Engineering Hoare-Logic based Program Verification in $\mathbb{K}$

Andrei Arusoaie

Faculty of Computer Science, Iași

SYNASC 2013

October 30, 2013

Motivation

- $\mathbb{K}$- (remember Grigore Roșu's talk)
- Framework for defining operational semantics of programming languages (PL)

Motivation

- $\mathbb{K}$- (remember Grigore Roșu's talk)
- Framework for defining operational semantics of programming languages (PL)
- We are interested in using the $\mathbb{K}$ semantics for program verification

Motivation

- $\mathbb{K}$- (remember Grigore Roșu's talk)
- Framework for defining operational semantics of programming languages (PL)
- We are interested in using the $\mathbb{K}$ semantics for program verification
- Reachability Logic (see MatchC approach)

Motivation

- $\mathbb{K}$- (remember Grigore Roșu's talk)
- Framework for defining operational semantics of programming languages (PL)
- We are interested in using the $\mathbb{K}$ semantics for program verification
- Reachability Logic (see MatchC approach)
- What about Hoare Logic?

Motivation

- What should we do (in practice) to a create a Hoare-like verifier using the $\mathbb{K}$ semantics of a PL?

Problem

- Given $\mathcal{S}$, the $\mathbb{K}$ semantics of a programming language $\mathcal{L}$

Problem

- Given $\mathcal{S}$, the $\mathbb{K}$ semantics of a programming language $\mathcal{L}$
- Provide a methodology, which applied to the semantics of L transforms the $\mathbb{K}$ tool into a Hoare-like verification tool

$\mathbb{K}$- remember Grigore Roșu's talk

- $\mathbb{K}$- framework for defining operational semantics of programming languages
- Imperative: IMP, SIMPLE (untyped, typed), **C**
- Object Oriented: KOOL (untyped, typed), **Java**, **Python**
- Assembly: SSRISC, Verilog
- Functional: Scheme, Haskell, OCaml

Defining languages in $\mathbb{K}$

1. Define $\mathcal{L}$'s syntax (BNF)

Defining languages in $\mathbb{K}$

1. Define $\mathcal{L}$'s syntax (BNF)
2. Define $\mathcal{L}$'s semantics
   1. configuration - nested structure of cells
   2. k - computations cell
   3. $\mathbb{K}$ rules - rewriting rules

IMP syntax

- IMP statements

$$
\begin{array}{lll}
\textsc{syntax} & \mathit{Stmt} ::= & \{\} \mid \{\mathit{Stmt}\} \\
& \mid & \mathit{Id} = \mathit{AExp}\ ;\ \text{[strict(2)]} \\
& \mid & \texttt{if}\ (\mathit{BExp})\mathit{Block}\ \texttt{else}\ \mathit{Block}\ \text{[strict(1)]} \\
& \mid & \texttt{while}\ (\mathit{BExp})\mathit{Block} \\
& \mid & \mathit{Stmt}\ \ \mathit{Stmt}
\end{array}
$$

IMP syntax

- IMP statements

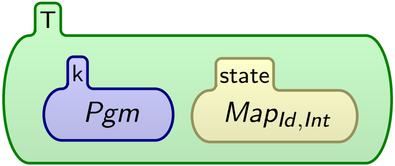
$$\begin{array}{lll}\textsc{syntax} & \mathit{Stmt} ::= & \{\} \mid \{\mathit{Stmt}\} \\ & \mid & \mathit{Id} = \mathit{AExp}\ ;\ [\text{strict}(2)] \\ & \mid & \texttt{if}\ (\mathit{BExp})\mathit{Block}\ \texttt{else}\ \mathit{Block}\ [\text{strict}(1)] \\ & \mid & \texttt{while}\ (\mathit{BExp})\mathit{Block} \\ & \mid & \mathit{Stmt}\ \ \mathit{Stmt}\end{array}$$

- Sample program `SUM`:

```
S = 0; i = 1;
while(i <= n) {
   S = S + i;
   i = i + 1;
}
```

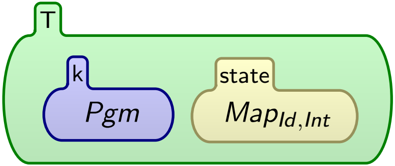
IMP configuration

IMP configuration:

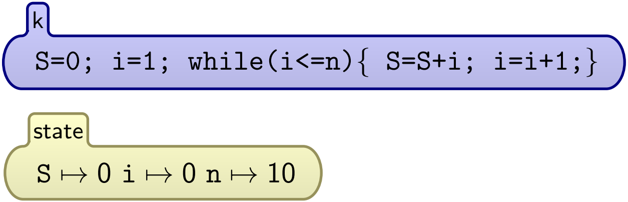


IMP configuration

IMP configuration:

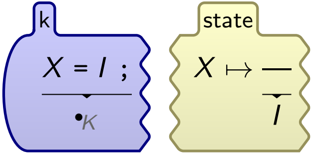


Concrete configuration:



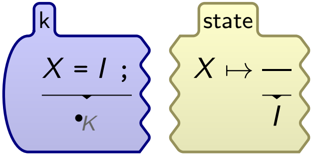
IMP semantics

Assignment:

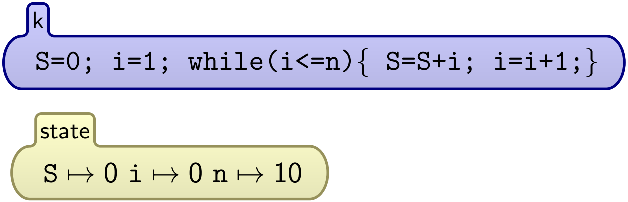


IMP semantics

Assignment:

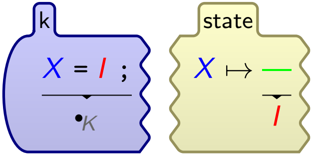


Concrete configuration:

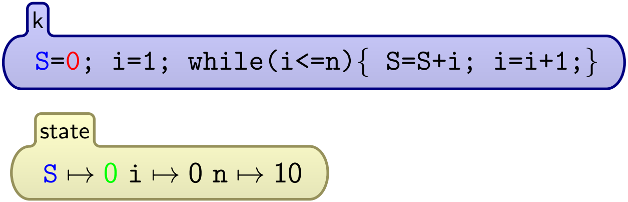


IMP semantics

Assignment:



Concrete configuration:



IMP semantics

Assignment:

k $\dfrac{X = I\ ;}{\bullet_K}$ **state** $X \mapsto \dfrac{\_}{I}$

Concrete configuration:

k
`i=1; while(i<=n){ S=S+i; i=i+1;}`

state
`S` $\mapsto$ 0 `i` $\mapsto$ 0 `n` $\mapsto$ 10

IMP semantics

Assignment:

$$\left\langle \frac{X = I\ ;}{\cdot_K} \cdots \right\rangle_{\mathsf{k}} \left\langle \cdots\ X \mapsto \frac{\_}{I}\ \cdots \right\rangle_{\mathsf{state}}$$

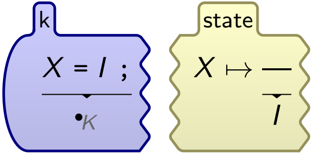
Concrete configuration:

$$\left\langle \texttt{i=1; while(i<=n)\{ S=S+i; i=i+1;\}} \right\rangle_{\mathsf{k}}$$

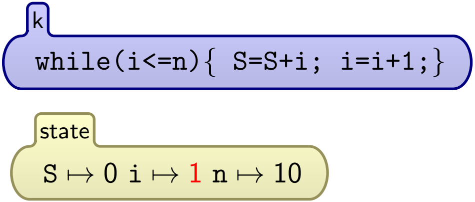
$$\left\langle \texttt{S} \mapsto 0\ \ \texttt{i} \mapsto 0\ \ \texttt{n} \mapsto 10 \right\rangle_{\mathsf{state}}$$

IMP semantics

Assignment:



Concrete configuration:



Creating Hoare Logic-based verification tools in $\mathbb{K}$

- In practice, a Hoare Logic verifier is language dependent

Creating Hoare Logic-based verification tools in $\mathbb{K}$

- In practice, a Hoare Logic verifier is language dependent
  - deductive rules depend on the language syntactical constructs
  - syntax of the logical assertions
  - ...

Creating Hoare Logic-based verification tools in $\mathbb{K}$

- In practice, a Hoare Logic verifier is language dependent
  - deductive rules depend on the language syntactical constructs
  - syntax of the logical assertions
  - ...
- We have to deal with

Creating Hoare Logic-based verification tools in $\mathbb{K}$

- In practice, a Hoare Logic verifier is language dependent
  - deductive rules depend on the language syntactical constructs
  - syntax of the logical assertions
  - ...
- We have to deal with
  - language dependency issues (syntax & semantics)

Creating Hoare Logic-based verification tools in $\mathbb{K}$

- In practice, a Hoare Logic verifier is language dependent
  - deductive rules depend on the language syntactical constructs
  - syntax of the logical assertions
  - ...
- We have to deal with
  - language dependency issues (syntax & semantics)
  - support for verification of Hoare triples in $\mathbb{K}$ (transform Hoare triples into Reachability formulas)

Hoare Logic vs. Reachability logic - example

- SUM = "`S=0; i=1; while(i<=n){ S=S+i; i=i+1;}`"

Hoare Logic vs. Reachability logic - example

- $\mathtt{SUM} =$ "`S=0; i=1; while(i<=n){ S=S+i; i=i+1;}`"

- Hoare triple:

$$\{\mathtt{n} \geq 0\}\ \mathtt{SUM}\ \{\mathtt{S} = \mathtt{n} * (\mathtt{n} + 1)/2\}$$

Hoare Logic vs. Reachability logic - example

- SUM = "`S=0; i=1; while(i<=n){ S=S+i; i=i+1;}`"
- Hoare triple:

$$\{\mathtt{n} \geq 0\}\ \mathtt{SUM}\ \{\mathtt{S} = \mathtt{n} * (\mathtt{n} + 1)/2\}$$

- Reachability rule:

$$\langle \mathtt{SUM} \rangle_{\mathsf{k}}\ \langle \mathtt{S} \mapsto s, \mathtt{n} \mapsto n, \mathtt{i} \mapsto i \rangle_{\mathsf{state}} \wedge n \geq 0 \Rightarrow$$

$$\langle \cdot \rangle_{\mathsf{k}}\ \langle \mathtt{S} \mapsto s', \mathtt{n} \mapsto n', \mathtt{i} \mapsto i' \rangle_{\mathsf{state}} \wedge s' = n * (n + 1)/\ 2$$

Hoare Logic vs. Reachability logic - example

- $\texttt{SUM} = $ "`S=0; i=1; while(i<=n){ S=S+i; i=i+1;}`"

- Hoare triple:

$$\{\mathtt{n} \geq 0\}\ \mathtt{SUM}\ \{\mathtt{S} = \mathtt{n} * (\mathtt{n}+1)/2\}$$

- Reachability rule:

$$\langle \mathtt{SUM} \rangle_{\mathsf{k}} \langle \mathtt{S} \mapsto s, \mathtt{n} \mapsto n, \mathtt{i} \mapsto i \rangle_{\mathsf{state}} \wedge n \geq 0 \Rightarrow$$

$$\langle \bullet \rangle_{\mathsf{k}} \langle \mathtt{S} \mapsto s', \mathtt{n} \mapsto n', \mathtt{i} \mapsto i' \rangle_{\mathsf{state}} \wedge s' = n * (n+1)/\ 2$$

- Grigore Roșu and Andrei Ștefănescu.
  From hoare logic to matching logic reachability.
  FM'12

Verification of Hoare triples using symbolic execution

- Intuition: transform Hoare triples into reachability rules and verify them using symbolic execution

Verification of Hoare triples using symbolic execution

- Intuition: transform Hoare triples into reachability rules and verify them using symbolic execution
- If $\varphi \Rightarrow \varphi'$, where $\varphi = \pi \wedge \psi$, then

Verification of Hoare triples using symbolic execution

- Intuition: transform Hoare triples into reachability rules and verify them using symbolic execution
- If $\varphi \Rightarrow \varphi'$, where $\varphi = \pi \wedge \psi$, then
  - `assume` φ: π the initial configuration, ψ the initial path condition

Verification of Hoare triples using symbolic execution

- Intuition: transform Hoare triples into reachability rules and verify them using symbolic execution
- If $\varphi \Rightarrow \varphi'$, where $\varphi = \pi \wedge \psi$, then
  - `assume` φ: π the initial configuration, ψ the initial path condition
  - perform symbolic execution of φ, and obtain $\{\varphi''\}$

Verification of Hoare triples using symbolic execution

- Intuition: transform Hoare triples into reachability rules and verify them using symbolic execution
- If $\varphi \Rightarrow \varphi'$, where $\varphi = \pi \wedge \psi$, then
  - `assume` φ: π the initial configuration, ψ the initial path condition
  - perform symbolic execution of φ, and obtain $\{\varphi''\}$
  - check if for all φ'', φ'' `implies` φ'

Methodology

- Steps:

Methodology

- Steps:
  - Add syntax and semantics for Hoare-like annotations

Methodology

- Steps:
  - Add syntax and semantics for Hoare-like annotations
  - Define `assume` & `implies` operations for checking reachability rules

Add syntax for Hoare-like annotations

```
pre:  0 <= n
post:  2 * S == n * (n + 1)
S = 0;
i = 1;
while (i <= n)
inv:  2*S == i * (i - 1) and i <= n + 1
{
  S = S + i;
  i = i + 1;
}
```

Semantics of the new constructs

- Pre/post conditions

$$\frac{\texttt{pre:}\ \psi\ \texttt{post:}\ \psi'\ \mathit{Stmt}}{\texttt{assume}\ (\langle \mathit{Stmt} \rangle_{\mathsf{k}}\ \langle \sigma \rangle_{\mathsf{state}} \wedge \psi) \curvearrowright \texttt{implies}\ (\langle \cdot \rangle_{\mathsf{k}}\ \langle \sigma' \rangle_{\mathsf{state}} \wedge \psi')}$$

Semantics of the new constructs

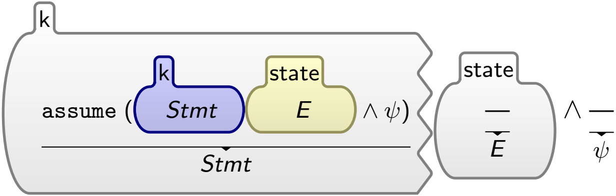
- `while` loop

$$\frac{\texttt{while}(B)\texttt{inv:}\psi\ \mathit{Stmt} \curvearrowright K}{\texttt{assume}(\langle \mathit{Stmt} \rangle_{\mathsf{k}}\ \langle \sigma \rangle_{\mathsf{state}} \wedge B \wedge \psi) \curvearrowright \texttt{implies}(\langle \bullet \rangle_{\mathsf{k}}\ \langle \sigma' \rangle_{\mathsf{state}} \wedge \psi)}$$

$$\frac{\texttt{while}(B)\texttt{inv:}\psi\ \mathit{Stmt} \curvearrowright K}{\texttt{implies}(\langle \bullet \rangle_{\mathsf{k}}\ \langle \sigma' \rangle_{\mathsf{state}} \wedge \psi) \curvearrowright \texttt{assume}(\langle K \rangle_{\mathsf{k}}\ \langle \sigma \rangle_{\mathsf{state}} \wedge \neg B \wedge \psi)}$$

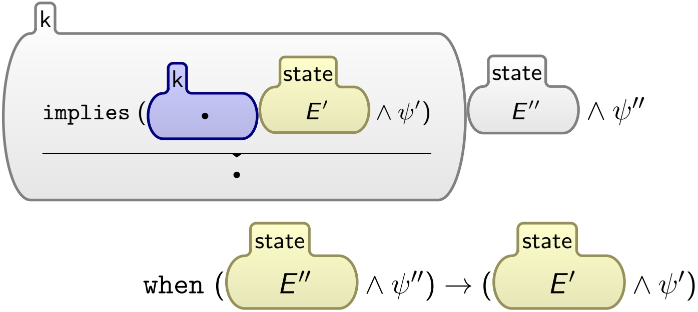
Define `assume`

- `assume` semantics:



Define `implies`

- `implies` semantics:



Conclusions

- Methodology to create a Hoare Logic verifier in $\mathbb{K}$

Conclusions

- Methodology to create a Hoare Logic verifier in $\mathbb{K}$
- Prove reachability rules generated from Hoare triples, using symbolic execution

Conclusions

- Methodology to create a Hoare Logic verifier in $\mathbb{K}$
- Prove reachability rules generated from Hoare triples, using symbolic execution
- Future work: eliminate `assume` and `implies`

Thank you!

The $\mathbb{K}$ framework webpage:

`http://kframework.org/`