-Sicherheitsanalyse heute unverzichtbar!
- Algorithmen für verschiedenste Anwendungen existieren
- meist ohne oder bestenfalls mit manuellem Korrektheitsbeweis

Quis custodiet ipsos custodes?

Kann ich sicher sein, dass der Algorithmus das tut, was er behauptet?

Ziel: Analyse und Verifikation von Sicherheitsanalysen

Analyse und Verifikation mittels formaler Semantik

Language Based Security: Verifikation mittels sprachlicher Eigenschaften

Vorliegende Semantik:

- exakte Formalisierung einer Sprache
- aussagekräftige Basis für Verifikation

Analyse und Verifikation mittels formaler Semantik

Language Based Security: Verifikation mittels sprachlicher Eigenschaften

Vorliegende Semantik:

- exakte Formalisierung einer Sprache
- aussagekräftige Basis für Verifikation
- ideale Grundlage für LBS!

Analyse und Verifikation mittels formaler Semantik

Language Based Security: Verifikation mittels sprachlicher Eigenschaften

Vorliegende Semantik:

- exakte Formalisierung einer Sprache
- aussagekräftige Basis für Verifikation
- ideale Grundlage für LBS!

Aktuelle und zukünftige Arbeit:

- Korrektheitsbeweise von Methoden der sprachbasierten Softwaresicherheit (z.B. Information Flow Control) mittels Theorembeweisern
- dazu nötig: Formalisierung von CFGs, PDGs etc...

Zusammenfassung

ausführbare Formalisierung von (Kern-)C++ und seiner Typsicherheit mittels eines Theorembeweisers

Zusammenfassung

ausführbare Formalisierung von (Kern-)C++ und seiner Typsicherheit mittels eines Theorembeweisers

$\longrightarrow$ Ziel erreicht!

Zusammenfassung

ausführbare Formalisierung von (Kern-)C++ und seiner Typsicherheit mittels eines Theorembeweisers

$\longrightarrow$ Ziel erreicht!

- vollständige Semantik und Typsystem für Kern-C++
- korrekte Behandlung von dynamischer Bindung und Subobjekt-Domination
- (maschinengeprüfter) Typsicherheitsbeweis
- Formalisierung unabhängig von Implementationsdetails
- neue Stufe der Komplexität in maschinellem Beweisen
Ein paar Zahlen: 14,000 LoC, $\sim$ 500 Lemmas, $\sim$ 120 Definitionen
Durchlauf des Beweis: $\sim$ 5min (Athlon 64 D.C. 3800+, 2GB RAM)
- Formalisierung ausführbar, C++ Code kann also geprüft werden
- ideale Basis für Verifikation von Sicherheitsanalysen