Definition and Formal Metatheory of the Machine Independent Language

Karl Palmskog, Xiaomo Yao, Ning Dong, Roberto Guanciale, and Mads Dam

Abstract

We define the Machine Independent Language (MIL), which captures microarchitectural features such as out-of-order execution, and describe its metatheory on out-of-order and in-order execution of microinstructions. All presented definitions and results have been formalized and checked using the HOL4 theorem prover.

1 Introduction

The Machine Independent Language (MIL) captures microarchitectural features such as out-of-order execution. MIL can be used as a form of abstract microcode language, e.g., as a target language for translating Instruction Set Architecture (ISA) instructions, and for reasoning about microarchitectural features that may cause unwanted information flows, e.g., side channels leaking secret information.

In this supplementary material, we define MIL and its Out-of-Order (OoO), and In-Order (IO) semantics. We then describe the metatheory of MIL. All definitions and results have been formalized and checked using the HOL4 theorem prover [1]. The document renders some of the formal, machine-checked HOL4 definitions and metatheory for MIL into rigorous but informal form, using a mix of mathematical notation and English. It should thus be seen as a guide to the HOL4 encoding rather than as a normative definition; the HOL4 theory is normative.

The document is structured as follows:

- Section 2 describes the syntax of MIL, including abstract syntax and some example programs using a basic surface syntax.
- Section 3 defines the OoO and IO dynamic semantics of MIL as transition step relations via rules.
- Section 4 describes the metatheory of MIL, including the *memory consistency* of the OoO and IO semantics.

2 Syntax

In this section, we define the abstract and concrete syntax of MIL. The syntax of expressions is open-ended; we include the word expressions from the BIR language.

2.1 Abstract Syntax

Definition (Names)**.** Variable t ranges over *names*, which are elements with decidable equality that are totally ordered by the relation $<$. Variables N, C, F, and P range over sets of names. $\mathit{max}\,(N)$ is the partial function that returns the greatest name in N according to $<$, when this name exists.

Definition (Values)**.** Variable v ranges over *values*, which are elements with decidable equality drawn from a finite domain. The domain includes the special values *true*, *false*, and 0, where *true* $\neq$ *false*. Values are also ranged over by the variables a and r when they are used as *memory addresses* and *register identifiers*, respectively.

τ	::=			resource
	|	$\mathcal{PC}$		program counter
	|	$\mathcal{R}$		register
	|	$\mathcal{M}$		memory

$e,\ c$	::=			expression
	|	v		value
	|	t		name
	|	$e_1 \& e_2$		and
	|	$e_1 \vert e_2$		or
	|	$e_1 \hat{} e_2$		exclusive or
	|	$e_1 + e_2$		addition
	|	$e_1 - e_2$		subtraction
	|	$e_1 * e_2$		multiplication
	|	e_1 / e_2		division
	|	$e_1 :/ e_2$		signed division
	|	$e_1 \mathbf{mod}\, e_2$		modulo
	|	$e_1 \mathbf{smod}\, e_2$		signed modulo
	|	$e_1 \ll e_2$		left shift
	|	$e_1 \gg e_2$		right shift
	|	$e :\gg e_2$		signed right shift
	|	$e_1 == e_2$		equal
	|	$e_1 != e_2$		not equal
	|	$e_1 < e_2$		less than
	|	$e_1 :< e_2$		signed less than
	|	$e_1 \leq e_2$		less or equal to
	|	$e_1 :\leq e_2$		signed less or equal to
	|	$\sim e$		complement
	|	$!e$		logical negation
	|	(e)	S	parenthesized expression

o	::=			operation
	|	e		internal
	|	$ld\, \tau\, t$		load
	|	$st\, \tau\, t\, t'$		store

ι	::=			microinstruction
	|	$t \leftarrow c ? o$		guarded assignment

Definition (Programs)**.** Variable I ranges over sets of microinstructions. A set of microinstructions is called a *program*.

2.2 Surface Syntax

MIL programs defined by their abstract syntax are difficult to read and comprehend. To address this problem, we define, by the grammar below, a simple C-like surface syntax for MIL that is easier to read than the abstract syntax.

```
Program ::= InstructionList
InstructionList ::= '' | InstructionList Instruction
Instruction ::= NAME ':=' [ Expression '?' ] Operation ';'
Operation ::= Expression | LOAD '(' Resource ',' NAME ')' |
 STORE '(' Resource ',' NAME ',' NAME ')'
Expression ::= NAME | VALUE | TRUE | FALSE | '!' Expression | ...
Resource ::= PC | REG | MEM
```

Remark. In a ground MIL program, all names (`NAME` tokens) are name literals. To enable composition of several MIL programs, names can also be variables, effectively defining a MIL program *template*. To properly handle dependencies between microinstructions, translation of a program template to a ground program can, e.g., guarantee that the syntactic instruction order is preserved when substituting name variables for name literals.

2.3 Example Programs

This section shows the concrete and abstract syntax for several MIL programs, and explains informally their meaning.

Assignment Program. The below program increments by one the value stored in the register r_1. In other words, it performs the assignment $r_1 = r_1 + 1$.

```
t0 := true ? 0 ;                        // zeroed name for PC loads and stores
t1 := true ? r1 ;                       // register identifier
t2 := true ? load(REG, t1) ;            // load contents of of register
t3 := true ? t2 + 1 ;                   // incremented value
t4 := true ? store(REG, t1, t3) ;       // store update
t5 := true ? load(PC, t0) ;             // load current PC
t6 := true ? t5 + 4 ;                   // increment PC
t7 := true ? store(PC, t0, t6) ;        // store incremented PC
```

$$\left\{ \begin{array}{l} t_0 \leftarrow \mathit{true}?0,\ t_1 \leftarrow \mathit{true}?r_1,\ t_2 \leftarrow \mathit{true}?\mathit{ld}\,\mathcal{R}\,t_1,\ t_3 \leftarrow \mathit{true}?t_2+1,\ t_4 \leftarrow \mathit{true}?\mathit{st}\,\mathcal{R}\,t_1\,t_3, \\ t_5 \leftarrow \mathit{true}?\mathit{ld}\,\mathcal{PC}\,t_0,\ t_6 \leftarrow \mathit{true}?t_5+4,\ t_7 \leftarrow \mathit{true}?\mathit{st}\,\mathcal{PC}\,t_0\,t_6 \end{array} \right\}$$

Conditional Program. The program below performs a conditional branch, i.e., it executes the high-level instruction *beq a*.

```
t00 := true ? 0 ;                       // zeroed name for PC loads and stores
t11 := true ? r ;                       // register identifier
t12 := true ? load(REG, t11) ;          // load contents of register
t21 := true ? t12 == 1 ;                // check if contents is equal to 1
t31 := true ? load(PC, t00) ;           // load current PC
t41 := true ? a ;                       // memory address
t42 := t21 ? store(PC, t00, t41) ;      // store address to PC if equal
t51 := true ? t31 + 4 ;                 // increment PC
t52 := !t21 ? store(PC, t00, t51) ;     // store incremented PC
```

$$\left\{ \begin{array}{l} t_{00} \leftarrow \mathit{true}?0,\ t_{11} \leftarrow \mathit{true}?r,\ t_{12} \leftarrow \mathit{true}?\mathit{ld}\,\mathcal{R}\,t_{11},\ t_{21} \leftarrow \mathit{true}?t_{12} == 1,\ t_{31} \leftarrow \mathit{true}?\mathit{ld}\,\mathcal{PC}\,t_{00}, \\ t_{41} \leftarrow \mathit{true}?a,\ t_{42} \leftarrow t_{21}?\mathit{st}\,\mathcal{PC}\,t_{00}\,t_{41},\ t_{51} \leftarrow \mathit{true}?t_{31}+4,\ t_{52} \leftarrow !t_{21}?\mathit{st}\,\mathcal{PC}\,t_{00}\,t_{51} \end{array} \right\}$$

Move Program. This program transfers a value from the register r_1 to the register r_5, i.e., it executes the high-level instruction $mov\, r_1, r_5$.

```
t00 := true ? 0 ;
t01 := true ? a0 ;
t02 := true ? store(PC, t00, t01) ;
t03 := true ? r5 ;
t04 := true ? v5 ;
t05 := true ? store(REG, t03, t04) ;
t10 := true ? r1 ;
t11 := true ? r5 ;
t12 := true ? load(REG, t11) ;
t13 := true ? store(REG, t10, t12) ;
t14 := true ? load(PC, t00) ;
t15 := true ? t14 + 4 ;
t16 := true ? store(PC, t00, t15) ;
```

$$\left\{\begin{array}{l} t_{00} \leftarrow true?0,\; t_{01} \leftarrow true?a_0,\; t_{02} \leftarrow true?st\,\mathcal{PC}\,t_{00}\,t_{01},\; t_{03} \leftarrow true?r_5,\; t_{04} \leftarrow true?v_5, \\ t_{05} \leftarrow true?st\,\mathcal{R}\,t_{03}\,t_{04},\; t_{10} \leftarrow true?r_1,\; t_{11} \leftarrow true?r_5,\; t_{12} \leftarrow true?ld\,\mathcal{R}\,t_{11}, \\ t_{13} \leftarrow true?st\,\mathcal{R}\,t_{10}\,t_{12},\; t_{14} \leftarrow true?ld\,\mathcal{PC}\,t_{00},\; t_{15} \leftarrow true?t_{14}+4,\; t_{16} \leftarrow true?st\,\mathcal{PC}\,t_{00}\,t_{15} \end{array}\right\}$$

3 Semantics

3.1 Runtime Syntax

Definition (Stores)**.** Variable s ranges over *stores*, which are finite maps from names to values.

- $dom\,(s)$ is the set of names for which the store is defined.
- $s(t)\downarrow$ ($s(t)\uparrow$) is true precisely when the store s is defined for t (resp. undefined for t).
- $s + [t \mapsto v]$ is the store that maps t to v but otherwise behaves as s.
- $s(t) = s'(t')$ is true precisely when either (i) $s(t)\uparrow$ and $s'(t')\uparrow$, or (ii) there is some v such that $s(t) = v$ and $s'(t') = v$.

σ	$::=$		state
	$\mid$	(I, s, C, F)	
α	$::=$		action
	$\mid$	EXE	execute
	$\mid$	CMT (a, v)	commit
	$\mid$	FTC (I)	fetch
obs	$::=$		observation
	$\mid$	ϵ	internal unobservable operation
	$\mid$	*dl v*	load from data cache
	$\mid$	*ds v*	store into data cache
	$\mid$	*il v*	load from instructions
l	$::=$		label
	$\mid$	(obs, α, t)	

3.2 Auxiliary Definitions

Definition (Names in Expressions and Operations)**.** For an expression e, its set of names $n\,(e)$ is defined recursively in the obvious way, e.g., $n\,(v) = \emptyset$, $n\,(t) = \{t\}$, $n\,(!e) = n\,(e)$, and $n\,(e_1 + e_2) = n\,(e_1) \cup n\,(e_2)$. For an operation o, its set of names $n\,(o)$ is defined similarly.

Definition (Names in Microinstructions)**.** A microinstruction ι has a *bound name*, written $\mathit{bn}\,(\iota)$, and a set of *free names*, written $\mathit{fn}\,(\iota)$; the set of all names in ι is written $n\,(\iota)$. The set of all bound names of microinstructions in a program I is written $\mathit{bn}\,(I)$.

- $\mathit{bn}\,(t \leftarrow c?o) = t$
- $\mathit{fn}\,(t \leftarrow c?o) = n\,(c) \cup n\,(o)$
- $n\,(\iota) = \{\mathit{bn}\,(\iota)\} \cup \mathit{fn}\,(\iota)$
- $\mathit{bn}\,(I) = \{\mathit{bn}\,(\iota) \,|\, \iota \in I\}$

Definition (Semantics of Expressions)**.** The semantics of an expression e is given by a partial function taking the expression and a store s as input, and returning a value. We write $[e]s\downarrow$ when the output value is defined, i.e., $[e]s = v$ for some value v, and $[e]s\uparrow$ otherwise. We write $[e]s = [e']s'$ when either (i) $[e]s\uparrow$ and $[e']s'\uparrow$, or (ii) there is some v such that $[e]s = v$ and $[e']s' = v$.

We do not designate a canonical function that defines the semantics of expressions, since it is application dependent. However, we require such a function to have the following properties:

1. For all e and s, $[e]s\downarrow$ if and only if $n\,(e) \subseteq \mathit{dom}\,(s)$.
2. For all e, s, and s', if $s(t) = s'(t)$ holds for all $t \in n\,(e)$, then $[e]s = [e]s'$.
3. For all v and s, $[v]s = v$.

Definition (True Guard Condition)**.** Given a store s, an expression c evaluates to a *true guard condition*, written $[c]s$, iff there exists v such that $[c]s = v$ and $v \neq \mathit{false}$. In particular, $[c]s$ holds if $[c]s = \mathit{true}$.

Definition (Microinstruction Translation)**.** The function *translate* takes as input a value v and a name t, and returns a set of microinstructions. The function has the following properties:

1. For all v and t, $\mathit{translate}\,(v, t)$ is a finite set.
   - MIL semantics require the maximum instruction name to be well defined.
2. For all $\iota \in \mathit{translate}\,(v, t)$ and $\iota' \in \mathit{translate}\,(v, t)$, if $\mathit{bn}\,(\iota) = \mathit{bn}\,(\iota')$ then $\iota = \iota'$.
   - Bound names of instructions must be unique.
3. For all $\iota \in \mathit{translate}\,(v, t)$ and $t' \in \mathit{fn}\,(\iota)$, $t' < \mathit{bn}\,(\iota)$.
   - The graph of instruction dependencies must be a acyclic.
4. For all $\iota \in \mathit{translate}\,(v, t)$ and $t' \in n\,(\iota)$, $t < t'$.

  - Instructions produced by *translate* must not contain t or names less than t.
  - Ensures instructions produced for different values do not intermingle.

5. For all $\iota \in \mathit{translate}\,(v, t)$ and $t \in \mathit{fn}\,(\iota)$, there exists ι' such that $\iota' \in \mathit{translate}\,(v, t)$ and $\mathit{bn}\,(\iota') = t$.
   - The graph of instruction dependencies must be proper, without dangling edges.
6. For all $t_1 \leftarrow c_1?o_1 \in \mathit{translate}\,(v, t)$ and $t_2 \leftarrow c_2?o_2 \in \mathit{translate}\,(v, t)$, if $t_2 \in n\,(c_1)$, then $c_2 = \mathit{true}$.
   - If a name is used in a guard expression, the instruction with that bound name must be trivially completable.
   - For example, if the instruction with that bound name has a *false* guard, there is no chance for the guard expression that contains the name to evaluate to a value.
7. For all stores s, if $t_1 \leftarrow c_1?o_1 \in \mathit{translate}\,(v, t)$, $t_2 \leftarrow c_2?o_2 \in \mathit{translate}\,(v, t)$, $t_2 \in n\,(o_1)$, $[c_1]s$, and $[c_2]s = v'$, then $v' \neq \mathit{false}$.
   - When an instruction with a true guard depends on another instruction, and the latter instruction's guard expression evaluates to a value, it must not be *false*.
8. If $t' \leftarrow c?\mathit{st}\,\mathcal{PC}\,t_1\,t_2 \in \mathit{translate}\,(v, t)$, then $t_1 \leftarrow \mathit{true}?0 \in \mathit{translate}\,(v, t)$.
   - PC stores are always to address zero.
9. If $t' \leftarrow c?\mathit{ld}\,\mathcal{PC}\,t'' \in \mathit{translate}\,(v, t)$, then $t'' \leftarrow \mathit{true}?0 \in \mathit{translate}\,(v, t)$.
   - PC loads are always from address zero.

Remark. The requirements of *translate* are tailored to preserved state well-formedness, as defined below. We do not provide any canonical complete definition of *translate*, since it is application dependent.

Definition (Address of Name)**.** The address of a name t for the program I, when it is defined, is a tuple (τ, t'). We define the partial function *addr* by case analysis.

- $\mathit{addr}\,(I, t) = (\tau, t')$, if $t \leftarrow c?\mathit{ld}\,\tau\,t' \in I$ for some c.
- $\mathit{addr}\,(I, t) = (\tau, t')$, if $t \leftarrow c?\mathit{st}\,\tau\,t'\,t'' \in I$ for some c and t''.

Definition (Store May)**.** The function *str-may* returns the set of store microinstructions in a state that *may* affect the instruction with the given bound name. For a state $\sigma = (I, s, C, F)$ and name t, *str-may* is defined as follows.

$$\begin{aligned}\mathit{str\text{-}may}\,(\sigma, t) = \{t' \leftarrow c'?\mathit{st}\,\tau\,t_1\,t_2 \in I \mid t' < t \wedge \exists t_0.\, \mathit{addr}\,(I, t) = (\tau, t_0) \wedge \\ ([c']s \vee [c']s \uparrow) \wedge ((\exists v.\, s(t_1) = v \wedge s(t_0) = v) \vee s(t_1) \uparrow \vee s(t_0) \uparrow)\}\end{aligned}$$

Definition (Store Active)**.** The function *str-act* returns the set of store microinstructions that are *active* for the instruction with the given bound name. For a state $\sigma = (I, s, C, F)$ and name t, *str-act* is defined as follows.

$$\begin{aligned}\mathit{str\text{-}act}\,(\sigma, t) = \{t' \leftarrow c'?\mathit{st}\,\tau\,t_1\,t_2 \in \mathit{str\text{-}may}\,(\sigma, t) \mid \exists t_0.\, \mathit{addr}\,(I, t) = (\tau, t_0) \wedge \\ \neg(\exists t'' \leftarrow c''?\mathit{st}\,\tau\,t_1'\,t_2' \in \mathit{str\text{-}may}\,(\sigma, t).\, t'' > t' \wedge [c'']s \wedge \\ ((\exists v.\, s(t_1') = v \wedge s(t_0) = v) \vee (\exists v.\, s(t_1') = v \wedge s(t_1) = v)))\}\end{aligned}$$

Definition (Semantics of Instructions)**.** The semantics of instructions is given by a partial function on instructions ι and states $\sigma = (I, s, C, F)$, returning a tuple of a value and an observation. We write $[\iota]\sigma = (v, obs)$ and define the function by case analysis on ι.

- $[t \leftarrow c?e]\sigma = (v, \epsilon)$, if $[e]s = v$.
- $[t \leftarrow c?ld\,\tau\,t']\sigma = (v, dl\,a)$, if $bn\,(\mathit{str\text{-}act}(\sigma, t)) = \{t''\}$, $s(t') = a$, $s(t'') = v$, $\tau = \mathcal{M}$, and $t'' \in C$.
- $[t \leftarrow c?ld\,\tau\,t']\sigma = (v, \epsilon)$, if $bn\,(\mathit{str\text{-}act}(\sigma, t)) = \{t''\}$, $s(t') = a$, $s(t'') = v$, and either $\tau \neq \mathcal{M}$ or $t'' \notin C$.
- $[t \leftarrow c?st\,\tau\,t_1\,t_2]\sigma = (v, \epsilon)$, if $s(t_1) = v$ and $s(t_2)\downarrow$.

Definition (Completed Microinstruction)**.** A microinstruction ι is *completed* in a state $\sigma = (I, s, C, F)$, written $\mathcal{C}\,(\sigma, \iota)$, precisely when:

- $\iota = t \leftarrow c?st\,\mathcal{M}\,t_1\,t_2$ and either $[c]s = \mathit{false}$ or $t \in C$.
- $\iota = t \leftarrow c?st\,\mathcal{PC}\,t_1\,t_2$ and either $[c]s = \mathit{false}$ or $t \in F$.
- $\iota = t \leftarrow c?o$ otherwise, and either $[c]s = \mathit{false}$ or $t \in dom\,(s)$.

3.3 Transition Step Relations

Definition (Out-of-Order Relation)**.** The labeled transition relation $\sigma \xrightarrow{l}\!\!\!\!\rightarrow \sigma'$ for out-of-order steps is the least relation defined by the three rules below.

$$\frac{\begin{array}{l}\sigma = (I, s, C, F)\\ \iota = t \leftarrow c?o\\ \iota \in I\\ [\iota]\sigma = (v, obs)\\ s(t)\uparrow\\ [c]s\end{array}}{(I, s, C, F) \xrightarrow{(obs,\,\textsc{Exe},\,t)} (I, s + [t \mapsto v], C, F)}\ \textsc{OoO\_Exe}$$

$$\frac{\begin{array}{l}\sigma = (I, s, C, F)\\ s(t)\downarrow\\ t \leftarrow c?st\,\mathcal{M}\,t_1\,t_2 \in I\\ t \notin C\\ s(t_1) = a\\ s(t_2) = v\\ bn\,(\mathit{str\text{-}may}(\sigma, t)) \subseteq C\end{array}}{(I, s, C, F) \xrightarrow{(ds\,a,\,\textsc{Cmt}\,(a, v),\,t)} (I, s, C \cup \{t\}, F)}\ \textsc{OoO\_Cmt}$$

$$\frac{\begin{array}{l}\sigma = (I, s, C, F)\\ t \leftarrow c?st\,\mathcal{PC}\,t_1\,t_2 \in I\\ s(t) = v\\ t \notin F\\ bn\,(\mathit{str\text{-}may}(\sigma, t)) \subseteq F\\ \mathit{translate}\,(v, \mathit{max}\,(bn\,(I))) = I'\end{array}}{(I, s, C, F) \xrightarrow{(il\,v,\,\textsc{Ftc}\,(I'),\,t)} (I \cup I', s, C, F \cup \{t\})}\ \textsc{OoO\_Ftc}$$

OoO-Exe: Computes the value v of an instruction with bound name t and records the result in the store by adding the mapping $[t \mapsto v]$.

OoO-Ftc: Fetches an already-executed PC store instruction, which potentially adds more instructions to the program in the state.

OoO-Cmt: Commits an already-executed memory store instruction to memory.

Definition (In-Order Relation)**.** The in-order labeled transition relation $\sigma \xrightarrow{l} \sigma'$ is the least relation defined by the single rule below.

$$\frac{\sigma \xrightarrow{(obs,\, \alpha,\, t)}\!\!\!\!\rightarrow \sigma' \qquad \forall\, \iota \,\in\, \sigma.\text{if } bn\,(\iota) < t \,\text{then}\, \mathcal{C}\,(\sigma, \iota)}{\sigma \xrightarrow{(obs,\, \alpha,\, t)} \sigma'} \ \text{IO\_STEP}$$

IO-Step: Processes instructions according to the OoO rules, but deterministically according to the program order.

4 Metatheory

This section describes the key semantic definitions and results in the metatheory of MIL.

4.1 List Theory

To define MIL executions, we use a standard theory of polymorphic lists. This subsection introduces our list notations and utility functions.

Definition (Lists)**.** The empty list is written $[\,]$, while a list with k elements is written $[e_1, \ldots, e_k]$. The concatenation of two lists π and π' is written $\pi +\!\!+ \pi'$. The list with head element e and tail π is written $e :: \pi$.

Definition (Element at List Position)**.** For a list π, the partial function *nth* returning the element at position n, starting from zero, is defined as follows:

- $nth(0, e :: \pi) = e$
- $nth(n + 1, e :: \pi) = nth(n, \pi)$

Definition (List Prefixes)**.** The list prefix relation $\preceq$ is defined inductively.

- $[\,] \preceq \pi$.
- If $\pi \preceq \pi'$, then $e :: \pi \preceq e :: \pi'$.

Definition (List Length)**.** The length of a list is defined in the obvious way:

- $length([\,]) = 0$
- $length(e :: \pi) = length(e) + 1$

Definition (List Head and Last Element)**.** Partial functions for obtaining the head and last element of a list are defined as per below.

- $hd(e :: \pi) = e$.

- $\mathit{last}(e :: \pi) = e$, if $\pi = [\,]$.
- $\mathit{last}(e :: \pi) = \mathit{last}(\pi)$, if $\pi \neq [\,]$.

Definition (Relation Based Permutations)**.** Let R be an equivalence relation. Then, two lists π and π' are permutations under R, written $\mathit{perm}_R(\pi, \pi')$, is the smallest relation closed under the following rules:

- $\mathit{perm}_R([\,], [\,])$.
- If $R(e_1, e_2)$ and $\mathit{perm}_R(\pi_1, \pi_2)$, then $\mathit{perm}_R(e_1 :: \pi_1, e_2 :: \pi_2)$.
- $\mathit{perm}_R(e_1 :: e_2 :: \pi, e_2 :: e_1 :: \pi)$.
- If $\mathit{perm}_R(\pi_1, \pi_2)$ and $\mathit{perm}_R(\pi_2, \pi_3)$, then $\mathit{perm}_R(\pi_1, \pi_3)$.

4.2 Auxiliary Semantic Definitions

Definition (Well Formed State)**.** A state $\sigma = (I, s, C, F)$ is *well formed* precisely when all the following properties hold:

1. I is a finite set.
   - MIL semantics require the maximum instruction name to be well defined.
   - Infinite collections of instructions do not occur in the intended applications of MIL.
2. $C \cup F \subseteq \mathit{dom}\,(s)$.
   - Instructions for names in C and F must have been executed.
3. $\mathit{dom}\,(s) \subseteq \mathit{bn}\,(I)$.
   - There must be an instruction for each name that has been recorded as executed.
4. For all $\iota \in I$ and $t \in \mathit{fn}\,(\iota)$, $t < \mathit{bn}\,(\iota)$.
   - The graph of instruction dependencies must be acyclic.
   - This property makes linear instruction execution guarantee completeness.
5. For all $\iota \in I$ and $\iota' \in I$, if $\mathit{bn}\,(\iota) = \mathit{bn}\,(\iota')$ then $\iota = \iota'$.
   - Bound names of instructions must be unique in I.
6. For all $\iota \in I$ and $t \in \mathit{fn}\,(\iota)$, there exists some ι' such that $\iota' \in I$ and $\mathit{bn}\,(\iota') = t$.
   - The graph of instruction dependencies must be proper, without dangling edges.
7. For all $t \in C$, there exists t_1, t_2, and c such that $t \leftarrow c?\mathit{st}\,\mathcal{M}\,t_1\,t_2 \in I$.
   - The set C contains only bound names of memory store instructions.
8. For all $t \in F$, there exists t_1, t_2, and c such that $t \leftarrow c?\mathit{st}\,\mathcal{PC}\,t_1\,t_2 \in I$.
   - The set C contains only bound names of PC store instructions.
9. For all $t \leftarrow c?\mathit{st}\,\tau\,t_1\,t_2 \in I$, if $s(t) = v$, then $s(t_1)\downarrow$ and $s(t_2) = v$.

- Store instruction execution results are consistent with results for instruction dependencies.

10. For all $t \leftarrow c?o \in I$, if $s(t)\downarrow$, then $[c]s$.
    - Executed instructions have true guards.
11. If $t \leftarrow c?st\,\mathcal{PC}\,t_1\,t_2 \in I$, then $t_1 \leftarrow \mathit{true}?0 \in I$.
    - PC stores are always to address zero.
12. For all $t \leftarrow c?o \in I$ and $t' \leftarrow c'?o' \in I$, if $t' \in n(c)$, then $c' = \mathit{true}$.
    - If a name is used in a guard expression, the instruction with that bound name must be trivially completable.
    - For example, if the instruction with that bound name has a *false* guard, there is no chance for the guard expression that contains the name to evaluate to a value.
13. For all stores s', if $t \leftarrow c?o \in I$, $t' \leftarrow c'?o' \in I$, $t' \in n(o)$, $[c]s'$, and $[c']s' = v'$, then $v' \neq \mathit{false}$.
    - When an instruction with a true guard depends on another instruction, and the latter instruction's guard expression evaluates to a value, it must not be *false*.
14. For all $t \leftarrow c?e \in I$, if $s(t) = v$, then $[t \leftarrow c?e]\sigma = (v, \epsilon)$.
    - Results for instructions with internal operations are consistent with the semantics of instructions.
15. If $t \leftarrow c?ld\,\mathcal{PC}\,t' \in I$, then $t' \leftarrow \mathit{true}?0 \in I$.
    - PC loads are always from address zero.
16. If $t \in C$, then $bn\,(\mathit{str\text{-}may}\,(\sigma, t)) \subseteq C$.
    - For all committed memory store instructions, all store instructions that may affect those instructions are already committed.
17. If $t \in F$, then $bn\,(\mathit{str\text{-}may}\,(\sigma, t)) \subseteq F$.
    - For all fetched PC store instructions, all stores instructions that may affect those instructions are already fetched.

Remark. A MIL program does not have any canonical initial state. In lieu of initial states, well-formedness collects some elementary state sanity properties. In particular, well-formedness prevents some errors during execution, e.g., due to dangling microinstruction references. However, the properties are not necessarily exhaustive in ruling out states with unintuitive behavior.

Definition (Empty State)**.** The *empty state* consists of the empty store and empty sets, $(\emptyset, [\,], \emptyset, \emptyset)$.

Definition (Executions)**.** Let R be a relation on triples (σ, l, σ') where σ and σ' are states, and l is a label. Let π be a list of such state-label triples. Then, π is an *execution* precisely when one of the following holds:

1. $\pi = [(\sigma, l, \sigma')]$ and $R(\sigma, l, \sigma')$.
2. $\pi = \pi' +\!\!+ [(\sigma_1, l_1, \sigma_2), (\sigma_2, l_2, \sigma_3)]$, $R(\sigma_2, l_2, \sigma_3)$, and $\pi' +\!\!+ [(\sigma_1, l_1, \sigma_2)]$ is an execution.

When π is an execution for $\twoheadrightarrow$, π is called an *OoO execution*. When π is an execution for $\rightarrow$, π is called an *IO execution*. We write $\pi(i)$ for $nth(n, \pi)$.

Definition (Triple Access Functions)**.** We define access functions using numbers for state-label triples used in executions.

- $1(\sigma, l, \sigma') = \sigma$.
- $2(\sigma, l, \sigma') = l$.
- $3(\sigma, l, \sigma') = \sigma'$.

Definition (Traces)**.** An observation *obs* is considered *visible* if $obs \neq \epsilon$. A *trace* is a list of visible observations. From an execution π, the function *trace* obtains a trace by first extracting out all observations from labels in π, and then keeping only visible ones.

Definition (Commits)**.** For an OoO or IO execution π and address value a, we define a recursive function $commits(\pi, a)$ which extracts out values in commit transitions associated with a.

- $commits([\,], a) = [\,]$
- $commits((\sigma, (obs, \text{CMT}\,(a, v), t), \sigma') :: \pi', a) = v :: commits(\pi', a)$
- $commits((\sigma, (obs, \text{CMT}\,(a', v), t), \sigma') :: \pi', a) = commits(\pi', a)$, if $a \neq a'$
- $commits((\sigma, l, \sigma') :: \pi', a) = commits(\pi', a)$, otherwise.

Definition (Same Action and Name Relation)**.** The relation R_{AN} is true for two triples $(\sigma_1, l_1, \sigma_1')$ and $(\sigma_2, l_2, \sigma_2')$ whenever the names and actions in l_1 and l_2 are the same. R_{AN} is then an equivalence relation.

Definition (Ordered Execution)**.** An execution π is an *ordered execution* precisely when, for all natural numbers i and j where $i < j < length(\pi)$, the name in the label at i is less than or equal to the name in the label at j in π.

Definition (Ordered Version)**.** The execution π' is an *ordered version* of the execution π precisely when

- $1(hd(\pi)) = 1(hd(\pi'))$,
- $3(last(\pi) = 3(last(\pi'))$,
- for every (address) value a, it holds that $commits(\pi)(a) = commits(\pi')(a)$,
- $perm_{R_{AN}}(\pi, \pi')$, and
- π' is an ordered execution.

Definition (Initialized Resource for Values)**.** The predicate $initialized\text{-}resource\text{-}in\text{-}set(\sigma, \tau, V)$ is true precisely when, for all $v \in V$, there exists a completed store instruction for τ in σ such that there is no earlier load instruction in σ for τ.

Definition (Initialized Resource)**.** The predicate *initialized-resource*(σ, τ) is defined by case analysis on τ.

- *initialized-resource*$(\sigma, \mathcal{PC})$ = *initialized-resource-in-set*$(\sigma, \mathcal{PC}, \{0\})$
- *initialized-resource*$(\sigma, \mathcal{R})$ = *initialized-resource-in-set*$(\sigma, \mathcal{R}, \mathcal{U}_v)$
- *initialized-resource*$(\sigma, \mathcal{M})$ = *initialized-resource-in-set*$(\sigma, \mathcal{M}, \mathcal{U}_v)$

Definition (Initialized State)**.** A state σ is *initialized* when all resources in σ are initialized, i.e., when *initialized-resource*(σ, τ) holds for $\tau = \mathcal{PC}, \mathcal{R}, \mathcal{M}$.

4.3 Transition Step Relation Properties

Lemma 1 (OoO Well-Formedness Preservation)**.** If σ is well formed and $\sigma \xrightarrow{(obs,\,\alpha,\,t)}\!\!\!\!\rightarrow \sigma'$, then σ' is well formed.

Lemma 2 (OoO Determinism)**.** Let σ be a well-formed state and suppose $\sigma \xrightarrow{(obs_1,\,\alpha_1,\,t)}\!\!\!\!\rightarrow \sigma_1$ and $\sigma \xrightarrow{(obs_2,\,\alpha_2,\,t)}\!\!\!\!\rightarrow \sigma_2$. Then $obs_1 = obs_2$, $\alpha_1 = \alpha_2$, and $\sigma_1 = \sigma_2$.

Proof. By case analysis on the possible transitions, using well-formedness properties such as instruction name uniqueness. □

Remark. This lemma shows that the only degree of freedom when performing OoO transitions is in the choice of instruction name.

Lemma 3 (IO Determinism)**.** Let σ be a well-formed state and suppose $\sigma \xrightarrow{l_1} \sigma_1$ and $\sigma \xrightarrow{l_2} \sigma_2$. Then $l_1 = l_2$ and $\sigma_1 = \sigma_2$.

4.4 Memory Consistency of the Out-of-Order and In-Order Semantics

Lemma 4. Let $\sigma_0 = (I_0, s_0, C_0, F_0)$ be a well-formed state and let $\sigma_1 = (I_1, s_1, C_1, F_1)$ and $\sigma_2 = (I_2, s_2, C_2, F_2)$ be states such that $\sigma_0 \xrightarrow{(obs_1,\,\alpha_1,\,t_1)}\!\!\!\!\rightarrow \sigma_1$ and $\sigma_1 \xrightarrow{(obs_2,\,\alpha_2,\,t_2)}\!\!\!\!\rightarrow \sigma_2$, with $t_2 < t_1$. Then, there is a state $\sigma_2' = (I_0 \cup I_2', s_0 \cup s_2', C_0 \cup C_2', F_0 \cup F_2')$ such that $\sigma_0 \xrightarrow{(obs,\,\alpha_2,\,t_2)}\!\!\!\!\rightarrow \sigma_2'$, where $I_2 = I_1 \cup I_2'$, $s_2 = s_1 \cup s_2'$, $C_2 = C_1 \cup C_2'$, $F_2 = F_1 \cup F_2'$, and $dom\,(s_0) \cap dom\,(s_2') = \emptyset$. Morever, for all a, if there exists v such that $\alpha_2 = \textsc{Cmt}\,(a, v)$, then there does not exist v' such that $\alpha_1 = \textsc{Cmt}\,(a, v')$.

Proof. By case analysis on the possible transitions. □

Lemma 5 (Two-Step Reordering)**.** Let σ_0 be a well-formed state and suppose $\sigma_0 \xrightarrow{(obs_1,\,\alpha_1,\,t_1)}\!\!\!\!\rightarrow \sigma_1$ and $\sigma_1 \xrightarrow{(obs_2,\,\alpha_2,\,t_2)}\!\!\!\!\rightarrow \sigma_2$, with $t_2 < t_1$. Then, there exists σ', obs_1', and obs_2' such that $\sigma_0 \xrightarrow{(obs_2',\,\alpha_2,\,t_2)}\!\!\!\!\rightarrow \sigma'$ and $\sigma' \xrightarrow{(obs_1',\,\alpha_1,\,t_1)}\!\!\!\!\rightarrow \sigma_2$. Morever, for all a, if there exists v such that $\alpha_2 = \textsc{Cmt}\,(a, v)$, then there does not exist v' such that $\alpha_1 = \textsc{Cmt}\,(a, v')$.

Proof. By Lemma 4 and case analysis on the possible transitions. □

Lemma 6 (Exists Reordering)**.** Let π be an OoO execution such that $1(\pi(0))$ is well-formed. Then, there is an OoO execution π' that is an ordered version of π.

Lemma 7 (Complete Consistent)**.** Let π be an OoO execution of length k with $1(\pi(0))$ well-formed, such that if $i \leq j < k$, then $n(2(\pi(j))) > t$. Then, if $\mathcal{C}(3(\pi(k-1)), t)$, we have $\mathcal{C}(3(\pi(i)), t)$.

Lemma 8 (Execution Result)**.** Let π be an ordered OoO execution of length k with $1(\pi(0))$ well-formed. Assume that if $t > t_{max}$, then there is no t in the store of $3(\pi(k-1))$. Then, for all $0 \leq i < k$, we have $n(2(\pi(i))) \leq t_{max}$.

Lemma 9 (Main Reordering)**.** Let π be an ordered OoO execution with $1(\pi(0))$ well-formed. Assume there exists t_{max} such that (i) if t is in the store of $3(\mathit{last}(\pi))$ and $t \leq t_{max}$, then $\mathcal{C}(3(\mathit{last}(\pi)), t)$, and (ii) if $t' > t_{max}$, then there is no t' in the store of $3(\mathit{last}(\pi))$. Then, π is an IO execution.

Theorem 1 (OoO-IO Memory Consistency)**.** Let π be an OoO execution where $1(\pi(0))$ is well-formed and initialized. Then, there is an IO execution π' such that $1(\pi(0)) = 1(\pi'(0))$ and for all a, $\mathit{commits}(\pi, a)$ is a prefix of $\mathit{commits}(\pi', a)$.

Remark. Let π be an IO execution. Then there exists an OoO execution π' such that $1(\pi(0)) = 1(\pi'(0))$ and for all a, $\mathit{commits}(\pi, a)$ is a prefix of $\mathit{commits}(\pi', a)$.

Proof. Take $\pi' = \pi$, since an IO execution is also an OoO execution. $\square$

References

[1] HOL Theorem Prover, version kananaskis-14. https://github.com/HOL-Theorem-Prover/HOL/releases/tag/kananaskis-14.