Karlsruhe Institute of Technology

Technische Universität München

Animating the Formalised Semantics of a Java-Like Language

Andreas Lochbihler, Lukas Bulwahn

A case study for code generation

Code generation from Isabelle/HOL

- Does it work in the large?
- What are the pitfalls?
- How efficient is the generated code?

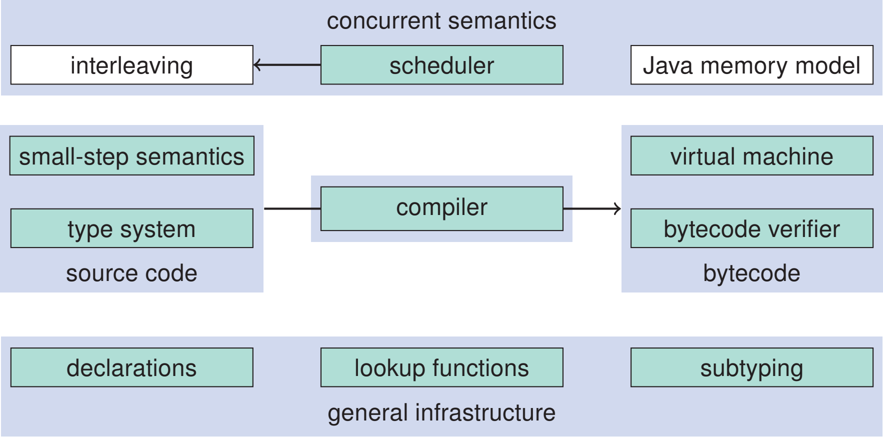
Case study JinjaThreads:

- formal semantics for multithreaded Java
- 80k lines of definitions & proofs
- uses wide range of Isabelle features
- validate the semantics by executing test cases

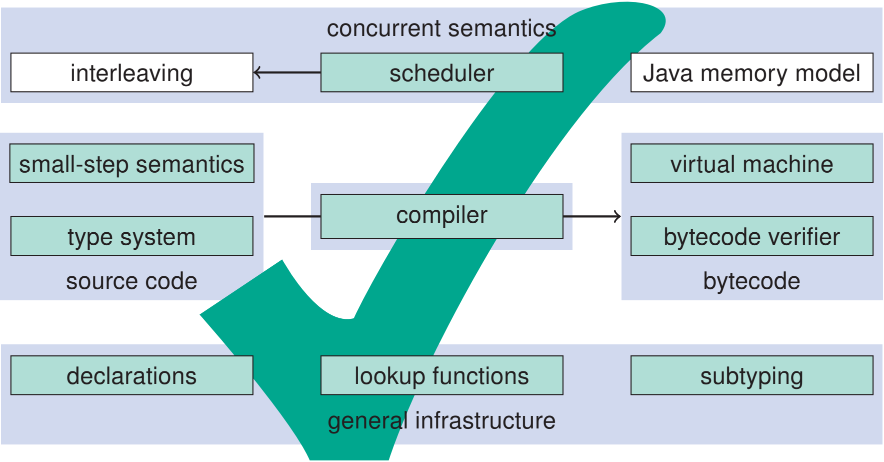
Outline

1. background
   - JinjaThreads
   - code generation in Isabelle
2. pitfalls
   - underspecification
   - inductive predicates
3. efficiency

JinjaThreads overview



JinjaThreads overview



Code generation in Isabelle

```
fun is_prefix where
      is_prefix [] ys = True
    | is_prefix (x#xs) [] = False
    | is_prefix (x#xs) (y#ys) = (x = y ∧ is_prefix xs ys)
```

```
export_code is_prefix in SML
```

```
fun is_prefix A_ [] ys = true
| is_prefix A_ (x :: xs) [] = false
| is_prefix A_ (x :: xs) (y :: ys) =
    HOL.eq A_ x y andalso is_prefix A_ xs ys;
```

Program refinement

execution is rewriting with unconditional equations

$\Rightarrow$ code generation partially correct w.r.t. all models of HOL

definition is_prefix $xs\ ys = (\exists zs.\ ys = xs\ @\ zs)$

lemma is_prefix [] ys = True

is_prefix $(x\#xs)$ [] = False

is_prefix $(x\#xs)$ $(y\#ys)$ = $(x = y \wedge$ is_prefix $xs\ ys)$

export_code is_prefix **in SML**

```
fun is_prefix A_ [] ys = true
| is_prefix A_ (x :: xs) [] = false
| is_prefix A_ (x :: xs) (y :: ys) =
    HOL.eq A_ x y andalso is_prefix A_ xs ys;
```

Data refinement

execution is rewriting with unconditional equations

datatype α list = [] | α # α list
definition is_prefix *xs ys* = ($\exists$*zs*. *ys* = *xs* @ *zs*)
definition Lazy :: (unit $\Rightarrow$ ($\alpha \times \alpha$ list) option) $\Rightarrow$ α list where ...
lemma is_prefix (Lazy *xs*) (Lazy *ys*) = ...

export_code is_prefix in SML

```
datatype 'a list = Lazy of (unit -> ('a * 'a list) option);
fun is_prefix A_ (Lazy xs) (Lazy ys) = ...
```

Predicate compiler

inductive definitions for type systems, semantics, ...

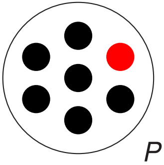
$$\frac{\Gamma\ V = \lfloor T \rfloor \qquad \Gamma \vdash e :: U \qquad U :\leq T}{\Gamma \vdash V := e :: \mathrm{Void}}$$

code_pred
  (modes: i ⇒ i ⇒ o ⇒ bool as infer_type,
          i ⇒ i ⇒ i ⇒ bool as type_check)
  _ ⊢ _ :: _

⇒ type inference: infer_type Γ e = { T. $\Gamma \vdash e :: T$ }

\+ functional equations for code generation

The pitfalls – underspecification

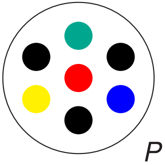


Task: Define a function to pick an arbitrary element of a set P.

- axiomatize Hilbert's ε-operator $$\frac{P\ y}{P\ (\varepsilon x.\ P\ x)}$$
- models of HOL must define $\varepsilon x.\ P\ x$ for all P
  but code generator correct w.r.t. *all* models

$\Rightarrow$ execution may only succeed if P is a singleton

The pitfalls – underspecification

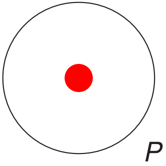


Task: Define a function to pick an arbitrary element of a set P.

- axiomatize Hilbert's ε-operator $$\frac{P\ y}{P\ (\varepsilon x.\ P\ x)}$$
- models of HOL must define $\varepsilon x.\ P\ x$ for all P
  but code generator correct w.r.t. *all* models

$\Rightarrow$ execution may only succeed if P is a singleton

The pitfalls – underspecification



Task: Define a function to pick an arbitrary element of a set *P*.

- axiomatize Hilbert's ε-operator $$\frac{P\ y}{P\ (\varepsilon x.\ P\ x)}$$
- models of HOL must define $\varepsilon x.\ P\ x$ for all P
  but code generator correct w.r.t. *all* models

$\Rightarrow$ execution may only succeed if P is a singleton

Avoid Hilbert's ε-operator! – solution 1

Example: Find a fresh address for memory allocation

$$\text{definition new\_Addr } h = (\varepsilon a.\ h\ a = \text{None})$$

Avoid Hilbert's ε-operator! – solution 1

Example: Find a fresh address for memory allocation

definition new_Addr h = ($\varepsilon a.\ h\ a$ = None)

Solutions:

1. manual translation to code
   – potentially unsound (some proofs trust code evaluation)

Avoid Hilbert's ε-operator! – solution 1

Karlsruhe Institute of Technology

Example: Find a fresh address for memory allocation

definition new_Addr h = ($\varepsilon a.\ h\ a$ = None)

Solutions:

1. ~~manual translation to code~~
   - ~~potentially unsound (some proofs trust code evaluation)~~

2. make definition fully specified & implement search algorithm

   definition new_Addr h = (LEAST $a.\ h\ a$ = None)
   lemma new_Addr h = find_least h 0
   find_least $h\ a$ = . . .

   + local change
   – change of definition

Avoid Hilbert's ε-operator! – solution 2

Example: Notify thread in wait set of monitor m

$$\text{upd\_wset}\ ws\ (\text{Notify}\ m) = ws(m := ws\ m - (\varepsilon t.\ t \in ws\ m))$$

Avoid Hilbert's ε-operator! – solution 2

Example: Notify thread in wait set of monitor *m*

$$\text{upd\_wset}\ ws\ (\text{Notify}\ m) = ws(m := ws\ m - (\varepsilon t.\ t \in ws\ m))$$

Solution:

- switch from function to relation

$$\frac{t \in ws\ m}{\text{upd\_wset}\ ws\ (\text{Notify}\ m)\ (ws(m := ws\ m - t))}$$

- replace model-theoretic underspecification with intra-logical non-determinism
  - – refactoring of all dependent definitions and proofs
  - + allows more proofs (e.g. irrelevance of concrete choice)
  - + supports specification refinement (scheduler)

Avoid Hilbert's ε-operator! – solution 3

Example: Kildall's work list algorithm

$$\text{kildall} = \ldots \text{while } (\lambda(\tau s, w).\ w \neq \emptyset)\ (\lambda(\tau s, w).\ \ldots(\varepsilon x.\ x \in w)\ \ldots)\ \ldots$$

Avoid Hilbert's ε-operator! – solution 3

Example: Kildall's work list algorithm

$$\text{kildall} = \ldots \text{while } (\lambda(\tau s, w).\ w \neq \emptyset)\ (\lambda(\tau s, w).\ \ldots (\varepsilon x.\ x \in w) \ldots) \ldots$$

Solution:

1. choice function *ch* as additional parameter

   $$\text{kildall } ch = \ldots \text{while } (\lambda(\tau s, w).\ w \neq \emptyset)\ (\lambda(\tau s, w).\ \ldots (ch\ w) \ldots) \ldots$$

   - \+ retains functional style
   - – no polymorphism

2. hide parameter & assumption in locale

   locale kildall_choice = fixes *ch* :: … assumes $w \neq \emptyset \Longrightarrow ch\ w \in w$

   $$\text{kildall} = \ldots \text{while } (\lambda(\tau s, w).\ w \neq \emptyset)\ (\lambda(\tau s, w).\ \ldots (ch\ w) \ldots) \ldots$$

Annotate your predicates with modes!

Example: Type checking & type inference for _ ⊢ _ :: _

$i \Rightarrow i \Rightarrow i \Rightarrow bool$ $\quad$ $i \Rightarrow i \Rightarrow o \Rightarrow bool$

$$\frac{\Gamma\ V = \lfloor T \rfloor \qquad \Gamma \vdash e :: U \qquad U :\leq T}{\Gamma \vdash V := e :: \mathrm{Void}}$$

Annotate your predicates with modes!

Example: Type checking & type inference for _ ⊢ _ :: _

i ⇒ i ⇒ i ⇒ bool i ⇒ i ⇒ o ⇒ bool

$$\frac{\Gamma\ V = \lfloor T \rfloor \qquad \Gamma \vdash e :: U \qquad U :\leq T}{\Gamma \vdash V := e :: \text{Void}}$$

Annotate your predicates with modes!

Example: Type checking & type inference for _ ⊢ _ :: _

i ⇒ i ⇒ i ⇒ bool   i ⇒ i ⇒ o ⇒ bool

$$\frac{\Gamma\ V = \lfloor T \rfloor \qquad \Gamma \vdash e :: U \qquad U :\leq T}{\Gamma \vdash V := e :: \mathrm{Void}}$$

Annotate your predicates with modes!

Example: Type checking & type inference for $\_ \vdash \_ :: \_$

$\mathrm{i} \Rightarrow \mathrm{i} \Rightarrow \mathrm{i} \Rightarrow \mathrm{bool}$ $\quad$ $\mathrm{i} \Rightarrow \mathrm{i} \Rightarrow \mathrm{o} \Rightarrow \mathrm{bool}$

$$\frac{\Gamma\ V = \lfloor T \rfloor \qquad \Gamma \vdash e :: U \qquad U \leq T}{\Gamma \vdash V := e :: \mathrm{Void}}$$

→ infer *e*'s type, check subtyping

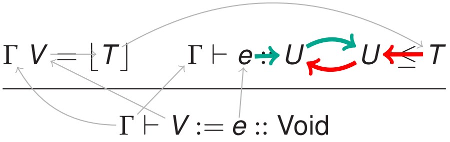
→ enumerate subtypes, type check *e*

does not terminate

Annotate your predicates with modes!

Example: Type checking & type inference for $\_ \vdash \_ :: \_$

$i \Rightarrow i \Rightarrow i \Rightarrow bool \quad i \Rightarrow i \Rightarrow o \Rightarrow bool$



- infer *e*'s type, check subtyping
- enumerate subtypes, type check *e*
  does not terminate

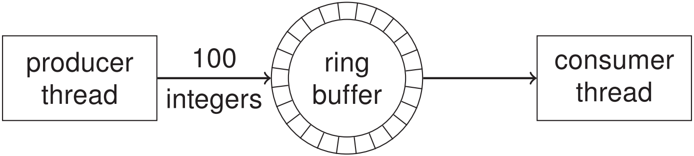
Solution:

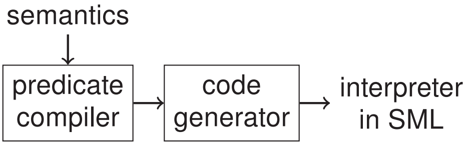
- mode annotations disallow non-terminating modes

⇒ mode checking faster than mode inference

Efficiency

1.





default setup	37.3 min
optimised setup	.6 s

- clause indexing
- remove laziness
- red-black trees
- tabulation

2. Parallel factorial [Lui, Moore: M6]

source code	26.7 s
M6 (ACL2 2.7)	6.2 s
virtual machine	.2 s

Conclusion

Code generation works in the large, but

- not yet push-button
- know the dark corners and how to avoid them

Validation of semantics

- found bug in division & modulo operator implementation

Conclusion

Code generation works in the large, but

- not yet push-button
- know the dark corners and how to avoid them

Validation of semantics

- found bug in division & modulo operator implementation

How do you do underspecification right?