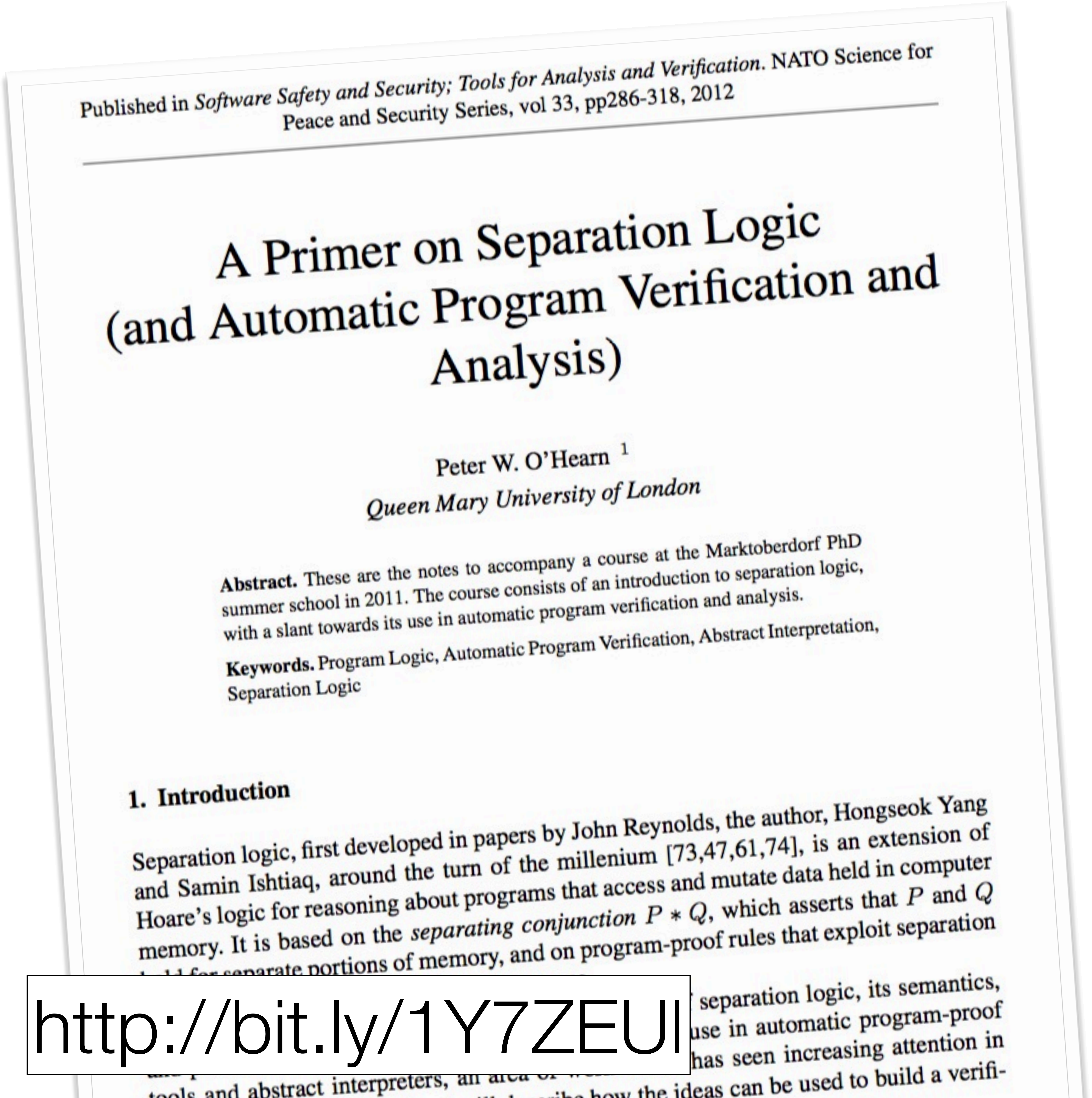
AUTOMATIZING LINEARIZABILITY VERIFICATION

Recommended Reading



Published in *Software Safety and Security; Tools for Analysis and Verification*. NATO Science for Peace and Security Series, vol 33, pp286-318, 2012

A Primer on Separation Logic (and Automatic Program Verification and Analysis)

Peter W. O'Hearn [1]

Queen Mary University of London

Abstract. These are the notes to accompany a course at the Marktoberdorf PhD summer school in 2011. The course consists of an introduction to separation logic, with a slant towards its use in automatic program verification and analysis.

Keywords. Program Logic, Automatic Program Verification, Abstract Interpretation, Separation Logic

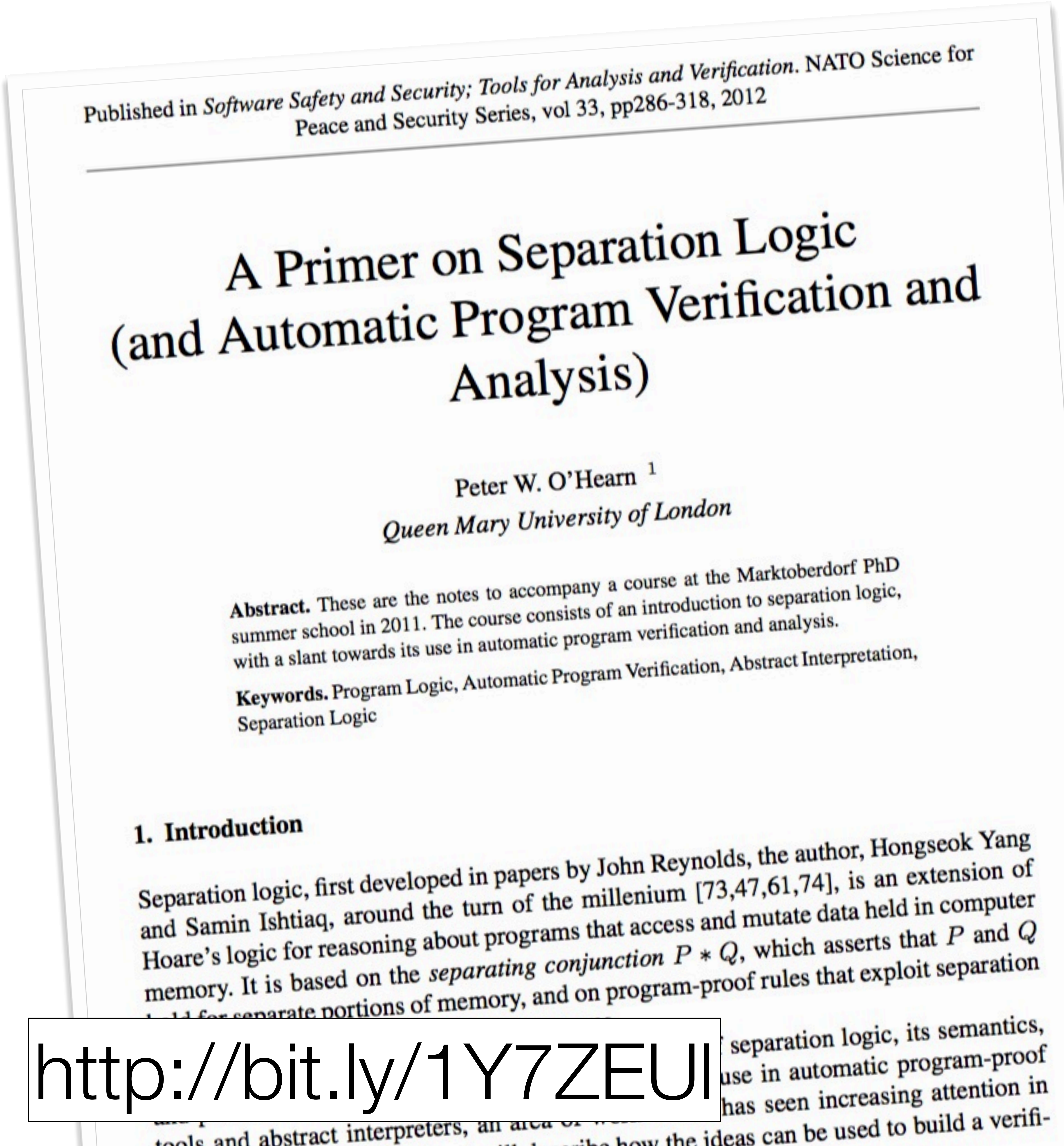
1. Introduction

Separation logic, first developed in papers by John Reynolds, the author, Hongseok Yang and Samin Ishtiaq, around the turn of the millenium [73,47,61,74], is an extension of Hoare's logic for reasoning about programs that access and mutate data held in computer memory. It is based on the *separating conjunction* $P * Q$, which asserts that P and Q hold for separate portions of memory, and on program-proof rules that exploit separation

separation logic, its semantics, use in automatic program-proof has seen increasing attention in tools and abstract interpreters, an area of ... how the ideas can be used to build a verifi-

http://bit.ly/1Y7ZEUI

Recommended Reading



Published in *Software Safety and Security; Tools for Analysis and Verification*. NATO Science for Peace and Security Series, vol 33, pp286-318, 2012

A Primer on Separation Logic (and Automatic Program Verification and Analysis)

Peter W. O'Hearn [1]
Queen Mary University of London

Abstract. These are the notes to accompany a course at the Marktoberdorf PhD summer school in 2011. The course consists of an introduction to separation logic, with a slant towards its use in automatic program verification and analysis.

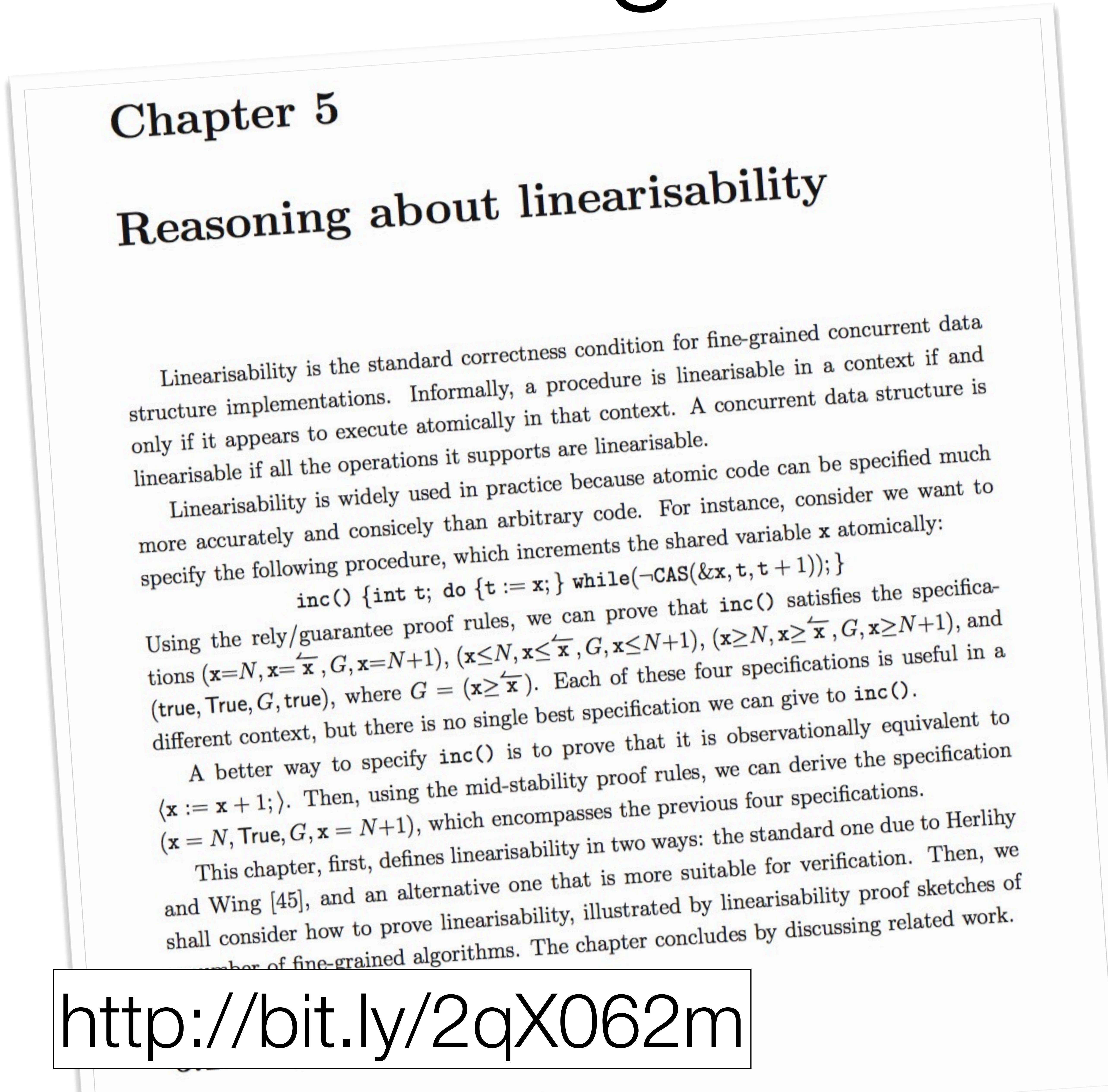
Keywords. Program Logic, Automatic Program Verification, Abstract Interpretation, Separation Logic

1. Introduction

Separation logic, first developed in papers by John Reynolds, the author, Hongseok Yang and Samin Ishtiaq, around the turn of the millenium [73,47,61,74], is an extension of Hoare's logic for reasoning about programs that access and mutate data held in computer memory. It is based on the *separating conjunction* $P * Q$, which asserts that P and Q hold for separate portions of memory, and on program-proof rules that exploit separation

separation logic, its semantics,
use in automatic program-proof
has seen increasing attention in
tools and abstract interpreters, an area of
how the ideas can be used to build a verifi-

http://bit.ly/1Y7ZEUl



Chapter 5

Reasoning about linearisability

Linearisability is the standard correctness condition for fine-grained concurrent data structure implementations. Informally, a procedure is linearisable in a context if and only if it appears to execute atomically in that context. A concurrent data structure is linearisable if all the operations it supports are linearisable.

Linearisability is widely used in practice because atomic code can be specified much more accurately and consicely than arbitrary code. For instance, consider we want to specify the following procedure, which increments the shared variable `x` atomically:

$$\texttt{inc() \{int t; do \{t := x;\} while(}\neg\texttt{CAS(\&x, t, t + 1));\}}$$

Using the rely/guarantee proof rules, we can prove that `inc()` satisfies the specifications $(\mathtt{x}{=}N, \mathtt{x}{=}\overleftarrow{\mathtt{x}}, G, \mathtt{x}{=}N{+}1)$, $(\mathtt{x}{\le}N, \mathtt{x}{\le}\overleftarrow{\mathtt{x}}, G, \mathtt{x}{\le}N{+}1)$, $(\mathtt{x}{\ge}N, \mathtt{x}{\ge}\overleftarrow{\mathtt{x}}, G, \mathtt{x}{\ge}N{+}1)$, and $(\mathsf{true}, \mathsf{True}, G, \mathsf{true})$, where $G = (\mathtt{x}{\ge}\overleftarrow{\mathtt{x}})$. Each of these four specifications is useful in a different context, but there is no single best specification we can give to `inc()`.

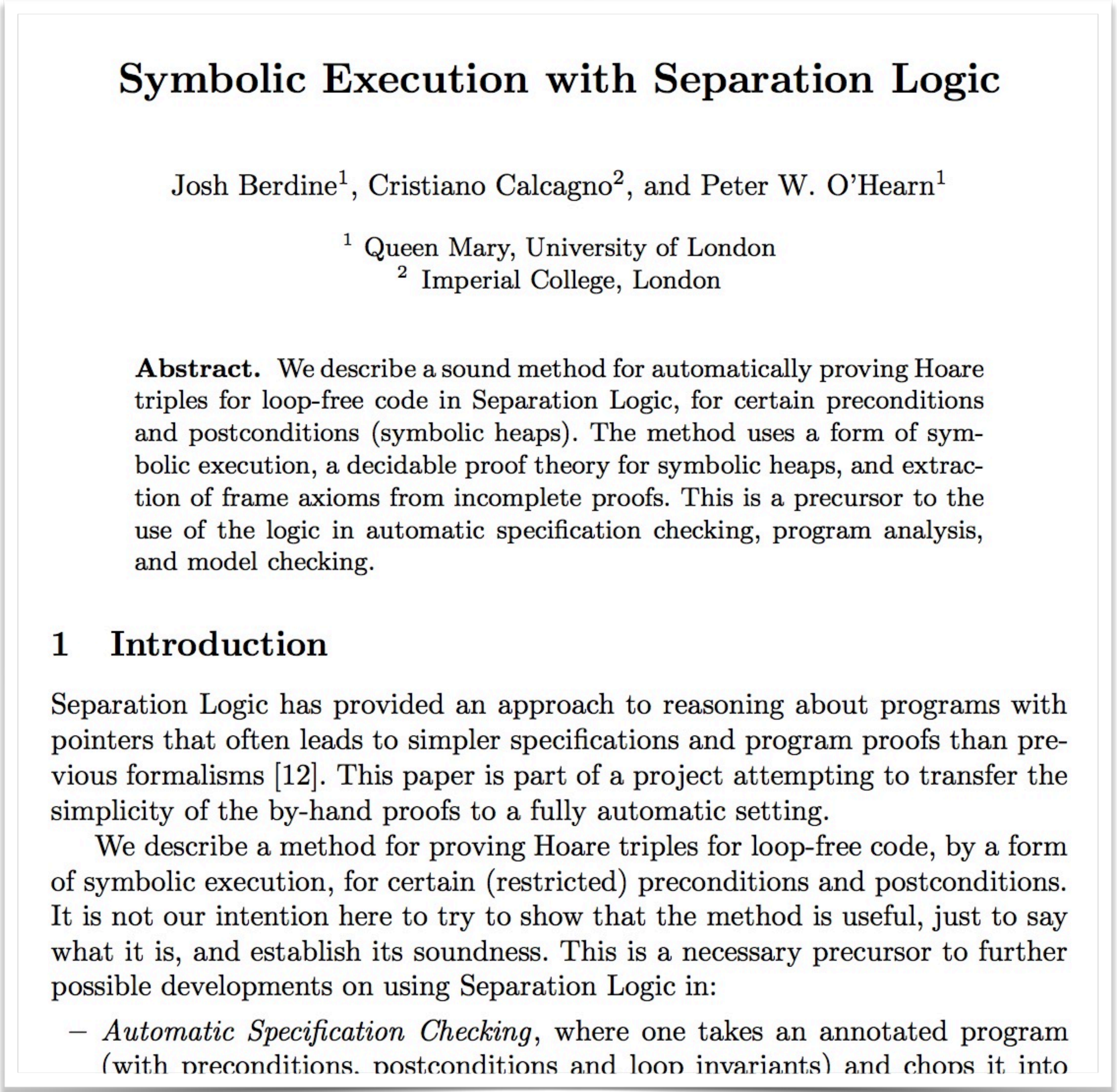
A better way to specify `inc()` is to prove that it is observationally equivalent to $\langle \mathtt{x} := \mathtt{x} + 1; \rangle$. Then, using the mid-stability proof rules, we can derive the specification $(\mathtt{x} = N, \mathsf{True}, G, \mathtt{x} = N{+}1)$, which encompasses the previous four specifications.

This chapter, first, defines linearisability in two ways: the standard one due to Herlihy and Wing [45], and an alternative one that is more suitable for verification. Then, we shall consider how to prove linearisability, illustrated by linearisability proof sketches of a number of fine-grained algorithms. The chapter concludes by discussing related work.

http://bit.ly/2qX062m

Verification Ingredients

- Specifying a Library: φ
- Implementing a Library: $\mathtt{L}$
- Verifying a Library implementation: $\mathtt{L} \models \varphi$



Symbolic Execution with Separation Logic

Josh Berdine[1], Cristiano Calcagno[2], and Peter W. O'Hearn[1]

[1] Queen Mary, University of London
[2] Imperial College, London

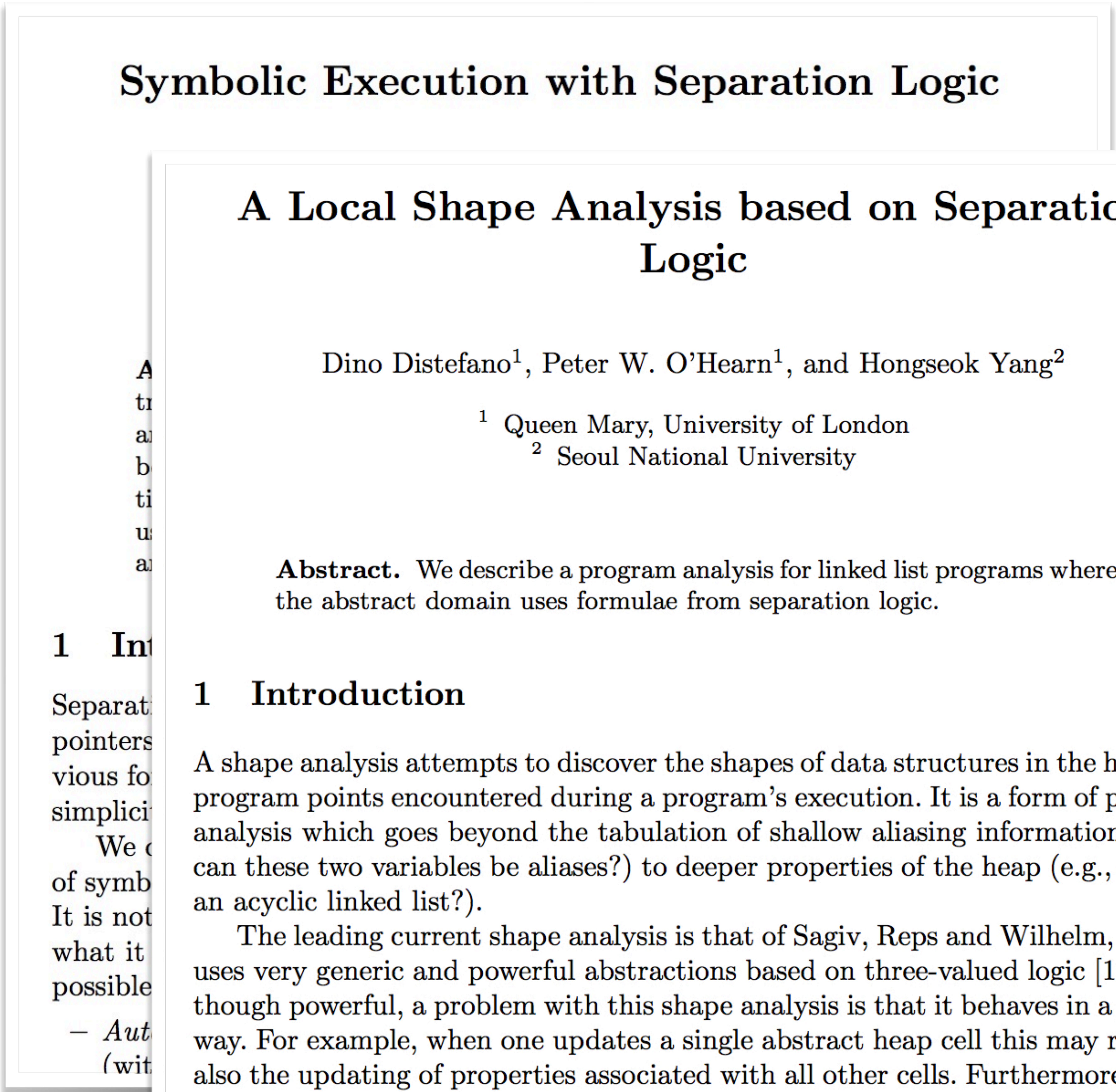
Abstract. We describe a sound method for automatically proving Hoare triples for loop-free code in Separation Logic, for certain preconditions and postconditions (symbolic heaps). The method uses a form of symbolic execution, a decidable proof theory for symbolic heaps, and extraction of frame axioms from incomplete proofs. This is a precursor to the use of the logic in automatic specification checking, program analysis, and model checking.

1 Introduction

Separation Logic has provided an approach to reasoning about programs with pointers that often leads to simpler specifications and program proofs than previous formalisms [12]. This paper is part of a project attempting to transfer the simplicity of the by-hand proofs to a fully automatic setting.

We describe a method for proving Hoare triples for loop-free code, by a form of symbolic execution, for certain (restricted) preconditions and postconditions. It is not our intention here to try to show that the method is useful, just to say what it is, and establish its soundness. This is a necessary precursor to further possible developments on using Separation Logic in:

– *Automatic Specification Checking*, where one takes an annotated program (with preconditions, postconditions and loop invariants) and chops it into

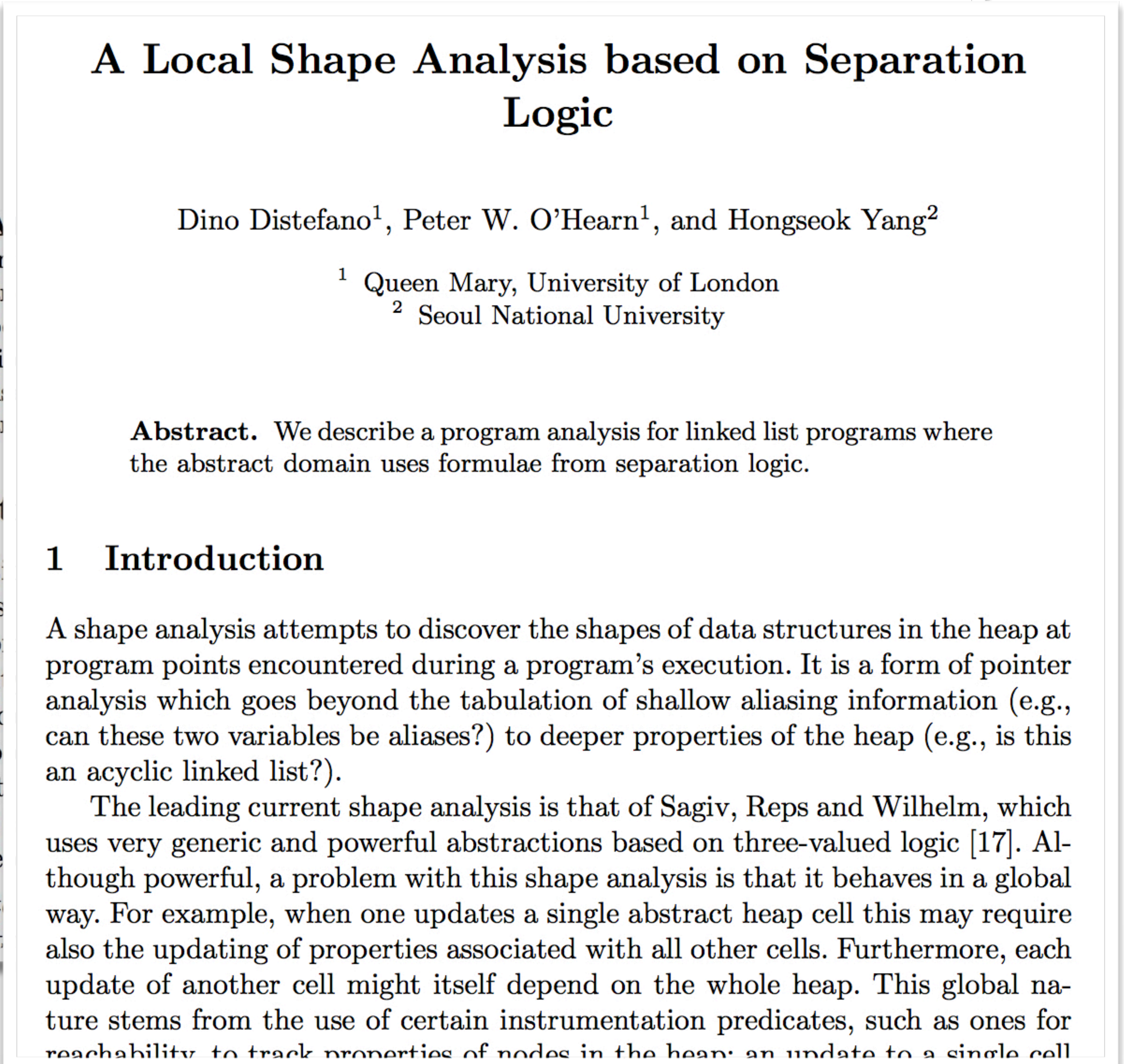

Symbolic Execution with Separation Logic

A
tr
a
b
ti
u
a

1 Int

Separat
pointers
vious fo
simplici
We c
of symb
It is not
what it
possible

– *Aut*
(wit


A Local Shape Analysis based on Separation Logic

Dino Distefano[1], Peter W. O'Hearn[1], and Hongseok Yang[2]

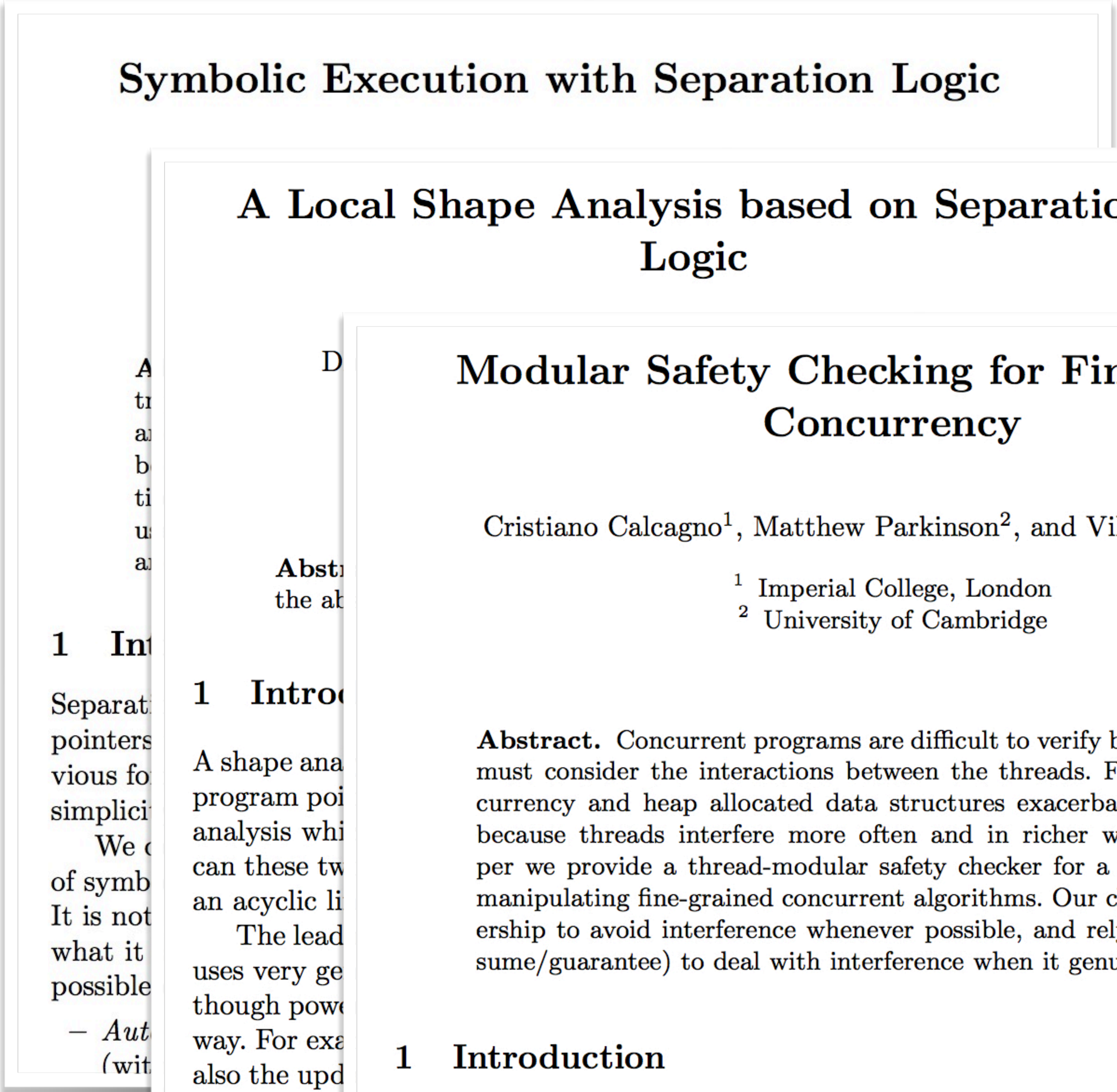
[1] Queen Mary, University of London
[2] Seoul National University

Abstract. We describe a program analysis for linked list programs where the abstract domain uses formulae from separation logic.

1 Introduction

A shape analysis attempts to discover the shapes of data structures in the heap at program points encountered during a program's execution. It is a form of pointer analysis which goes beyond the tabulation of shallow aliasing information (e.g., can these two variables be aliases?) to deeper properties of the heap (e.g., is this an acyclic linked list?).

The leading current shape analysis is that of Sagiv, Reps and Wilhelm, which uses very generic and powerful abstractions based on three-valued logic [17]. Although powerful, a problem with this shape analysis is that it behaves in a global way. For example, when one updates a single abstract heap cell this may require also the updating of properties associated with all other cells. Furthermore, each update of another cell might itself depend on the whole heap. This global nature stems from the use of certain instrumentation predicates, such as ones for reachability, to track properties of nodes in the heap; an update to a single cell

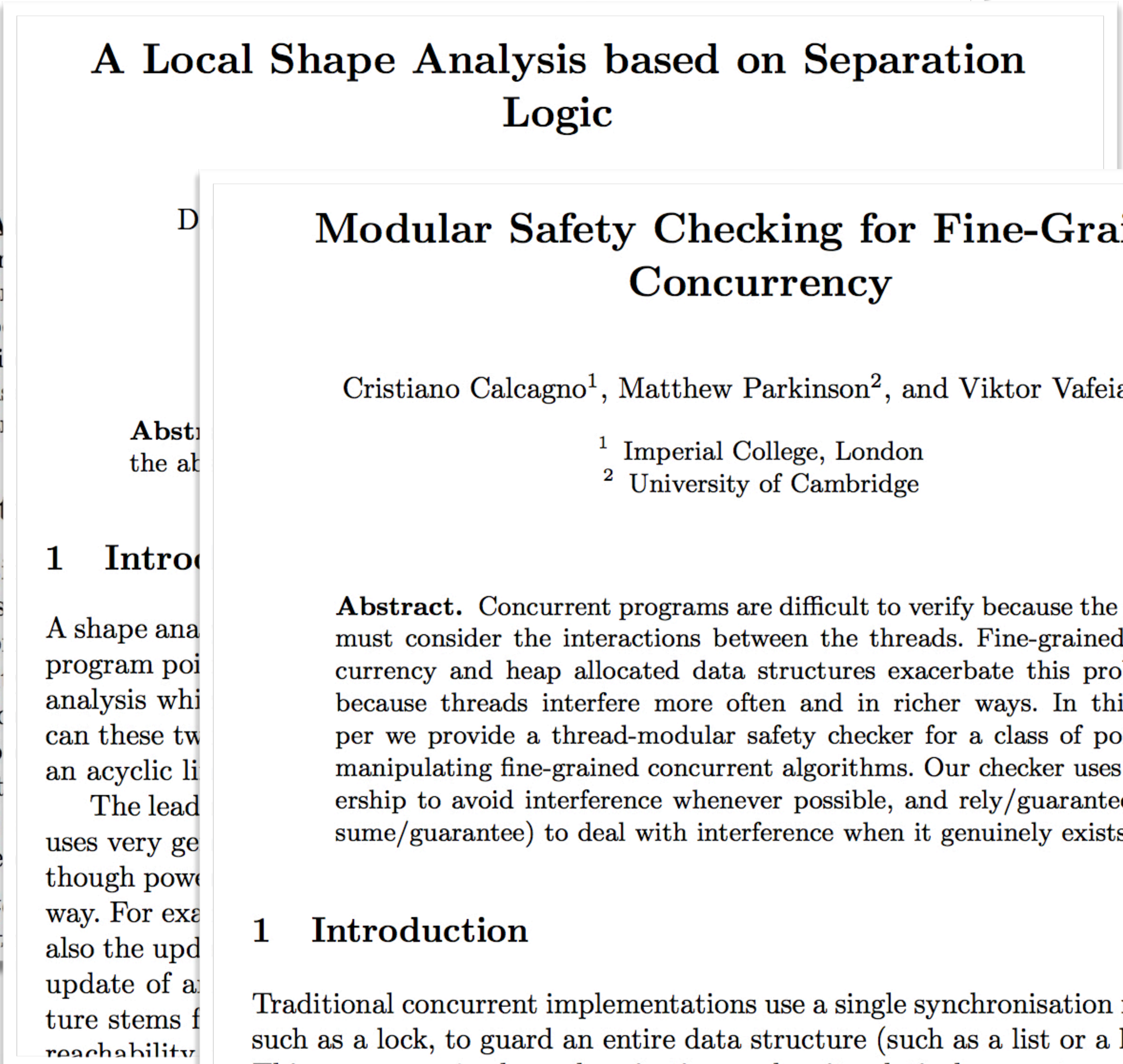

Symbolic Execution with Separation Logic

A
tı
aı
b
ti
u
aı

1 Int

Separati
pointers
vious fo
simplici
We c
of symb
It is not
what it
possible

– Aut
(wit

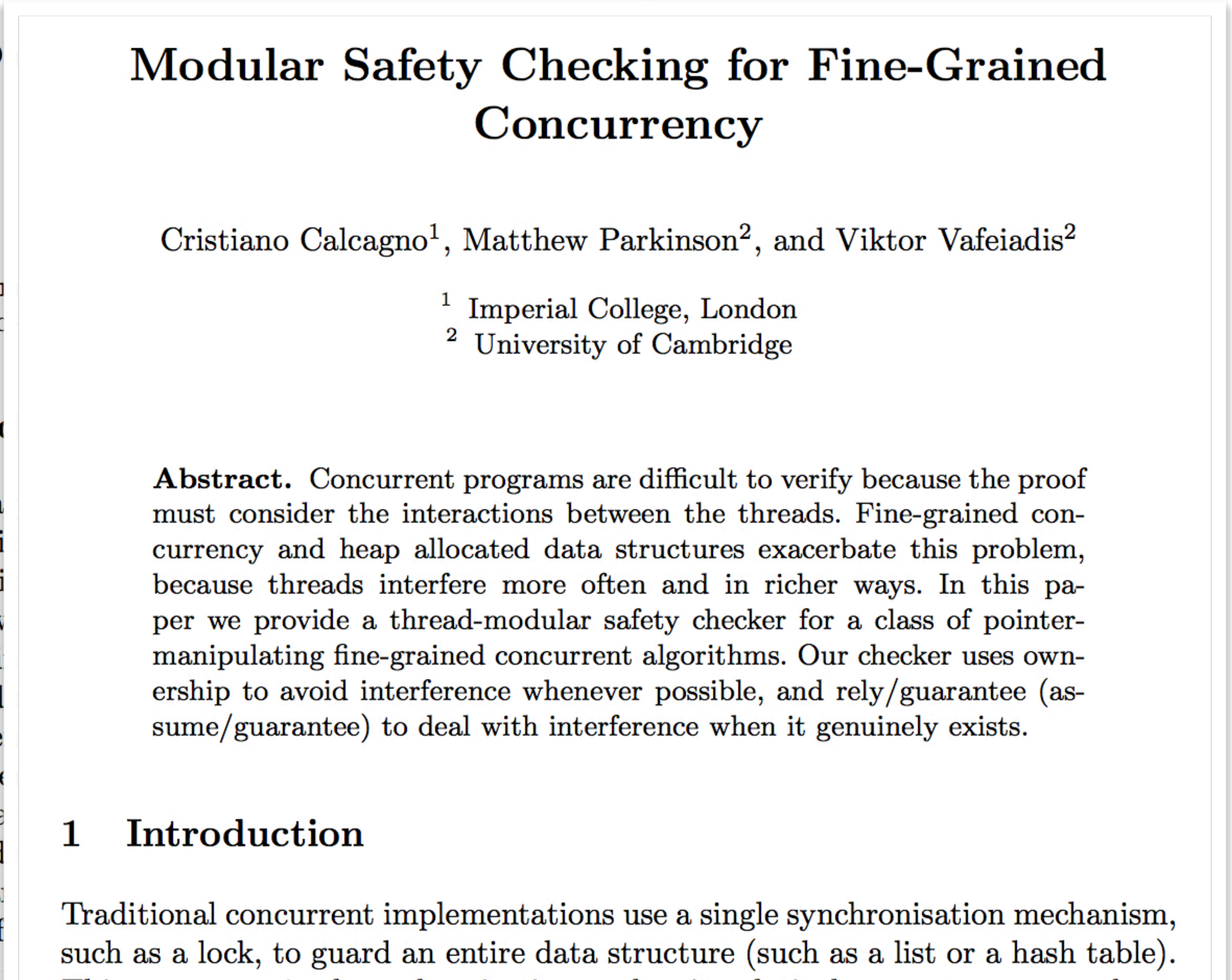

A Local Shape Analysis based on Separation Logic

D

Abstı
the ab

1 Intro

A shape ana
program poi
analysis whi
can these tw
an acyclic li
The lead
uses very ge
though powe
way. For exa
also the upd
update of aı
ture stems f
reachability


Modular Safety Checking for Fine-Grained Concurrency

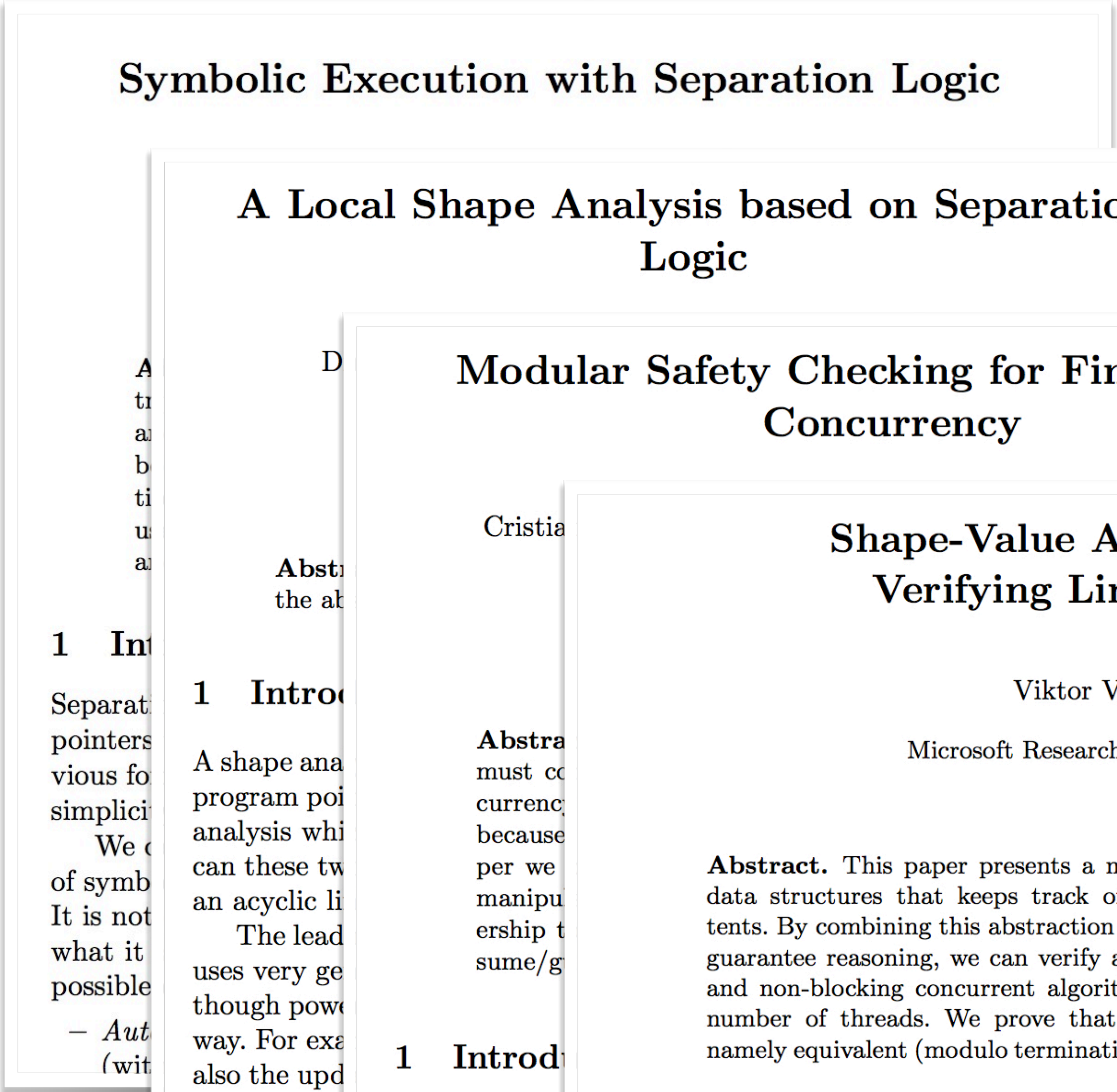
Cristiano Calcagno[1], Matthew Parkinson[2], and Viktor Vafeiadis[2]

[1] Imperial College, London
[2] University of Cambridge

Abstract. Concurrent programs are difficult to verify because the proof must consider the interactions between the threads. Fine-grained concurrency and heap allocated data structures exacerbate this problem, because threads interfere more often and in richer ways. In this paper we provide a thread-modular safety checker for a class of pointer-manipulating fine-grained concurrent algorithms. Our checker uses ownership to avoid interference whenever possible, and rely/guarantee (assume/guarantee) to deal with interference when it genuinely exists.

1 Introduction

Traditional concurrent implementations use a single synchronisation mechanism, such as a lock, to guard an entire data structure (such as a list or a hash table).



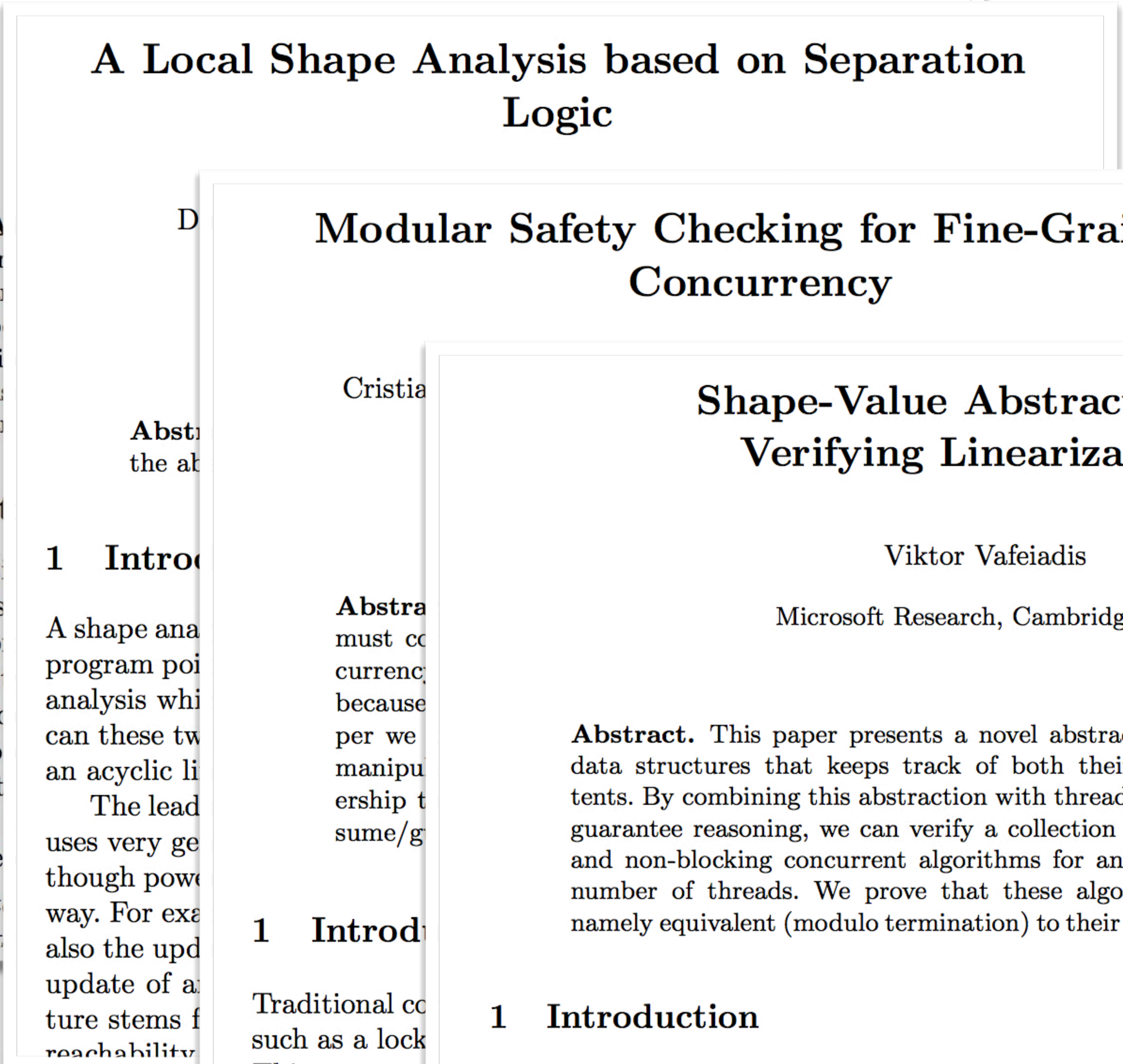
Symbolic Execution with Separation Logic

A
tr
a
b
ti
u
a

1 In

Separat
pointers
vious fo
simplici
We
of symb
It is not
what it
possible

– Aut
(wit



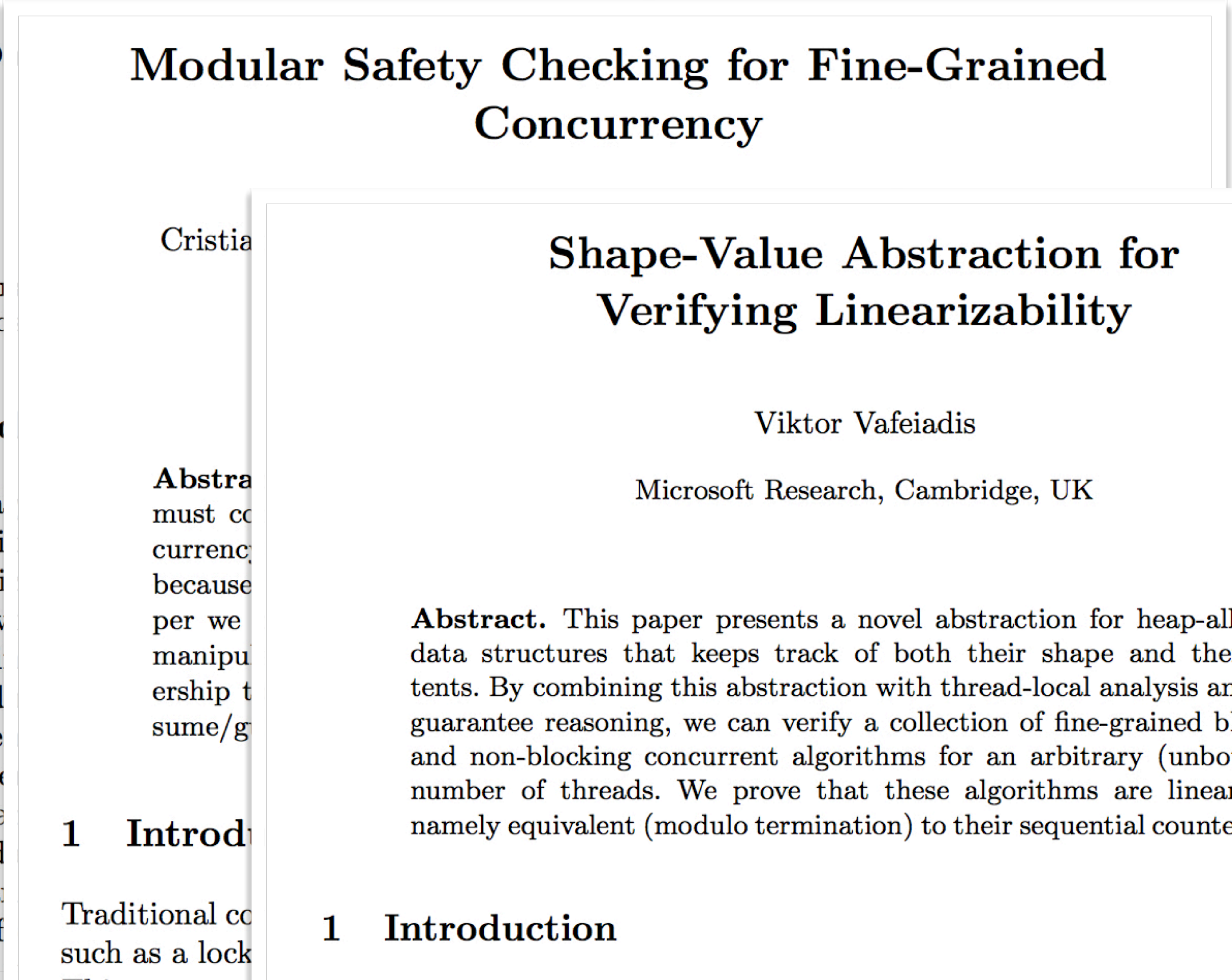
A Local Shape Analysis based on Separation Logic

D

Abst
the ab

1 Intro

A shape ana
program poi
analysis whi
can these tw
an acyclic li
an acyclic li
The lead
uses very ge
though powe
way. For exa
also the upd
update of a
ture stems f
reachability



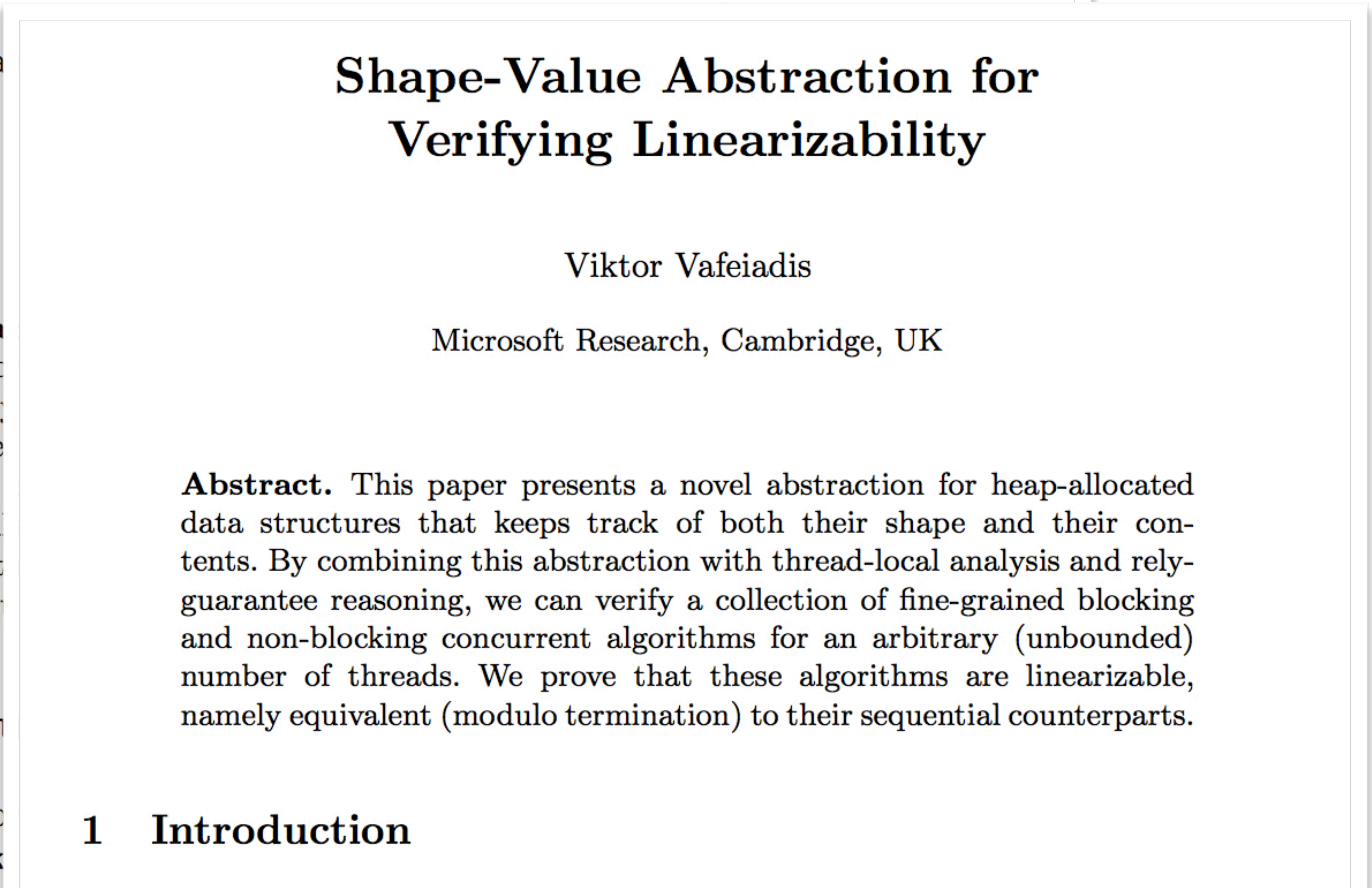
Modular Safety Checking for Fine-Grained Concurrency

Cristia

Abstra
must co
currenc
because
per we
manipu
ership t
sume/g

1 Introd

Traditional co
such as a lock



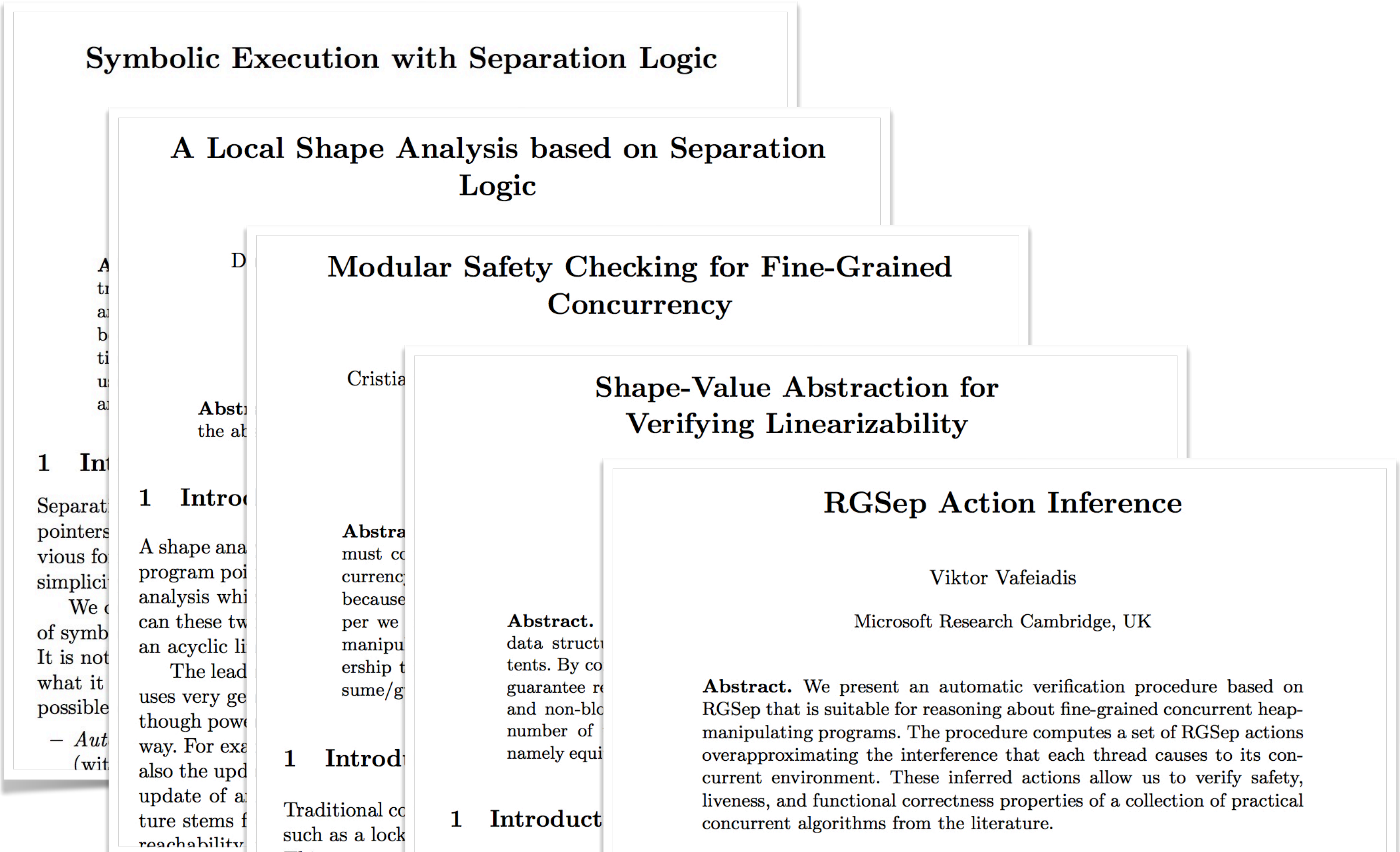
Shape-Value Abstraction for Verifying Linearizability

Viktor Vafeiadis

Microsoft Research, Cambridge, UK

Abstract. This paper presents a novel abstraction for heap-allocated data structures that keeps track of both their shape and their contents. By combining this abstraction with thread-local analysis and rely-guarantee reasoning, we can verify a collection of fine-grained blocking and non-blocking concurrent algorithms for an arbitrary (unbounded) number of threads. We prove that these algorithms are linearizable, namely equivalent (modulo termination) to their sequential counterparts.

1 Introduction



Symbolic Execution with Separation Logic

A
tr
ar
b
ti
u
ar

1 Int

Separati
pointers
vious fo
simplici
We c
of symb
It is not
what it
possible

– Aut
(wit

A Local Shape Analysis based on Separation Logic

D

Abstr
the ab

1 Introd

A shape ana
program poi
analysis whi
can these tw
an acyclic li
an acyclic li
The lead
uses very ge
though powe
way. For exa
also the upd
update of a
ture stems f
reachability

Modular Safety Checking for Fine-Grained Concurrency

Cristia

Abstra
must co
currency
because
per we
manipu
ership t
sume/g

1 Introdu

Traditional co
such as a lock

Shape-Value Abstraction for Verifying Linearizability

Abstract.
data structu
tents. By co
guarantee re
and non-blo
number of
namely equi

1 Introduct

RGSep Action Inference

Viktor Vafeiadis

Microsoft Research Cambridge, UK

Abstract. We present an automatic verification procedure based on RGSep that is suitable for reasoning about fine-grained concurrent heap-manipulating programs. The procedure computes a set of RGSep actions overapproximating the interference that each thread causes to its concurrent environment. These inferred actions allow us to verify safety, liveness, and functional correctness properties of a collection of practical concurrent algorithms from the literature.

Symbolic Execution with Separation Logic

A
tr
ar
b
ti
u
ar

1 Int

Separati
pointers
vious fo
simplici
We
of symb
It is not
what it
possible
– Aut
(wit

A Local Shape Analysis based on Separation Logic

D

Abstr
the ab

1 Introd

A shape ana
program poi
analysis whi
can these tw
an acyclic li
The lead
uses very ge
though powe
way. For exa
also the upd
update of a
ture stems f
reachability

Modular Safety Checking for Fine-Grained Concurrency

Cristia

Abstra
must co
currency
because
per we
manipu
ership t
sume/g

1 Introdu

Traditional co
such as a lock

Shape-Value Abstraction for Verifying Linearizability

Abstract.
data structu
tents. By co
guarantee re
and non-blo
number of
namely equi

1 Introduct

RGSep Action Inference

Abstr
RGSe
manip
overap
curren
livenes
concur

Automatically Proving Linearizability

Viktor Vafeiadis

University of Cambridge

Abstract. This paper presents a practical automatic verification procedure for proving linearizability (i.e., atomicity and functional correctness) of concurrent data structure implementations. The procedure employs a

A Local Shape Analysis based on Separation Logic

SL Symbolic Execution

- Idea: Automatically prove assertions about the shape of data structures in the memory
- Use symbolic execution
  - Abstract Domain: Separation Logic Formulae
- Provides an algorithm for checking SL properties
  - Eg. Memory Safety
- We will consider linked-list data structures

A (very) simple language

Syntax

$$b ::= E{=}E \mid E{\neq}E$$

$$p ::= x{:=}\,E \mid x{:=}\,[E] \mid [E]{:=}\,F \mid \mathbf{new}(x) \mid \mathbf{dispose}(E)$$

$$c ::= p \mid c\,;c \mid \mathbf{while}\ b\ \mathbf{do}\ c \mid \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c$$

A (very) simple language

Syntax

$$b ::= E{=}E \mid E{\neq}E$$

$$p ::= x{:=}\,E \mid x{:=}\,[E] \mid [E]{:=}\,F \mid \mathbf{new}(x) \mid \mathbf{dispose}(E)$$

$$c ::= p \mid c\,;c \mid \mathbf{while}\; b\; \mathbf{do}\; c \mid \mathbf{if}\; b\; \mathbf{then}\; c\; \mathbf{else}\; c$$

Semantics

Heap: $Loc \rightarrow (Field \rightarrow Val)$

$Loc \subseteq Val$

Stack: $Var \rightarrow Val$

$$s, h, c \Rightarrow s, h$$

A (very) simple language

Syntax

$$b ::= E{=}E \mid E{\neq}E$$

$$p ::= x{:=}\,E \mid x{:=}\,[E] \mid [E]{:=}\,F \mid \mathbf{new}(x) \mid \mathbf{dispose}(E)$$

$$c ::= p \mid c\,;c \mid \mathbf{while}\ b\ \mathbf{do}\ c \mid \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c$$

Semantics

Heap: $Loc \to (Field \to Val)$

$Loc \subseteq Val$

Stack: $Var \to Val$

$$s, h, c \Rightarrow s, h$$

$$\frac{\mathcal{C}[\![E]\!]s = n}{s, h,\ x{:=}\,E \implies (s \mid x \mapsto n), h} \qquad \frac{\mathcal{C}[\![E]\!]s = \ell \quad h(\ell) = n}{s, h,\ x{:=}\,[E] \implies (s \mid x \mapsto n), h}$$

$$\frac{\mathcal{C}[\![E]\!]s = \ell \quad \mathcal{C}[\![F]\!]s = n \quad \ell \in \mathsf{dom}(h)}{s, h,\ [E]{:=}\,F \implies s, (h \mid \ell \mapsto n)} \qquad \frac{\ell \notin \mathsf{dom}(h)}{s, h,\ \mathbf{new}(x) \implies (s \mid x \mapsto \ell), (h \mid \ell \mapsto n)}$$

$$\frac{\mathcal{C}[\![E]\!]s = \ell}{s, h * [\ell \mapsto n],\ \mathbf{dispose}(E) \implies s, h} \qquad \frac{\mathcal{C}[\![E]\!]s \notin \mathsf{dom}(h)}{s, h,\ A(E) \implies \top}$$

$$A(E) ::= [E]{:=}\,F \mid x{:=}\,[E] \mid \mathbf{dispose}(E)$$

A (very) simple language

Syntax

$$b ::= E{=}E \mid E{\neq}E$$

$$p ::= x{:=}\,E \mid x{:=}\,[E] \mid [E]{:=}\,F \mid \mathbf{new}(x) \mid \mathbf{dispose}(E)$$

$$c ::= p \mid c\,;c \mid \mathbf{while}\ b\ \mathbf{do}\ c \mid \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c$$

Semantics

Heap: $Loc \to (Field \to Val)$

$Loc \subseteq Val$

Stack: $Var \to Val$

$$s, h, c \Rightarrow s, h$$

$$\frac{\mathcal{C}[\![E]\!]s = n}{s, h,\ x{:=}\,E \Longrightarrow (s \mid x \mapsto n), h} \qquad \frac{\mathcal{C}[\![E]\!]s = \ell \quad h(\ell) = n}{s, h,\ x{:=}\,[E] \Longrightarrow (s \mid x \mapsto n), h}$$

$$\frac{\mathcal{C}[\![E]\!]s = \ell \quad \mathcal{C}[\![F]\!]s = n \quad \ell \in \mathsf{dom}(h)}{s, h,\ [E]{:=}\,F \Longrightarrow s, (h \mid \ell \mapsto n)} \qquad \frac{\ell \notin \mathsf{dom}(h)}{s, h,\ \mathbf{new}(x) \Longrightarrow (s \mid x \mapsto \ell), (h \mid \ell \mapsto n)}$$

$$\frac{\mathcal{C}[\![E]\!]s = \ell}{s, h * [\ell \mapsto n],\ \mathbf{dispose}(E) \Longrightarrow s, h} \qquad \frac{\mathcal{C}[\![E]\!]s \notin \mathsf{dom}(h)}{s, h,\ A(E) \Longrightarrow \top}$$

$$A(E) ::= [E]{:=}\,F \mid x{:=}\,[E] \mid \mathbf{dispose}(E)$$

`if`, `while`, sequential composition are as usual

Abstract Domain

Symbolic Heaps $\Pi \mid \Sigma$

Abstract Domain

Symbolic Heaps

Pure

$E = F$

$\Pi \mid \Sigma$

Abstract Domain

Symbolic Heaps

Pure

$E = F$

$\Pi \mid \Sigma$

Spatial

$E \mapsto F$

$\mathsf{ls}(E, F)$

junk

Abstract Domain

Symbolic Heaps

Pure

$E = F$

$\Pi \mid \Sigma$

Spatial

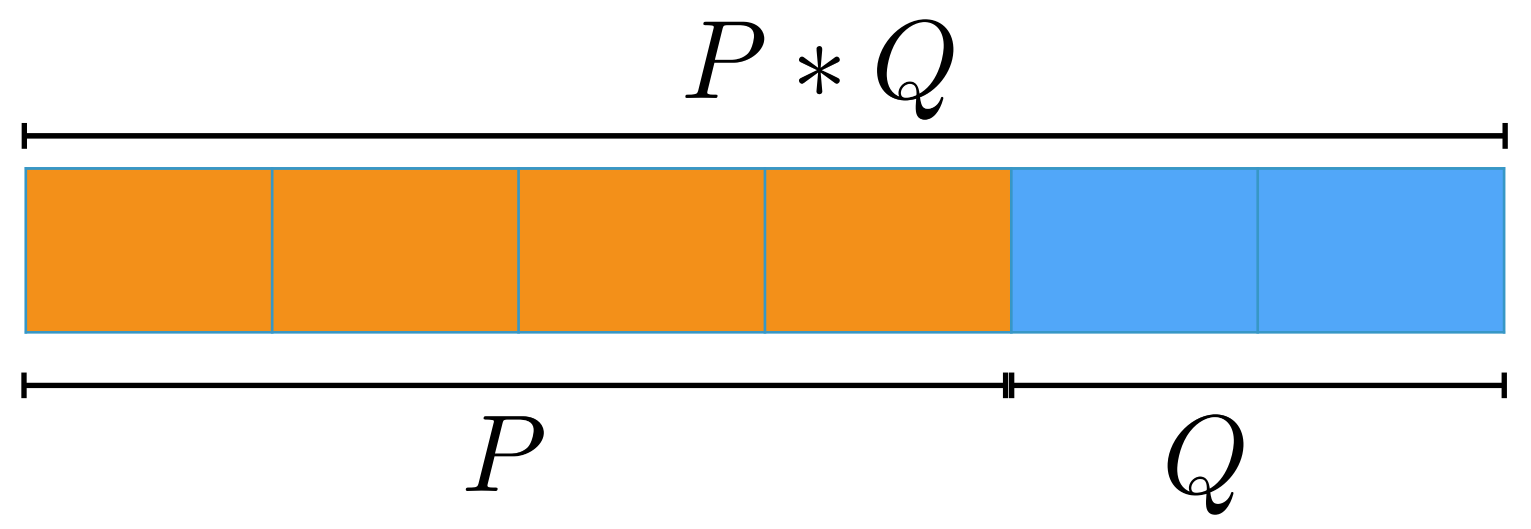
$E \mapsto F$

ls(E, F)

junk

Intuitively

$P * Q$



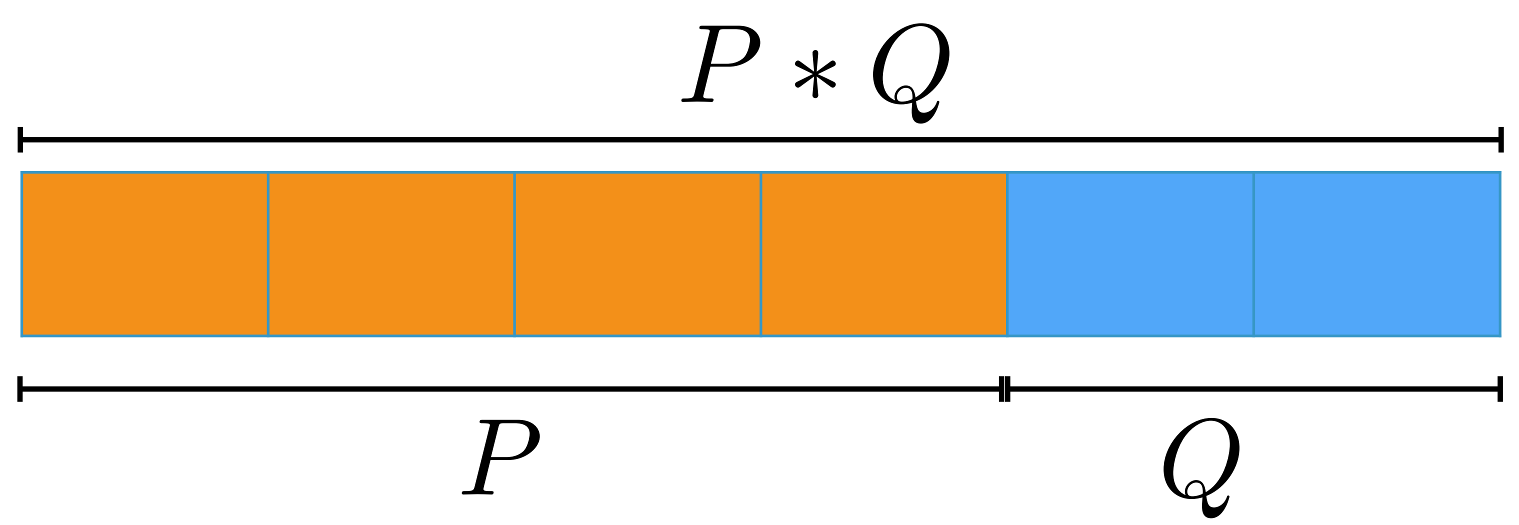
Abstract Domain

Symbolic Heaps

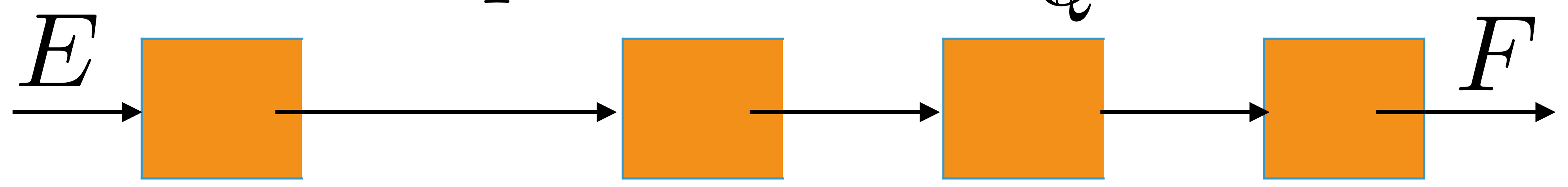
Pure	$\Pi \mid \Sigma$	Spatial
$E = F$		$E \mapsto F$
		$\mathsf{ls}(E, F)$
		junk

Intuitively

$P * Q$



$\mathsf{ls}(E, F)$



$$\mathsf{ls}(E, F) \iff E \neq F \wedge (E \mapsto F \vee (\exists x'. E \mapsto x' * \mathsf{ls}(x', F)))$$

Abstract Domain

Symbolic Heaps

Pure

$E = F$

$\Pi \mid \Sigma$

Spatial

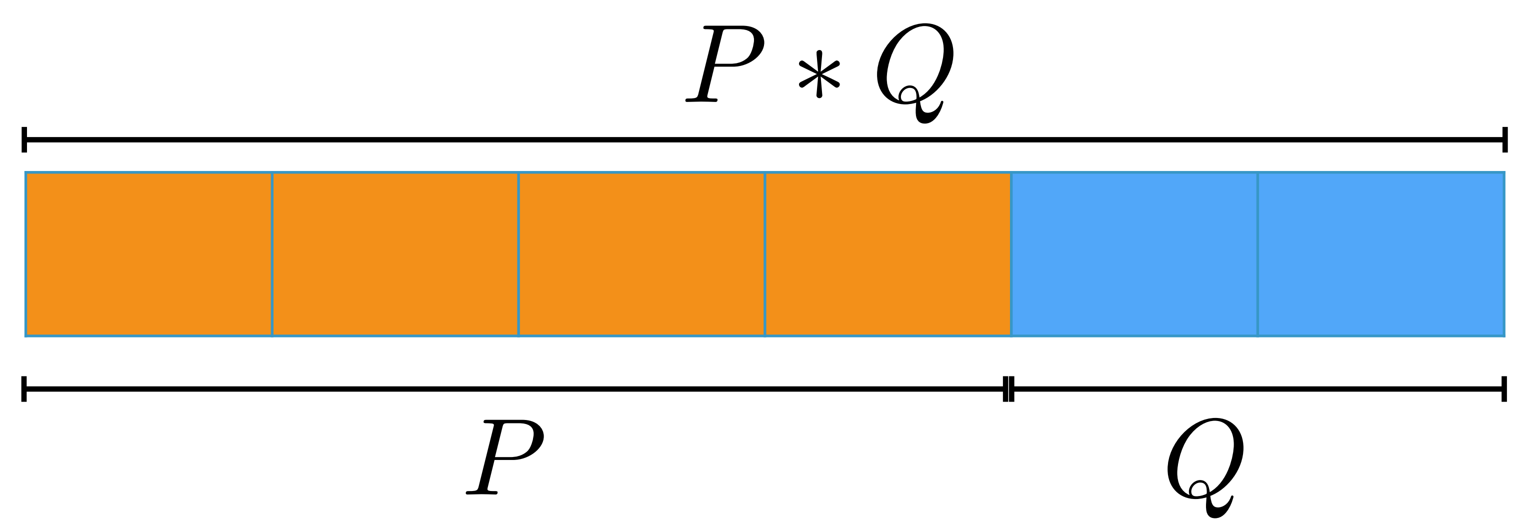
$E \mapsto F$

$\mathsf{ls}(E, F)$

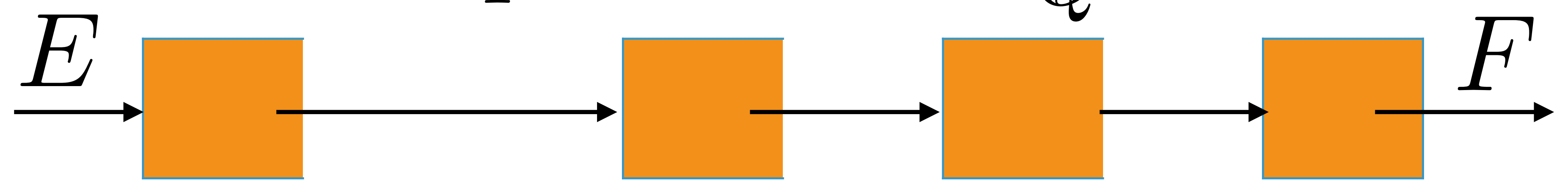
junk

Intuitively

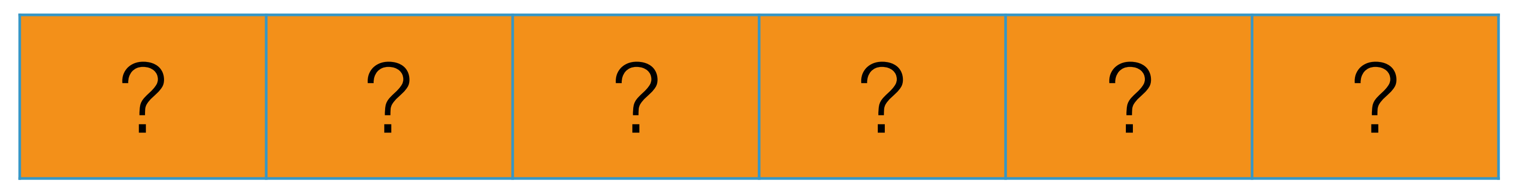
$P * Q$



$\mathsf{ls}(E, F)$



junk



Abstract Domain

Symbolic Heaps

Pure

$E = F$

$\Pi \mid \Sigma$

Spatial

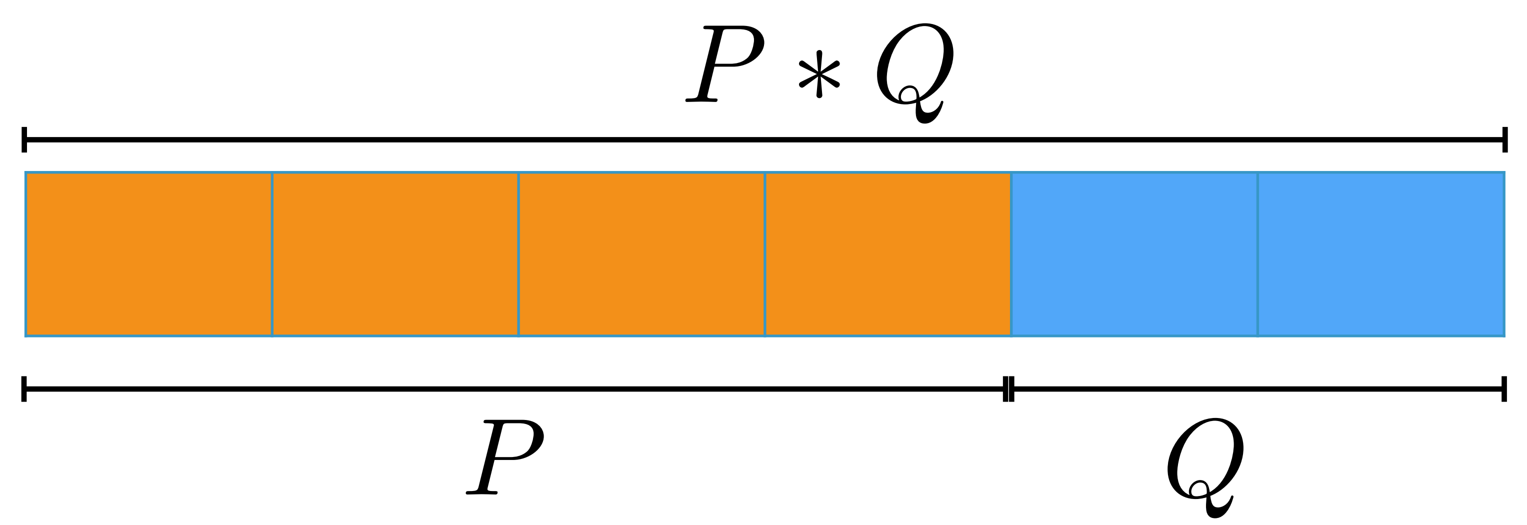
$E \mapsto F$

$\mathsf{ls}(E, F)$

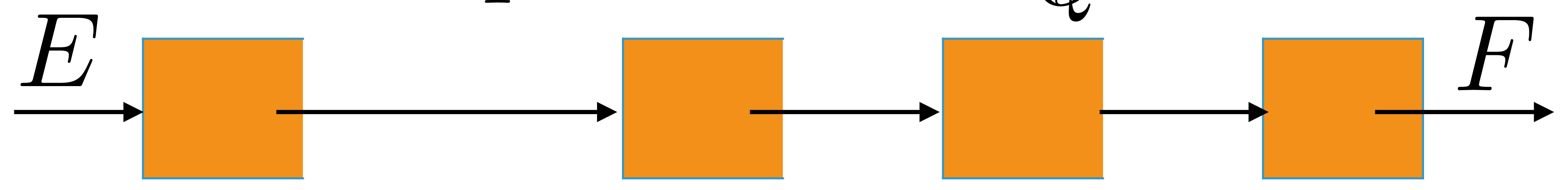
junk

Intuitively

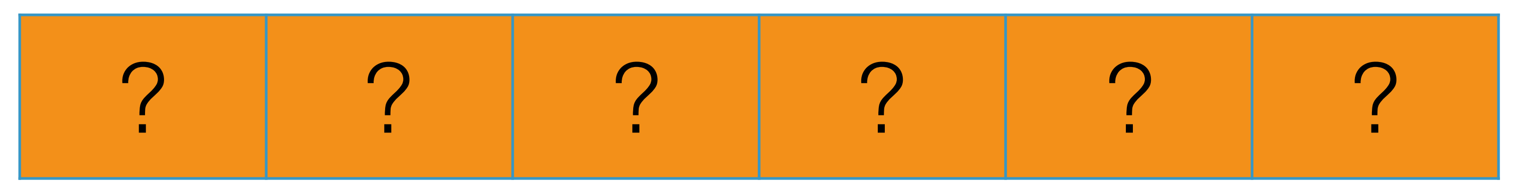
$P * Q$



$\mathsf{ls}(E, F)$



junk



Interpretation

$$\exists x'_1 x'_2 \dots x'_n . \Big(\bigwedge_{P \in \Pi} P \Big) \wedge \Big(\bigstar_{Q \in \Sigma} Q \Big)$$

Symbolic States Semantics

Pure Part 🟨

$$
\begin{array}{rcl}
s \models \{\} & \iff & true \\
s \models E = F & \iff & \mathcal{C}(E)s = \mathcal{C}(F)s \\
s \models \Pi_0 \cup \Pi_1 & \iff & s \models \Pi_0 \wedge s \models \Pi_1
\end{array}
$$

Symbolic States Semantics

Pure Part 🟨

$$
\begin{array}{rcl}
s \models \{\} & \iff & \mathit{true} \\
s \models E = F & \iff & \mathcal{C}(E)s = \mathcal{C}(F)s \\
s \models \Pi_0 \cup \Pi_1 & \iff & s \models \Pi_0 \wedge s \models \Pi_1
\end{array}
$$

Spatial Part 🟧

$$
\begin{array}{rcl}
s, h \models \{\} & \iff & h = \emptyset \\
s, h \models E \mapsto F & \iff & \mathcal{C}(E)s = p \wedge \mathcal{C}(F)s = v \wedge h = p \mapsto v \\
s, h \models \mathsf{ls}(E, F) & \iff & \mathcal{C}(E)s = p \wedge \mathcal{C}(F)s = q \wedge \left(\begin{array}{c} h \text{ contains a non-cyclic} \\ \text{path from } p \text{ to } q \end{array} \right) \\
s, h \models \mathsf{junk} & \iff & h \neq \emptyset \\
s, h \models P * Q & \iff & \exists h_0, h_1 . h = h_0 \cup h_1 \wedge s, h_0 \models P \wedge s, h_1 \models Q
\end{array}
$$

Symbolic States Semantics

Pure Part 🟨

$$\begin{aligned}
s \vDash \{\} &\iff true \\
s \vDash E = F &\iff \mathcal{C}(E)s = \mathcal{C}(F)s \\
s \vDash \Pi_0 \cup \Pi_1 &\iff s \vDash \Pi_0 \land s \vDash \Pi_1
\end{aligned}$$

Spatial Part 🟧

$$\begin{aligned}
s, h \vDash \{\} &\iff h = \emptyset \\
s, h \vDash E \mapsto F &\iff \mathcal{C}(E)s = p \land \mathcal{C}(F)s = v \land h = p \mapsto v \\
s, h \vDash \mathsf{ls}(E, F) &\iff \mathcal{C}(E)s = p \land \mathcal{C}(F)s = q \land \left(\begin{array}{c} h \text{ contains a non-cyclic} \\ \text{path from } p \text{ to } q \end{array}\right) \\
s, h \vDash \mathsf{junk} &\iff h \neq \emptyset \\
s, h \vDash P * Q &\iff \exists h_0, h_1 . h = h_0 \cup h_1 \land s, h_0 \vDash P \land s, h_1 \vDash Q
\end{aligned}$$

Whole State 🟨 🟧

$$s, h \vDash \mathit{\Pi} | \mathit{\Sigma} \iff \exists \mathbf{v}'.\ (s(\mathbf{x}' \mapsto \mathbf{v}') \vDash \mathit{\Pi}) \land (s(\mathbf{x}' \mapsto \mathbf{v}'), h \vDash \mathit{\Sigma})$$

Decidable Entailment

Queries

$$\begin{array}{ll} \Pi \vdash E = F & \Pi | \Sigma \vdash E \neq F \text{ when } \mathsf{Vars}'(E, F) = \emptyset \\ \Pi | \Sigma \vdash \mathsf{false} & \Pi | \Sigma \vdash \mathsf{Allocated}(\mathsf{E}) \text{ when } \mathsf{Vars}'(\mathsf{E}) = \emptyset \end{array}$$

Decidable Entailment

Queries

$$\Pi \vdash E = F \qquad \Pi | \Sigma \vdash E \neq F \text{ when } \mathsf{Vars}'(E, F) = \emptyset$$
$$\Pi | \Sigma \vdash \mathsf{false} \qquad \Pi | \Sigma \vdash \mathsf{Allocated}(\mathsf{E}) \text{ when } \mathsf{Vars}'(\mathsf{E}) = \emptyset$$

Calculation

$$\Pi \vdash E = F \iff \begin{array}{c} E \text{ and } F \text{ are in the same} \\ \text{equivalence class cf. } \Pi \end{array}$$

Decidable Entailment

Queries

$$\Pi \vdash E = F \qquad \Pi | \Sigma \vdash E \neq F \text{ when } \mathsf{Vars}'(E, F) = \emptyset$$
$$\Pi | \Sigma \vdash \mathsf{false} \qquad \Pi | \Sigma \vdash \mathsf{Allocated}(\mathsf{E}) \text{ when } \mathsf{Vars}'(\mathsf{E}) = \emptyset$$

Calculation

$$\Pi \vdash E = F \iff \begin{array}{c} E \text{ and } F \text{ are in the same} \\ \text{equivalence class cf. } \Pi \end{array}$$

$$\Pi, \Sigma \vdash E \neq F \iff E = F \wedge \Pi, \Sigma \vdash \mathsf{false}$$

Decidable Entailment

Queries

$$\mathit{\Pi} \vdash E = F \qquad \mathit{\Pi}|\mathit{\Sigma} \vdash E \neq F \text{ when } \mathsf{Vars}'(E,F) = \emptyset$$
$$\mathit{\Pi}|\mathit{\Sigma} \vdash \mathsf{false} \qquad \mathit{\Pi}|\mathit{\Sigma} \vdash \mathsf{Allocated}(\mathsf{E}) \text{ when } \mathsf{Vars}'(\mathsf{E}) = \emptyset$$

Calculation

$$\mathit{\Pi} \vdash E = F \iff \begin{array}{c} E \text{ and } F \text{ are in the same} \\ \text{equivalence class cf. } \mathit{\Pi} \end{array}$$

$$\mathit{\Pi}, \mathit{\Sigma} \vdash E \neq F \iff E = F \wedge \mathit{\Pi}, \mathit{\Sigma} \vdash \mathsf{false}$$

$$\mathit{\Pi}, \mathit{\Sigma} \vdash \mathsf{false} \iff \begin{array}{l} (\exists E.\ \mathit{\Pi} \vdash E = \mathsf{nil} \wedge \mathsf{allocated}(\mathit{\Sigma}, \mathsf{E})) \vee \\ (\exists E, F.\ \mathit{\Pi} \vdash E = F \wedge \mathsf{ls}(E,F) \in \Sigma) \vee \\ (\exists E, F.\ \mathit{\Pi} \vdash E = F \wedge \left(\begin{array}{c} \{E \mapsto \_, \mathsf{ls}(E, \_)\} \cap \Sigma \neq \emptyset) \wedge \\ \{F \mapsto \_, \mathsf{ls}(F, \_)\} \cap \Sigma \neq \emptyset) \end{array} \right) \end{array}$$

$$\mathsf{allocated}(\mathit{\Sigma}, E) = \exists E',\ (E \mapsto E' \in \mathit{\Sigma}) \vee (\mathsf{ls}(E, E') \in \mathit{\Sigma})$$

Decidable Entailment

Queries

$$\Pi \vdash E = F \qquad \Pi | \Sigma \vdash E \neq F \text{ when } \mathsf{Vars}'(E,F) = \emptyset$$
$$\Pi | \Sigma \vdash \mathsf{false} \qquad \Pi | \Sigma \vdash \mathsf{Allocated}(\mathsf{E}) \text{ when } \mathsf{Vars}'(\mathsf{E}) = \emptyset$$

Calculation

$$\Pi \vdash E = F \iff \begin{array}{c} E \text{ and } F \text{ are in the same} \\ \text{equivalence class cf. } \Pi \end{array}$$

$$\Pi, \Sigma \vdash E \neq F \iff E = F \wedge \Pi, \Sigma \vdash \mathsf{false}$$

$$\Pi, \Sigma \vdash \mathsf{false} \iff \begin{array}{l} (\exists E.\ \Pi \vdash E = \mathsf{nil} \wedge \mathsf{allocated}(\Sigma, \mathsf{E})) \vee \\ (\exists E, F.\ \Pi \vdash E = F \wedge \mathsf{ls}(E,F) \in \Sigma) \vee \\ (\exists E, F.\ \Pi \vdash E = F \wedge \left(\begin{array}{c} \{E \mapsto \_, \mathsf{ls}(E, \_)\} \cap \Sigma \neq \emptyset) \wedge \\ \{F \mapsto \_, \mathsf{ls}(F, \_)\} \cap \Sigma \neq \emptyset) \end{array} \right) \end{array}$$

$$\mathsf{allocated}(\Sigma, E) = \exists E',\ (E \mapsto E' \in \Sigma) \vee (\mathsf{ls}(E, E') \in \Sigma)$$

$$\Pi, \Sigma \vdash \mathsf{Allocated}(\mathsf{E}) \iff \begin{array}{l} \Pi, \Sigma \vdash \mathsf{false} \vee \\ (\exists E'.\ \Pi \vdash E = E' \wedge \mathsf{allocated}(\Sigma, E')) \end{array}$$

Symbolic Execution

$$\sigma, c \Rightarrow \sigma'$$

Symbolic Execution

Command
(atomic)

Symbolic State
Pre-Condition

$$\sigma, c \Rightarrow \sigma'$$

Symbolic State
Post-Condition

Symbolic Execution

Command
(atomic)

Symbolic State
Pre-Condition

$$\sigma, c \Rightarrow \sigma'$$

Symbolic State
Post-Condition

$$\begin{array}{rrcl}
\Pi|\Sigma, & x := E & \Rightarrow & x = E[x'/x] \wedge (\Pi|\Sigma)[x'/x] \\
\Pi|\Sigma * E \mapsto F, & x := [E] & \Rightarrow & x = F[x'/x] \wedge (\Pi|\Sigma * E \mapsto F)[x'/x] \\
\Pi|\Sigma * E \mapsto F, & [E] := G & \Rightarrow & x = \Pi|\Sigma * E \mapsto G \\
\Pi|\Sigma, & \mathsf{new}(x) & \Rightarrow & x = (\Pi|\Sigma)[x'/x] * x \mapsto y' \\
\Pi|\Sigma * E \mapsto F, & \mathsf{dispose}(E) & \Rightarrow & x = (\Pi|\Sigma)
\end{array}$$

Symbolic Execution

Command (atomic)

Symbolic State Pre-Condition

$$\sigma, c \Rightarrow \sigma'$$

Symbolic State Post-Condition

$$\begin{array}{rrcl}
\Pi|\Sigma, & x := E & \Rightarrow & x = E[x'/x] \wedge (\Pi|\Sigma)[x'/x] \\
\Pi|\Sigma * E \mapsto F, & x := [E] & \Rightarrow & x = F[x'/x] \wedge (\Pi|\Sigma * E \mapsto F)[x'/x] \\
\Pi|\Sigma * E \mapsto F, & [E] := G & \Rightarrow & x = \Pi|\Sigma * E \mapsto G \\
\Pi|\Sigma, & \mathsf{new}(x) & \Rightarrow & x = (\Pi|\Sigma)[x'/x] * x \mapsto y' \\
\Pi|\Sigma * E \mapsto F, & \mathsf{dispose}(E) & \Rightarrow & x = (\Pi|\Sigma)
\end{array}$$

$$\frac{\Pi, \Sigma \nvdash \mathsf{Allocated}(E)}{\Pi, \Sigma,\ A(E) \ \Rightarrow\ \top}$$

Symbolic Execution

Command (atomic)

Symbolic State Pre-Condition

$$\sigma, c \Rightarrow \sigma'$$

Symbolic State Post-Condition

$$\begin{array}{rrcl}
\Pi|\Sigma, & x := E & \Rightarrow & x = E[x'/x] \wedge (\Pi|\Sigma)[x'/x] \\
\Pi|\Sigma * E \mapsto F, & x := [E] & \Rightarrow & x = F[x'/x] \wedge (\Pi|\Sigma * E \mapsto F)[x'/x] \\
\Pi|\Sigma * E \mapsto F, & [E] := G & \Rightarrow & x = \Pi|\Sigma * E \mapsto G \\
\Pi|\Sigma, & \mathsf{new}(x) & \Rightarrow & x = (\Pi|\Sigma)[x'/x] * x \mapsto y' \\
\Pi|\Sigma * E \mapsto F, & \mathsf{dispose}(E) & \Rightarrow & x = (\Pi|\Sigma)
\end{array}$$

$$\frac{\Pi, \Sigma \nvdash \mathsf{Allocated}(E)}{\Pi, \Sigma,\ A(E) \ \Rightarrow\ \top}$$

Rearrangment $P(E,F) \ ::= \ E \mapsto F \mid \mathsf{ls}(E,F)$

$$\frac{\Pi_0 \mid \Sigma_0 * P(E,G),\ A(E) \Longrightarrow \Pi_1 \mid \Sigma_1}{\Pi_0 \mid \Sigma_0 * P(F,G),\ A(E) \Longrightarrow \Pi_1 \mid \Sigma_1}\ \Pi_0 \vdash E{=}F$$

$$\frac{\Pi_0 \mid \Sigma_0 * E \mapsto x' * \mathsf{ls}(x',G),\ A(E) \Longrightarrow \Pi_1 \mid \Sigma_1}{\Pi_0 \mid \Sigma_0 * \mathsf{ls}(E,G),\ A(E) \Longrightarrow \Pi_1 \mid \Sigma_1} \qquad \frac{\Pi \mid \Sigma * E \mapsto F,\ A(E) \Longrightarrow \Pi' \mid \Sigma'}{\Pi \mid \Sigma * \mathsf{ls}(E,F),\ A(E) \Longrightarrow \Pi' \mid \Sigma'}$$

Abstraction

- Reduce the number of existential variables to guarantee a finite domain: termination

Abstraction

- Reduce the number of existential variables to guarantee a finite domain: termination

Canonicalization

$$\frac{}{E{=}x' \wedge \Pi \mid \Sigma \rightsquigarrow (\Pi \mid \Sigma)[E/x']}\ \text{(St1)} \qquad \frac{}{x'{=}E \wedge \Pi \mid \Sigma \rightsquigarrow (\Pi \mid \Sigma)[E/x']}\ \text{(St2)}$$

$$\frac{x' \notin \mathsf{Vars}'(\Pi, \Sigma)}{\Pi \mid \Sigma * P(x', E) \rightsquigarrow \Pi \mid \Sigma \cup \mathsf{junk}}\ \text{(Gb1)} \qquad \frac{x', y' \notin \mathsf{Vars}'(\Pi, \Sigma)}{\Pi \mid \Sigma * P_1(x', y') * P_2(y', x') \rightsquigarrow \Pi \mid \Sigma \cup \mathsf{junk}}\ \text{(Gb2)}$$

$$\frac{x' \notin \mathsf{Vars}'(\Pi, \Sigma, E, F) \qquad \Pi \vdash F{=}\mathsf{nil}}{\Pi \mid \Sigma * P_1(E, x') * P_2(x', F) \rightsquigarrow \Pi \mid \Sigma * \mathsf{ls}(E, \mathsf{nil})}\ \text{(Abs1)}$$

$$\frac{x' \notin \mathsf{Vars}'(\Pi, \Sigma, E, F, G, H) \quad \Pi \vdash F{=}G}{\Pi \mid \Sigma * P_1(E, x') * P_2(x', F) * P_3(G, H) \rightsquigarrow \Pi \mid \Sigma * \mathsf{ls}(E, F) * P_3(G, H)}\ \text{(Abs2)}$$

Algorithm

- For each atomic command:
  - Canonicalize the symbolic state (ST) to obtain a canonical symbolic state (CST)
  - Execute the symbolic semantics on the atomic step
- For each composite command
  - Use the composition rules using the atomic rules in each step

Example

```
$\{\} \mid \{\mathsf{ls}(c, 0)\}$
$p := 0;$
$\textbf{while}\ (c \neq 0)\ \textbf{do}$
    $n := c \to tl;$
    $c \to tl := p;$
    $p := c;$
    $c := n$
$\textbf{od}$
$\{c = 0 \wedge c = n \wedge n = 0\} \mid \{\mathsf{ls}(p, 0)\} \vee \{c = 0 \wedge c = n \wedge n = 0\} \mid \{p \mapsto 0\}$
```

Example

```
$\{\} \mid \{\mathsf{ls}(c,0)\}$
$p := 0;$
$\textbf{while } (c \neq 0) \textbf{ do}$
        $n := c \to tl;$
        $c \to tl := p;$
        $p := c;$
        $c := n$
$\textbf{od}$
$\{c = 0 \wedge c = n \wedge n = 0\} \mid \{\mathsf{ls}(p,0)\} \vee \{c = 0 \wedge c = n \wedge n = 0\} \mid \{p \mapsto 0\}$
```

Loop Invariant:

$$\begin{aligned}
&\{p = 0\} \mid \{\mathsf{ls}(c,0)\} \vee \\
&\quad \{c = n \wedge n = 0\} \mid \{p \mapsto 0\} \vee \\
&\quad\quad \{c = n \wedge n = 0\} \mid \{\mathsf{ls}(p,0)\} \vee \\
&\quad\quad\quad \{c = n\} \mid \{p \mapsto 0 * \mathsf{ls}(n,0)\} \vee \\
&\quad\quad\quad\quad \{c = n\} \mid \{\mathsf{ls}(p,0) * \mathsf{ls}(n,0)\}
\end{aligned}$$

Modular Safety Checking for Fine-Grained Concurrency

RGSep Symbolic Execution

- Extend the symbolic execution above to RGSep
- Calculate the interference of the "*environment*" (i.e. other threads)
- Check that the assertions are *stable* w.r.t. interference
- Check *memory safety* for fine-grained concurrent programs
  - We will extend this to Linearizability later

RGSep (Review)

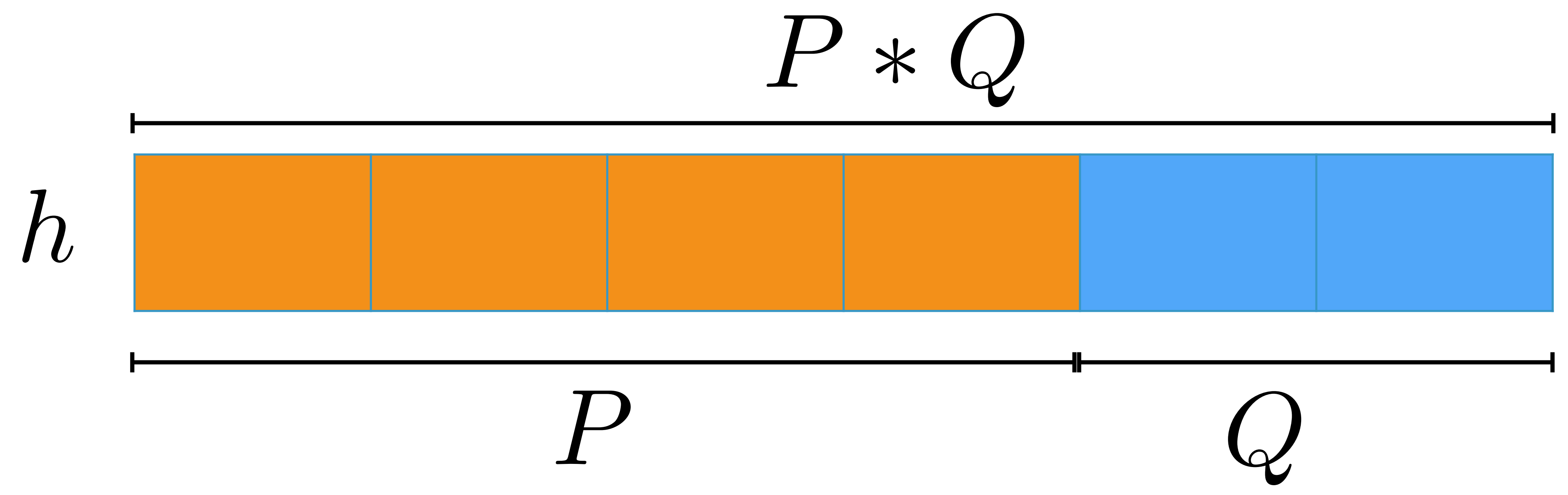
Separation

$$h, i \vDash_{SL} (P * Q) = \exists\ h1\ h2, (h_1 \uplus h_2 = h)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

RGSep (Review)

Separation

$$h, i \vDash_{SL} (P * Q) = \exists\ h1\ h2, (h_1 \uplus h_2 = h)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$



RGSep (Review)

Separation

$$h, i \vDash_{SL} (P * Q) = \exists\ h1\ h2, (h_1 \uplus h_2 = h)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

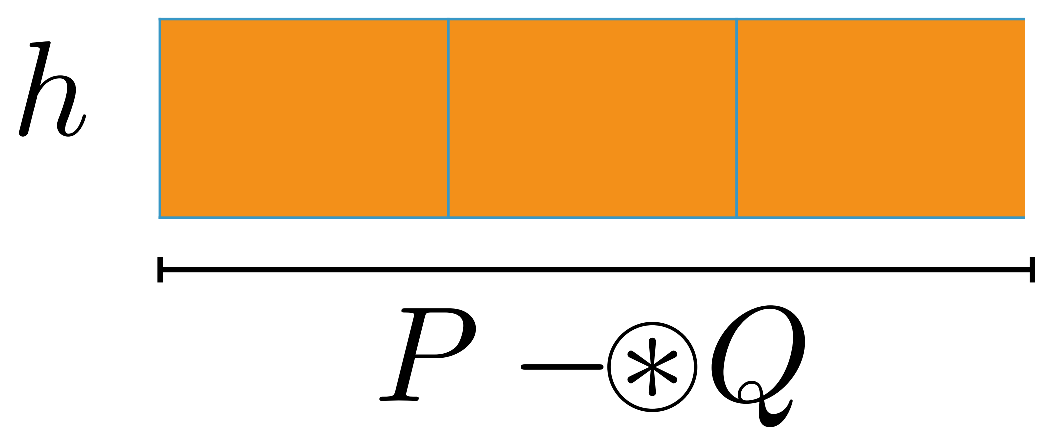
$$h, i \vDash_{SL} (P -\!\circledast Q) = \exists\ h1\ h2, (h_1 \uplus h = h_2)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

RGSep (Review)

Separation

$$h, i \vDash_{SL} (P * Q) = \exists\ h1\ h2, (h_1 \uplus h_2 = h)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

$$h, i \vDash_{SL} (P \mathrel{-\!\circledast} Q) = \exists\ h1\ h2, (h_1 \uplus h = h_2)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

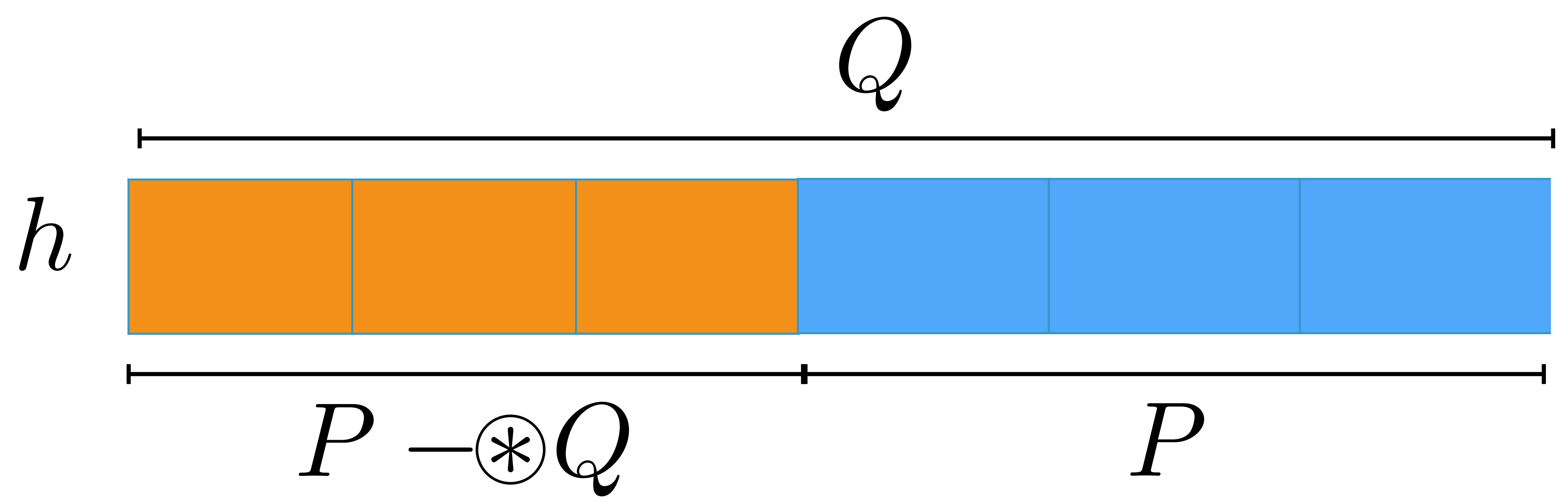


RGSep (Review)

Separation

$$h, i \vDash_{SL} (P * Q) = \exists\ h1\ h2, (h_1 \uplus h_2 = h)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

$$h, i \vDash_{SL} (P \mathrel{-\!\!\circledast} Q) = \exists\ h1\ h2, (h_1 \uplus h = h_2)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$



RGSep (Review)

Separation

$$h, i \vDash_{SL} (P * Q) = \exists\ h1\ h2, (h_1 \uplus h_2 = h)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

$$h, i \vDash_{SL} (P -\!\circledast Q) = \exists\ h1\ h2, (h_1 \uplus h = h_2)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

Locality

$$p, q, r ::= P \mid \boxed{P} \mid p * q \mid p \wedge q \mid p \vee q \mid \exists x.\ p \mid \forall x.\ p$$

RGSep (Review)

Separation

$$h, i \vDash_{SL} (P * Q) = \exists\ h1\ h2, (h_1 \uplus h_2 = h)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

$$h, i \vDash_{SL} (P \mathbin{-\!\circledast} Q) = \exists\ h1\ h2, (h_1 \uplus h = h_2)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

Locality

$$p, q, r ::= \underbrace{P}_{\text{Local State}} \mid \underbrace{\boxed{P}}_{\text{Global State}} \mid p * q \mid p \wedge q \mid p \vee q \mid \exists x.\ p \mid \forall x.\ p$$

RGSep (Review)

Separation

$$h, i \vDash_{SL} (P * Q) = \exists\ h1\ h2, (h_1 \uplus h_2 = h)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

$$h, i \vDash_{SL} (P \mathrel{-\!\!\circledast} Q) = \exists\ h1\ h2, (h_1 \uplus h = h_2)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

Locality

$$p, q, r ::= P \mid \boxed{P} \mid p * q \mid p \wedge q \mid p \vee q \mid \exists x.\ p \mid \forall x.\ p$$

RGSep (Review)

Separation

$$h, i \vDash_{SL} (P * Q) = \exists\ h1\ h2, (h_1 \uplus h_2 = h)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

$$h, i \vDash_{SL} (P -\!\circledast Q) = \exists\ h1\ h2, (h_1 \uplus h = h_2)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

Locality

$$p, q, r ::= P \mid \boxed{P} \mid p * q \mid p \wedge q \mid p \vee q \mid \exists x.\ p \mid \forall x.\ p$$

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\text{Frame}$$

RGSep (Review)

Separation

$$h, i \vDash_{SL} (P * Q) = \exists\ h1\ h2, (h_1 \uplus h_2 = h)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

$$h, i \vDash_{SL} (P -\!\!\circledast Q) = \exists\ h1\ h2, (h_1 \uplus h = h_2)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

Locality

$$p, q, r ::= P \mid \boxed{P} \mid p * q \mid p \wedge q \mid p \vee q \mid \exists x.\ p \mid \forall x.\ p$$

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\text{Frame} \qquad \frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \| C2\ \{Q_1 * Q_2\}}\text{Parallel}$$

RGSep (Review)

Separation

$$h, i \vDash_{SL} (P * Q) = \exists\ h1\ h2, (h_1 \uplus h_2 = h)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

$$h, i \vDash_{SL} (P -\!\!\circledast Q) = \exists\ h1\ h2, (h_1 \uplus h = h_2)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

Locality

$$p, q, r ::= P \mid \boxed{P} \mid p * q \mid p \wedge q \mid p \vee q \mid \exists x.\ p \mid \forall x.\ p$$

Interference

$$[\![P \rightsquigarrow Q]\!] = \{(h_1 \uplus h_0, h_2 \uplus h_0) \mid h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q\}$$

RGSep (Review)

Separation

$$h, i \vDash_{SL} (P * Q) = \exists\ h1\ h2, (h_1 \uplus h_2 = h)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

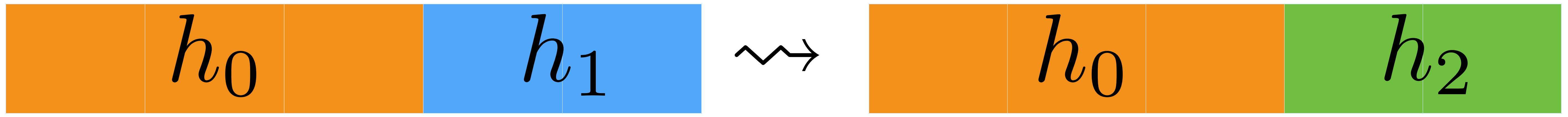
$$h, i \vDash_{SL} (P \mathrel{-\!\circledast} Q) = \exists\ h1\ h2, (h_1 \uplus h = h_2)\ \wedge\ h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q$$

Locality

$$p, q, r ::= P \mid \boxed{P} \mid p * q \mid p \wedge q \mid p \vee q \mid \exists x.\, p \mid \forall x.\, p$$

Interference

$$[\![P \rightsquigarrow Q]\!] = \{(h_1 \uplus h_0, h_2 \uplus h_0) \mid h_1, i \vDash_{SL} P\ \wedge\ h_2, i \vDash_{SL} Q\}$$



RGSep Judgment

$$R, G \vDash \overset{\text{Pre}}{\{P\}}\ c\ \overset{\text{Post}}{\{Q\}}$$

RGSep Judgment

$$R, \underset{\downarrow}{G} \models \overset{\text{Pre}}{\{P\}}\ c\ \overset{\text{Post}}{\{Q\}}$$

Set of Guarantee Actions:
Global actions allowed to this command

RGSep Judgment

$$R, G \vDash \{\overset{\text{Pre}}{P}\}\ c\ \{\overset{\text{Post}}{Q}\}$$

Set of Guarantee Actions:
  Global actions allowed to this command

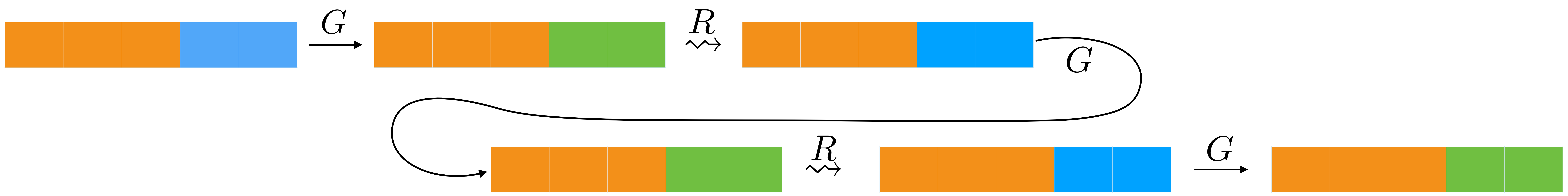
Set of Rely Actions:
  Global actions of other concurrent commands

RGSep Judgment

$$R, G \vDash \overset{\text{Pre}}{\{P\}}\ c\ \overset{\text{Post}}{\{Q\}}$$

Set of Guarantee Actions:
  Global actions allowed to this command

Set of Rely Actions:
  Global actions of other concurrent commands



RGSep Proof Rules

RGSep Proof Rules

$$\frac{R, G \vdash \{P\}\ C\ \{Q\} \qquad F \text{ stable for } (R \cup G) \text{ or } C \text{ has no } \mathsf{atomic}}{R, G \vdash \{P * F\}\ C\ \{Q * F\}}$$

RGSep Proof Rules

$$\frac{R, G \vdash \{P\}\ C\ \{Q\} \qquad F \text{ stable for } (R \cup G) \text{ or } C \text{ has no } \mathsf{atomic}}{R, G \vdash \{P * F\}\ C\ \{Q * F\}}$$

$$\frac{Q \equiv (P * X \mapsto Y) \qquad x \notin fv(P)}{R, G \vdash \{\boxed{\text{Q}} \wedge e = X\}\ x := [e]\ \{\boxed{\text{Q}} * x = Y\}}$$

RGSep Proof Rules

$$\frac{R, G \vdash \{P\}\ C\ \{Q\} \qquad F \text{ stable for } (R \cup G) \text{ or } C \text{ has no } \mathsf{atomic}}{R, G \vdash \{P * F\}\ C\ \{Q * F\}}$$

$$\frac{Q \equiv (P * X \mapsto Y) \qquad x \notin fv(P)}{R, G \vdash \{\boxed{\mathrm{Q}} \wedge e = X\}\ x := [e]\ \{\boxed{\mathrm{Q}} * x = Y\}}$$

$$\frac{R, G \vdash \{P\}\ C_1\ \{R\} \qquad R, G \vdash \{R\}\ C_2\ \{Q\}}{R, G \vdash \{P\}\ C_1; C_2\ \{Q\}}$$

RGSep Proof Rules

$$\frac{R, G \vdash \{P\}\ C\ \{Q\} \qquad F \text{ stable for } (R \cup G) \text{ or } C \text{ has no } \mathsf{atomic}}{R, G \vdash \{P * F\}\ C\ \{Q * F\}}$$

$$\frac{Q \equiv (P * X \mapsto Y) \qquad x \notin fv(P)}{R, G \vdash \{\boxed{\text{Q}} \wedge e = X\}\ x := [e]\ \{\boxed{\text{Q}} * x = Y\}}$$

$$\frac{R, G \vdash \{P\}\ C_1\ \{R\} \qquad R, G \vdash \{R\}\ C_2\ \{Q\}}{R, G \vdash \{P\}\ C_1; C_2\ \{Q\}}$$

$$\frac{\vdash \{P_1 * P_2\}\ C\ \{Q_1 * Q_2\} \qquad \boxed{\text{Q}} \text{ stable for } R \qquad \bar{y} \cap fv(P_2) = \varnothing \qquad P \Rightarrow P_1 * F \qquad Q_1 * F \Rightarrow Q \qquad (P_1 \rightsquigarrow Q_1) \subseteq G}{\vdash \{\boxed{\exists \bar{y}.\ P} * P_2\}\ \mathsf{atomic}\ C\ \{\exists \bar{y}.\ \boxed{\text{Q}} * Q_2\}}$$

RGSep Proof Rules

$$\frac{R, G \vdash \{P\}\ C\ \{Q\} \qquad F \text{ stable for } (R \cup G) \text{ or } C \text{ has no } \mathsf{atomic}}{R, G \vdash \{P * F\}\ C\ \{Q * F\}}$$

$$\frac{Q \equiv (P * X \mapsto Y) \qquad x \notin fv(P)}{R, G \vdash \{\boxed{\mathrm{Q}} \wedge e = X\}\ x := [e]\ \{\boxed{\mathrm{Q}} * x = Y\}}$$

$$\frac{R, G \vdash \{P\}\ C_1\ \{R\} \qquad R, G \vdash \{R\}\ C_2\ \{Q\}}{R, G \vdash \{P\}\ C_1; C_2\ \{Q\}}$$

$$\frac{\begin{array}{c}\vdash \{P_1 * P_2\}\ C\ \{Q_1 * Q_2\} \qquad \boxed{\mathrm{Q}} \text{ stable for } R \\ \bar{y} \cap fv(P_2) = \varnothing \qquad P \Rightarrow P_1 * F \qquad Q_1 * F \Rightarrow Q \qquad (P_1 \rightsquigarrow Q_1) \subseteq G\end{array}}{\vdash \{\boxed{\exists \bar{y}.\ P} * P_2\}\ \mathsf{atomic}\ C\ \{\exists \bar{y}.\ \boxed{\mathrm{Q}} * Q_2\}}$$

$$\frac{\begin{array}{ll}R \cup G_2, G_1 \vdash \{P_1\}\ C_1\ \{Q_1\} & P_1 \text{ stable for } R \cup G_2 \\ R \cup G_1, G_2 \vdash \{P_2\}\ C_2\ \{Q_2\} & P_2 \text{ stable for } R \cup G_1\end{array}}{R, G_1 \cup G_2 \vdash \{P_1 * P_2\}\ C_1 \| C_2\ \{Q_1 * Q_2\}}$$

RGSep Proof Rules

$$\frac{}{x \mapsto y \rightsquigarrow x \mapsto y \subseteq G}\ \text{G-EXACT}$$

$$\frac{P \rightsquigarrow Q \in G}{P \rightsquigarrow Q \subseteq G}\ \text{G-AX}$$

$$\frac{P_1 \rightsquigarrow S * Q_1 \subseteq G \quad P_2 * S \rightsquigarrow Q_2 \subseteq G}{P_1 * P_2 \rightsquigarrow Q_1 * Q_2 \subseteq G}\ \text{G-SEQ}$$

$$\frac{P \rightsquigarrow Q \subseteq G}{P[e/x] \rightsquigarrow Q[e/x] \subseteq G}\ \text{G-SUB}$$

$$\frac{\models_{\text{SL}} P' \Rightarrow P \quad P \rightsquigarrow Q \subseteq G \quad \models_{\text{SL}} Q' \Rightarrow Q}{P' \rightsquigarrow Q' \subseteq G}\ \text{G-CONS}$$

$$\frac{(P * F) \rightsquigarrow (Q * F) \subseteq G}{P \rightsquigarrow Q \subseteq G}\ \text{G-COFRM}$$

RGSep Stability

Definition 1 (Stability). $S;\mathcal{R} \Longrightarrow S$ *iff for all* s, s' *and* i *such that* $s, i \vDash_{\mathrm{SL}} S$ *and* $(s, s') \in \mathcal{R}$, *then* $s', i \vDash_{\mathrm{SL}} S$

Lemma 1. $S; [\![P \rightsquigarrow Q]\!] \Longrightarrow S$ *iff* $\vDash_{\mathrm{SL}} (P -\!\!\circledast S) * Q \Longrightarrow S$

Rely's are Environment Actions

$$\begin{aligned}
l, s, i \vDash_R P &\iff l, i \vDash_{\mathrm{SL}} P \\
l, s, i \vDash_R \boxed{P} &\iff l = \emptyset \wedge s, i \vDash_{\mathrm{SL}} \mathrm{wssa}_{[\![R]\!]}(P) \\
l, s, i \vDash_R p_1 * p_2 &\iff \exists l_1, l_2.\ (l = l_1 \uplus l_2) \wedge (l_1, s, i \vDash_R p_1) \wedge (l_2, s, i \vDash_R p_2) \\
l, s, i \vDash_R p_1 \wedge p_2 &\iff (l, s, i \vDash_R p_1) \wedge (l, s, i \vDash_R p_2) \\
\ldots
\end{aligned}$$

Abstract Domain

$$
\begin{aligned}
A, B &::= E_1{=}E_2 \mid E_1{\neq}E_2 \mid E \mapsto \rho \mid \mathsf{lseg}(E_1, E_2) \mid \mathsf{junk} \\
P, Q, R, S &::= A \mid P \vee Q \mid P * Q \mid P \mathrel{-\!\!\circledast} Q \mid P\!\restriction_{E_1,\ldots,E_n} \\
p, q &::= p \vee q \mid P * \boxed{Q}
\end{aligned}
$$

Abstract Domain

$$
\begin{aligned}
A, B &::= E_1 = E_2 \mid E_1 \neq E_2 \mid E \mapsto \rho \mid \mathsf{lseg}(E_1, E_2) \mid \mathsf{junk} \\
P, Q, R, S &::= A \mid P \vee Q \mid P * Q \mid P \mathrel{-\!\circledast} Q \mid P\!\downharpoonleft_{E_1,\ldots,E_n} \\
p, q &::= p \vee q \mid P * \boxed{Q}
\end{aligned}
$$

Everything as before except

$$P \downharpoonleft_{(E_1,\ldots,E_n)} \iff P \wedge \neg((E_1 \mapsto \_) * true) \wedge \cdots \wedge \neg((E_n \mapsto \_) * true)$$

locations $(E_1, \ldots, E_n)$ are not allocated

Abstract Domain

$$\begin{aligned} A, B &::= E_1{=}E_2 \mid E_1{\neq}E_2 \mid E \mapsto \rho \mid \mathsf{lseg}(E_1, E_2) \mid \mathsf{junk} \\ P, Q, R, S &::= A \mid P \vee Q \mid P * Q \mid P \mathrel{-\!\!\circledast} Q \mid P{\downharpoonleft}_{E_1,\ldots,E_n} \\ p, q &::= p \vee q \mid P * \boxed{Q} \end{aligned}$$

Everything as before except

$$P \downharpoonleft_{(E_1,\ldots,E_n)} \iff P \wedge \neg((E_1 \mapsto \_) * true) \wedge \cdots \wedge \neg((E_n \mapsto \_) * true)$$

locations $(E_1, \ldots, E_n)$ are not allocated

We extend `lseg` to account for `D`

$$\mathsf{lsegi}(E_1 1, E_2, D) = (E_1 = E_2) \vee \exists x.\ (E_1 \mapsto F) \downharpoonleft_D * \mathsf{lsegi}(F, E_2, D)$$

Elimination Rules

$$\begin{aligned}
(F \mapsto \rho)\lfloor_D &\iff F \neq D * (F \mapsto \rho) \\
\mathsf{lsegi}_{tl,\rho}(E, F, D')\lfloor_D &\iff \mathsf{lsegi}_{tl,\rho}(E, F, D \cup D') \\
(P * Q)\lfloor_D &\iff P\lfloor_D * Q\lfloor_D \\
(P \vee Q)\lfloor_D &\iff P\lfloor_D \vee Q\lfloor_D
\end{aligned}$$

Elimination Rules

$$
\begin{aligned}
(F \mapsto \rho)\downharpoonleft_D &\iff F \neq D * (F \mapsto \rho) \\
\mathsf{lsegi}_{tl,\rho}(E, F, D')\downharpoonleft_D &\iff \mathsf{lsegi}_{tl,\rho}(E, F, D \cup D') \\
(P * Q)\downharpoonleft_D &\iff P\downharpoonleft_D * Q\downharpoonleft_D \\
(P \vee Q)\downharpoonleft_D &\iff P\downharpoonleft_D \vee Q\downharpoonleft_D
\end{aligned}
$$

$$
\begin{aligned}
(E_1 \mapsto \rho_1) \mathrel{-\!\!\circledast} (E_2 \mapsto \rho_2) &\iff E_1 = E_2 * \rho_1 = \rho_2 \\
(E_1 \mapsto \mathtt{tl}{=}E_2, \rho) \mathrel{-\!\!\circledast} \mathsf{lsegi}_{tl,\rho'}(E, E', D) &\iff \\
E_1{\neq}0 * E_1{\neq}D * \rho{=}\rho' * \mathsf{lsegi}_{tl,\rho'}(E, E_1, D)\downharpoonleft_{E'} &* \mathsf{lsegi}_{tl,\rho'}(E_2, E', D)\downharpoonleft_{E_1} \\
(E \mapsto \rho) \mathrel{-\!\!\circledast} (P * Q) &\iff P\downharpoonleft_E * (E \mapsto \rho \mathrel{-\!\!\circledast} Q) \\
&\qquad \vee \ (E \mapsto \rho \mathrel{-\!\!\circledast} P) * Q\downharpoonleft_E \\
(E \mapsto \rho) \mathrel{-\!\!\circledast} (P \vee Q) &\iff (E \mapsto \rho \mathrel{-\!\!\circledast} P) \ \vee \ (E \mapsto \rho \mathrel{-\!\!\circledast} Q) \\
(P * Q) \mathrel{-\!\!\circledast} R &\iff P \mathrel{-\!\!\circledast} (Q \mathrel{-\!\!\circledast} R) \\
(P \vee Q) \mathrel{-\!\!\circledast} R &\iff (P \mathrel{-\!\!\circledast} R) \vee (Q \mathrel{-\!\!\circledast} R)
\end{aligned}
$$

Dealing with Interference

Action definitions

$$\texttt{action Lock}(\texttt{x}) \quad x \mapsto lk = 0 \leadsto x \mapsto lk = \textsf{TID}$$

$$\texttt{action Unlock}(\texttt{x}) \quad x \mapsto lk = \textsf{TID} \leadsto x \mapsto lk = 0$$

Dealing with Interference

Action definitions

$$\texttt{action Lock}(\texttt{x}) \quad x \mapsto lk = 0 \rightsquigarrow x \mapsto lk = \text{TID}$$

$$\texttt{action Unlock}(\texttt{x}) \quad x \mapsto lk = \text{TID} \rightsquigarrow x \mapsto lk = 0$$

Stable Assertions

$$S_0 = S \qquad\qquad S_{n+1} = S_n \vee (P \mathrel{-\!\!\circledast} S_n) * Q$$

Dealing with Interference

Action definitions

$$\texttt{action Lock}(\texttt{x}) \quad x \mapsto lk = 0 \leadsto x \mapsto lk = \text{TID}$$

$$\texttt{action Unlock}(\texttt{x}) \quad x \mapsto lk = \text{TID} \leadsto x \mapsto lk = 0$$

Stable Assertions

$$S_0 = S \qquad\qquad S_{n+1} = S_n \vee (P \mathbin{-\!\!\circledast} S_n) * Q$$

Just as before, this might add an unbounded number of primed variables

Dealing with Interference

Action definitions

$$\texttt{action Lock}(\texttt{x}) \quad x \mapsto lk = 0 \leadsto x \mapsto lk = \mathsf{TID}$$

$$\texttt{action Unlock}(\texttt{x}) \quad x \mapsto lk = \mathsf{TID} \leadsto x \mapsto lk = 0$$

Stable Assertions

$$S_0 = S \qquad\qquad S_{n+1} = S_n \vee (P -\!\circledast S_n) * Q$$

Just as before, this might add an unbounded number of primed variables

Stable Abstractions

$$S_0 = \alpha(S) \qquad S_{n+1} = S_n \vee \alpha((P -\!\circledast S_n) * Q)$$

Abstraction

Use a technique similar to the one we saw before

Stabilize

$$x \mapsto lk = 0 \land y \mapsto lk = \text{TID}$$

Rely actions

$$\begin{aligned}
\texttt{Lock \_tid} \quad &= \quad \texttt{\_tid} \neq 0 \land \texttt{\_tid} \neq \text{TID} \land x \mapsto lk = 0 \leadsto \\
&\qquad\qquad \texttt{\_tid} \neq 0 \land \texttt{\_tid} \neq \text{TID} \land x \mapsto lk = \texttt{\_tid} \\
\texttt{Unlock \_tid} \quad &= \quad \texttt{\_tid} \neq 0 \land \texttt{\_tid} \neq \text{TID} \land x \mapsto lk = \text{TID} \leadsto \\
&\qquad\qquad \texttt{\_tid} \neq 0 \land \texttt{\_tid} \neq \text{TID} \land x \mapsto lk = 0
\end{aligned}$$

Abstraction: example

$S_0 \iff$

Abstraction: example

$$S_0 \iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID})$$

Abstraction: example

$$S_0 \iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) = x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}$$

Abstraction: example

$$S_0 \iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) = x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}$$

action **`lock`**

Abstraction: example

$$S_0 \iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) = x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}$$

action `lock`

$$S_1 \iff S_0 \vee \alpha(\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID})$$

Abstraction: example

$$S_0 \iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) = x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}$$

$$\text{action } \mathtt{lock}$$

$$\begin{aligned} S_1 &\iff S_0 \vee \alpha(\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \\ &\iff S_0 \vee (\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \end{aligned}$$

Abstraction: example

$$
\begin{aligned}
S_0 &\iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) = x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID} \\
&\qquad\qquad\qquad\qquad\qquad \text{action } \mathtt{lock} \\
S_1 &\iff S_0 \vee \alpha(\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \\
&\iff S_0 \vee (\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \\
&\iff \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}
\end{aligned}
$$

Abstraction: example

$$S_0 \iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) = x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}$$

$$\text{action } \mathtt{lock}$$

$$\begin{aligned} S_1 &\iff S_0 \vee \alpha(\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \\ &\iff S_0 \vee (\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \\ &\iff \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID} \end{aligned}$$

$$\text{action } \mathtt{lock}$$

Abstraction: example

$$S_0 \iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) = x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}$$

action **lock**

$$S_1 \iff S_0 \vee \alpha(\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID})$$
$$\iff S_0 \vee (\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID})$$
$$\iff \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}$$

action **lock**

$$S_2 \iff S_1 \vee \alpha(\_tid' \neq 0 * \_tid' \neq \mathtt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \mathtt{TID})$$

Abstraction: example

$$S_0 \iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) = x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}$$

action $\mathtt{lock}$

$$\begin{aligned}
S_1 &\iff S_0 \lor \alpha(\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \\
&\iff S_0 \lor (\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \\
&\iff \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}
\end{aligned}$$

action $\mathtt{lock}$

$$\begin{aligned}
S_2 &\iff S_1 \lor \alpha(\_tid' \neq 0 * \_tid' \neq \mathtt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \mathtt{TID}) \\
&\iff S_1 \lor (\_tid' \neq 0 * \_tid' \neq \mathtt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \mathtt{TID})
\end{aligned}$$

Abstraction: example

$$
\begin{aligned}
S_0 &\iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) = x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID} \\
&\qquad\qquad\qquad\qquad \text{action } \mathtt{lock} \\
S_1 &\iff S_0 \vee \alpha(\mathit{\_tid} \neq 0 * \mathit{\_tid} \neq \mathtt{TID} * x \mapsto lk = \mathit{\_tid} * y \mapsto lk = \mathtt{TID}) \\
&\iff S_0 \vee (\mathit{\_tid} \neq 0 * \mathit{\_tid} \neq \mathtt{TID} * x \mapsto lk = \mathit{\_tid} * y \mapsto lk = \mathtt{TID}) \\
&\iff \mathit{\_tid} \neq \mathtt{TID} * x \mapsto lk = \mathit{\_tid} * y \mapsto lk = \mathtt{TID} \\
&\qquad\qquad\qquad\qquad \text{action } \mathtt{lock} \\
S_2 &\iff S_1 \vee \alpha(\mathit{\_tid}' \neq 0 * \mathit{\_tid}' \neq \mathtt{TID} * x \mapsto lk = \mathit{\_tid}' * y \mapsto lk = \mathtt{TID}) \\
&\iff S_1 \vee (\mathit{\_tid}' \neq 0 * \mathit{\_tid}' \neq \mathtt{TID} * x \mapsto lk = \mathit{\_tid}' * y \mapsto lk = \mathtt{TID}) \\
&\iff (\mathit{\_tid} \neq \mathtt{TID} * x \mapsto lk = \mathit{\_tid} * y \mapsto lk = \mathtt{TID})
\end{aligned}
$$

Abstraction: example

$$
\begin{aligned}
S_0 &\iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) = x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID} \\
&\qquad\qquad\text{action } \mathtt{lock} \\
S_1 &\iff S_0 \vee \alpha(\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \\
&\iff S_0 \vee (\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \\
&\iff \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID} \\
&\qquad\qquad\text{action } \mathtt{lock} \\
S_2 &\iff S_1 \vee \alpha(\_tid' \neq 0 * \_tid' \neq \mathtt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \mathtt{TID}) \\
&\iff S_1 \vee (\_tid' \neq 0 * \_tid' \neq \mathtt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \mathtt{TID}) \\
&\iff (\_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \iff S_1
\end{aligned}
$$

Abstraction: example

$$S_0 \iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) = x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}$$

action **lock**

$$S_1 \iff S_0 \vee \alpha(\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID})$$

$$\iff S_0 \vee (\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID})$$

$$\iff \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}$$

action **lock**

$$S_2 \iff S_1 \vee \alpha(\_tid' \neq 0 * \_tid' \neq \mathtt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \mathtt{TID})$$

$$\iff S_1 \vee (\_tid' \neq 0 * \_tid' \neq \mathtt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \mathtt{TID})$$

$$\iff (\_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \iff S_1$$

action **unlock**

Abstraction: example

$$
\begin{aligned}
S_0 &\iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) = x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID} \\
&\qquad\qquad \text{action } \mathtt{lock} \\
S_1 &\iff S_0 \vee \alpha(\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \\
&\iff S_0 \vee (\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \\
&\iff \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID} \\
&\qquad\qquad \text{action } \mathtt{lock} \\
S_2 &\iff S_1 \vee \alpha(\_tid' \neq 0 * \_tid' \neq \mathtt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \mathtt{TID}) \\
&\iff S_1 \vee (\_tid' \neq 0 * \_tid' \neq \mathtt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \mathtt{TID}) \\
&\iff (\_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \iff S_1 \\
&\qquad\qquad \text{action } \mathtt{unlock} \\
S_3 &\iff S_2 \vee \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID})
\end{aligned}
$$

Abstraction: example

$$\begin{aligned}
S_0 &\iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) = x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID} \\
&\qquad\qquad \text{action } \mathtt{lock} \\
S_1 &\iff S_0 \vee \alpha(\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \\
&\iff S_0 \vee (\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \\
&\iff \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID} \\
&\qquad\qquad \text{action } \mathtt{lock} \\
S_2 &\iff S_1 \vee \alpha(\_tid' \neq 0 * \_tid' \neq \mathtt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \mathtt{TID}) \\
&\iff S_1 \vee (\_tid' \neq 0 * \_tid' \neq \mathtt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \mathtt{TID}) \\
&\iff (\_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \iff S_1 \\
&\qquad\qquad \text{action } \mathtt{unlock} \\
S_3 &\iff S_2 \vee \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) \\
&\iff S_2 \vee (x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID})
\end{aligned}$$

Abstraction: example

$$S_0 \iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) = x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}$$

action **lock**

$$\begin{aligned} S_1 &\iff S_0 \vee \alpha(\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \\ &\iff S_0 \vee (\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \\ &\iff \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID} \end{aligned}$$

action **lock**

$$\begin{aligned} S_2 &\iff S_1 \vee \alpha(\_tid' \neq 0 * \_tid' \neq \mathtt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \mathtt{TID}) \\ &\iff S_1 \vee (\_tid' \neq 0 * \_tid' \neq \mathtt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \mathtt{TID}) \\ &\iff (\_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \iff S_1 \end{aligned}$$

action **unlock**

$$\begin{aligned} S_3 &\iff S_2 \vee \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) \\ &\iff S_2 \vee (x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) \\ &\iff S_2 \end{aligned}$$

Abstraction: example

$$S_0 \iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}) = x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}$$

action **lock**

$$S_1 \iff S_0 \vee \alpha(\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID})$$
$$\iff S_0 \vee (\_tid \neq 0 * \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID})$$
$$\iff \_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}$$

action **lock**

$$S_2 \iff S_1 \vee \alpha(\_tid' \neq 0 * \_tid' \neq \mathtt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \mathtt{TID})$$
$$\iff S_1 \vee (\_tid' \neq 0 * \_tid' \neq \mathtt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \mathtt{TID})$$
$$\iff (\_tid \neq \mathtt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \mathtt{TID}) \iff S_1$$

action **unlock**

$$S_3 \iff S_2 \vee \alpha(x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID})$$
$$\iff S_2 \vee (x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID})$$
$$\iff S_2$$

Stable!

Symbolic Execution

- Assumptions
  - We restrict all global state modifying commands to happen in atomic blocks

Only if B is true

Producing action Act(x)

$$\texttt{atomic}\ (B)\ C\ \texttt{as}\ \texttt{Act}(x)$$

Execute C atomically

  - All *non-atomic* commands are easy
  - Lets see about atomic

Symbolic Execution

$$\frac{\begin{array}{c}\{X * S\}\ \texttt{assume}(B)\ \{X * P * F\} \\ \{X * P\}\ C\ \{X'\} \qquad X' \vdash Q * Y \qquad stab(Q * F) = R\end{array}}{\{X * \boxed{\mathrm{S}}\}\ \texttt{atomic}\ (B)\ \texttt{as}\ \texttt{Act}(x)\ \{Y * \boxed{\mathrm{R}}\}}$$

Symbolic Execution

$$\frac{\{X * S\}\ \texttt{assume}(B)\ \{X * P * F\} \qquad \{X * P\}\ C\ \{X'\} \qquad X' \vdash Q * Y \qquad stab(Q * F) = R}{\{X * \boxed{\mathrm{S}}\}\ \texttt{atomic}\ (B)\ \texttt{as Act}(x)\ \{Y * \boxed{\mathrm{R}}\}}$$

🟨 Call theorem prover to infer frame F

Symbolic Execution

$$\frac{\{X * P\}\ C\ \{X'\} \qquad X' \vdash Q * Y \qquad stab(Q * F) = R \qquad \{X * S\}\ \mathtt{assume}(B)\ \{X * P * F\}}{\{X * \boxed{\mathrm{S}}\}\ \mathtt{atomic}\ (B)\ \mathtt{as}\ \mathtt{Act}(x)\ \{Y * \boxed{\mathrm{R}}\}}$$

🟨 Call theorem prover to infer frame F

🟩 Symbolically execute C ignoring F

Symbolic Execution

$$\frac{\{X * S\}\ \texttt{assume}(B)\ \{X * P * F\} \qquad \{X * P\}\ C\ \{X'\} \qquad X' \vdash Q * Y \qquad stab(Q * F) = R}{\{X * \boxed{\mathrm{S}}\}\ \texttt{atomic}\ (B)\ \texttt{as Act}(x)\ \{Y * \boxed{\mathrm{R}}\}}$$

🟨 Call theorem prover to infer frame F

🟩 Symbolically execute C ignoring F

🟦 Infer frame Y (local post) and check that Q is satisfied

Symbolic Execution

$$\frac{\{X * S\}\ \texttt{assume}(B)\ \{X * P * F\} \quad \{X * P\}\ C\ \{X'\} \qquad X' \vdash Q * Y \qquad stab(Q * F) = R}{\{X * \boxed{\mathrm{S}}\}\ \texttt{atomic}\ (B)\ \texttt{as}\ \texttt{Act}(x)\ \{Y * \boxed{\mathrm{R}}\}}$$

🟨 Call theorem prover to infer frame F

🟩 Symbolically execute C ignoring F

🟦 Infer frame Y (local post) and check that Q is satisfied

🟥 Stabilize the frame F and the shared post-condition Q

Shape-Value Abstraction for Verifying Linearizability

Verifying Linearizability

- Simulation based approach:
  1. For each method locate the linearization point in the code (conceptually the atomic execution)
  2. Embed an "*abstract atomic operation*" at an atomic command
  3. Define an *abstraction map* relating concrete and abstract states
  4. Prove that the abstraction map is invariant and the abstract and concrete operations return the same values

Modular fine-grained concurrency verification
Vafeiadis 2008

Verifying Linearizability

- Linearization points are provided by the programmer
- Shape analysis recalls values to track the simulation between the concrete and abstract data structure
- Check that the simulation is preserved at the linearization points during symbolic execution

Treiber Stack (revisited)

```
struct stack {
  struct node *Top;
};

struct stack *S;

value_t pop() { struct node *t, *x;
  do {
    t = S->Top;                       // @2
    if (t == NULL)
      return EMPTY;
    x = t->next;
  } while (¬CAS(&S->Top,t,x));      // @3
  return t->data;
}
```

```
void init() {
   S = alloc();
   S->Top = NULL;
/* ABS->val = ϵ; */
}

void push(value_t v) { struct node *t, *x;
  x = alloc();
  x->data = v;
  do {
    t = S->Top;
    x->next = t;
  } while (¬CAS(&S->Top,t,x));    // @1
}
```

Treiber Stack (revisited)

```
struct stack {
  struct node *Top;
};

struct stack *S;

value_t pop() { struct node *t, *x;
  do {
    t = S->Top;                    // @2
    if (t == NULL)
      return EMPTY;
    x = t->next;
  } while (¬CAS(&S->Top,t,x));     // @3
  return t->data;
}
```

```
void init() {
   S = alloc();
   S->Top = NULL;
/* ABS->val = ε; */
}

void push(value_t v) { struct node *t, *x;
  x = alloc();
  x->data = v;
  do {
    t = S->Top;
    x->next = t;
  } while (¬CAS(&S->Top,t,x));     // @1
}
```

```
action APush() [S↦Top:n                           * ABS↦val:A]
               [S↦Top:y * y↦data:e,next:n * ABS↦val:⟨e⟩·A]
action APop()  [S↦Top:y * y↦data:e,next:n * ABS↦val:⟨e⟩·A]
               [S↦Top:n * y↦data:e,next:n * ABS↦val:A]
```

Treiber Stack (revisited)

```
struct stack {
  struct node *Top;
};

struct stack *S;

value_t pop() { struct node *t, *x;
  do {
    t = S->Top;                         // @2
    if (t == NULL)
      return EMPTY;
    x = t->next;
  } while (¬CAS(&S->Top,t,x));         // @3
  return t->data;
}
```

```
void init() {
   S = alloc();
   S->Top = NULL;
/* ABS->val = ϵ; */
}

void push(value_t v) { struct node *t, *x;
  x = alloc();
  x->data = v;
  do {
    t = S->Top;
    x->next = t;
  } while (¬CAS(&S->Top,t,x));     // @1
}
```

```
action APush() [S↦Top:n                             * ABS↦val:A]
               [S↦Top:y * y↦data:e,next:n * ABS↦val:⟨e⟩·A]
action APop()  [S↦Top:y * y↦data:e,next:n * ABS↦val:⟨e⟩·A]
               [S↦Top:n * y↦data:e,next:n * ABS↦val:A]
```

Invariant: $J \overset{\text{def}}{=} \exists n v.\ \mathsf{S}{\mapsto}\mathtt{Top}{:}n * \mathsf{lseg}(n, \mathtt{NULL}, v) * \mathtt{ABS}{\mapsto}\mathtt{val}{:}v$

Treiber Stack (revisited)

```
struct stack {
  struct node *Top;
};

struct stack *S;

value_t pop() { struct node *t, *x;
  do {
    t = S->Top;                  // @2
    if (t == NULL)
      return EMPTY;
    x = t->next;
  } while (¬CAS(&S->Top,t,x));   // @3
  return t->data;
}
```

```
void init() {
   S = alloc();
   S->Top = NULL;
/* ABS->val = ϵ; */
}

void push(value_t v) { struct node *t, *x;
  x = alloc();
  x->data = v;
  do {
    t = S->Top;
    x->next = t;
  } while (¬CAS(&S->Top,t,x));    // @1
}
```

```
action APush() [S↦Top:n                        * ABS↦val:A]
               [S↦Top:y * y↦data:e,next:n * ABS↦val:⟨e⟩·A]
action APop()  [S↦Top:y * y↦data:e,next:n * ABS↦val:⟨e⟩·A]
               [S↦Top:n * y↦data:e,next:n * ABS↦val:A]
```

Invariant: $J \overset{\text{def}}{=} \exists nv.\ \mathsf{S}{\mapsto}\mathtt{Top}{:}n * \mathsf{lseg}(n, \mathtt{NULL}, v) * \mathtt{ABS}{\mapsto}\mathtt{val}{:}v$

CAVE

Shape-Value Domain

Abstraction

$$\alpha_{\mathsf{total}} = \alpha_{\mathsf{value}} \circ \alpha_{\mathsf{shape}}$$

Concretization

$$\gamma_{\mathsf{total}} = \gamma_{\mathsf{shape}} \circ \gamma_{\mathsf{value}}$$

Shape-Value Domain

Abstraction

$$\alpha_{\mathsf{total}} = \alpha_{\mathsf{value}} \circ \alpha_{\mathsf{shape}}$$

Concretization

$$\gamma_{\mathsf{total}} = \gamma_{\mathsf{shape}} \circ \gamma_{\mathsf{value}}$$

Old Domain

$$\mathsf{lseg}(x, y) \stackrel{\text{def}}{=} (x = y \land \mathsf{emp}) \lor (\exists bz.\ \mathsf{Node}(x, z, b) * \mathsf{lseg}(z, y))$$

$$\mathsf{Node}(x, y, v) \stackrel{\text{def}}{=} x \mapsto \{\texttt{.next} = y, \texttt{.data} = v\}$$

Shape-Value Domain

Abstraction

$$\alpha_{\mathsf{total}} = \alpha_{\mathsf{value}} \circ \alpha_{\mathsf{shape}}$$

Concretization

$$\gamma_{\mathsf{total}} = \gamma_{\mathsf{shape}} \circ \gamma_{\mathsf{value}}$$

Old Domain

$$\mathsf{lseg}(x, y) \stackrel{\text{def}}{=} (x = y \wedge \mathsf{emp}) \vee (\exists bz.\ \mathsf{Node}(x, z, b) * \mathsf{lseg}(z, y))$$

$$\mathsf{Node}(x, y, v) \stackrel{\text{def}}{=} x \mapsto \{\texttt{.next} = y, \texttt{.data} = v\}$$

New Domain

$$\begin{aligned}\mathsf{lseg}_{\mathsf{new}}(x, y, a) \stackrel{\text{def}}{=}\ & (x = y \wedge a = \epsilon \wedge \mathsf{emp}) \\ & \vee \exists bcz.\ a = \langle b \rangle \cdot c * \mathsf{Node}(x, z, b) * \mathsf{lseg}_{\mathsf{new}}(z, y, c)\end{aligned}$$

Shape Abstraction

c.f. the old abstraction rules

$$\begin{array}{rcl}
\mathsf{Node}(y,z,b) & \Longrightarrow & \mathsf{junk} \\
\mathsf{Node}(x,y,a) * \mathsf{Node}(y,z,b) & \Longrightarrow & \mathsf{lseg}_{\mathsf{new}}(x,z,\langle a\rangle\cdot\langle b\rangle) \\
\mathsf{lseg}_{\mathsf{new}}(x,y,a) * \mathsf{Node}(y,z,b) & \Longrightarrow & \mathsf{lseg}_{\mathsf{new}}(x,z,a\cdot\langle b\rangle) \\
\mathsf{lseg}_{\mathsf{new}}(y,z,b) & \Longrightarrow & \mathsf{junk} \\
\mathsf{Node}(x,y,a) * \mathsf{lseg}_{\mathsf{new}}(y,z,b) & \Longrightarrow & \mathsf{lseg}_{\mathsf{new}}(x,z,\langle a\rangle\cdot b) \\
\mathsf{lseg}_{\mathsf{new}}(x,y,a) * \mathsf{lseg}_{\mathsf{new}}(y,z,b) & \Longrightarrow & \mathsf{lseg}_{\mathsf{new}}(x,z,a\cdot b)
\end{array}$$

Value Abstraction

How do we abstract this shape to keep track of the value equalities?

$$\mathsf{lseg}(k, 0, b{\cdot}c{\cdot}d{\cdot}e) * \mathsf{lseg}(l, 0, a{\cdot}b) * \mathsf{lseg}(m, 0, a{\cdot}b) * \mathsf{lseg}(n, 0, e)$$

Value Abstraction

How do we abstract this shape to keep track of the value equalities?

$$\mathsf{lseg}(k, 0, b{\cdot}c{\cdot}d{\cdot}e) * \mathsf{lseg}(l, 0, a{\cdot}b) * \mathsf{lseg}(m, 0, a{\cdot}b) * \mathsf{lseg}(n, 0, e)$$

The most precise answer is:

$$\exists tuvw.\ \mathsf{lseg}(k, 0, u{\cdot}v{\cdot}w) * \mathsf{lseg}(l, 0, t{\cdot}u) * \mathsf{lseg}(m, 0, t{\cdot}u) * \mathsf{lseg}(n, 0, w)$$

Value Abstraction

How do we abstract this shape to keep track of the value equalities?

$$\mathsf{lseg}(k, 0, b{\cdot}c{\cdot}d{\cdot}e) * \mathsf{lseg}(l, 0, a{\cdot}b) * \mathsf{lseg}(m, 0, a{\cdot}b) * \mathsf{lseg}(n, 0, e)$$

The most precise answer is:

$$\exists tuvw.\ \mathsf{lseg}(k, 0, u{\cdot}v{\cdot}w) * \mathsf{lseg}(l, 0, t{\cdot}u) * \mathsf{lseg}(m, 0, t{\cdot}u) * \mathsf{lseg}(n, 0, w)$$

Computing this abstraction for sequences:

- Identify values S that can be safely existentially quantified

```
$S := T \setminus \{\epsilon\};$
$\mathbf{while}\ \left(\begin{array}{l} \exists x \in S, y \in S.\ \exists z, x_1, x_2, y_1, y_2. \\ \qquad x \neq y \wedge z \neq \epsilon \wedge x = x_1{\cdot}z{\cdot}x_2 \wedge y = y_1{\cdot}z{\cdot}y_2 \end{array}\right)\ \mathbf{do}$
  $S := (S \setminus \{x, y\}) \cup (\{x_1, x_2, y_1, y_2, z\} \setminus \{\epsilon\})$
```

Shape-Value Abstraction

- We can now run the symbolic execution algorithm that we presented before
- Check that the *values* in the symbolic state correspond to the *values* in the specification state

RGSep Action Inference

Abstractions

As before, abstraction is necessary to guarantee termination of symbolic state transformations like

$$P \leftarrow P \vee \alpha(\mathit{transform}(P))$$

$$\textsc{Abstract}(P) \text{ over-approximates } P \ (\llbracket P \rrbracket_{\mathcal{I}} \subseteq \llbracket \textsc{Abstract}(P) \rrbracket_{\mathcal{I}})$$

Actions under a context R

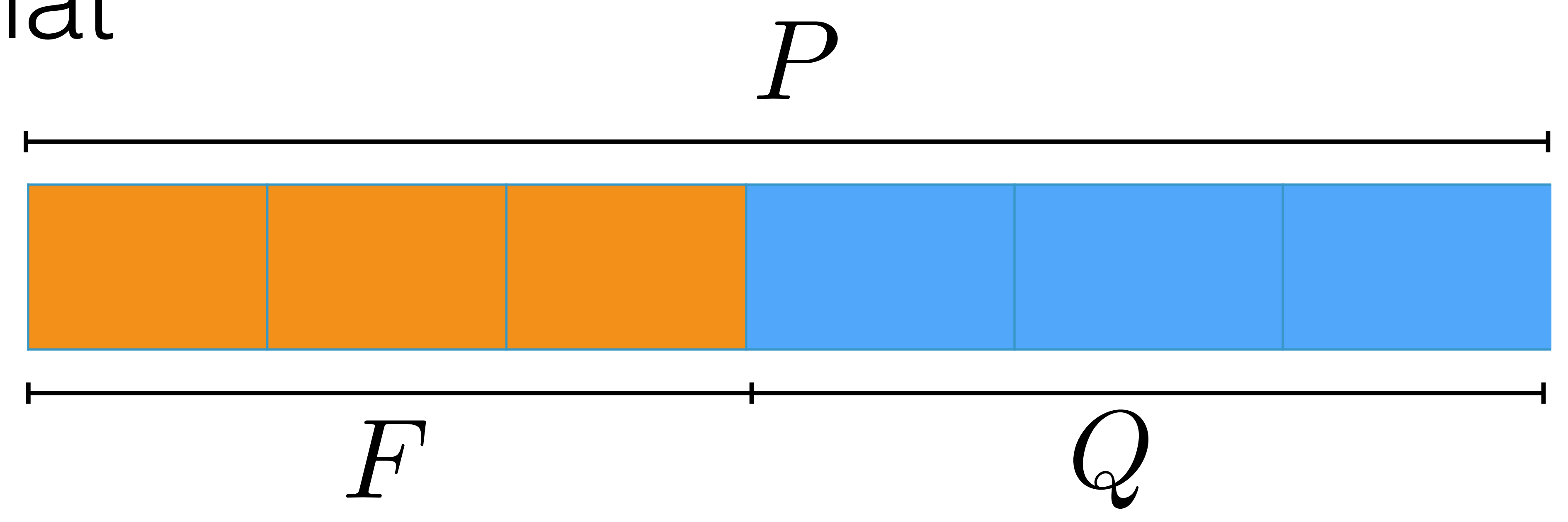
$$\begin{aligned} \mathcal{A}\llbracket R \mid P \rightsquigarrow Q \rrbracket &\overset{\text{def}}{=} \mathcal{A}\llbracket P \rightsquigarrow Q \rrbracket \cap \mathcal{A}\llbracket P * R \rightsquigarrow Q * R \rrbracket \\ &= \{(s \uplus s_0, s' \uplus s_0) \mid \exists \mathcal{I}.\ \ s \in \llbracket P \rrbracket_{\mathcal{I}} \wedge s' \in \llbracket Q \rrbracket_{\mathcal{I}} \wedge s_0 \in \llbracket R * \mathsf{true} \rrbracket_{\mathcal{I}}\} \end{aligned}$$

R allows to delimit when the action can be executed

May-Subtraction

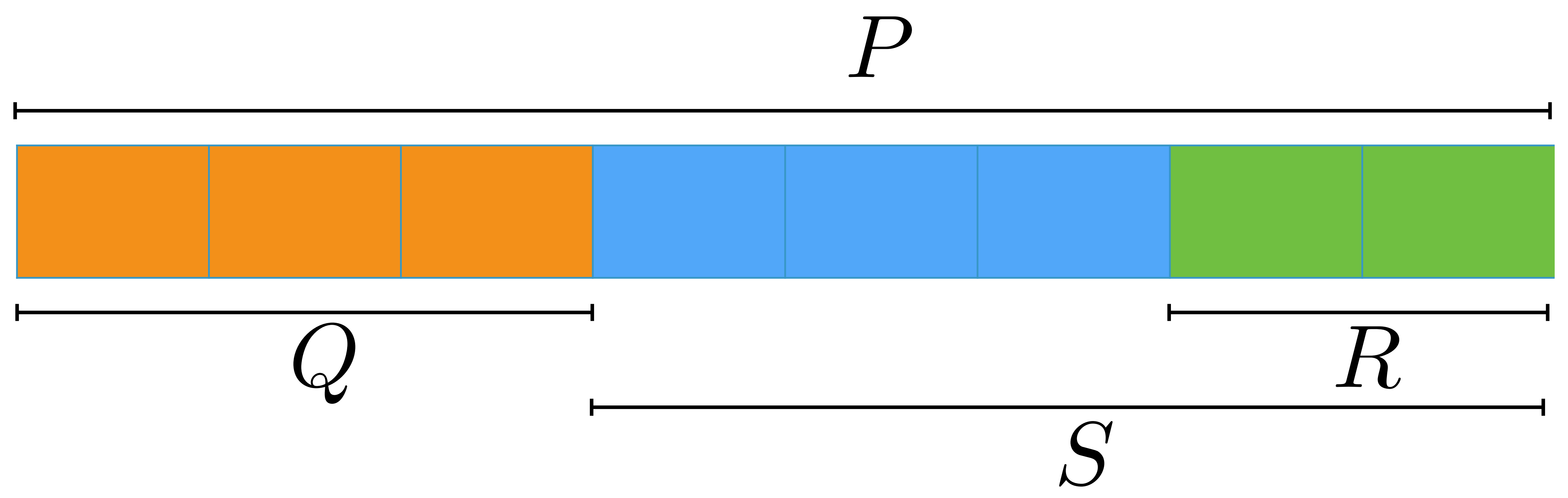
$\textsc{Subtract}(P, Q, A)$ Find F such that

$$P \Rightarrow \exists A.\ Q * F$$



$\textsc{May-Subtract}(P, Q, R)$ Find S such that S is the result of removing Q and R from P and adding R back

$$h_1 \uplus h_2 \models P \wedge h_1 \models Q \wedge h_2 \models R * \mathit{true} \;\; \Rightarrow \;\; h_2 \models S$$



Stabilization

For $P = x \mapsto 1 * y \mapsto 2$ $\qquad Q = a \mapsto b$

Stabilization

For $P = x \mapsto 1 * y \mapsto 2 \qquad Q = a \mapsto b$

$\textsc{Subtract}(P, Q, \emptyset)$ fails since we can't prove that a is always allocated

Stabilization

For $P = x \mapsto 1 * y \mapsto 2$ $\quad Q = a \mapsto b$

$\textsc{Subtract}(P, Q, \emptyset)$ fails since we can't prove that a is always allocated

$\textsc{May-Substract}(P, Q, \mathsf{emp})$

$$(a = x \wedge b = 1 \wedge y \mapsto 2) \vee (a = y \wedge b = 2 \wedge x \mapsto 1)$$

Stabilization

For $P = x \mapsto 1 * y \mapsto 2$ $Q = a \mapsto b$

$\textsc{Subtract}(P, Q, \emptyset)$ fails since we can't prove that a is always allocated

$\textsc{May-Substract}(P, Q, \mathsf{emp})$

$$(a = x \wedge b = 1 \wedge y \mapsto 2) \vee (a = y \wedge b = 2 \wedge x \mapsto 1)$$

$\textsc{May-Substract}(P, \mathsf{emp}, Q)$

$$(a = x \wedge b = 1 \wedge x \mapsto 1 * y \mapsto 2) \vee$$
$$(a = y \wedge b = 2 \wedge x \mapsto 1 * y \mapsto 2)$$

Stabilization

For $P = x \mapsto 1 * y \mapsto 2$ $\qquad Q = a \mapsto b$

$\textsc{Subtract}(P, Q, \emptyset)$ fails since we can't prove that a is always allocated

$\textsc{May-Substract}(P, Q, \textsf{emp})$

$$(a = x \wedge b = 1 \wedge y \mapsto 2) \vee (a = y \wedge b = 2 \wedge x \mapsto 1)$$

$\textsc{May-Substract}(P, \textsf{emp}, Q)$

$$(a = x \wedge b = 1 \wedge x \mapsto 1 * y \mapsto 2) \vee$$
$$(a = y \wedge b = 2 \wedge x \mapsto 1 * y \mapsto 2)$$

```
$\textsc{Stabilize}(S, \mathit{Rely})$
  **repeat**
    $S_{\text{old}} \leftarrow S$
    **for all** $(R \mid P \rightsquigarrow Q) \in \mathit{Rely}$ **do**
      $S \leftarrow S \vee \textsc{Abstract}(\textsc{May-Subtract}(S, P, R) * Q)$
  **until** $S = S_{\text{old}}$
  **return** $S$
```

Action Inference

- We extend the symbolic execution algorithm to obtain potential Guarantees
- We iterate the procedure until we find a fixpoint

```
$\textsc{Infer-Actions}(\mathit{init}, \mathit{Ms})$
    $G \leftarrow \emptyset$
    $(-, \mathit{Inv}) \leftarrow \textsc{Symb-Exec}(\mathsf{emp}, \emptyset, \mathit{init})$
    repeat
        $G_{\text{old}} \leftarrow G$
        $\mathit{Inv} \leftarrow \textsc{Stabilize}(\mathit{Inv}, G)$
        for all $C \in \mathit{Ms}$ do
            $(G_{\text{new}}, -) \leftarrow \textsc{Symb-Exec}(\boxed{\mathit{Inv}}, G, C)$
            $G \leftarrow G \cup G_{\text{new}}$
    until $G = G_{\text{old}}$
    return $(G, \mathit{Inv})$
```

Extended Symbolic Execution

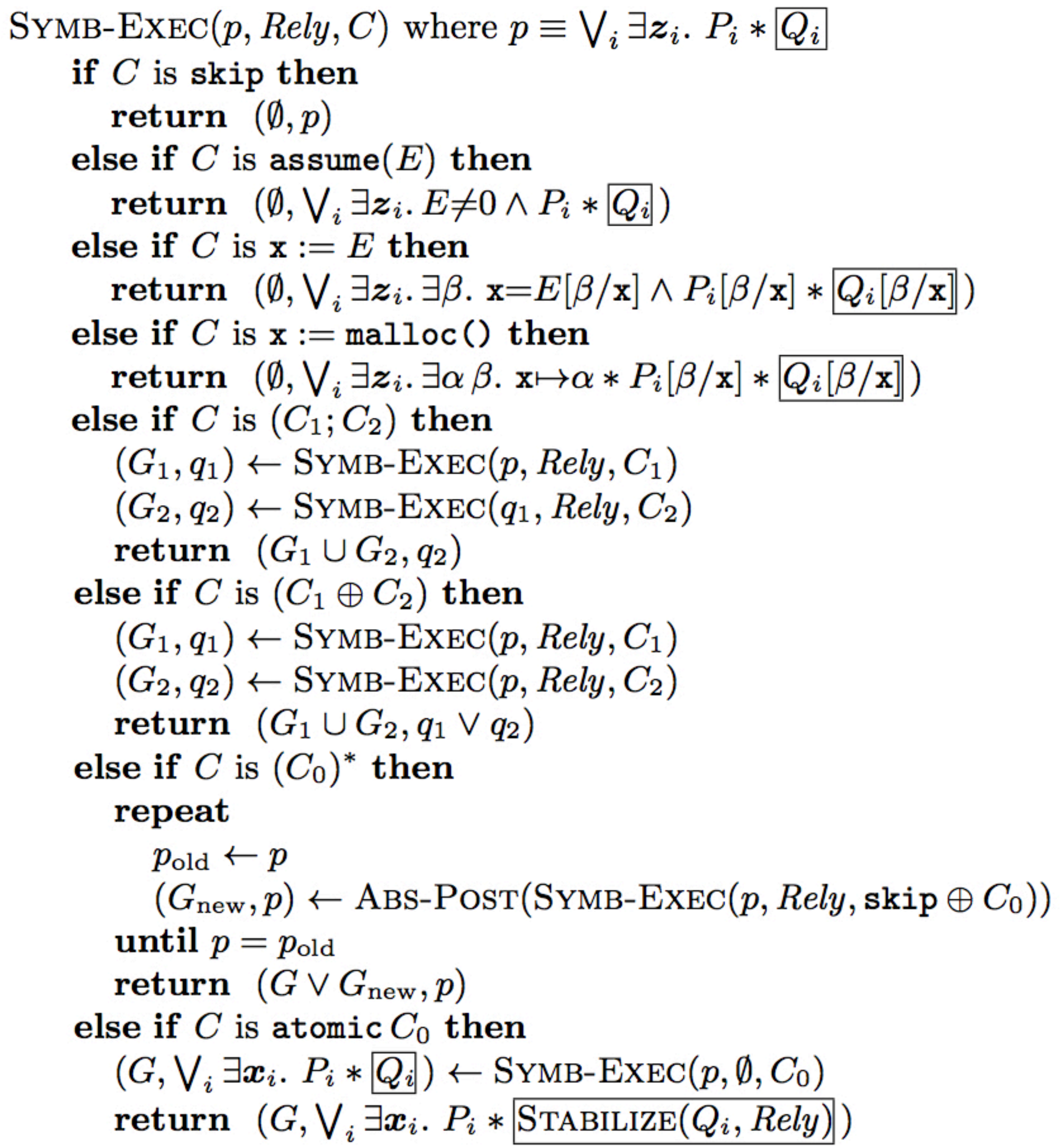
Reads

```
$\textsc{Symb-Exec}(\exists \boldsymbol{z}.\ P_{\mathrm{L}} * \boxed{P_{\mathrm{S}}}, \mathit{Rely}, \mathbf{x} := [E])$
  if $\textsc{Subtract}(P_{\mathrm{L}}, E \mapsto \alpha, \{\alpha\}) = R_{\mathrm{L}}$ then
    return $(\emptyset,\ \exists \boldsymbol{z}\, \alpha\, \beta.\ \mathbf{x} = \alpha \wedge E \mapsto \alpha * R_{\mathrm{L}}[\beta/\mathbf{x}] * \boxed{P_{\mathrm{S}}[\beta/\mathbf{x}]})$
  else if inside an atomic block and $\textsc{Subtract}(P_{\mathrm{S}}, E \mapsto \alpha, \{\alpha\}) = R_{\mathrm{S}}$ then
    return $(\emptyset,\ \exists \boldsymbol{z}\, \alpha\, \beta.\ \mathbf{x} = \alpha \wedge P_{\mathrm{L}}[\beta/\mathbf{x}] * \boxed{E \mapsto \alpha * R_{\mathrm{S}}[\beta/\mathbf{x}]})$
  else
    return $\textsc{Error}$
```

Writes

```
$\textsc{Symb-Exec}(\exists \boldsymbol{z}.\ P_{\mathrm{L}} * \boxed{P_{\mathrm{S}}}, \mathit{Rely}, [E] := E')$
  if $\textsc{Subtract}(P_{\mathrm{L}}, E \mapsto \alpha, \{\alpha\}) = R_{\mathrm{L}}$ then
    return $(\emptyset,\ \exists \boldsymbol{z}.\ E \mapsto E' * R_{\mathrm{L}} * \boxed{P_{\mathrm{S}}})$
  else if inside an atomic block and $\textsc{Subtract}(P_{\mathrm{S}}, E \mapsto \alpha, \{\alpha\}) = R_{\mathrm{S}}$ then
    $(P_{\mathrm{L2S}}, P'_{\mathrm{L}}) \leftarrow \textsc{Reachable-Split}(P_{\mathrm{L}}, E \mapsto E')$
    $\mathit{act} \leftarrow \textsc{A-Abs}(R_{\mathrm{S}} \mid E \mapsto \alpha \rightsquigarrow E \mapsto E' * P_{\mathrm{L2S}})$
    return $(\{\mathit{act}\},\ \exists \boldsymbol{z}.\ P'_{\mathrm{L}} * \boxed{E \mapsto E' * P_{\mathrm{L2S}} * R_{\mathrm{S}}})$
  else
    return $\textsc{Error}$
```

Extended Symbolic Execution



```
$\textsc{Symb-Exec}(p, \mathit{Rely}, C)$ where $p \equiv \bigvee_i \exists \boldsymbol{z}_i.\ P_i * \boxed{Q_i}$
  if $C$ is skip then
    return $(\emptyset, p)$
  else if $C$ is assume$(E)$ then
    return $(\emptyset, \bigvee_i \exists \boldsymbol{z}_i.\, E{\neq}0 \wedge P_i * \boxed{Q_i})$
  else if $C$ is $\mathbf{x} := E$ then
    return $(\emptyset, \bigvee_i \exists \boldsymbol{z}_i.\, \exists \beta.\ \mathbf{x}{=}E[\beta/\mathbf{x}] \wedge P_i[\beta/\mathbf{x}] * \boxed{Q_i[\beta/\mathbf{x}]})$
  else if $C$ is $\mathbf{x} :=$ malloc() then
    return $(\emptyset, \bigvee_i \exists \boldsymbol{z}_i.\, \exists \alpha\, \beta.\ \mathbf{x}{\mapsto}\alpha * P_i[\beta/\mathbf{x}] * \boxed{Q_i[\beta/\mathbf{x}]})$
  else if $C$ is $(C_1; C_2)$ then
    $(G_1, q_1) \leftarrow \textsc{Symb-Exec}(p, \mathit{Rely}, C_1)$
    $(G_2, q_2) \leftarrow \textsc{Symb-Exec}(q_1, \mathit{Rely}, C_2)$
    return $(G_1 \cup G_2, q_2)$
  else if $C$ is $(C_1 \oplus C_2)$ then
    $(G_1, q_1) \leftarrow \textsc{Symb-Exec}(p, \mathit{Rely}, C_1)$
    $(G_2, q_2) \leftarrow \textsc{Symb-Exec}(p, \mathit{Rely}, C_2)$
    return $(G_1 \cup G_2, q_1 \vee q_2)$
  else if $C$ is $(C_0)^*$ then
    repeat
      $p_{\text{old}} \leftarrow p$
      $(G_{\text{new}}, p) \leftarrow \textsc{Abs-Post}(\textsc{Symb-Exec}(p, \mathit{Rely}, \textbf{skip} \oplus C_0))$
    until $p = p_{\text{old}}$
    return $(G \vee G_{\text{new}}, p)$
  else if $C$ is atomic $C_0$ then
    $(G, \bigvee_i \exists \boldsymbol{x}_i.\ P_i * \boxed{Q_i}) \leftarrow \textsc{Symb-Exec}(p, \emptyset, C_0)$
    return $(G, \bigvee_i \exists \boldsymbol{x}_i.\ P_i * \boxed{\textsc{Stabilize}(Q_i, \mathit{Rely})})$
```

Automatically Proving Linearizability

CAVE

- It all comes together in a tool called CAVE
- CAVE takes as input
  - A concurrent data structure specification
    - Uses specific abstract constructs for lists, sets, etc.
  - A concurrent data structure implementation (C-like)
  - Checks for linearizability

Some tricks

- Pure verification checker
  - Linearizable executions that do not modify the state are usually hard to check, additional state is added to those cases
- Action Inference
- Linearization discovery (normally where the abstract state is modified by writing in the shared state)

CAVE (homework)

<u>Some examples in CAVE</u>

- Stacks
  - Treiber
  - Treiber Extended
- Queues
  - 2 Lock Queue
  - DGK Queue
  - MS Queue
- Sets
  - CG Set
  - Noam Set
  - LC Set

CAVE

$$\&\mathsf{S}\mapsto y * \&\mathsf{Abs} \mapsto A \;\rightsquigarrow\; \&\mathsf{S}\mapsto x * x\mapsto \mathsf{Cell}(v,y) * \&\mathsf{Abs} \mapsto v\cdot A \qquad \text{(Push)}$$

$$\&\mathsf{S}\mapsto x * x\mapsto \mathsf{Cell}(v,y) * \&\mathsf{Abs} \mapsto v\cdot A \;\rightsquigarrow\; \&\mathsf{S}\mapsto y * x\mapsto \mathsf{Cell}(v,y) * \&\mathsf{Abs} \mapsto A \qquad \text{(Pop)}$$

```
void push (value v) {
  Cell t, x;
  $\{\mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} * \mathit{StackInv}\}$
  x := new Cell();
  x.data := v;
  $\left\{\begin{array}{l}\mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} \\ * \mathtt{x}\mapsto\mathsf{Cell}(\mathtt{v},\_) * \mathit{StackInv}\end{array}\right\}$
  do {
    $\left\{\begin{array}{l}\mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} \\ * \mathtt{x}\mapsto\mathsf{Cell}(\mathtt{v},\_) * \mathit{StackInv}\end{array}\right\}$
    $\langle$t := S;$\rangle$
    x.next := t;
    $\left\{\begin{array}{l}\mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} \\ * \mathtt{x}\mapsto\mathsf{Cell}(\mathtt{v},\mathtt{t}) * \mathit{StackInv}\end{array}\right\}$
  } while ($\neg$CAS$_{\mathtt{this}}$(&S, t, x));
  $\{\mathtt{AbsResult} \overset{s}{\mapsto} \mathtt{v} * \mathit{StackInv}\}$
}
```

```
value pop () {
  Cell t, x, temp;
  $\{\mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} * \mathit{StackInv}\}$
  do {
    $\langle$t := S; Lin$_{\mathtt{this}}$(t = null);$\rangle$
    $\left\{\begin{array}{l}(\mathtt{t}{=}\mathtt{null} \wedge \mathtt{AbsResult} \overset{s}{\mapsto} \mathtt{EMPTY} * \mathit{StackInv}) \\ \vee (\exists x.\ \mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} * K(x))\end{array}\right\}$
    if(t = null)  return EMPTY;
    $\{\exists x.\ \mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} * K(x)\}$
    x := t.next;
    $\{\mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} * K(\mathtt{x})\}$
  } while($\neg$CAS$_{\mathtt{this}}$(&S, t, x));
  $\left\{\begin{array}{l}\exists v.\ \mathtt{AbsResult} \overset{s}{\mapsto} v \\ * \boxed{\begin{array}{l}\exists x\,A.\ \&\mathsf{Abs} \mapsto A * \&\mathsf{S} \mapsto x \\ * \mathsf{lseg}(x,\mathtt{null},A) * \mathtt{x}\mapsto\mathsf{Cell}(v,\_) * \mathit{true}\end{array}}\end{array}\right\}$
  temp := t.data;
  $\{\exists v.\ \mathtt{AbsResult} \overset{s}{\mapsto} \mathtt{temp} * \mathit{StackInv}\}$
  return temp;
}
```

$$\mathit{StackInv} \stackrel{\text{def}}{=} \boxed{\exists x\,A.\ \&\mathsf{S} \mapsto x * \&\mathsf{Abs} \mapsto A * \mathsf{lseg}(x,\mathtt{null},A) * \mathit{true}}$$

$$K(y) \stackrel{\text{def}}{=} \boxed{\begin{array}{l}\left(\begin{array}{l}\exists x\,v\,A\,B.\ \&\mathsf{Abs} \mapsto A\cdot v\cdot B * \&\mathsf{S} \mapsto x * \mathsf{lseg}(x,\mathtt{t},A) \\ * \mathtt{t}\mapsto\mathsf{Cell}(v,y) * \mathsf{lseg}(y,\mathtt{null},B) * \mathit{true}\end{array}\right) \\ \vee (\exists x\,A.\ \&\mathsf{Abs} \mapsto A * \&\mathsf{S} \mapsto x * \mathsf{lseg}(x,\mathtt{null},A) * \mathtt{t}\mapsto\mathsf{Cell}(\_,\_) * \mathit{true})\end{array}}$$

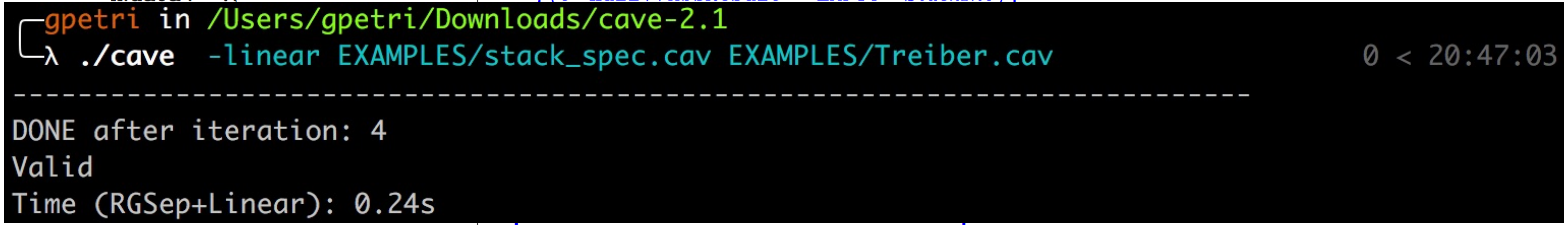
CAVE

$$\&\mathsf{S}{\mapsto}y * \&\mathsf{Abs} \mapsto A \ \rightsquigarrow\ \&\mathsf{S}{\mapsto}x * x{\mapsto}\mathsf{Cell}(v,y) * \&\mathsf{Abs} \mapsto v{\cdot}A \qquad \text{(Push)}$$

$$\&\mathsf{S}{\mapsto}x * x{\mapsto}\mathsf{Cell}(v,y) * \&\mathsf{Abs} \mapsto v{\cdot}A \ \rightsquigarrow\ \&\mathsf{S}{\mapsto}y * x{\mapsto}\mathsf{Cell}(v,y) * \&\mathsf{Abs} \mapsto A \qquad \text{(Pop)}$$

```
void push (value v) {
  Cell t, x;
  {$\mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} * \mathit{StackInv}$}
  x := new Cell();
  x.data := v;
      {$\mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef}$
       $* \mathtt{x}{\mapsto}\mathsf{Cell}(\mathtt{v},\mathtt{t}) * \mathit{StackInv}$}
  } while ($\neg$CAS$_{\mathtt{this}}$(&S, t, x));
  {$\mathtt{AbsResult} \overset{s}{\mapsto} \mathtt{v} * \mathit{StackInv}$}
}
```

```
value pop () {
  Cell t, x, temp;
  {$\mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} * \mathit{StackInv}$}
  do {
    ⟨t := S; Lin$_{\mathtt{this}}$(t = null);⟩
    {(t=null $\wedge$ $\mathtt{AbsResult} \overset{s}{\mapsto} \mathtt{EMPTY} * \mathit{StackInv}$)
  {  * $\boxed{\exists x\, A.\ \&\mathsf{Abs} \mapsto A * \&\mathsf{S} \mapsto x * \mathsf{lseg}(x,\mathtt{null},A) * \mathtt{x}{\mapsto}\mathsf{Cell}(v,\_) * \mathit{true}}$ }
  temp := t.data;
  {$\exists v.\ \mathtt{AbsResult} \overset{s}{\mapsto} \mathtt{temp} * \mathit{StackInv}$}
  return temp;
}
```



```
┌gpetri in /Users/gpetri/Downloads/cave-2.1
└λ ./cave  -linear EXAMPLES/stack_spec.cav EXAMPLES/Treiber.cav                0 < 20:47:03
---------------------------------------------------------------------------------
DONE after iteration: 4
Valid
Time (RGSep+Linear): 0.24s
```

$$\mathit{StackInv} \overset{\text{def}}{=} \boxed{\exists x\, A.\ \&\mathsf{S} \mapsto x * \&\mathsf{Abs} \mapsto A * \mathsf{lseg}(x,\mathtt{null},A) * \mathit{true}}$$

$$K(y) \overset{\text{def}}{=} \boxed{\begin{array}{l}\left(\begin{array}{l}\exists x\, v\, A\, B.\ \&\mathsf{Abs} \mapsto A{\cdot}v{\cdot}B * \&\mathsf{S} \mapsto x * \mathsf{lseg}(x,\mathtt{t},A) \\ * \ \mathtt{t}{\mapsto}\mathsf{Cell}(v,y) * \mathsf{lseg}(y,\mathtt{null},B) * \mathit{true}\end{array}\right) \\ \vee\ (\exists x\, A.\ \&\mathsf{Abs} \mapsto A * \&\mathsf{S} \mapsto x * \mathsf{lseg}(x,\mathtt{null},A) * \mathtt{t}{\mapsto}\mathsf{Cell}(\_,\_) * \mathit{true})\end{array}}$$

The END