A bytecode logic for JML and types (Isabelle/HOL sources)

Lennart Beringer and Martin Hofmann

May 26, 2024

Abstract

This document contains the Isabelle/HOL sources underlying our paper *A bytecode logic for JML and types* [2], updated to Isabelle 2008. We present a program logic for a subset of sequential Java bytecode that is suitable for representing both, features found in high-level specification language JML as well as interpretations of high-level type systems. To this end, we introduce a fine-grained collection of assertions, including strong invariants, local annotations and VDM-reminiscent partial-correctness specifications. Thanks to a goal-oriented structure and interpretation of judgements, verification may proceed without recourse to an additional control flow analysis. The suitability for interpreting intensional type systems is illustrated by the proof-carrying-code style encoding of a type system for a first-order functional language which guarantees a constant upper bound on the number of objects allocated throughout an execution, be the execution terminating or non-terminating.

Like the published paper, the formal development is restricted to a comparatively small subset of the JVML, lacking (among other features) exceptions, arrays, virtual methods, and static fields. This shortcoming has been overcome meanwhile, as our paper has formed the basis of the Mobius base logic [9], a program logic for the full sequential fragment of the JVML. Indeed, the present formalisation formed the basis of a subsequent formalisation of the Mobius base logic in the proof assistant Coq, which includes a proof of soundness with respect to the Bicolano operational semantics [10].

Contents

1 Preliminaries: association lists

Finite maps are used frequently, both in the representation of syntax and in the program logics. Instead of restricting Isabelle's partial map type $\alpha \rightharpoonup \beta$ to finite domains, we found it easier for the present development to use the following adhoc data type of association lists.

type-synonym $('a,'b)$ *AssList* $= ('a \times 'b)$ *list*

primrec *lookup*::$('a, 'b)$ *AssList* $\Rightarrow$ $'a \Rightarrow ('b$ *option*$)$ (-↓- [*90*,*0*] *90*)
where
lookup [] l = *None* |
lookup $(h \# t)$ l = (*if fst* $h = l$ *then Some*(*snd* h) *else lookup* t l)

The statement following the type declaration of *lookup* indicates that we may use the infix notation $L{\downarrow}a$ for the lookup operation, and asserts some precedence for bracketing. In a similar way, shorthands are introduced for various operations throughout this document.

primrec *delete*::$('a, 'b)$ *AssList* $\Rightarrow$ $'a \Rightarrow ('a, 'b)$ *AssList*
where
delete [] a = [] |
delete $(h \# t)$ a = (*if* $(fst\ h) = a$ *then delete* t a *else* $(h \# (delete\ t\ a))$)

definition *upd*::$('a, 'b)$ *AssList* $\Rightarrow$ $'a \Rightarrow$ $'b \Rightarrow ('a, 'b)$ *AssList*

(-[-↦-] [1000,0,0] 1000)
where *upd L a b* = (*a*,*b*) # (*delete L a*)

The empty map is represented by the empty list.

definition *emp*::('*a*, '*b*)*AssList*
where *emp* = []
definition *contained*::('*a*,'*b*) *AssList* ⇒ ('*a*,'*b*) *AssList* ⇒ *bool*
where *contained L M* = (∀ *a b* . *L*↓*a* = *Some b* ⟶ *M*↓*a* = *Some b*)

The following operation defined the cardinality of a map.

fun *AL-Size* :: ('*a*, '*b*) *AssList* ⇒ *nat* (|-| [1000] 1000) **where**
AL-Size [] = *0*
| *AL-Size* (*h* # *t*) = *Suc* (*AL-Size* (*delete t* (*fst h*)))

Some obvious basic properties of association lists and their operations are easily proven, but have been suppressed during the document preparation.

2 Language

2.1 Syntax

We have syntactic classes of (local) variables, class names, field names, and method names. Naming restrictions, namespaces, long Java names etc. are not modelled.

typedecl *Var*
typedecl *Class*
typedecl *Field*
typedecl *Method*

Since arithmetic operations are modelled as unimplemented functions, we introduce the type of values in this section. The domain of heap locations is arbitrary.

typedecl *Addr*

A reference is either null or an address.

datatype *Ref* = *Nullref* | *Loc Addr*

Values are either integer numbers or references.

datatype *Val* = *RVal Ref* | *IVal int*

The type of (instruction) labels is fixed, since the operational semantics increments the program counter after each instruction.

type-synonym *Label* = *int*

Regarding the instructions, we support basic operand-stack manipulations, object creation, field modifications, casts, static method invocations, conditional and unconditional jumps, and a return instruction.

For every (Isabelle) function $f : Val \Rightarrow Val \Rightarrow Val$ we have an instruction *binop f* whose semantics is to invoke f on the two topmost values on the operand stack and replace them with the result. Similarly for *unop f*.

datatype *Instr* =
  const Val
| *dup*
| *pop*
| *swap*
| *load Var*
| *store Var*
| *binop Val* ⇒ *Val* ⇒ *Val*
| *unop Val* ⇒ *Val*
| *new Class*
| *getfield Class Field*
| *putfield Class Field*
| *checkcast Class*
| *invokeS Class Method*
| *goto Label*
| *iftrue Label*
| *vreturn*

Method body declarations contain a list of formal parameters, a mapping from instruction labels to instructions, and a start label. The operational semantics assumes that instructions are labelled consecutively[1].

type-synonym *Mbody* = *Var list* × (*Label*, *Instr*) *AssList* × *Label*

A class definition associates method bodies to method names.

type-synonym *Classdef* = (*Method*, *Mbody*) *AssList*

Finally, a program consists of classes.

type-synonym *Prog* = (*Class*, *Classdef*) *AssList*

Taken together, the three types *Prog*, *Classdef*, and *Mbody* represent an abstract model of the virtual machine environment. In our opinion, it would be desirable to avoid modelling this environment at a finer level, at least for the purpose of the program logic. For example, we prefer not to consider in detail the representation of the constant pool.

[1] In the paper, we slightly abstract from this by including a successor functions on labels

2.2 Dynamic semantics

2.2.1 Semantic components

An object consists of the identifier of its dynamic class and a map from field names to values. Currently, we do not model type-correctness, nor do we require that all (or indeed any) of the fields stem from the static definition of the class, or a super-class. Note, however, that type correctness can be expressed in the logic.

type-synonym *Object* = *Class* × (*Field*, *Val*) *AssList*

The heap is represented as a map from addresses to values. The JVM specification does not prescribe any particular object layout. The proposed type reflects this indeterminacy, but allows one to calculate the byte-correct size of a heap only after a layout scheme has been supplied. Alternative heap models would be the store-less semantics in the sense of Jonkers [8] and Deutsch [5], (where the heap is modelled as a partial equivalence relation on access paths), or object-based semantics in the sense of Reddy [11], where the heap is represented as a history of update operations. Hähnle et al. use a variant of the latter in their dynamic logic for a JavaCard [6].

type-synonym *Heap* = (*Addr*, *Object*) *AssList*

Later, one might extend heaps by a component for static fields.

The types of the (register) store and the operand stack are as expected.

type-synonym *Store* = (*Var*, *Val*) *AssList*
type-synonym *OpStack* = *Val list*

States contain an operand stack, a store, and a heap.

type-synonym *State* = *OpStack* × *Store* × *Heap*

definition *heap*::*State* ⇒ *Heap*
where *heap s* = *snd*(*snd s*)

The operational semantics and the program logic are defined relative to a fixed program *P*. Alternatively, the type of the operational semantics (and proof judgements) could be extended by a program component. We also define the constant value *TRUE*, the representation of which does not matter for the current formalisation.

axiomatization *P*::*Prog* **and** *TRUE*::*Val*

In order to obtain more readable rules, we define operations for extracting method bodies and instructions from the program.

definition *mbody-is*::*Class* ⇒ *Method* ⇒ *Mbody* ⇒ *bool*
where *mbody-is C m M* = (∃ *CD* . *P*↓*C* = *Some CD* ∧ *CD*↓*m* = *Some M*)

definition *get-ins*::*Mbody* ⇒ *Label* ⇒ *Instr option*
where *get-ins M l* = (*fst*(*snd M*))↓*l*

definition *ins-is*::*Class* ⇒ *Method* ⇒ *Label* ⇒ *Instr* ⇒ *bool*
where *ins-is C m l ins* = (∃ *M* . *mbody-is C m M* ∧ *get-ins M l* = *Some ins*)

The transfer of method arguments from the caller's operand stack to the formal parameters of an invoked method is modelled by the predicate

inductive-set *Frame*::(*OpStack* × (*Var list*) × *Store* × *OpStack*) *set*
where
FrameNil: ⟦*oo*=*ops*⟧ ⟹ (*ops*,[],*emp*,*oo*) : *Frame*
|
Frame-cons: ⟦(*oo*,*par*,*S*,*ops*) : *Frame*; *R* =*S*[*x*↦*v*]⟧
⟹ (*v* # *oo*, *x* # *par*,*R*,*ops*):*Frame*

In order to obtain a deterministic semantics, we assume the existence of a function, with the obvious freshness axiom for this construction.

axiomatization *nextLoc*::*Heap* ⇒ *Addr*
where *nextLoc-fresh*: *h*↓(*nextLoc h*) = *None*

2.2.2 Operational judgements

Similar to Bannwart-Müller [1], we define two operational judgements: a one-step relation and a relation that represents the transitive closure of the former until the end of the current method invocation. These relations are mutually recursive, since the method invocation rule contracts the execution of the invoked method to a single step. The one-step relation associates a state to its immediate successor state, where the program counter is interpreted with respect to the current method body. The transitive closure ignores the bottom part of the operand stack and the store of the final configuration. It simply returns the heap and the result of the method invocation, where the latter is given by the topmost value on the operand stack. In contrast to [1], we do not use an explicit *return* variable. Both relations take an additional index of type *nat* that monitors the derivation height. This is useful in the proof of soundness of the program logic.

Intuitively, (*M*,*l*,*s*,*n*,*l*′,*s*′):*Step* means that method (body) *M* evolves in one step from state *s* to state *s*′, while statement (*M*,*s*,*n*,*h*,*v*):*Exec* indicates that executing from *s* in method *M* leads eventually to a state whose final value is *h*, where precisely the last step in this sequence is a *vreturn* instruction and the return value is *v*.

Like Bannwart and Müller, we define a "frame-less" semantics. i.e. the execution of a method body is modelled by a transitive closure of the basic step-relation, which results in a one-step reduction at the invocation site.

Arguably, an operational semantics with an explicit frame stack is closer to the real JVM. It should not be difficult to verify the operational soundness of the present system w.r.t. such a finer model, or to modify the semantics.

inductive-set
  Step::(*Mbody* × *Label* × *State* × *nat* × *Label* × *State*) *set*
and
  Exec::(*Mbody* × *Label* × *State* × *nat* × *Heap* × *Val*) *set*
where
Const:⟦*get-ins M l* = *Some* (*const v*); *NEXT* = (*v* # *os*,*s*,*h*); *ll*=*l*+*1*⟧
  ⟹ (*M*,*l*,(*os*,*s*,*h*), *1*, *ll*, *NEXT*) : *Step*
|
Dup: ⟦*get-ins M l* = *Some dup*; *NEXT* = (*v* # *v* # *os*,*s*,*h*); *ll* =*l*+*1*⟧
  ⟹ (*M*,*l*,(*v* # *os*,*s*,*h*), *1*, *ll*, *NEXT*) : *Step*
|
Pop: ⟦*get-ins M l* = *Some pop*; *NEXT* = (*os*,*s*,*h*); *ll*=*l*+*1*⟧
  ⟹ (*M*,*l*,(*v* # *os*,*s*,*h*), *1*, *ll*, *NEXT*) : *Step*
|
Swap: ⟦*get-ins M l* = *Some swap*; *NEXT* = (*w* # (*v* # *os*),*s*,*h*); *ll*= *l*+*1*⟧
  ⟹ (*M*,*l*,(*v* # (*w* # *os*),*s*,*h*), *1*, *ll*, *NEXT*) : *Step*
|
Load: ⟦*get-ins M l* = *Some* (*load x*); *s*↓*x* = *Some v*;
    NEXT = (*v* # *os*,*s*,*h*); *ll*=*l*+*1*⟧
  ⟹ (*M*,*l*,(*os*,*s*,*h*), *1*, *ll*,*NEXT*) : *Step*
|
Store:⟦*get-ins M l* = *Some* (*store x*); *NEXT* = (*os*,*s*[*x*↦*v*],*h*); *ll*= *l*+*1*⟧
  ⟹ (*M*,*l*,(*v* # *os*,*s*,*h*), *1*, *ll*, *NEXT*) : *Step*
|
Binop:⟦*get-ins M l* = *Some* (*binop f*); *NEXT* = ((*f v w*) # *os*,*s*,*h*); *ll*=*l*+*1*⟧
  ⟹ (*M*,*l*,(*v* # (*w* # *os*),*s*,*h*), *1*, *ll*,*NEXT*) : *Step*
|
Unop: ⟦*get-ins M l* = *Some* (*unop f*); *NEXT* = ((*f v*) # *os*,*s*,*h*);*ll*=*l*+*1*⟧
  ⟹ (*M*,*l*,(*v* # *os*,*s*,*h*), *1*, *ll*, *NEXT*) : *Step*
|
New: ⟦*get-ins M l* = *Some* (*new d*); *newobj* = (*d*, *emp*); *a*=*nextLoc h*;
    NEXT = ((*RVal* (*Loc a*)) # *os*,*s*,*h*[*a*↦*newobj*]); *ll* = *l*+*1*⟧
  ⟹ (*M*,*l*,(*os*,*s*,*h*), *1*, *ll*,*NEXT*) : *Step*
|
Get: ⟦*get-ins M l* = *Some* (*getfield d F*); *h*↓*a* = *Some* (*d*, *Flds*);
    Flds↓*F* = *Some v*; *NEXT* = (*v* # *os*,*s*,*h*); *ll*=*l*+*1*⟧
  ⟹ (*M*,*l*,((*RVal* (*Loc a*)) # *os*,*s*,*h*), *1*, *ll*,*NEXT*) : *Step*
|
Put: ⟦*get-ins M l* = *Some* (*putfield d F*); *h*↓*a* = *Some* (*d*, *Flds*);
    newobj = (*d*, *Flds*[*F*↦*v*]); *NEXT* = (*os*,*s*,*h*[*a*↦*newobj*]); *ll*=*l*+*1*⟧
  ⟹ (*M*,*l*,(*v* # ((*RVal* (*Loc a*)) # *os*),*s*,*h*), *1*, *ll*, *NEXT*) : *Step*
|
Cast: ⟦*get-ins M l* = *Some* (*checkcast d*); *h*↓*a* = *Some* (*d*, *Flds*);
    NEXT = ((*RVal* (*Loc a*)) # *os*,*s*,*h*); *ll*=*l*+*1*⟧
  ⟹ (*M*,*l*,((*RVal* (*Loc a*)) # *os*,*s*,*h*), *1*, *ll*,*NEXT*) : *Step*
|

Goto: $[\![$*get-ins M l = Some (goto pc)*$]\!] \Longrightarrow$ (*M*,*l*,*S*, *1*, *pc*,*S*) : *Step*
|
IfT: $[\![$*get-ins M l = Some (iftrue pc)*; *NEXT* = (*os*,*s*,*h*)$]\!]$
$\Longrightarrow$ (*M*,*l*,(*TRUE* # *os*,*s*,*h*), *1*, *pc*, *NEXT*) : *Step*
|
IfF: $[\![$*get-ins M l = Some (iftrue pc)*; *v* $\neq$ *TRUE*; *NEXT* = (*os*,*s*,*h*); *ll*=*l*+*1*$]\!]$
$\Longrightarrow$ (*M*,*l*,(*v* # *os*,*s*,*h*), *1*, *ll*, *NEXT*) : *Step*
|
InvS: $[\![$*get-ins M l = Some (invokeS C m)*; *mbody-is C m* (*par*,*code*,*l0*);
((*par*,*code*,*l0*),*l0*,([], *S*,*h*), *n*, *hh*, *v*): *Exec*;
(*ops*,*par*,*S*,*os*) : *Frame*; *NEXT* = (*v* # *os*,*s*,*hh*); *ll* = *l*+*1*$]\!]$
$\Longrightarrow$ (*M*,*l*,(*ops*,*s*,*h*), *Suc n*, *ll*, *NEXT*) : *Step*
|
Vret: $[\![$*get-ins M l = Some vreturn*$]\!] \Longrightarrow$ (*M*,*l*,(*v* # *os*,*s*,*h*), *1*, *h*, *v*) : *Exec*
|
Run: $[\![$(*M*,*l*,*s*,*n*,*ll*,*t*):*Step*; (*M*,*ll*,*t*,*m*,*h*,*v*):*Exec*; *k* = (*max n m*) +*1* $]\!]$
$\Longrightarrow$ (*M*,*l*,*s*,*k*,*h*,*v*) : *Exec*

A big-step operational judgement that abstracts from the derivation height is easily defined.

definition *Opsem*::*Mbody* $\Rightarrow$ *Label* $\Rightarrow$ *State* $\Rightarrow$ *Heap* $\Rightarrow$ *Val* $\Rightarrow$ *bool*
where *Opsem M l s h v* = ($\exists$ *n* . (*M*,*l*,*s*,*n*,*h*,*v*):*Exec*)

2.3 Basic properties

We provide elimination lemmas for the inductively defined relations

inductive-cases *eval-cases*:
(*M*,*l*,*s*,*n*,*ll*,*t*) : *Step*
(*M*,*l*,*s*,*n*,*h*,*v*) : *Exec*

and observe that no derivations of height 0 exist.

lemma *no-zero-height-Step-derivs*: (*M*,*l*,*s*,*0*,*ll*,*t*):*Step* $\Longrightarrow$ *False*
lemma *no-zero-height-Exec-derivs*: (*M*,*l*,*s*,*0*,*h*,*v*):*Exec* $\Longrightarrow$ *False*

By induction on the derivation system one can show determinism.

lemma *Step-determ*:
$[\![$(*M*,*l*,*s*,*n*,*l1*,*t*) $\in$ *Step*; (*M*,*l*,*s*,*m*,*l2*,*r*):*Step*$]\!] \Longrightarrow$ *n*=*m* $\wedge$ *t*=*r* $\wedge$ *l1*=*l2*
lemma *Exec-determ*:
$[\![$(*M*,*l*,*s*,*n*,*h*,*v*) $\in$ *Exec*; (*M*,*l*,*s*,*m*,*k*,*w*):*Exec*$]\!] \Longrightarrow$ *n*=*m* $\wedge$ *h*=*k* $\wedge$ *v*=*w*

3 Axiomatic semantics

3.1 Assertion forms

We introduce two further kinds of states. Initial states do not contain operand stacks, terminal states lack operand stacks and local variables, but include return values.

type-synonym *InitState* = *Store* × *Heap*
type-synonym *TermState* = *Heap* × *Val*

A judgements relating to a specific program point $C.m.l$ consists of pre- and post-conditions, an invariant, and – optionally – a local annotation. Local pre-conditions and annotations relate initial states to states, i.e. are of type

type-synonym *Assn* = *InitState* ⇒ *State* ⇒ *bool*

Post-conditions additionally depend on a terminal state

type-synonym *Post* = *InitState* ⇒ *State* ⇒ *TermState* ⇒ *bool*

Invariants hold for the heap components of all future (reachable) states in the current frame as well as its subframes. They relate these heaps to the current state and the initial state of the current frame.

type-synonym *Inv* = *InitState* ⇒ *State* ⇒ *Heap* ⇒ *bool*

Local annotations of a method implementation are collected in a table of type

type-synonym *ANNO* = (*Label*, *Assn*) *AssList*

Implicitly, the labels are always interpreted with respect to the current method. In addition to such a table, the behaviour of methods is specified by a partial-correctness assertion of type

type-synonym *MethSpec* = *InitState* ⇒ *TermState* ⇒ *bool*

and a method invariant of type

type-synonym *MethInv* = *InitState* ⇒ *Heap* ⇒ *bool*

A method invariant is expected to be satisfied by the heap components of all states during the execution of the method, including states in subframes, irrespectively of the termination behaviour.

All method specifications are collected in a table of type

type-synonym *MSPEC* = (*Class* × *Method*, *MethSpec* × *MethInv* × *ANNO*) *AssList*

A table of this type assigns to each method a partial-correctness specification, a method invariant, and a table of local annotations for the instructions in this method.

3.2 Proof system

The proof system derives judgements of the form $G \triangleright \{A\} C, m, l \{B\}\ I$ where A is a pre-condition, B is a post-condition, I is a strong invariant, $C.m.l$ represents a program point, and G is a proof context (see below). The proof

system consists of syntax-directed rules and structural rules. These rules are formulated in such a way that assertions in the conclusions are unconstrained, i.e. a rule can be directly applied to derive a judgement. In the case of the syntax-directed rules, the hypotheses require the derivation of related statements for the control flow successor instructions. Judgements occurring as hypotheses involve assertions that are notationally constrained, and relate to the conclusions' assertions via uniform constructions that resemble strongest postconditions in Hoare-style logics.

On pre-conditions, the operator

definition *SP-pre*::*Mbody* ⇒ *Label* ⇒ *Assn* ⇒ *Assn*
where *SP-pre M l A* = (λ *s0 r* . (∃ *s l1 n*. *A s0 s* ∧ (*M*,*l*,*s*,*n*,*l1*,*r*):*Step*))

constructs an assertion that holds of a state r precisely if the argument assertion A held at the predecessor state of r. Similar readings explain the constructions on post-conditions

definition *SP-post*::*Mbody* ⇒ *Label* ⇒ *Post* ⇒ *Post*
where *SP-post M l B* = (λ *s0 r t* . (∀ *s l1 n*. (*M*,*l*,*s*,*n*,*l1*,*r*):*Step* ⟶ *B s0 s t*))

and invariants

definition *SP-inv*::*Mbody* ⇒ *Label* ⇒ *Inv* ⇒ *Inv*
where *SP-inv M l I* = (λ *s0 r h* . ∀ *s l1 n*. (*M*,*l*,*s*,*n*,*l1*,*r*):*Step* ⟶ *I s0 s h*)

For the basic instructions, the appearance of the single-step execution relation in these constructions makes the strongest-postcondition interpretation apparent, but could easily be eliminated by unfolding the definition of *Step*. In the proof rule for static method invocations, such a direct reference to the operational semantics is clearly undesirable. Instead, the proof rule extracts the invoked method's specification from the specification table. In order to simplify the formulation of the proof rule, we introduce three operators which manipulate the extracted assertions in a similar way as the above *SP*-operators.

definition *SINV-pre*::*Var list* ⇒ *MethSpec* ⇒ *Assn* ⇒ *Assn* **where**
SINV-pre par T A =
(λ *s0 s* . (∃ *ops1 ops2 S R h k w*.
(*ops1*,*par*,*R*,*ops2*) : *Frame* ∧ *T* (*R*, *k*) (*h*,*w*) ∧
A s0 (*ops1*,*S*,*k*) ∧ *s* = (*w*#*ops2*,*S*, *h*)))

definition *SINV-post*::*Var list* ⇒ *MethSpec* ⇒ *Post* ⇒ *Post* **where**
SINV-post par T B =
(λ *s0 s t* . ∀ *ops1 ops2 S R h k w r*.
(*ops1*,*par*,*R*,*ops2*) : *Frame* ⟶ *T* (*R*, *k*) (*h*,*w*) ⟶
s=(*w*#*ops2*,*S*,*h*) ⟶ *r*=(*ops1*,*S*,*k*) ⟶ *B s0 r t*)

definition *SINV-inv*::*Var list* ⇒ *MethSpec* ⇒ *Inv* ⇒ *Inv* **where**
SINV-inv par T I =

$(\lambda\ s0\ s\ h'\ .\ \forall\ ops1\ ops2\ S\ R\ h\ k\ w.$
$(ops1,par,R,ops2) : Frame \longrightarrow T\ (R,k)\ (h,w) \longrightarrow$
$s=(w\#ops2,S,h) \longrightarrow I\ s0\ (ops1,S,k)\ h')$

The derivation system is formulated using contexts G of proof-theoretic assumptions representing local judgements. The type of contexts is

type-synonym $CTXT = (Class \times Method \times Label, Assn \times Post \times Inv)\ AssList$

The existence of the proof context also motivates that the hypotheses in the syntax-directed rules are formulated using an auxiliary judgement form, $G \rhd \langle A\rangle C, m, l\langle B\rangle\ I$. Statements for this auxiliary form can be derived by essentially two rules. The first rule, AX, allows us to extract assumptions from the context, while the second rule, $INJECT$, converts an ordinary judgement $G \rhd \{A\}C, m, l\{B\}\ I$ into $G \rhd \langle A\rangle C, m, l\langle B\rangle\ I$. No rule is provided for a direct embedding in the opposite direction. As a consequence of this formulation, contextual assumptions cannot be used directly to justify a statement $G \rhd \{A\}C, m, l\{B\}\ I$. Instead, the extraction of such an assumption has to be followed by at least one "proper" (syntax-driven) rule. In particular, attempts to verify jumps by assuming a judgement and immediately using it to prove the rule's hypothesis are ruled out. More specifically, the purpose of this technical device will become obvious when we discharge the context in the proof of soundness (Section 5.3). Here, each entry $((C', m', l'), (A', B', I'))$ of the context will need to be justified by a derivation $G \rhd \{A'\}C', m', l'\{B'\}\ I'$. This justification should in principle be allowed to rely on other entries of the context. However, an axiom rule for judgements $G \rhd \{A\}C, m, l\{B\}\ I$ would allow a trivial discharge of any context entry. The introduction of the auxiliary judgement form $G \rhd \langle A\rangle C, m, l\langle B\rangle\ I$ thus ensures that the discharge of a contextual assumption involves at least one application of a "proper" (i.e. syntax-directed) rule. Rule $INJECT$ is used to chain together syntax-directed rules. Indeed, it allows us to discharge a hypothesis $G \rhd \langle A\rangle C, m, l\langle B\rangle\ I$ of a syntax-directed rule using a derivation of $G \rhd \{A\}C, m, l\{B\}\ I$.

The proof rules are defined relative to a fixed method specification table MST.

axiomatization MST::$MSPEC$

In Isabelle, the distinction between the two judgement forms may for example be achieved by introducing a single judgement form that includes a boolean flag, and definiing two pretty-printing notions.

inductive-set *SP-Judgement* ::
$(bool \times CTXT \times Class \times Method \times Label \times Assn \times Post \times Inv)\ set$
and
SP-Deriv :: $CTXT$ => $Assn$ => $Class$ => $Method$ => $Label$ =>
$Post$ => Inv => $bool$

$(\text{-} \rhd \{\!|\ \text{-}\ |\!\} \text{-},\text{-},\text{-} \{\!|\ \text{-}\ |\!\} \text{-} [100,100,100,100,100,100,100]\ 50)$

and

SP-Assum :: *CTXT* => *Assn* => *Class* => *Method* => *Label* =>
Post => *Inv* => *bool*
$(\text{-} \rhd \langle\ \text{-}\ \rangle \text{-},\text{-},\text{-} \langle\ \text{-}\ \rangle \text{-} [100,100,100,100,100,100,100]\ 50)$

where

$G \rhd \{\!|\ A\ |\!\}\ C,m,l \{\!|\ B\ |\!\}\ I == (False,\ G,\ C,\ m,\ l,\ A,\ B,\ I)$:*SP-Judgement*

|

$G \rhd \langle\ A\ \rangle\ C,m,l \langle\ B\ \rangle\ I == (True,\ G,\ C,\ m,\ l,\ A,\ B,\ I)$:*SP-Judgement*

|

INSTR:

$[\![\ mbody\text{-}is\ C\ m\ M;\ get\text{-}ins\ M\ l = Some\ ins;$
$lookup\ MST\ (C,m) = Some\ (Mspec,Minv,Anno);$
$\forall\ Q\ .\ lookup\ Anno\ l = Some\ Q \longrightarrow (\forall\ s0\ s\ .\ A\ s0\ s \longrightarrow Q\ s0\ s);$
$\forall\ s0\ s\ .\ A\ s0\ s \longrightarrow I\ s0\ s\ (heap\ s);$
$ins \in \{\ const\ c,\ dup,\ pop,\ swap,\ load\ x,\ store\ x,\ binop\ f,\ unop\ g,$
$new\ d,\ getfield\ d\ F,\ putfield\ d\ F,\ checkcast\ d\};$
$G \rhd \langle\ (SP\text{-}pre\ M\ l\ A)\ \rangle\ C,m,(l+1)\ \langle\ (SP\text{-}post\ M\ l\ B)\ \rangle\ (SP\text{-}inv\ M\ l\ I)\ ]\!]$
$\Longrightarrow G \rhd \{\!|\ A\ |\!\}\ C,m,l \{\!|\ B\ |\!\}\ I$

|

GOTO:

$[\![\ mbody\text{-}is\ C\ m\ M;\ get\text{-}ins\ M\ l = Some\ (goto\ pc);$
$MST{\downarrow}(C,m) = Some\ (Mspec,Minv,Anno);$
$\forall\ Q\ .\ Anno{\downarrow}(l) = Some\ Q \longrightarrow (\forall\ s0\ s\ .\ A\ s0\ s \longrightarrow Q\ s0\ s);$
$\forall\ s0\ s\ .\ A\ s0\ s \longrightarrow I\ s0\ s\ (heap\ s);$
$G \rhd \langle SP\text{-}pre\ M\ l\ A\rangle\ C,m,pc\ \langle SP\text{-}post\ M\ l\ B\rangle\ (SP\text{-}inv\ M\ l\ I)]\!]$
$\Longrightarrow G \rhd \{\!|\ A\ |\!\}\ C,m,l \{\!|\ B\ |\!\}\ I$

|

IF:

$[\![\ mbody\text{-}is\ C\ m\ M;\ get\text{-}ins\ M\ l = Some\ (iftrue\ pc);$
$MST{\downarrow}(C,m) = Some\ (Mspec,Minv,Anno);$
$\forall\ Q\ .\ Anno{\downarrow}(l) = Some\ Q \longrightarrow (\forall\ s0\ s\ .\ A\ s0\ s \longrightarrow Q\ s0\ s);$
$\forall\ s0\ s\ .\ A\ s0\ s \longrightarrow I\ s0\ s\ (heap\ s);$

$G \rhd \langle\ SP\text{-}pre\ M\ l\ (\lambda\ s0\ s\ .\ (\forall\ ops\ S\ k\ .\ s{=}(TRUE\#ops,S,k) \longrightarrow A\ s0\ s))\rangle$
C,m,pc
$\langle\ SP\text{-}post\ M\ l\ (\lambda\ s0\ s\ t\ .$
$(\forall\ ops\ S\ k\ .\ s{=}(TRUE\#ops,S,k) \longrightarrow B\ s0\ s\ t))\rangle$
$(\ SP\text{-}inv\ M\ l\ (\lambda\ s0\ s\ t\ .$
$(\forall\ ops\ S\ k\ .\ s{=}(TRUE\#ops,S,k) \longrightarrow I\ s0\ s\ t))\ );$

$G \rhd \langle\ SP\text{-}pre\ M\ l\ (\lambda\ s0\ s\ .$
$(\forall\ ops\ S\ k\ v\ .\ s = (v\#ops,S,k) \longrightarrow v \neq TRUE \longrightarrow A\ s0\ s))\rangle$
$C,m,(l+1)$
$\langle\ SP\text{-}post\ M\ l\ (\lambda\ s0\ s\ t.$
$(\forall\ ops\ S\ k\ v\ .\ s = (v\#ops,S,k) \longrightarrow v \neq TRUE \longrightarrow B\ s0\ s\ t))\rangle$
$(\ SP\text{-}inv\ M\ l\ (\lambda\ s0\ s\ t.$
$(\forall\ ops\ S\ k\ v\ .\ s = (v\#ops,S,k) \longrightarrow v \neq TRUE \longrightarrow I\ s0\ s\ t))\ )]\!]$
$\Longrightarrow G \rhd \{\!|\ A\ |\!\}\ C,m,l \{\!|\ B\ |\!\}\ I$

|
VRET:
⟦ *mbody-is C m M*; *get-ins M l* = *Some vreturn*;
MST↓(*C*,*m*) = *Some* (*Mspec*,*Minv*,*Anno*);
∀ *Q* . *Anno*↓(*l*) = *Some Q* ⟶ (∀ *s0 s* . *A s0 s* ⟶ *Q s0 s*);
∀ *s0 s* . *A s0 s* ⟶ *I s0 s* (*heap s*);
∀ *s0 s* . *A s0 s* ⟶ (∀ *v ops S h* . *s* = (*v*#*ops*,*S*,*h*) ⟶ *B s0 s* (*h*,*v*))⟧
⟹ *G* ▷ {| *A* |} *C*,*m*,*l* {| *B* |} *I*
|
INVS:
⟦ *mbody-is C m M*; *get-ins M l* = *Some* (*invokeS D m*′);
MST↓(*C*,*m*) = *Some* (*Mspec*,*Minv*,*Anno*);
MST↓(*D*,*m*′) = *Some* (*T*,*MI*,*Anno2*); *mbody-is D m*′ (*par*,*code*,*l0*);
∀ *Q* . *Anno*↓(*l*) = *Some Q* ⟶ (∀ *s0 s* . *A s0 s* ⟶ *Q s0 s*);
∀ *s0 s* . *A s0 s* ⟶ *I s0 s* (*heap s*);
∀ *s0 ops1 ops2 S R k t* . (*ops1*,*par*,*R*,*ops2*) : *Frame* ⟶
A s0 (*ops1*,*S*,*k*) ⟶ *MI* (*R*,*k*) *t* ⟶ *I s0* (*ops1*,*S*,*k*) *t*;
G ▷ ⟨*SINV-pre par T A*⟩ *C*,*m*,(*l*+1) ⟨*SINV-post par T B*⟩
(*SINV-inv par T I*)⟧
⟹ *G* ▷ {| *A* |} *C*,*m*,*l* {| *B* |} *I*
|
CONSEQ:
⟦ (*b*,*G*,*C*,*m*,*l*,*AA*,*BB*,*II*) ∈ *SP-Judgement*;
∀ *s0 s* . *A s0 s* ⟶ *AA s0 s*; ∀ *s0 s t* . *BB s0 s t* ⟶ *B s0 s t*;
∀ *s0 s k*. *II s0 s k* ⟶ *I s0 s k*⟧
⟹ (*b*,*G*,*C*,*m*,*l*,*A*,*B*,*I*) ∈ *SP-Judgement*
|
INJECT: ⟦ *G* ▷ {| *A* |} *C*,*m*,*l* {| *B* |} *I*⟧ ⟹ *G* ▷ ⟨ *A* ⟩ *C*,*m*,*l* ⟨ *B* ⟩ *I*
|
AX:
⟦ *G*↓(*C*,*m*,*l*) = *Some* (*A*,*B*,*I*); *MST*↓(*C*,*m*) = *Some* (*Mspec*,*Minv*,*Anno*);
∀ *Q* . *Anno*↓(*l*) = *Some Q* ⟶ (∀ *s0 s* . *A s0 s* ⟶ *Q s0 s*);
∀ *s0 s* . *A s0 s* ⟶ *I s0 s* (*heap s*)⟧
⟹ *G* ▷ ⟨ *A* ⟩ *C*,*m*,*l* ⟨ *B* ⟩ *I*

As a first consequence, we can prove by induction on the proof system that a derivable judgement entails its strong invariant and an annotation that may be attached to the instruction.

lemma *AssertionsImplyMethInvariants*:
⟦ *G* ▷ {| *A* |} *C*,*m*,*l* {| *B* |} *I*; *A s0 s*⟧ ⟹ *I s0 s* (*heap s*)
lemma *AssertionsImplyAnnoInvariants*:
⟦ *G* ▷ {| *A* |} *C*,*m*,*l* {| *B* |} *I*; *MST*↓(*C*,*m*) = *Some*(*Mspec*,*Minv*,*Anno*);
Anno↓(*l*) = *Some Q*; *A s0 s*⟧ ⟹ *Q s0 s*

For *verified programs*, all preconditions can be justified by proof derivations, and initial labels of all methods (again provably) satisfy the method preconditions.

definition *mkState*::*InitState* ⇒ *State*
where *mkState s0* = ([],*fst s0*,*snd s0*)

definition *mkPost*::*MethSpec* ⇒ *Post*
where *mkPost T* = (λ *s0 s t* . *s*=*mkState s0* ⟶ *T s0 t*)

definition *mkInv*::*MethInv* ⇒ *Inv*
where *mkInv MI* = (λ *s0 s t* . *s*=*mkState s0* ⟶ *MI s0 t*)

definition *VP-G*::*CTXT* ⇒ *bool* **where**
VP-G G =
((∀ *C m l A B I*. *G*↓(*C*,*m*,*l*) = *Some* (*A*,*B*,*I*) ⟶ *G* ⊳ ⦃*A*⦄ *C*,*m*,*l* ⦃*B*⦄ *I*) ∧
(∀ *C m par code l0 T MI Anno*.
mbody-is C m (*par*,*code*,*l0*) ⟶ *MST*↓(*C*,*m*) = *Some*(*T*,*MI*,*Anno*) ⟶
G ⊳ ⦃(λ *s0 s*. *s* = *mkState s0*)⦄ *C*,*m*,*l0* ⦃*mkPost T*⦄ (*mkInv MI*)))

definition *VP*::*bool* **where** *VP* = (∃ *G* . *VP-G G*)

4 Auxiliary operational judgements

Beside the basic operational judgements *Step* and *Exec*, the interpretation of judgements refers to two multi-step relations which we now define.

4.1 Multistep execution

The first additional operational judgement is the reflexive and transitive closure of *Step*. It relates states s and t if the latter can be reached from the former by a chain if single steps, all in the same frame. Note that t does not need to be a terminal state. As was the case in the definition of *Step*, we first define a relation with an explicit derivation height index (*MStep*).

inductive-set
MStep::(*Mbody* × *Label* × *State* × *nat* × *Label* × *State*) *set*
where
MS-zero: ⟦*k*=*0*; *t*=*s*; *ll*=*l*⟧ ⟹ (*M*,*l*,*s*,*k*,*ll*,*t*):*MStep*
|
MS-step: ⟦(*M*,*l*,*s*,*n*,*l1*,*r*):*Step*; (*M*,*l1*,*r*,*k*,*l2*,*t*):*MStep*; *m*=*Suc k*+*n*⟧
⟹ (*M*,*l*,*s*,*m*,*l2*,*t*) : *MStep*

The following properties of *MStep* are useful to notice.

lemma *ZeroHeightMultiElim*: (*M*,*l*,*s*,*0*,*ll*,*r*) ∈ *MStep* ⟹ *r*=*s* ∧ *ll*=*l*
lemma *MultiSplit*:
⟦(*M*, *l*, *s*, *k*, *ll*, *t*) ∈ *MStep*; *1* ≤ *k*⟧ ⟹
∃ *n m r l1*. (*M*,*l*,*s*,*n*,*l1*,*r*):*Step* ∧ (*M*,*l1*,*r*,*m*,*ll*,*t*):*MStep* ∧ *Suc m* + *n* =*k*
lemma *MStep-returnElim*:
⟦(*M*,*l*,*s*,*k*,*ll*,*t*) ∈ *MStep*; *get-ins M l* = *Some vreturn*⟧ ⟹ *t*=*s* ∧ *ll* = *l*
lemma *MultiApp*:
⟦(*M*,*l*,*s*,*k*,*l1*,*r*):*MStep*; (*M*,*l1*,*r*,*n*,*l2*,*t*):*Step*⟧ ⟹ (*M*,*l*,*s*,*Suc k*+*n*,*l2*,*t*):*MStep*

lemma *MStep-Compose*:
⟦(*M*,*l*,*s*,*n*,*l1*,*r*):*MStep*; (*M*,*l1*,*r*,*k*,*l2*,*t*):*MStep*; *nk*=*n*+*k*⟧
⟹ (*M*,*l*,*s*,*nk*,*l2*,*t*):*MStep*

Here are two simple lemmas relating the operational judgements.

lemma *MStep-Exec1*:
⟦(*M*, *l*, *s*, *kb*, *l1*, *t*) ∈ *MStep*; (*M*, *l*, *s*, *k*, *hh*, *v*) ∈ *Exec*⟧
⟹ ∃ *n*. (*M*, *l1*, *t*, *n*, *hh*, *v*) ∈ *Exec*
lemma *MStep-Exec2*:
⟦ (*M*, *l*, *s*, *kb*, *l1*, *t*) ∈ *MStep*; (*M*, *l1*, *t*, *k*, *hh*, *v*) ∈ *Exec*⟧
⟹ ∃ *n*. (*M*, *l*, *s*, *n*, *hh*, *v*) ∈ *Exec*

Finally, the definition of the non-height-indexed relation.

definition *MS*::*Mbody* ⇒ *Label* ⇒ *State* ⇒ *Label* ⇒ *State* ⇒ *bool*
where *MS M l s ll t* = (∃ *k* . (*M*,*l*,*s*,*k*,*ll*,*t*):*MStep*)

4.2 Reachability relation

The second auxiliary operational judgement is required for the interpretation of invariants and method invariants. Invariants are expected to be satisfied in all heap components of (future) states that occur either in the same frame as the current state or a subframe thereof. Likewise, method invariants are expected to be satisfied by all heap components of states observed during the execution of a method, including subframes. None of the previous three operational judgements allows us to express these interpretations, as *Step* injects the execution of an invoked method as a single step. Thus, states occurring in subframes cannot be related to states occurring in the parent frame using these judgements. This motivates the introduction of predicates relating states s and t whenever the latter can be reach from the former, i.e. whenever t occurs as a successor of s in the same frame as s or one of its subframes. Again, we first define a relation that includes an explicit derivation height index.

inductive-set
Reachable::(*Mbody* × *Label* × *State* × *nat* × *State*) *set*
where
Reachable-zero: ⟦*k*=*0*; *t*=*s*⟧ ⟹ (*M*,*l*,*s*,*k*,*t*):*Reachable*
|
Reachable-step:
⟦ (*M*,*l*,*s*,*n*,*ll*,*r*):*Step*; (*M*,*ll*,*r*,*m*,*t*):*Reachable*; *k*=*Suc m*+*n*⟧
⟹ (*M*,*l*,*s*,*k*,*t*) : *Reachable*
|
Reachable-invS:
⟦ *mbody-is C m* (*par*,*code*,*l0*); *get-ins M l* = *Some* (*invokeS C m*);
s = (*ops*,*S*,*h*); (*ops*,*par*,*R*,*ops1*):*Frame*;
((*par*,*code*,*l0*), *l0*, ([],*R*,*h*), *n*, *t*):*Reachable*; *k*=*Suc n*⟧
⟹ (*M*,*l*,*s*,*k*,*t*) : *Reachable*

The following properties of are useful to notice.

lemma *ZeroHeightReachableElim*: $(M,l,s,0,r) \in Reachable \Longrightarrow r{=}s$
lemma *ReachableSplit*[*rule-format*]:
$(M,\ l,s,\ k,\ t) \in Reachable \Longrightarrow$
$1 \leq k \longrightarrow$
$((\exists\ n\ m\ r\ ll.\ (M,l,s,n,ll,r){:}Step \wedge$
$(M,ll,r,m,t){:}Reachable \wedge Suc\ m + n = k) \vee$
$(\exists\ n\ ops\ S\ h\ c\ m\ par\ R\ ops1\ code\ l0.$
$s{=}(ops,S,h) \wedge get\text{-}ins\ M\ l = Some\ (invokeS\ c\ m) \wedge$
$mbody\text{-}is\ c\ m\ (par,code,l0) \wedge (ops,par,R,ops1){:}Frame \wedge$
$((par,code,l0),\ l0,\ ([],R,h),\ n,\ t){:}Reachable \wedge Suc\ n = k))$
lemma *Reachable-returnElim*[*rule-format*]:
$(M,l,s,k,t) \in Reachable \Longrightarrow get\text{-}ins\ M\ l = Some\ vreturn \longrightarrow t{=}s$

Similar to the operational semantics, we define a variation of the reachability relation that hides the index.

definition *Reach*::*Mbody* ⇒ *Label* ⇒ *State* ⇒ *State* ⇒ *bool*
where *Reach M l s t* = (∃ *k* . (*M*,*l*,*s*,*k*,*t*):*Reachable*)

5 Soundness

This section contains the soundness proof of the program logic. In the first subsection, we define our notion of validity, thus formalising our intuitive explanation of the terms preconditions, specifications, and invariants. The following two subsections contain the details of the proof and can easily be skipped during a first pass through the document.

5.1 Validity

A judgement is valid at the program point $C.m.l$ (i.e. at label l in method m of class C), written *valid C m l A B I* or, in symbols,

$$\vDash \{A\}\, C, m, l\, \{B\}\, I,$$

if A is a precondition for B and for all local annotations following l in an execution of m, and all reachable states in the current frame or yet-to-be created subframes satisfy I. More precisely, whenever an execution of the method starting in an initial state s_0 reaches the label l with state s, the following properties are implied by $A(s_0, s)$.

1. If the continued execution from s reaches a final state t (i.e. the method terminates), then that final state t satisfies $B(s_0, s, t)$.

2. Any state s' visited in the current frame during the remaining program execution whose label carries an annotation Q will satisfy $Q(s_0, s')$, even if the execution of the frame does not terminate.

3. Any state s' visited in the current frame or a subframe of the current frame will satisfy $I(s_0, s, heap(s'))$, again even if the execution does not terminate.

Formally, this interpretation is expressed as follows.

definition *valid*::*Class* ⇒ *Method* ⇒ *Label* ⇒ *Assn* ⇒ *Post* ⇒ *Inv* ⇒ *bool* **where**
valid C m l A B I =
(∀ *M*. *mbody-is C m M* ⟶
(∀ *Mspec Minv Anno* . *MST*↓(*C*,*m*) = *Some*(*Mspec*,*Minv*,*Anno*) ⟶
(∀ *par code l0* . *M* = (*par*,*code*,*l0*) ⟶
(∀ *s0 s* . *MS M l0* (*mkState s0*) *l s* ⟶ *A s0 s* ⟶
((∀ *h v*. *Opsem M l s h v* ⟶ *B s0 s* (*h*,*v*)) ∧
(∀ *ll r* . (*MS M l s ll r* ⟶ (∀ *Q* . *Anno*↓(*ll*) = *Some Q* ⟶ *Q s0 r*)) ∧
(*Reach M l s r* ⟶ *I s0 s* (*heap r*)))))))

abbreviation *valid-syntax* :: *Assn* ⇒ *Class* ⇒ *Method* ⇒
Label ⇒ *Post* ⇒ *Inv* ⇒ *bool*
(⊨ {| - |} - , - , - {| - |} - [*200*,*200*,*200*,*200*,*200*,*200*] *200*)
where *valid-syntax A C m l B I* == *valid C m l A B I*

This notion of validity extends that of Bannwart-Müller by allowing the post-condition to differ from method specification and to refer to the initial state, and by including invariants. In the logic of Bannwart and Müller, the validity of a method specification is given by a partial correctness (Hoare-style) interpretation, while the validity of preconditions of individual instructions is such that a precondition at l implies the preconditions of its immediate control flow successors.

Validity us lifted to contexts and the method specification table. In the case of the former, we simply require that all entries be valid.

definition *G-valid*::*CTXT* ⇒ *bool* **where**
G-valid G = (∀ *C m l A B I*. *G*↓(*C*,*m*,*l*) = *Some* (*A*,*B*,*I*)⟶
⊨ {|*A*|} *C*, *m*, *l* {|*B*|} *I*)

Regarding the specification table, we require that the initial label of each method satisfies an assertion that ties the method precondition to the current state.

definition *MST-valid* ::*bool* **where**
MST-valid = (∀ *C m par code l0 T MI Anno*.
mbody-is C m (*par*,*code*,*l0*) ⟶ *MST*↓(*C*, *m*) = *Some* (*T*,*MI*,*Anno*) ⟶
⊨ {|(λ *s0 s*. *s* = *mkState s0*)|} *C*, *m*, *l0* {|(*mkPost T*)|} (*mkInv MI*))

definition *Prog-valid*::*bool* **where**
Prog-valid = (∃ *G* . *G-valid G* ∧ *MST-valid*)

The remainder of this section contains a proof of soundness, i.e. of the property

$$VP \Longrightarrow \textit{Prog-valid},$$

and is structured into two parts. The first step (Section 5.2) establishes a soundness result where the *VP* property is replaced by validity assumptions regarding the method specification table and the context. In the second step (Section 5.3), we show that these validity assumptions are satisfied by verified programs, which implies the overall soundness theorem.

5.2 Soundness under valid contexts

The soundness proof proceeds by induction on the axiomatic semantics, based on an auxiliary lemma for method invocations that is proven by induction on the derivation height of the operational semantics. For the latter induction, relativised notions of validity are employed that restrict the derivation height of the program continuations affected by an assertion. The appropriate definitions of relativised validity for judgements, for the precondition table, and for the method specification table are as follows.

definition *validn*::
  nat ⇒ *Class* ⇒ *Method* ⇒ *Label* ⇒ *Assn* ⇒ *Post* ⇒ *Inv* ⇒ *bool* **where**
validn K C m l A B I =
  (∀ *M*. *mbody-is C m M* ⟶
  (∀ *Mspec Minv Anno* . *MST*↓(*C*,*m*) = *Some*(*Mspec*,*Minv*,*Anno*) ⟶
  (∀ *par code l0* . *M* = (*par*,*code*,*l0*) ⟶
  (∀ *s0 s* . *MS M l0* (*mkState s0*) *l s* ⟶ *A s0 s* ⟶
  (∀ *k* . *k* ≤ *K* ⟶
   ((∀ *h v*. (*M*,*l*,*s*,*k*,*h*,*v*):*Exec* ⟶ *B s0 s* (*h*,*v*)) ∧
    (∀ *ll r* . ((*M*,*l*,*s*,*k*,*ll*,*r*):*MStep* ⟶
                (∀ *Q* . *Anno*↓(*ll*) = *Some Q* ⟶ *Q s0 r*)) ∧
               ((*M*,*l*,*s*,*k*,*r*):*Reachable* ⟶ *I s0 s* (*heap r*)))))))))

abbreviation *validn-syntax* :: *nat* ⇒ *Assn* ⇒ *Class* ⇒ *Method* ⇒
                  Label ⇒ *Post* ⇒ *Inv* ⇒ *bool*
(⊨- {| - |} - , - , - {| - |} - [*200*,*200*,*200*,*200*,*200*,*200*,*200*] *200*)
where *validn-syntax K A C m l B I* == *validn K C m l A B I*

definition *G-validn*::*nat* ⇒ *CTXT* ⇒ *bool* **where**
G-validn K G = (∀ *C m l A B I*. *G*↓(*C*,*m*,*l*) = *Some* (*A*,*B*,*I*) ⟶
                  ⊨$_K$ {|*A*|} *C*, *m*, *l* {|*B*|} *I*)

definition *MST-validn*::*nat* ⇒ *bool* **where**
MST-validn K = (∀ *C m par code l0 T MI Anno*.
  mbody-is C m (*par*,*code*,*l0*) ⟶ *MST*↓(*C*, *m*) = *Some* (*T*,*MI*,*Anno*) ⟶
  ⊨$_K$ {|(λ *s0 s*. *s* = *mkState s0*)|} *C*, *m*, *l0* {|(*mkPost T*)|} (*mkInv MI*))

definition *Prog-validn*::*nat* ⇒ *bool* **where**
Prog-validn K = (∃ *G* . *G-validn K G* ∧ *MST-validn K*)

The relativised notions are related to each other, and to the native notions of validity as follows.

lemma *valid-validn*: $\models \{\!|A|\!\}\ C,\ m,\ l\ \{\!|B|\!\}\ I \Longrightarrow \models_K \{\!|A|\!\}\ C,\ m,\ l\ \{\!|B|\!\}\ I$
lemma *validn-valid*: $[\![\forall\ K\ .\ \models_K \{\!|A|\!\}\ C,\ m,\ l\ \{\!|B|\!\}\ I]\!] \Longrightarrow \models \{\!|A|\!\}\ C,\ m,\ l\ \{\!|B|\!\}\ I$
lemma *validn-lower*:
$[\![\models_K \{\!|A|\!\}\ C,\ m,\ l\ \{\!|B|\!\}\ I;\ L \leq K]\!] \Longrightarrow \models_L \{\!|A|\!\}\ C,\ m,\ l\ \{\!|B|\!\}\ I$
lemma *G-valid-validn*: *G-valid G* $\Longrightarrow$ *G-validn K G*
lemma *G-validn-valid*:$[\![\forall\ K\ .\ G\text{-}validn\ K\ G]\!] \Longrightarrow G\text{-}valid\ G$
lemma *G-validn-lower*: $[\![G\text{-}validn\ K\ G;\ L \leq K]\!] \Longrightarrow G\text{-}validn\ L\ G$
lemma *MST-validn-valid*:$[\![\forall\ K.\ MST\text{-}validn\ K]\!] \Longrightarrow MST\text{-}valid$
lemma *MST-valid-validn*:*MST-valid* $\Longrightarrow$ *MST-validn K*
lemma *MST-validn-lower*: $[\![MST\text{-}validn\ K;\ L \leq K]\!] \Longrightarrow MST\text{-}validn\ L$

We define an abbreviation for the side conditions of the rule for static method invocations...

definition *INVS-SC*::
Class $\Rightarrow$ *Method* $\Rightarrow$ *Label* $\Rightarrow$ *Class* $\Rightarrow$ *Method* $\Rightarrow$ *MethSpec* $\Rightarrow$ *MethInv* $\Rightarrow$ *ANNO* $\Rightarrow$ *ANNO* $\Rightarrow$ *Mbody* $\Rightarrow$ *Assn* $\Rightarrow$ *Inv* $\Rightarrow$ *bool* **where**
INVS-SC C m l D m′ T MI Anno Anno2 M′ A I = ($\exists$ *M par code l0 T1 MI1*.
mbody-is C m M $\wedge$ *get-ins M l* = *Some* (*invokeS D m′*) $\wedge$
MST$\downarrow$(*C*,*m*) = *Some* (*T1*,*MI1*,*Anno*) $\wedge$
MST$\downarrow$(*D*,*m′*) = *Some* (*T*,*MI*,*Anno2*) $\wedge$
mbody-is D m′ M′ $\wedge$ *M′*=(*par*,*code*,*l0*) $\wedge$
($\forall$ *Q* . *Anno*$\downarrow$(*l*) = *Some Q* $\longrightarrow$ ($\forall$ *s0 s* . *A s0 s* $\longrightarrow$ *Q s0 s*)) $\wedge$
($\forall$ *s0 s* . *A s0 s* $\longrightarrow$ *I s0 s* (*heap s*)) $\wedge$
($\forall$ *s0 ops1 ops2 S R h t* . (*ops1*,*par*,*R*,*ops2*) : *Frame* $\longrightarrow$
A s0 (*ops1*,*S*,*h*) $\longrightarrow$ *MI* (*R*,*h*) *t* $\longrightarrow$ *I s0* (*ops1*,*S*,*h*) *t*))

...and another abbreviation for the soundness property of the same rule.

definition *INVS-soundK*::
nat $\Rightarrow$ *CTXT* $\Rightarrow$ *Class* $\Rightarrow$ *Method* $\Rightarrow$ *Label* $\Rightarrow$ *Class* $\Rightarrow$ *Method* $\Rightarrow$ *MethSpec* $\Rightarrow$ *MethInv* $\Rightarrow$ *ANNO* $\Rightarrow$ *ANNO* $\Rightarrow$ *Mbody* $\Rightarrow$ *Assn* $\Rightarrow$ *Post* $\Rightarrow$ *Inv* $\Rightarrow$ *bool* **where**
INVS-soundK K G C m l D m′ T MI Anno Anno2 M′ A B I =
(*INVS-SC C m l D m′ T MI Anno Anno2 M′ A I* $\longrightarrow$
G-validn K G $\longrightarrow$ *MST-validn K* $\longrightarrow$
$\models_K \{\!|$(*SINV-pre* (*fst M′*) *T A*)$|\!\}$ *C*,*m*,(*l*+1)
$\{\!|$(*SINV-post* (*fst M′*) *T B*)$|\!\}$ (*SINV-inv* (*fst M′*) *T I*)
$\longrightarrow \models_{(K+1)} \{\!|\ A\ |\!\}\ C,m,l\ \{\!|\ B\ |\!\}\ I$)

The proof that this property holds for all K proceeds by induction on K.

lemma *INVS-soundK-all*:
INVS-soundK K G C m l D m′ T MI Anno Anno2 M′ A B I

The heart of the soundness proof - the induction on the axiomatic semantics.

lemma *SOUND-Aux*[*rule-format*]:
(*b*,*G*,*C*,*m*,*l*,*A*,*B*,*I*):*SP-Judgement* $\Longrightarrow$ *G-validn K G* $\longrightarrow$ *MST-validn K* $\longrightarrow$
((*b* $\longrightarrow \models_K \{\!|A|\!\}\ C,\ m,\ l\ \{\!|B|\!\}\ I$) $\wedge$
((¬ *b*) $\longrightarrow \models_{(Suc\ K)} \{\!|A|\!\}\ C,\ m,\ l\ \{\!|B|\!\}\ I$))

The statement of this lemma gives a semantic interpretation of the two judgement forms, as *SP-Assum*-judgements enjoy validity up to execution height K, while *SP-Deriv*-judgements are valid up to level $K + 1$.

From this, we obtain a soundness result that still involves context validity.

theorem *SOUND-in-CTXT*:
$[\![G \rhd \{\!|A|\!\}\ C,m,l\ \{\!|B|\!\}\ I;\ G\text{-}valid\ G;\ MST\text{-}valid]\!] \Longrightarrow \models \{\!|A|\!\}\ C,\ m,\ l\ \{\!|B|\!\}\ I$

We will now show that the two semantic assumptions can be replaced by the verified-program property.

5.3 Soundness of verified programs

In order to obtain a soundness result that does not require validity assumptions of the context or the specification table, we show that the *VP* property implies context validity. First, the elimination of contexts. By induction on k we prove

lemma *VPG-MSTn-Gn*[*rule-format*]:
$VP\text{-}G\ G \longrightarrow MST\text{-}validn\ k \longrightarrow G\text{-}validn\ k\ G$

which implies

lemma *VPG-MST-G*: $[\![VP\text{-}G\ G;\ MST\text{-}valid]\!] \Longrightarrow G\text{-}valid\ G$

Next, the elimination of *MST-valid*. Again by induction on k, we prove

lemma *VPG-MSTn*[*rule-format*]: $VP\text{-}G\ G \longrightarrow MST\text{-}validn\ k$

which yields

lemma *VPG-MST*: $VP\text{-}G\ G \Longrightarrow MST\text{-}valid$

Combining these two results, and unfolding the definition of program validity yields the final soundness result.

theorem *VP-VALID*: $VP \Longrightarrow Prog\text{-}valid$

6 A derived logic for a strong type system

In this section we consider a system of derived assertions, for a type system for bounded heap consumption. The type system arises by reformulating the analysis of Cachera, Jensen, Pichardie, and Schneider [4] for a high-level functional language. The original approach of Cachera et al. consists of formalising the correctness proof of a certain analysis technique in Coq. Consequently, the verification of a program requires the execution of the analysis algorithm inside the theorem prover, which involves the computation of the (method) call graph and fixed point iterations. In contrast, our

approach follows the proof-carrying code paradigm more closely: the analysis amounts to a type inference which is left unformalised and can thus be carried out outside the trusted code base. Only the result of the analysis is communicated to the code recipient. The recipient verifies the validity of the certificate by a largely syntax-directed single-pass traversal of the (low-level) code using a domain-specific program logic. This approach to proof-carrying code was already explored in the MRG project, with respect to program logics of partial correctness [3] and a type system for memory consumption by Hofmann and Jost [7]. In order to obtain syntax-directedness of the proof rules, these had to be formulated at the granularity of typing judgements. In contrast, the present proof system admits proof rules for individual JVM instructions.

Having derived proof rules for individual JVM instructions, we introduce a type system for a small functional language, and a compilation into bytecode. The type system associates a natural number n to an expression e, in a typing context Σ. Informally, the interpretation of a typing judgement $\Sigma \rhd e : n$ is that the evaluation of e (which may include the invocation of functions whose resource behaviour is specified in Σ) does not perform more than n allocations. The type system is then formally proven sound, using the derived logic for bytecode. By virtue of the invariants, the guarantee given by the present system is stronger than the one given by our encoding of the Hofmann-Jost system, as even non-terminating programs can be verified in a meaningful way.

6.1 Syntax and semantics of judgements

The formal interpretation at JVM level of a type n is given by a triple

$$Cachera(n) = (A, B, I)$$

consisting of a (trivial) precondition, a post-condition, and a strong invariant.

definition *Cachera*::*nat* ⇒ (*Assn* × *Post* × *Inv*) **where**
Cachera n = (λ *s0 s* . *True*,
λ *s0* (*ops*,*s*,*h*) (*k*,*v*) . |*k*| ≤ |*h*| + *n*,
λ *s0* (*ops*,*s*,*h*) *k*. |*k*| ≤ |*h*| + *n*)

This definition is motivated by the expectation that $\rhd\{A\}\ulcorner e\urcorner\{B\}\, I$ should be derivable whenever the type judgement $\Sigma \rhd e : n$ holds, where $\ulcorner e\urcorner$ is the translation of compiling the expression e into JVML, and the specification table MST contains the interpretations of the entries in Σ.

We abbreviate the above construction of judgements by a predicate *deriv*.

definition *deriv*::*CTXT* ⇒ *Class* ⇒ *Method* ⇒ *Label* ⇒
(*Assn* × *Post* × *Inv*) ⇒ *bool* **where**

deriv G C m l (ABI) = (let (A,B,I) = ABI in (G ▷ {| A |} C,m,l {| B |} I))

Thus, the intended interpretation of a typing judgement $\Sigma \rhd e : n$ is

$$deriv\ C\ m\ l\ (Cachera\ n)$$

if e translates to a code block whose first instruction is at $C.m.l$.

We also define a judgement of the auxiliary form of sequents.

definition *derivAssum*::*CTXT* ⇒ *Class* ⇒ *Method* ⇒ *Label* ⇒
(*Assn* × *Post* × *Inv*) ⇒ *bool* **where**
derivAssum G C m l (ABI) = (let (A,B,I) = ABI in G ▷ ⟨ A ⟩ C,m,l ⟨ B ⟩ I)

The following operation converts a derived judgement into the syntactical form of method specifications.

definition *mkSPEC*::(*Assn* × *Post* × *Inv*) ⇒ *ANNO* ⇒
(*MethSpec* × *MethInv* × *ANNO*) **where**
mkSPEC (ABI) Anno = (let (A,B,I) = ABI in
(λ s0 t . B s0 (mkState s0) t, λ s0 h . I s0 (mkState s0) h, Anno))

This enables the interpretation of typing contexts Σ as a set of constraints on the specification table *MST*.

6.2 Derived proof rules

We are now ready to prove derived rules, i.e. proof rules where assumptions as well as conclusions are of the restricted assertion form. While their justification unfolds the definition of the predicate *deriv*, their application will not. We first give syntax-directed proof rules for all JVM instructions:

lemma *CACH-NEW*:
⟦ *ins-is C m l (new c)*; *MST↓(C,m)=Some(Mspec,Minv,Anno)*;
Anno↓(l) = None; *n = k + 1*; *derivAssum G C m (l+1) (Cachera k)* ⟧
⟹ *deriv G C m l (Cachera n)*
lemma *CACH-INSTR*:
⟦ *ins-is C m l I*;
I ∈ { *const c*, *dup*, *pop*, *swap*, *load x*, *store x*, *binop f*,
unop g, *getfield d F*, *putfield d F*, *checkcast d*};
MST↓(C,m)=Some(Mspec,Minv,Anno); *Anno↓(l) = None*;
derivAssum G C m (l+1) (Cachera n) ⟧
⟹ *deriv G C m l (Cachera n)*
lemma *CACH-RET*:
⟦ *ins-is C m l vreturn*; *MST↓(C,m)=Some(Mspec,Minv,Anno)*;
Anno↓(l) = None ⟧
⟹ *deriv G C m l (Cachera 0)*
lemma *CACH-GOTO*:
⟦ *ins-is C m l (goto pc)*; *MST↓(C,m)=Some(Mspec,Minv,Anno)*;
Anno↓(l) = None; *derivAssum G C m pc (Cachera n)* ⟧
⟹ *deriv G C m l (Cachera n)*

lemma *CACH-IF*:
⟦ *ins-is C m l* (*iftrue pc*); *MST*↓(*C*,*m*)=*Some*(*Mspec*,*Minv*,*Anno*);
Anno↓(*l*) = *None*; *derivAssum G C m pc* (*Cachera n*);
derivAssum G C m (*l*+1) (*Cachera n*) ⟧
⟹ *deriv G C m l* (*Cachera n*)
lemma *CACH-INVS*:
⟦ *ins-is C m l* (*invokeS D m'*); *mbody-is D m'* (*par*,*code*,*l0*);
MST↓(*C*,*m*)=*Some*(*Mspec*,*Minv*,*Anno*); *Anno*↓(*l*) = *None*;
MST↓(*D*, *m'*) = *Some*(*mkSPEC* (*Cachera k*) *Anno2*);
nk = *n*+*k*; *derivAssum G C m* (*l*+1) (*Cachera n*)⟧
⟹ *deriv G C m l* (*Cachera nk*)

In addition, we have two rules for subtyping

lemma *CACH-SUB*:
⟦ *deriv G C m l* (*Cachera n*); *n* ≤ *k*⟧ ⟹ *deriv G C m l* (*Cachera k*)
lemma *CACHAssum-SUB*:
⟦ *derivAssum G C m l* (*Cachera n*); *n* ≤ *k*⟧
⟹ *derivAssum G C m l* (*Cachera k*)

and specialised forms of the axiom rule and the injection rule.

lemma *CACH-AX*:
⟦ *G*↓(*C*,*m*,*l*) = *Some* (*Cachera n*); *MST*↓(*C*,*m*)=*Some*(*Mspec*,*Minv*,*Anno*);
Anno↓(*l*) = *None*⟧
⟹ *derivAssum G C m l* (*Cachera n*)
lemma *CACH-INJECT*:
deriv G C m l (*Cachera n*) ⟹ *derivAssum G C m l* (*Cachera n*)

Finally, a verified-program rule relates specifications to judgements for the method bodies. Thus, even the method specifications may be given as derived assertions (modulo the *mkSPEC*-conversion).

lemma *CACH-VP*:
⟦ ∀ *c m par code l0*. *mbody-is c m* (*par*, *code*, *l0*) ⟶
(∃ *n Anno* . *MST*↓(*c*,*m*) = *Some*(*mkSPEC*(*Cachera n*) *Anno*) ∧
deriv G c m l0 (*Cachera n*));
∀ *c m l A B I*. *G*↓(*c*,*m*,*l*) = *Some*(*A*,*B*,*I*) ⟶
(∃ *n* . (*A*,*B*,*I*) = *Cachera n* ∧ *deriv G c m l* (*Cachera n*))⟧
⟹ *VP*

6.3 Soundness of high-level type system

We define a first-order functional language where expressions are stratified into primitive expressions and general expressions. The language supports the construction of lists using constructors *NilPrim* and *ConsPrim h t*, and includes a corresponding pattern match operation. In order to simplify the compilation, function identifiers are taken to be pairs of class names and method names.

type-synonym *Fun* = *Class* × *Method*

datatype *Prim* =
  IntPrim int
| *UnPrim Val* ⇒ *Val Var*
| *BinPrim Val* ⇒ *Val* ⇒ *Val Var Var*
| *NilPrim*
| *ConsPrim Var Var*
| *CallPrim Fun Var list*

datatype *Expr* =
  PrimE Prim
| *LetE Var Prim Expr*
| *CondE Var Expr Expr*
| *MatchE Var Expr Var Var Expr*

type-synonym *FunProg* = (*Fun*,*Var list* × *Expr*) *AssList*

The type system uses contexts that associate a type (natural number) to function identifiers.

type-synonym *TP-Sig* = (*Fun*, *nat*) *AssList*

We first give the rules for primitive expressions.

inductive-set *TP-prim*::(*TP-Sig* × *Prim* × *nat*)*set*
where
TP-int: (Σ,*IntPrim i*,*0*) : *TP-prim*
|
TP-un: (Σ,*UnPrim f x*,*0*) : *TP-prim*
|
TP-bin: (Σ,*BinPrim f x y*,*0*) : *TP-prim*
|
TP-nil: (Σ,*NilPrim*,*0*) : *TP-prim*
|
TP-cons: (Σ,*ConsPrim x y*,*1*) : *TP-prim*
|
TP-Call: ⟦Σ↓*f* = *Some n*⟧ ⟹ (Σ,*CallPrim f args*,*n*) : *TP-prim*

Next, the rules for general expressions.

inductive-set *TP-expr*::(*TP-Sig* × *Expr* × *nat*) *set*
where
TP-sub: ⟦(Σ,*e*,*m*):*TP-expr*; *m* ≤ *n*⟧ ⟹ (Σ,*e*,*n*):*TP-expr*
|
TP-prim:⟦(Σ,*p*,*n*):*TP-prim*⟧ ⟹ (Σ,*PrimE p*,*n*) : *TP-expr*
|
TP-let: ⟦(Σ,*p*,*k*):*TP-prim*; (Σ,*e*,*m*):*TP-expr*; *n* = *k*+*m*⟧
  ⟹ (Σ,*LetE x p e*,*n*) : *TP-expr*
|
TP-Cond:⟦(Σ,*e1*,*n*):*TP-expr*; (Σ,*e2*,*n*):*TP-expr*⟧
  ⟹(Σ,*CondE x e1 e2*,*n*) : *TP-expr*
|

TP-Match:⟦(Σ,*e1*,*n*):*TP-expr*; (Σ,*e2*,*n*):*TP-expr* ⟧
⟹ (Σ,*MatchE x e1 h t e2*,*n*):*TP-expr*

A functional program is well-typed if its domain agrees with that of some context such that each function body validates the context entry.

definition *TP*::*TP-Sig* ⇒ *FunProg* ⇒ *bool* **where**
TP Σ *F* = ((∀ *f* . (Σ↓*f* = *None*) = (*F*↓*f* = *None*)) ∧
(∀ *f n par e* . Σ↓*f* = *Some n* ⟶ *F*↓*f* = *Some* (*par*,*e*) ⟶ (Σ,*e*,*n*):*TP-expr*))

For the translation into bytecode, we introduce identifiers for a class of lists, the expected field names, and a temporary (reserved) variable name.

axiomatization
LIST::*Class* **and**
HD::*Field* **and**
TL::*Field* **and**
tmp::*Var*

The compilation of primitive expressions extends a code block by a sequence of JVM instructions that leave a value on the top of the operand stack.

inductive-set *compilePrim*::
(*Label* × (*Label*,*Instr*) *AssList* × *Prim* × ((*Label*,*Instr*) *AssList* × *Label*)) *set*
where
compileInt: (*l*, *code*, *IntPrim i*, (*code*[*l*↦(*const* (*IVal i*))],*l*+*1*)) : *compilePrim*
|
compileUn:
(*l*, *code*, *UnPrim f x*, (*code*[*l*↦(*load x*)][(*l*+*1*)↦(*unop f*)],*l*+*2*)) : *compilePrim*
|
compileBin:
(*l*, *code*, *BinPrim f x y*,
(*code*[*l*↦(*load x*)][(*l*+*1*)↦(*load y*)][(*l*+*2*)↦(*binop f*)],*l*+*3*)) : *compilePrim*
|
compileNil:
(*l*, *code*, *NilPrim*, (*code*[*l*↦(*const* (*RVal Nullref*))],*l*+*1*)) : *compilePrim*
|
compileCons:
(*l*, *code*, *ConsPrim x y*,
(*code*[*l*↦(*load y*)][(*l*+*1*)↦(*load x*)]
[(*l*+*2*)↦(*new LIST*)][(*l*+*3*)↦*store tmp*]
[(*l*+*4*)↦*load tmp*][(*l*+*5*)↦(*putfield LIST HD*)]
[(*l*+*6*)↦*load tmp*][(*l*+*7*)↦(*putfield LIST TL*)]
[(*l*+*8*)↦(*load tmp*)], *l*+*9*)) : *compilePrim*
|
compileCall-Nil:
(*l*, *code*, *CallPrim f* [],(*code*[*l*↦*invokeS* (*fst f*) (*snd f*)],*l*+*1*)): *compilePrim*
|
compileCall-Cons:
⟦ (*l*+*1*,*code*[*l*↦*load x*], *CallPrim f args*, *OUT*) : *compilePrim*⟧
⟹ (*l*, *code*, *CallPrim f* (*x*#*args*), *OUT*): *compilePrim*

The following lemma shows that the resulting code is an extension of the code submitted as an argument, and that the new instructions define a contiguous block.

lemma *compilePrim-Prop1*[*rule-format*]:
$(l,\ code,\ p,\ OUT) : compilePrim \Longrightarrow$
$(\forall\ code1l1\ .\ OUT = (code1,\ l1) \longrightarrow$
$(l < l1 \wedge (\forall\ ll\ .\ ll < l \longrightarrow code1{\downarrow}ll = code{\downarrow}ll) \wedge$
$(\forall\ ll\ .\ l \leq ll \longrightarrow ll < l1 \longrightarrow (\exists\ ins\ .\ code1{\downarrow}ll = Some\ ins))))$

A signature corresponds to a method specification table if all context entries are represented as MST entries and method names that are defined in the global program P.

definition *Sig-good*::*TP-Sig* $\Rightarrow$ *bool* **where**
Sig-good $\Sigma =$
$(\forall\ C\ m\ n.\ \Sigma{\downarrow}(C,m) = Some\ n \longrightarrow$
$(MST{\downarrow}(C,\ m) = Some\ (mkSPEC\ (Cachera\ n)\ emp) \wedge$
$(\exists\ par\ code\ l0\ .\ mbody\text{-}is\ C\ m\ (par,code,l0))))$

This definition requires MST to associate the specification

$$mkSPEC\ (Cachera\ n)\ emp$$

to each method to which the type signature associates the type n. In particular, this requires the annotation table of such a method to be empty. Additionally, the global program P is required to contain a method definition for each method (i.e. function) name occurring in the domain of the signature.

An auxiliary abbreviation that captures when a block of code has trivial annotations and only comprises defined program labels.

definition *Segment*::
Class $\Rightarrow$ *Method* $\Rightarrow$ *Label* $\Rightarrow$ *Label* $\Rightarrow$ (*Label*,*Instr*)*AssList* $\Rightarrow$ *bool*
where
Segment C m l l1 code $=$
$(\exists\ Mspec\ Minv\ Anno\ .\ MST{\downarrow}(C,m) = Some(Mspec,Minv,Anno) \wedge$
$(\forall\, ll.\ l \leq ll \longrightarrow ll < l1 \longrightarrow$
$Anno{\downarrow}(ll) = None \wedge (\exists\, ins.\ ins\text{-}is\ C\ m\ ll\ ins \wedge code{\downarrow}ll = Some\ ins)))$

The soundness of (the translation of) a function call is proven by induction on the list of arguments.

lemma *Call-SoundAux*[*rule-format*]:
$\Sigma{\downarrow}f = Some\ n \longrightarrow$
$MST{\downarrow}(fst\ f,snd\ f) = Some(mkSPEC\ (Cachera\ n)\ Anno2) \longrightarrow$
$(\exists\ par\ body\ l0\ .\ mbody\text{-}is\ (fst\ f)\ (snd\ f)\ (par,body,l0)) \longrightarrow$
$(\forall\, l\ code\ code1\ l1\ G\ C\ m\ T\ MI\ k.$
$(l,\ code,\ CallPrim\ f\ args,\ code1,\ l1) \in compilePrim \longrightarrow$

$MST{\downarrow}(C,\ m) = Some\ (T,\ MI,Anno) \longrightarrow Segment\ C\ m\ l\ l1\ code1 \longrightarrow$
$derivAssum\ G\ C\ m\ l1\ (Cachera\ k) \longrightarrow$
$deriv\ G\ C\ m\ l\ (Cachera\ (n+k)))$

lemma *Call-Sound*:
$[\![\ Sig\text{-}good\ \Sigma;\ \Sigma{\downarrow}f = Some\ n;$
$(l,\ code,\ CallPrim\ f\ args,\ code1\ ,\ l1) \in compilePrim;$
$MST{\downarrow}(C,m) = Some\ (T,\ MI,Anno);\ Segment\ C\ m\ l\ l1\ code1;$
$derivAssum\ G\ C\ m\ l1\ (Cachera\ nn);\ k = n+nn]\!]$
$\Longrightarrow deriv\ G\ C\ m\ l\ (Cachera\ k)$

The definition of basic instructions.

definition *basic*::*Instr* $\Rightarrow$ *bool* **where**
$basic\ ins = ((\exists\ c\ .\ ins = const\ c) \vee ins = dup \vee$
$ins{=}\ pop \vee ins{=}\ swap \vee (\exists\ x.\ ins{=}\ load\ x) \vee$
$(\exists\ y.\ ins{=}\ store\ y) \vee (\exists\ f.\ ins{=}\ binop\ f) \vee$
$(\exists\ g.\ ins{=}\ unop\ g) \vee (\exists\ c1\ F1.\ ins{=}\ getfield\ c1\ F1) \vee$
$(\exists\ c2\ F2.\ ins{=}\ \ putfield\ c2\ F2) \vee$
$(\exists\ c3.\ ins{=}\ \ checkcast\ c3))$

Next, we prove the soundness of basic instructions. The hypothesis refers to instructions located at the program continuation.

lemma *Basic-Sound*:
$[\![\ Segment\ C\ m\ l\ ll\ code;\ MST{\downarrow}(C,m) = Some\ (T,\ MI,Anno);\ l{\leq}l1;\ l1 < ll;$
$l2{=}l1{+}1;\ code{\downarrow}l1 = Some\ ins;\ basic\ ins;\ derivAssum\ G\ C\ m\ l2\ (Cachera\ n)]\!]$
$\Longrightarrow deriv\ G\ C\ m\ l1\ (Cachera\ n)$

Following this, the soundness of the type system for primitive expressions. The proof proceeds by induction on the typing judgement.

lemma *TP-prim-Sound*[*rule-format*]:
$(\Sigma,p,n){:}TP\text{-}prim \Longrightarrow$
$Sig\text{-}good\ \Sigma \longrightarrow$
$(\forall\ l\ code\ code1\ l1\ G\ C\ m\ T\ MI\ Anno\ nn\ k.$
$(l,\ code,\ p,\ (code1,l1)) : compilePrim \longrightarrow$
$MST{\downarrow}(C,m) = Some\ (T,MI,Anno) \longrightarrow$
$Segment\ C\ m\ l\ l1\ code1 \longrightarrow derivAssum\ G\ C\ m\ l1\ (Cachera\ nn) \longrightarrow$
$k = n+nn \longrightarrow deriv\ G\ C\ m\ l\ (Cachera\ k))$

The translation of general expressions is defined similarly, but no code continuation is required.

inductive-set *compileExpr*::
$(Label \times (Label,Instr)\ AssList \times Expr \times ((Label,Instr)\ AssList \times Label))\ set$
where
compilePrimE:
$[\![(l,\ code,\ p,\ (code1,l1)) : compilePrim;\ OUT = (code1[l1{\mapsto}vreturn],l1{+}1)]\!]$
$\Longrightarrow (l,\ code,\ PrimE\ p,\ OUT){:}compileExpr$
|
compileLetE:
$[\![(l,\ code,\ p,\ (code1,l1)) : compilePrim;\ (code2,l2) = (code1[l1{\mapsto}(store\ x)],l1{+}1);$

```
    (l2, code2, e, OUT) : compileExpr⟧
  ⟹ (l, code, LetE x p e, OUT) : compileExpr
|
compileCondE:
  ⟦(l+2, code, e2, (codeElse,XXX)) : compileExpr;
    (XXX, codeElse, e1, (codeThen,YYY) ) : compileExpr ;
    OUT = (codeThen[l↦load x][(l+1)↦(iftrue XXX)], YYY)⟧
  ⟹ (l, code, CondE x e1 e2, OUT): compileExpr
|
compileMatchE:
  ⟦(l+9, code, e2, (codeCons,lNil)) : compileExpr;
    (lNil, codeCons, e1, (codeNil,lRes) ) : compileExpr ;
    OUT = (codeNil[l↦(load x)]
                [(l+1)↦(unop (λ v . if v = RVal Nullref
                                then TRUE else FALSE))]
                [(l+2)↦(iftrue lNil)]
                [(l+3)↦(load x)]
                [(l+4)↦(getfield LIST HD)]
                [(l+5)↦(store h)]
                [(l+6)↦(load x)]
                [(l+7)↦(getfield LIST TL)]
                [(l+8)↦(store t)], lRes) ⟧
  ⟹ (l, code, MatchE x e1 h t e2, OUT): compileExpr
```

Again, we prove an auxiliary result on the emitted code, by induction on the compilation judgement.

```
lemma compileExpr-Prop1[rule-format]:
(l,code,e,OUT) : compileExpr ⟹
  (∀ code1 l1 . OUT = (code1, l1) ⟶
     (l < l1 ∧
      (∀ ll . ll < l ⟶ code1↓ll = code↓ll) ∧
      (∀ ll . l ≤ ll ⟶ ll < l1 ⟶ (∃ ins . code1↓ll = Some ins))))
```

Then, soundness of the epxression type system is proven by induction on the typing judgement.

```
lemma TP-epxr-Sound[rule-format]:
(Σ,e,n):TP-expr ⟹ Sig-good Σ ⟶
 (∀ l code code1 l1 G C m T MI Anno.
    (l, code, e, (code1,l1)):compileExpr ⟶
    MST↓(C,m) = Some (T,MI,Anno) ⟶
    Segment C m l l1 code1 ⟶ deriv G C m l (Cachera n))
```

The full translation of a functional program into a bytecode program is defined as follows.

```
definition compileProg::FunProg ⇒ bool where
compileProg F =
  ((∀ C m par e. F↓(C,m) = Some(par,e) ⟶
        (∃ code l0 l. mbody-is C m (rev par,code,l0) ∧
```

$$(l0,[],e,(code,l)){:}compileExpr)) \wedge$$
$$(\forall\ C\ m.\ (\exists\ M.\ mbody\text{-}is\ C\ m\ M) = (\exists\ fdecl\ .\ F{\downarrow}(C,m) = Some\ fdecl)))$$

The final condition relating a typing context to a method specification table.

definition *TP-MST*::*TP-Sig* ⇒ *bool* **where**
TP-MST Σ =
  (∀ *C m* . *case* (*MST*↓(*C*,*m*)) *of*
      None ⇒ Σ↓(*C*,*m*) = *None*
    | *Some*(*T*,*MI*,*Anno*) ⇒ *Anno* = *emp* ∧
        (∃ *n* . Σ↓(*C*,*m*)=*Some n* ∧
        (*T*,*MI*,*Anno*) = *mkSPEC* (*Cachera n*) *emp*))

For well-typed programs, this property implies the earlier condition on signatures.

lemma *translation-good*: ⟦*compileProg F*; *TP-MST* Σ; *TP* Σ *F*⟧ ⟹ *Sig-good* Σ

We can thus prove that well-typed function bodies satisfy the specifications asserted by the typing context.

lemma *CACH-BodiesDerivable*[*rule-format*]:
  ⟦ *mbody-is C m* (*par*, *code*, *l*); *compileProg F*; *TP-MST* Σ; *TP* Σ *F*⟧
  ⟹ ∃ *n* . *MST*↓(*C*,*m*) = *Some*(*mkSPEC*(*Cachera n*) *emp*) ∧
      deriv [] *C m l* (*Cachera n*)

From this, the overall soundness result follows easily.

theorem *CACH-VERIFIED*: ⟦*TP* Σ *F*; *TP-MST* Σ; *compileProg F*⟧ ⟹ *VP*

References

[1] F. Y. Bannwart and P. Müller. A logic for bytecode. *Electronic Notes in Theoretical Computer Science*, 141(1):255–273, 2005.

[2] L. Beringer and M. Hofmann. A bytecode logic for JML and types. In N. Kobayashi, editor, *Proceedings of the 4th Asian Symposium on Programming Languages and Systems (APLAS 2006)*, volume 4279 of *LNCS*, pages 389–405. Springer, 2006.

[3] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic certification of heap consumption. In A. V. Franz Baader, editor, *LPAR 2004*, volume 3425 of *LNCS*, pages 347–362. Springer-Verlag, 2005.

[4] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. In *Proceedings of the 13th International Symposium on Formal Methods (FM'05)*, volume 3582 of *LNCS*, pages 91–106. Springer-Verlag, 2005.

[5] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the International Conference on Computer Languages*, pages 2–13. IEEE, Apr. 1992.

[6] R. Hähnle and W. Mostowski. Verification of safety properties in the presence of transactions. In *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *LNCS*, pages 151–171. Springer, 2005.

[7] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages (POPL'03)*, pages 185–197. ACM Press, Jan. 2003.

[8] H. M. B. Jonkers. Abstract storage structures. *Algorithmic languages*, 1981.

[9] Mobius Consortium. Deliverable 3.1: Bytecode specification language and program logic, 2006. Available online from http://mobius.inria.fr.

[10] D. Pichardie. Bicolano – Byte Code Language in Coq. http://mobius.inria.fr/twiki/bin/view/Bicolano. Summary appears in [9], 2006.

[11] U. S. Reddy. Global state considered unnecessary: An introduction to object-based semantics. *Journal of Lisp and Symbolic Computation*, 9:7–76, 1996.