Formalization of Programming Languages

Define semantics of PL in proof assistant

문순원

증명언어 소개

형식증명 (formal proof)

- 증명을 형식적으로 표현
- ↔ 자연어 증명
- **기계로 검증할 수 있다**

증명언어 소개

형식증명 (formal proof)

- 증명을 형식적으로 표현
- ↔ 자연어 증명
- **기계로 검증할 수 있다**

증명보조기 (proof assistant)

- 형식증명의 작성을 도와주는 소프트웨어

증명언어 소개

형식증명 (formal proof)

- 증명을 형식적으로 표현
- ↔ 자연어 증명
- **기계로 검증할 수 있다**

증명보조기 (proof assistant)

- 형식증명의 작성을 도와주는 소프트웨어

증명언어 (proof language)

- 형식증명을 표현할 수 있는 프로그래밍 언어
- Agda, Coq, Isabelle, Lean 등

증명언어의 특징

Propositions as types

- 증명을 수학적 대상으로 다룬다
- 명제는 프로그램의 타입에 대응
- 명제의 증명은 프로그램에 대응

증명언어의 특징

Propositions as types

- 증명을 수학적 대상으로 다룬다
- 명제는 프로그램의 타입에 대응
- 명제의 증명은 프로그램에 대응

강타입 함수형 언어와 공유하는 특징 (e.g. Haskell, OCaml)

- Purely functional
- Strongly typed

증명언어의 특징

Propositions as types

- 증명을 수학적 대상으로 다룬다
- 명제는 프로그램의 타입에 대응
- 명제의 증명은 프로그램에 대응

강타입 함수형 언어와 공유하는 특징 (e.g. Haskell, OCaml)

- Purely functional
- Strongly typed

공유하지 않는 특징

- Total functional
- Dependent type

증명언어와 관련된 분야

증명언어에 대한 연구

- 수학, PL 분야의 공통적 연구대상
- 타입이론

증명언어와 관련된 분야

증명언어에 대한 연구

- 수학, PL 분야의 공통적 연구대상
- 타입이론

증명언어를 활용

- 수학 증명 형식화 (e.g. Four color theorem)
- 프로그래밍 언어 형식화

증명언어와 관련된 분야

증명언어에 대한 연구

- 수학, PL 분야의 공통적 연구대상
- 타입이론

증명언어를 활용

- 수학 증명 형식화 (e.g. Four color theorem)
- 프로그래밍 언어 형식화

증명언어와 관련된 분야

증명언어에 대한 연구

- 수학, PL 분야의 공통적 연구대상
- 타입이론

증명언어를 활용

- 수학 증명 형식화 (e.g. Four color theorem)
- 프로그래밍 언어 형식화

PL 형식화의 필요성

- 프로그래밍 언어 설계상의 결함을 제거
- 프로그램을 검증하기 위한 밑바탕

증명언어로 검증된 프로그램을 만드는 방법

프로그램을 증명언어로 작성

- 증명언어를 OCaml, Haskell, Scheme, JavaScript 등의 언어로 컴파일
- 비교적 쉽다

증명언어로 검증된 프로그램을 만드는 방법

프로그램을 증명언어로 작성

- 증명언어를 OCaml, Haskell, Scheme, JavaScript 등의 언어로 컴파일
- 비교적 쉽다

프로그램을 다른 언어로 작성

- C 언어를 증명언어에서 형식화
- 형식화된 대상언어로 작성된 프로그램을 증명언어로 검증
- 매우 어렵다

증명언어로 검증된 프로그램을 만드는 방법

프로그램을 증명언어로 작성

- 증명언어를 OCaml, Haskell, Scheme, JavaScript 등의 언어로 컴파일
- 비교적 쉽다

프로그램을 다른 언어로 작성

- C 언어를 증명언어에서 형식화
- 형식화된 대상언어로 작성된 프로그램을 증명언어로 검증
- 매우 어렵다

예시

- CompCert : Coq
- CertiKOS : C, Coq
- seL4 : C, Isabelle

프로그래밍 언어의 의미론

Denotational semantics

- 프로그램을 수학적 대상에 대응 시킨다
- 주로 FP언어에서 많이 사용

프로그래밍 언어의 의미론

Denotational semantics

- 프로그램을 수학적 대상에 대응 시킨다
- 주로 FP언어에서 많이 사용

Operational semantics

- 추상기계의 상태 전이를 통해 의미를 부여
- 명령형 언어에서 흔히 사용되는 접근

프로그래밍 언어의 의미론

Denotational semantics

- 프로그램을 수학적 대상에 대응 시킨다
- 주로 FP언어에서 많이 사용

Operational semantics

- 추상기계의 상태 전이를 통해 의미를 부여
- 명령형 언어에서 흔히 사용되는 접근

Axiomatic semantics

- 프로그램에 대한 공리들을 통해 프로그램을 검증
- Hoare logic 등

프로그래밍 언어의 의미론

Denotational semantics

- 프로그램을 수학적 대상에 대응 시킨다
- 주로 FP언어에서 많이 사용

Operational semantics

- 추상기계의 상태 전이를 통해 의미를 부여
- 명령형 언어에서 흔히 사용되는 접근

Axiomatic semantics

- 프로그램에 대한 공리들을 통해 프로그램을 검증
- Hoare logic 등

Operational semantics를 통해 Hoare logic의 공리들을 증명하는 것도 가능

Program logic

> *Verifiable C is based on a 21st-century version of Hoare logic called higher-order impredicative concurrent separation logic. Back in the 20th century, computer scientists discovered that Hoare Logic was not very good at verifying programs with pointer data structures; so separation logic was developed. Hoare Logic was clumsy at verifying concurrent programs, so concurrent separation logic was developed. Hoare Logic could not handle higher-order object-oriented programming patterns or function-closures, so higher-order impredicative program logics were developed.*
>
> — *Preface, Software Foundations volume 5*

- Hoare logic
- separation logic
- concurrent separation logic
- higher-order impredicative program logics

Behavioral refinement

$$\mathrm{Beh}(T) \subseteq \mathrm{Beh}(S)$$

- source S를 target T로 변환하는 것이 옳바른가?
- Compiler : S = 프로그래머가 작성한 소스코드, T = 컴파일러의 출력물
- Software verification : S = 추상화된 스펙, T = 검증 대상 프로그램

$$\mathrm{Beh}(T) \subseteq \mathrm{Beh}(S)$$

undefined behavior

- 전체 집합
- S에서 UB가 발생할 경우, 임의의 T에 대해 refinement가 성립
- T에서 UB가 발생할 경우, S에서 UB가 발생해야 refinement가 성립

$$\mathrm{Beh}(T) \subseteq \mathrm{Beh}(S)$$

undefined behavior

- 전체 집합
- S에서 UB가 발생할 경우, 임의의 T에 대해 refinement가 성립
- T에서 UB가 발생할 경우, S에서 UB가 발생해야 refinement가 성립

no behavior

- 공집합
- T에서 NB가 발생할 경우, 임의의 S에 대해 refinement가 성립
- S에서 NB가 발생할 경우, T에서 NB가 발생해야 refinement가 성립

Compiler

- T에서 NB를 발생시킬 경우 컴파일이 자명하게 옳바름
- target에서 NB를 발생시키는 것을 금지해야 한다

프로그래밍 언어의 의미론

Compiler

- T에서 NB를 발생시킬 경우 컴파일이 자명하게 옳바름
- target에서 NB를 발생시키는 것을 금지해야 한다

Software verification

- S에서 UB를 발생시킬 경우 증명 대상이 스펙을 자명하게 만족
- source에서 UB를 발생시키는 것을 금지해야 한다

Compiler

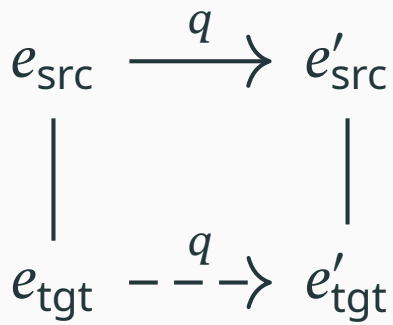
- T에서 NB를 발생시킬 경우 컴파일이 자명하게 옳바름
- target에서 NB를 발생시키는 것을 금지해야 한다

Software verification

- S에서 UB를 발생시킬 경우 증명 대상이 스펙을 자명하게 만족
- source에서 UB를 발생시키는 것을 금지해야 한다

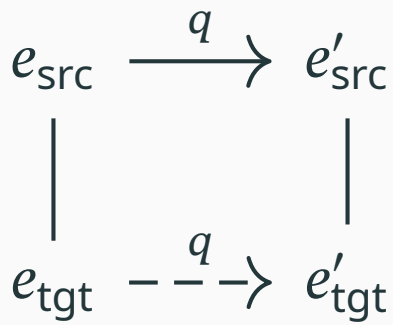
$$\mathrm{Beh}(\mathsf{Asm}) \subseteq \mathrm{Beh}(\mathsf{Imp}) \subseteq \mathrm{Beh}(\mathsf{Spec})$$

Simulation



- labeled state transition system 사이의 preorder
- Behavioral refinement를 증명하기 위한 중간 단계로 자주 사용됨

Simulation



- labeled state transition system 사이의 preorder
- Behavioral refinement를 증명하기 위한 중간 단계로 자주 사용됨

개인적인 감상

- 프로그램의 behavior는 event 만으로 정의됨
- event에 대해 논증하기 위해 state가 필요함

Program logic vs Behavioral refinement

Program logic

- VST (Verified Software Toolchain)
- Iris
- RefinedC

Behavioral refinement

- CompCert
- CertiKOS