Jakarta

a tool support for formal verification

Gilles Barthe
Pierre Courtieu
Guillaume Dufay
Marieke Huisman
Simão Melo de Sousa
Sorin Stratulat

FirstName.LastName@sophia.inria.fr

INRIA Sophia-Antipolis
France
Talk Overview

- Background
- Presentation of the JaKarTa toolset
- JaKarTa’s Preliminary Results
- Conclusion and Perspectives
Background


- Definitions are (a bit) cluttered and difficult to modify
- Case-distinctions make proofs tedious
- Low level of automation (both in proofs and writing specification)
- Difficult to make variations on the specification (such as abstractions)

However, all these problems are not insurmountable: JaKarTa is designed to provide solutions.
JaKarTa, a toolset for formal verification

A dedicated specification language can have a positive impact on formal specification and formal verification.

Designed with the following goals in mind

- Clarity and Executability of specifications
- Specifications easily transformable
- Tool independence
- Support for partial functions (by automatic transformation into total functions)
- Proof automation (generation of inversion principles)
- Support for refinement and abstractions
JaKarTa basic architecture

JaKarTa Basic Architecture
JSL: *JaKarTa Specification Language*

- JSL types: first-order polymorphic types
- JSL expressions: first-order algebraic terms built from
  - variables
  - constructor symbols (data type declarations)
  - defined symbols (function definitions)

\[ E ::= \mathcal{V} | E ::= E | c \tilde{E} | f \tilde{E} \]
JSL: JaKarTa Specification Language

Functions defined by set of rewrite rules:

*Constructor-based oriented conditional rewriting with extra variables*

\[ l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n \Rightarrow g \rightarrow d \]

- \( g = f \bar{x} \) with all \( x_i \in \mathcal{V} \) pairwise distinct
- \( r_i \) are patterns with fresh variables
- \( \text{var}(l_i) \subseteq \text{var}(g) \cup \text{var}(r_1) \cup \ldots \cup \text{var}(r_{i-1}) \)
- \( \text{var}(r_i) \cap \text{var}(g) = \emptyset \) and \( \forall i,j. \text{var}(r_i) \cap \text{var}(r_j) = \emptyset \)
function take : nat -> list 'a -> list 'a :=
n -> Zero => (take n l) -> Nil;
n -> (Succ m),
l -> (Cons hd tl) => (take n l) -> (Cons hd (take m tl)).

- Partial function
- First condition of second rule: binds m
- Second condition binds hd and tl
- Result uses fresh variables
Interaction with other tools

2 internal data representations:
- Jast = JaKarTa Abstract Syntax Tree
- JIR = JaKarTa Intermediate Representation = Jast complemented by a tree structured case distinction

2 kinds of translation:
- To "rewrite rule" based tools (such as ELAN or SPIKE): Translation from Jast to the target language.
- To tools with tree structured case distinction languages (COQ, PVS or Ocaml): translation from JIR to the target language. For Coq, a "two-ways" translation is provided.

JAK: JaKarTa Automation Kit

- Automatic generation of appropriate theorems to be used in formal verification
- Tailored towards specific theorem prover
- JAK’s Current Focus: generation of adequate inversion principles for functions. This is particularly useful for Coq proofs.
JTK: *JaKarTa Transformation Kit*

- For each datatype $\sigma$ define $\hat{\sigma}$ and $\lfloor . \rfloor_\sigma : \sigma \to \hat{\sigma}$
- For each defined function $f : \sigma \to \tau$, define $\hat{f} : \hat{\sigma} \to \hat{\tau}$ by transforming

$$l_1 \mapsto r_1, \ldots, l_n \mapsto r_n \Rightarrow g \mapsto d$$

into

$$\lfloor l_1 \rfloor \mapsto \lfloor r_1 \rfloor, \ldots, \lfloor l_n \rfloor \mapsto \lfloor r_n \rfloor \Rightarrow \lfloor g \rfloor \mapsto \lfloor d \rfloor$$

- Not a legal rule: substitution and cleaning required
The user can:

- introduce their own solution to local abstraction. "Intelligence" in abstraction is introduced by this way.
- introduce special guidance to optimize the treatment of
  - dead rules
  - type conversions
  - functions that become total
Current JaKarTa Focus

- Input: Defensive Virtual Machine
- Output:
  - Offensive and Abstract Virtual Machines
  - Diagrams commute
  - Offensive and Defensive machines coincide on well-typed programs

Automating the correctness proof of the BCV is yet out of reach
Current JTK Focus: Offensive Abstraction

data valu_prim =
    VReturnAddress nat |
    VBoolean z    |
    VByte z      |
    VShort z     |
    VInt z.

becomes

type abs_valu_prim = z.
function abstract_valu_prim
   : valu_prim -> abs_valu_prim :=
=>abstract_valu_prim (VReturnAddress v) -> (inject_nat v) ;
=>abstract_valu_prim (VBoolean v) -> v ;
=>abstract_valu_prim (VByte v) -> v ;
=>abstract_valu_prim (VShort v) -> v ;
=>abstract_valu_prim (VInt v) -> v .
Current JTK Focus: Offensive Abstraction

\[
\text{<pUTSTATIC\_rule\_6>}
\]

\[
(\text{stack\_f state}) \rightarrow (\text{Cons h lf}),
\]

\[
(\text{head (opstack h)}) \rightarrow (\text{Value x}),
\]

\[
(\text{nth\_elt (sheap\_f state) idx}) \rightarrow (\text{Value nod}),
\]

\[
\text{nod} \rightarrow (\text{VPrim (VBoolean z0)}),
\]

\[
\text{t} \rightarrow (\text{Prim Byte})
\]

\[
\Rightarrow (\text{pUTSTATIC t idx state cap}) \rightarrow
\]

\[
(\text{res\_putstatic state x idx});
\]
Current Focus: Offensive Abstraction

\[
<\text{abstracted\_pUTSTATIC\_rule\_6}>
(\text{abstracted\_stack\_f\ state})\rightarrow(\text{Cons\ h\ lf}),
(\text{head\ (abstracted\_opstack\ h)})\rightarrow(\text{Value\ x}),
(\text{nth\_elt\ (abstracted\_sheap\_f\ state)})
\quad \text{idx})\rightarrow(\text{Value\ nod}),
\text{nod}\rightarrow z0,
\text{t}\rightarrow(\text{Prim\ Byte})
\quad \Rightarrow (\text{abstracted\_pUTSTATIC\ t\ idx\ state\ cap})\rightarrow
\quad (\text{abstracted\_res\_putstatic\ state\ x\ idx});
\]
Current Focus: Offensive Abstraction

\[<\text{cONV\_rule\_2}>\]
\[(\text{stack\_f state}) \rightarrow (\text{Cons h l f}),\]
\[(\text{extr\_from\_opstack t (head (opstack h))) \rightarrow (\text{Value k})\]
\[\Rightarrow (\text{cONV t t' state}) \rightarrow\]
\[(\text{update\_frame (update\_opstack}}\]
\[(\text{Cons (VPrim (tpz2vp t'}\]
\[(\text{t\_convert t t' k}))\]
\[(\text{opstack h))) h) state);\]
Current Focus: Offensive Abstraction

<abstracted_cONV_rule_2>
(abstracted_stack_f state)->(Cons h 1f),
(head (abstracted_opstack h))->(Value k)
=> (abstracted_cONV t t' state)->
   (abstracted_update_frame
     (abstracted_update_opstack
       (Cons (abstracted_tpz2vp t'
          (t_convert t t' k))
       (abstracted_opstack h))) h) state)
Current Focus: Offensive Abstraction

Script $\approx 40$ lines $\Rightarrow$ whole offensive virtual machine

abstract exec_instruction with
  abstract_valu_prim (etc...)
and inject_nat (etc...)
(* user intervention directives start here *)
conversion using inject_nat z2n
(etc...)
in cONV replace 2,2,1 by
  (head (abstracted_opstack h))
(etc...)
reject (abstracted_abortCode Type_error state)
into jcvm_off_functions log jcvm_log.
Conclusion

- right now JaKarTa is proof of concept
- Tool independence (translations to theorem provers, rewrite systems etc...)
- Generated offensive virtual machine, abstract machine underway
- Used JAK tactics to good effect
- Automation of equational reasoning is on the way