

# Verifying Separation for a Monadic Scheduler

Tom Harke

Department of Computer Science, Portland State University, Portland OR, USA  
harke@cs.pdx.edu

**Abstract.** In this paper we study separation proofs for schedulers written in a typed  $\lambda$ -calculus plus a monad. First we demonstrate separation proofs at a high level of reasoning for a variety of simple schedulers. Second, we lay the foundations for mechanizing these in the proof assistant Coq. We also argue why a proof of separation for a model extends to a proof of separation for code running on hardware.

## 1 Introduction

This paper demonstrates separation proofs at a high level of reasoning, for schedulers written in the  $\lambda$ -calculus plus a monad. It then lays the foundations for mechanizing these. The intent of this foundation is to allow proofs about more elaborate schedulers.

The separation problem is: given a scheduler managing many processes, does any one process have an undue influence upon another? If we focus on a particular process, called the *process of interest* (abbreviated POI) then an alternative phrasing of the problem is: when the POI runs interleaved with unrelated processes, does its behavior differ from when it runs alone? A more concrete phrasing depends on fixing parameters such as: the number of POIs, the specific scheduler, and whether the processes are non-deterministic. One instance of the problem is:

$$\text{LR } d \rightarrow \varrho \langle c, d \rangle \cong \varrho \langle c \rangle$$

where  $\varrho$  is a round-robin scheduler,  $\langle c, d \rangle$  and  $\langle c \rangle$  are queues of processes, the precondition on  $d$ , similar in spirit to Rushby's local respect [Rus92], says that atomic events associated with  $d$  are ignored by the equivalence relation  $\cong$ .

The separation problem comprises two questions. First, whether the components of the scheduler are "correct," for an appropriate notion of correctness. For the memory system, one simple notion of correctness is that distinct processes access disjoint regions of memory. Second, assuming correct parts, whether the scheduler maintains separation. We refer to this latter question as *the Assembly Problem*.

This paper focuses on the Assembly Problem for various schedulers, on the proof techniques needed to solve it, and how to mechanize these proofs in Coq. There is little detail here about the ongoing mechanization in Coq, but Coq's limitations influenced this presentation.

The context for this work is the *House* operating system [HJLT05]. House is written in a strongly typed pure functional language, Haskell, plus the *H monad* (or *hardware monad*). The H monad is intended to be a small, coherent interface, a more easily formalized alternative to Haskell’s IO monad. The type system rules out many sources of bugs, but it not strong enough to show separation.

### 1.1 Equivalences.

A central notion in the proofs is an equivalence relation between computations. The intuition behind it is that, if we could instrument the code to log memory accesses of the POI, we could detect an undue influence by a change in the logs. However, the presence of some accesses should have no effect: if LR  $d$ , then work done by  $d$  should be ignored by the equivalence.

There are a number of notions of equivalence in this paper. In most cases we use  $=$ , e.g. to express laws. There are three cases where a more explicit equivalence is used. The first is definitional equality,  $:=$ , where the name on the left is defined by the expression on the right. A name in a definition may have parameters and may involve pattern matching, as is common in Haskell. The second is the equivalence on OS schedulings,  $\cong$ , which is biased to ignore some actions. The third, an arbitrary equivalence relation on type  $A$  is denoted by  $\equiv_A$  (or  $\equiv$  when the type is clear). It will turn out that  $\cong$  is  $\equiv_{\mathbb{M}A}$ , for an appropriate monad  $\mathbb{M}$ .

### 1.2 Overview

The rest of the paper is as follows. §2 reviews background material: types, monads, sufficient completeness, and coinduction. §3 has a number of high level proofs of separation, assuming an adequate notion of equivalence. §4 discusses the preliminaries to mechanizing proofs: embedding a subset of Haskell into Coq, an applicative structure on setoids, details of a new monad used to implement that notion of equivalence, and a new monad used to allow non-terminating computation. §5 concludes.

## 2 Background

### 2.1 Types

Terms are classified by type with the ‘:’ (‘inhabits’ or ‘has type’) relation. For instance  $3 : \mathbb{N}$  asserts that 3 is a natural number.

There are a number of base types. Some are concrete: 1 is the type with one inhabitant,  $()$ ;  $\mathbb{N}$  is the usual type of natural numbers. Some are abstract:  $\mathbb{P}$  is a type of process identifiers, with decidable equality;  $\mathbb{C}$  is a type of process contexts, each of which has an associated process identifier accessed via  $ID : \mathbb{C} \rightarrow \mathbb{P}$ , and each of which holds enough information to pause and resume a computation. Other operations on these abstract types are introduced as needed.

There are five type constructions: products, sums, arrows, and least and greatest fixed points. The first three are binary: given types  $A$  and  $B$ , the product, sum, and arrow are denoted by  $A \times B$ ,  $A + B$  and  $A \rightarrow B$  respectively. Product terms are built by tupling, and taken apart with projection functions ( $\pi_1 : A \times B \rightarrow A$  and  $\pi_2 : A \times B \rightarrow B$ ). Sum terms are built with injection functions ( $\iota_1 : A \rightarrow A + B$  and  $\iota_2 : B \rightarrow A + B$ ) and taken apart with case analysis. One very useful sum is *Option*  $A := 1 + A$ , where *None* is often used in place of  $\iota_1 ()$ , and *Some*  $a$  for  $\iota_2 a$ . Arrow terms are built by  $\lambda$ -abstraction, and consumed by function application. We use the style of functional programmers, writing functions curried, instead of the fully applied style of mathematicians. That is, instead of  $f(a, b)$  we write  $f a b$ .

The last two constructors are fixed points. They build recursive types by taking a functor  $F$  describing one level of data and repeating it. For instance, when  $F X := 1 + \mathbb{N} \times X$ , then  $\mu X.1 + \mathbb{N} \times X$  is the type of finite lists of natural numbers, while  $\nu X.1 + \mathbb{N} \times X$  contains both finite and infinite lists. Terms from either fixed point can be built with  $F$ , or decomposed with case analysis. The least fixed point  $\mu X.F X$  creates inductive data, that is, terms built up by finitely many steps of  $F$ . The greatest fixed point  $\nu X.F X$  creates co-inductive data, that is, terms which permit finitely many decomposition steps along the structure of  $F$ .

## 2.2 Monads

Monads allow the integration of effectful computations with a pure language by using a type distinction to isolate pure terms from impure ones, by restricting access to impure terms, and by requiring sequencing of impure terms. State, exceptions and nondeterminism are effects that monads can model. We will introduce two more monads in §4.3 and §4.4: the first adds the ability to define custom program equivalences, the basis for  $\cong$ , and the second allows Coq to naturally model non-termination.

**Operations and Laws.** A monad is a triple:

$$\begin{aligned} \mathbb{M} &: Type \rightarrow Type \\ \eta &: \forall A. A \rightarrow \mathbb{M}A \\ \star &: \forall A B. \mathbb{M}A \rightarrow (A \rightarrow \mathbb{M}B) \rightarrow \mathbb{M}B \end{aligned}$$

satisfying the laws:

$$\begin{aligned} \text{associativity: } & (p \star q) \star r = p \star (\lambda a. qa \star r) \\ \text{left-identity: } & \eta a \star q = qa \\ \text{right-identity: } & p \star \eta = p \end{aligned}$$

$\mathbb{M}$  is a type function taking a *value* type to a *computation* type. For instance, if  $\mathbb{N}$  is the type of natural number values, then  $\mathbb{M}\mathbb{N}$  is the type of a computation that, when run, has *side effects* but ultimately produces a natural number. We'll

use the term *A-value* to denote a term from  $A$ , and *A-computation* to denote one from  $\mathbb{M}A$ . The polymorphic function  $\eta$  (read as ‘return’ or ‘eta’) has no effect: it simply promotes a value to a computation. The operator  $\star$  (‘bind’) sequences two computations: the effects of the first occur before those of the second, and the value computed by the first seeds the second.

The operator  $\gg$  (‘sequence’) is a specialization of  $\star$  which doesn’t pass a value:

$$\gg: \forall AB. \mathbb{M}A \rightarrow \mathbb{M}B \rightarrow \mathbb{M}B \qquad p \gg q := p \star \lambda..q$$

These operators are rough analogs to constructs in imperative languages. Bind is similar to semicolon (;), except for the value explicitly passed by the  $\lambda$ -binding. Sequence corresponds exactly to semicolon. Return is similar to an empty block {} in C, or `skip` in Algol, except for its value. The reason for the explicit value is that monads do not presume the presence of state; in contrast, in C the two subcomputation of ‘;’ communicate implicitly by writing to and reading from state. Extending the analogy, the associativity law justifies why procedural abstraction is valid.

A wide variety of effects (and many combinations) can be modelled with monads. Each effect has associated operations and laws. For instance one interface for a state monad, which silently threads a component of type  $S$ , has operations *put* to set the state, *get* to retrieve the state within a computation, and *run* to execute the computation (Figure ??). There are additional laws relating *put*, *get*, *run*,  $\star$  and  $\eta$ .

name	type
get	$\mathbb{M}S$
put	$S \rightarrow \mathbb{M}1$
run	$\mathbb{M}A \rightarrow S \rightarrow A \times S$

**Fig. 1.** State Monad Operations

run $(\eta a) s$	$= (a, s)$
run $(p \star q) s$	$= \text{uncurry } (\text{run } q) (\text{run } p s)$
run get $s$	$= (s, s)$
run (put $t) s$	$= ((), t)$

**Fig. 2.** State Monad Axioms

There is another formulation of monads as a 4-tuple  $(\mathbb{M}, \eta, \text{join}, \text{map})$  where  $\text{join} : \forall A. \mathbb{M}(\mathbb{M}A) \rightarrow \mathbb{M}A$  and  $\text{map} : \forall A B. (A \rightarrow B) \rightarrow (\mathbb{M}A \rightarrow \mathbb{M}B)$  with laws relating the operators [Mac71]. While this alternative is more natural in many contexts, the first formulation is more natural for imperative programming. The two formulations are equivalent using:

$$\text{join } m = m \star \text{id} \qquad \text{map } f \ m = m \star (\eta \circ f) \qquad m \star k = \text{join } (\text{map } k \ m)$$

### 2.3 Sufficient Completeness

The H-interface is a common interface to both actual hardware and a model implemented in the proof assistant Coq. In order to claim that a property of the

latter holds in the former we need a means to show that the observable behavior of the model, with respect to the interface, is the same as hardware. To do so we view the H-monad as an *abstract data type* (or ADT), and provide a *sufficiently complete* axiomatization of it [TM92][Chapter 12].

**Definition 2.1.** *When a new type is defined as an ADT it has associated operators. Each operator is classified as either a constructor, if its return type is the new type, or an accessor, if its return type is an existing type.*

**Definition 2.2.** *An axiomatization of an ADT is sufficiently complete if every well-formed ground term built with that ADT's operators either has the new type, or is equivalent, via the axioms, to a term from an existing type.*

Given a sufficiently complete axiomatization of an ADT and two implementations, the behaviors observable via accessors are identical.

**Queues.** Queues are used both to implement round robin schedulers, and as an example of a sufficiently complete specification.

Let  $\mathbb{Q}A$  denote the type of queues containing elements of type  $A$ . Figure 2 shows one possible set of queue operators: the four constructors are *empty* (abbreviated  $\langle \rangle$ ), *enqueue* ( $\triangleleft$ ), *append* ( $\dashv$ ), and *serve*, while the sole accessor is *next*. The axioms are in Figure 3. Informally we'll write  $\langle c_1, c_2 \dots c_n \rangle$  for  $\langle \rangle \triangleleft c_1 \triangleleft c_2 \dots \triangleleft c_n$ .

name	abbrev.	type
empty	$\langle \rangle$	$\mathbb{Q}A$
enqueue	$-\triangleleft-$	$\mathbb{Q}A \rightarrow A \rightarrow \mathbb{Q}A$
append	$-\dashv-$	$\mathbb{Q}A \rightarrow \mathbb{Q}A \rightarrow \mathbb{Q}A$
serve		$\mathbb{Q}A \rightarrow \mathbb{Q}A$
next		$\mathbb{Q}A \rightarrow \text{Option } A$

**Fig. 3.** Queue Operations

$p \dashv \langle \rangle$	$= p$
$p \dashv (q \triangleleft a)$	$= (p \dashv q) \triangleleft a$
serve $\langle \rangle$	$= \langle \rangle$
serve $(\langle \rangle \triangleleft a)$	$= \langle \rangle$
serve $(p \triangleleft a \triangleleft b)$	$= (\text{serve } (p \triangleleft a)) \triangleleft b$
next $\langle \rangle$	$= \text{None}$
next $(\langle \rangle \triangleleft a)$	$= \text{Some } a$
next $(p \triangleleft a \triangleleft b)$	$= \text{next } (p \triangleleft a)$

**Fig. 4.** Queue Axioms

There are well known techniques for showing sufficient completeness (e.g. [TM92][Chapter 12]). For queues we can build a *canonical term algebra* generated by the operators  $\langle \rangle$  and  $\triangleleft$ , then group the axioms as indicated in Figure 3 to define total functions on the canonical term algebra. A straightforward structural induction suffices to show that  $\dashv$  and *serve* map canonical queues (i.e. ones generated by  $\langle \rangle$  and  $\triangleleft$  and values of pre-existing types) to canonical queues, and that *next* maps canonical queues to ground terms.

**Monads.** We propose to apply sufficient completeness to monads as well.

Haskell’s IO monad was the ‘killer application’ which introduced monads to programmers. One key feature of IO is that there are no accessors, hence the containment of effects, but also the admission that one does not know how IO behaves.

Many other monads, however, do have accessors. Figure 1 shows a sufficiently complete axiomatization of a state monad with a single accessor, *run*. It may seem unintuitive, but *get* is not an accessor, due to its type.

### 3 High-Level Proofs

This section sketches proofs for various OS features. Throughout this section, we assume an equivalence on computations. Later, §4.3 shows how to build this equivalence.

**Hypothesis 3.1** *we have an equivalence relation,  $\cong$ , on computations. Moreover, it is a congruence with respect to bind ( $\star$ ), and it is co-inductive: that is, an equivalence may safely be used in its own proof provided that each side is unrolled at least once.*

The high level separation proofs use co-induction. This technique shows *observational* properties of potentially infinite objects; that is, it shows that there is no finitary evidence contradicting the property. The idea is: if one unfolding of a goal reveals a similar subgoal but produces no evidence that the goal is false, then no amount of unfolding can produce such evidence.

§3.1 defines a simple scheduler and proves two simple separation results. §3.2 presents a more realistic scheduler with timer interrupts. §3.3 is more realistic in an orthogonal way, modelling page faults. Each subsection also discusses primitive operations and assumptions presumed to be part of the H-interface. The term *high level* distinguishes these proofs from mechanized ones in Coq. High level proofs convey meaning to a mathematician, even if details are sketchy, while mechanized proofs are thorough, to the point of being tedious.

#### 3.1 Basic Round Robin

The round-robin scheduler,  $\rho : \mathbb{QC} \rightarrow \mathbb{M1}$ , repeatedly cycles through the processes in the queue. It is used in two problems. Solo/Duo separation is the simplest problem, illustrating the primitives and the reasoning. The  $n$ -process round-robin with one POI shows how the complexity scales with problem size.

**Hypothesis 3.2** *Assume an atomic step of execution  $\epsilon : \mathbb{C} \rightarrow \mathbb{MC}$  which leaves process identifier associated with a context invariant.*

The computation  $\epsilon$  plays two roles: first, it is the smallest computation managed by the scheduler, and second, it is the atomic step of machine execution.

**Definition 3.3 (Local Respect).** A process  $i : \mathbb{P}$  has local respect, denoted by  $\text{LR } i$ , iff a step of its execution is unobservable by the equivalence. Specifically:

$$\text{LR } i := \forall c. i = \text{ID } c \rightarrow \exists c'. \epsilon c \cong \eta c'$$

In addition, we overload the predicate  $\text{LR}$  as follows:  $\text{LR } c$ , for  $c : \mathbb{C}$ , means  $\text{LR } (\text{ID } c)$ , and  $\text{LR } \vec{c}$ , for  $\vec{c} : \mathbb{QC}$  means that for each  $c$  in  $\vec{c}$ ,  $\text{LR } c$ .

**Definition 3.4 (Round Robin Scheduler).**

$$\varrho \vec{c} := \begin{cases} \eta () & \text{if next } \vec{c} = \text{None} \\ \epsilon c \star \lambda c'. \varrho ((\text{serve } \vec{c}) \triangleleft c') & \text{if next } \vec{c} = \text{Some } c \end{cases}$$

Informally, for a nonempty queue this definition amounts to:

$$\varrho \langle c_1, c_2 \dots c_n \rangle := \epsilon c_1 \star \lambda c'_1. \varrho \langle c_2 \dots c_n, c'_1 \rangle$$

or, more compactly:  $\varrho \langle c, \vec{d} \rangle := \epsilon c \star \lambda c'. \varrho \langle \vec{d}, c' \rangle$ .

**Solo/Duo Separation.** The solo and duo schedulers instantiate round-robin on queues of length 1 and 2 respectively.

**Theorem 3.5 (Separation).**  $\forall c_1 c_2 : \mathbb{C}. \text{LR } c_2 \rightarrow \varrho \langle c_1, c_2 \rangle \cong \varrho \langle c_1 \rangle$

*Proof.* Use co-induction. Fix  $c_1$  and  $c_2$ , and assume  $\text{LR } c_2$ . By  $\text{LR } c_2$  it follows that there is a  $c$  for which  $\epsilon c_2 \cong \eta c$  and, moreover  $\text{LR } c$  holds because the step  $\epsilon$  leaves the process  $\text{ID}$  unchanged. Now, comparing schedulings, observe that:

$$\begin{aligned} & \varrho \langle c_1, c_2 \rangle \\ & \cong \epsilon c_1 \star \lambda c'_1. \epsilon c_2 \star \lambda c'_2. \varrho \langle c'_1, c'_2 \rangle && \text{(unfolding } \varrho \text{ twice)} \\ & \cong \epsilon c_1 \star \lambda c'_1. \eta c \star \lambda c'_2. \varrho \langle c'_1, c'_2 \rangle && \text{(local respect of } c_2) \\ & \cong \epsilon c_1 \star \lambda c'_1. \varrho \langle c'_1, c \rangle && (\eta \text{ is the left identity of } \star) \\ & \cong \epsilon c_1 \star \lambda c'_1. \varrho \langle c'_1 \rangle && \text{(coinductive hypothesis, guarded by } \epsilon c_1) \\ & \cong \varrho \langle c_1 \rangle && \text{(folding } \varrho) \end{aligned}$$

**$n$ -Process Round Robin.** We next extend separation to a general round robin scheduler with one POI and any number of processes showing local respect. But first we need some lemmas.

**Lemma 3.6 (Single Rotation).** For any context  $c$ , if  $\text{LR } c$  then there exists  $c'$  such that  $\text{LR } c'$  and for any queue of contexts,  $\vec{d}$ , we have  $\varrho \langle c, \vec{d} \rangle \cong \varrho \langle \vec{d}, c' \rangle$

*Proof.* Unfold the  $\varrho$  on the left, apply  $\text{LR}$ , and note  $\eta$  is the left identity of  $\star$ .

**Lemma 3.7 (Multi-Rotation).** For any lists of context,  $\vec{d}$  and  $\vec{e}$  if  $\text{LR } \vec{d}$  then there is a  $\vec{d}'$  for which  $\text{LR } \vec{d}'$  and  $\varrho \langle \vec{d}, \vec{e} \rangle \cong \varrho \langle \vec{e}, \vec{d}' \rangle$

*Proof.* By induction on the length of  $\vec{d}$ , using the Single Rotation Lemma.

**Theorem 3.8 (Separation — Round Robin).** For any lists of contexts  $\vec{e}, \vec{d}$  for which  $\text{LR } \vec{e}$  and  $\text{LR } \vec{d}$ , and for any  $c$  we have  $\varrho \langle \vec{e}, c, \vec{d} \rangle \cong \varrho \langle c \rangle$

*Proof.* By Multi-rotation, unfolding and coinduction.

### 3.2 Timer Interrupts

This section sketches separation for a scheduler that executes without switching contexts until the hardware timer interrupts it.

As this is a redevelopment of §3.1 with different details, none of Hypothesis 3.3 through Theorem 3.9 are available. Hypothesis 3.10 through Remark 3.20 are independent of that work.

**Primitives.** Recall that in §3.1  $\epsilon$  was both the smallest computation managed by the scheduler, and the atomic step of machine execution. For timer interrupts, we distinguish these two roles, keeping  $\epsilon$  to denote the former, but for the latter pushing the interface downwards to reveal  $\epsilon_1$ . Now take  $\epsilon_1$  to be atomic and build  $\epsilon$  from it. Local respect is defined for  $\epsilon_1$  and then proven to extend to  $\epsilon$ .

**Hypothesis 3.9** *Assume an atomic step of execution  $\epsilon_1 : \mathbb{C} \rightarrow \text{MC}$  which leaves process IDs invariant.*

**Definition 3.10 (Time Slice).**  $\epsilon : \mathbb{N} \rightarrow \mathbb{C} \rightarrow \text{MC}$

$$\epsilon 0 c := \epsilon_1 c \qquad \epsilon (S m) c := \epsilon_1 c \star \lambda c'. \epsilon m c'$$

Note that  $\epsilon$  takes at least one step regardless of the value of  $n$ .

**Definition 3.11 (Local Respect).**  $\text{LR } i := \forall c. i = \text{ID } c \rightarrow \exists c'. \epsilon_1 c \cong \eta c'$

If the equivalence is oblivious to one step of computation of a process, then it is oblivious to any finite sequence of steps.

**Lemma 3.12 (Local Respect of  $\epsilon$ ).**  $\text{LR } c \rightarrow \forall n. \exists c'. \epsilon n c \cong \eta c'$

*Proof.* Induction on  $n$

**Definition 3.13 (Oracles).** *An oracle is an infinite stream of natural numbers. The type of oracles is denoted by  $\mathbb{N}^\omega$ . That is:  $\mathbb{N}^\omega := \nu X. \mathbb{N} \times X$*

An *oracle*, as an additional argument to  $\varrho$ , models time-slices. Each number in the oracle determines how many  $\epsilon_1$ -steps a process takes between context switches. This model is faithful, since for any real execution in hardware, there is a corresponding stream of naturals.

**Definition 3.14 (Round Robin with Oracle).**

$$\begin{aligned} \varrho : \mathbb{N}^\omega &\rightarrow \text{QC} \rightarrow \text{M1} \\ \varrho (h, t) \langle c, \vec{d} \rangle &:= \epsilon h c \star \lambda c'. \varrho t \langle \vec{d}, c' \rangle \end{aligned}$$

Separation needs a few lemmas for manipulating oracles. The proofs are omitted as they are similar enough to the proof of Theorem 3.6. The first shows a correspondence between solo and duo schedulings: if  $\text{LR } d$  then any duo execution involving  $d$ , corresponds to a solo execution without  $d$  but using a different oracle. Specifically:

**Lemma 3.15 (Correspondence).**  $\text{LR } d \rightarrow \forall o. \varrho \ o \ \langle c, d \rangle \cong \varrho \ (\text{alt } o) \ \langle c \rangle$  where  $\text{alt } (h_1, (h_2, t)) := (h_1, \text{alt } t)$

The next shows an independence of the behaviour of a solo scheduling from the oracle it uses, but it is easier to establish as a corollary of a more general result.

**Lemma 3.16 (Independence, generalized).**  $\forall n_1 \ n_2 \ o_1 \ o_2.$

$$\epsilon \ n_1 \ c \star \lambda c'. \varrho \ o_1 \ \langle c' \rangle \cong \epsilon \ n_2 \ c \star \lambda c'. \varrho \ o_2 \ \langle c' \rangle$$

**Corollary 3.17 (Independence).**  $\forall o_1 \ o_2. \varrho \ o_1 \ \langle c \rangle \cong \varrho \ o_2 \ \langle c \rangle$

**Theorem 3.18 (Separation).**  $\forall o_1 \ o_2. \text{LR } d \rightarrow \varrho \ o_1 \ \langle c, d \rangle \cong \varrho \ o_2 \ \langle c \rangle$

*Proof.* Both sides are equivalent to  $\varrho \ (\text{alt } o_1) \ \langle c \rangle$ , by Independence and Correspondence, respectively.

*Remark 3.19.* It is possible to recast this work with an implicit oracle, incorporated into the monad. The landmark results (the definition of round-robin, the statement of separation) are then identical to those in §3.1, but the proof details differ.

### 3.3 Page Faults

Physical memory has two problems: it is limited, and it must be shared among processes. Yet most operating systems give user processes an abstraction of memory, called *virtual memory*, which emulates an unbounded amount of contiguous memory. It does so with a technique called *paging*. Memory is divided into uniformly sized *pages*. When all physical memory is in use, yet more is needed, one page of physical memory is moved to disk. When a page that resides on disk is needed in memory, space is found in physical memory.

In the following, the key feature is that a step may either advance as normal, or raise a *page fault* which halts normal execution, invokes a kernel process to page the needed data from disk to memory, and then resumes execution.

The separation argument extends to schedulers with a restricted form of page fault. Page maps, drawn from an abstract type  $\mathbb{V}$ , are explicitly present as an extra argument/return value to atomic step,  $\epsilon$ , and the scheduler,  $\varrho$ . An abstract type of memory addresses,  $\mathbb{A}$ , is also explicit.

**Primitives.** As in §3.2, we push the interface downwards to work with lower level primitives. In this case there are two primitive computations. The first is an atomic step of execution,  $\epsilon_f$ , that may either succeed, returning an updated context, or raise a page fault, returning the address that is not currently mapped. The second,  $\xi$ , extends a page map to include a specific address.

#### Hypothesis 3.20

$$\begin{aligned} \epsilon_f &: \mathbb{V} \rightarrow \mathbb{C} \rightarrow \mathbb{M}(\mathbb{C} + \mathbb{A}) \\ \xi &: \mathbb{V} \rightarrow \mathbb{A} \rightarrow \mathbb{M}\mathbb{V} \end{aligned}$$

The memory system for virtual memory is responsible for paging between memory and disk. For purposes of separation,  $\xi$  is a primitive, given by the memory system. We require an assumption, similar to local respect, that the behavior of  $\xi$  is not observable:

**Hypothesis 3.21**  $\forall v. \exists v'. \xi v a \cong \eta v'$

**Handling Single-Faultedness.** The first step is to define a non-faulting step of execution, again called  $\epsilon$ , for which we can prove separation in the style of §3.1. A *handled step* of execution  $\epsilon_h$  combines  $\epsilon_f$  with a page fault handler of type  $\mathbb{V} \rightarrow \mathbb{C} \rightarrow \mathbb{A} \rightarrow \mathbb{M}(\mathbb{V} \times \mathbb{C})$

$$\begin{aligned} \epsilon_h &: \mathbb{V} \rightarrow \mathbb{C} \rightarrow (\mathbb{V} \rightarrow \mathbb{C} \rightarrow \mathbb{A} \rightarrow \mathbb{M}(\mathbb{V} \times \mathbb{C})) \rightarrow \mathbb{M}(\mathbb{V} \times \mathbb{C}) \\ \epsilon_h v c h &:= \epsilon_f v c \star \lambda x. \begin{cases} \eta(v, c') & \text{if } x = \iota_1 c' \\ h v c a & \text{if } x = \iota_2 a \end{cases} \end{aligned}$$

The result is robust to page faults, though we don't yet have a handler  $h$ . The details of  $h$  depend upon the nature of page faults. For concreteness, we make the following assumption.

**Hypothesis 3.22 (single-faultedness)** *If one step of the execution of context  $c$  with page map  $v$  faults, specifying address  $i$ , and if extension of  $v$  via  $\xi$  to include  $i$  yields page map  $v'$ , then retrying that step on  $v'$  will not fault.*

Given single-faultedness, a robust step function can be coded using  $\epsilon_h$ , and letting  $\perp$  stand for an unreachable computation:

$$\begin{aligned} \epsilon &: \mathbb{V} \rightarrow \mathbb{C} \rightarrow \mathbb{M}(\mathbb{V} \times \mathbb{C}) \\ \epsilon v c &:= \epsilon_h v c (\lambda v. \lambda c. \lambda i. \xi v i \star \lambda v'. \epsilon_h v' c \perp) \end{aligned}$$

Note that  $\epsilon_h$  occurs twice here, with the second occurrence being the handler for the first. The single-faultedness of  $\xi$  guarantees that the second use is robust despite the occurrence of  $\perp$ . The sketch of separation is: from local respect for  $\epsilon_f$  and  $\xi$ , show local respect for  $\epsilon_h$  and for  $\epsilon$ ; the proof of separation for Theorem 3.6 then goes through with minor modifications.

**More General Faults.** The single-faultedness assumption is simplistic, but the approach scales. If the only memory access is through load and store then there are at most two faults: one accessing the instruction, and one accessing the data. If a machine operation like ADD can access indirect memory then there may be numerous faults due to both the number of arguments and the indirect lookup. If the number of faults is bounded, then a generalization to  $n$ -faultedness suffices. Alternatively, using the iteration monad of §4.4,  $\epsilon_f$  can loop until it succeeds.

*Remark 3.23.* Page faults are not yet integrated with timer interrupts. To do so requires  $\epsilon_f : \mathbb{V} \rightarrow \mathbb{C} \rightarrow \mathbb{M}(\mathbb{C} \times \text{Option } \mathbb{A})$ , which may do non-trivial work before the interrupt, so it always returns an updated context.

### 3.4 Summary of High-level Proofs

The proofs for full round-robin, timer interrupts and page faults are independent generalizations of the Solo/Duo scheduler problem. It remains to combine all three generalizations into the same system. We see no fundamental impediment to combining them, except for an increase in complexity. We anticipate that having mechanized theorem prover support will be beneficial for that integration. Next we turn to issues in mechanization.

## 4 Mechanization Design

The eventual goal of this research is to mechanically verify separation for Haskell code. Mechanization is necessary to verify actual code, and to deal with tedious details that arise as schedulers become more complex.

To compensate for the lack of logic in Haskell, we embed the scheduler into Coq. To support the custom equivalence relation  $\cong$ , which is a central feature of separation proofs, we use setoids with an applicative structure instead of mere types. The mechanization strategy leverages the monadic nature of the scheduler in two different ways. First, as an ADT for computations. Second, to add effects: logging, which is the basis for  $\cong$ , and non-termination, which is a work-around for a difficulty in Coq.

### 4.1 Embedding Haskell in Coq

By an *applicative structure* we mean an implementation of typed  $\lambda$ -calculus. Terms may be built by function application and  $\lambda$ -abstraction. Terms evaluate by  $\beta$ -reduction. Each term has a type, and there is an *arrow* type constructor, so that  $A \rightarrow B$  classifies functions from type  $A$  to type  $B$ .

The features of Haskell that the scheduler uses are the statically typed applicative structure, non-terminating tail recursion, and the monadic interface to hardware. Coq implements a simply typed  $\lambda$ -calculus augmented by inductive and co-inductive types. It provides in an integrated framework both an executable language of types & terms, and a language of properties & proofs. To maintain logical consistency, recursion is restricted in Coq, but it is, in principle, still powerful enough to implement a scheduler.

At first glance, Coq is the ideal target for the embedding: inductive types can model the hardware monad, tail recursion maps to co-recursion, and the applicative structure maps directly. It turns out that we both need a novel applicative structure, as types are too poor to model our equivalence, and we need a novel means of iteration, as co-recursion is difficult to use.

### 4.2 Setoids

Now we discuss the change to the applicative structure. The notion of equality in Coq is Leibniz equality: two terms are equal iff no context can distinguish

them. Separation requires a coarser equivalence. Leibniz equality distinguishes  $\epsilon d$  from  $\eta d'$ , as  $\epsilon d$  has effects, while  $\eta d'$  does not: when we assert  $\text{LR } d$ , we need to equate these two computations.

To model an equivalence relation on a type, Coq uses a *setoid*, which is a type paired with an equivalence relation. An operator on a setoid must be a *congruence*, that is a function respecting the equivalence: if  $f : A \rightarrow B$  and  $a_1 \equiv_A a_2$  then  $f a_1 \equiv_B f a_2$ . Each definition of a  $\lambda$ -term requires a proof of congruence. We'll use the term *A-equivalence* to refer to  $\equiv_A$ .

In this setting,  $\mathbb{M}$  is a function at the type level which maps any setoid to another setoid. That is, given a type  $A$ , an equivalence  $\equiv_A$  as well as proofs of reflexivity, symmetry, transitivity of  $\equiv_A$ , a monad  $\mathbb{M}$  will generate at least five new items: a type of computations (denoted by  $\mathbb{M}A$ ), an equivalence ( $\equiv_{\mathbb{M}A}$ , which is based on  $\equiv_A$ ), and proofs of each of reflexivity, symmetry and transitivity for  $\equiv_{\mathbb{M}A}$ .  $\mathbb{M}$  may introduce other operations (e.g. *put* and *get*) and establish more laws. Each operation (e.g.  $\eta$ ,  $\star$ , *put*, and *get*) must be a congruence.

### 4.3 Logging Monad

Logging is key to implementing the coarse-grained equivalence of separation proofs. For our purposes, the interface is not important; what matters is that it provides a way to instrument *other* monads without changing *their* interfaces.

A logging computation is parameterized by a type of *token*. Intuitively, an  $A$ -computation is a finite list of these tokens, ending in a value from  $A$ . Return builds an empty list of tokens plus a pure  $A$ -value. Bind concatenates two lists, where the second list depends on the  $A$ -value embedded in the first. A new operation,  $\text{log} : T \rightarrow \text{Log } 1$ , creates a singleton list from a token.

Formally, fix parameter  $T$ , then the fixed point  $\text{Log } A := \mu X. A + T \times X$  defines the type of computation. Using  $\text{Val } a := \iota_1 a$  and  $\text{Step } t m := \iota_2 (t, m)$  as mnemonic constructors:

$$\begin{aligned} \text{log } t &:= \text{Step } t (\text{Val } ()) & \text{Val } a \star r &:= r a \\ \eta a &:= \text{Val } a & (\text{Step } t q) \star r &:= \text{Step } t (q \star r) \end{aligned}$$

It is straightforward to show that the monad laws hold. Two computations are  $\text{Log } A$ -equivalent iff their return values are  $A$ -equivalent and their token lists are equivalent (that is, the lists have the same length and  $T$ -equivalent entries occur in corresponding positions).

**Instrumentation.** The value of logging is in combination with other monads. Given a set of process identifiers, assert for which ones local respect holds. Given a monad  $\mathbb{U}$  with operation  $\epsilon : \mathbb{C} \rightarrow \mathbb{U}\mathbb{C}$  create an instrumented monad  $\mathbb{M}A = \mathbb{U} (\text{Log } A)$  by instantiating  $T$  to  $\mathbb{C}$ , then instrumenting a step of execution:

$$\epsilon_{\mathbb{M}} c := \begin{cases} \epsilon c & \text{if } \text{LR } c \\ \text{log } c \gg \epsilon c & \text{if } \neg(\text{LR } c) \end{cases}$$

The coarsened equivalence relation on  $\mathbb{M}$  is then the relation of 3.1.

**Distributivity.** The logging monad is distributive: its effects can be combined with those of any other monad  $\mathbb{U}$  because it has an operator,  $\delta : \forall A. \text{Log } (A) \rightarrow \mathbb{U}(\text{Log } A)$ , satisfying laws of interaction with the *join*, *map* and  $\eta$  operations of the two monads.

$$\delta (\text{Val } u) := \text{map}_{\mathbb{U}} \text{Val } u \quad \delta (\text{Step } t \ m') := \text{map}_{\mathbb{U}} (\text{Step } t) \ m'$$

#### 4.4 Iteration Monad Transformer

One limitation of Coq is its restriction on recursion. To relax the restriction for monadic code, we add the ability to loop as a monadic effect, which is sufficient for OS code. The additional interface is a monad transformer which adds an iteration operator [BÉ93]. That is, it is a monad  $(\mathbb{M}, \eta_{\mathbb{M}}, \star_{\mathbb{M}})$  parameterized by an underlying monad  $(\mathbb{U}, \eta_{\mathbb{U}}, \star_{\mathbb{U}})$ , where  $\mathbb{M}$  gains the effects of  $\mathbb{U}$  through lifting, while  $\mathbb{M}$  also adds the effect of non-termination by adding a *loop* operation.

name	abbrev.	type
lift		$\mathbb{U}A \rightarrow \mathbb{M}A$
loop	$_^\dagger$	$(A \rightarrow \mathbb{M}(A + R)) \rightarrow A \rightarrow \mathbb{M}R$
unroll		$\mathbb{M}A \rightarrow \mathbb{N} \rightarrow \mathbb{U}(\text{Option}(\mathbb{N} \times A))$

**Fig. 5.** Iteration Operations

$\eta_{\mathbb{M}} a = \text{lift } (\eta_{\mathbb{U}} (\text{Some } a))$	
unroll (lift $u$ ) $n = u \star_{\mathbb{U}} \lambda a. \eta_{\mathbb{U}} (\text{Some } (n, a))$	
unroll ( $p \star_{\mathbb{M}} q$ ) $n = \text{unroll } p \ n \star_{\mathbb{U}} \lambda x. \begin{cases} \eta_{\mathbb{U}} \text{None} & \text{if } x = \text{None} \\ \text{unroll } (q \ a) \ m & \text{if } x = \text{Some } (m, a) \end{cases}$	
unroll ( $e^\dagger a$ ) $n = \begin{cases} \eta_{\mathbb{U}} \text{None} & \text{if } n = 0 \\ \text{unroll } (ea \star_{\mathbb{M}} [e^\dagger, \eta_{\mathbb{U}}]) \ m & \text{if } n = S \ m \end{cases}$	

**Fig. 6.** Iteration Axioms

The accessor *unroll* provides a basis for defining  $\equiv_{\mathbb{M}A}$  in terms of  $\equiv_{\mathbb{U}A}$ . First, define a preorder on  $\mathbb{U}$  computations: for  $u_i : \mathbb{U}A_i$  let  $u_1 \sqsubseteq_{\mathbb{U}} u_2$  mean  $\exists q. u_1 \star q \equiv_{\mathbb{U}A_2} u_2$ . Next, lift it to a preorder on  $\mathbb{M}$ : for  $m_i : \mathbb{M}A_i$  let  $m_1 \sqsubseteq_{\mathbb{M}} m_2$  mean  $\forall n_1. \exists n_2. \text{unroll } m_1 \ n_1 \sqsubseteq_{\mathbb{U}} \text{unroll } m_2 \ n_2$ . To define equality, given  $m_i : \mathbb{M}A, i \in \{1, 2\}$  let  $m_1 \equiv_{\mathbb{M}A} m_2$  mean  $m_1 \sqsubseteq_{\mathbb{M}} m_2$  and  $m_2 \sqsubseteq_{\mathbb{M}} m_1$  and the return values, if any, of  $m_1$  and  $m_2$  are equal.

We have the following results (proofs omitted due to space constraints).

**Theorem 4.1.** *if  $(\mathbb{U}, \eta_{\mathbb{U}}, \star_{\mathbb{U}})$  is a monad then  $(\mathbb{M}, \eta_{\mathbb{M}}, \star_{\mathbb{M}})$  is a monad*

**Theorem 4.2.** *The axiomatization of iteration is sufficiently complete.*

Arguably, hardware informally is a model as each unrolling corresponds to a backwards local `jmp` in hardware.

It is possible to implement a model of the iteration monad in Coq by taking  $\eta$ ,  $\star$ , *lift*, and *loop* as constructors, and to define *unroll* recursively, though showing that *unroll* is well-founded is a little challenging.

**Guardedness.** Iteration theory is an alternative to using co-induction in Coq, which has proven difficult to work with. Coq uses a very conservative syntactic test to ensure that co-recursion is sound, however many simple functions violate this constraint, as well as proofs that mix induction with co-induction, as Multi-Rotation does. In general, using co-induction in Coq requires a deal of insight and creative refactoring of problems. There is active research into better support for coinductive proofs in Coq, and better programming styles to interact with co-induction (for instance [BK08], [?]) but it is not yet at the state we require.

Co-inductive proofs are more readable than ones using approximation lemmas or pre-orders, but at the time of writing, it seems preferable to mechanize using the latter. Our earlier attempts at mechanization depended on co-induction in Coq, and this kept us from progressing.

## 5 Conclusions

### 5.1 Summary & Future Work

This paper formulates a way of framing the separation problem, shows a variety of high level, but simple separation proofs (*n*-process round robin, timer interrupts, page faults), and sketches a way of mechanizing those proofs, and others, in Coq. Mechanization is essential for two reasons: first, the ultimate goal of this research is to verify code, and second the amount of proof detail increases with the scheduler complexity and with H-monad complexity. The hypotheses in this work are drastic simplifications of the actual H-monad.

We also discussed a strategy for mechanization, comprising an embedding of Haskell code into Coq, albeit with an applicative structure on setoids instead of using native types. We use monads to define an appropriate equivalence relation,  $\cong$ , and to work around limits with Coq's implementation of coinduction. Finally, we use sufficient completeness to justify why a proof about a model can demonstrate a property about real hardware.

This is a work in progress, and much work remains. Foremost, the mechanization is only partially done. Second, we must use more realistic schedulers and interfaces; a start is to combine *n*-process round robin, timer interrupts and page faults into one separation problem. We also need a sufficiently complete axiomatization for a monad of timer interruptions. Some technical work remains: We conjecture that the five laws of iteration theory hold (see [BÉ93]) though we have not yet finished all five proofs.

One reviewer suggested looking at *effect terms* and imposing equational constraints on the resulting Lawvere theories [PP08]. Logging, as presented here, is an ad hoc mechanism that achieves some of what effect terms do. Preliminary tests with effect terms show they are a more direct and versatile way of recording the actions of a computation, so we hope to integrate them in future work. For instance it may be fruitful to treat non-determinism as a binary (or  $\mathbb{N}$ -ary) term constructor instead of an oracle. Unfortunately, effect terms still violate Coq's guardedness constraint

## 5.2 Related Work

Rushby [Rus92] provides an automata-theoretic means of showing noninterference. His basic vocabulary includes three notions: local respect, step consistency, and output consistency. We keep all three, explicitly adopting a variant of the first, while the two notions of consistency are implicitly maintained by setoids. In general, his technique is quite powerful, addressing intransitive non-interference. We address neither transitivity nor intransitivity. On the other hand, his work only applies to models. Our work is based on the idea of a monad, and lends itself to proofs about actual code.

## References

- [BÉ93] Stephen L. Bloom and Zoltán Ésik. *Iteration Theory: The Equational Logic Of Iterative Processes*. Springer-Verlag, 1993.
- [BK08] Yves Bertot and Ekaterina Komendantskaya. Using structural recursion for corecursion. In Stefano Berardi, Feruccio Damiani, and Ugo de Liguoro, editors, *Types 2008*, Springer-Verlag, 2008.
- [DA09] Nils Anders Danielsson and Thorsten Altenkirch. Mixing induction and coinduction, 2009. Draft.
- [HJLT05] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming*, pages 116–128, 2005.
- [Mac71] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.
- [PP08] Gordon Plotkin and Matija Pretnar. A logic for algebraic effects. In *LICS '08: Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science*, pages 118–129, 2008.
- [Rus92] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, Stanford Research Institute, December 1992.
- [TM92] J. L. Turner and T. L. McCluskey. *The Construction of Formal Specifications: An Introduction to the Model-Based and Algebraic Approaches*. McGraw Hill, 1992. Online at <http://scom.hud.ac.uk/scomt1m/book.pdf>.