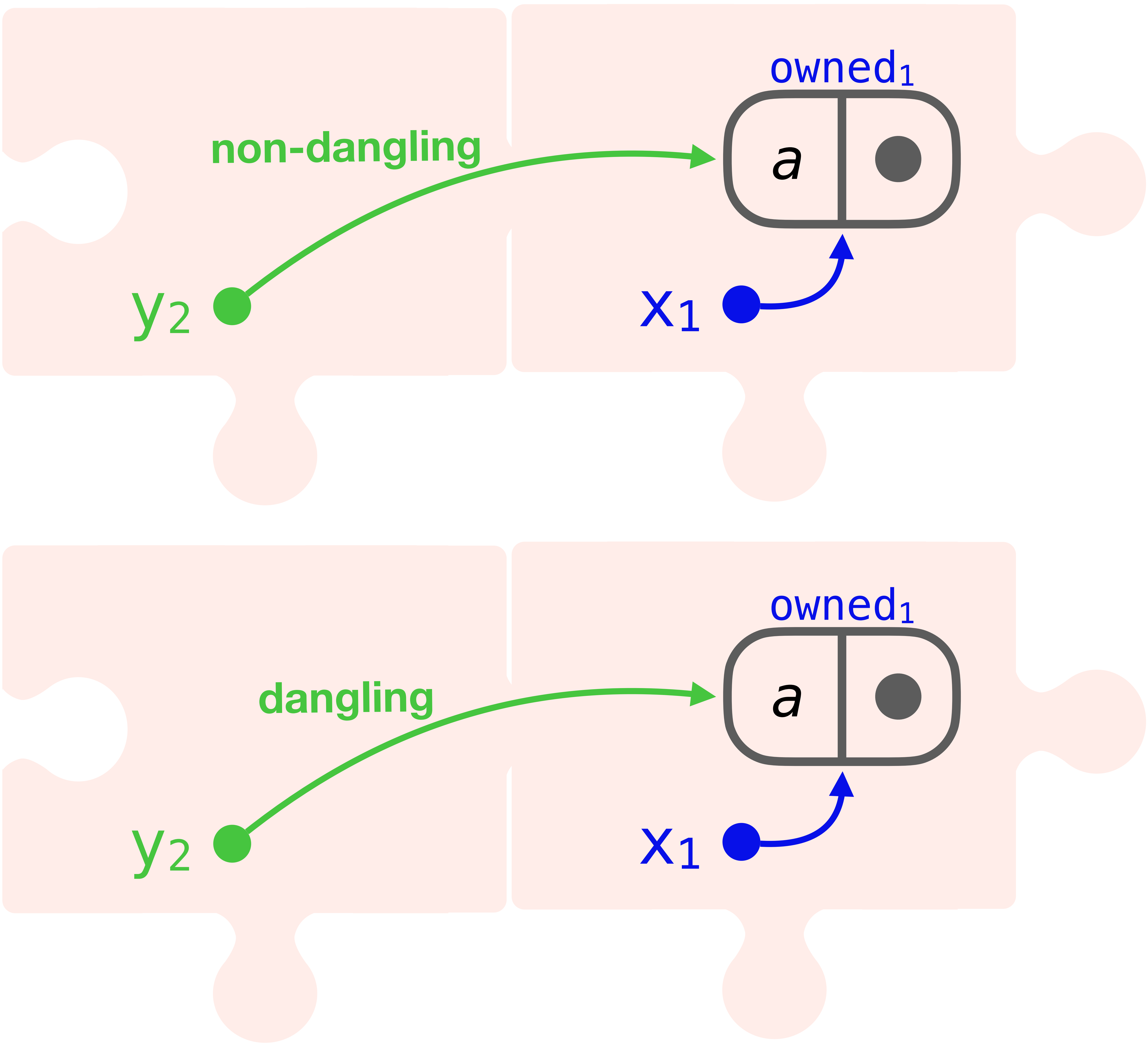
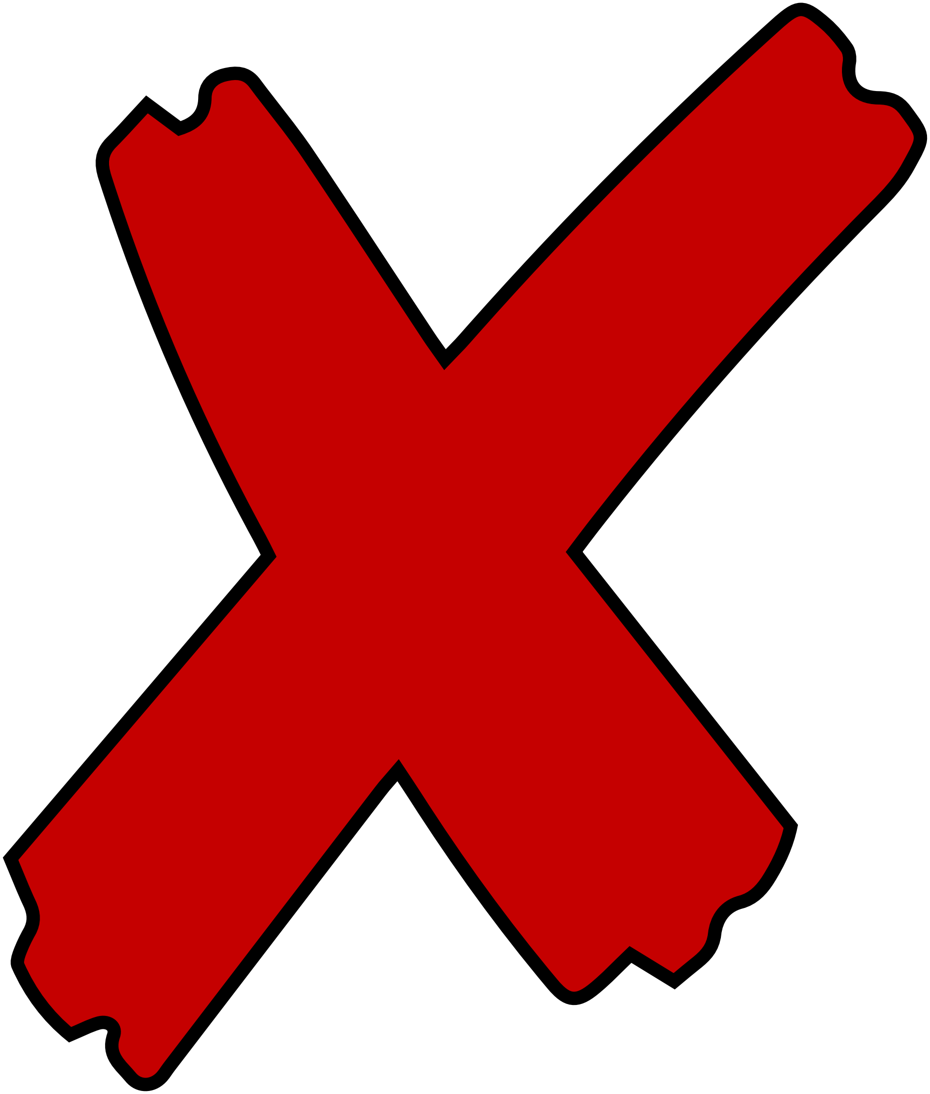


owned$_1$
non-dangling
a
y_2
x_1
owned$_1$
dangling
a
y_2
x_1

Ownership Reasoning

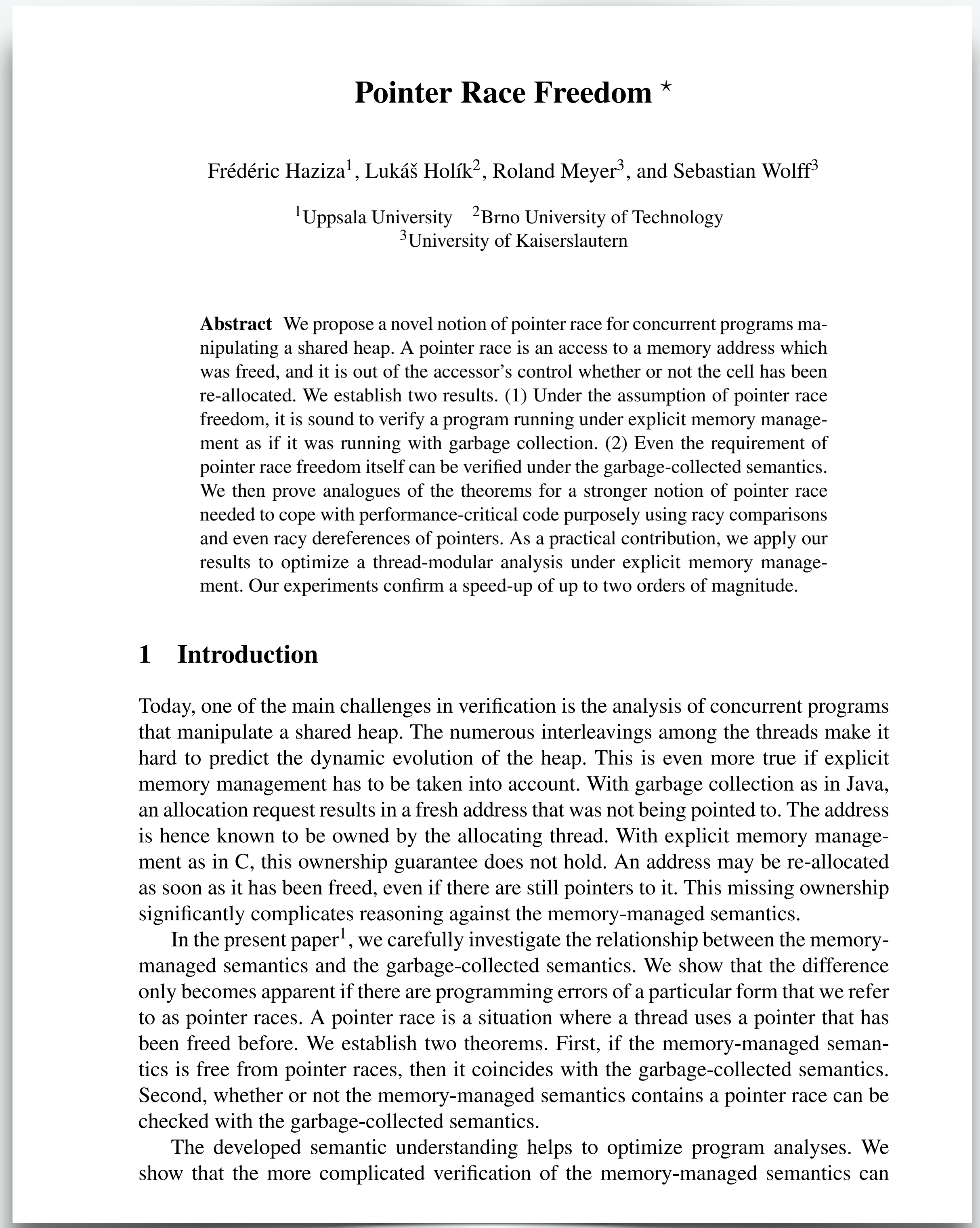
Weak:

- ➡ exclusivity only for non-dangling pointers
- ➡ sufficient precision
- ➡ sound for SMR



[VMCAI'16]

with F. Haziza, L. Holík, and R. Meyer



Pointer Race Freedom *

Frédéric Haziza[1], Lukáš Holík[2], Roland Meyer[3], and Sebastian Wolff[3]

[1]Uppsala University [2]Brno University of Technology
[3]University of Kaiserslautern

Abstract We propose a novel notion of pointer race for concurrent programs manipulating a shared heap. A pointer race is an access to a memory address which was freed, and it is out of the accessor's control whether or not the cell has been re-allocated. We establish two results. (1) Under the assumption of pointer race freedom, it is sound to verify a program running under explicit memory management as if it was running with garbage collection. (2) Even the requirement of pointer race freedom itself can be verified under the garbage-collected semantics. We then prove analogues of the theorems for a stronger notion of pointer race needed to cope with performance-critical code purposely using racy comparisons and even racy dereferences of pointers. As a practical contribution, we apply our results to optimize a thread-modular analysis under explicit memory management. Our experiments confirm a speed-up of up to two orders of magnitude.

1 Introduction

Today, one of the main challenges in verification is the analysis of concurrent programs that manipulate a shared heap. The numerous interleavings among the threads make it hard to predict the dynamic evolution of the heap. This is even more true if explicit memory management has to be taken into account. With garbage collection as in Java, an allocation request results in a fresh address that was not being pointed to. The address is hence known to be owned by the allocating thread. With explicit memory management as in C, this ownership guarantee does not hold. An address may be re-allocated as soon as it has been freed, even if there are still pointers to it. This missing ownership significantly complicates reasoning against the memory-managed semantics.

In the present paper[1], we carefully investigate the relationship between the memory-managed semantics and the garbage-collected semantics. We show that the difference only becomes apparent if there are programming errors of a particular form that we refer to as pointer races. A pointer race is a situation where a thread uses a pointer that has been freed before. We establish two theorems. First, if the memory-managed semantics is free from pointer races, then it coincides with the garbage-collected semantics. Second, whether or not the memory-managed semantics contains a pointer race can be checked with the garbage-collected semantics.

The developed semantic understanding helps to optimize program analyses. We show that the more complicated verification of the memory-managed semantics can

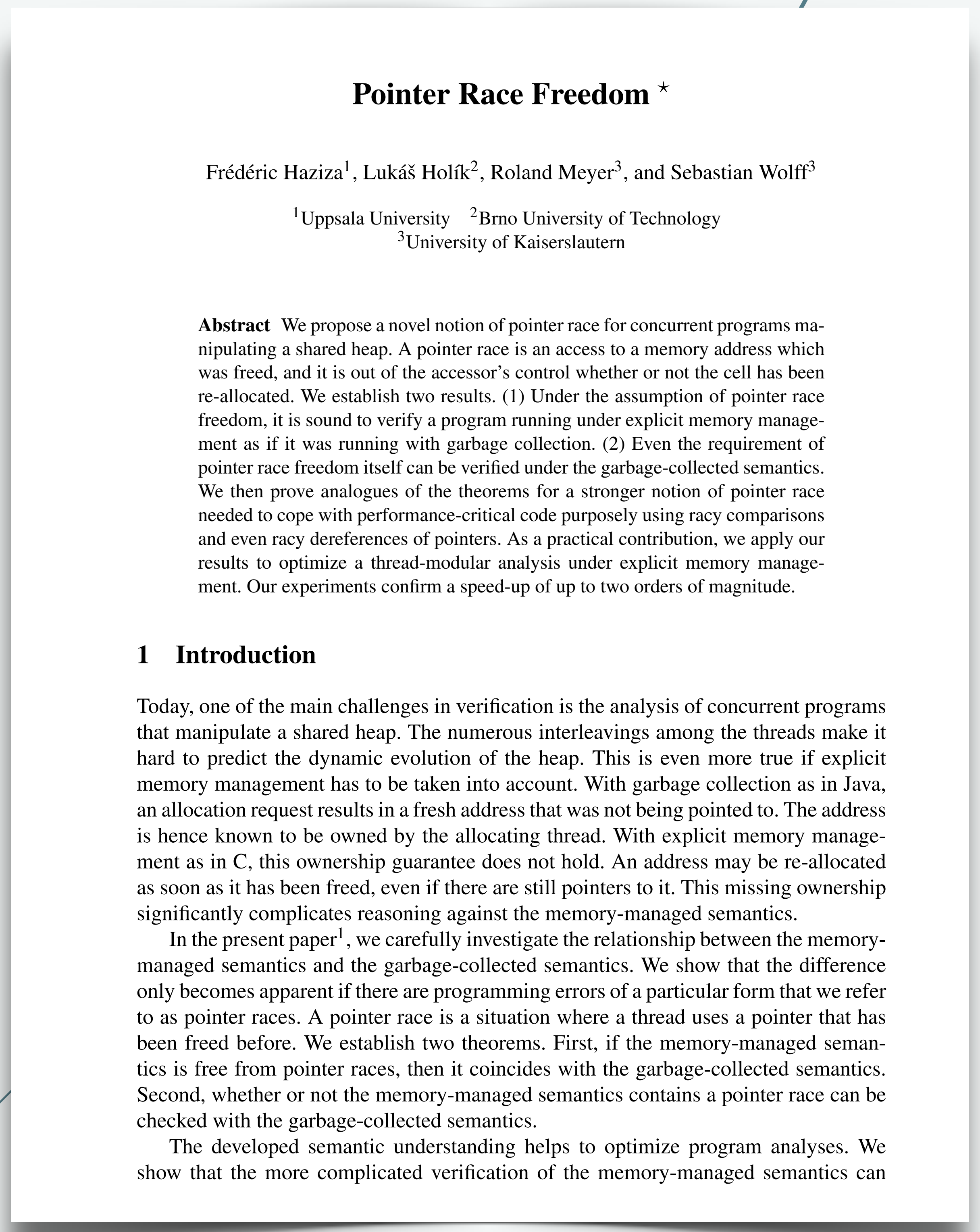
Ownership Reasoning

Weak:

- ➡ exclusivity only for non-dangling pointers
- ➡ sufficient precision
- ➡ sound for SMR

[VMCAI'16]

with F. Haziza, L. Holík, and R. Meyer



Pointer Race Freedom *

Frédéric Haziza[1], Lukáš Holík[2], Roland Meyer[3], and Sebastian Wolff[3]

[1]Uppsala University [2]Brno University of Technology
[3]University of Kaiserslautern

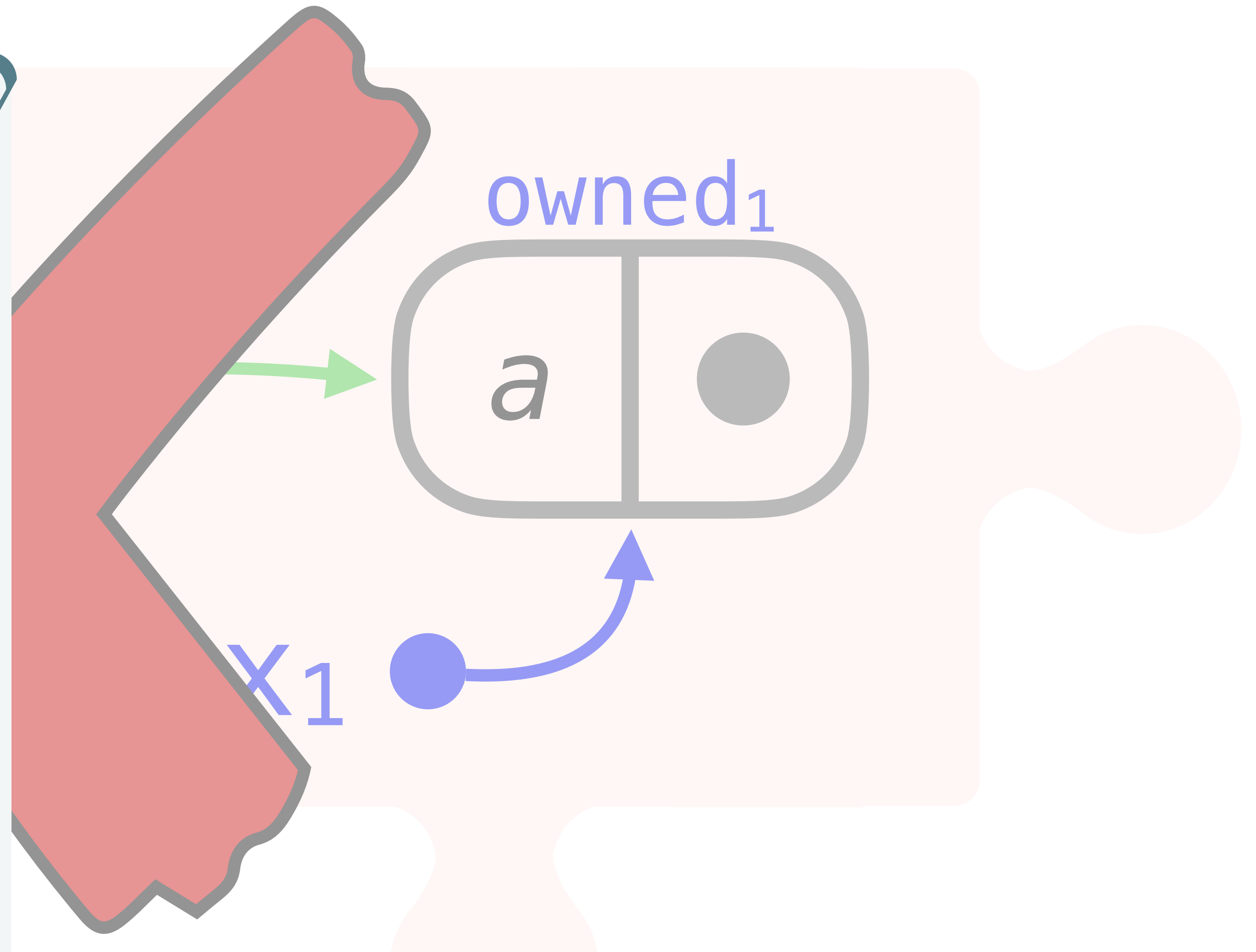
Abstract We propose a novel notion of pointer race for concurrent programs manipulating a shared heap. A pointer race is an access to a memory address which was freed, and it is out of the accessor's control whether or not the cell has been re-allocated. We establish two results. (1) Under the assumption of pointer race freedom, it is sound to verify a program running under explicit memory management as if it was running with garbage collection. (2) Even the requirement of pointer race freedom itself can be verified under the garbage-collected semantics. We then prove analogues of the theorems for a stronger notion of pointer race needed to cope with performance-critical code purposely using racy comparisons and even racy dereferences of pointers. As a practical contribution, we apply our results to optimize a thread-modular analysis under explicit memory management. Our experiments confirm a speed-up of up to two orders of magnitude.

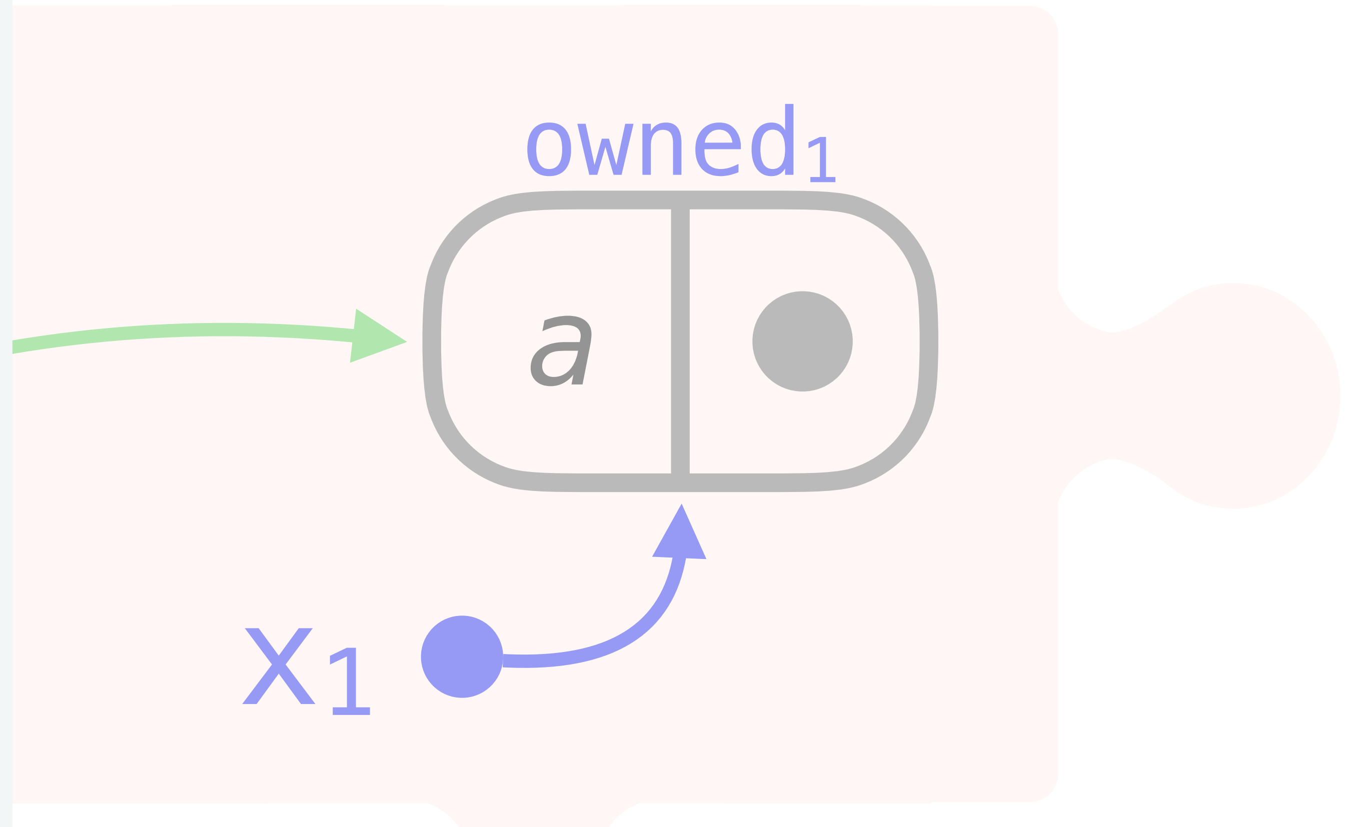
1 Introduction

Today, one of the main challenges in verification is the analysis of concurrent programs that manipulate a shared heap. The numerous interleavings among the threads make it hard to predict the dynamic evolution of the heap. This is even more true if explicit memory management has to be taken into account. With garbage collection as in Java, an allocation request results in a fresh address that was not being pointed to. The address is hence known to be owned by the allocating thread. With explicit memory management as in C, this ownership guarantee does not hold. An address may be re-allocated as soon as it has been freed, even if there are still pointers to it. This missing ownership significantly complicates reasoning against the memory-managed semantics.

In the present paper[1], we carefully investigate the relationship between the memory-managed semantics and the garbage-collected semantics. We show that the difference only becomes apparent if there are programming errors of a particular form that we refer to as pointer races. A pointer race is a situation where a thread uses a pointer that has been freed before. We establish two theorems. First, if the memory-managed semantics is free from pointer races, then it coincides with the garbage-collected semantics. Second, whether or not the memory-managed semantics contains a pointer race can be checked with the garbage-collected semantics.

The developed semantic understanding helps to optimize program analyses. We show that the more complicated verification of the memory-managed semantics can





Experiments

Data Structure with FL	Ownership	Linearizability
Treiber's stack	traditional	0.06s ✓
	weak	2.4s ✓
	none	10m ✓
Michael&Scott's queue	traditional	2.5s ✓
	weak	3h ✓
	none	– ✗