Reasoning on Assembly Code using Linear Logic
Jérôme Vouillon

To cite this version:
Jérôme Vouillon. Reasoning on Assembly Code using Linear Logic. 2011. hal-00782149

HAL Id: hal-00782149
https://hal.science/hal-00782149
Preprint submitted on 29 Jan 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Reasoning on Assembly Code using Linear Logic

Jérôme Vouillon
CNRS, UMR 7126, PPS, Univ Paris Diderot, Sorbonne Paris Citè,
F-75205 Paris, France
jerome.vouillon@pps.jussieu.fr

Abstract

We present a logic for reasoning on assembly code. The logics are an extension of intuitionistic linear logic with greatest fixed points, pointer assertions for reasoning about the heap, and modalities for reasoning about program execution. One of the modalities corresponds to the step relation of the semantics of an assembly code interpreter. Safety is defined as the greatest fixed point of this modal operator. We can deal with first class code pointers, in a modular way, by defining an indexed model of the logic.

1. Introduction

We present a logic for reasoning on assembly code. The core of this logic is intuitionistic linear logic [GL87]. The multiplicative conjunction $P \otimes Q$ of linear logic corresponds to the separating conjunction $P \ast Q$ of separation logic [ORY01, Rey02]: it asserts that subformulas $P$ and $Q$ hold for disjoint parts of the heap. Likewise, the multiplicative unit $1$ corresponds to formula $\text{emp}$ that asserts that the heap is empty. These constructions make it possible to reason in a modular way about the heap.

A major difference with separation logic is that the assertions we consider pure are the assertions that hold for empty portions of the heap, rather than assertions that are independent of the machine state. These assertions can be characterised using the of course modality of linear logic: a proposition $P$ is pure when $P \vdash 1 P$. The contraction rule $!P \vdash !P \otimes !P$ and the weakening rule $!P \vdash \top$ show that pure assertions can be duplicated or ignored, at will. Note that we use the separating conjunction $\otimes$ here, rather than the additive conjunction $\&$ (corresponding to conjunction $\land$ in separation logic). We believe this is an advantage of this choice of pure assertions: one does not have to juggle with two distinct conjunction operators. In practice, we hardly use the additive conjunction.

In order to reason about machine states, we define a suitable model of the logic. The objects $w$ of this model are basically pairs of a machine state and a domain (a set of locations). The domain specifies which part of the heap we are currently focused on. We define a Kripke semantics [Kri63], hence we call these objects worlds. We write $w \models P$ to states that a world $w$ satisfies an assertion $P$. The model differs from usual models of programlogics in that we consider whole machine states rather than heap portions. The use of a Kripke semantics allows to enforce that two equivalent machine states (that differs only on parts of the heap that are outside the set of locations in focus) cannot be distinguished by the logic: we define a preorder $\preceq$ between states and the semantics of the logic should satisfy a persistence property: if $w \preceq w'$ and $w \models P$ then $w' \models P$.

As the machine state is a component of our worlds, we can define a modality $\Box$ to reason on program execution: the formula $\Box P$ asserts that we can make progress from current state and that we reach after one step a state satisfying formula $P$. Hoare triples can be encoded using this modality: the assertion $\{P\} c \{Q\}$ holds when:

$$\vdash \forall c, ([c;\xi] \otimes P \land (\zeta \otimes Q)),$$

where assertion $\xi$ states that the current sequence of instructions is $c$. (A slightly more complex interpretation of Hoare triples can also be adopted in order to validate the frame rule.)

The logic has greatest fixed points $\nu x, F(x)$ for monotone operators $F$. This makes it possible to define safety as the greatest fixed point of operator $\Box$: we define safe $\nu x, \Box x, \text{Ox}$. A program is safe if and only progress can be made while remaining safe. Besides, safe is the greatest post-fixed point of operator $\Box$, satisfying the rule below.

$$P \vdash \Box P$$

$$\frac{P \vdash \text{safe}}{\vdash P}$$

Indeed, given some initial world $w$, suppose we can find a proposition $P$ such that $P \vdash \Box P$. Then, we have a proof of progress and preservation: it is always possible to take another step, and after this step, the program still satisfies $P$. The greatest fixed point construction can also be used to enforce stronger program invariants: programs that satisfy $\nu x, \text{Ox}$. $P$ & $\text{Ox}$ are safe programs with invariant $P$.

Finally, in order to deal with first class code pointers, we define an indexed model $\{AM01, AAV03, Ahm04, AMRV07\}$. With such a model, one can define fixed points for operators that are not monotone, but rather contractive. One also has a modality $\Box$ that satisfies Gödel-Löb rule:

$$\Box \vdash P \vdash P$$

This rule can be read as an induction principle: 1 is the least pre-fixed point of $\Box$ (1 $\vdash P$ stands for $1 \vdash P$). In indexed models, the relation $\preceq$ is also used to enforce preservation properties: if a state satisfies a property, then we want the property to remain satisfied in the future if the part of the state we are considering remains unchanged.

To illustrate the logic, we show how it can be used to verify a destructive list-append function written in CPS. This example was initially suggested by Reynolds. [Rey02]. We use the assembly code version of Ni and Shao. [NS09].

We first present the core logic (Section 2). A simple machine is specified for illustration purposes (Section 3). The logic is extended with rules for reasoning on the heap and on program execution (Section 4). We build an indexed model, which makes it possible to deal with first class code pointers (Section 5). Finally, we show how the logic can be used to prove program soundness (Section 6).

2. The Logic

2.1 Syntax and Rules

The syntax of the core logic is given in Figure 1. It is an extension of intuitionistic linear logic [GL87] with the greatest fixed
### Intuitionistic linear logic

$$
\begin{align*}
P &\vdash P & P &\vdash Q & Q &\vdash R & P &\otimes Q & \vdash Q &\otimes P & P &\otimes (Q &\otimes R) &\vdash (P &\otimes Q) &\otimes R & P &\vdash P' & Q &\vdash Q' & P &\vdash P' & Q &\vdash Q' \\
\end{align*}
$$

$$
\begin{align*}
P &\otimes 1 &\vdash P & P &\otimes 1 &\vdash P & P &\vdash Q &\otimes 1 &\vdash P & P &\vdash Q &\otimes R & Q &\vdash R & P &\vdash R & Q &\vdash R & 0 &\vdash P & 0 &\vdash P & 0 &\vdash P \\
\end{align*}
$$

$$
\begin{align*}
!P &\vdash P & !P &\vdash !P & P &\vdash Q & !P &\vdash Q & !P &\vdash Q &!P &\vdash !P & P &\vdash !P & P &\vdash 1 & P &\vdash 1 & P &\vdash 1 & P &\vdash 1 \\
\end{align*}
$$

$$
\begin{align*}
P &\vdash x & x &\text{not free in } P & a &\in A & Q &\vdash P & x &\text{not free in } P & a &\in A & Q &\vdash P & a/x &\vdash P & a/x &\vdash P & a/x &\vdash P \\
\end{align*}
$$

### Greatest fixed points

$$
\begin{align*}
Q &\vdash P & Q &\vdash \nu x. P & \nu x. P &\vdash P &\nu x. P &\vdash P &\nu x. P &\vdash P \\
\end{align*}
$$

### Lifted meta propositions

$$
\begin{align*}
P &\text{implies } (1 &\vdash P) & (p &\vdash P) & 1 &\vdash (p) & (p &\vdash ! (p) \\
\end{align*}
$$

---

**Figure 3.** Inference rules.

---

$P := x$ \hspace{1cm} \text{variable}

$1$ \hspace{1cm} \text{one}

$P \otimes P$ \hspace{1cm} \text{multiplicative conjunction (times)}

$P \rightarrow P$ \hspace{1cm} \text{linear implication}

$0$ \hspace{1cm} \text{zero}

$P \& P$ \hspace{1cm} \text{additive disjunction (plus)}

$P$ \hspace{1cm} \text{of course}

$\forall x : A. P$ \hspace{1cm} \text{universal quantifier}

$\exists x : A. P$ \hspace{1cm} \text{existential quantifier}

$\nu x. P$ \hspace{1cm} \text{greatest fixed point}

$\langle p \rangle$ \hspace{1cm} \text{lifited meta proposition}

---

**Figure 1.** Grammar of the logic.

---

**Primitives**

$$
\begin{align*}
l &\mapsto v & \text{singleton heap assertion} & (\text{sect. 1.1)} \\
\text{code}(l, \varsigma) &\text{code assertion} & (\text{sect. 1.1)} \\
\n &\text{“next” modality} & (\text{sect. 1.1)} \\
\n &\text{“later” modality} & (\text{sect. 1.2)} \\
\mu x. P &\text{fixed point of} & (\text{sect. 1.3)} \\
\text{contractive operators} &
\end{align*}
$$

---

**Abbreviations**

$$
\begin{align*}
l &\mapsto \_ \equiv \exists x. l &\mapsto x & (\text{sect. 4.1)} \\
\varsigma &\equiv \text{pc }\mapsto \varsigma & (\text{sect. 4.1)} \\
\text{safe} &\equiv \nu x. \odot x & (\text{sect. 4.2)} \\
\end{align*}
$$

---

**Figure 2.** Additional constructions
Indeed, the usual definition \( \{ w | \forall w_1 \forall w_2, w_2 \approx w \cdot w_1 \Rightarrow w_1 \in [P_1]_\rho \Rightarrow w_2 \in [P_2]_\rho \} \) is not persistent in general. This is standard for a Kripke semantics. We have the equality \([P]_\rho = [P \cup \{ 1 \}]_\rho\), which can be found in some other models of linear logic [LS95]. In other words, a proposition is made pure by restricting it to units.

A consequence of persistence is that the logic cannot distinguish equivalent worlds, that is worlds \( w \) and \( w' \) such that \( w \leq w' \) and \( w' \leq w \). This is crucial for us, as it will make it possible to reason locally on heap portions, while considering worlds that contain whole machine states.

The semantics of sequents is defined as follows:
\[
[P \vdash Q]_\rho \triangleq [P]_\rho \subseteq [Q]_\rho.
\]

One can show that all the rules of Figure 3 are sound with this definition.

### 2.3 Building Separation Algebras

It is convenient to build complex separation algebras by combining simpler algebras. We propose several such constructions.

#### Trivial algebra.

We associate to a set \( W \) the separation algebra \( (W, \leq, \cdot) \) by taking \( U = W \), and \( w \approx w_1 \cdot w_2 \) when \( w = w_1 = w_2 \).

With this algebra, the two conjunctions \( \otimes \) and \& coincide, the assertion 1 holds for all worlds, and proposition \( \! P \) is equivalent to proposition \( P \).

#### Discrete algebra.

We associate to a set \( W \) the trivial separation algebra associated to the preorder set \( W = \approx \).

#### Algebra of subsets.

Given a set \( A \), we can define a separation algebra \( (W, \leq, \cdot, U) \) where
- \( W = \mathcal{P}(A) \),
- \( w \leq w' \) when \( w \subseteq w' \),
- \( w \approx w_1 \cdot w_2 \) when \( w = w_1 \cup w_2 \) and \( w_1 \cap w_2 = \emptyset \),
- \( U = \{ \emptyset \} \).

#### Product of two algebras.

Given two separation algebras \( (W_1, \leq_1, \cdot_1, U_1) \) and \( (W_2, \leq_2, \cdot_2, U_2) \), we can define a product separation algebra \( (W, \leq, \cdot, U) \) where
- \( W = W_1 \times W_2 \),
- \( (w_1, w_2) \leq (w'_1, w'_2) \) when \( w_1 \leq_1 w'_1 \) and \( w_2 \leq_2 w'_2 \),
- \( (w_1, w_2) \approx (w'_1, w'_2) \) when \( w_1 \approx_1 w'_1 \) and \( w_2 \approx_2 w'_2 \),
- \( U = U_1 \times U_2 \).

### 3. A Machine

In order to illustrate our logic, we specify a machine that interpret assembly code. The machine is that of Ni and Shao [NS06], except that we leave memory management functions free and alloc out of the definition of the machine. Instead, we later assume that they can be implemented and give axioms to type them. We assume given a set of registers. We write \( r \) for registers and \( i \) for machine words. The syntax of machine code is given in Figure 6.

An instruction sequence \( \zeta \) is a sequence of commands \( c \) followed by a jump.

A machine state is a quadruple \( s = (\mu, \rho, \kappa, \zeta) \) composed of a memory heap \( \mu \), a register file \( \rho \), a code heap \( \kappa \) and an instruction sequence \( \zeta \). A memory heap is a partial function from words to words. A register file is a total function from registers to words. A code heap is a partial function from words to instruction sequences.

The operational semantics is defined in two steps. The semantics of commands \( c \) is given in Figure 7. It specifies how the memory heap \( \mu \) and the registers file \( \rho \) are modified by the command.
Given a partial function \( f \) we write \( f(x) := v \) to denote, when \( f(x) \) is defined, the function that maps \( x \) to \( v \) and otherwise behaves as function \( f \); the expression \( f(x) := v \) is undefined when \( f(x) \) is undefined. Then, we define in Figure 5 the step relation \( \rightarrow \) between a machine state and the next. Note that it is illegal to access or modify unallocated parts of the memory heap, or to jump to a non-existing code label. The execution gets stuck in all these cases.

### 4. Machine Logic

#### 4.1 The Heap

We are going to define assertions for specifying the state of the machine. We handle in a uniform way the register file, the memory heap and the current instruction sequence. For that, we define locations \( l \) as either a register, a machine word, or the keyword \( pc \).

\[
\text{l} := r \mid i \mid \text{pc}
\]

Then, to a machine state \( s = (\mu, \rho, \kappa, \varsigma) \), we associate an abstract heap \( h = \text{heap}(s) \), a partial dependent function from locations \( l \) to values \( v \in \text{type}(l) \), which maps a register \( r \) to its value \( \rho(r) \), a memory location \( i \) to its contents \( \mu(i) \) if any, and the keyword \( \text{pc} \) to the current instruction sequence \( \varsigma \). We write \( \text{code}(s) \) for the code heap \( \kappa \) in machine state \( s \). Note that the machine state \( s \) is isomorphic to the pair \((\text{code}(s), \text{heap}(s))\).

We define worlds \( w \) as pairs of a machine state \( s \) and a set of locations \( d \). We define a separation algebra on worlds in a modular way by combining separation algebras on code heaps, on abstract heaps and on set of locations. For the sake of clarity, we state afterwards explicitly the resulting relations. Given a set of locations \( d \), we define an equivalence relation between heaps:

\[
h \equiv_d h' \quad \text{when} \quad \forall l \in d, h(l) = h'(l)\]

By the trivial algebra construction, this gives us a family of pre-ordered separation algebra \((H, \equiv_d, \cdot_d, \cup_d)\) on heaps. On the other hand, we have a canonical separation algebra \((D, =, \cdot_d, \cup_d)\) on sets of locations. We take a kind of dependent product of these two separation algebras:

- \( W = H \times D \);
- \( (h, d) \preceq (h', d') \) when \( h \equiv_d h' \) and \( d = d' \);
- \( (h, d) \preceq (h_1, d_1) \cdot (h_1, d_1) \) when \( h \equiv h_1 \cdot h_1 \) and \( d = d_1 \cdot d_2 \);
- \( U = U_H \times U_D \).

One can show that this defines a separation algebra. Finally, the separation algebra on worlds is the product of this algebra and of the discrete algebra on code heaps \( \kappa \).

This modular definition can be restated directly as follows. The accessibility relation is an equivalence relation. Two worlds

### Figure 5. Semantics of the logic.

**Commands**

\[
\begin{align*}
\text{add} & \quad: r, r, r & \text{add registers} \\
\text{add}i & \quad: r, r, i & \text{add register and word} \\
\text{mov} & \quad: r, r & \text{move between registers} \\
\text{mov} & \quad: r, i & \text{move word into register} \\
\text{ld} & \quad: r, i & \text{load} \\
\text{st} & \quad: r, i & \text{store} \\
\text{bgt} & \quad: r, r, i & \text{branch when greater than} \\
\text{bgti} & \quad: r, r, i & \text{branch when greater than word}
\end{align*}
\]

**Instruction sequences**

\[
\varsigma ::= \varsigma ; \varsigma \quad \text{sequence} \\
\text{jd} & \quad: i & \text{jump} \\
\text{jmp} & \quad: r & \text{computed jump}
\]

**Figure 6. Syntax of machine code.**

<table>
<thead>
<tr>
<th>Command ( c )</th>
<th>Outcome ((\mu', \rho'))</th>
</tr>
</thead>
<tbody>
<tr>
<td>add ( r_d, r_s, r_t )</td>
<td>((\mu, \rho, r_d) = (\rho(r_d) + \rho(r_t)))</td>
</tr>
<tr>
<td>addi ( r_d, r_s, i )</td>
<td>((\mu, \rho, r_d) = (\rho(r_d) + i))</td>
</tr>
<tr>
<td>mov ( r_d, r_s )</td>
<td>((\mu, \rho, r_d) = (\rho(r_s)))</td>
</tr>
<tr>
<td>movi ( r_d, i )</td>
<td>((\mu, \rho, r_d) = i)</td>
</tr>
<tr>
<td>ld ( r_d, r_s(i) )</td>
<td>((\mu, \rho, r_d) = \mu(\rho(r_s) + i))</td>
</tr>
<tr>
<td>st ( r_d(l), r_s )</td>
<td>((\mu, \rho(r_d) + l) = \rho(r_s))</td>
</tr>
</tbody>
</table>

**Figure 7. Command semantics \((\mu', \rho') \mapsto (\mu', \rho')\).**

<table>
<thead>
<tr>
<th>Sequence ( \varsigma )</th>
<th>Next state ((\mu', \rho', \kappa', \varsigma'))</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{jd } l \quad r_s \quad \text{bgt } r_s, r_l, l; \varsigma' \quad \text{jmp } r_s \quad \text{bgti } r_s, i, l; \varsigma' \quad c; \varsigma' \quad</td>
<td>((\mu, \rho, \kappa, (\rho(r))))</td>
</tr>
</tbody>
</table>

**Figure 8. Semantics \((\mu, \rho, \kappa, \varsigma) \mapsto (\mu', \rho', \kappa', \varsigma')\).**

Given a partial function \( f \), we write \( f(x) := v \) to denote, when \( f(x) \) is defined, the function that maps \( x \) to \( v \) and otherwise behaves as function \( f \); the expression \( f(x) := v \) is undefined when \( f(x) \) is undefined. Then, we define in Figure 5 the step relation \( \rightarrow \) between a machine state and the next. Note that it is illegal to access or modify unallocated parts of the memory heap, or to jump to a non-existing code label. The execution gets stuck in all these cases.

By the trivial algebra construction, this gives us a family of pre-ordered separation algebra \((H, \equiv_d, \cdot_d, \cup_d)\) on heaps. On the other hand, we have a canonical separation algebra \((D, =, \cdot_d, \cup_d)\) on sets of locations. We take a kind of dependent product of these two separation algebras:

- \( W = H \times D \);
- \( (h, d) \preceq (h', d') \) when \( h \equiv_d h' \) and \( d = d' \);
- \( (h, d) \preceq (h_1, d_1) \cdot (h_1, d_1) \) when \( h \equiv h_1 \cdot h_1 \) and \( d = d_1 \cdot d_2 \);
- \( U = U_H \times U_D \).

One can show that this defines a separation algebra. Finally, the separation algebra on worlds is the product of this algebra and of the discrete algebra on code heaps \( \kappa \).

This modular definition can be restated directly as follows. The accessibility relation is an equivalence relation. Two worlds
(s₁, d₁) and (s₂, d₂) are equivalent when they share the same code (\text{code}(s₁) = \text{code}(s₂)), they are focused on the same set of locations (d₁ = d₂), and the corresponding heap portion is shared (heap(s₁) = heap(s₂)). Two worlds (s₁, d₁) and (s₂, d₂) can be joined when they share the same machine state (s₁ = s₂) and are focused on disjoint set of locations (d₁ ∩ d₂ = ∅). The resulting world (s₁, d₁ ∪ d₂) is composed of the common machine state and the union of the locations. Units are worlds (s, ∅) with an empty set of locations.

We now define two assertions to specify the heaps. The first one holds when we are focused on a single location l and that location contains value v. The second one states that code location \( i \) contains instruction sequence \( c \).

\[
\begin{align*}
[l \mapsto v]_i & \triangleq \{ w \in W \mid \text{heap}(w)(l) = v \land \text{dom}(w) = \{l\} \} \\
[c(i, \varsigma)] & \triangleq \{ u \in U \mid \text{code}(u)(i) = \varsigma \}
\end{align*}
\]

The inference rules corresponding to these two assertions are given in Figure 2. Disjoint heap portions cannot have a location l in common. The assertion code(i, σ) is pure. There cannot be two distinct instruction sequences at the same location. It is convenient to define abbreviations for some common uses of the first assertion. We write \( l \mapsto x \) for \( \exists \varsigma : \text{type}(l). l \mapsto x \) and \( [c] \) for \( \text{pc} \mapsto c \).

4.2 “Next” Modality and Safety

In order to reason on program execution, we define a modal operator \( \Box \), which we call the “next” modality. The assertion \( \Box P \) holds when either the current execution has terminated successfully, or it is possible to perform a step and the world reached satisfies \( P \).

We have already defined the step relation \( \rightarrow \) for machine states. It can naturally be lifted to worlds: \( (s, d) \mapsto (s', d') \) when \( s \mapsto s' \) and \( d = d' \). We also define a success predicate that indicates whether the machine is in a successful termination state. We take \( \text{success}(w) = \emptyset \) for the moment, as our machine does not have a notion of success. We will have to adopt a different definition later on when considering indexed models (Section 5). This predicate can also be useful with other machines, for instance when a value is reached during the evaluation of a lambda term.

The step relation and the success predicate should satisfy three conditions. The success predicate should be persistent (10). The step relation should be deterministic (11) and compatible with the accessibility relation (12).

\[
\begin{align*}
(10) \quad \text{success}(w) & \implies \forall w' \exists w'' \text{ } w \rightarrow w'' \implies \text{success}(w') \\
(11) \quad w \rightarrow w' \implies \forall w'' \text{ } w \rightarrow w'' \implies w'' = w' \\
(12) \quad w₁ ≈ u \cdot w₂ \implies w₁ \rightarrow w₂' \implies \exists w₁', \exists u', \exists w₁' \cdot w₂' \wedge w₁ \mapsto w₁' \land u \leq u' 
\end{align*}
\]

One can define the product of two separation algebras \( (W₁, ≤₁, \cdot₁, U₁) \) and \( (W₂, ≤₂, \cdot₂, U₂) \) with associated step relations \( \rightarrow₁ \) and \( \rightarrow₂ \), and success predicates \( \text{success₁} \) and \( \text{success₂} \); we define \( (w₁, w₂) \mapsto (w₁', w₂') \) when \( w₁ \rightarrow₁ w₁' \) and \( w₂ \rightarrow₂ w₂' \) and \( \text{success₁}(w₁) \) and \( \text{success₂}(w₂) \) when either \( \text{success₁}(w₁) \) or \( \text{success₂}(w₂) \). The disjunction might look surprising, but it ensures that the pair \((w₁, w₂)\) is safe when worlds \( w₁ \) and \( w₂ \) are safe. One can check that the relation and the predicate satisfy conditions (10), (11) and (12). This construction is used to build indexed models (Section 5).

4.3 Instruction Types

The instruction types can be specified as axioms, listed in Figure 11. The axioms corresponding to instructions add, add1 and bgc have been omitted, as they are similar to other instructions. All these axioms are sound in our assembly code model. The type of commands e which do not change the flow of control follows the scheme:

\[ [c; \varsigma] \otimes R \otimes P \vdash \circ ([\varsigma] \otimes R \otimes Q) \]
two contiguous memory locations. The first location contains the reversed order. A list is either the machine word that appends destructively the elements of a list to another list, in We illustrate the logic by proving the soundness of a function intermediate step.

We consider the following tail y elements two lists of the first list and the second list, now in variable x.

\[
\begin{align*}
\text{ld } r_d, r_s(i) &; \mid \text{ld } r, r(i) &; \mid \text{mov } r_d, r_s &; \mid \text{mov } r, r &; \\
\text{st } r_d(i), r_s &; \mid \text{mov } r_d &; \mid \text{imp } r &; \mid \text{bgti } r, i, l; & \mid \text{free } r, n &; \mid \text{alloc } r, n &; \mid \text{freespace } r &; \mid \text{mem } r, i &; \mid \text{ld, st } &; \mid \text{mov, r } &; \mid \text{jmp } &; \mid \text{cont }\end{align*}
\]

This corresponds to a refined interpretation of Hoare triple \(\{ P \} c \{ Q \}\), which validate the frame rule:

\[
\frac{\{ P \} c \{ Q \}}{\{ P \circ R \} c \{ Q \circ R \}}
\]

The types of jump instructions are similar, except that the initial and subsequent sequences of instruction are unrelated. For the conditional branch instruction \texttt{bgti}, the two possible following states are specified by a disjunction.

In order to type memory allocation and release, we will assume the existence of sequences of instructions alloc and free. The type of these operations is given in Figure 11. An abstract datastructure, specified by an assertion freespace contains available free space. These operations respectively extract memory blocks from this space, and put them back. We specify memory blocks at address \(i\) of length \(n\) using the following inductive definition:

\[
\begin{align*}
\text{mem} \_ \text{block}(i, 0) & = 1 \\
\text{mem} \_ \text{block}(i, n + 1) & = i \mapsto _\_ \text{mem} \_ \text{block}(i + 1, n)
\end{align*}
\]

It takes several steps for these operations to terminate. An existentially quantified assertion \(\Delta\) specifies the machine state at each intermediate step.

\[\Pi = \text{code}(i_1, s_1) \circ \ldots \circ \text{code}(i_n, s_n)\]

where \(s\) ranges over all subsequences of instructions and \(P_i\) specifies the precondition required for executing sequence \(s\). The entry point of the function is given precondition:

\[P_\text{revapp} = \exists x. \exists y. \exists l. \exists t.\]

\[
\begin{align*}
P & \circ r_0 \mapsto x \circ r_1 \mapsto y \circ r_2 \mapsto _\_ \\
\text{list}(l, x) \circ \text{list}(l, y) & \circ (l = \text{rew } l + l + l))
\end{align*}
\]

We assume giving a specification \(l\) of the resulting list and an assertion \(P\) describing the part of the heap which is not modified by the function. The two lists are respectively in registers \(r_0\) and \(r_1\). Registers \(r_2\) is a temporary register that the function is allowed to use. We state that the returned list should be the first list reversed and appended to the second list. We have:

\[
\text{revapp} \circ P_{\text{revapp}} \vdash \Sigma
\]

When the function terminates, it jumps to location \texttt{cont}. The precondition for the sequence of instructions at this location is the following:

\[
P_{\text{cont}} = \exists y. P \circ r_0 \mapsto y \circ r_1 \mapsto _\_ \circ r_2 \mapsto _\_ \circ \text{list}(l, y)
\]

Register \(r_0\) contains the resulting list, which satisfies the specification given by \(l\). The part of the heap specified by assertion \(P\) and the two registers \(r_1\) and \(r_2\) are available.
We further assume that the program continuation is well-typed, that is, that we can find a proposition $\Delta$ such that:

$$\{\text{cont}\} \otimes P \text{cont} \vdash \Delta$$

$$\Pi \otimes \Delta \vdash \otimes \Delta.$$  

We can now perform the soundness proof. First, we prove for each sequence of instructions $\varsigma$ in the function above that:

$$\Pi \otimes (\varsigma \otimes P_s) \vdash \otimes (\varsigma \otimes (P_s \text{cont}))$$

Then, by combining this with the continuation soundness assumption, we get:

$$\Pi \otimes (\varsigma \otimes \Delta) \vdash \otimes (\varsigma \otimes \Delta).$$

Then, by constancy, we have:

$$\Pi \otimes (\varsigma \otimes \Delta) \vdash \Pi (\varsigma \otimes \Delta).$$

Finally, by definition of safety, we have:

$$\Pi \otimes (\varsigma \otimes \Delta) \vdash \text{safe}.$$  

In particular,

$$\Pi \otimes [\text{revapp}] \otimes P \text{revapp} \vdash \text{safe}.$$  

that is, if we are at the beginning of the function (the current sequence of instructions is $\text{revapp}$) and the heap satisfies the precondition $P \text{revapp}$, the program will execute safely.

This soundness proof can be done entirely using the given inference rules: there is no need to unfold definitions at any point. We prove safety, but it would be possible to prove stronger invariants by making appropriate assumptions on $\Sigma$ and $\Delta$. This proof scheme applies when the control flow is rather static: loops, simple functions. On the other hand, to deal with first class functions, we need a last ingredient...

5. Indexed Models

We now present how to build indexed models. The idea is to consider truncated execution traces. This is implemented by associating to each world an integer that is decremented at each step. Then it becomes possible to reason by induction of the length of these traces. As described in [AMRV07], we do not have to deal explicitly with indices in the logic. Instead, the logic is extended with, first, a modal operator $\Box$, called the “later” modality, and an associated induction principle, and second, a fixed point operator $\mu x. P$ for so-called contractive operators.

5.1 A Modality

We first define indexed models in an abstract way. They are characterized by the existence of a relation $\prec$ that satisfies the following conditions. The relation is stable with respect to the accessibility relation [13]. It is more precise than the accessibility relation [14]. It preserves joins [15]. The step relation is compatible with the relation [16]. Last, the relation is well-founded [17].

$$w \preceq w' \Rightarrow w' \prec w' \Rightarrow w \prec w'$$

$$w \prec w' \Rightarrow w \preceq w'$$

$$w \approx w_1 \cdot w_2 \Rightarrow w \prec w' \Rightarrow \exists w'_1, w'' \approx w'_1 \cdot w_2' \wedge w_1 \prec w'_1 \land w_2 < w'_2$$

$$w_1 \approx w \cdot w_2 \Rightarrow w_2 \preceq w'' \Rightarrow \exists w'_1, w'_2, w'_3 \approx w'_1 \cdot w'_2 \wedge w_1 \prec w'_1 \wedge u < w'$$

$$w_1 \prec w_1, \ldots, \prec w_n, \prec \ldots$$

The relation $\prec$ is transitive, by [13] and [14]. Conditions [16] and [17] imply the compatibility of the step relation with the accessibility relation [12]. An assertion $w \prec w$ intuitively means that world $w$ is a world strictly in the future with respect to world $w$, but otherwise satisfies the same properties [14]. We can define a simple separation algebra satisfying all these conditions by taking the trivial separation algebra corresponding to the preordered set $([N], \geq)$ and adopting the following definitions:

- $w \mapsto w'$ when $w = w' + 1$;
- $\text{success}(w)$ when $w = 0$;
- $w \prec w'$ when $w > w'$.

Furthermore, given two separations algebras $(W_1, \leq_1, \cdot_1, U_1)$ and $(W_2, \leq_2, \cdot_2, U_2)$ with associated step relations and success predicates, the second also having a suitable relation $\prec_2$, one can define a suitable relation $\prec$ on the product separation algebra (as defined in Section 4.2) by $(w_1, w_2) \prec (w'_1, w'_2)$ when $w_1 \leq w'_1$ and $w_2 < w'_2$.

Our model for assembly code can thus be turned into an indexed model by considering worlds $W \times N$ and lifting appropriately the definitions:

- $(w, n) \preceq (w', n')$ when $w \leq w'$ and $n \geq n'$;
- $(w, n) \approx (w_1, n_1) \cdot (w_2, n_2)$ when $w \approx w_1 \cdot w_2$ and $n = n_1 = n_2$;
- the units are $U \times N$;
- $(w, n) \mapsto (w', n')$ when $w \mapsto w'$ and $n = n' + 1$;
- $\text{success}(w, n)$ when $n = 0$;
- $(w, n) \prec (w', n')$ when $w \preceq w'$ and $n < n'$.

The semantics of the “later” modality associated to relation $\prec$ is defined in a standard way:

$$[\Box P]_\mu \triangleq \{w \in W | \forall w' \in W, w \prec w' \Rightarrow w' \in [P]_\mu\}.$$  

In particular, this is the same definition as in [AMRV07]. The set is indeed persistent, by condition [13].

The inference rules corresponding to this modality are listed in Figure 13. The first rule is standard for modalities. The second rule is a consequence of condition [14]. Then, there are four distributivity rules. The last two rules are especially interesting. The penultimate rule is Gödel-Löb rule, adapted to linear logic by inserting the “of course” modality in the hypothesis. This is the induction principle corresponding to condition [17]. The standard Gödel-Löb rule, given in the introduction, also holds, but is weaker. The following derived rule is useful when we have a pure invariant $!P$ and want to prove that it implies so proposition $Q$:

$$\frac{!P \otimes \Box Q \vdash Q}{!P \vdash Q}.$$  

The last rule is a stronger version of the constancy rule in Figure 10 consequence of condition [16]. Basically, it is sufficient for prop-
We follow the same approach for specifying assembly code as Ni and Shao [NS06]. The formalisation and proof techniques follows the work of Tan and Appel [Tan05, TA06, AMRV07]. We define typing judgements \( \Gamma \vdash \{ P \} : \rho \) for instruction sequences, \( \Gamma \vdash \{ P \} s \) for whole programs. The code is typed in a modular way. A code heap specification

\[ \Gamma = i_1 : P_1 ; \ldots ; i_n : P_n \]

defines the interface of a code heap. It states that the piece of code at location \( i_j \) has precondition \( P_j \). We adopt the following semantics for specifications:

\[ [\Gamma] \overset{syn}{\models} \exists \kappa_1, \ldots, \exists \kappa_n. \text{code}(i_1, \kappa_1) \otimes \ldots \otimes \text{code}(i_n, \kappa_n) \otimes \text{!}(\{ \kappa \} \otimes P) \rightarrow \text{safe} \]

That is, each location \( i_j \) contains an instruction sequence \( \kappa_j \), and whenever (! modality) in the future (\( \square \) modality) we reach one of these sequences and the associated precondition \( P_j \) is satisfied, the program executes safely thereafter.

The semantics of the typing judgment for instruction sequences \( \Gamma \vdash \{ P \} \kappa \) is:

\[ [\Gamma] \overset{syn}{\models} \exists \kappa. \text{code}(i, \kappa) \otimes \text{!}(\{ \kappa \} \otimes P) \rightarrow \text{safe} \]

that is, if the remainder of the program follows specification \( \Gamma \), when the program reaches the code sequence \( \kappa \) and precondition \( P \) is satisfied, it executes safely thereafter.

The semantics of the typing judgment for code heaps \( \Gamma \vdash \{ P \} : \Gamma' \) is defined, assuming \( \text{dom} (\kappa) = \text{dom}(\Gamma') \), by:

\[ [\Gamma] \overset{syn}{\models} [\Gamma] \overset{syn}{\models} \exists \kappa. [\Gamma] \vdash [\kappa] \rightarrow \text{safe} \]

where

\[ \Sigma_{\kappa, \Gamma'} \overset{syn}{\models} \bigoplus_{i \in \text{dom}(\kappa)} ([i\kappa(i)] \otimes \Gamma'(i)) \]

that is, if the remainder of the program follows specification \( \Gamma \), when the program reaches any of the code sequences \( i\kappa(i) \) and the corresponding precondition \( \Gamma'(i) \) is satisfied, it executes safely thereafter.

Finally, the semantics of the typing judgment for machine states \( \Gamma \vdash \{ P \} s \) is just a soundness statement:

\[ \exists d, \forall n \in \mathbb{N}, ((\{ \mu, \rho, \kappa, \varsigma \}, d), n) \vdash \text{safe} \]

The rules for these judgements are given in Figure 13. They can be easily proved in the logic. Regarding instruction sequence judgements, there is a consequence rule, elimination rules for existentials and meta propositions, a rule for single command executions, a rule to extract a code pointer from the specification \( \Gamma \), and three rules for flow control instructions. We have omitted rules for \# Ook and \# Free. Code heaps are typed in a modular way: there is one rule for typing a heap composed of a single instruction sequence, and a rule for combining code heaps. In this second rule, import interfaces \( \Gamma_1 \) and \( \Gamma_2 \) are allowed to overlap. Finally, the last rule states the soundness of a machine state.

The semantics of code pointers correspond to a specification with a single location:

\[ \text{codeptr}(i, P) \overset{syn}{\models} \exists \kappa. (\text{code}(i, \kappa) \otimes \text{!}(\{ \kappa \} \otimes P) \rightarrow \text{safe}) \]

Note that when \( (i : P) \in \Gamma \), we have \( [\Gamma] \vdash \text{codeptr}(i, P) \). This is how a code pointer can be extracted from the specification and

persistent set of worlds in general. This should be forced by using the persist operator.

6. Using the Logic

6.1 Typing Programs

The contractiveness property can either be a well-formedness condition of the syntax of propositions, or a side-condition of these propositions. In the second case, the definition above does not yield a
\[
\begin{align*}
\Gamma \vdash Q & \quad \Gamma \models \{Q\} \varsigma & \quad \text{x not free in } \Gamma \\
\varphi \ implies \ \Gamma \models \{P\} \varsigma & \quad \Gamma \models \{\varphi \land \{P\} \varsigma\} & \quad \Gamma \models \{\varphi \lor \{P\} \varsigma\} \\
(i : Q) \in \Gamma & \quad \Gamma \models \{P \land \text{codeptr}(i, Q)\} \varsigma & \quad \Gamma \models \{P \lor \text{codeptr}(i, Q)\} \varsigma \\
(j : Q) \in \Gamma & \quad \Gamma \models \{P \land \{v \land \{v > i\}\} \varsigma\} & \quad \Gamma \models \{P \lor \{v \land \{v > i\}\} \varsigma\} \\
(j : Q) \in \Gamma & \quad \Gamma \models \{P \land \{v \land \{v < i\}\} \varsigma\} & \quad \Gamma \models \{P \lor \{v \land \{v < i\}\} \varsigma\} \\
\end{align*}
\]

Figure 13. Derived rules for judgments \(\Gamma \models \{P\} \varsigma\), \(\Gamma \models \kappa : \Gamma'\) and \(\Gamma \models \{P\} s\).

append: \(\text{bgti } r0, 0, \text{else}\) else: \(\text{alloc } r3 \ 2\) \(k: \text{ld } r2, \text{r0}(0)\)
\(\text{ld } r31, r2(0)\) \(\text{st } r3(0), r0\)
\(\text{st } r3(1), r2\) \(\text{free } r0, 2\)
\(\text{free } r2, 2\) \(\text{ld } r0, r0(1)\)
\(\text{movi } r3, k\) \(\text{ld } r0, r3(1)\)
\(\text{jmp } r31\) \(\text{st } r3(0), r2(1)\) \(\text{mov } r1, r2\)
\(\text{st } r3(1), r3\) \(\text{free } r3, 2\)
\(\text{free } r3, 2\)

Figure 14. Destructive list append function in CPS.

\[
P_{\text{append}} \models \exists x. \exists y. \exists z. \exists t. \text{freespace} \land \text{r0} \rightarrow x \land r1 \rightarrow y \land r2 \rightarrow rk \land r3 \rightarrow \neg \land r31 \rightarrow \neg \land \text{list}(ls, x) \land \text{list}(lt, y) \land \text{clo}((l + h), rk) \\
\text{env} \models \text{envtype}. \exists \text{cnt}. \exists \text{env}. i \rightarrow \text{cnt} \land i + 1 \rightarrow \text{env} \land \text{envtype} \land \text{codeptr}(\text{cnt}, \text{cont}(\text{envtype}, ls)) \\
\text{envtype, env} \models \exists \text{env}. \exists z. \text{freespace} \land \text{r0} \rightarrow \text{env} \land r1 \rightarrow z \land r2 \rightarrow \neg \land r3 \rightarrow \neg \land r31 \rightarrow \neg \land \text{envtype}(\text{env}) \land \text{list}(ls, z)
\]

Figure 15. Precondition to the append function.

\[\Pi_{\kappa} \models \Sigma \rightarrow \text{safe}\].

By Gödel-Löb rule, we get

\[\Pi_{\kappa} \models \Sigma \rightarrow \text{safe}\].

In particular, the program entry point is safe if its precondition holds:

\[\Pi_{\kappa} \models ([k] \land P) \rightarrow \text{safe}\].

We have proved the soundness of the destructive list append function written in CPS from Ni and Shao [NS06] given in Figure 14. This function corresponds to the following piece of code due to Reynolds [Rey02].
let rec append(x, y, z) = 
  if x = nil then r(y) else 
  let a = [x] and b = [x+1] 
  and k(z) = let w = cons(a, z) in r(w) 
  in append(b, y, k)

The structure of our proof follows closely the proof by Ni and Shao. It is about 250 lines of Coq, compared to about 1700 lines for their proof. The reason is that we are able to work entirely in the logic rather than expanding definitions and reasoning at a lower level. We do not explain the proof here. We refer you to [NS06] for details. However, we find it interesting to give the append function preconditions (Figure [13]), as this gives an idea of what the logic can express.

Function append allocates and frees memory blocks and thus must have access to freespace. Registers r0 and r1 contains respectively the two lists x and y, which satisfy respectively some specifications ls and lt. Register r2 contains a pointer rk to the continuation closure. We assert that the continuation will be applied to the concatenation of the two lists (last assertion).

The continuation closure is a pair of two memory locations at address i. The first location contains a pointer cnt to the code of the continuation. The second contains a pointer enp to the continuation environment. The environment pointer should satisfy a predicate envtype. Assertion envtype(enp) typically specify the part of the memory heap that contains the environment. The code pointer assertion states that the continuation precondition should be satisfied in order to jump to the code at location cnt.

The precondition of the continuation state that it expects a pointer env to the environment in registers r0 and the resulting list z in register r1. The resulting list should satisfy the specification given by ls.

6.3 Proving Preservation Results

In Section 6.1 we have presented a way to prove program soundness. We show here how one can prove a stronger result: that a whole program satisfies some invariant I such that \( \Pi \otimes I \vdash (\otimes I) \), where \( \Pi \) specify that the program is loaded into memory. This gives us a partial-correctness result: invariant I remains satisfied after any number of step.

The idea is to replace safe by some indeterminate invariant \( I \) in the semantics of specification \( \Gamma \) and code pointers:

\[
\text{codeptr}_\Gamma(i, P) \equiv \exists \varsigma, \text{(code}(i, \varsigma) \otimes \text{core}((\varsigma \Downarrow P) \rightarrow I))
\]

The semantics of the typing judgment for instruction sequences \( \Gamma \models \{ P \} \varsigma \) becomes

\[
[\Gamma] \models \{ P \} \varsigma \quad \exists \Delta. \text{I}(\Delta(I) 
\rightarrow I) \rightarrow ((\varsigma \Downarrow P) \oplus \Delta(I)) \rightarrow (\otimes I)
\]

The existentially quantified predicate \( \Delta \) stands for the type of all subexpressions of instructions in sequence \( \varsigma \); we assume that the invariant is satisfied whenever we arrive in the future to any instruction in the sequence with the associated precondition satisfied; then we prove that if we are on any of these instruction and the corresponding precondition is satisfied, then we can take a step and then satisfy the invariant.

By typing the whole program, one finally get an equation of the shape:

\[
\Pi \otimes (\text{safe} \rightarrow \otimes I) \vdash F(I) \rightarrow \otimes I
\]

where \( F \) specifies the preconditions of all subsequences of instructions occurring in the program. There are two ways to continue from here. The first solution is to take \( I = \text{safe} \). Then, as \( \otimes \text{safe} \vdash \text{safe} \), we have:

\[
\Pi \otimes (\text{safe} \rightarrow \otimes \text{safe}) \vdash F(\text{safe}) \rightarrow \otimes \text{safe}.
\]

From this, we can conclude as in Section 6.1.

The other solution is to take \( I = \mu \varsigma. F(x) \). Operator \( F \) should indeed be contractive, as \( I \) only occurs inside a \( \Box \) modality in specifications and code pointers. Then, the hypothesis \( \Box(F(I) \rightarrow \otimes I) \) always holds, and we get:

\[
\Pi \vdash F(I) \rightarrow \Box(I).
\]

As \( I \) is a fixed point of \( F \), this can be rewritten into:

\[
\Pi \otimes I \vdash (\Box I).
\]

that is, \( I \) is an invariant of the program.

From this, we can prove soundness. As \( \Pi \) is pure, we can derive by constancy:

\[
\Pi \otimes I \vdash \Box(I).
\]

Finally, as safe is the greatest fixed point of \( \Box \),

\[
\Pi \otimes I \vdash \Box \text{safe}.
\]

7. Related Work

Separation logic [ORY01, Rey02] is the main starting point for this work. We use the same constructions (singleton heap, empty heap and separating conjunctions) to reason about the heap. However, as Spalding and Jia [SJ06], we use intuitionistic linear logic rather than the logic of bunched implications and we characterise pure propositions using the “of course” modality.

The code assembly machine and our main example are taken from the work by Ni and Shao on certifying programs with embedded code pointers [NS06]. We are able to use similar judgments, such as judgment \( \Gamma \models \{ P \} \varsigma \), to reason on programs. However, our judgments are syntactic sugar for logical assertions, rather than primitive constructs. Another difference if that we give an axiomatisation of our logic, which makes it possible to prove the soundness of programs without having to unfold any definition. This results in much smaller proofs, even with hardly any automation.

The \( \Box \) modality and the associated fixed point operator were introduced by Appel, Mellies, Richards and Vuillot [AMRV07]. We additionally have a \( \otimes \) modality to reason in a direct way on program execution. We similarly start from a logic with very atomic constructions from which we define high-level constructions as syntactic sugar. Our models are simpler as we do not deal with memory references.

Following Calcagno, O’Hearn and Yang [CY09], we define separation algebras to specify in an abstract way families of models for our logic. Our notion of separation algebra is significantly different as, first, we do not attempt to reason on concurrent programs and second, we adopt a Kripke semantics. Dockins, Hobor and Appel [DHA09, HDA10] have extended separation algebras to deal with indexed models. They also propose modular constructions for these separation algebras. Overall, we put weaker conditions on algebras.

8. Conclusion and Future Work

We have presented a logic, and associated models, for reasoning on assembly code. A novelty is the “next” modality \( \otimes \) that makes it possible to reason on program execution. Our definitions are modular and flexible. For instance, indexing can be lefted out. Given a machine whose state can be split into a heap and a constant part, it is straightforward to define a corresponding separation algebra (following Section 4.1).

Most of the results in this paper have been machine-checked in Coq. Modularity is currently achieved through the module system of Coq, which shows its limits. We plan to adopt the packaging techniques of Garillon, Gonthier, Mahboubi and Rideau [GMR09].
Though we have only very simple tactics so far, proofs are reasonably short. Still, more powerful tactics, for instance following the recent work by Gonthier, Ziliani, Nanevski and Dreyer [GZND11], would further help.

References


