A Denotational Semantic Model for Validating JVML/CLDC Optimizations under Isabelle/HOL

Hamdi Yahyaoui[†‡] Mourad Debbabi[‡] Nadia Tawbi[††]

[†]Electrical and Computer Engineering Department,
University Of Sharjah, Sharjah, UAE.
[‡]Concordia Institute for Information Systems Engineering,
Concordia University, Quebec, Canada.
[††]Computer Science and Software Engineering Department,
Laval University, Quebec, Canada.
hyahyaoui@sharjah.ac.ae,{hamdi,debbabi}@ciise.concordia.ca,tawbi@ift.ulaval.ca

Abstract

The main intent of this paper is to present a semantic framework for the validation of JVML/CLDC optimizations. The semantic style of the framework is denotational and rests on an extension of the resource pomsets semantics of Gastin and Mislove [12]. The resource pomsets is a fully abstract semantic model that is based on true concurrency. However, it does not support non-determinism that emerges while interpreting JVML/CLDC programs. In this paper, we present an extension of this model that aims to support unbounded non-determinism. More precisely, we give an overview of the construction of the process space and exhibit its algebraic properties. The elaborated semantics is embedded in the proof assistant Isabelle [28] in order to validate optimizations of JVML/CLDC programs. A case study for the validation of some optimizations of JVML/CLDC programs is also presented. The studied optimizations are: constant propagation and dead assignment elimination.

Keywords: Denotational Semantics, True Concurrency, Optimizations Validation, Proof Assistant, JVML/CLDC, KVM

I. Motivations and Background

The main intent of this paper is to provide a semantic framework for the validation of JVML/CLDC optimizations. This work is a natural continuation of a successful project on the acceleration of embedded Java virtual machines [8], [10]. In fact, we have designed and implemented successfully several acceleration techniques for the threading, caching, lookup and interpretation mechanisms. The most effective among these techniques was the acceleration of the virtual machine interpretation through selective dynamic compilation. This technique consists of detecting, at runtime, the so-called hotspots (most frequently called code fragments) and translating them into the native code of the host machine. By doing so, we reached significant speedup ratios. After completing the design, implementation and benchmarking all of these techniques, arose the issue of formally establishing their semantic correctness.

Establishing the semantic correctness of an optimization technique consists of proving that the optimization preserves the semantics i.e. the original program and the optimized one are semantically equivalent. This entails the elaboration of one semantics if the original and optimized programs are both expressed in the same language. In the case of dynamic compilation, we are in the presence of two languages[1]: the source and the target languages. Hence, this means that two semantics are needed.

In the literature on programming languages, many researchers have used the method introduced by Morris in [26], further promoted in [40], to establish the correctness of a compilation/optimization process. This approach advocates the use of algebraic data types and algebraic semantics to capture the optimization correctness. This amounts to the commutation of the diagram reported in Figure 1. Later, this approach was accommodated to use an operational semantic style as what Stephenson proposed in [38] or a denotational semantic style as what Wand proposed in [42]. In a denotational semantic setting, the correctness of the compiler is expressed as the equality of the denotation of the source program and the denotation of its translation. This paradigm for proving compiler correctness is outlined in Figure 2.

Many advantages cater for the adoption of a denotational semantic style for proving compiler correctness. In fact, denotational semantics has strong mathematical foundations, is compositional and more abstract than operational semantics. Furthermore, by adopting a denotational semantic style, we can encode the semantics of the source and target languages in the same model. This is unaffordable in an operational semantic setting if the source and target languages are different.

The source language in our compilation framework is JVML/CLDC, which is a concurrent language. Hence, we have to select or elaborate an adequate concurrency model. Concurrency models are classified w.r.t three major criteria [43]: the focus on the state or the behavior (intensional versus extensional), the treatment of parallelism through interleaving or true concurrency and the way non-determinism is handled (branching versus linear).

[1]In our project, the source code is JVML/CLDC (Java Virtual Machine Language for Connected Limited Device Configuration) and the target code is the binary language of ARM processors.

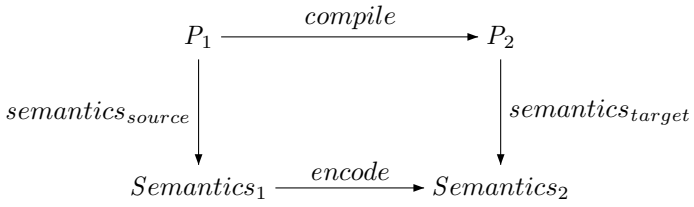

Fig. 1. Morris Approach to Compiler Correctness

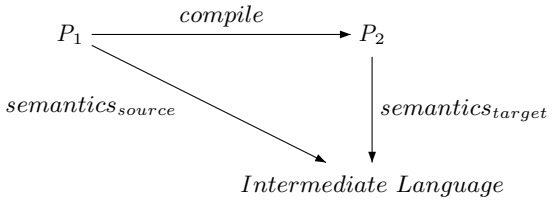

Fig. 2. Wand Paradigm for Proving Compiler Correctness

Famous denotational models such as failure sets and acceptance trees [7], [16] are extensional and interleaving models. More precisely, in these models, the parallel composition of two processes is defined as all the possible interleaving between them. Unfortunately, this leads to a state explosion problem. Accordingly, true concurrency models minimize this state explosion using partial orders. In fact, in these models, independent events/actions can be executed simultaneously. However, the major limitation of the well-known non-interleaving models, such as labelled event structures [44], is the lack of explicit description of true concurrency. Lately, Gastin and Mislove [12] provide such explicit description in a model they called resource pomsets. The main interesting feature of this model is the concept of resources. For our research, the resource concept is useful since we aim to provide a semantics that describes in an explicit way the JVML/CLDC synchronization mechanism. For instance, the resources can be used to denote objects that can be locked by threads. Unfortunately, the resource pomset model reduces parallel composition of events which share the same resource to a deadlock. This makes the model inadequate for JVML/CLDC. In fact, non-determinism emerges in the execution of some JVML/CLDC programs. For instance, if two JVML/CLDC threads simultaneously try to execute the same synchronized method on the same object, the virtual machine interpreter allows just one of them to execute this method. The second one is executed when the first finishes executing that method. The execution order of these threads depends on many parameters: time, processor speed, thread priority, etc. Hence, there is a need for introducing non-determinism in the resource pomset model in order to get a model that can be accommodated to JVML/CLDC.

In this paper, we provide an overview of the extension of the resource pomset model with non-determinism. Moreover, we provide a case study that shows how our semantic model can be embedded in the proof assistant Isabelle [28] in order to validate some optimizations of JVML/CLDC programs. The studied optimizations are: constant propagation and dead assignment elimination.

The rest of the paper is organized as follows. Section II is dedicated to the presentation of the semantic model. Sections III and IV are dedicated to the embedding of our semantic model in the theorem prover Isabelle and its use for the validation of some optimizations of JVML/CLDC programs. Finally, we provide some concluding remarks in Section V.

II. A Denotational Semantic Model for JVML/CLDC

In this section, our concern is to present the construction of the process space.

A. Dependence Maps

In order to capture the order between events that emerge while a JVML/CLDC program is executed, we designed a dependence maps space $\mathbb{M}$. An element of the space $\mathbb{M}$ is a map that associates, a pair (p_e,e), where p_e is a finite set of events representing the direct predecessors of the event e (element of the events set V), with another map that represents the successors of e. More formally, let $\rightarrow_{\omega_1}$ be the constructor of infinite maps in which an element can be associated with a infinite number of elements. The space $\mathbb{M}$ is defined as follows:

$$\mathbb{M} \quad = \quad \mathcal{P}_f(V) \times V \rightarrow_{\omega_1} \mathbb{M}$$

The existence proof of $\mathbb{M}$ is based on the transfinite recursive space construction technique, proposed by Di Gianantonio et al. [15]. The space $\mathbb{M}$ is endowed with a prefix ordering $\rhd$, which is defined as follows. Let $dom(M)$ denote the domain of the map M. Let M, $M' \in \mathbb{M}$ and $[\,]$ denote the empty map. We have:

1) $[\,] \rhd M$
2) $M \rhd M' \Leftrightarrow dom(M) \subseteq dom(M') \wedge \forall a \in dom(M).\ M(a) \rhd M'(a)$

In what follows, we present some utility functions that we use in the elaboration of our semantics.

Given two maps M and M', we write $M \dagger M'$ for the overwriting of the map M by the associations of the map M' i.e. the domain of $M \dagger M'$ is $dom(M) \cup dom(M')$ and we have:

$$(M \dagger M')(a) \quad = \quad \begin{cases} M'(a), & \text{if } a \in dom(M'); \\ M(a), & \text{Otherwise.} \end{cases}$$

We use a tuple projection function π_n, which selects the element at position n in a tuple. We also define a function φ that computes the elements of a map.

$$\begin{array}{lcl} \varphi & : & \mathbb{M} \rightarrow \mathcal{P}(\mathcal{P}_f(V) \times V) \text{ defined by} \\ \varphi(M) & = & \begin{cases} \emptyset, & \text{if } M = [\,]; \\ \bigcup_{a \in dom(M)} \{a\} \ \cup \ \varphi(M(a)), & \text{Otherwise.} \end{cases} \end{array}$$

Moreover, we define a function $\mathcal{D}$ that computes the dependence relation between the elements of a map as follows:

$$\begin{array}{lcl} \mathcal{D} & : & \mathcal{P}(\mathcal{P}_f(V) \times V) \rightarrow \mathcal{P}(V \times V) \text{ defined by} \\ \mathcal{D}(S) & = & \{(e, \pi_2(a)) |\ a \in S \wedge e \in \pi_1(a)\} \end{array}$$

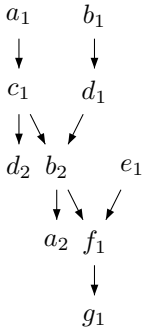

Fig. 4. Example of a graphical representation of a dependence map

We define $\mathbb{T} \subseteq \mathbb{M}$ to be the space of dependence maps such that $M \in \mathbb{T}$ if the reflexive transitive closure of $\mathcal{D}(\varphi(M))$ is a partial order relation. This condition states that we have no cycles in any element of $\mathbb{T}$. It is the first *healthiness condition* in our semantic model. Moreover, the domain of each map should contain just *initials* (an initial is an event having an empty set of predecessors) i.e. $\forall M \in \mathbb{T}.\ \forall e \in dom(M).\ \pi_1(e) = \emptyset$.

Example of a Dependence Map

Figure 4 outlines the graphical representation of the dependence map defined in Figure 3.

B. Labelled Dependence Maps

The space of labelled dependence maps $\mathbb{R}$ is defined as follows:

$$\mathbb{R} \quad = \quad \mathbb{T} \times (V \xrightarrow{\sim} \Sigma)$$

This space contains dependence maps with their labelling functions. A labelling function associates each event of a dependence map with an action (element of the actions set Σ). We compute the events set of an element of $\mathbb{R}$ by the function:

$$\begin{array}{lcl} \xi & : & \mathbb{R} \rightarrow \mathcal{P}(V) \text{ defined by} \\ \xi((M,\lambda)) & = & dom(\lambda) \end{array}$$

C. Deterministic Processes

Let $\mathcal{R}$ be a finite set of resources. We define the space of deterministic processes $\mathbb{C}$ as follows:

$$\mathbb{C} \quad = \quad \mathbb{R} \times \mathcal{P}(\mathcal{R})$$

A process in $\mathbb{C}$ is a pair where the first element is a pair composed of a dependence map with its labelling function and the second element refers to the resources needed by the process for its continuation. In this context, the dependence map represents what is already observed. It can evolve to any dependence map provided that any new executed action claims a subset of the resources specified in the continuation resources of the process as it will be shown when we specify the ordering over the space $\mathbb{C}$.

A process $p = (r_p, R_p) \in \mathbb{C}$ such that $r_p = (M_p, \lambda_p)$ is considered as finite if and only if r_p is finite i.e. it has a finite events set, which means that its dependence map M_p is finite.

We also suppose that we have a function $res : \Sigma \rightarrow \mathcal{P}(\mathcal{R})$, which associates an action with a resources set. Note that we assume that each action uses a non-empty set of resources. To lighten the notation, we use $\hat{a}$ to denote the resource set needed by an action a i.e. $\hat{a} = res(a)$. The definition of this function is extended to $\mathcal{P}(\Sigma)$ as follows:

$$\begin{array}{lcl} res & : & \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\mathcal{R}) \\ res(A) & = & \bigcup_{a \in A} res(a) \end{array}$$

Besides, the definition of res is extended to the space of labelled dependence maps $\mathbb{R}$ as follows:

$$\begin{array}{lcl} res & : & \mathbb{R} \rightarrow \mathcal{P}(\mathcal{R}) \text{ defined by} \\ res((M,\lambda)) & = & \bigcup_{e \in \xi((M,\lambda))} res(\lambda(e)) \end{array}$$

Finally, the definition of res is extended to the space deterministic processes $\mathbb{C}$ as follows:

$$\begin{array}{lcl} res & : & \mathbb{C} \rightarrow \mathcal{P}(\mathcal{R}) \text{ defined by} \\ res(p) & = & res(r_p) \cup R_p \end{array}$$

Informally, this function computes the resources needed by a deterministic process, which are those that are already used and those that are needed for the continuation i.e. for the future.

The space $\mathbb{C}$ is endowed with an ordering $\sqsubseteq_{\mathbb{C}}$, which is defined as follows. Let $p, q \in \mathbb{C}$, p is denoted by (r_p, R_p) where $r_p = (M_p, \lambda_p)$ and q by (r_q, R_q) where $r_q = (M_q, \lambda_q)$. We have:

$$\begin{array}{lcl} p \sqsubseteq_{\mathbb{C}} q & \Leftrightarrow & r_p \preceq r_q \wedge R_p \supseteq R_q \cup res(r_p^{-1} r_q) \quad \text{where} \\ r_p \preceq r_q & \Leftrightarrow & M_p \rhd M_q \wedge \lambda_p \subseteq \lambda_q \\ r_p^{-1} r_q & = & (M_q \backslash M_p, \lambda_q \setminus \lambda_p) \\ M' \backslash M & = & \begin{cases} M'[\emptyset / \pi_1(b) \mid b \in dom(M')], \text{ if } M = [\,]; \\ [a \mapsto M'(a) \mid a \in dom(M') \setminus dom(M)] \ \dagger \\ \dagger_{a \in dom(M) \cap dom(M')} M'(a) \backslash M(a), \text{ Otherwise.} \end{cases} \end{array}$$

Note that the operator †, which is already defined in Section II-A, is used in a symmetric way since there are no cycles and no auto concurrency (recall that any action uses a non-empty set of resources) in our coding of dependence maps.

The meaning of the ordering over $\mathbb{C}$ is that any process p can evolve to another process q with the constraint that only the resources specified in R_p are used. Note that if $R_p = \emptyset$, the process p is finite and maximal with respect to the ordering $\sqsubseteq_{\mathbb{C}}$. Otherwise, it is a non-terminated process. We also have the following constraint over the elements of $\mathbb{C}$: $\forall p \in \mathbb{C}.\ resinf(r_p) \subseteq R_p$ where:

$$\begin{array}{lcl} resinf & : & \mathbb{R} \rightarrow \mathcal{P}(\mathcal{R}) \text{ defined by} \\ resinf(x) & = & \bigcap \{res(r^{-1} x),\ r \text{ finite and } r \preceq x\} \end{array}$$

This condition means that all the needed resources for the continuation are specified in R_p. This is the second *healthiness condition* in our model.

In order to give meaning to recursion, we have to establish, in what follows, some results about our process space. Note that due to the lack of space, we do not provide the proofs of the established results.

Proposition 2.1: ($\mathbb{C}$,$\sqsubseteq_{\mathbb{C}}$) is algebraic. The compact elements of $\mathbb{C}$ are processes having a finite events set.

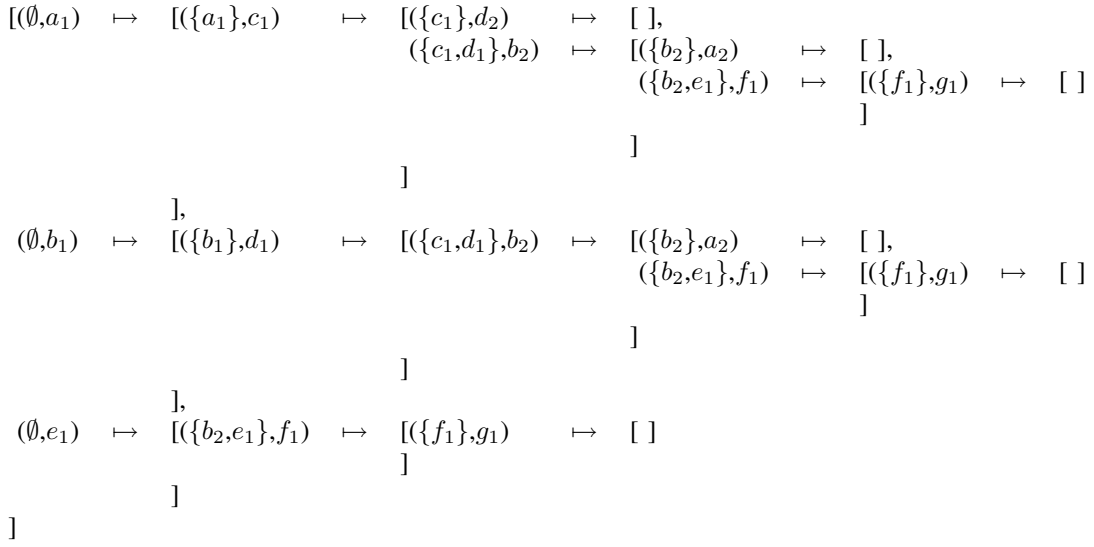

Fig. 3. Example of a Dependance Map

D. Space of Non-Deterministic Processes

In what follows, we present the space of non-deterministic processes. Let $\mathbb{D}$ be the space of non-deterministic processes. $\mathbb{D}$ is defined as the set of non-empty upper subsets of $\mathbb{C}$:

$$\begin{array}{lcl} \mathbb{D} & = & \{X \subseteq \mathbb{C} \mid X \neq \emptyset \wedge X = \uparrow X\} \text{ where} \\ \uparrow X & = & \bigcup_{x \in X} \uparrow x \text{ and } \uparrow x = \{y \in \mathbb{C} \mid x \sqsubseteq_{\mathbb{C}} y\} \end{array}$$

The space $\mathbb{D}$ is endowed with an ordering $\sqsubseteq_{\mathbb{D}}$, which is defined as follows:

$$\begin{array}{lcl} P \sqsubseteq_{\mathbb{D}} Q & \Leftrightarrow & P \supseteq Q \\ & \Leftrightarrow & \forall q \in Q.\ \exists p \in P.\ p \sqsubseteq_{\mathbb{C}} q \end{array}$$

Proposition 2.2: $(\mathbb{D},\sqsubseteq_{\mathbb{D}})$ is a local cpo.

The space $\mathbb{D}$ is meant to model unbounded non-determinism and it enjoys the algebraic property of local cpo that allows to define the semantic of recursive JVML/CLDC methods. Due to the pages number restriction, we omit details about the proofs and the semantic operators of our model. These details can be found in [46].

III. Embedding the Semantic Model in Isabelle

Isabelle[28] is a generic and interactive theorem prover that provides a logical framework in which theorems and proofs about programming languages and programs can be established. Few research initiatives have dealt with the embedding of Java in Isabelle/HOL[2] in order to prove compiler or optimization correctness. Hereafter, we review these research initiatives.

Nipkow et al. [31] formalized a subset of Java in Isabelle-/HOL. The primary objective of their work is to prove the type safety for this subset. In another project, Nipkow et al. [29] formalized a subset of JVML in Isabelle/HOL. The main goal of this formalization is to prove that the verifier is sound and also that the Java compiler is correct. Their strategy is based on an operational semantic style.

Strecker [39] proved the correctness of a Java compiler in an operational small step style. The compiler translates Micro-Java source code[3] into Micro-JVM bytecode. The correctness of the compiler is expressed as a commuting diagram as the one presented in Figure 1. The designed framework is based on an operational semantic style.

Glesner et al. [3] formalized the generation of code from Static Single Assignment (SSA) form [4] in Isabelle/HOL. They established the correctness of this generation process. The correctness proofs give also checkable correctness criteria characterizing correct compilation results obtained from different implementations of code generation algorithms. This work is also based on an operational semantic style.

As discussed before, Our strategy is denotational. The motivations and justifications behind this choice are already presented. In the sequel, we present the embedding of a denotational semantics for a subset of JVML/CLDC, which is based on the already presented semantic model. The studied subset excludes backward jumps. Moreover, we suppose that we are in the context of just one thread in this embedding work.

A. Embedding the Process Space

In what follows, we present an overview of the embedding of our process space. More precisely, we consider in this embedding two kinds of actions: `Lock` and `Unlock`. These actions are performed on objects which are considered as resources. The resource mapping between actions and resources is represented by a constant called `resMap`. A dependence map is embedded as a list of infinite branching graphs that capture the dependence between events, which are embedded as occurrences of actions. An infinite branching graph is represented by a datatype called `M`. Besides, the prefix relation between two dependence maps is established using a

[2]Isabelle/HOL [28] is the specialization of Isabelle for HOL, which is a Higher Order Logic

[3]A small subset of Java

[4]In such form, a variable is assigned just once in the program

function called F, which is defined co-inductively, while the deterministic process space is represented by a set called C. The embedding of the main features of our model is as follows:

```
typedecl classType
datatype fields = "int list"
types    lockCounter = "int"
types    object = "classType × fields × lockCounter"
types    resources = "object set"
datatype action =  Lock object |
                   Unlock object
types    event = "action × nat"
consts   resMap :: "action ⇒ resources"
types    PEvent = "event set × event"
datatype M = Bot | Node PEvent "nat ⇒ M"
types    lab = "(event × action) set"
types    process = "(M list × lab) × resources"

consts dom :: "M ⇒ 'a set" defs
dom_def: "dom M == (case M of
                         Bot ⇒ {}
                         | (Node a F) ⇒
                           {x. ∃ y. y = F x})"


(*Prefix relation*)
consts F :: "(M × M) set ⇒ (M × M) set"
coinductive "F(Q)"
  intros
    BOT_I:
      "(Bot,T) ∈ F(Q)"
    TRACE_I:
      "[|a = b; M = Node a G ; N = Node b H;
         ∀ i ∈ dom(M).∃ j ∈ dom(N).
         (G i,H j) ∈ Q |] ⇒ (M,N) ∈ F(Q)"

consts prefixRel :: "M list × M list ⇒ bool"
defs   prefixRel_def:
"prefixRel R S == ∀ x ∈ set R. ∃ y ∈ set S.
                  ∃ H. (x,y) ∈ F(H)"

(*Healthiness condition*)
consts  por :: "'a list ⇒ bool"
primrec "por [] = True"
        "por (l#ls) = (if (antisym l) then (por ls)
                                      else False)"

constdefs C :: "(process) set"
"C == {x. por (fst (fst x))}"
```

B. JVML/CLDC Subset Syntax

Hereafter, we present the JVML/CLDC subset which we consider in this paper.

```
types    index = nat
types    val = int
datatype bytecode =

ILOAD    index | ISTORE   index  | ICONST   val    |
BIPUSH   val   | DUP             | IADD            |
IMUL           | IFEQ     index  | IFICMPEQ index  |
IFICMPNE index | IFICMPLE index  | GOTOF    index  |
ALOAD    index | ASTORE   index  | IRETURN         |
MONITORENTER   | MONITOREXIT
```

The informal semantics of each bytecode can be found in [20].

C. Bytecodes Semantics

The semantics of a bytecode is specified by a semantic function called exec. This function computes a pair composed of a denotation and a continuation. The denotation is a tuple composed of a process, a stack, a local variable table and the returned value (if any). The continuation represents the remaining code of the program to be executed. Hereafter, we present the embedding of some structures that are useful for the elaboration of the semantics.

```
types stack = "val list"
types locvars = "val list"
types heap ="int ⇒ object option"
types prog = "bytecode list"
datatype result = NoValue | Value "int"
types denotation = "process × heap × stack × locvars ×
                    result"
types cont = "prog"
```

In what follows, we provide an overview of the embedding of the semantic function exec.

```
consts exec :: "bytecode ⇒ denotation ⇒ cont ⇒ prog ⇒
                denotation × cont"
primrec
"exec (ILOAD  ind)  d  c p =
(let (pr,he,sk,lv,rs) = d
 in  ((seqcomp (pr,(([],{}),{})),he,(lv ! ind) # sk,lv,
       rs),c))"

"exec (IFEQ ind)   d  c p =
(let (pr,he,sk,lv,rs) = d;
      v = hd sk;
      v1 = (length p) - (length c);
      disp = (ind - v1)
 in   if (v ≠ 0)
      then ((seqcomp (pr,(([],{}),{})),he,tl sk,lv,
             rs),c)
      else ((seqcomp (pr,(([],{}),{})),he,tl sk,lv,
             rs),drop disp c))"

"exec MONITORENTER  d  c p = (let (pr,he,sk,lv,rs) = d;
      v = hd sk
 in  (case he(v) of
           None ⇒ (d,[]) |
           Some (cl,fs,lc) ⇒
                (let lc1 = lc + 1;
                     he1 = he(v |-> (cl,fs,lc1));
                     pr1 =(([Node ({},(Lock (he v),1))
                            (λ i. Bot)],((Lock (he v),
                            1),Lock (he v))),{})
                 in  ((seqcomp (pr,pr1),he1, tl sk,lv,
                       rs),c))))"

"exec MONITOREXIT   d  c p =
(let (pr,he,sk,lv,rs) = d ;
      v = hd sk
 in  (case he(v) of
           None ⇒(d,[]) |
           Some (cl,fs,lc) ⇒
                (let lc1 = lc - 1;
                     he1 = he(v |-> (cl,fs,lc1));
                     pr1 =(([Node ({},(Unlock (he v),1))
                            (λ i. Bot)],((Unlock (he v),
                            1),Unlock (he v))),{})
                 in  ((seqcomp (pr,pr1),he1,tl sk,lv,
                       rs),c))))"
```

The function seqcomp performs a sequential composition of two processes. This is the embedding of the strict sequential composition semantic operator of our semantic model[46]. Note also that the semantics of MONITORENTER and MONITOREXIT are more complicated than what we presented. The semantics of these bytecodes are provided just to show the emergence of observable actions in the denotation and the use of our semantic model to prove JVML/CLDC programs

equivalence.

D. JVML/CLDC Program Semantics

Before providing the semantics of a JVML/CLDC program, we present two lemmas that prove that the continuation argument is decreasing in size during execution. In these lemmas, induction, auto tactics and simplification are needed.

```
declare Let_def[simp] option.split[split]

lemma listsize_eq:
"(length (snd (exec ins d bl p)) < Suc (length bl))
  =
(length (snd (exec ins d bl p)) ≤ length bl)"
apply(auto)
done

lemma listsize [simp]:
"length (snd (exec ins d bl p)) ≤ length bl"
apply(induct_tac ins)
apply(simp_all)
apply(induct_tac[!] d)
apply(auto)
done
```

The semantics of a JVML/CLDC program is specified by a semantic interpretation function called `sem`, which is defined recursively and which uses the function `exec`. Its termination depends on the size of the first argument that refers to the continuation and which should decrease. To help the prover doing the full proof, we give a "hint" that suggests the use of the previous lemmas as a simplification rule. In the sequel, we present the embedding of the function `sem`.

```
consts  sem :: "prog × denotation × prog ⇒
                denotation"
recdef sem "measure (λ(xs,a). length xs)"

"sem ([],d,p)  = d"
"sem ((b#bl),d,p) =
(sem (snd (exec b d bl p), fst (exec b d bl p),p))"
(hints recdef_simp: listsize_eq)
```

IV. A Case Study about Optimizations Validation

In this section, we present some specific optimizations that can be performed on JVML/CLDC programs. To make the presentation clear, we provide the Java source of each JVML/CLDC program, which is the target of the optimization. Moreover, we suppose that for each Java source optimization there is a similar transformation of its compilation output i.e. this optimization can be performed on the compiled file, which is a JVML/CLDC program. Note that the representation of JVML/CLDC programs in Isabelle/HOL is the result of an abstraction at the instruction level.

A. Constant Propagation

A constant propagation transformation is performed if one variable is assigned to a constant value. The variable is replaced by its value in order to avoid more computations. The following Java method contains a code on which compilers can perform constant propagation.

```
public int foo() {
    int x,y;
    x = 3;
    y = x + 4;
    return y;}
```

The definition in Isabelle/HOL of the JVML/CLDC code, which is the compilation output of the already presented Java code, is the following:

```
constdefs orprog1 :: "prog"
"orprog1==[ICONST 3, ISTORE 1, ILOAD 1, ICONST 4,
           IADD, ISTORE 2, ILOAD 2, IRETURN]"
```

In the Java source, we see that the variable "x" can be replaced by 3 in the expression "x + 4". The optimized Java code is the following:

```
public int foo() {
    int x,y;
    x = 3;
    y = 7;
    return y;}
```

The associated compiled code is the following:

```
constdefs opprog1 :: "prog"
"opprog1==[ICONST 3, ISTORE 1, BIPUSH 7, ISTORE 2,
           ILOAD 2, IRETURN]"
```

As mentioned previously, we assume that we have a transformation between orprog1 and opprog1, which is a constant propagation optimization. Proving that this constant propagation optimization is correct means that orprog1 and opprog1 are associated with the same denotation. This is proved later when we speak about equivalence relations in Section IV-C.

B. Dead Assignment Elimination

A dead assignment elimination is an optimization targeting the removal of dead variables. These variables are never used after their assignment. The following Java code contains a dead assignment for the variable "x" in the statement "x = 3".

```
public int foo() {
    int x,y;
    y = 0;
    x = 3;
    y = y + 2;
    return y;}
```

The associated compiled code, as presented in Isabelle, is the following:

```
constdefs orprog2 :: "prog"
"orprog2==[ICONST 0, ISTORE 2, ICONST 3, ISTORE 1,
           ILOAD 2, ICONST 2, IADD, ISTORE 2,
           ILOAD 2, IRETURN]"
```

The removal of this statement produces the following code:

```
public int foo() {
    int x,y;
    y = 0;
    y = y + 2;
    return y;}
```

The associated compiled code is the following:

```
constdefs opprog2 :: "prog"
"opprog2==[ICONST 0, ISTORE 2, ILOAD 2, ICONST 2,
           IADD, ISTORE 2, ILOAD 2, IRETURN]"
```

C. Equivalence Relations

In what follows, we provide two possible relations that can be used to establish the semantic equivalence between JVML/CLDC programs:

- *Strong* equivalence: the original and optimized programs are equivalent if they are associated with the same denotation i.e. do the same lock/unlock actions on the same objects and the two associated heaps, stacks, local variable tables and returned values are the same after the execution of these programs. Here, we consider that we observe just actions that lock or unlock objects but it is worth to mention that other abstractions can be considered. In fact, we can observe exceptions or communications. This means that two programs are equivalent if they lock and unlock the same objects, throw the same exceptions, do the same communications, have the same stack and local variables table and return the same result. As defined, this equivalence relation is too restrictive. In fact, two equivalent programs can return the same value without doing the same treatments on the stack or having the same local variable table. This is exemplified by the optimization of the specified program in Section IV-B. Henceforth, we provide another definition of *weak* equivalence.
- *Weak* equivalence: the original and optimized program are equivalent if they return the same value after their execution.

Hereafter, we suppose that the dependence map, the stack and the local variables table are empty for any program before its execution:

```
constdefs d:: "denotation"
"d ==((([],{}),{}),map_of [(0,this)],[0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0],NoValue)"
```

Note that the variable "this" refers to the instance on which a given method was called. We suppose that this object is stored at the memory address zero to simplify the presentation.

The strong equivalence is defined in Isabelle/HOL as follows:

```
constdefs equivalent :: "prog ⇒ prog ⇒ bool"
"equivalent p1 p2 ==
(sem (p1,d,p1)
 =
 sem (p2,d,p2))"
```

The weak equivalence is defined in Isabelle/HOL as follows:

```
constdefs weakequivalent :: "prog ⇒ prog ⇒ bool"
"weakequivalent p1 p2 ==
(snd (snd (snd (snd  (sem (p1,d,p1)))))
 =
 snd (snd (snd (snd (sem (p2,d,p2))))))"
```

The programs orprog2 and opprog2 are proved, using Isabelle, to be weakly but not strongly equivalent. In fact, the semantics of orprog2 is the following:

```
((([],{}),{}),map_of [(0,this)],[0,0,0,0,0,0,0],
 [0,3,2,0,0,0,0],Value 2)
```

While the semantics of oprpog2 is the following:

```
((([],{}),{}),map_of [(0,this)],[0,0,0,0,0,0,0],
 [0,0,2,0,0,0,0],Value 2)
```

The programs orprog1 and opprog1 have the same semantics:

```
((([],{}),{}),map_of [(0,this)],[0,0,0,0,0,0,0],
 [0,3,7,0,0,0,0],Value 7)
```

orprog1 and opprog1 are proved to be strongly equivalent since they are associated with the same denotation.

The following example shows the emergence of observable actions (lock/unlock) in the semantics of the original and optimized JVML/CLDC programs: orprog3 and opprog3. Hereafter, we provide the specification in Isabelle of these two programs.

```
constdefs orprog3 :: "prog"
"orprog3  == [ALOAD 0, DUP, ASTORE 3, MONITORENTER,
              ICONST 3, ISTORE 1, ILOAD 1, ICONST 4,
              IADD, ISTORE 2, ILOAD 2, ALOAD 3,
              MONITOREXIT, IRETURN]"

constdefs opprog3 :: "prog"
"opprog3 == [ALOAD 0, DUP, ASTORE 3, MONITORENTER,
             ICONST 3, ISTORE 1, BIPUSH 7, ISTORE 2,
             ILOAD  2, ALOAD 3, MONITOREXIT, IRETURN]"
```

The program opprog3 is the transformation of orprog3 by constant propagation. These two programs are proved to be strongly equivalent. In fact, they are associated with the same following denotation:

```
((([Node({},(Lock this,1))(fun_upd (λ i.Bot) 0
    Node ({(Lock this,1)},(Unlock this,1))(λ i.Bot))],
    {((Lock this,1),Lock this),((Unlock this,1),
    Unlock this)}),{}), map_of [(0,this)],
    [0,0,0,0,0,0,0],[0,3,7,6,0,0,0],Value 7)
```

V. Conclusion

In this paper, we presented a new semantic model for true concurrency with unbounded non-determinism. The model is denotational and rests on an extension of the resource pomsets semantics of Gastin and Mislove. We presented the construction of the process space and exhibited its algebraic properties. In addition, we provided an embedding of the main features of this semantic model in the theorem prover Isabelle together with a case study that shows how this model can be used in order to validate some optimizations of JVML/CLDC programs. We defined also some equivalence relations and discussed the impact of their definitions on the assessment of correctness of optimizations. We think that an equivalence relation should be defined with respect to the specificities of the studied optimization. These specificities allow to know the required abstractions to be performed in order to prove that original and optimized programs have the same semantics. Currently, we are working on the full embedding of our semantic model in Isabelle in order to prove the correctness of our dynamic compilation technique and of other fancy optimizations.

REFERENCES

[1] S. Abramsky and A. Jung. Domain Theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

[2] M. Bednarczyk. *Categories of Asynchronous Systems*. PhD thesis, University of Sussex, 1988.

[3] J. Blech, S. Glesner, J. Leitner, and S. Mlling. Optimizing Code Generation from SSA Form: A Comparison Between Two Formal Correctness Proofs in Isabelle/HOL. In *Proceedings of the Workshop on Compiler Optimization meets Compiler Verification (COCV'05)*, Edinburgh, Scotland, April 2005. Elsevier.

[4] D. Bolignano and M. Debbabi. A Semantic Theory for Concurrent ML. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS'94)*, volume 789 of *Lecture Notes in Computer Science*, pages 766–785, Sendai, Japan, April 1994. Springer-Verlag.

[5] H. Boom. A Weaker Precondition for Loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4:668–677, October 1982.

[6] S. Brookes, C. Hoare, and A. Roscoe. An Improved Failures Model for Communicating Processes. In *Seminar on Concurrency, Carnegie-Mellon University*, volume 197, pages 281–305. Springer-Verlag, 1985.

[7] S. Brooks, C. Hoare, and A. Roscoe. A Theory for Communicating Sequential Processes. *JACM*, 31(3):560–599, January 1984.

[8] M. Debbabi, A. Gherbi, L. Ketari, C. Talhi, N. Tawbi, H. Yahyaoui, and S. Zhioua. A Dynamic Compiler for Embedded Java Virtual Machines. In *Proceedings of the 3^{rd} ACM Conference on the Principles and Practice of Programming in Java (ACM PPPJ'04)*, pages 100–107, Las Vegas, USA, June 2004. ACM Press.

[9] M. Debbabi, A. Gherbi, L. Ketari, C. Talhi, N. Tawbi, H. Yahyaoui, and S. Zhioua. A Synergy between Efficient Interpretation and Fast Selective Dynamic Compilation for the Acceleration of Embedded Java Virtual Machines. In *Proceedings of the 3^{rd} ACM Conference on the Principles and Practice of Programming in Java (ACM PPPJ'04)*, pages 108–115, Las Vegas, USA, June 2004. ACM Press.

[10] M. Debbabi, A. Gherbi, A. Mourad, and H. Yahyaoui. A Selective Dynamic Compiler for Embedded Java Virtual Machines Targeting ARM Processors. *Journal of Science of Computer Programming*, 59(1–2):38–63, January 2006.

[11] P. Gastin and M. Mislove. A Truly Concurrent Semantics for a Simple Parallel Programming Language. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'99)*, volume 1683 of *Lecture Notes in Computer Science*, pages 515–529, Madrid, Spain, September 1999. Springer-Verlag.

[12] P. Gastin and M. Mislove. A Truly Concurrent Semantics for a Process Algebra using Resource Pomsets. *Theoretical Computer Science (TCS)*, 281(1–2):369–421, June 2002.

[13] P. Gastin and A. Petit. *The Book of Traces*, chapter Infinite traces, pages 393–486. World Scientific, 1995.

[14] P. Gastin and D. Teodosiu. Resource Traces: A Domain for Processes sharing Exclusive Resources. *Theoretical Computer Science (TCS)*, 278:195–221, May 2002.

[15] P. Di Gianantonio, F. Honsell, and G. Plotkin. Uncountable Limits and the Lambda Calculus. *Nordic Journal of Computing*, 2(2):126–145, Spring 1995.

[16] M. Hennessy. Acceptance Trees. *Journal of the ACM (JACM)*, 32(4):896–928, January 1985.

[17] M. Kwiatkowska and G. Norman. Metric Denotational Semantics for PEPA. In *Proceedings of the 4^{th} Workshop on Process Algebras and Performance Modelling (PAPM'96)*, pages 120–138, Torino, Italy, July 1996. CLUT.

[18] D. Lacey, N. Jones, E. Wyk, and C. Frederiksen. Proving Correctness of Compiler Optimizations by Temporal Logic. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'02)*, pages 283–294, Portland, OR, USA, January 2002.

[19] S. Lerner, T. Millstein, and C. Chambers. Automatically Proving the Correctness of Compiler Optimizations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 220–231, San Diego, California, USA, June 2003.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Version*. Addison-Wesley, CA, USA, 1999.

[21] Sun Microsystems. Connected, Limited Device Configuration. Specification Version 1.0, Java 2 Platform Micro Edition, May 2000. White paper.

[22] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, Secaucus, NJ, USA, 1982.

[23] R. Milner. *Concurrency and Communication*. Prentice-Hall, 1989.

[24] M. Mislove, A. Roscoe, and S. Schneider. Fixed Points without Completeness. *Theoretical Computer Science (TCS)*, 138(2):273–314, February 1995.

[25] M. Mislove. Denotational Models for Unbounded Nondeterminism. *Electronic Notes in Theoretical Computer Science*, 1, April 1995.

[26] F. Morris. Advice on Structuring Compilers and Proving them Correct. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'73)*, pages 144–152, Boston, Massachusetts, USA, October 1973. ACM Press.

[27] A. Mycroft, P. Degano, and C. Priami. Complexity as a Basis for Comparing Semantic Models of Concurrency. In *Proceedings of the Asian Computing Science Conference (ACSC'95)*, volume 1023, pages 141–155, Pathumthani, Tailand, December 1995. Springer-Verlag.

[28] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[29] T. Nipkow, D. Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In *Proceedings of the Summer School on Foundations of Secure Computation*, volume 175 of *NATO Science Series F: Computer and Systems Sciences*, pages 117–144. IOS Press, 2000.

[30] M. Nunez, D. de Frutos, and L. Llana. Acceptance Trees for Probabilistic Processes. In *Proceedings of the 6^{th} International Conference on Concurrency Theory (CONCUR'95)*, pages 249–263, Philadelphia, PA, August 1995.

[31] D. Oheimb and T. Nipkow. Machine-Checking the Java Specification: Proving Type Safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523, pages 119–156. Springer-Verlag, June 1999.

[32] G. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[33] G. Plotkin. An Operational Semantics for CSP. In *Proceedings of the IFIP TC 2-Working Conference on Formal Description of Programming Concepts*, pages 199–225, Amsterdam, Netherland, 1983.

[34] Vaughan R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, May 1986.

[35] A. Roscoe and G. Barrett. Unbounded Nondeterminism in CSP. In *Proceedings of the 5^{th} International Conference on Mathematical Foundations of Programming Semantics (MFPS'89)*, volume 442, pages 160–193, Tulane University, New Orleans, Louisiana, USA, March 1989. Springer-Verlag.

[36] V. Sassone, M. Nielsen, and G. Winskel. Models for Concurrency: Towards a Classification. *Theoretical Computer Science (TCS)*, 170(1–2):297–348, January 1996.

[37] I. Siveroni. *Correctness of Analysis-based Program Transformations of Functional Programming Languages*. PhD thesis, College of Computer Science, Norhteastern University, 2002.

[38] K. Stephenson. Compiler Correctness using Algebraic Operational Semantics. Technical Report CSR 1-97, University of Wales Swansea, 1997.

[39] M. Strecker. Formal Verification of a Java Compiler in Isabelle. In *Proceedings of the Conference on Automated Deduction (CADE'02)*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77, Copenhagen, Denmark, July 2002. Springer-Verlag.

[40] J. Thatcher, E. Wagner, and J. Wright. More Advice on Structuring Compilers and Proving them Correct. *Theoretical Computer Science (TCS)*, 15:223–249, 1981.

[41] M. Walicki and S. Meldal. Algebraic Approaches to Nondeterminism-An Overview. *ACM Computing Surveys*, 29(1):30–81, 1997.

[42] M. Wand. Compiler Correctness for Parallel Languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 120–134, La Jolla, CA, USA, June 1995. ACM Press.

[43] G. Winskel and M. Nielsen. *Handbook of Logic in Computer Science*, volume 4, chapter Models for Concurrency. Clarendon Press, 1995.

[44] G. Winskel. *Events in Computation*. PhD thesis, University of Edinburgh, 1980.

[45] G. Winskel. Synchronization Trees. *Theoretical Computer Science (TCS)*, 34:32–82, 1984.

[46] H. Yahyaoui. *Acceleration and Semantic Foundations of Embedded Java Virtual Machines*. PhD thesis, Departement of Computer Science and Software Engineering, Laval University, 2006.