Reasoning on Assembly Code using Linear Logic

Jérôme Vouillon

Reasoning on Assembly Code using Linear Logic

Jérôme Vouillon

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

Abstract

We present a logic for reasoning on assembly code. The logic is 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 modality 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 $P * Q$ of separation logic [ORY01, Rey02]: it asserts that subformulas P and Q hold for disjoint parts of the heap. Likewise, the multiplicative unit $\mathbf{1}$ correspond to formula **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 $P \vdash !P$. The contraction rule $!P \vdash !P \otimes !P$ and the weakening rule $!P \vdash \mathbf{1}$ 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 $\wedge$ 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 $w \Vdash P$ to states that a world w satisfies an assertion P. The model differs from usual models of program logics 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 $w \Vdash P$ then $w' \Vdash P$.

As the machine state is a component of our worlds, we can define a modality $\bigcirc$ to reason on program execution: the formula $\bigcirc P$ 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:

$$\vdash \forall \varsigma.\, ([c\,;\varsigma] \otimes P \multimap \bigcirc([\varsigma] \otimes Q)),$$

where assertion $[\varsigma]$ states that the current sequence of instructions is ς. (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 $\bigcirc$: we define $\mathrm{safe} \stackrel{\mathrm{syn}}{=} \nu x.\, \bigcirc x$. A program is safe if and only progress can be made while remaining safe. Besides, safe is the greatest post-fixed point of operator $\bigcirc$, satisfying the rule below.

$$\frac{P \vdash \bigcirc P}{P \vdash \mathrm{safe}}$$

Indeed, given some initial world w, suppose we can find a proposition P such that $P \vdash \bigcirc P$. 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 $\nu x.\, (P \,\&\, \bigcirc x)$ are safe programs with invariant P.

Finally, in order to deal with first class code pointers, we define an indexed model [AM01, AAV02, 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:

$$\frac{\Box P \vdash P}{\vdash P}$$

This rule can be read as an induction principle: $\mathbf{1}$ is the least prefixed point of $\Box$ ($\vdash P$ stands for $\mathbf{1} \vdash P$). 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 [NS06].

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

$$P \vdash P \qquad \frac{P \vdash Q \qquad Q \vdash R}{P \vdash R} \qquad P \otimes Q \vdash Q \otimes P \qquad P \otimes (Q \otimes R) \vdash (P \otimes Q) \otimes R \qquad \frac{P \vdash P' \qquad Q \vdash Q'}{P \otimes Q \vdash P' \otimes Q'}$$

$$P \otimes \mathbf{1} \vdash P \qquad \frac{P \otimes Q \vdash R}{P \vdash Q \multimap R} \qquad P \vdash P \oplus Q \qquad Q \vdash P \oplus Q \qquad \frac{P \vdash R \qquad Q \vdash R}{P \oplus Q \vdash R} \qquad \mathbf{0} \vdash P$$

$$P \vdash P \otimes \mathbf{1} \qquad \frac{P \vdash Q \multimap R}{P \otimes Q \vdash R} \qquad P \,\&\, Q \vdash P \qquad P \,\&\, Q \vdash Q \qquad \frac{R \vdash P \qquad R \vdash Q}{R \vdash P \,\&\, Q} \qquad P \vdash \top$$

$$!P \vdash P \qquad !P \vdash !!P \qquad \frac{P \vdash Q}{!P \vdash !Q} \qquad !P \otimes !Q \vdash !(P \otimes Q) \qquad \mathbf{1} \vdash !\mathbf{1} \qquad !P \vdash !P \otimes !P \qquad !P \vdash \mathbf{1}$$

$$\frac{P \vdash Q \qquad x \text{ not free in } P}{P \vdash \forall x{:}A.\, Q} \qquad \frac{a \in A}{\forall x{:}A.\, P \vdash P[a/x]} \qquad \frac{Q \vdash P \qquad x \text{ not free in } P}{\exists x{:}A.\, Q \vdash P} \qquad \frac{a \in A}{P[a/x] \vdash \exists x{:}A.\, P}$$

Greatest fixed points

$$\frac{Q \vdash P[Q/x]}{Q \vdash \nu x.\, P} \qquad \nu x.\, P \vdash P[\nu x.\, P/x]$$

Lifted meta propositions

$$\frac{p \text{ implies } (\mathbf{1} \vdash P)}{\langle p \rangle \vdash P} \qquad \frac{p}{\mathbf{1} \vdash \langle p \rangle} \qquad \langle p \rangle \vdash !\langle p \rangle$$

Figure 3. Inference rules.

P	::=	x	variable
		$\mathbf{1}$	*one*
		$P \otimes P$	multiplicative conjunction (*times*)
		$P \multimap P$	linear implication
		$\mathbf{0}$	*zero*
		$P \oplus P$	additive disjunction (*plus*)
		$\top$	*top*
		$P \,\&\, P$	additive conjunction (*with*)
		$!P$	*of course*
		$\forall x{:}A.\, P$	universal quantifier
		$\exists x{:}A.\, P$	existential quantifier
		$\nu x.\, P$	greatest fixed point
		$\langle p \rangle$	lifted meta proposition

Figure 1. Grammar of the logic.

Primitives

$l \mapsto v$	singleton heap assertion	(sect. 4.1)
$\mathrm{code}(i, \varsigma)$	code assertion	(sect. 4.1)
$\bigcirc P$	"next" modality	(sect. 4.2)
$\Box P$	"later" modality	(sect. 5)
$\mu x.\, P$	fixed point of contractive operators	(sect. 5)

Abbreviations

$l \mapsto \_$	$\stackrel{\text{syn}}{=}$	$\exists x.\, l \mapsto x$	(sect. 4.1)
$[\varsigma]$	$\stackrel{\text{syn}}{=}$	$\mathtt{pc} \mapsto \varsigma$	(sect. 4.1)
safe	$\stackrel{\text{syn}}{=}$	$\nu x.\, \bigcirc x$	(sect. 4.2)

Figure 2. Additional constructions

point operator of modal μ-calculus [BS06] (penultimate construction). The logic can be specified in a proof assistant using a shallow embedding. It is then very natural to lift arbitrary propositions from the meta-logic, such as equality predicates $x = y$, that do not involve the world (last construction). So far, we only have purely logical constructions. The logic will be extended in later sections with assertions and operators to deal specifically with programs. We list these additional constructions in Figure 2.

The greatest fixed point operator $\nu x.\, P$ only makes sense for monotonic formulas P, that is, formulas such that $[\![P]\!]_{\rho;x\mapsto S} \subseteq [\![P]\!]_{\rho;x\mapsto S'}$ whenever $S \subseteq S'$ (where $[\![P]\!]_\rho$ is the semantics of formula P, defined in Section 2.2). This can be enforced syntactically be demanding that all occurrences of variable x be in positive position in P, that is, one should always go through the right hand side of an implication an even number of times to reach each occurrence of variable x. When formalising the logic in a proof assistant, it may me more convenient to let the user write propositions $\nu x.\, P$ for arbitrary propositions P and rather add a side-condition ensuring monotonicity to the unfolding rule $\nu x.\, P \vdash P[\nu x.\, P/x]$.

The inference rules of the logic are given in Figure 3. We use symmetric sequents $P \vdash Q$ rather than the asymmetric sequents $P_1, \ldots, P_n \vdash Q$ of the traditional presentation of intuitionistic linear logic. Indeed, we find this presentation more convenient for mechanised proofs. Symmetric sequents define a preorder between propositions and we can use rewriting tactics that take advantage of this. The traditional presentation satisfies a cut elimination result, but this is not a property we are interested in. We write $\vdash P$ for $\mathbf{1} \vdash P$.

In linear logic, there are two choices to define *pure* propositions, that do not refer to the heap. One can either state that they are independent of the heap, or that they hold only on empty heap portions. The first choice is taken in separation logic. We prefer the second choice. Indeed, the pure proposition can then be characterised in a simple way, as the propositions P such that $P \vdash !P$. Besides, they can be combined with other propositions using the $\otimes$ operator. One can thus avoid the use of the $\&$ operator most of the time.

(1) $w \preceq w$

(2) $w \preceq w' \Longrightarrow w' \preceq w' \Longrightarrow w \preceq w''$

(3) $w \approx w_1 \bullet w_2 \Longrightarrow w \approx w_2 \bullet w_1$

(4) $w \approx w_1 \bullet w' \Longrightarrow w' \approx w_2 \bullet w_3 \Longrightarrow$
$\exists w'', w'' \approx w_1 \bullet w_2 \wedge w \approx w'' \bullet w_3$

(5) $\exists u, w \approx u \bullet w$

(6) $u \approx u \bullet u$

(7) $w \approx u \bullet w' \Longrightarrow w' \preceq w$

(8) $u \preceq w \Longrightarrow w \in U$

(9) $w \approx w_1 \bullet w_2 \Longrightarrow w \preceq w' \Longrightarrow$
$\exists w_1', \exists w_2', w' \approx w_1' \bullet w_2' \wedge w_1 \preceq w_1' \wedge w_2 \preceq w_2'$

Figure 4. Properties of separation algebras.

2.2 Models

To reason about machine states, we define models of the logic above. This can be done in a generic way by defining an abstract notion of *separation algebras*, as initiated by Calcagno, O'Hearn and Yang [COY07]. We use, though, a fairly different definition, in part because we define a Kripke model.

A separation algebra is a quadruple $(W, \preceq, \bullet, U)$ composed of a set of *worlds* W, an *accessibility* relation $\preceq \in W \times W$, a *join* relation $\bullet \in W \times W \times W$ and a set of *units* $U \subseteq W$. This quadruple should satisfy the conditions listed in Figure 4. We write w for worlds in W and u for units in U. Free variables in these conditions should be considered universally quantified. The accessibility relation $\preceq$ is used to define the Kripke semantics. It is a preorder, that is, it is reflexive (1) and transitive (2). The join relation and the set of units give us a model of linear logic. Following, Dockins, Hobor and Andrew [DHA09], the join relation $\bullet$ is defined as a 3-place relation, rather than a partially defined binary function. We write $w \approx w_1 \bullet w_2$ for $(w, w_1, w_2) \in \bullet$ to suggest that world w is in some way a combination of worlds w_1 and w_2. The operation can be partial: there is no guarantee that there exists a world w such that $w \approx w_1 \bullet w_2$; the notation also emphasises that the operation may not be functional: there may be several such worlds w. The join relation is commutative (3) and associative (4). Each element has a unit (5). Each unit is a unit for itself (6). Finally, the accessibility and join relations interact in different ways. When a world is combined with a unit, one gets a related world (7). (This condition is used to validate rule $P \otimes \mathbf{1} \vdash P$.) The accessibility relation preserves units (8) and joins (9).

Given a separation algebra, we can define the semantic of the logic as a function $[\![\_]\!]_\_$ that maps propositions P and environments ρ to so-called persistent sets of worlds (Figure 5). An environment is a sequence of bindings $x \mapsto v$. We say that a set of worlds $\mathcal{W} \subseteq W$ is *persistent* when it is upward closed with respect to the accessibility relation, that is, when for all worlds $w \in \mathcal{W}$, if $w \preceq w'$ then $w' \in \mathcal{W}$. The following operator extracts the persistent worlds in a set of worlds $\mathcal{W}$:

$$\operatorname{persist} \mathcal{W} \triangleq \{w \in W \mid \forall w' \in W, w \preceq w' \Rightarrow w' \in \mathcal{W}\}.$$

Though we do not formalise it here, we assume that environments are suitably kinded. In particular, they should be such that the sets $[\![x]\!]_\rho$ are indeed persistent. With this assumption, one can prove that the semantics is well-defined.

The semantics of most constructions is straightforward. There are two noteworthy points. The semantics of the linear implication $P_1 \multimap P_2$, must be explicitly restricted to persistent worlds. Indeed, the usual definition $\{w \mid \forall w_1 \forall w_2, w_2 \approx w \bullet w_1 \Rightarrow w_1 \in [\![P_1]\!]_\rho \Rightarrow w_2 \in [\![P_2]\!]_\rho\}$ is not persistent in general. This is standard for a Kripke semantics. We have the equality $[\![!P]\!]_\rho = [\![P \,\&\, \mathbf{1}]\!]_\rho$, which can be found in some other models of linear logic [LS96]. In other words, a proposition is made pure by restricting it to units.

A consequence of persistency is that the logic cannot distinguish equivalent worlds, that is worlds w and w' such that both $w \preceq w'$ and $w' \preceq w$. 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 preordered set $(W, \preceq)$ the separation algebra $(W, \preceq, \bullet, U)$ by taking $U = W$, and $w \approx w_1 \bullet w_2$ when $w = w_1 = w_2$.

With this algebra, the two conjunctions $\otimes$ and $\&$ coincide, the assertion $\mathbf{1}$ 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 preordered set $(W, =)$.

Algebra of subsets. Given a set A, we can define a separation algebra $(W, \preceq, \bullet, U)$ where

- $W = \mathcal{P}(A)$,
- $w \preceq w'$ when $w = w'$,
- $w \approx w_1 \bullet w_2$ when $w = w_1 \cup w_2$ and $w_1 \cap w_2 = \emptyset$,
- $U = \{\emptyset\}$.

Product of two algebras. Given two separation algebras $(W_1, \preceq_1, \bullet_1, U_1)$ and $(W_2, \preceq_2, \bullet_2, U_2)$, we can define a product separation algebra $(W, \preceq, \bullet, U)$ where

- $W = W_1 \times W_2$,
- $(w_1, w_2) \preceq (w_1', w_2')$ when $w_1 \preceq_1 w_1'$ and $w_2 \preceq_2 w_2'$,
- $(w_1, w_2) \approx (w_1', w_2') \bullet (w_1'', w_2'')$ when $w_1 \approx w_1' \bullet_1 w_1''$ and $w_2 \approx w_2' \bullet_2 w_2''$,
- $U = U_1 \times U_2$.

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 ς is a sequence of commands c followed by a jump.

A *machine state* is a quadruple $s = (\mu, \rho, \kappa, \varsigma)$ composed of a memory heap μ, a register file ρ, a code heap κ and an instruction sequence ς. 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 μ and the registers file ρ are modified by the command.

$$[\![\mathbf{1}]\!]_\rho \triangleq U \qquad [\![P_1 \otimes P_2]\!]_\rho \triangleq \{w \in W \mid \exists w_1, w_2 \in W, w \approx w_1 \bullet w_2 \wedge w_1 \in [\![P_1]\!]_\rho \wedge w_2 \in [\![P_2]\!]_\rho\}$$

$$[\![P_1 \multimap P_2]\!]_\rho \triangleq \mathrm{persist}\{w \in W \mid \forall w_1, w_2 \in W, w_2 \approx w \bullet w_1 \Rightarrow w_1 \in [\![P_1]\!]_\rho \Rightarrow w_2 \in [\![P_2]\!]_\rho\}$$

$$[\![\mathbf{0}]\!]_\rho \triangleq \emptyset \qquad [\![P_1 \oplus P_2]\!]_\rho \triangleq [\![P_1]\!]_\rho \cup [\![P_2]\!]_\rho \qquad [\![\top]\!]_\rho \triangleq W \qquad [\![P_1 \,\&\, P_2]\!]_\rho \triangleq [\![P_1]\!]_\rho \cap [\![P_2]\!]_\rho$$

$$[\![!P]\!]_\rho \triangleq [\![P]\!]_\rho \cap U \qquad [\![\forall x{:}A.\, P]\!]_\rho \triangleq \bigcap_{a \in A} [\![P]\!]_{\rho; x \mapsto a} \qquad [\![\exists x{:}A.\, P]\!]_\rho \triangleq \bigcup_{a \in A} [\![P]\!]_{\rho; x \mapsto a}$$

$$[\![\nu x.\, P]\!]_\rho \triangleq \bigcup \{\mathcal{W} \subseteq W \mid \mathcal{W} \text{ persistent} \wedge \mathcal{W} \subseteq [\![P]\!]_{\rho; x \mapsto \mathcal{W}}\} \qquad [\![x]\!]_\rho \triangleq \rho(x) \qquad [\![\langle p \rangle]\!]_\rho \triangleq \begin{cases} U & \text{if } [\![p]\!]_\rho \text{ holds} \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 5. Semantics of the logic.

Commands

c	::=	$\texttt{add}\, r, r, r$	add registers
		$\texttt{addi}\, r, r, i$	add register and word
		$\texttt{mov}\, r, r$	move between registers
		$\texttt{movi}\, r, i$	move word into register
		$\texttt{ld}\, r, r(i)$	load
		$\texttt{st}\, r(i), r$	store
		$\texttt{bgt}\, r, r, i$	branch when greater than
		$\texttt{bgti}\, r, i, i$	branch when greater than word

Instruction sequences

ς	::=	$c\,;\varsigma$	sequence
		$\texttt{jd}\, i$	jump
		$\texttt{jmp}\, r$	computed jump

Figure 6. Syntax of machine code.

Command c	Outcome (μ', ρ')
$\texttt{add}\, r_d, r_s, r_t$	$(\mu, \rho(r_d) {:=} \rho(r_s) + \rho(r_t))$
$\texttt{addi}\, r_d, r_s, i$	$(\mu, \rho(r_d) {:=} \rho(r_s) + i)$
$\texttt{mov}\, r_d, r_s$	$(\mu, \rho(r_d) {:=} \rho(r_s))$
$\texttt{movi}\, r_d, i$	$(\mu, \rho(r_d) {:=} i)$
$\texttt{ld}\, r_d, r_s(i)$	$(\mu, \rho(r_d) {:=} \mu(\rho(r_s) + i))$
$\texttt{st}\, r_d(i), r_s$	$(\mu(\rho(r_d) + i) {:=} \rho(r_s), \rho)$

Figure 7. Command semantics $(\mu, \rho) \overset{c}{\mapsto} (\mu', \rho')$.

Sequence ς	Next state $(\mu', \rho', \kappa', \varsigma')$
$\texttt{jd}\, l$	$(\mu, \rho, \kappa, \kappa(l))$
$\texttt{jmp}\, r_s$	$(\mu, \rho, \kappa, \kappa(\rho(r)))$
$\texttt{bgt}\, r_s, r_t, l\,;\varsigma'$	$(\mu, \rho, \kappa, \varsigma')$ when $\rho(r_s) \leq \rho(r_t)$; $(\mu, \rho, \kappa, \kappa(l))$ when $\rho(r_s) > \rho(r_t)$
$\texttt{bgti}\, r_s, i, l\,;\varsigma'$	$(\mu, \rho, \kappa, \varsigma')$ when $\rho(r_s) \leq i$; $(\mu, \rho, \kappa, \kappa(l))$ when $\rho(r_s) > i$
$c\,;\varsigma'$	$(\mu', \rho', \kappa, \varsigma')$ when $(\mu, \rho) \overset{c}{\mapsto} (\mu', \rho')$

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 8 the step relation $\mapsto$ 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`.

$$l ::= r \mid i \mid \texttt{pc}$$

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

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 explicitely 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 $(H, \equiv_d, \bullet_H, U_H)$ on heaps. On the other hand, we have a canonical separation algebra $(D, =, \bullet_D, U_D)$ 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 $d = d'$;
- $(h, d) \approx (h_1, d_1) \bullet (h_1, d_1)$ when $h \approx h_1 \bullet_H h_1$ and $d \approx d_1 \bullet_D d_2$;
- $U = U_H \times U_D$.

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 κ.

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

$$l \mapsto v \otimes l \mapsto v' \vdash \mathbf{0} \qquad \mathrm{code}(i,\varsigma) \vdash !\mathrm{code}(i,\varsigma')$$

$$\mathrm{code}(i,\varsigma) \otimes \mathrm{code}(i,\varsigma') \vdash \langle \varsigma = \varsigma' \rangle$$

Figure 9. Heap assertions.

(s_1, d_1) and (s_2, d_2) are equivalent when they share the same code ($\mathrm{code}(s_1) = \mathrm{code}(s_2)$), they are focused on the same set of locations ($d_1 = d_2$), and the corresponding heap portion is shared ($\mathrm{heap}(s_1) \equiv_{d_1} \mathrm{heap}(s_2)$). Two worlds (s_1, d_1) and (s_2, d_2) can be joined when they share the same machine state ($s_1 = s_2$) and are focused on disjoint set of locations ($d_1 \cap d_2 = \emptyset$). The resulting world $(s_1, d_1 \cup d_2)$ is composed of the common machine state and the union of the locations. Units are worlds $(s, \emptyset)$ 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 ς.

$$[\![l \mapsto v]\!]_\rho \triangleq \{w \in W \mid \mathrm{heap}(w)(l) = v \ \wedge \mathrm{dom}(w) = \{l\}\}$$

$$[\![\mathrm{code}(i,\varsigma)]\!]_\rho \triangleq \{u \in U \mid \mathrm{code}(u)(i) = \varsigma\}$$

The inference rules corresponding to these two assertions are given in Figure 9. Disjoint heap portions cannot have a location l in common. The assertion $\mathrm{code}(i,\varsigma)$ 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 $l \mapsto \_$ for $\exists x{:}\mathrm{type}(l).\, l \mapsto x$ and $[\varsigma]$ for $\mathtt{pc} \mapsto \varsigma$.

4.2 "Next" Modality and Safety

In order to reason on program execution, we define a modal operator $\bigcirc$, which we call the "next" modality. The assertion $\bigcirc P$ 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 $\mapsto$ 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 $\mathrm{success}(w) = \emptyset$ 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).

$$(10) \qquad \mathrm{success}(w) \Longrightarrow w \preceq w' \Longrightarrow \mathrm{success}(w')$$

$$(11) \qquad w \mapsto w' \Longrightarrow w \mapsto w'' \Longrightarrow w'' = w'$$

$$(12) \qquad w_1 \approx u \bullet w_2 \Longrightarrow w_2 \mapsto w_2' \Longrightarrow \exists w_1', \exists u', w_1' \approx u' \bullet w_2' \wedge w_1 \mapsto w_1' \wedge u \preceq u'$$

One can define the product of two separations algebras $(W_1, \preceq_1, \bullet_1, U_1)$ and $(W_2, \preceq_2, \bullet_2, U_2)$ with associated step relations $\mapsto_1$ and $\mapsto_2$, and success predicates $\mathrm{success}_1$ and $\mathrm{success}_2$: we define $(w_1, w_2) \mapsto (w_1', w_2')$ when $w_1 \mapsto_1 w_1'$ and $w_2 \mapsto_1 w_2'$ and $\mathrm{success}(w_1, w_2)$ when either $\mathrm{success}_1(w_1)$ or $\mathrm{success}_2(w_2)$. The disjunction might look surprising, but it ensures that the pair (w_1, w_2) is safe when worlds w_1 and w_2 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).

$$\frac{P \vdash Q}{\bigcirc P \vdash \bigcirc Q} \qquad \bigcirc P \,\&\, \bigcirc Q \vdash \bigcirc(P \,\&\, Q)$$

$$!P \otimes \bigcirc Q \vdash \bigcirc(!P \otimes Q)$$

Figure 10. Inference rule for "next" modality.

We then define the semantics of the "next" modality as follows:

$$[\![\bigcirc P]\!]_\rho \triangleq \mathrm{persist}\{w \in W \mid \mathrm{success}(w) \vee \exists w' \in W, w \mapsto w' \wedge w' \in [\![P]\!]_\rho\}.$$

Remark that we have to explicitly restrict ourselves to persistent worlds. This means that $\bigcirc P$ holds when we are able to perform a step and reach a world satisfying P not just starting from w but also from any world accessible from w (and in particular, from any equivalent world).

As the machine state is part of the world, it is straightforward to define the modality. That should also be possible with a standard presentation where the world only contains the constant part of the machine and a heap portion, by defining a Galois connection between set of worlds and set of machine states. But that would be much less natural.

There are three inference rules associated to the "next" modality (Figure 10). The first two are standard for modalities. The last one comes from conditions (10) and (12). It can be read as a constancy rule. If a pure property $!P$ holds, then it will still holds after taking a step. In particular, if one assume that $Q \vdash \bigcirc R$ then one can derive $!P \otimes Q \vdash \bigcirc(!P \otimes R)$. If one interpret Hoare triples $\{Q\}c\{R\}$ as

$$\vdash \forall \varsigma.\, ([c\,;\varsigma] \otimes Q \multimap \bigcirc([\varsigma] \otimes R)),$$

as in the introduction, this corresponds precisely to the constancy rule of Hoare logic:

$$\frac{\{Q\}c\{R\}}{\{!P \otimes Q\}c\{!P \otimes R\}}$$

We consider that a machine state is safe when after any number of steps, either a success state is reached, or a further step can be performed. We therefore adopt the following definition of the safety predicate.

$$\mathrm{safe} \stackrel{\mathrm{syn}}{=} \nu x.\, \bigcirc x$$

This definition is actually a bit more restrictive than the standard definition of safety, due to our definition of the "next" modality: the machine state is allowed to change in an unobservable way between each step.

It is also possible to define stronger notions of safety, where an invariant property P remains satisfied at each step, by refining the above definition to:

$$\mathrm{safe}_P \stackrel{\mathrm{syn}}{=} \nu x.\, (P \,\&\, \bigcirc x).$$

4.3 Instruction Types

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

$$[c\,;\varsigma] \otimes R \otimes P \vdash \bigcirc([\varsigma] \otimes R \otimes Q).$$

$$[\texttt{ld}\ r_d, r_s(i)\,;\varsigma] \otimes P \otimes r_s \mapsto l \otimes r_d \mapsto \_ \otimes l + i \mapsto v \vdash \bigcirc([\varsigma] \otimes P \otimes r_s \mapsto l \otimes r_d \mapsto v \otimes l + i \mapsto v)$$

$$[\texttt{ld}\ r, r(i)\,;\varsigma] \otimes P \otimes r \mapsto l \otimes l + i \mapsto v \vdash \bigcirc([\varsigma] \otimes P \otimes r \mapsto v \otimes l + i \mapsto v)$$

$$[\texttt{st}\ r_d(i), r_s\,;\varsigma] \otimes P \otimes r_s \mapsto v \otimes r_d \mapsto l \otimes l + i \mapsto \_ \vdash \bigcirc([\varsigma] \otimes P \otimes r_s \mapsto v \otimes r_d \mapsto l \otimes l + i \mapsto v)$$

$$[\texttt{mov}\ r_d, r_s\,;\varsigma] \otimes P \otimes r_s \mapsto v \otimes r_d \mapsto \_ \vdash \bigcirc([\varsigma] \otimes P \otimes r_s \mapsto v \otimes r_d \mapsto v) \qquad [\texttt{movi}\ r, i\,;\varsigma] \otimes P \otimes r \mapsto \_ \vdash \bigcirc([\varsigma] \otimes P \otimes r \mapsto i)$$

$$[\texttt{jmp}\ r] \otimes P \otimes r \mapsto i \otimes \mathrm{code}(i, \varsigma) \vdash \bigcirc([\varsigma] \otimes P \otimes r \mapsto i) \qquad [\texttt{jd}\ i] \otimes P \otimes \mathrm{code}(i, \varsigma) \vdash \bigcirc([\varsigma] \otimes P)$$

$$[\texttt{bgti}\ r, i, l\,;\varsigma] \otimes P \otimes r \mapsto v \otimes \mathrm{code}(l, \varsigma') \vdash \bigcirc(([\varsigma] \otimes P \otimes r \mapsto v \otimes \langle v \leq i \rangle) \oplus ([\varsigma'] \otimes P \otimes r \mapsto v \otimes \langle v > i \rangle))$$

$$\vdash \exists \Delta.\, (([\texttt{alloc}\ r, n\,;\varsigma] \otimes P \otimes \mathrm{freespace} \otimes r \mapsto \_) \oplus \Delta) \multimap \bigcirc(\Delta \oplus ([\varsigma] \otimes P \otimes \mathrm{freespace} \otimes \exists i.\, r \mapsto i \otimes \mathrm{mem\_block}(i, n)))$$

$$\vdash \exists \Delta.\, (([\texttt{free}\ r, n\,;\varsigma] \otimes P \otimes \mathrm{freespace} \otimes \exists i.\, r \mapsto i \otimes \mathrm{mem\_block}(i, n)) \oplus \Delta) \multimap \bigcirc(\Delta \oplus ([\varsigma] \otimes P \otimes \mathrm{freespace} \otimes r \mapsto \_))$$

Figure 11. Instruction types.

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

$$\frac{\{P\}c\{Q\}}{\{P \otimes R\}c\{Q \otimes R\}}$$

The types of jump instructions are similar, except that the initial and subsequent sequences of instruction are unrelated. For the conditional branch instruction `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 $\mathrm{freespace}$ 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:

$$\mathrm{mem\_block}(i, 0) = \mathbf{1}$$
$$\mathrm{mem\_block}(i, n+1) = i \mapsto \_ \otimes \mathrm{mem\_block}(i+1, n)$$

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

4.4 An Example

We illustrate the logic by proving the soundness of a function that appends destructively the elements of a list to another list, in reversed order. A list is either the machine word 0, or a pointer to two contiguous memory locations. The first location contains the first element of the list, and the second contains a pointer to the remainder of the list.

$$\mathrm{list}(\texttt{nil}, i) = \langle i = 0 \rangle$$
$$\mathrm{list}(v :: ls, i) = \exists j.\, \langle i \neq 0 \rangle \otimes i \mapsto v \otimes i + 1 \mapsto j \otimes \mathrm{list}(ls, j)$$

We consider the following `rev_append` function. It takes as arguments two lists `x` and `y`. If the list `x` is empty, then list `y` is returned. Otherwise, the first cell of list `x` is put at the beginning of list `y` (by setting its tail to `y`), and the function is called recursively with the tail `z` of the first list and the second list, now in variable `x`.

```
let rec revapp(x,y) =
  if x = nil then y else
    let z = [x+1]
    in [x+1] := y; revapp(z,x)
```

This corresponds to the following piece of assembly code.

```
revapp: bgti r0, 0, else
        mov r0, r1
        jd cont
 else:  ld r2, r0(1)
        st r0(1), r1
        mov r1, r0
        mov r0, r2
        jd revapp
```

The whole program can be described by an assertion Π of the following shape:

$$\Pi = \mathrm{code}(i_1, \varsigma_1) \otimes \ldots \otimes \mathrm{code}(i_n, \varsigma_n).$$

The type of the piece of assembly code above can be specified by defining an assertion:

$$\Sigma = \bigoplus_{\varsigma} ([\varsigma] \otimes P_\varsigma).$$

where ς ranges over all subsequences of instructions and P_ς specifies the precondition required for executing sequence ς. The entry point of the function is given precondition:

$$\begin{aligned} P_{\texttt{revapp}} = & \exists x.\, \exists y.\, \exists ls.\, \exists lt. \\ & P \otimes r_0 \mapsto x \otimes r_1 \mapsto y \otimes r_2 \mapsto \_ \otimes \\ & \mathrm{list}(ls, x) \otimes \mathrm{list}(lt, y) \otimes \langle l = \texttt{rev}\ \ ls +\!\!+ lt \rangle. \end{aligned}$$

We assume given 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:

$$[\varsigma_{\texttt{revapp}}] \otimes P_{\texttt{revapp}} \vdash \Sigma.$$

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

$$P_{\texttt{cont}} = \exists y.\, P \otimes r_0 \mapsto y \otimes r_1 \mapsto \_ \otimes r_2 \mapsto \_ \otimes \mathrm{list}(l, y).$$

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 Δ such that:

$$[\varsigma_{cont}] \otimes P_{cont} \vdash \Delta$$
$$\Pi \otimes \Delta \vdash \bigcirc\Delta.$$

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

$$\Pi \otimes ([\varsigma] \otimes P_\varsigma) \vdash \bigcirc(\Sigma \oplus ([\varsigma_{cont}] \otimes P_{cont})),$$

that is, if we are executing this sequence, we can take a step and either we arrive in Σ or we exit the function. From this, we deduce:

$$\Pi \otimes \Sigma \vdash \bigcirc(\Sigma \oplus ([\varsigma_{cont}] \otimes P_{cont})).$$

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

$$\Pi \otimes (\Sigma \oplus \Delta) \vdash \bigcirc(\Sigma \oplus \Delta).$$

Then, by constancy, we have:

$$\Pi \otimes (\Sigma \oplus \Delta) \vdash \bigcirc(\Pi \otimes (\Sigma \oplus \Delta).$$

Finally, by definition of safety, we have:

$$\Pi \otimes (\Sigma \oplus \Delta) \vdash \mathrm{safe}.$$

In particular,

$$\Pi \otimes [\varsigma_{revapp}] \otimes P_{revapp} \vdash \mathrm{safe},$$

that is, if we are at the beginning of the function (the current sequence of instructions is ς_{revapp}) and the heap satisfies the precondition P_{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 Σ and Δ. 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 characterised 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).

(13) $\quad w \preceq w' \Longrightarrow w' \prec w'' \Longrightarrow w \prec w''$

(14) $\quad w \prec w' \Longrightarrow w \preceq w'$

(15) $\quad w \approx w_1 \bullet w_2 \Longrightarrow w \prec w' \Longrightarrow$
$\exists w_1', \exists w_2', w' \approx w_1' \bullet w_2' \wedge w_1 \prec w_1' \wedge w_2 \prec w_2'$

(16) $\quad w_1 \approx u \bullet w_2 \Longrightarrow w_2 \mapsto w_2' \Longrightarrow$
$\exists w_1', \exists u', w_1' \approx u' \bullet w_2' \wedge w_1 \mapsto w_1' \wedge u \prec u'$

(17) $\quad$ There is no infinite chain:
$w_1 \prec w_1 \prec \ldots \prec w_n \prec \ldots$

The relation $\prec$ is transitive, by (13) and (14). Conditions (16) and (14) 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 (16) 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 $(\mathbb{N}, \geq)$ and adopting the following definitions:

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

Furthermore, given two separations algebras $(W_1, \preceq_1, \bullet_1, U_1)$ and $(W_2, \preceq_2, \bullet_2, U_2)$ 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 $w_1 \preceq w_1'$ and $w_2 \prec w_2'$.

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

- $(w, n) \preceq (w', n')$ when $w \preceq w'$ and $n \geq n'$;
- $(w, n) \approx (w_1, n_1) \bullet (w_2, n2)$ when $w \approx w_1 \bullet w_2$ and $n = n_1 = n_2$;
- the units are $U \times \mathbb{N}$;
- $(w, n) \mapsto (w', n')$ when $w \mapsto w'$ and $n = n' + 1$;
- $\mathrm{success}(w, n)$ 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:

$$[\![\Box P]\!]_\rho \triangleq \{w \in W \mid \forall w' \in W, w \prec w' \Rightarrow w' \in [\![P]\!]_\rho\}.$$

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 12. 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-

$$\frac{P \vdash Q}{\Box P \vdash \Box Q} \qquad P \vdash \Box P \qquad \Box P \otimes \Box Q \vdash \Box(P \otimes Q)$$

$$\Box P \,\&\, \Box Q \vdash \Box(P \,\&\, Q) \qquad \forall x{:}A.\, \Box P \vdash \Box \forall x{:}A.\, P$$

$$!\Box P \vdash \Box !P \qquad \frac{!\Box P \vdash P}{\mathbf{1} \vdash P} \qquad !\Box P \otimes \bigcirc Q \vdash \bigcirc(!P \otimes Q)$$

$$\mu x.\, P \vdash P[\mu x.\, P/x] \qquad P[\mu x.\, P/x] \vdash \mu x.\, P$$

Figure 12. Later modality and associated fixed point.

erty P to hold in the future for property P to hold after one step. We have the following derived rule, that nicely connects the modalities:

$$!\Box(P \multimap Q) \vdash \bigcirc P \multimap \bigcirc Q.$$

We use this rule to type code pointers.

5.2 Fixed Points

We are interested in having fixed points for so-called contractive propositions. We first define contractiveness. Then, we specify additional conditions on models neede to prove the existence of fixed points. Finally, we define a fixed point operator for contractive propositions.

In order to specify contractiveness, we first need to define the equivalence between two propositions:

$$P \Leftrightarrow Q \stackrel{\text{syn}}{=} !(P \multimap Q) \otimes !(Q \multimap P).$$

Then, a proposition P is *contractive* with respect to a variable x when the following sequent holds:

$$\vdash \forall y.\, \forall z.\, !\Box(y \Leftrightarrow z) \multimap (P[y/x] \Leftrightarrow P[z/x])$$

Intuitively, this holds when all occurences of variable y can be replaced by variable z in $P[y/x]$, by doing it as many time as necessary ("of course" modality), but only in the future ($\Box$ modality). A simple syntactic criterion that implies this property is that any occurrence of variable x in proposition P should be guarded by a $\Box$ modality.

In order to have fixed points, we need to make a further assumption on models. To each world w, one should be able to associate its *level* $|w| \in \mathbb{N}$. Levels should satisfy three conditions: levels decrease when following relation $\preceq$ (18); they decrease strictly when following relation $\prec$ (19); joining worlds preserves levels (20).

(18) If $w \preceq w'$, then $|w| \geq |w'|$.

(19) If $w \prec w'$, then $|w| > |w'|$.

(20) If $w \approx w_1 \bullet w_2$, then $|w| = |w_1|$.

Then, to define the semantics of the fixed point construction $\mu x.\, P$ in some environment ρ, we iteratively define a sequence $\mathcal{W}_n$ of sets of worlds:

$$\mathcal{W}_0 = \emptyset \qquad \mathcal{W}_{n+1} = [\![P]\!]_{\rho; x \mapsto \mathcal{W}_n}$$

Finally, we define:

$$[\![\mu x.\, P]\!]_\rho = \{w \in W \mid w \in \mathcal{W}_{|w|+1}\}$$

When P is contractive, this defines a persistent set of worlds that validates fixed point folding and unfolding (last two rules in Figure 12). We do not give a reasoning principle for this fixed point operator. The existence of a fixed point is enough for our needs.

The contractiveness property can either be a well-formedness condition of the syntax of propositions, or a side-condition of these rules. In the second case, the definition above does not yield a persistent set of worlds in general. This should be forced by using the persist operator.

6. Using the Logic

6.1 Typing Programs

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 $\Gamma \models \{P\}\, \varsigma$ for instruction sequences, $\Gamma \models \kappa : \Gamma'$ for code heaps, and $\Gamma \models \{P\}\, s$ 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:

$$\begin{aligned}[\![\Gamma]\!] \stackrel{\text{syn}}{=}\ & \exists \varsigma_1.\, \ldots \exists \varsigma_n. \\ & \text{code}(i_1, \varsigma_1) \otimes \ldots \otimes \text{code}(i_n, \varsigma_n) \otimes \\ & !\Box((([\varsigma_1] \otimes P_1) \oplus \ldots \oplus ([\varsigma_n] \otimes P_n)) \multimap \text{safe})\end{aligned}$$

That is, each location i_j contains an instruction sequence ς_j, and whenever (! modality) in the future ($\Box$ 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 $\Gamma \models \{P\}\, \varsigma$ is:

$$[\![\Gamma \models \{P\}\, \varsigma]\!] \stackrel{\text{syn}}{=} [\![\Gamma]\!] \vdash ([\varsigma] \otimes P) \multimap \text{safe},$$

that is, if the remainder of the program follows specification Γ, when the program reaches the code sequence ς and precondition P is satisfied, it executes safely thereafter.

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

$$[\![\Gamma \models \kappa : \Gamma']\!] \stackrel{\text{syn}}{=} [\![\Gamma]\!] \vdash \Sigma_{\kappa,\Gamma'} \multimap \text{safe}$$

where

$$\Sigma_{\kappa,\Gamma'} \stackrel{\text{syn}}{=} \bigoplus_{i \in \text{dom}(\kappa)} ([\kappa(i)] \otimes \Gamma'(i)),$$

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

Finally, the semantics of the typing judgment for machine states $\Gamma \models \{P\}\, s$ 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 Γ, and three rules for flow control instructions. We have omitted rules for `alloc` 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 Γ_1 and Γ_2 are allowed to overlap. Finally, the last rule states the soundess of a machine state.

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

$$\text{codeptr}(i, P) \stackrel{\text{syn}}{=} \exists \varsigma.\, (\text{code}(i, \varsigma) \otimes !\Box(([\varsigma] \otimes P) \multimap \text{safe})).$$

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

$$\frac{P \vdash Q \qquad \Gamma \models \{Q\}\,\varsigma}{\Gamma \models \{P\}\,\varsigma} \qquad \frac{\Gamma \models \{P\}\,\varsigma \qquad x \text{ not free in } \Gamma}{\Gamma \models \{\exists x{:}A.\, P\}\,\varsigma} \qquad \frac{p \text{ implies } \Gamma \models \{P\}\,\varsigma}{\Gamma \models \{P \otimes \langle p\rangle\}\,\varsigma} \qquad \frac{\Gamma \models \{Q\}\,\varsigma' \qquad [\varsigma] \otimes P \vdash \bigcirc([\varsigma'] \otimes Q)}{\Gamma \models \{P\}\,\varsigma}$$

$$\frac{(i : Q) \in \Gamma \qquad \Gamma \models \{P \otimes \mathrm{codeptr}(i, Q)\}\,\varsigma}{\Gamma \models \{P\}\,\varsigma} \qquad \frac{P \otimes r \mapsto i \otimes \mathrm{codeptr}(i, Q) \vdash Q}{\Gamma \models \{P \otimes r \mapsto i \otimes \mathrm{codeptr}(i, Q)\}\,\texttt{jmp}\ r} \qquad \frac{(i : P) \in \Gamma}{\Gamma \models \{P\}\,\texttt{jd}\ i}$$

$$\frac{(j : Q) \in \Gamma \qquad \begin{array}{c} P \otimes r \mapsto v \otimes \langle v > i\rangle \vdash Q \\ \Gamma \models \{P \otimes r \mapsto v \otimes \langle v \le i\rangle\}\,\varsigma \end{array}}{\Gamma \models \{P \otimes r \mapsto v\}\,\texttt{bgti}\ r, i, j\,;\varsigma} \qquad \frac{\Gamma \models \{P\}\,\varsigma}{\Gamma \models (i : \varsigma) : (i : P)} \qquad \frac{\Gamma_1 \models \kappa_1 : \Gamma_1' \qquad \Gamma_2 \models \kappa_2 : \Gamma_2'}{\Gamma_1\,;\Gamma_2 \models \kappa_1\,;\kappa_2 : \Gamma_1'\,;\Gamma_2'}$$

$$\frac{\Gamma \models \kappa : \Gamma \qquad \Gamma \models \{P\}\,\varsigma \qquad \exists d, \forall n \in \mathbb{N}, (((\mu, \rho, \kappa, \varsigma), d), n) \Vdash P}{\Gamma \models \{P\}(\mu, \rho, \kappa, \varsigma)}$$

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

```
append: bgti r0, 0, else
        ld r31, r2(0)
        ld r0, r2(1)
        free r2, 2
        jmp r31

else:   alloc r3 2
        st r3(0), r0
        st r3(1), r2
        ld r0, r0(1)
        alloc r2, 2
        st r2(1) r3
        movi r3, k
        st r2(0), r3
        jd append

k:      ld r2, r0(0)
        ld r3, r0(1)
        free r0, 2
        st r2(1), r1
        mov r1, r2
        ld r31, r3(0)
        ld r0, r3(1)
        free r3, 2
        jmp r31
```

Figure 14. Destructive list append function in CPS.

$$\begin{aligned} P_{\mathit{append}} &\overset{\text{syn}}{=} \exists x.\, \exists y.\, \exists \mathit{ls}.\, \exists \mathit{lt}.\, \text{freespace} \otimes r_0 \mapsto x \otimes r_1 \mapsto y \otimes r_2 \mapsto \mathit{rk} \otimes r_3 \mapsto \_ \otimes r_{31} \mapsto \_ \otimes \\ &\quad \mathrm{list}(\mathit{ls}, x) \otimes \mathrm{list}(\mathit{lt}, y) \otimes \mathrm{clos}((\mathit{ls} +\!+ \mathit{lt}), \mathit{rk}) \\ \mathrm{clos}(\mathit{ls}, i) &\overset{\text{syn}}{=} \exists \mathit{envtype}.\, \exists \mathit{cnt}.\, \exists \mathit{env}.\, i \mapsto \mathit{cnt} \otimes i + 1 \mapsto \mathit{env} \otimes \mathit{envtype}\ \mathit{env} \otimes \mathrm{codeptr}(\mathit{cnt}, \mathrm{cont}(\mathit{envtype}, \mathit{ls})) \\ \mathrm{cont}(\mathit{envtype}, \mathit{ls}) &\overset{\text{syn}}{=} \exists \mathit{env}.\, \exists z.\, \text{freespace} \otimes r_0 \mapsto \mathit{env} \otimes r_1 \mapsto z \otimes r_2 \mapsto \_ \otimes r_3 \mapsto \_ \otimes r_{31} \mapsto \_ \otimes \mathit{envtype}(\mathit{env}) \otimes \mathrm{list}(\mathit{ls}, z) \end{aligned}$$

Figure 15. Precondition to the append function.

added to the precondition. To type jump instructions, we rely on rule $!\Box(P \multimap Q) \vdash \bigcirc P \multimap \bigcirc Q$. Indeed, by instantiating P and Q, we get:

$$!\Box([\varsigma] \otimes P) \multimap \text{safe}) \vdash \bigcirc([\varsigma] \otimes P) \multimap \text{safe},$$

that is, if we have a code pointer to some instruction sequence ς with precondition P, the program jumps to the sequence ς (which takes one step), and the precondition is satisfied, then the program executes safely thereafter.

We explain the last rule of Figure 13. The specification of the code heap of a state $(\mu, \rho, \kappa, \varsigma)$ is:

$$\Pi_\kappa \overset{\text{syn}}{=} \bigotimes_{i \in \mathrm{dom}(\kappa)} \mathrm{code}(i, \kappa(i)).$$

From the two first premises,

$$[\![\Gamma]\!] \vdash (\Sigma_{\kappa,\Gamma} \oplus ([\varsigma] \otimes P)) \multimap \text{safe}.$$

We write $\Sigma = \Sigma_{\kappa,\Gamma} \oplus ([\varsigma] \otimes P)$. We can derive

$$\Pi_\kappa \otimes !\Box(\Sigma \multimap \text{safe}) \vdash [\![\Gamma]\!].$$

Hence,

$$\Pi_\kappa \otimes !\Box(\Sigma \multimap \text{safe}) \vdash \Sigma \multimap \text{safe}.$$

By Gödel-Löb rule, we get

$$\Pi_\kappa \vdash \Sigma \multimap \text{safe}.$$

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

$$\Pi_\kappa \vdash ([\varsigma] \otimes P) \multimap \text{safe}.$$

The third premise tell us that there exists a set of locations d and a natural number n such that:

$$(((\mu, \rho, \kappa, \varsigma), d), n) \Vdash P.$$

We also have:

$$(((\mu, \rho, \kappa, \varsigma), d), n) \Vdash \Pi_\kappa$$
$$(((\mu, \rho, \kappa, \varsigma), d), n) \Vdash [\varsigma].$$

By combining these, we finally get soundness:

$$(((\mu, \rho, \kappa, \varsigma), d), n) \Vdash \text{safe}.$$

6.2 An Example

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,r) =
  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 that 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 precondition (Figure 15), 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 r_0 and r_1 contains respectively the two lists x and y, which satisfy respectively some specifications *ls* and *lt*. Register r_2 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 *env* to the continuation environment. The environment pointer should satisfy a predicate *envtype*. Assertion *envtype*(*env*) 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 states that it expects a pointer *env* to the environment in registers r_0 and the resulting list z in register r_1. 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 $\mathcal{I}$ such that $\Pi \otimes \mathcal{I} \vdash \bigcirc\mathcal{I}$, where Π specify that the program is loaded into memory. This gives us a partial-correctness result: invariant $\mathcal{I}$ remains satisfied after any number of step.

The idea is to replace safe by some indeterminate invariant $\mathcal{I}$ in the semantics of specification Γ and code pointers:

$$\mathrm{codeptr}_{\mathcal{I}}(i,P) \stackrel{\text{syn}}{=} \exists\varsigma.\,(\mathrm{code}(i,\varsigma)\otimes !\Box((([\varsigma]\otimes P)\multimap \mathcal{I}))$$

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

$$[\![\Gamma \models \{P\}\,\varsigma]\!] \stackrel{\text{syn}}{=} [\![\Gamma]\!] \vdash \exists\Delta.\,!\Box(\Delta(\mathcal{I})\multimap \mathcal{I})\multimap (([\varsigma]\otimes P)\oplus\Delta(\mathcal{I}))\multimap \bigcirc\mathcal{I}$$

The existentially quantified predicate Δ stands for the type of all subsequences of instructions in sequence ς: 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:

$$\Pi\otimes !\Box(F(\mathcal{I})\multimap\mathcal{I})\vdash F(\mathcal{I})\multimap\bigcirc\mathcal{I}$$

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 $\mathcal{I} = \mathrm{safe}$. Then, as $\bigcirc\mathrm{safe}\vdash\mathrm{safe}$, we have:

$$\Pi\otimes !\Box(F(\mathrm{safe})\multimap\mathrm{safe})\vdash F(\mathrm{safe})\multimap\mathrm{safe}.$$

From this, we can conclude as in Section 6.1.

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

$$\Pi\vdash F(\mathcal{I})\multimap\bigcirc\mathcal{I}.$$

As $\mathcal{I}$ is a fixed point of F, this can be rewritten into:

$$\Pi\otimes\mathcal{I}\vdash\bigcirc\mathcal{I},$$

that is, $\mathcal{I}$ is an invariant of the program.

From this, we can prove soundness. As Π is pure, we can derive by constancy:

$$\Pi\otimes\mathcal{I}\vdash\bigcirc(\Pi\otimes\mathcal{I}).$$

Finally, as safe is the greatest fixed point of $\bigcirc$,

$$\Pi\otimes\mathcal{I}\vdash\mathrm{safe}.$$

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 assembly code 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, Melliès, Richards and Vouillon [AMRV07]. We additionally have a $\bigcirc$ 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[COY07], 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 $\bigcirc$ that makes it possible to reason on program execution. Our definitions are modular and flexible. For instance, indexing can be leaved 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 Garillot, Gonthier, Mahboubi and Rideau [GGMR09].

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

[AAV02] Amal Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *LICS '02*, June 2002.

[Ahm04] Amal J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, Princeton, NJ, Nov. 2004. Tech Report TR-713-04.

[AM01] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. on Programming Languages and Systems*, 23(5):657–683, Sept. 2001.

[AMRV07] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *POPL '07*, 2007.

[BS06] Julian Bradfield and Colin Stirling. Modal mu-calculi. In Patrick Blackburn, Johan van Benthem, and Frank Wolter, editors, *The Handbook of Modal Logic*. Elsevier, 2006.

[COY07] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS '07*, 2007.

[DHA09] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *The 7th Asian Symposium on Programming Languages and Systems*, 2009.

[GGMR09] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *TPHOLs '09*, 2009.

[GL87] J. Y. Girard and Y. Lafont. Linear logic and lazy computation. In *Theory and Practice of Software Development (TAPSOFT 87)*, 1987.

[GZND11] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. In *ICFP '11*, 2011.

[HDA10] Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A logical mix of approximation and separation. In *Asian Symposium on Programming Languages and Systems*, 2010.

[Kri63] Saul A. Kripke. Semantical considerations on modal logic. In *Proceedings of a Colloquium: Modal and Many Valued Logics*, vol. 16, pp. 83–94, 1963.

[LS96] Yves Lafont and Andre Scedrov. The undecidability of second order multiplicative linear logic. *Inf. Comput.*, 125, February 1996.

[NS06] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *POPL '06*, 2006.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic*, CSL '01, 2001.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02*, 2002.

[SJ06] Frances Spalding and Limin Jia. Asserting memory shape using linear logic. In *Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*, Jan. 2006.

[TA06] Gang Tan and Andrew W. Appel. A compositional logic for control flow. In *7th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'06)*, Jan. 2006.

[Tan05] Gang Tan. *A Compositional Logic for Control Flow and its Application to Foundational Proof-Carrying Code*. PhD thesis, Princeton University, Princeton, NJ, Aug. 2005. Tech Report CS-TR-731-05.