Proving a Petri net model-checker implementation

Lukasz Fronc, Franck Pommereau

Proving a Petri net model-checker implementation

IBISC technical report, March 2011

Lukasz Fronc and Franck Pommereau

IBISC, University of Évry, Tour Évry 2
523 place des terrasses de l'Agora, 91000 Évry, France
{fronc,pommereau}@ibisc.univ-evry.fr

Abstract. *Petri nets* are a widely used tool in verification through *model-checking*. In this approach, a Petri Net model of the system of interest is produced and its reachable states are computed, searching for erroneous executions. *Compilation* of such a Petri net model is one way to accelerate its verification. It consists in generating code to explore the reachable states of the considered Petri net, which avoids the use of a fixed exploration tool involving an "interpretation" of the Petri net structure.
In this paper, we show how to compile Petri nets targeting the *LLVM language* (a high-level assembly language) and formally prove the *correctness* of the produced code. To this aim, we define a *structural operational semantics* for the fragment of LLVM we use. The acceleration obtained from the presented compilation techniques has been evaluated in [6].

Keywords: explicit model-checking, model compilation, LLVM, formal semantics, correctness

1 Introduction

Verification through *model-checking* [2] consists in defining a formal model of the system to be analysed and then using automated tools to check whether the expected properties are met or not. We consider here more particularly the widely adopted setting in which models are expressed in *coloured Petri nets* [7] and there states are explored using *explicit model-checking* that enumerates them all (contrasting with symbolic model-checking that handles directly sets of states).

Model compilation is one of the numerous techniques to speedup explicit model-checking, it relies on generating source code (then compiled into machine code) to produce a high-performance implementation of the state space exploration primitives. For instance, this approach is successfully used in the popular coloured Petri net model-checker Helena [15, 4] that generates C code.

In this paper, we present an approach for proving the correctness of such an approach. More precisely, we focus on the produced code and prove that the object computed by its execution is an actual representation of the state space of the compiled model. We consider the *Low-Level Virtual Machine* (*LLVM*)

language as our target language for the models compilation, which reconciles two otherwise contradictory objectives: on the one hand, this is a typed language with reasonably high-level operations allowing to express algorithms quite naturally; on the other hand, it is still a low-level language and can be equipped with formal semantics allowing to formally prove the correctness of the programs. To carry on these proofs, we define a structural operational semantics of the fragment of LLVM we use and use it in various ways to establish the properties of our programs.

To the best of our knowledge, this is the first attempt to provide a formal semantics for LLVM. Moreover, if model-checkers are widely used tools, there exists surprisingly few attempts to prove them [18], contrasting with the domain of proof assistants [3, 16] where the importance of "proving the prover" is widely acknowledged. Others approaches like certifying model-checkers [8] produce deductive proofs on success or failure, and thus only that proof needs to be checked (for each result). On the side of compiler proving, proofs address mostly semantic preservation [13, 14], *i.e.*, ensure that the compilation process respects the semantics of the input language when producing the compiled code. Others, address specific optimisation phases in a compiler [1].

The rest of the paper is organised as follows. We first recall the main notions about coloured Petri nets. Then, we present the LLVM framework, in particular the syntax of the language and its intuitive semantics, and how it can be embedded LLVM into a Petri net as a concrete colour domain. In section 4, we present algorithms and data structures for state space exploration. We formally define the operational semantics for LLVM, including a memory model with explicit heap and stack. Finally we present our correctness results. Due to limited number of pages, some definitions and intermediary results have been omitted, as well as the detailed proves. This material can be found in [5]. Notice also that our compilation approach is evaluated from a performance point of view in [6].

2 Coloured Petri nets

A (coloured) Petri net involves objects defined by a *colour domain* that provides data values, variables, operators, a syntax for expressions, possibly typing rules, etc. Usually, elaborated colour domains are used to ease modelling; in particular, one may consider a functional programming language [7, 17] or the functional fragment (expressions) of an imperative programming language like in Helena [15]. In this paper we will consider LLVM as a concrete colour domain. All these can be seen as implementations of a more general *abstract colour domain*:

- $\mathbb{D}$ is the set of *data values*;
- $\mathbb{V}$ is the set of *variables*;
- $\mathbb{E}$ is the set of *expressions*. Let $e \in \mathbb{E}$, we denote by $vars(e)$ the set of variables from $\mathbb{V}$ involved in e. Moreover, variables or values may be considered as (simple) expressions, *i.e.*, we assume $\mathbb{D} \cup \mathbb{V} \subseteq \mathbb{E}$.

We do not make any assumption about the typing or syntactical correctness of expressions; instead, we assume that any expression can be evaluated, possibly to $\bot \notin \mathbb{D}$ (undefined value) in case of any error. More precisely, a *binding* is a partial function $\beta : \mathbb{V} \to \mathbb{D} \cup \{\bot\}$. Let $e \in \mathbb{E}$ and β be a binding, we denote by $\beta(e)$ the evaluation of e under β; if the domain of β does not include $vars(e)$ then $\beta(e) \stackrel{\text{df}}{=} \bot$. The application of a binding to evaluate an expression is naturally extended to sets and multisets of expressions.

Definition 1 (Petri nets). *A* Petri net *is a tuple* (S, T, ℓ) *where:*

- S *is the finite set of* places*;*
- T*, disjoint from* S*, is the finite set of* transitions*;*
- ℓ *is a* labelling *function such that:*
  - *for all* $s \in S$*,* $\ell(s) \subseteq \mathbb{D}$ *is the* type *of* s*,* i.e.*, the set of values that* s *is allowed to carry,*
  - *for all* $t \in T$*,* $\ell(t) \in \mathbb{E}$ *is the* guard *of* t*,* i.e.*, a condition for its execution,*
  - *for all* $(x, y) \in (S \times T) \cup (T \times S)$*,* $\ell(x, y)$ *is a multiset over* $\mathbb{E}$ *and defines the* arc *from* x *toward* y*.*

A marking *of a Petri net is a map that associates to each place* $s \in S$ *a multiset of values from* $\ell(s)$*. From a marking* M*, a transition* t *can be* fired *using a binding* β *and yielding a new marking* M'*, which is denoted by* $M[t, \beta\rangle M'$*, iff:*

- *there are enough tokens: for all* $s \in S$*,* $M(s) \geq \beta(\ell(s, t))$*;*
- *the guard is validated:* $\beta(\ell(t)) = true$*;*
- *place types are respected: for all* $s \in S$*,* $\beta(\ell(t, s))$ *is a multiset over* $\ell(s)$*;*
- M' *is* M *with tokens consumed and produced according to the arcs: for all* $s \in S$*,* $M'(s) = M(s) - \beta(\ell(s, t)) + \beta(\ell(t, s))$*.*

Such a binding β *is called a* mode *of* t *at marking* M*.*

For a Petri net node $x \in S \cup T$*, we define* ${}^\bullet x \stackrel{\text{df}}{=} \{y \in S \cup T \mid \ell(y, x) \neq \emptyset\}$ *and* $x^\bullet \stackrel{\text{df}}{=} \{y \in S \cup T \mid \ell(x, y) \neq \emptyset\}$ *where* $\emptyset$ *is the empty multiset. Finally, we extend the notation vars to a transition by taking the union of the variable sets in its guard and connected arcs.* ◊

In this paper, we assume that the considered Petri nets are such that, for all place $s \in S$ and all transition $t \in T$, $\ell(s, t)$ is either $\emptyset$ or contains a single variable $x \in \mathbb{V}$. We also assume that $vars(t) = \bigcup_{s \in S} vars(\ell(s, t))$, *i.e.*, all the variables involved in a transition can be bound using the input arcs. The second assumption is a classical one that allows to simplify the discovery of modes. The first assumption is made without loss of generality to simplify the presentation.

3 LLVM

The *LLVM* project (*Low Level Virtual Machine*) [10] is a modern and modular toolkit for compiler development. It began as a research project at the University of Illinois and has grown to an umbrella project for number of sub-projects, many

of which being used by a wide variety of commercial and open source projects [12] as well as academic researches [11].

The *LLVM-IR* (*LLVM Intermediate Representation*) [9] is a part of the LLVM project and is a low-level platform-independent intermediate language. Every program written in this LLVM language can be run in a virtual machine or compiled to native code on all the platforms supported by the LLVM project. Importantly, the LLVM compiler runs a set of optimisation passes on the LLVM-IR code, which allows us to produce simple source code knowing it will be optimised by LLVM, and so lets us concentrate on the higher-level optimisations of algorithms and data structures.

3.1 Syntax and intuitive semantics.

A LLVM program is composed of a set of *blocks* identified by *labels*. Each block contains a sequence of instructions, some are entry points for subprograms (functions or procedures) and are identified by labels including the formal parameters. Entering or leaving a block is always explicit through branching instructions (jumps), subprograms calls or return instructions. Moreover, LLVM is a *SSA* (*Single Static Assignment*) based representation. It means that each variable is assigned exactly once, but since a block can be executed many times the value may be dynamically modified. This property is useful in some proofs because it can serve as an invariant.

To define the syntax, we consider the following pairwise disjoint sets:

- $\mathbb{P}$ is the set of *pointers*;
- $\mathbb{T}$ is the set of *types*, defined as the smallest set containing $\{int, bool, object\}$ (*i.e.*, integers, Boolean values and other values) and such that if $t_0, \ldots, t_n \in \mathbb{T}$ then $struct(t_0, \ldots, t_n) \in \mathbb{T}$, it represents a data structure with $n+1$ fields of types t_0 to t_n;
- $\mathbb{L}$ is the set of *labels*, it contains some specific labels like $f(\!| a_1, \ldots, a_n |\!)$, where $a_i \in \mathbb{V}$ for $1 \leq i \leq n$, that correspond to subprograms entry points (including the formal parameters). We define a set $\mathbb{L}_\perp \stackrel{\text{df}}{=} \mathbb{L} \cup \{\perp\}$ where $\perp$ is an undefined label.

A program is represented as a partial function P with $\mathbb{L}$ as its domain and that associates each label in its domain to a sequence of instructions.

For our purpose, we need to consider a fragment of LLVM that is restricted to the types presented above and is formed by three main syntax classes: sequences (*seq*), commands (*cmd*) and expressions (*expr*). A sequence is a list of commands which may end with an expression, in which case it is considered as an expression.

We assume that programs are syntaxically correct and well typed, so that we can simplify the syntax by forgetting all types in LLVM source code. The resulting syntax is presented in figure 1. We introduce the sequencing operator ";", which corresponds to the line endings, in order to write one-line sequences. We introduce the *skip* command that denotes the empty sequence and does not exist in LLVM. It may be noted that *pcall* (procedure call) and *fcall* (function

call) do not exist in LLVM but are different instances of the *call* instruction. This distinction can be easily made in LLVM because the instruction contains the type of the subprogram (function or procedure). Instruction *store* (resp. *load*) is the action of storing (resp. loading) data into (resp. from) the memory through a pointer. Instruction *icmp* compares two integers. Instruction *phi* is used to access variables defined in previously executed blocks. Instruction *gep* corresponds to pointer arithmetic, we freeze the second argument to 0, which is enough to access fields in structures by numbers.

seq	::= cmd	(statement)
	$\mid$ $expr$	(expression)
	$\mid$ $cmd;\ seq$	(sequence of instructions)
cmd	::= $br\ label$	(unconditional branching)
	$\mid$ $br\ rvalue,\ label,\ label$	(conditional branching)
	$\mid$ $pcall\ label(rvalue, \ldots, rvalue)$	(procedure call)
	$\mid$ ret	(return from a procedure)
	$\mid$ $var = expr$	(variable assignment)
	$\mid$ $store\ rvalue,\ rvalue$	(assignment through a pointer)
	$\mid$ $skip$	(empty sequence)
$expr$	::= $add\ rvalue,\ rvalue$	(addition)
	$\mid$ $load\ rvalue$	(read a value through a pointer)
	$\mid$ $gep\ rvalue,\ 0,\ nat$	(get a pointer from a structure)
	$\mid$ $icmp\ op,\ rvalue, rvalue,$	(integers comparison)
	$\mid$ $phi\ (rvalue, label), \ldots, (rvalue, label)$	(get a value after branching)
	$\mid$ $fcall\ label(rvalue, \ldots, rvalue)$	(function call)
	$\mid$ $alloc\ type$	(memory allocation)
	$\mid$ $ret\ rvalue$	(return a value from a function)
	$\mid$ $rvalue$	(value)

Fig. 1. Our fragment of the LLVM syntax, where $label \in \mathbb{L}$, $rvalue \in \mathbb{D} \cup \mathbb{P} \cup \mathbb{V}$, $var \in \mathbb{V}$, $type \in \mathbb{T}$, $nat \in \mathbb{N}$ and $op \in \{<, \leq, =, \neq, \geq, >\}$.

3.2 LLVM-labelled Petri Nets.

To compile Petri nets as defined previously into LLVM programs, we need to consider a variant where the colour domain explicitly refers to a LLVM program.

Definition 2 (LLVM labelled Petri nets). *A LLVM labelled Petri net is thus a tuple $N \stackrel{\text{df}}{=} (S, T, \ell, P)$, where P is a LLVM program, and such that (S, T, ℓ) is a coloured Petri net with the following changes:*

- *for all place $s \in S$, $\ell(s)$ is a LLVM type in $\mathbb{T}$, which can be directly interpreted as a subset of $\mathbb{D}$;*
- *for all transition $t \in T$, $\ell(t)$ is a call to a Boolean function in P with the elements of $vars(t)$ as parameters;*

– *moreover, for all* $s \in t^\bullet$, $\ell(t,s)$ *is a singleton multiset whose unique element is a call to a* $\ell(s)$*-typed function in* P *with the elements of* $vars(t)$ *as parameters.* $\Diamond$

With respect to the previous definition, we have concretized the place types and each expression is now implemented as a LLVM function called from the corresponding annotation. To simplify the presentation, we have also restricted the output arcs to be singleton multisets, but this can be easily generalised. Moreover, the definitions of binding and modes are extended to LLVM. A *LLVM binding* is a partial function $\beta : \mathbb{V} \mapsto \mathbb{D} \cup \mathbb{P}$, which maps each variable from its domain to a pointer or a value, and is widened to $\mathbb{D}$ by the identity function. $\mathbb{B}$ is the set of all LLVM bindings. Thus, A *LLVM mode* is a LLVM binding enabling a transition in a LLVM labelled Petri net.

4 State space exploration

Given an *initial marking* M_0, the state space we want to compute in this paper is the smallest set R such that $M_0 \in R$ and, if $M \in R$ and $M[t,\beta\rangle M'$ then $M' \in R$ also.

Interfaces. To implement algorithms, we need to define interfaces to the data structures we will use: multisets, places, markings, and (marking) sets. An interface is presented as a set of functions that manipulates a data structure through a pointer (C-like interfaces). Moreover, each of interface functions have a formal specification of their comportment, an example is given in section 4, and structure marking have to hold two properties defined in section 6. Below, for a given structure multiset of type $Multiset\langle d\rangle$, what we call its domain is the set of values from d having non-zero occurrences in the multiset. We explicit only the functions used in this document. The *multiset* interface contains the following elements:

– two functions $add_{mset}(\!| p_{mset}, elt |\!)$ and $rem_{mset}(\!| p_{mset}, elt |\!)$ that respectively adds an element elt into the multiset p_{mset} and removes an element from the multiset;
– a function $size_{mset}(\!| p_{mset} |\!)$ that returns the domain size of a multiset;
– a function $nth_{mset}(\!| p_{mset}, n |\!)$ that returns the n^{th} element from the multiset domain (for an arbitrary fixed order);

As a container of tokens, a place can be basically implemented as a multiset of tokens; so the place interface is exactly the multiset interface. The sole difference will be the annotation, a place interface function will be annotated by the place name, for instance add_s is the function add_{mset} specific to the place s. Alternative implementations are considered for optimisation purpose but omitted in this document.

The *markings* interface contains:

– a function $get_s(\!| p_{mrk} |\!)$ for each place s that returns a pointer to the corresponding place structure;
– a function $copy_{mrk}(\!| p_{mrk} |\!)$ that returns a copy of the structure marking.

Finally, the *set* interface contains:

– a function $cons_{set}(\!|\,|\!)$ that builds a new empty set;
– a procedure $add_{set}(\!| p_{set}, elt |\!)$ that adds an element elt to the set p_{set}.

Transitions firing. Let $t \in T$ be a transition such that $s_1, \ldots, s_n$ are its input places (*i.e.*, ${}^\bullet t = \{s_1, \ldots, s_n\}$) and $s_1, \ldots, s'_m$ are its output places (*i.e.*, $t^\bullet = \{s'_1, \ldots, s'_m\}$). Then, function $fire_t$ can be written as shown in figure 2. This function simply creates a copy M' of the marking M, removes from it the consumed tokens and adds the produced ones before to return M'. One could remark that it avoids a loop over the Petri net places but instead it executes a sequence of statements. There are at least two advantages in this approach: it is more efficient (no branching penalties, no loop overhead, no array for the functions $f_{t,s'_j}, \ldots$) and the resulting code is simple to produce. Let also x_{mrq} be a pointer to a structure marking implementing M, and let $x_1, \ldots, x_n$ be representations of the tokens as in figure 2. Then the firing algorithm can be implemented as shown in figure 4. The correctness and termination of this implementation can be formally proved and a sketch of the proof is given in section 6, in particular it is proved that the returned pointer actually represents M'.

$fire_t : M : Mrk,\ x_1 : \ell(s_1),\ \ldots,\ x_n : \ell(s_n) \to Mrk$

$M' \leftarrow copy(M)$
$M'(s_1) \leftarrow M(s_1) - \{x_1\}$
$\ldots$
$M'(s_n) \leftarrow M(s_n) - \{x_n\}$
$M'(s'_1) \leftarrow M(s'_1) + f_{t,s'_1}(x_1, \ldots, x_n)$
$\ldots$
$M'(s'_m) \leftarrow M(s'_m) + f_{t,s'_m}(x_1, \ldots, x_n)$
return M'

Fig. 2. Transition firing algorithm, where $f_{t,s}$ is the function that evaluates $\ell(t,s)$ and Mrk is the *Marking* type.

$$P(fire_t(\!| x_{mrq}, x_1, \ldots, x_n |\!)) \stackrel{\text{df}}{=} \begin{cases} \textit{// copy the structure marking} \\ x'_{mrq} = fcall\ copy_{mrq}(x_{mrq}) \\ \textit{// consume tokens} \\ \left.\begin{array}{l} x_{s_i} = fcall\ get_{s_i}(x'_{mrq}) \\ pcall\ rem_{s_i}(x_{s_i}, x_i) \end{array}\right\} \text{for } 1 \le i \le n \\ \textit{// produce tokens} \\ \left.\begin{array}{l} x_{s'_j} = fcall\ get_{s'_j}(x'_{mrq}) \\ o_{s'_j} = fcall\ f_{t,s'_j}(x_1, \ldots, x_n) \\ pcall\ add_{s'_j}(x_{s'_j}, o_{s'_j}) \end{array}\right\} \text{for } 1 \le j \le m \\ \textit{// return the new marking} \\ ret\ x'_{mrq} \end{cases}$$

Fig. 3. LLVM translation of the firing algorithm, where x_i is the variable in $\ell(s_i, t)$ for all $1 \le i \le n$, and f_{t,s'_j} is the function called in $\ell(t, s'_j)$ for all $1 \le j \le m$.

Successors computation. To discover all the possible modes for a transition, the algorithm enumerates all the combinations of tokens from the input places. If a combination corresponds to a mode then the suitable transition firing function is called to produce a new marking. This algorithm is shown in figure 4. Note the nesting of loops that avoids an iteration on ${}^\bullet t$, which saves from querying the Petri net structure and avoids the explicit construction of a binding. Moreover, if g_t is written in the target language, we avoid an interpretation of the corresponding expression. For the LLVM version, let x_{mrq} be a pointer to a structure marking and x_{next} be a pointer to a marking set structure. Then, the algorithm from figure 4 can be implemented as shown in figure 5. Each loop at the level k is implemented as a set of blocks subscribed by t, k (for $n \geq k \geq 1$), the blocks subscribed by $t, 0$ corresponding to the code inside the innermost loop. Note the *phi* instruction to update the value of index i_{s_i} allowing to enumerate the tokens in place s_i: when the program enters block $loop_i$ for the first time, it comes from block $header_i$, so we initialise the value of i_{s_i} to the maximal index value in the domain of s_i in the current marking; later, the program comes back to block $loop_i$ from block $footer_i$, so it assigns i'_{s_i} to i_{s_i} which is exactly the value of i_{s_i} minus one (the previous index). The correctness and termination of this implementation can be formally proved and a sketch of the proof is given in section 6.

$succ_t : M : Marking,\ next : MarkingSet \rightarrow$

```
for x_n in M(s_n) do
  ...
  for x_1 in M(s_1) do
    if g_t(x_1, ..., x_n) then
      next ← next ∪ {fire_t(M, x_1, ..., x_n)}
    endif
  endfor
  ...
endfor
```

Fig. 4. Transition specific successors computation algorithm, where g_t is the function that evaluates the guard $\ell(t)$.

The global successor function *succ* returns the set of all the successors of a marking by calling all the transition specific successor functions and accumulating the discovered markings into the same set. The algorithm is shown in figure 6 and its translation in LLVM is shown in figure 7.

$$
\begin{aligned}
P(succ_t(\!| x_{mrq}, x_{next} |\!)) &\stackrel{\text{df}}{=} \{\ br\ header_{t,n} \\
P(header_{t,k}) &\stackrel{\text{df}}{=} \begin{cases} x_{s_k} = fcall\ get_{s_k}(x_{mrq}) \\ s_{s_k} = fcall\ size_{s_k}(x_{s_k}) \\ br\ loop_{t,k} \end{cases} \\
P(loop_{t,k}) &\stackrel{\text{df}}{=} \begin{cases} i_{s_k} = phi\ (s_{s_k}, header_{t,k}), (i'_{s_k}, footer_{t,k}) \\ c_{s_k} = icmp\ >,\ i_{s_k},\ 0 \\ br\ c_{s_k},\ body_{t,k},\ footer_{t,k+1} \end{cases} \\
P(body_{t,k}) &\stackrel{\text{df}}{=} \begin{cases} x_k = fcall\ nth_{s_k}(x_{s_k}, i_{s_k}) \\ br\ header_{t,k-1} \end{cases} \\
P(footer_{t,k}) &\stackrel{\text{df}}{=} \begin{cases} i'_{s_k} = add\ i_{s_k},\ -1 \\ br\ loop_{t,k} \end{cases} \\
P(header_{t,0}) &\stackrel{\text{df}}{=} \begin{cases} c_g = fcall\ g_t(x_1, \ldots, x_n) \\ br\ c_g,\ body_{t,0},\ footer_{t,1} \end{cases} \\
P(body_{t,0}) &\stackrel{\text{df}}{=} \begin{cases} x'_{mrq} = fcall\ fire_t(x_{mrq}, x_1, \ldots, x_n) \\ pcall\ add_{set}(x_{next}, x'_{mrq}) \\ br\ footer_{t,1} \end{cases} \\
P(footer_{t,n+1}) &\stackrel{\text{df}}{=} \{\ ret
\end{aligned}
$$

Fig. 5. LLVM successor function, for $1 \leq k \leq n$.

---

succ : *M* : *Marking* → *MarkingSet*

$next \leftarrow \emptyset$
$succ_{t_1}(M, next)$
$succ_{t_2}(M, next)$
...
$succ_{t_n}(M, next)$
return *next*

Fig. 6. Successors computation algorithm.

$$
P(succ(\!| x_{mrq} |\!)) \stackrel{\text{df}}{=}
\begin{cases}
x_{next} = fcall\ cons_{set}() \\
pcall\ succ_{t_1}(x_{mrq}, x_{next}) \\
pcall\ succ_{t_2}(x_{mrq}, x_{next}) \\
\ldots \\
pcall\ succ_{t_n}(x_{mrq}, x_{next}) \\
ret\ x_{next}
\end{cases}
$$

Fig. 7. LLVM implementation of the global successor function, where x_{mrq} is a pointer to a structure marking.

5 Formal semantics of LLVM

5.1 Memory model

Heap We use pointers, therefore we need a heap to access pointed data.

Definition 3 (Heaps). *A* heap *is a partial function* $H : \mathbb{P} \to \mathbb{T} \times (\mathbb{D} \cup \mathbb{P})^*$ *with a finite domain. Each heap maps a pointer to a pair formed of a type and a tuple of values or pointers. The set of all heaps is* $\mathbb{H}$. ◊

Definition 4 (Well formed heaps). *A heap* H *is well formed for a pair* (t, p) *where* t *is a type and* p *a pointer in* $dom(H)$*, if the value pointed by* p *in* H *is sound with* t*, more precisely:*

- *if* $H(p) = (int, d)$*, then* d *is an integer;*

- *if* $H(p) = (bool, d)$*, then* d *is a Boolean;*
- *if* $H(p) = (struct(t_0, \ldots, t_n), d)$*, then* d *is a tuple* $(p_0, \ldots, p_n)$ *and* H *is well formed for each* (t_i, p_i) *such that* $0 \leq i \leq n$*;*

A heap H *is well formed if* H *is well former for each* (t, p) *in* $Im(H)$. ◊

Definition 5 (Accessibility). *The set of all accessible pointers from a pointer* p *in a heap* H *is denoted as* $p{\downarrow_H}$*. This operation is defined for each* p *in* $\mathbb{P}$ *as follows:*

$$\begin{array}{ll} p{\downarrow_H} \stackrel{\text{df}}{=} \{\} & \text{if } p \notin dom(H) \\ p{\downarrow_H} \stackrel{\text{df}}{=} \{p\} & \text{if } H(p) = (t, v) \text{ and } t \in \{int, bool\} \\ p{\downarrow_H} \stackrel{\text{df}}{=} \{p\} \cup p_0{\downarrow_H} \cup \cdots \cup p_n{\downarrow_H} & \text{if } H(p) = (struct(t_0, \ldots, t_n), (p_0, \ldots, p_n)) \\ & \qquad \text{and } t_0, \ldots, t_n \in \mathbb{T} \end{array}$$

◊

One could show that if a heap H is well formed then $p{\downarrow_H} \subseteq dom(H)$ for $p \in dom(H)$, and even that $dom(H) = \bigcup_{p \in \mathbb{P}} p{\downarrow_H}$.

Property 1. *Let* H *be a well formed heap. If* $p \in dom(H)$ *then* $p{\downarrow_H} \subseteq dom(H)$*.* □

Corollary 1. *For each well formed heap* H*, we have* $dom(H) = \bigcup_{p \in \mathbb{P}} p{\downarrow_H}$. □

Definition 6 (Data structure traversal). *For each heap* H*, we define a data structure traversal function* $\cdot[\cdot]_H : \mathbb{P} \times \mathbb{N} \mapsto \mathbb{P} \cup \mathbb{D}$ *as:*

$$\begin{array}{ll} p[i]_H \stackrel{\text{df}}{=} \textit{undefined} & \text{if } p \notin dom(H) \\ p[i]_H \stackrel{\text{df}}{=} \textit{undefined} & \text{if } H(p) = (t, v) \text{ and } t \in \{int, bool\} \\ p[i]_H \stackrel{\text{df}}{=} p_i & \text{if } H(p) = (struct(t_0, \ldots, t_n), (p_0, \ldots, p_n)) \text{ and } 0 \leq i \leq n \end{array}$$

◊

Definition 7 (Heaps overwriting). *The overwriting function* $\oplus : \mathbb{H} \times \mathbb{H} \to \mathbb{H}$*, which represents the writing into memory, is defined for each* $p \in \mathbb{P}$ *as:*

$$(H \oplus H')(p) \stackrel{\text{df}}{=} \begin{cases} H'(p) & \text{if} \quad p \in dom(H') \\ H(p) & \text{if} \quad p \notin dom(H') \wedge p \in dom(H) \\ \textit{undefined otherwise} & \end{cases}$$

◊

Property 2. *The heap overwriting operation* $\oplus$ *is associative.* □

In order to compare heaps, a notion of structural equivalence have been defined. This relation ensures that two heap contain the same data accessible from a pointer with the same layout but different pointers, more precisely we consider $p{\downarrow_H}$ and $p'{\downarrow_{H'}}$ for two heaps H, H' and two pointers p, p'. This relation is written $(H, p) =_{st} (H', p')$, and is defined as the identity check for heap-value pairs. More formally:

Definition 8 (Structural equivalence). *Let H and H' be two well formed heaps, v and v' two values or pointers. We have $(H, v) =_{st} (H', v')$ if one of the following conditions is true:*

- $v \notin dom(H)$, $v' \notin dom(H')$ *and* $v = v'$;
- $H(v) = ((t_0, \ldots, t_n), (p_0, \ldots, p_n))$, $H'(v') = ((t_0, \ldots, t_n), (p'_0, \ldots, p'_n))$ *and* $(H, p_i) =_{st} (H', p'_i)$ *where* $0 \leq i \leq n$. ◊

Property 3. $=_{st}$ *is an equivalence relation.* □

We need also to define an operation to build new heaps. The first step is to define a helper function *alloc* then the function *new* that we seek.

Definition 9 (Alloc and new). *The* $new : \mathbb{H} \times \mathbb{T} \to \mathbb{H} \times \mathbb{P}$ *and* $alloc : 2^{\mathbb{P}} \times \mathbb{P} \times \mathbb{T} \to \mathbb{H}$ *functions are defined as follows:*

$$new(H, t) \stackrel{\text{df}}{=} (alloc(dom(H) \cup \{p\}, p, t), p) \quad \text{for } p \notin dom(H) \text{ a "fresh" pointer}$$

$$alloc(d, p, t) \stackrel{\text{df}}{=} \{p \mapsto (t, \bot)\} \quad \text{for } t \in \{int, bool\}$$

$$\begin{aligned} alloc(d, p, struct(t_0, \ldots, t_n)) \stackrel{\text{df}}{=} & \{p \mapsto (struct(t_0, \ldots, t_n), (p_0, \ldots, p_n))\} \\ & \oplus alloc(d \cup \{p_0, \ldots, p_n\}, p_0, t_0) \\ & \oplus \ldots \\ & \oplus alloc(d \cup \{p_0, \ldots, p_n\}, p_n, t_n) \\ & \quad \text{for } p_0, \ldots, p_n \notin d \text{ "fresh" pointers} \end{aligned}$$

◊

One could show that *new* returns a well formed heap, and two calls of *new*, with equivalent heaps, return two equivalent heaps.

Property 4. *Let H be a well formed heap, and let t be a type. If $(H', p) = new(H, t)$ then H' is a well formed heap.* □

Property 5. *Let H_1, H_2 be two heaps and t a type. If $new(H_1, t) = (H, p)$ and $new(H_2, t) = (H', p')$ then $(H, p) =_{st} (H', p')$.* □

Corollary 2. *Let H_1, H_2 be two well formed heaps and t a type. If $new(H_1, t) = (H, p)$ and $new(H_2, t) = (H', p')$ then H and H' are well formed.* □

Stack Our memory model uses also a *stack* for subprogram calls. This stack is manipulated implicitly by the inference rules and only one part called *frame* is manipulated explicitly.

Definition 10 (Frames). *A* frame *is a tuple in $\mathbb{F} \stackrel{\text{df}}{=} \mathbb{L}_\bot \times \mathbb{L} \times \mathbb{B}$, thus $\mathbb{F}$ is the set of all frames. For each frame $F \stackrel{\text{df}}{=} (l_{p,F},\ l_{c,F},\ \beta_F)$, we have:*

- $l_{p,F}$ *the label of a previous block (where we come from), or undefined;*
- $l_{c,F}$ *the label of the current block (where we are);*
- β_F *a LLVM binding, representing the current evaluation context.*

We widen the binding functional notation to the frames in order to access the binding, we note $F(x)$ *the binding* $\beta_F(x)$ *of* x *by* β_F. ◊

In matter of simplification, we always note for a frame F: $l_{p,F}$ its first component, $l_{c,F}$ its second component and β_F its third component. As for heaps we need operations to handle frames.

Definition 11 (Binding and frame binding overwriting). *The binding overwriting operation* $\oplus : \mathbb{B} \to \mathbb{B}$ *and the frame binding overwriting operation* $\oplus : \mathbb{F} \times \mathbb{B} \to \mathbb{F}$ *are defined as follows:*

$$(\beta \oplus \beta')(p) \stackrel{\text{df}}{=} \begin{cases} \beta'(p) & \text{if} \quad p \in dom(\beta') \\ \beta(p) & \text{if} \quad p \notin dom(\beta') \wedge p \in dom(\beta) \\ \text{undefined otherwise} \end{cases}$$

$$(l,\ l',\ \beta) \oplus \beta' \stackrel{\text{df}}{=} (l,\ l',\ \beta \oplus \beta')$$

◊

The operator $\oplus$ is used as well for the heap overwriting as for the (frame) binding overwriting because of the similar nature of the performed operation.

Property 6. *The binding overwriting operation* $\oplus$ *is associative.* □

Definition 12 (Structural equivalence 2). *Let* H, H' *be two heaps and* F, F' *two frames then the structural equivalence for heap-frame pairs is defined as:*

$$(H, F) =_{st} (H', F') \Leftrightarrow \forall x \in dom(\beta_F) \cup dom(\beta_{F'}),\ (H, F(x)) =_{st} (H', F'(x))$$

◊

The structural equivalence has been widened to pairs of heaps and frames noted $(H, F) =_{st} (H', F')$ for H, H' two heaps and F, F' two frames. Intuitively, we check that all data accessible from the frame bindings are structurally equivalent. This holds also for values in bindings since the heap equivalence has been defined on $\mathbb{D}$ as the identity check.

5.2 Inference rules

The operational semantics is defined for a fixed and immutable program P, it means that any new function, nor block cannot be created or modified during the execution. We introduce a new notation for computations, we note $\overline{\cdot}$ the result of a computation, for example $5 \stackrel{\text{df}}{=} \overline{2+3}$. The main object used in our rewriting rules are *configurations* which represent a state of the program during the execution.

Definition 13 (Configurations). *A configuration is a tuple* (seq, H, F)*, where:*

- *seq is a sequence of instructions;*
- *H is a heap;*
- *F is a frame.*

A configuration is noted $\big(\, seq \,\big)_{H,F}$. ◊

The set of inference rules for expressions is shown in figure 8; expressions are evaluated to values in a frame context. The set of inference rules for sequence and commands are shown in figure 9; sequences and commands are evaluated to other sequences in a heap-frame context.

One can remark the stacking up of frames in *pcall* and *fcall* rules, it corresponds to the frame replacement in the subsumption of these rules for evaluating the body of a subprogram.

This semantics mixes up small-step and big-step reductions. Indeed, the most of rules are small-step except for *pcall* and *fcall* rules. In these two rules, we link the computation to its result and we make a sequence of reductions in the rule subsumption.

6 Results

6.1 Interpreting data structures.

The link between Petri Nets and their implementation is formalised with a family of interpretation functions for all data structures.

Definition 14 (Interpretations). *An interpretation is a partial function which maps a pair formed of a heap and a pointer to a marking, a set of markings, a multiset of tokens or a single token depending on the interpreted object. The interpretations are noted* $[\![H, p]\!]^*$ *with* $H \in \mathbb{H}$*,* $p \in \mathbb{P} \cup \mathbb{D}$ *and where* $*$ *is an annotation describing the interpreted object. In case where p is a pointer, we suppose that the result depends only on the accessible data from p,* i.e.*,* $p{\downarrow_H}$*. Moreover every interpretation function have to hold the following requirement.* ◊

Requirement. *Let* H*,* H' *be two heaps,* $[\![\cdot, \cdot]\!]^*$ *the interpretation function, and* p*,* p' *two pointers or values. If* $(H, p) =_{st} (H', p')$ *then* $[\![H, p]\!]^* = [\![H', p']\!]^*$. ◊

As presented in section 4, we use data structures and functions in order to reach our goal, they are supposed given or produced by the Petri net compilation process. Each of these functions and data structures are specified or axiomatized by a formal interface. Mainly, it helps to ensure independence and modularity between components in a programmatic and formal way. Specifying an interface leads to define a set of primitives that hold certain set of derivations and interpretations. For example, let H be a heap and F a frame such that, $F(x_{mset}) = p_{mset}$ and is a pointer on a structure multiset of elements of type

$$\frac{F(x) = p \quad H(p) = (t, v)}{\left(\, load\ x \,\right)_{H,F} \rightsquigarrow \left(\, v \,\right)_{H}}\ load \qquad \frac{F(x) = p \quad p[i] \text{ défini}}{\left(\, gep\ x,\ 0,\ i \,\right)_{H,F} \rightsquigarrow \left(\, p[i] \,\right)_{H}}\ gep_0$$

$$\frac{\overline{F(x_1) + F(x_2)} = v}{\left(\, add\ x_1,\ x_2 \,\right)_{H,F} \rightsquigarrow \left(\, v \,\right)_{H}}\ add \qquad \frac{(H', p) = new(H, t)}{\left(\, alloc\ t \,\right)_{H,F} \rightsquigarrow \left(\, p \,\right)_{H \oplus H'}}\ alloc$$

$$\frac{\overline{F(x_1)\ op\ F(x_2)} = v \quad op \in \{=, \neq, <, \leq\}}{\left(\, icmp\ op,\ x_1,\ x_2 \,\right)_{H,F} \rightsquigarrow \left(\, v \,\right)_{H}}\ icmp$$

$$\frac{1 \leq i \leq n}{\left(\, phi\ (x_1, l_1), \ldots, (x_n, l_n) \,\right)_{H,(l_i,\ l_c,\ \beta)} \rightsquigarrow \left(\, F(x_i) \,\right)_{H}}\ phi$$

$$\frac{\begin{array}{c} f(\!|\, a_1, \ldots, a_n \,|\!) \in dom(P) \\ F_0 = (\bot,\ f(\!|\, a_1, \ldots, a_n \,|\!),\ \{a_1 \mapsto F(r_1), \ldots, a_n \mapsto F(r_n)\}) \\ \left(\, P(f(\!|\, a_1, \ldots, a_n \,|\!)) \,\right)_{H,F_0} \rightsquigarrow^* \left(\, v \,\right)_{H'} \end{array}}{\left(\, fcall\ f(r_1, \ldots, r_n) \,\right)_{H,F} \rightsquigarrow \left(\, v \,\right)_{H'}}\ fcall$$

$$\frac{}{\left(\, ret\ r \,\right)_{H,F} \rightsquigarrow \left(\, F(r) \,\right)_{H}}\ ret$$

Fig. 8. Rules for expressions.

$$\frac{\left(\, cmd \,\right)_{H,F} \rightsquigarrow \left(\, seq' \,\right)_{H',F'}}{\left(\, cmd;\ seq \,\right)_{H,F} \rightsquigarrow \left(\, seq';\ seq \,\right)_{H',F'}}\ seq$$

$$\frac{}{\left(\, skip;\ seq \,\right)_{H,F} \rightsquigarrow \left(\, seq \,\right)_{H,F}}\ seq_{skip}$$

$$\frac{}{\left(\, br\ l \,\right)_{H,(l_p,\ l_c,\ \beta)} \rightsquigarrow \left(\, P(l) \,\right)_{H,(l_c,\ l,\ \beta)}}\ branch_1$$

$$\frac{(\beta(r) = true \wedge l = l_1) \vee (\beta(r) = false \wedge l = l_2)}{\left(\, br\ r,\ l_1,\ l_2 \,\right)_{H,(l_p,\ l_c,\ \beta)} \rightsquigarrow \left(\, P(l) \,\right)_{H,(l_c,\ l,\ \beta)}}\ branch_2$$

$$\frac{\begin{array}{c} f(\!|\, a_1, \ldots, a_n \,|\!) \in dom(P) \\ F_0 = (\bot,\ f(\!|\, a_1, \ldots, a_n \,|\!),\ \{a_1 \mapsto F(r_1), \ldots, a_n \mapsto F(r_n)\}) \\ \left(\, P(f(\!|\, a_1, \ldots, a_n \,|\!)) \,\right)_{H,F_0} \rightsquigarrow^* \left(\, ret \,\right)_{H',F'} \end{array}}{\left(\, pcall\ f(r_1, \ldots, r_n) \,\right)_{H,F} \rightsquigarrow \left(\, skip \,\right)_{H',F}}\ pcall$$

$$\frac{\left(\, expr \,\right)_{H,F} \rightsquigarrow \left(\, v \,\right)_{H'}}{\left(\, x = expr \,\right)_{H,F} \rightsquigarrow \left(\, skip \,\right)_{H',F \oplus \{x \mapsto v\}}}\ assign$$

$$\frac{F(r_p) = p \quad H(p) = (t, d) \quad H' = \{p \mapsto t, F(r_{new})\}}{\left(\, store\ r_{new},\ r_p \,\right)_{H,F} \rightsquigarrow \left(\, skip \,\right)_{H \oplus H',F}}\ store$$

Fig. 9. Rules for sequences and commands.

t in H. In these conditions, the function add_{mset} that adds an element into a multiset containing objects of type t is specified by:

$$\left(\mathit{pcall}\ add_{mset}(x_{mset}, x) \right)_{H,F} \rightsquigarrow \left(\mathit{skip} \right)_{H \oplus H', F} \tag{1}$$

$$dom(H) \cap dom(H') \subseteq p_{mset}{\downarrow_H} \tag{2}$$

$$[\![H \oplus H', p_{mset}]\!]^{mset(t)} = [\![H, p_{mset}]\!]^{mset(t)} + \{[\![H, F(x)]\!]^t\} \tag{3}$$

The condition (1) describes the reduction, the condition (2) localises the modifications in the heap and the condition (3) explains the comportment in terms of Petri nets.

Moreover, the structures marking, *i.e.*, every implementation, have to hold two properties:

Requirement. (soundness) *Let H, H' be two heaps, F, F' two frames, p_{mrq} a pointer in $dom(H)$ on a structure marking, and p_s a pointer on a structure place in the pointed marking. If $p_s \in p_{mrq}{\downarrow_H}$, $[\![H, p_{mrq}]\!]^{mrq}(s) = [\![H, p_s]\!]^s$, $\left(seq \right)_{H,F} \rightsquigarrow \left(seq' \right)_{H \oplus H', F'}$ et $p_{mrq} \notin dom(H')$ then*

$$[\![H \oplus H', p_{mrq}]\!]^{mrq}(s) = [\![H \oplus H', p_s]\!]^s$$

$\Diamond$

Requirement. (separation) *Let p_{mrq} be a pointer on a structure marking in a heap H. If p_s and $p_{s'}$ are pointers on a two distinct places in this structure then we have $p_s{\downarrow_H} \cap p_{s'}{\downarrow_H} = \emptyset$.* $\Diamond$

The *soundness* property ensures that any modification made on a place with a pointer returned by get_s is reflected on the marking (not on a copy). The *separation* property ensures that places do not share memory, *i.e.*, when one modifies a place in the marking these modifications have no side effects on other places.

6.2 Insulation theorem.

One of the main results on the semantics is the insulation theorem. It allows to consider reductions in minimal contexts, then generalize them to more complex ones. The exposition of the theorem is long because we must address commands and expressions in the same time.

Theorem 7. (insulation). *Let seq be a sequence, H a heap and F a frame. In this context, if*

$$\left(seq \right)_{H,F} \rightsquigarrow^* \left(v' \right)_{H \oplus H'} \quad \textit{or, resp.,} \quad \left(seq \right)_{H,F} \rightsquigarrow^* \left(seq' \right)_{H \oplus H', F'}$$

then for all heap H_0 and all binding β_0 such that

(i) $dom(H) \cap dom(H_0) = \emptyset$
(ii) $dom(\beta) \cap dom(\beta_0) = \emptyset$

(iii) any assigned variable in seq (or one of its redex) is not in $dom(\beta_0)$

then on can make the same reduction in an environments widen by the heap H_0 *and the binding* β_0, i.e.*:*

$$\big(seq\big)_{H\oplus H_0,F\oplus\beta_0} \rightsquigarrow^* \big(v''\big)_{H\oplus H_0\oplus H''}$$
$$\textit{or, resp.,}\quad \big(seq\big)_{H\oplus H_0,F\oplus\beta_0} \rightsquigarrow^* \big(seq'\big)_{H\oplus H_0\oplus H'',F''\oplus\beta_0}$$

with H'' *a heap and* F'' *a frame such that:*

$$dom(H_0)\cap dom(H'') = \emptyset \tag{4}$$

$$\forall p \in dom(H),\quad (H',p) =_{st} (H'',p) \tag{5}$$

$$(H\oplus H',v') =_{st} (H\oplus H'',v'') \tag{6}$$

or, respectively, (4), (5) *and*

$$(F',H') =_{st} (F'',H'') \tag{7}$$

$$dom(\beta_0)\cap dom(\beta_{F''}) = \emptyset \tag{8}$$

$$dom(\beta_{F'}) = dom(\beta_{F''}) \tag{9}$$

Intuitively, the result (4) ensures that one does not modify the heap H_0, and so the evaluation of the sequence have no side effects. The result (5) ensures that H' is sound with H'', *i.e.* that every structure reachable from H which was modified during the reduction have been modified in H'' in the same way. The results (6) and (7) ensures that the reduction produces the same computation. The result (8) ensure that one does not modify a variable in $dom(\beta_0)$. The result (9) ensures that the same variables were added into the frames.

Proof. The proof is made by induction on derivation trees ordered by a lexicographic order with respect to the maximal height of the derivation trees during the reduction and the step count in the reduction.

- initial case:
$$\big(seq\big)_{H,F} \rightsquigarrow^* \big(seq'\big)_{H\oplus H',F'}$$
we have:
$$\big(seq\big)_{H\oplus H_0,F\oplus\beta_0} \rightsquigarrow^* \big(seq'\big)_{H\oplus H_0\oplus H'',F''\oplus\beta_0}$$
where $seq = seq'$, $H' = H'' = \emptyset$ and $F = F' = F''$. Structural equivalence consequences and equalities on domains are trivial.
- Inductive cases: we consider the first step during the reduction sequence. Indeed, if one prove that the property holds for the first step then we can conclude by the induction hypothesis since the remaining reduction is smaller (the remaining reduction sequence maximal height is less or equal but a step shorter).

1. *seq* case: If the first step is an application of the *seq* rule:

$$\frac{\left(\, cmd \,\right)_{H,F} \rightsquigarrow \left(\, seq' \,\right)_{H \oplus H', F'}}{\left(\, cmd;\ seq \,\right)_{H,F} \rightsquigarrow \left(\, seq';\ seq \,\right)_{H \oplus H', F'}}\ seq$$

   We apply the induction hypothesis on the rule subsumption, we have:

$$\left(\, cmd \,\right)_{H \oplus H_0, F \oplus \beta_0} \rightsquigarrow \left(\, seq' \,\right)_{H \oplus H_0 \oplus H'', F'' \oplus \beta_0}$$

   with a heap H'' and a frame F'' such that:

$$dom(H_0) \cap dom(H'') = \emptyset \tag{10}$$

$$\forall p \in dom(H), \quad (H', p) =_{st} (H'', p) \tag{11}$$

$$dom(\beta_0) \cap dom(\beta_{F''}) = \emptyset \tag{12}$$

$$dom(\beta_{F'}) = dom(\beta_{F''}) \tag{13}$$

$$(F', H') =_{st} (F'', H'') \tag{14}$$

   By reinjecting into the rule, we have:

$$\left(\, cmd;\ seq \,\right)_{H \oplus H_0, F \oplus \beta_0} \rightsquigarrow \left(\, seq';\ seq \,\right)_{H \oplus H_0 \oplus H'', F'' \oplus \beta_0}$$

   Structural equality and domain constraints are the same.
2. seq_{skip}: similar to the initial case.
3. $branch_1$ and $branch_2$ cases:

$$\frac{}{\left(\, br\ l \,\right)_{H,(l_p,l_c,\beta)} \rightsquigarrow \left(\, P(l) \,\right)_{H \oplus H', (l_c,l,\beta)}}\ branch_1$$

   where $H' = \emptyset$. We have

$$\frac{}{\left(\, br\ l \,\right)_{H \oplus H_0, (l_p,l_c,\beta) \oplus \beta_0} \rightsquigarrow \left(\, P(l) \,\right)_{H \oplus H_0 \oplus H'', (l_c,l,\beta) \oplus \beta_0}}\ branch_1$$

   where $H'' = \emptyset$, because

$$(l_p, l_c, \beta) \oplus \beta_0 = (l_p, l_c, \beta \oplus \beta_0)$$

$$(l_c, l, \beta) \oplus \beta_0 = (l_c, l, \beta \oplus \beta_0)$$

   Structural equality and domain constraints are trivial since $H' = H'' = \emptyset$ and $dom(\beta') = dom(\beta'') = dom(\beta)$.
   The $branch_2$ case is similar.
4. *pcall* case: If the first reduction step is *pcall*, we know that

$$\frac{\begin{array}{c} f(\!|\, a_1, \ldots, a_n \,|\!) \in dom(P) \\ F_0 = (\bot, f(\!|\, a_1, \ldots, a_n \,|\!), \{a_1 \mapsto F(r_1), \ldots, a_n \mapsto F(r_n)\}) \\ \left(\, P(f(\!|\, a_1, \ldots, a_n \,|\!)) \,\right)_{H,F_0} \rightsquigarrow^* \left(\, ret \,\right)_{H \oplus H', F_1} \end{array}}{\left(\, pcall\ f(r_1, \ldots, r_n) \,\right)_{H,F} \rightsquigarrow \left(\, skip \,\right)_{H \oplus H', F}}\ pcall$$

where $H' = \emptyset$. We build the conclusion by applying induction hypothesis on the subsumption (smaller derivation tree).

$$\frac{\begin{array}{c} f(\!|\, a_1, \ldots, a_n \,|\!) \in dom(P) \\ F_0 = (\bot, f(\!|\, a_1, \ldots, a_n \,|\!), \{a_1 \mapsto F(r_1), \ldots, a_n \mapsto F(r_n)\}) \\ \big(P(f(\!|\, a_1, \ldots, a_n \,|\!))\big)_{H\oplus H_0, F_0\oplus\emptyset} \rightsquigarrow^* \big(ret\big)_{H\oplus H_0\oplus H'', F_1\oplus\emptyset} \end{array}}{\big(pcall\, f(r_1, \ldots, r_n)\big)^{H\oplus H_0,}_{F\oplus\beta_0} \rightsquigarrow \big(skip\big)^{H\oplus H_0\oplus H'',}_{F\oplus\beta_0}}\ pcall$$

Structural equivalence and domain equalities are true because they were for the subsumption (induction hypothesis) and $F' = F'' = F \oplus \beta_0$. The *fcall* case is similar, the induction hypothesis provides conclusions for the expressions.

5. *assign* case: If the first reduction step is *assign*, we know that

$$\frac{\big(expr\big)_{H,F} \rightsquigarrow \big(v'\big)_{H\oplus H'}}{\big(x = expr\big)_{H,F} \rightsquigarrow \big(skip\big)_{H\oplus H', F\oplus\{x \mapsto v'\}}}\ assign$$

By applying the induction hypothesis (smaller derivation tree)

$$\big(expr\big)_{H\oplus H_0, F\oplus\beta_0} \rightsquigarrow \big(v''\big)_{H\oplus H_0\oplus H''}$$

where H'' is a heap such that:

$$dom(H_0) \cap dom(H'') = \emptyset \tag{15}$$

$$\forall p \in H, \quad (H', p) =_{st} (H'', p) \tag{16}$$

$$(H', v') =_{st} (H'', v'') \tag{17}$$

By reinjecting in the derivation we have:

$$\big(x = expr\big)_{H\oplus H_0, F\oplus\beta_0} \rightsquigarrow \big(skip\big)_{H\oplus H_0\oplus H'', F\oplus\beta_0\oplus\{x \mapsto v''\}}$$

The consequences (4) and (5) are true beacuse of the induction hypothesis consequences (15) and (16). Since x is not an assigned variable $x \notin dom(\beta_0)$, so β_0 and $\{x \mapsto v''\}$ commutes, and so $F \oplus \beta_0 \oplus \{x \mapsto v''\} = F \oplus \{x \mapsto v''\} \oplus \beta_0$ and $dom(F \oplus \{x \mapsto v''\}) \cap dom(\beta_0) = \emptyset$ which gives us the consequence (8). The consequence (9) is trivial because $dom(\beta') = dom(\beta'') = \{x\}$. It remains to show that $(F', H \oplus H') =_{st} (F'', H \oplus H'')$, *i.e.*. $(F \oplus \{x \mapsto v'\}, H \oplus H') =_{st} (F \oplus \{x \mapsto v''\}, H \oplus H'')$, showing that $(\{x \mapsto v'\}, H') =_{st} (\{x \mapsto v''\}, H'')$ is sufficient since (16) implies the result for F, but this result is a immediate consequence of (17).

6. *store* case:

$$\frac{F(r_p) = p \quad H(p) = (t, d) \quad H' = \{p \mapsto t, F(r_{new})\}}{\big(store\ r_{new},\ r_p\big)_{H,F} \rightsquigarrow \big(skip\big)_{H\oplus H', F'}}\ store$$

where $F = F'$, we build the conclusion:

$$\frac{\begin{array}{c}(F \oplus \beta_0)(r_p) = F(r_p) = p \qquad (H \oplus H_0)(p) = H(p) = (t, d) \\ H'' = \{p \mapsto t, F(r_{new})\}\end{array}}{\big(\, store\ r_{new},\ r_p \,\big)_{H \oplus H_0, F \oplus \beta_0} \rightsquigarrow \big(\, skip \,\big)_{H \oplus H_0 \oplus H'', F'' \oplus \beta_0}}\ store$$

where $F = F''$, because $dom(F) \cap dom(\beta_0) = \emptyset$ and $dom(H) \cap dom(H_0) = \emptyset$. Since $H' = H''$ and $F = F' = F''$, structural equalities and domain constraints are trivially held.

7. *load* case:

$$\frac{F(x) = p \quad H(p) = (t, v)}{\big(\, load\ x \,\big)_{H,F} \rightsquigarrow \big(\, v \,\big)_{H \oplus H'}}\ load$$

where $H' = \emptyset$, immediate

$$\frac{(F \oplus \beta_0)(x) = F(x) = p \quad (H \oplus H_0)(p) = H(p) = (t, v)}{\big(\, load\ x \,\big)_{H \oplus H_0, F \oplus \beta_0} \rightsquigarrow \big(\, v \,\big)_{H \oplus H_0 \oplus H''}}\ load$$

where $H'' = \emptyset$, because $dom(F) \cap dom(\beta_0) = \emptyset$ and $dom(H) \cap dom(H_0) = \emptyset$. Structural equalities and domain constraints are trivial.

8. gep_0 case:

$$\frac{F(x) = p \quad p[i]_H \text{ défini}}{\big(\, gep\ x,\ 0,\ i \,\big)_{H,F} \rightsquigarrow \big(\, p[i]_H \,\big)_{H \oplus H'}}\ gep_0$$

where $H' = \emptyset$. Since $p[i]_H$ is defined, we heve $p,\ p[i]_H \in H$, and we conclude with

$$\frac{(F \oplus \beta_0)(x) = F(x) = p \quad p[i]_{H \oplus H_0} = p[i]_H \text{ defined}}{\big(\, gep\ x,\ 0,\ i \,\big)_{H \oplus H_0, F \oplus \beta_0} \rightsquigarrow \big(\, p[i]_{H \oplus H_0} \,\big)_{H \oplus H_0 \oplus H''}}\ gep_0$$

where $H'' = \emptyset$, because $dom(H) \cap dom(H_0) = \emptyset$. The structural equalities and domain constraints are trivially held.

9. *icmp* and *add* cases:

$$\frac{\overline{F(x_1)\ op\ F(x_2)} = v \quad op \in \{=, \neq, <, \leq\}}{\big(\, icmp\ op,\ x_1,\ x_2 \,\big)_{H,F} \rightsquigarrow \big(\, v \,\big)_{H \oplus H'}}\ icmp$$

where $H' = \emptyset$, we conclude

$$\frac{\overline{(F \oplus \beta_0)(x_1)\ op\ (F \oplus \beta_0)(x_2)} = v \quad op \in \{=, \neq, <, \leq\}}{\big(\, icmp\ op,\ x_1,\ x_2 \,\big)_{H \oplus H_0, F \oplus \beta_0} \rightsquigarrow \big(\, v \,\big)_{H \oplus H_0 \oplus H''}}\ icmp$$

where $H'' = \emptyset$ because $(F \oplus \beta_0)(x_1) = F(x_1)$ et $(F \oplus \beta_0)(x_2) = F(x_2)$. Structural equalities and domain constraints are trivial.
The *add* case is similar.

10. *phi* case:

$$\frac{1 \leq i \leq n}{\big(\, phi\ (x_1, l_1), \ldots, (x_n, l_n)\,\big)_{H,(l_i,l_c,\beta)} \rightsquigarrow \big(\,\beta(x_i)\,\big)_{H \oplus H'}}\ phi$$

where $H' = \emptyset$, we have

$$\frac{1 \leq i \leq n}{\big(\, phi\ (x_1, l_1), \ldots, (x_n, l_n)\,\big)_{\substack{H \oplus H_0,\\ (l_i,l_c,\beta)\oplus\beta_0}} \rightsquigarrow \big(\,(\beta \oplus \beta_0)(x_i)\,\big)_{H \oplus H_0 \oplus H''}}\ phi$$

where $H'' = \emptyset$. By hypothesis $dom(\beta) \cap dom(\beta_0) = \emptyset$, which implies that $(\beta \oplus \beta_0)(x_i) = \beta(x_i)$, thus we have the same computation result (6). $H' = H'' = \emptyset$ and so (4) and (5) are trivial.

11. *alloc* case:

$$\frac{(H', p') = new(H, t)}{\big(\, alloc\ t\,\big)_{H,F} \rightsquigarrow \big(\, p'\,\big)_{H \oplus H'}}\ alloc$$

We build the conclusion:

$$\frac{(H'', p'') = new(H \oplus H_0, t)}{\big(\, alloc\ t\,\big)_{H \oplus H_0, F} \rightsquigarrow \big(\, p''\,\big)_{H \oplus H_0 \oplus H''}}\ alloc$$

by definition of *new* (section 5.1, property 5.), we have $(H', p') =_{st} (H'', p'')$, moreover $dom(H'') \cap dom(H \oplus H_0) = \emptyset$, and so $dom(H'') \cap dom(H_0) = \emptyset$. (4), (5) and (6) are true.

12. *ret* case: trivial because $(\beta \oplus \beta_0)(r) = \beta(r)$.

6.3 Correction of $\boldsymbol{fire_t}$

Theorem 8. (Correction of $\boldsymbol{fire_t}$). *Let M be a marking, H a heap and p_{mrq} a pointer on a structure marking such that $dom(H) = p_{mrq}{\downarrow_H}$ and $[\![H, p_{mrq}]\!]^{mrq} = M$. Let $\beta \stackrel{\text{df}}{=} \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ be a LLVM mode of the transition t, which implies that each v_i is a value or a pointer encoding a token (for the place s_i) and $[\![H, v_i]\!]^{t(s_i)} \in M(s_i)$. Let F be a frame such that $\beta_F \stackrel{\text{df}}{=} \beta \oplus \{x_{mrq} \mapsto p_{mrq}\}$. If*

$$M[t, \beta\rangle M' \quad \textit{and} \quad \big(\, fcall\ fire_t(x_{mrq}, x_1, \ldots, x_n)\,\big)_{H,F} \rightsquigarrow \big(\, p'_{mrq}\,\big)_{H \oplus H'}$$

then

$$[\![H \oplus H', p'_{mrq}]\!]^{mrq} = M' \quad \textit{and} \quad dom(H) \cap dom(H') = \emptyset$$

In order to achieve the proof we need two auxiliary lemmas: one for the token consumption and one for the token production.

Lemma 1. *Let H be a heap, p'_{mrq} a pointer on a structure marking such that $dom(H) = p'_{mrq}\downarrow_H$ and $[\![H, p'_{mrq}]\!]^{mrq} = M$. Let F be a frame such that $\beta_F \stackrel{\text{df}}{=} \{x'_{mrq} \mapsto p'_{mrq}, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ where all v_i are pointers or values of s_i places types and $[\![H, v_i]\!]^{t(s_i)} \in M(s_i)$ (for $1 \leq i \leq n$). We consider the seq sequence corresponding to the consumption of tokens in the $fire_t$ function:*

$$seq \stackrel{\text{df}}{=} \begin{pmatrix} x_{s_1} = fcall\ get_{s_1}(x'_{mrq}) \\ pcall\ rem_{s_1}(x_{s_1}, x_1) \\ \ldots \\ x_{s_n} = fcall\ get_{s_n}(x'_{mrq}) \\ pcall\ rem_{s_n}(x_{s_n}, x_n) \end{pmatrix}$$

If one reduces the sequence totally, i.e., *if*

$$\big(seq\big)_{H,F} \rightsquigarrow^* \big(skip\big)_{H\oplus H', F\oplus\beta}$$

then

$$\beta \stackrel{\text{df}}{=} \{x_{s_1} \mapsto p_{s_1}, \ldots, x_{s_n} \mapsto p_{s_n}\} \tag{18}$$

$$dom(H) \cap dom(H') \subseteq p_{s_1}\downarrow_H \cup \cdots \cup p_{s_n}\downarrow_H \tag{19}$$

$$[\![H \oplus H', p'_{mrq}]\!]^{mrq}(s_i) = M(s_i) - \{[\![H, v_i]\!]^{t(s_i)}\}\ pour\ i \in \{1, \ldots, n\} \tag{20}$$

Proof. All three consequences are proved simultaneously by induction on n using the specifications of *get* and *rem* functions as well as the *SSA* property. The *SSA* property is needed in the inductive case for building a suitable binding from the biding provided by the induction hypothesis. □

Lemma 2. *Let H be a heap and p'_{mrq} a pointer on a structure marking such that $dom(H) = p'_{mrq}\downarrow_H$ and $[\![H, p'_{mrq}]\!]^{mrq} = M$. Let F be a frame such that $\beta_F \stackrel{\text{df}}{=} \{x'_{mrq} \mapsto p'_{mrq}, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ where all v_i are pointers or values of s_i places types (for $1 \leq i \leq n$). We consider the seq sequence that corresponds to token production in the function $fire_t$:*

$$seq \stackrel{\text{df}}{=} \begin{pmatrix} x_{s'_1} = fcall\ get_{s'_1}(x'_{mrq}) \\ o_{s'_1} = fcall\ f_{s'_1}(x_1, \ldots, x_n) \\ pcall\ add_{s'_1}(x_{s'_1}, o_{s'_1}) \\ \ldots \\ x_{s'_m} = fcall\ get_{s'_m}(x'_{mrq}) \\ o_{s'_m} = fcall\ f_{s'_m}(x_1, \ldots, x_n) \\ pcall\ add_{s'_m}(x_{s'_m}, o_{s'_m}) \end{pmatrix}$$

If one reduces the sequence totally, i.e., *if* $\big(seq\big)_{H,F} \rightsquigarrow^* \big(skip\big)_{\substack{H\oplus H', \\ F\oplus\beta}}$ *then*

$$\beta \stackrel{\text{df}}{=} \left\{ \begin{array}{l} x_{s'_1} \mapsto p_{s'_1},\quad o_{s'_1} \mapsto \overline{f_{s'_1}(v_1, \ldots, v_n)}, \\ \ldots \\ x_{s'_m} \mapsto p_{s'_m}, o_{s'_m} \mapsto \overline{f_{s'_m}(v_1, \ldots, v_n)} \end{array} \right\} \tag{21}$$

$$dom(H) \cap dom(H') \subseteq p_{s'_1}{\downarrow_H} \cup \cdots \cup p_{s'_m}{\downarrow_H} \tag{22}$$

$$[\![H \oplus H', p'_{mrq}]\!]^{mrq}(s'_i) = M(s'_i) + \left\{ [\![H \oplus H', \overline{f_{s'_i}(v_1, \ldots, v_n)}]\!]^{t(s'_i)} \right\}_{pour\ 1 \leq i \leq m} \tag{23}$$

Proof (Correction $fire_t$). The proof of the function $fire_t$ uses the two lemmas: one for the token consumption, and one for the token production. Both proved by induction on the number of places in the structure marking. The final proof consists in applying the two previous lemmas.

Corollary 3. *Under the same hypothesis, the call of function $fire_t$ terminates.* □

Proof. Immediate consequence of the correction theorem. □

6.4 Correction of $succ_t$

Theorem 9. (Correction of $succ_t$). *Let F be a frame and p_{mrq} a pointer in a heap H such that $p_{mrq}{\downarrow_H} = dom(H)$, $[\![H, p_{mrq}]\!]^{mrq} = M$, $\beta_F(x_{mrq}) = p_{mrq}$ and $\beta_F(x_{set}) = p_{set}$. If*

$$\big(fcall\ succ_t(x_{mrq})\big)_{H,F} \rightsquigarrow^* \big(p_{set}\big)_{H \oplus H'} \quad and \quad [\![H, p_{set}]\!]^{set} = E$$

then

$$dom(H) \cap dom(H')$$

$$[\![H \oplus H', p_{set}]\!]^{set} = E \cup \{M' \mid \exists \beta,\ M[t, \beta\rangle M'\}$$

In order to achieve the proof we introduce a property that will serve as an invariant.

Property 10. *A pair (H, F) where H is a heap and F a frame respects the property P if the following conditions are true:*

- $p_{mrq} \in dom(H)$ *and* $[\![H, p_{mrq}]\!]^{mrq} = M$;
- $p_{set} \in dom(H)$ *and* $[\![H, p_{set}]\!]^{set}$ *is a set of markings ;*
- $F(x_{mrq}) = p_{mrq}$;
- $F(x_{set}) = p_{set}$.

The first lemma ensures that if the execution flow passes through a $header_{t,k}$ block then it will reach a $footer_{t,k+1}$ block, Moreover, it guaranties that we will execute any block annotated by an index greater than k.

Lemma 3. *Let n be the number of input places of t, let H be a heap and F a frame such that (H, F) satisfies P, then we have:*

$$\big(P(header_{t,n})\big)_{H,F} \rightsquigarrow^* \big(P(footer_{t,n+1})\big)_{H',F'}$$

where (H', F') *satisfies* P *and without executing any block with an index greater than* n, i.e.,

$$\left(P(header_{t,n})\right)_{H,F} \rightsquigarrow^* \left(P(block_{t,i})\right)_{H'',F''}$$

for $i > n$ *and block* $\in \{header, loop, body, footer\}$ *is not a strict prefix of the previous reduction.*

Proof. We proceed by induction on n with a second induction on the number of tokens remaining in the n^{th} place during the inductive case.

Corollary 4. *Under the same hypothesis, the call of function* $succ_t$ *terminates.*

Proof. The proof proceeds by reducing the body of the $succ_t$ function, checking that the heap and the frame verify the property P and applying the previous lemma.

The second lemma is used to show that we enumerate all combinations of tokens for the input places. It implies that we enumerate all modes of t.

Lemma 4. *Let* H *be a heap and* F *a frame such that* (H, F) *satisfies* P. *Let* $\beta = \{x_n \mapsto v_n, \ldots, x_1 \mapsto v_1\}$ *be a LLVM binding such that* $[\![H, v_i]\!]^{t(s_i)} \in M(s_i)$ *for* $1 \leq i \leq n$. *If*

$$\left(P(header_{t,n})\right)_{H,F} \rightsquigarrow^* \left(P(header_{t,0})\right)_{H',F'}$$

then $\beta \subseteq \beta_{F'}$ *and* (H', F') *satisfies* P. *Moreover, the execution flow did not reach any block indexed by* i *such that* $i > n$, i.e.,

$$\left(P(header_{t,n})\right)_{H,F} \rightsquigarrow^* \left(P(block_{t,i})\right)_{H'',F''}$$

for $i > n$ *and block* $\in \{header, loop, body, footer\}$ *is not a strict prefix of the previous reduction.*

Proof. The proof proceeds exactly like the previous lemma.

Proof (correction $succ_t$*).* The first result is showed by remarking that any pointer is used explicitly, thus it can be showed by using the specifications of called functions. The second result is showed in two steps:

$\supseteq$. This part is proved using two auxiliary lemmas:
- one that if the execution flow passes through a $header_{t,k}$ block then it will reach a $footer_{t,k+1}$ block, Moreover, it guaranties that we will execute any block annotated by an index greater than k;
- the second lemma is used to show that we enumerate all combinations of tokens for the input places, which implies that we enumerate all modes of t.

$\subseteq$. next we show that we only add the successor markings into the set, which gives the other inclusion. This part is proved by remarking that if a marking is added into the set, then it is added from the $body_{t,0}$ block. So we show that this block is executed if and only if the binding contains a mode of t.

Corollary 5. *Under the same hypothesis, the call of function* $succ_t$ *terminates.* □

7 Conclusion

We have shown how a Petri Net can be compiled, targeting a fragment of the LLVM language. This compilation process produces code that provides the primitives to compute the state space of the compiled Petri net model. Then we have defined formal semantics for the fragment of the LLVM language we use. To produce a readable and usable system of inference rules, we have defined a memory model based on heaps and stacks. Finally we presented two theorems proving the correction of the code generated by our compiler. The detailed proofs provided in [5] are quite long because they are very much detailed to improve our confidence into their correctness, but they are at the same time easy to follow.

Current and future work will address a generalisation of the presented approach to compile a wider variety of coloured Petri nets, in particular nets embedding annotation languages easier to use for the modeller than LLVM. We are also interested in particular in exploiting remarkable structures of Petri net models that allow to optimise the code generated by the compiler. Such optimisations also need to be formally proved; preliminary results about this can be found in [5]. A complementary aspect is to evaluate the performance of the state space generation, that is of course another important motivation, see [6].

References

1. J.O. Blench and S. Glesner. A formal correctness proof for code generation from ssa form in isabelle/hol. *In Proceedings der 3. Arbeitstagung Programmiersprachen (ATPS) auf der 34. Jahrestagung der Gesellschaft für Informatik*, 2004.
2. E. Clarke, A. Emerson, and J. Sifakis. Model checking: Algorithmic verification and debugging. *ACM Turing Award*, 2007. ⟨`http://www-verimag.imag.fr/~sifakis/TuringAwardPaper-Apr14.pdf`⟩.
3. ADT Coq/INRIA. The Coq proof assistant. ⟨`http://coq.inria.fr`⟩.
4. S. Evangelista. *Méthodes et outils de vérification pour les réseaux de Petri de haut niveau*. PhD thesis, CNAM, Paris, France, 2006.
5. L. Fronc. Analyse efficace des réseaux de Petri par des techniques de compilation. Master's thesis, MPRI, university of Paris 7, 2010.
6. L. Fronc and F. Pommereau. Optimizing the compilation of Petri nets models. Technical report, IBISC, University of Évry, 2011.
7. K. Jensen and L.M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009, ISBN 978-3-642-00283-0.
8. N. Kedar. Certifying model checkers. *CAV*, 2001.
9. C. Lattner. LLVM language reference manual. ⟨`http://llvm.org/docs/LangRef.html`⟩.
10. C. Lattner and al. The LLVM compiler infrastructure. ⟨`http://llvm.org`⟩.
11. C. Lattner and al. LLVM related publications. ⟨`http://llvm.org/pubs`⟩.
12. C. Lattner and al. LLVM users. ⟨`http://llvm.org/Users.html`⟩.
13. X. Leroy. Formal certification of a compiler back-end. *In 33rd symposium Principles of Programming Languages*, pages 42–54, 2006.
14. X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, pages 107–115, 2009.

15. C. Pajault and S. Evangelista. Helena: a high level net analyzer. ⟨`http://helena.cnam.fr`⟩.
16. L. Paulson, T. Nipkow, and M. Wenzel. The Isabelle proof assistant. ⟨`http://www.cl.cam.ac.uk/research/hvg/Isabelle`⟩.
17. C. Reinke. Haskell-coloured Petri nets. In *IFL'99*, volume 1868 of *LNCS*. Springer, 1999.
18. K.N. Verma, J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *ASIAN'00*, volume 1961 of *LNCS*. Springer, 2000.