Karlsruhe Institute of Technology

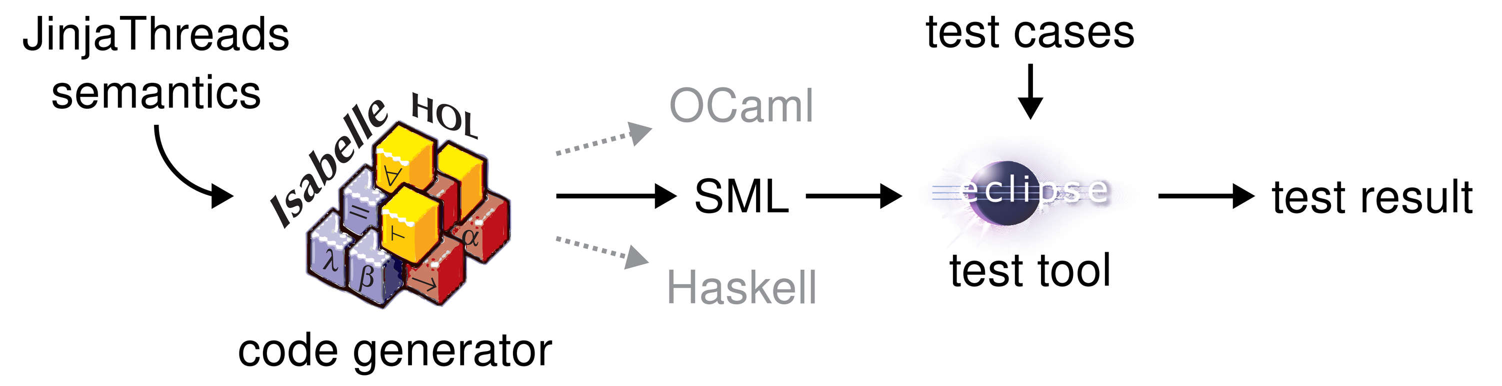
Technische Universität München

Animating the Formalised Semantics of a Java-like Language*

Andreas Lochbihler, Lukas Bulwahn

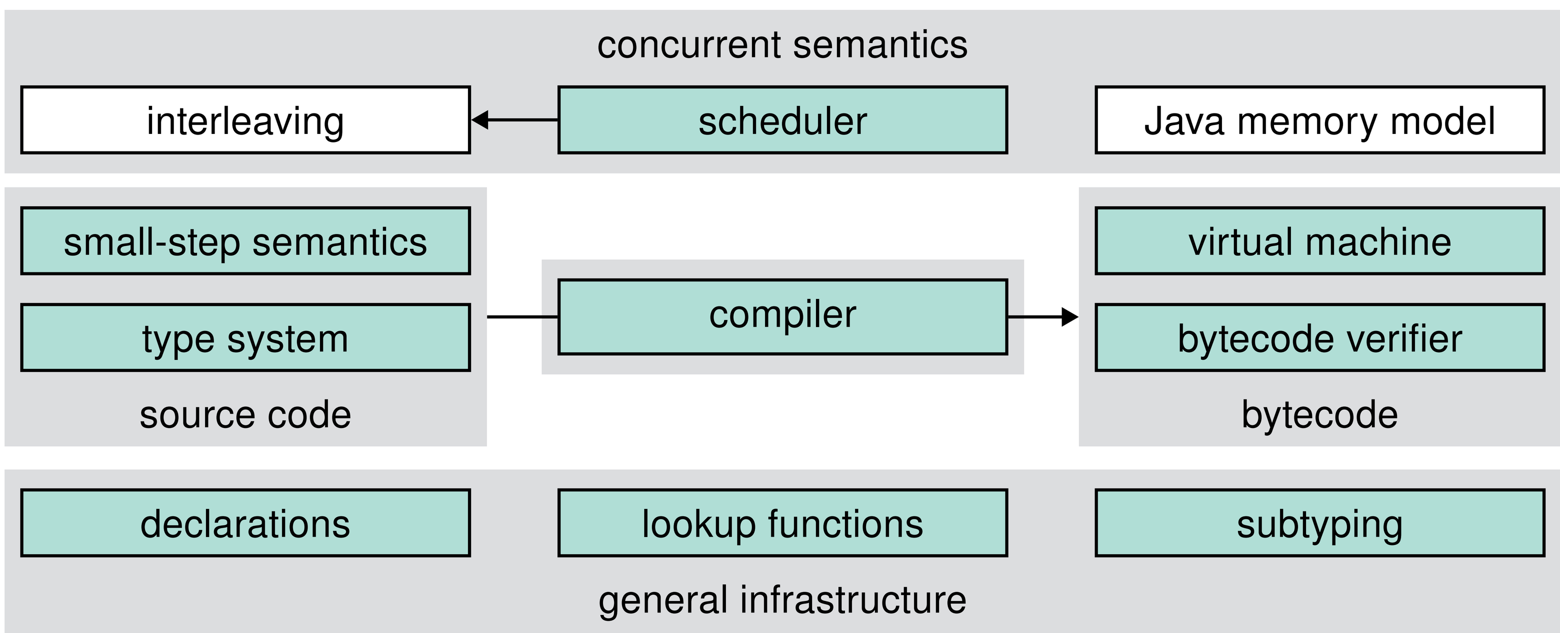
Goals

- Study whether code generation works in the large.
- Validate the semantics by running test programs from test suites such as Jacks [1], OpenJDK [4], and Jbook [5].



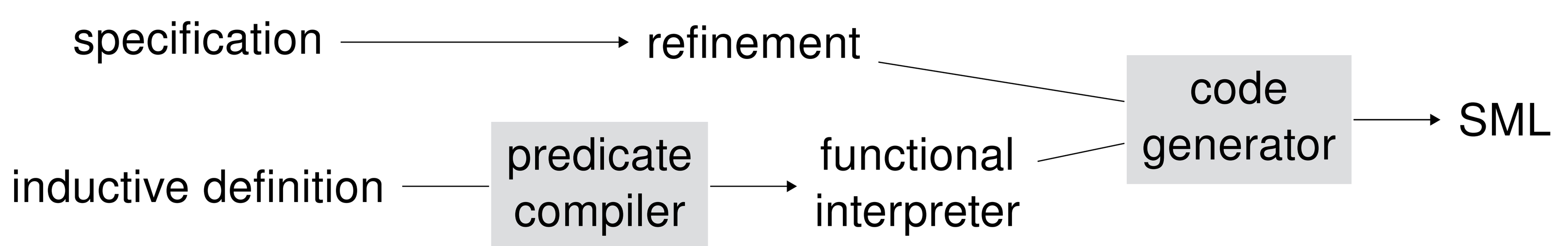
JinjaThreads

JinjaThreads [2] models a substantial subset of multithreaded Java source and bytecode. Executability was of little concern throughout its development. Now, we have generated code via Isabelle's code generator for all definitions in green boxes in the structure diagram below.



Code Generation in Isabelle

Execution is rewriting with unconditional equations.



Correctness Code generation partially correct w.r.t. *all* models of HOL, because rewriting in the logic can simulate the execution.

Program Refinement *Locally* derive new (code) equations to use upon code generation, as any (executable) equational theorem suffices for code generation. Existing definitions and proofs remain unaffected.

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 $xs\ ys)$

Data Refinement Replace constructors of a datatype by other constants and derive equations for code generation that pattern-match on these new (pseudo-)constructors.

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

Inductive Definitions Generate from inductive definitions (type systems, operational semantics) code equations for a functional interpreter.

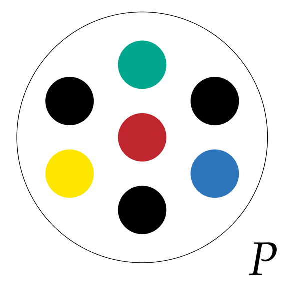
$$\frac{\Gamma\ V = \lfloor T \rfloor \qquad \Gamma \vdash e :: U \qquad U :\leq T}{\Gamma \vdash V := e :: \text{Void}}$$

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

Reasons for Non-Executability

The original specifications contained inherently non-executable parts (Hilbert's ε-operator). We replaced them by

- appropriately modelling underspecification and refinement, or
- new specifications.



$$\frac{P\ y}{P\ (\varepsilon x.\ P\ x)}$$

We developed and applied three solutions:

Solution 1: Change definition to full specification

Example: Find a fresh address for memory allocation
Replace **definition** new_Addr h = ($\varepsilon a.\ h\ a$ = None)
with **definition** new_Addr h = (LEAST $a.\ h\ a$ = None)
and implement as **lemma** new_Addr h = find_least h 0
find_least $h\ a$ = ...

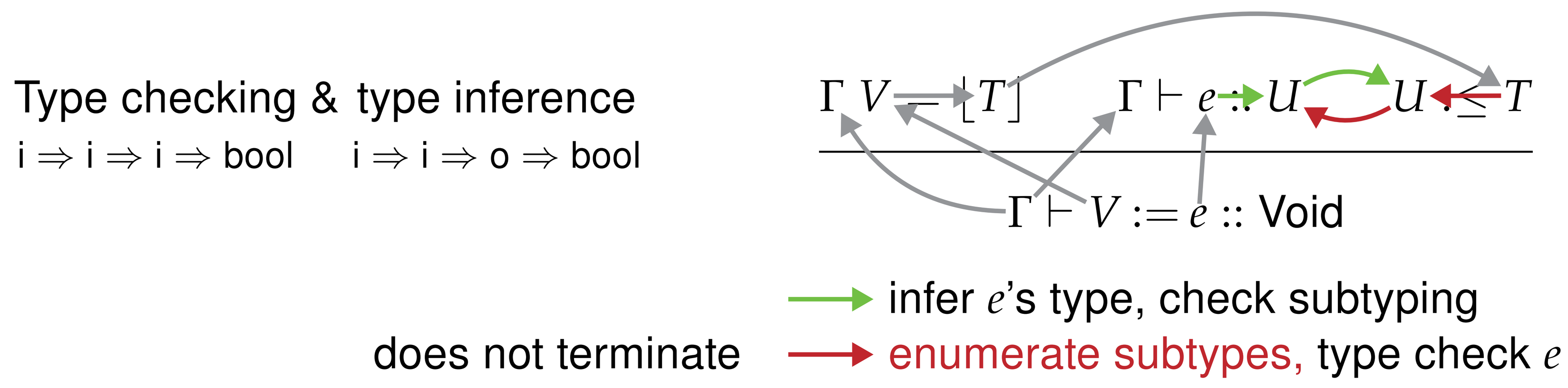
Solution 2: Switch from function to relation

Example: Notify thread in wait set of monitor m
Replace upd_wset ws (Notify m) = $ws(m := ws\ m - (\varepsilon t.\ t \in ws\ m))$
with
$$\frac{t \in ws\ m}{\text{upd\_wset}\ ws\ (\text{Notify}\ m)\ (ws(m := ws\ m - t))}$$

Solution 3: Parametrize the choice function

Example: Kildall's work list algorithm
Replace **definition** kildall = while $(\lambda(\tau,w).\ w \neq \emptyset)\ (\lambda(\tau,w).\ \ldots(\varepsilon x.\ x \in w)\ \ldots)$
with **locale** kildall_choice = fixes ch :: ... assumes $w \neq \emptyset \Longrightarrow ch\ w \in w$
definition (in kildall_choice)
kildall = while $(\lambda(\tau s, w).\ w \neq \emptyset)\ (\lambda(\tau s, w).\ \ldots(ch\ w)\ \ldots)$
interpretation kildall_choice <concrete choice implementation>

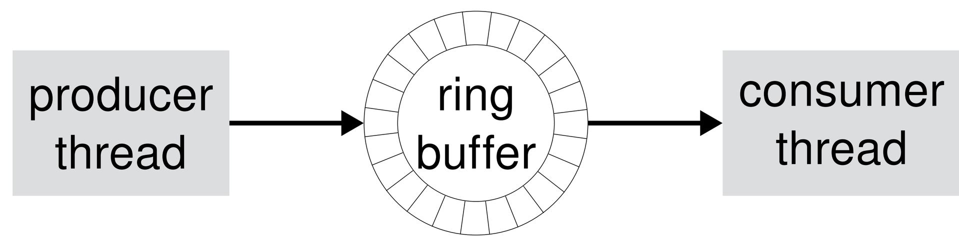
Mode Annotations Guide Program Synthesis



- Disallow non-terminating modes through mode annotations.
- Gain better performance as mode checking is faster than mode inference.

Efficiency

Run times (in seconds) for running a producer-consumer program on n integer objects for different adjustments to the interpreter; — denotes timeout after 1h.



n	without adjustments	with clause indexing	almost strict	heap as red-black tree	with tabulation
10	229.9	1.9	.1	<.1	<.1
100	2,240.3	14.1	1.7	.7	.6
1,000	—	625.6	492.3	7.2	6.2
10,000	—	—	—	71.8	62.6

References

[1] Jacks is an automated compiler killing suite, November 2005. http://sourceware.org/cgi-bin/cvsweb.cgi/~checkout~/jacks/jacks.html?cvsroot=mauve.

[2] A. Lochbihler. Jinja with threads. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs.* http://afp.sourceforge.net/entries/JinjaThreads.shtml, 2011. Formal proof development.

[3] A. Lochbihler and L. Bulwahn. Animating the formalised semantics of a Java-like language. In *Interactive Theorem Proving*, volume 6898 of *LNCS*, pages 216–232. Springer, 2011.

[4] OpenJDK 6. http://openjdk.java.net/.

[5] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine.* Springer, 2001.

*This work has been published at ITP 2011[3]. We acknowledge funding from DFG grants Sn11/10-1,2 and DFG doctorate program 1480 (PUMA).