

# A Two-Phase Infinite/Finite Low-Level Memory Model

Reconciling integer–pointer casts, finite space, and undef at the LLVM IR level of abstraction

Calvin Beck<sup>1</sup> Irene Yoon<sup>2</sup> Hanxi Chen<sup>1</sup> Yannick Zakowski<sup>2</sup>  
Steve Zdancewic<sup>1</sup>

<sup>1</sup>University of Pennsylvania

<sup>2</sup>Inria

1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

A Two-Phase Infinite/Finite Low-Level  
Memory Model  
Reconciling integer–pointer casts, finite space, and undef at the  
LLVM IR level of abstraction

Calvin Beck<sup>1</sup> Irene Yoon<sup>2</sup> Hanxi Chen<sup>1</sup> Yannick Zakowski<sup>2</sup>  
Steve Zdancewic<sup>1</sup>

<sup>1</sup>University of Pennsylvania

<sup>2</sup>Inria

## 1. Memory models for low-level languages like C and LLVM IR



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

### └ Overview



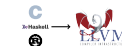
1. LLVM IR is a common compilation target.



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

### └ Overview



1. Used by a variety of higher-level languages as a common backend in order to



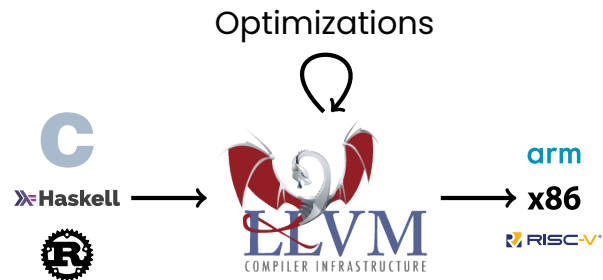
1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

### └ Overview



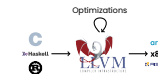
### 1. target different ISAs



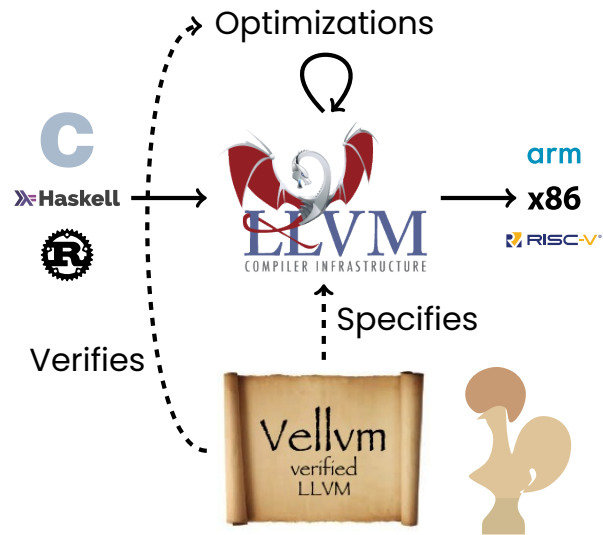
1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

### Overview



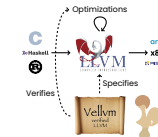
1. Additionally, LLVM is designed to be an excellent place to perform optimizations
2. Especially as these optimization passes can be shared by different frontends



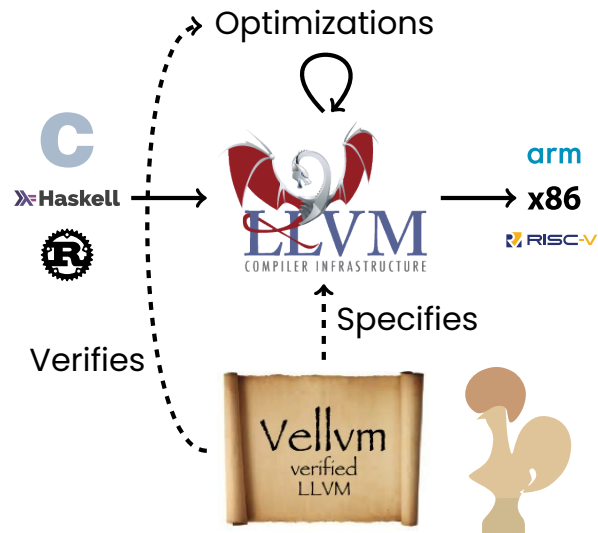
1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

### Overview



1. Vellvm is our semantics and interpreter for LLVM IR in the Coq proof assistant
2. The goals are to be able to...
3. Do proofs about LLVM IR programs
4. And also to be able to verify optimization passes



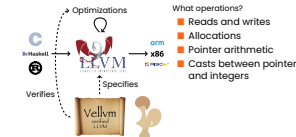
What operations?

- Reads and writes
- Allocations
- Pointer arithmetic
- Casts between pointers and integers

1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

### Overview



1. To define our semantics...
2. Describe how operations on memory behave
3. Things like reads / writes, allocations, and operations on pointers
4. But we also want to define these operations in a way that allows for optimizations

# Infinite / Finite Gap



1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

└ Overview

└ Infinite / Finite Gap

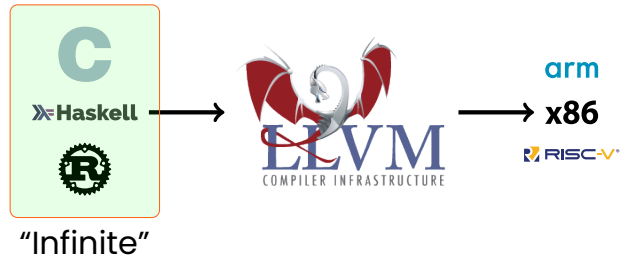
Infinite / Finite Gap



1. We noticed that LLVM sits in the middle between



# Infinite / Finite Gap



1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

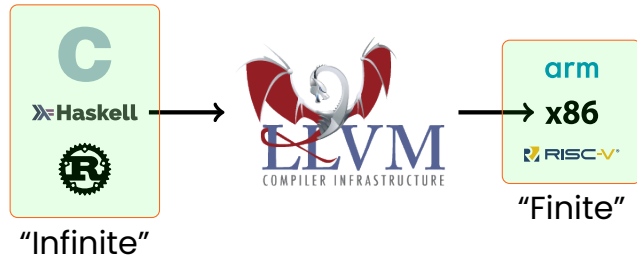
└ Overview

└ Infinite / Finite Gap



1. languages that have infinite memory

# Infinite / Finite Gap



1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

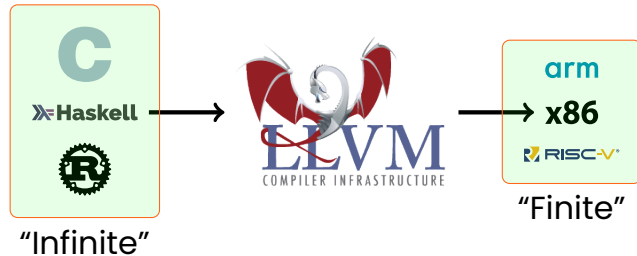
└ Overview

└ Infinite / Finite Gap



1. and assembly languages which explicitly have finite address spaces

# Infinite / Finite Gap



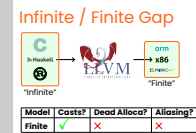
Model	Casts?	Dead Alloca?	Aliasing?
Finite	✓	✗	✗

1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

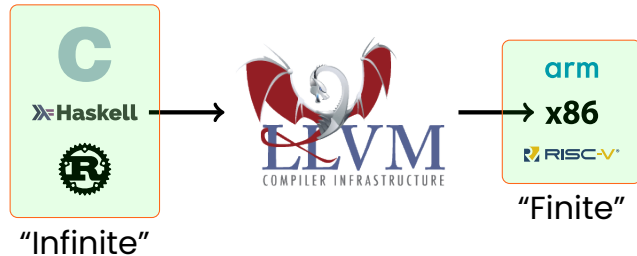
- Overview

- Infinite / Finite Gap



1. Finite memory models are perhaps the most realistic
2. But they cause problems with optimizations
3. Adding and removing allocations can be difficult to justify in a finite setting
4. And it can further complicate alias analysis

# Infinite / Finite Gap



Model	Casts?	Dead Alloca?	Aliasing?
Finite	✓	✗	✗
Infinite	✗	✓	✓

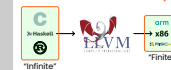
1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

### Overview

### Infinite / Finite Gap

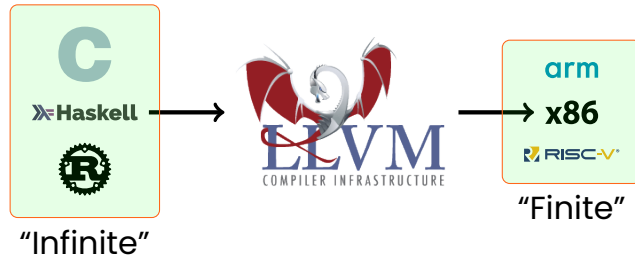
### Infinite / Finite Gap



Model	Casts?	Dead Alloca?	Aliasing?
Finite	✓	✗	✗
Infinite	✗	✓	✓

1. Infinite memory models are more abstract
2. Generally allow a broader range of optimizations
3. Existing models struggle to implement operations which care about the size of pointers, like PTOI casts
4. We will provide a semantics for both infinite and finite LLVM
5. And we will formally bridge the gap between the two

# Infinite / Finite Gap



$$p_1^{inf} \xrightarrow{\exists_{VIR}} p_2^{inf}$$

"Phase 1"

Model	Casts?	Dead Alloca?	Aliasing?
Finite	✓	✗	✗
Infinite	✗	✓	✓

1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

### Overview

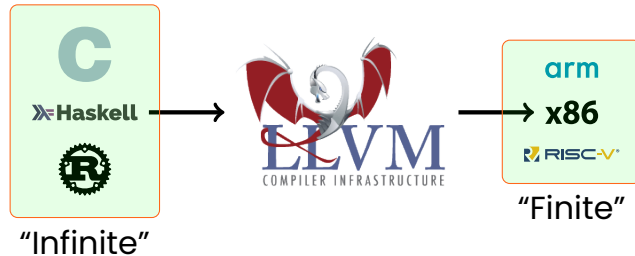
### Infinite / Finite Gap

### Infinite / Finite Gap



1. Programs will be optimized under the infinite LLVM semantics
2. Which allows for more optimizations

# Infinite / Finite Gap



"Phase 1"

$$p_1^{inf} \xrightarrow{\exists_{VIR}} p_2^{inf}$$

Model	Casts?	Dead Alloca?	Aliasing?
Finite	✓	✗	✗
Infinite	✗ → ✓	✓	✓

1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

### Overview

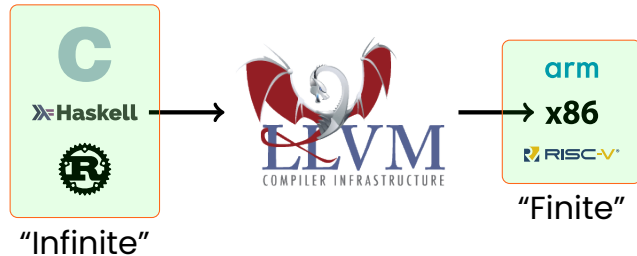
### Infinite / Finite Gap

### Infinite / Finite Gap

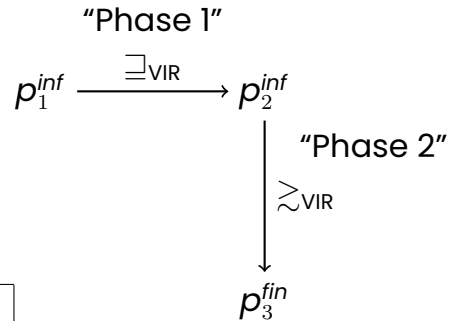


1. And notably we will address an issue with pointer casts, to allow for even more optimizations

# Infinite / Finite Gap



Model	Casts?	Dead Alloca?	Aliasing?
Finite	✓	✗	✗
Infinite	✗ → ✓	✓	✓

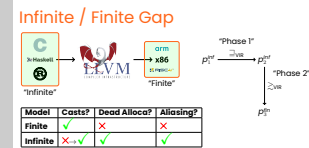


1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

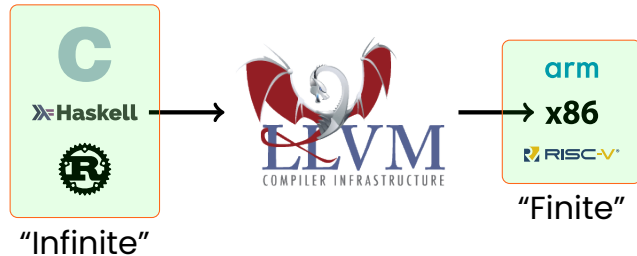
### Overview

### Infinite / Finite Gap

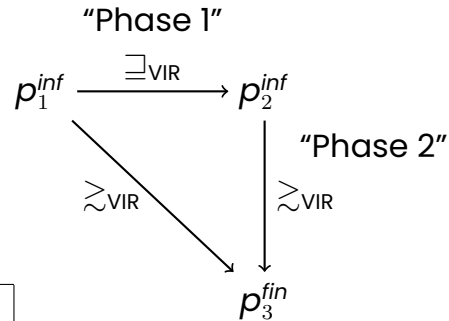


1. After programs have been optimized in infinite LLVM we will safely lower them to finite LLVM
2. Finite LLVM is less amenable to optimizations
3. But it better matches the semantics of the assembly languages in the backend

# Infinite / Finite Gap



Model	Casts?	Dead Alloca?	Aliasing?
Finite	✓	✗	✗
Infinite	✗ → ✓	✓	✓

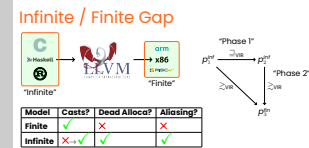


1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

### Overview

### Infinite / Finite Gap



1. This translation has been verified to be correct
2. And you also get a nice transitive property, where the final finite program you get is a refinement of your original program in the infinite semantics



# Perspective: Undefined Behavior

- Optimizations and the semantics of LLVM have to take into account undefined behavior (UB).
- Must condense all memory UB into one succinct and sound memory model 🤪.



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

└ Overview

└ Perspective: Undefined Behavior

### Perspective: Undefined Behavior

- Optimizations and the semantics of LLVM have to take into account undefined behavior (UB).
- Must condense all memory UB into one succinct and sound memory model 🤪.



1. It's hard to convey all of the ways that building a semantics for LLVMs memory is difficult and why existing memory models fall short of our goals in such a short talk.
2. Simply implementing LLVMs memory instructions is not difficult.

# Perspective: Undefined Behavior

- Optimizations and the semantics of LLVM have to take into account undefined behavior (UB).
- Must condense all memory UB into one succinct and sound memory model 🤪.



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

└ Overview

└ Perspective: Undefined Behavior

1. However, our goal is to verify optimization passes in LLVM, and since optimizations depend upon undefined behavior our semantics needs to be able to accurately determine when programs contain undefined behavior.

**Perspective: Undefined Behavior**

- Optimizations and the semantics of LLVM have to take into account undefined behavior (UB).
- Must condense all memory UB into one succinct and sound memory model 🤪.



# Perspective: Undefined Behavior

- Optimizations and the semantics of LLVM have to take into account undefined behavior (UB).
- Must condense all memory UB into one succinct and sound memory model 🤪.



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

└ Overview

└ Perspective: Undefined Behavior

**Perspective: Undefined Behavior**  
■ Optimizations and the semantics of LLVM have to take into account undefined behavior (UB).  
■ Must condense all memory UB into one succinct and sound memory model 🤪.



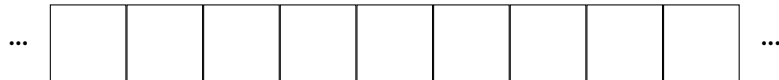
1. This hopefully gives you some understanding of the magnitude of this task.
2. Think about how much subtle and tricky undefined behavior is in languages like C and LLVM.
3. Now consider how painful it would be to condense all of that into an intuitive and sound Coq specification.
4. That is effectively the position we find ourselves in, and plotting a course through this minefield is tricky!

# Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;  
printf("%d\n", (int)p[0]);
```

Optimize →

```
printf("0\n");
```



## A Two-Phase Infinite/Finite Low-Level Memory Model

### └ Overview

### └ Motivating Optimizations: UB

Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;  
printf("%d\n", (int)p[0]);
```

Optimize →

```
printf("0\n");
```

...

--	--	--	--	--	--	--	--	--

...

1. Let's go through a basic example to show the kinds of optimizations we're concerned with, as well as what motivates compilers to introduce UB
2. To do this we need to consider whether the store to the q array may alias with p

# Motivating Optimizations: UB

```
char *p = malloc(3);
```

```
p[0] = 0;
```

```
char *q = malloc(2);
```

```
q[f(0)] = 3;
```

```
printf("%d\n", (int)p[0]);
```

Optimize

```
printf("0\n");
```



1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

└ Overview

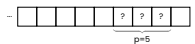
└ Motivating Optimizations: UB

Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;  
printf("%d\n", (int)p[0]);
```


Optimize

```
printf("0\n");
```



1. Stepping through the example, we allocate the p array somewhere in memory...

# Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;   
char *q = malloc(2);  
q[f(0)] = 3;  
printf("%d\n", (int)p[0]);
```

Optimize  $\longrightarrow$  `printf("0\n");`




## A Two-Phase Infinite/Finite Low-Level Memory Model


### Overview

### Motivating Optimizations: UB

Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;   
char *q = malloc(2);  
q[f(0)] = 3;  
printf("%d\n", (int)p[0]);
```

Optimize  $\longrightarrow$  `printf("0\n");`



1. We then store the value 0 at the start of the array.

# Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;  
printf("%d\n", (int)p[0]);
```

Optimize →



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

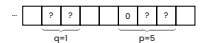
### Overview

### Motivating Optimizations: UB

### Motivating Optimizations: UB


```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;  
printf("%d\n", (int)p[0]);
```

Optimize →



1. Allocate the q array somewhere in memory

# Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;   
printf("%d\n", (int)p[0]);
```

Optimize →

```
printf("0\n");
```



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

### Overview


### Motivating Optimizations: UB

Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;  
printf("%d\n", (int)p[0]);
```

Optimize →

```
printf("0\n");
```



1. And then store the value 3 to a specific index in the `q` array...
2. But where does this value actually get stored?



# Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;  
printf("%d\n", (int)p[0]);
```

Optimize → printf("0\n");

**What is  $f(0)$ ?**

$f(0) = 1$



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

- Overview
- Motivating Optimizations: UB

Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;  
printf("%d\n", (int)p[0]);
```

Optimize → printf("0\n");

**What is  $f(0)$ ?**

$f(0) = 1$

A memory diagram showing two arrays, q and p, in memory. Array q is located at address 0x1 and has 2 elements. The first element (q[0]) contains a question mark, and the second element (q[1]) contains the value 3. Array p is located at address 0x5 and has 3 elements. The first element (p[0]) contains the value 0, and the second and third elements (p[1] and p[2]) contain question marks. A dashed arrow points from the text  $f(0) = 1$  to the element q[1].

1. It could be within the bounds of q...

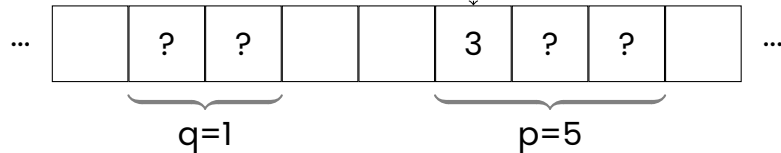
# Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;  
printf("%d\n", (int)p[0]);
```

Optimize → printf("0\n");

**What is  $f(0)$ ?**

$f(0) = 3$



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

└ Overview

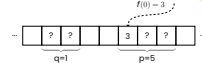
└ Motivating Optimizations: UB

Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;  
printf("%d\n", (int)p[0]);
```

Optimize → printf("0\n");

**What is  $f(0)$ ?**



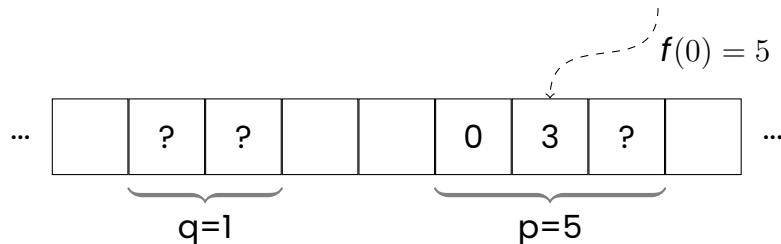
1. Or it could overlap with  $p[0]$

# Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;  
printf("%d\n", (int)p[0]);
```

Optimize → printf("0\n");

**What is  $f(0)$ ?**



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

- Overview
- Motivating Optimizations: UB

Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;  
printf("%d\n", (int)p[0]);
```

Optimize → printf("0\n");

**What is  $f(0)$ ?**

Memory layout diagram showing a sequence of memory cells. A bracket labeled  $q=1$  covers the first two cells, which contain question marks. A bracket labeled  $p=5$  covers the next five cells, which contain 0, 3, and a question mark. A dashed arrow points from the text  $f(0) = 5$  to the third cell of the  $p=5$  block, which contains a question mark.

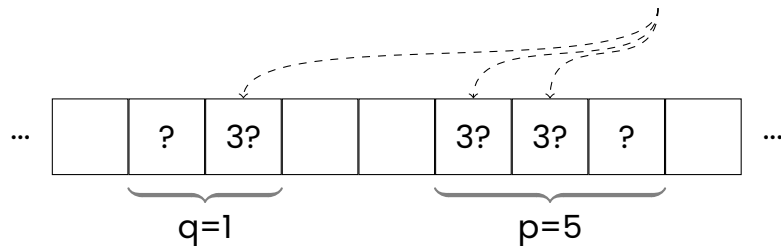
1. Or it could go somewhere else

# Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;  
printf("%d\n", (int)p[0]);
```

Optimize → printf("0\n");

**What is  $f(0)$ ?**



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

- Overview
- Motivating Optimizations: UB

Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;  
printf("%d\n", (int)p[0]);
```

Optimize → printf("0\n");

**What is  $f(0)$ ?**

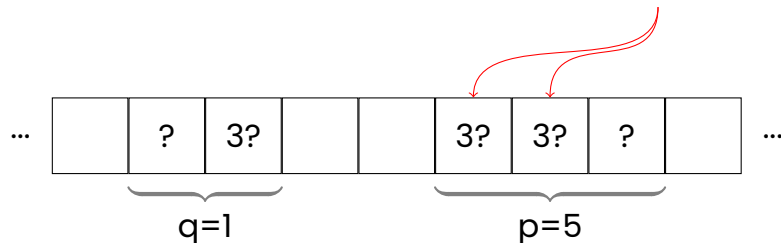
1. We don't know where it's stored!
2. Unless we know where  $p$  and  $q$  are allocated and what value  $f(0)$  is
3. This can be difficult or impossible for a compiler to keep track of.  $f(0)$  could be a complex expression, or an external call that the compiler could not determine the result of.

# Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;   
printf("%d\n", (int)p[0]);
```

Optimize → printf("0\n");

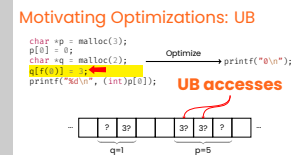
**UB accesses**



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

- Overview
- Motivating Optimizations: UB



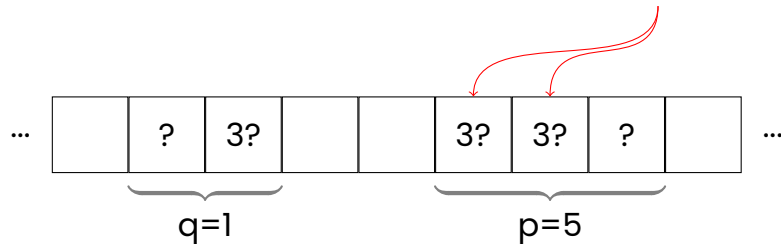
1. Programs and compilers reason in exactly the same way to justify our optimizations, though.
2. Simply assume that the store is in-bounds, and therefore cannot overwrite  $p$ . If that's the case we can perform store forwarding and eventually dead code elimination to get our optimized program.
3. Programs that invalidate this assumption have UB, and optimizations may not be correct on programs with UB.

# Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;   
printf("%d\n", (int)p[0]);
```

Optimize → printf("0\n");

**UB accesses**



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

- Overview
- Motivating Optimizations: UB

Motivating Optimizations: UB

```
char *p = malloc(3);  
p[0] = 0;  
char *q = malloc(2);  
q[f(0)] = 3;  
printf("%d\n", (int)p[0]);
```

Optimize → printf("0\n");

**UB accesses**

A small memory diagram showing a horizontal sequence of memory cells. The first three cells are grouped by a bracket labeled 'q=1' and contain '?', '3?', and an empty cell. The next five cells are grouped by a bracket labeled 'p=5' and contain an empty cell, '3?', '3?', '3?', and '?'. Red arrows point from the text 'UB accesses' to the '3?' values in the overlapping region, indicating out-of-bounds accesses.

1. This is what makes optimizations in low-level languages so tricky.
2. Direct memory access provides many situations for aliasing to arise which blocks optimizations.
3. Compilers are not smart enough to detect things like out-of-bounds accesses, and instead assume they don't happen

# Dead Allocation Removal

Are these equivalent?

```
char *a = malloc(10);  
printf("Hello, world!\n");
```

1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

- └ Overview
  - └ Finite Memory Models
    - └ Dead Allocation Removal

## Dead Allocation Removal

Are these equivalent?

```
char *a = malloc(10);  
printf("Hello, world!\n");
```

1. The next conundrum we'll explore is finite memory.
2. Tempting to make LLVM a finite memory language as it matches the assembly languages it targets
3. Unfortunately, finite memory is disastrous for optimizations which change the amount of memory being allocated!

# Dead Allocation Removal

Are these equivalent?

```
char *a = malloc(10);  
printf("Hello, world!\n");
```

1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

└ Overview

└ Finite Memory Models

└ Dead Allocation Removal

Dead Allocation Removal

Are these equivalent?

```
char *a = malloc(10);  
printf("Hello, world!\n"); printf("Hello, world!\n");
```

1. We'd like these two programs to be equivalent
2. But in a finite world they are not!
3. The program on the left may run out of memory and never print anything.
4. Removing the allocation would allow the program to print instead of running out of memory.
5. But this introduces a new behavior to the program, meaning the optimization is not valid.



# Dead Allocation Removal

Are these equivalent?

```
char *a = malloc(10);  
printf("Hello, world!\n");  
printf("Hello, world!\n");
```

1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

- └ Overview
  - └ Finite Memory Models
    - └ Dead Allocation Removal

## Dead Allocation Removal

Are these equivalent?

```
char *a = malloc(10);  
printf("Hello, world!\n");  
printf("Hello, world!\n");
```

1. All of this gets in the way of important program transformations. This can be problematic for register allocation and spilling.
2. So, finite memory is problematic for optimizations in at least one way, but the problems go even deeper!

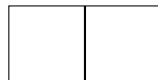
# Finite Address Guessing

```
// Assume a 2-byte memory
char *p = malloc(1);
char *q = malloc(1);
char *x = (char*) (((int) p == 0) ? 1 : 0);
```

Execution 1:



Execution 2:



## A Two-Phase Infinite/Finite Low-Level Memory Model

- Overview
  - Finite Memory Models
    - Finite Address Guessing

### Finite Address Guessing

```
// Assume a 2-byte memory
char *p = malloc(1);
char *q = malloc(1);
char *x = (char*) (((int) p == 0) ? 1 : 0);
```

Execution 1:



Execution 2:

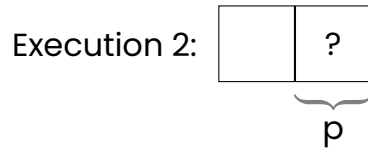
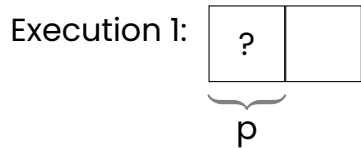


1. Finite memory introduces side-channels for creating aliases (which, again, block optimizations like store-forwarding).
2. Let's consider an example of a 2-byte memory where we allocate two bytes separately.

# Finite Address Guessing

```
// Assume a 2-byte memory
```

```
char *p = malloc(1);  
char *q = malloc(1);  
char *x = (char*) (((int) p == 0) ? 1 : 0);
```



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

- Overview
  - Finite Memory Models
    - Finite Address Guessing

## Finite Address Guessing

```
// Assume a 2-byte memory
```

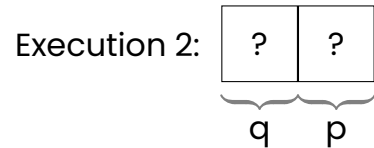
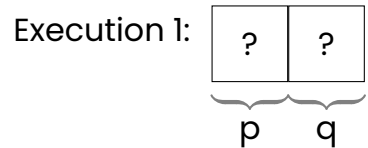
```
char *p = malloc(1);  
char *q = malloc(1);  
char *x = (char*) (((int) p == 0) ? 1 : 0);
```



1. No matter where we allocate the first byte

# Finite Address Guessing

```
// Assume a 2-byte memory  
char *p = malloc(1);  
char *q = malloc(1);  
char *x = (char*) (((int) p == 0) ? 1 : 0);
```



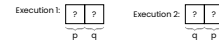
1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

- Overview
- Finite Memory Models
  - Finite Address Guessing

### Finite Address Guessing

```
// Assume a 2-byte memory  
char *p = malloc(1);  
char *q = malloc(1);  
char *x = (char*) (((int) p == 0) ? 1 : 0);
```



1. We know that the second allocation has to be in the other location in memory

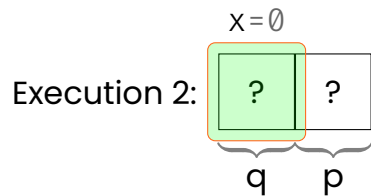
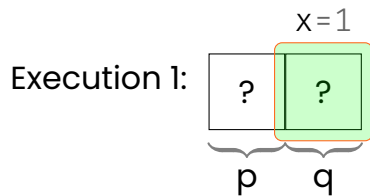
# Finite Address Guessing

```
// Assume a 2-byte memory
```

```
char *p = malloc(1);
```

```
char *q = malloc(1);
```

```
char *x = (char*) (((int) p == 0) ? 1 : 0);
```



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

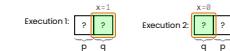
Overview

Finite Memory Models

Finite Address Guessing

## Finite Address Guessing

```
// Assume a 2-byte memory  
char *p = malloc(1);  
char *q = malloc(1);  
char *x = (char*) (((int) p == 0) ? 1 : 0);
```



1. This allows us to construct an alias to the second allocation, even though we have never directly observed its location.
2. This might seem like an artificial problem, and you might be thinking "who cares about this crazy cursed program?"

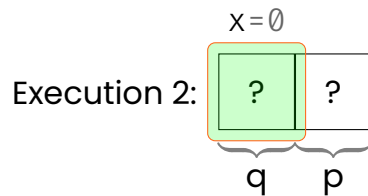
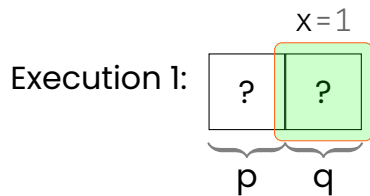
# Finite Address Guessing

```
// Assume a 2-byte memory
```

```
char *p = malloc(1);
```

```
char *q = malloc(1);
```

```
char *x = (char*) (((int) p == 0) ? 1 : 0);
```



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

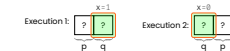
Overview

Finite Memory Models

Finite Address Guessing

## Finite Address Guessing

```
// Assume a 2-byte memory  
char *p = malloc(1);  
char *q = malloc(1);  
char *x = (char*) (((int) p == 0) ? 1 : 0);
```



1. But the issue is if you want to prove an optimization correct, then you will have to consider all programs, and only trigger the optimization when it's valid. Detecting these side-channels is difficult, making it hard for a compiler to rule out aliasing.

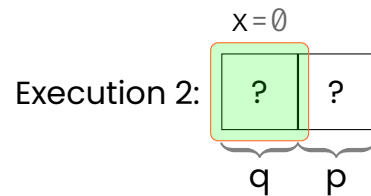
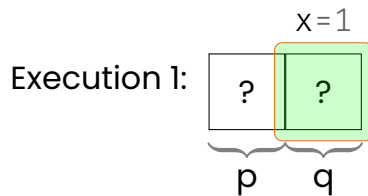
# Finite Address Guessing

```
// Assume a 2-byte memory
```

```
char *p = malloc(1);
```

```
char *q = malloc(1);
```

```
char *x = (char*) (((int) p == 0) ? 1 : 0);
```



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

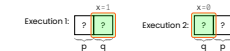
Overview

Finite Memory Models

Finite Address Guessing

## Finite Address Guessing

```
// Assume a 2-byte memory  
char *p = malloc(1);  
char *q = malloc(1);  
char *x = (char*) (((int) p == 0) ? 1 : 0);
```



1. There are tricks to get around this problem, like the twin allocation which uses a clever trick with extra allocations and non-determinism to detect this and trigger UB. But this is pretty sophisticated and has some limitations, and we wanted a simpler solution.

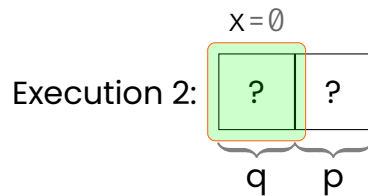
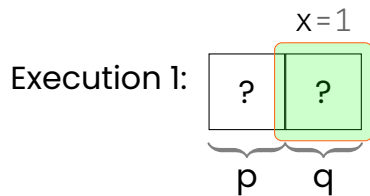
# Finite Address Guessing

```
// Assume a 2-byte memory
```

```
char *p = malloc(1);
```

```
char *q = malloc(1);
```

```
char *x = (char*) (((int) p == 0) ? 1 : 0);
```



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

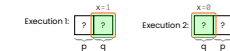
Overview

Finite Memory Models

Finite Address Guessing

## Finite Address Guessing

```
// Assume a 2-byte memory  
char *p = malloc(1);  
char *q = malloc(1);  
char *x = (char*) (((int) p == 0) ? 1 : 0);
```



1. Notably infinite memory models don't suffer from this problem, because we always have an unlimited number of locations that each allocation could reserve.



# Two-Phase Memory: Overview

Finite memory is hard...

- Allocations can't be removed
- Address guessing :(

1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

└ Two-Phase Memory Models

└ Two-Phase Memory: Overview

Two-Phase Memory: Overview

Finite memory is hard...

- Allocations can't be removed
- Address guessing :(

1. So, this is where we get to our two-phase model
2. Finite memory is hard and causes tricky problems for a lot of optimizations

# Two-Phase Memory: Overview

Finite memory is hard...

- Allocations can't be removed
- Address guessing :(

Our solution...

1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

└ Two-Phase Memory Models

└ Two-Phase Memory: Overview

1. So instead, we decided to just...

Two-Phase Memory: Overview

Finite memory is hard...

■ Allocations can't be removed

■ Address guessing :(

Our solution...

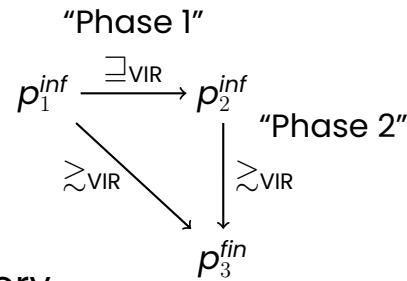
# Two-Phase Memory: Overview

Finite memory is hard...

- Allocations can't be removed
- Address guessing :(

Our solution...

- Pretend we have unlimited memory
- Convert to finite memory program explicitly for assembly backend



1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

└ Two-Phase Memory Models

└ Two-Phase Memory: Overview

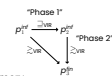
Two-Phase Memory: Overview

Finite memory is hard...

- Allocations can't be removed
- Address guessing :(

Our solution...

- Pretend we have unlimited memory
- Convert to finite memory program explicitly for assembly backend



1. Pretend that we have unlimited memory.
2. We can do all of our optimizations in the well-behaved infinite memory world
3. And then safely convert our infinite programs to finite ones with the same behavior, modulo possibly running out of memory.

# Model Via Specification Monad

$$\begin{aligned} \text{MemSpec}(X) &\triangleq \text{Memory} \rightarrow \mathcal{P}(\text{Result}(\text{Memory} \times X)) \\ \text{Result}(A) &\triangleq \text{UB} + \text{OOM} + \text{ok}(A) \end{aligned}$$

1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

└ Two-Phase Memory Models

└ Model Via Specification Monad

Model Via Specification Monad

$\text{MemSpec}(X) \triangleq \text{Memory} \rightarrow \mathcal{P}(\text{Result}(\text{Memory} \times X))$   
 $\text{Result}(A) \triangleq \text{UB} + \text{OOM} + \text{ok}(A)$

1. We model our primitive memory operations via this MemSpec monad.
2. The MemSpec monad maps initial memory configurations to a set of possible results
3. This may indicate that an operation raises UB
4. Runs out of memory
5. Or the operation can run successfully, yielding the resulting (possibly unchanged) state of memory, as well as the result of the operation with the appropriate type

# Model Via Specification Monad

$$\begin{aligned} \text{MemSpec}(X) &\triangleq \text{Memory} \rightarrow \mathcal{P}(\text{Result}(\text{Memory} \times X)) \\ \text{Result}(A) &\triangleq \text{UB} + \text{OOM} + \text{ok}(A) \end{aligned}$$

- Shared between infinite / finite models.
- Only the size of *Memory* differs.

1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

### └ Two-Phase Memory Models

### └ Model Via Specification Monad

## Model Via Specification Monad

$$\begin{aligned} \text{MemSpec}(X) &\triangleq \text{Memory} \rightarrow \mathcal{P}(\text{Result}(\text{Memory} \times X)) \\ \text{Result}(A) &\triangleq \text{UB} + \text{OOM} + \text{ok}(A) \end{aligned}$$

- Shared between infinite / finite models.
- Only the size of *Memory* differs.

1. The specifications of operations will be shared between both the infinite and finite stages, but the size of the *Memory* component of *MemSpec* will differ between the two

# Specification of Operations

$$\begin{aligned} \text{MemSpec}(X) &\triangleq \text{Memory} \rightarrow \mathcal{P}(\text{Result}(\text{Memory} \times X)) \\ \text{Result}(A) &\triangleq \text{UB} + \text{OOM} + \text{ok}(A) \end{aligned}$$

```

      readb (p : Ptr) : MemSpec(SByte)
writeb (p : Ptr) (b : SByte) : MemSpec(unit)

  alloca ( $\vec{b}$  : list SByte) : MemSpec(Ptr)
  malloc ( $\vec{b}$  : list SByte) : MemSpec(Ptr)
      free (p : Ptr) : MemSpec(unit)

```

1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

### └ Two-Phase Memory Models

### └ Specification of Operations

#### Specification of Operations

$$\begin{aligned} \text{MemSpec}(X) &\triangleq \text{Memory} \rightarrow \mathcal{P}(\text{Result}(\text{Memory} \times X)) \\ \text{Result}(A) &\triangleq \text{UB} + \text{OOM} + \text{ok}(A) \end{aligned}$$

```

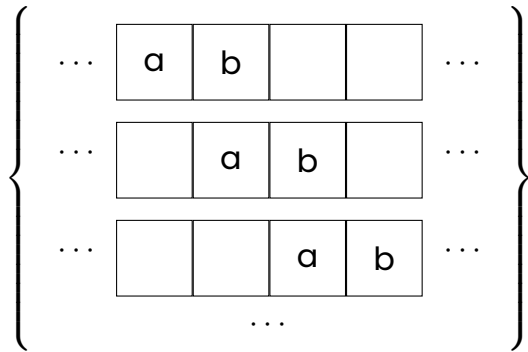
      readb (p : Ptr) : MemSpec(SByte)
writeb (p : Ptr) (b : SByte) : MemSpec(unit)

  alloca ( $\vec{b}$  : list SByte) : MemSpec(Ptr)
  malloc ( $\vec{b}$  : list SByte) : MemSpec(Ptr)
      free (p : Ptr) : MemSpec(unit)

```

1. The operations that we model are reads, writes, stack allocations, heap allocations, and free

# MemSpec Example: malloc([a, b])



Infinite Memory Behaviors

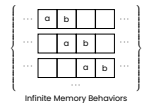
1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

└ Two-Phase Memory Models

└ MemSpec Example: malloc([a, b])

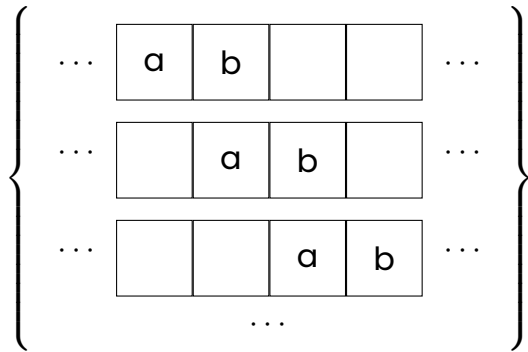
MemSpec Example: malloc([a, b])



Infinite Memory Behaviors

1. Okay, let's take a quick peak at how these specifications work for malloc
2. We'll allocate these two bytes on the heap
3. In the infinite memory model we will get a set containing every possible way we could allocate these bytes in the current memory

# MemSpec Example: malloc([a, b])



Infinite Memory Behaviors

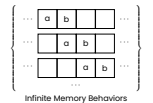
1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

└ Two-Phase Memory Models

└ MemSpec Example: malloc([a, b])

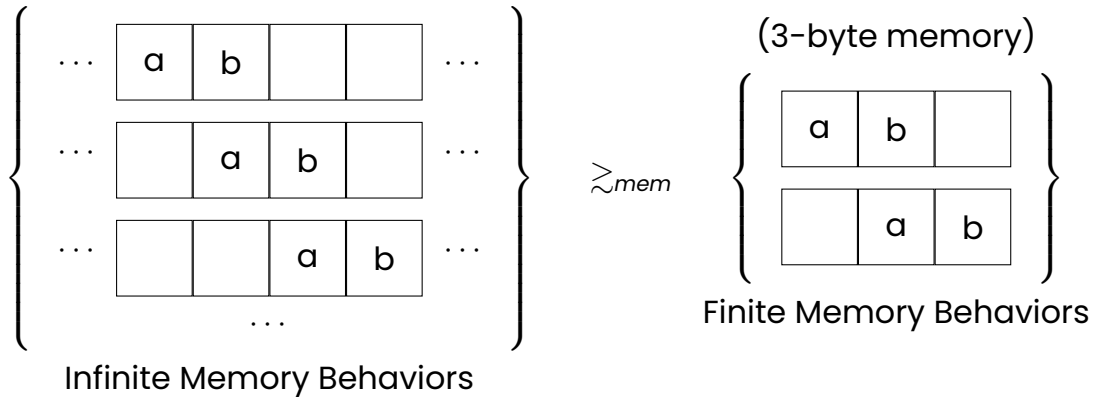
MemSpec Example: malloc([a, b])



1. Our semantics is nondeterministic, allowing us to detect address guessing and model things like pointer comparisons appropriately
2. With infinite memory this is an unbounded set as there are unlimited free locations to allocate from

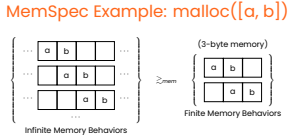


# MemSpec Example: malloc([a, b])



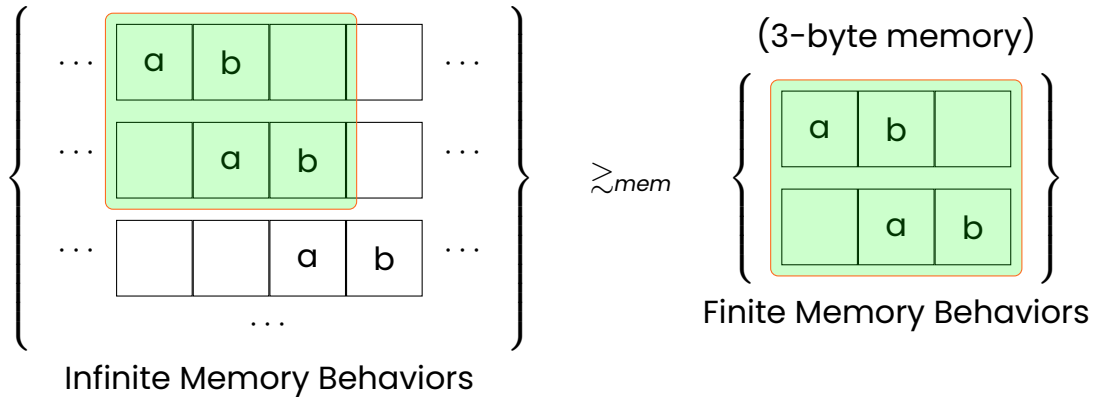
1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model  
 └ Two-Phase Memory Models  
 └ MemSpec Example: malloc([a, b])



1. The malloc specification in a finite model (in this case with a 3-byte memory) will look similar, but notably there will only be a finite number of possible ways to allocate these two bytes

# MemSpec Example: malloc([a, b])



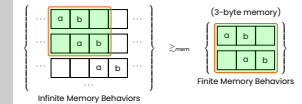
1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

└ Two-Phase Memory Models

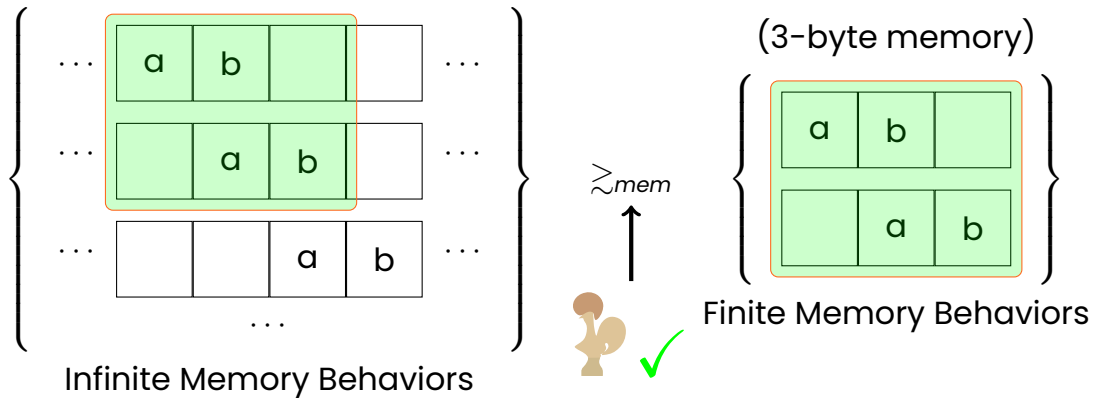
└ MemSpec Example: malloc([a, b])

MemSpec Example: malloc([a, b])



1. We've set up a refinement relation between infinite and finite operations.
2. It ensures that the finite memories on the right-hand-side can be lifted to infinite memories on the left-hand-side
3. This guarantees that finite operations can effectively simulate the infinite ones.

# MemSpec Example: `malloc([a, b])`



1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

└ Two-Phase Memory Models

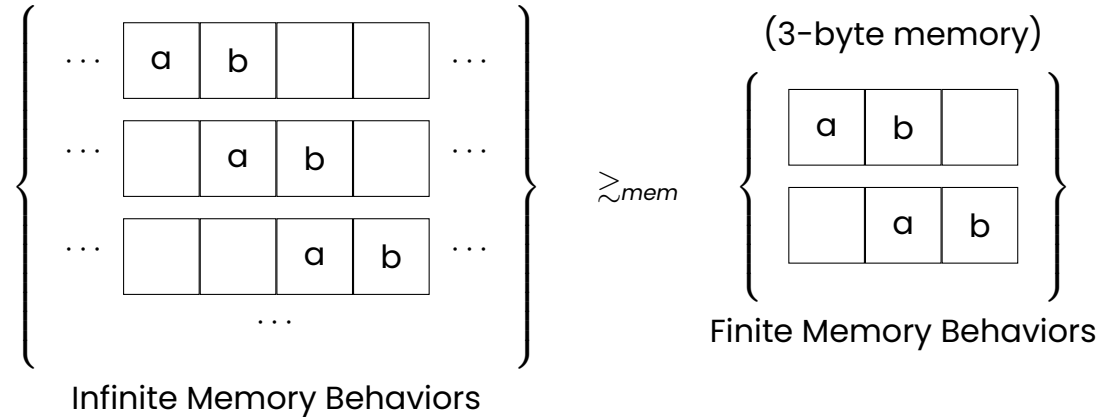
└ MemSpec Example: `malloc([a, b])`

MemSpec Example: `malloc([a, b])`



1. And of course we've verified this relation for every operation in our memory model

# UB and OOM in Refinement



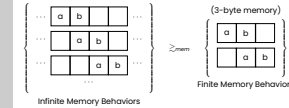
1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

└ Two-Phase Memory Models

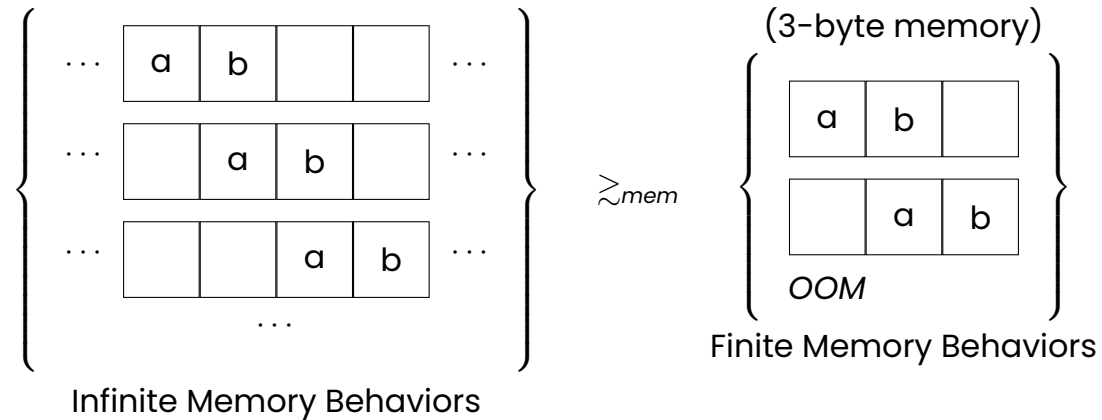
└ UB and OOM in Refinement

UB and OOM in Refinement



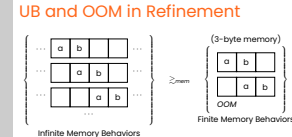
1. This refinement relation also takes into account undefined behavior and out-of-memory events

# UB and OOM in Refinement



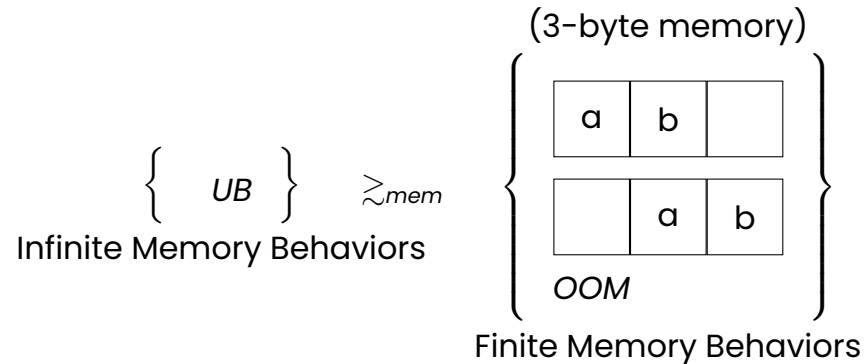
1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model  
 └ Two-Phase Memory Models  
 └ UB and OOM in Refinement



1. We always allow out-of-memory on the right hand side to allow finite programs to run out of memory instead of continuing execution

# UB and OOM in Refinement



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

### └ Two-Phase Memory Models

### └ UB and OOM in Refinement

## UB and OOM in Refinement



1. And if UB is on the left hand side we allow any behavior on the right hand side, allowing us to perform optimizations without worrying about ill-defined programs

# Integrating into Vellvm



## ■ Tricky issues:

- ▶ Casts between pointers and integers
- ▶ undef (non-determinism / uninitialized values)
- ▶ Structured data types
- ▶ Scale of the language

1980-01-01

- A Two-Phase Infinite/Finite Low-Level Memory Model
  - └ Two-Phase Memory Models
    - └ Integration into LLVM
      - └ Integrating into Vellvm

Integrating into Vellvm



## ■ Tricky issues:

- ▶ Casts between pointers and integers
- ▶ undef (non-determinism / uninitialized values)
- ▶ Structured data types
- ▶ Scale of the language

1. As previously noted, we have integrated this two-phase memory into Vellvm to stress-test it under a large and realistic semantics
2. This has highlighted some tricky situations around...
3. Casts
4. undef, the non-deterministic uninitialized values in LLVM
5. Structured data types
6. And just the overall scale of the language

# PTOI: Truncation Aliasing

```
char *p = malloc(4);  
*p = 0;  
p2 = (char*)(int32_t)p;
```

- Truncation introduces aliasing!
- p2 may alias something other than p.

1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

- └ Two-Phase Memory Models
  - └ Integration into LLVM
    - └ PTOI: Truncation Aliasing

PTOI: Truncation Aliasing

```
char *p = malloc(4);  
*p = 0;  
p2 = (char*)(int32_t)p;  
■ Truncation introduces aliasing!  
■ p2 may alias something other than p.
```

1. So one tricky issue is pointer / integer casts.
2. In an infinite memory model pointers are unbounded, so casting them to finite integers can result in truncation
3. Which means the casted 'p2' may alias with an unexpected location in memory, instead of pointing to the same place as 'p'.
4. And again, unexpected aliasing can block optimizations like store forwarding, so how do we deal with this?



# Intptr and Out-of-Memory

- Need a type akin to `intptr_t` in languages like C.
- Different implementations in infinite / finite memory.
  - ▶ Avoids truncation of pointers on casts (aliasing!)
  - ▶ No such type exists in LLVM IR, though 😞.

1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

- └ Two-Phase Memory Models
  - └ Integration into LLVM
    - └ Intptr and Out-of-Memory

### Intptr and Out-of-Memory

- Need a type akin to `intptr_t` in languages like C.
- Different implementations in infinite / finite memory.
  - ▶ Avoids truncation of pointers on casts (aliasing!)
  - ▶ No such type exists in LLVM IR, though 😞.

1. The solution is quite simple.
2. We just need an integer type with the same cardinality as pointers, which we'll call the `intptr` type
3. Such a type even exists in C already, but we'll have to add such a type to LLVM
4. In the infinite model `intptr`s will be big ints, and in the finite model they can be, for instance, bounded 64-bit integers if we have a 64-bit address space.

# Intptr and Out-of-Memory

- Need a type akin to `intptr_t` in languages like C.
- Different implementations in infinite / finite memory.
  - ▶ Avoids truncation of pointers on casts (aliasing!)
  - ▶ No such type exists in LLVM IR, though 😞.

Model	Casts?	Dead Alloca?	Aliasing?
Finite	✓	✗	✗
Infinite	✗	✓	✓

1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

- └ Two-Phase Memory Models
  - └ Integration into LLVM
    - └ Intptr and Out-of-Memory

Intptr and Out-of-Memory

- Need a type akin to `intptr_t` in languages like C.
- Different implementations in infinite / finite memory.
  - ▶ Avoids truncation of pointers on casts (aliasing!)
  - ▶ No such type exists in LLVM IR, though 😞.

Model	Casts?	Dead Alloca?	Aliasing?
Finite	✓	✗	✗
Infinite	✗	✓	✓

1. This allows us to patch up the issues with casts...

# Intptr and Out-of-Memory

- Need a type akin to `intptr_t` in languages like C.
- Different implementations in infinite / finite memory.
  - ▶ Avoids truncation of pointers on casts (aliasing!)
  - ▶ No such type exists in LLVM IR, though 😞.

Model	Casts?	Dead Alloca?	Aliasing?
Finite	✓	✗	✗
Infinite	✓	✓	✓

1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

- └ Two-Phase Memory Models
  - └ Integration into LLVM
    - └ Intptr and Out-of-Memory

### Intptr and Out-of-Memory

- Need a type akin to `intptr_t` in languages like C.
- Different implementations in infinite / finite memory.
  - ▶ Avoids truncation of pointers on casts (aliasing!)
  - ▶ No such type exists in LLVM IR, though 😞.

Model	Casts?	Dead Alloca?	Aliasing?
Finite	✓	✗	✗
Infinite	✓	✓	✓

1. Easing aliasing concerns and allowing us to perform optimizations even in the presence of these casts

# Careful Translating Intptrs!

```
intptr_t i = 1;
while (0 < i) { ++i; printf("%zd\n", i); }
do_evil();
```

1980-01-01

- A Two-Phase Infinite/Finite Low-Level Memory Model
  - Two-Phase Memory Models
    - Integration into LLVM
      - Careful Translating Intptrs!

Careful Translating Intptrs!

```
intptr_t i = 1;
while (0 < i) { ++i; printf("%zd\n", i); }
do_evil();
```

1. Because the implementation of intptr differs in the infinite and finite models we have to be careful when translating infinite programs to finite programs
2. In the infinite world this program counts up indefinitely, but if the finite intptr value is allowed to overflow the loop would terminate instead, allowing the program to do evil new behaviors

# Careful Translating Intptrs!

```
intptr_t i = 1;
while (0 < i) { ++i; printf("%zd\n", i); }
do_evil();
```

- Need bounds checks on intptr arithmetic.

1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

- └ Two-Phase Memory Models
  - └ Integration into LLVM
    - └ Careful Translating Intptrs!

## Careful Translating Intptrs!

```
intptr_t i = 1;
while (0 < i) { ++i; printf("%zd\n", i); }
do_evil();
```

- Need bounds checks on intptr arithmetic.

1. So a correct translation of this program to the finite world will need to add bounds checks on intptr arithmetic

# Careful Translating Intptrs!

```
intptr_t i = 1;
while (0 < i) { ++i; printf("%zd\n", i); }
do_evil();
```

- Need bounds checks on intptr arithmetic.
- This should have minimal performance impact.

1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

- └ Two-Phase Memory Models
  - └ Integration into LLVM
    - └ Careful Translating Intptrs!

### Careful Translating Intptrs!

```
intptr_t i = 1;
while (0 < i) { ++i; printf("%zd\n", i); }
do_evil();
```

- Need bounds checks on intptr arithmetic.
- This should have minimal performance impact.

1. There are more details in the paper, but we believe that this will have a minimal impact on performance as these checks can be eliminated in most cases where pointer to integer casts are used in real programs

# Infinite / Finite Program Refinement

```
printf("Start\n");  
printf("%d\n", (int) malloc(10));  
printf("End\n");
```

1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

- └ Two-Phase Memory Models
  - └ Integration into LLVM
    - └ Infinite / Finite Program Refinement

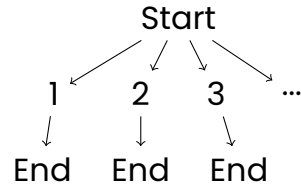
## Infinite / Finite Program Refinement

```
printf("Start\n");  
printf("%d\n", (int) malloc(10));  
printf("End\n");
```

1. We essentially just lift the refinement relations on memory operations to the level of the language

# Infinite / Finite Program Refinement

```
printf("Start\n");  
printf("%d\n", (int) malloc(10));  
printf("End\n");
```



1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

### └ Two-Phase Memory Models

#### └ Integration into LLVM

#### └ Infinite / Finite Program Refinement

## Infinite / Finite Program Refinement

```
printf("Start\n");  
printf("%d\n", (int) malloc(10));  
printf("End\n");
```

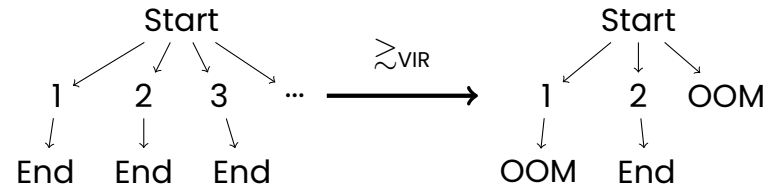


1. Our semantics is nondeterministic, so we end up with a tree of possible traces our program can have.
2. This shows the semantics of this program under the infinite memory model
3. It prints starts, and then prints the address allocated. There are an unbounded number of possible values that could be printed.
4. And finally the program prints End and exits



# Infinite / Finite Program Refinement

```
printf("Start\n");  
printf("%d\n", (int) malloc(10));  
printf("End\n");
```



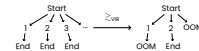
1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

- Two-Phase Memory Models
  - Integration into LLVM
    - Infinite / Finite Program Refinement

Infinite / Finite Program Refinement

```
printf("Start\n");  
printf("%d\n", (int) malloc(10));  
printf("End\n");
```

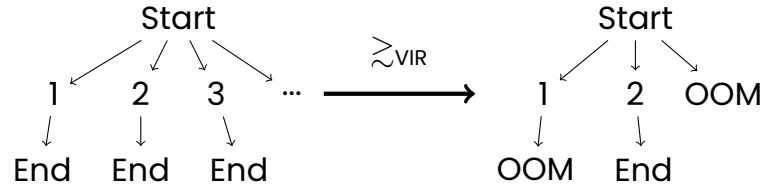


1. In the finite semantics the behavior of the program is identical except...

# Infinite / Finite Program Refinement

```
printf("Start\n");  
printf("%d\n", (int) malloc(10));  
printf("End\n");
```

- Finite programs agree...
  - ▶ But may run out of memory early!



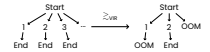
1980-01-01

- ## A Two-Phase Infinite/Finite Low-Level Memory Model
- └ Two-Phase Memory Models
    - └ Integration into LLVM
      - └ Infinite / Finite Program Refinement

Infinite / Finite Program Refinement

```
printf("Start\n");  
printf("%d\n", (int) malloc(10));  
printf("End\n");
```

- Finite programs agree...
  - ▶ But may run out of memory early!

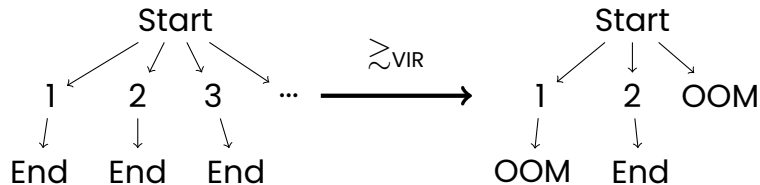


1. It may give up early with out-of-memory
2. And there are not as many possibilities for addresses returned by malloc

# Infinite / Finite Program Refinement

```
printf("Start\n");  
printf("%d\n", (int) malloc(10));  
printf("End\n");
```

- Finite programs agree...
  - ▶ But may run out of memory early!



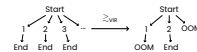
1980-01-01

- A Two-Phase Infinite/Finite Low-Level Memory Model
  - └ Two-Phase Memory Models
    - └ Integration into LLVM
      - └ Infinite / Finite Program Refinement

## Infinite / Finite Program Refinement

```
printf("Start\n");  
printf("%d\n", (int) malloc(10));  
printf("End\n");
```

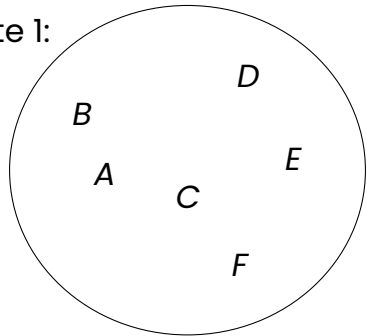
- Finite programs agree...
  - ▶ But may run out of memory early!



1. We have also proven that the finite version of the language agrees with the infinite version up to this refinement relation

# Infinite / Finite Refinement

Infinite 1:



1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

- └ Two-Phase Memory Models
  - └ Integration into LLVM
    - └ Infinite / Finite Refinement

Infinite / Finite Refinement

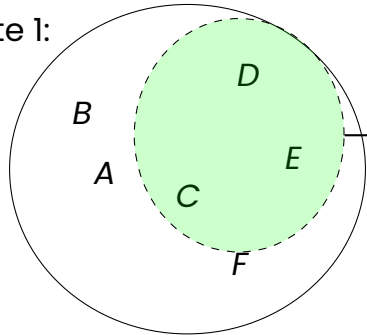
Infinite 1:



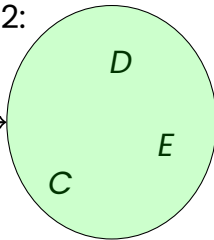
1. For some more intuition on the infinite to finite refinement relations, we can consider a set-based semantics. Each of these letters represents a different behavior.

# Infinite / Finite Refinement

Infinite 1:



Infinite 2:

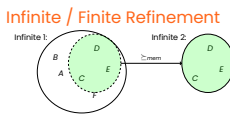


$\succeq_{mem}$

1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

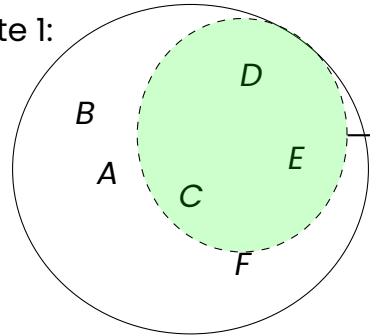
- └ Two-Phase Memory Models
- └ Integration into LLVM
- └ Infinite / Finite Refinement



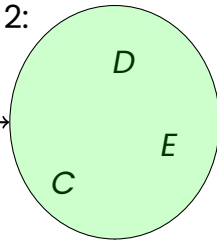
1. We can refine infinite memory programs to subsets of the original set of behaviors...

# Infinite / Finite Refinement

Infinite 1:

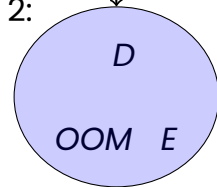


Infinite 2:



$\succeq_{mem}$

Finite 2:



$\approx$

1980-01-01

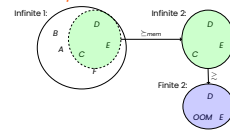
A Two-Phase Infinite/Finite Low-Level Memory Model

└ Two-Phase Memory Models

└ Integration into LLVM

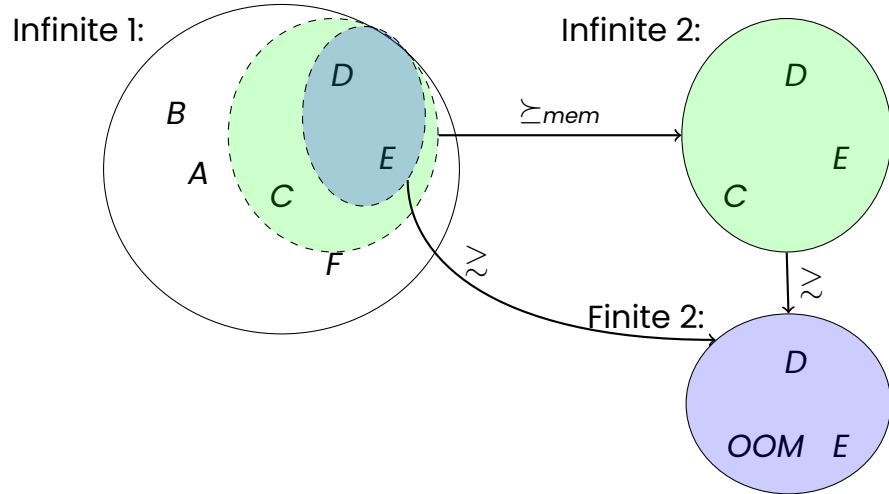
└ Infinite / Finite Refinement

Infinite / Finite Refinement



1. and then translate them to finite versions.

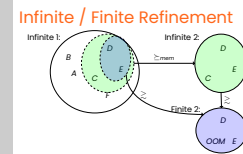
# Infinite / Finite Refinement



1980-01-01

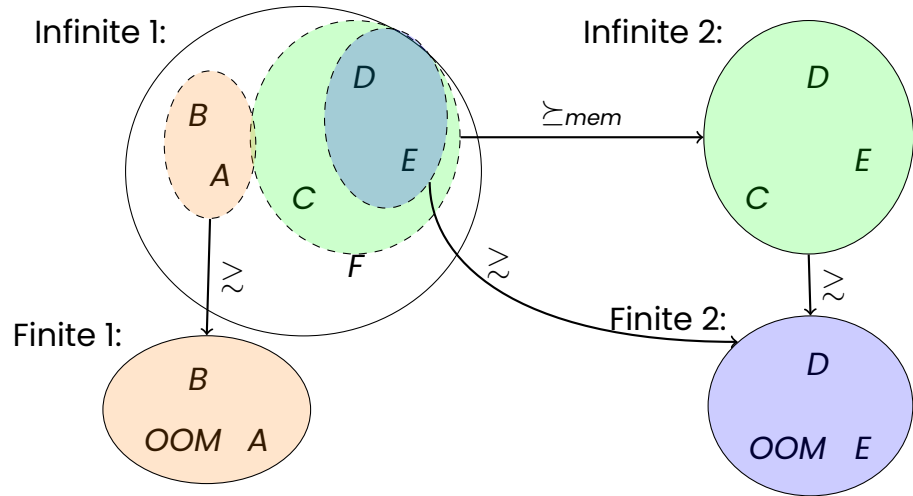
A Two-Phase Infinite/Finite Low-Level Memory Model

- Two-Phase Memory Models
- Integration into LLVM
- Infinite / Finite Refinement



1. And we can see the transitive property we get. The final set of behaviors of the finite program after all these steps will be a refinement of the original set of behaviors

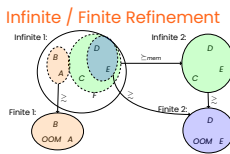
# Infinite / Finite Refinement



1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

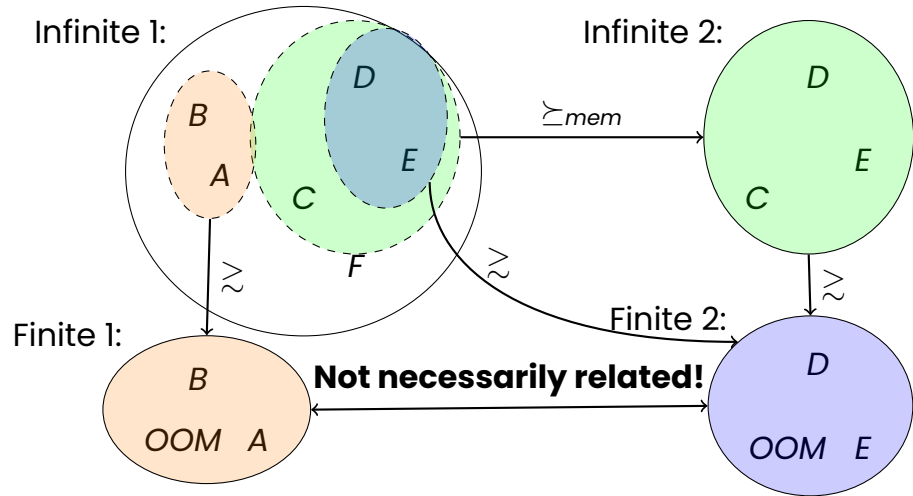
- Two-Phase Memory Models
- Integration into LLVM
- Infinite / Finite Refinement



- Note, though, that if we translate to a finite program and then perform program transformations, we may not necessarily reach the same set of behaviors.



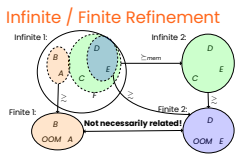
# Infinite / Finite Refinement



1980-01-01

A Two-Phase Infinite/Finite Low-Level Memory Model

- Two-Phase Memory Models
- Integration into LLVM
- Infinite / Finite Refinement



1. For instance, our program may run out of memory immediately if translated to the finite world right away, but we could perform optimizations to remove allocations in the infinite world, which could allow the finite program to make more progress.

# See the Paper For

- Better handling of UB (time travel)
- Supporting undef values (quantum entagled bytes)
- Verified executable implementation
- Deeper comparison to related work
- More details about Coq development



### A Two-Phase Infinite/Finite Low-Level Memory Model

Reconciling Integer-Pointer Casts, Finite Space, and undef at the LLVM IR Level of Abstraction

CALVIN BECK, University of Pennsylvania, USA  
IRENE YOON, Inria, France  
HANXI CHEN, University of Pennsylvania, USA  
YANNICK ZAKOWSKI, Inria, ENS de Lyon, CNRS, UCBL1, LIP, UMR 5668, France  
STEVE ZDANCEWIC, University of Pennsylvania, USA

This paper provides a novel approach to reconciling complex low-level memory model features, such as pointer-integer casts, with desired refinements that are needed to justify the correctness of program transformations. The idea is to use a “two-phase” memory model, one with an unbounded memory and corresponding unbounded integer type, and one with a finite memory; the connection between the two levels is made explicit by a notion of refinement that handles out-of-memory behaviors. This approach allows for more optimizations to be performed and establishes a clear boundary between the idealized semantics of a program and the implementation of that program on finite hardware.

The two-phase memory model has been incorporated into an LLVM IR semantics, demonstrating its utility in practice in the context of a low-level language with features like `undef` and `bitcast`. This yields infinite and finite memory versions of the language semantics that are proven to be in refinement with respect to out-of-memory behaviors. Each semantics is accompanied by a verified executable reference interpreter. The semantics justify optimizations, such as `dead-all-locs`-elimination, that were previously impossible or difficult to prove correct.

CCS Concepts: • **Theory of computation** → **Denotational semantics**; **Program verification**; **Program specifications**; • **Software and its engineering** → **Compilers**; **Semantics**.

Additional Key Words and Phrases: low-level memory model, integer-pointer casts, semantics, coq

**ACM Reference Format:**  
Calvin Beck, Irene Yoon, Hanxi Chen, Yannick Zakowski, and Steve Zdancewic. 2024. A Two-Phase Infinite/Finite Low-Level Memory Model: Reconciling Integer-Pointer Casts, Finite Space, and undef at the LLVM IR Level of Abstraction. *Proc. ACM Program. Lang.* 8, ICFP, Article 263 (August 2024), 29 pages. <https://doi.org/10.1145/3674652>

#### 1 Introduction

After 50 years the memory model for a programming language like C should be well understood! Unfortunately, memory models for low-level languages like C and LLVM IR are quite subtle and complex, especially when considered in the context of optimizations and program transformations [3–6, 13, 18–20, 22, 25, 26, 31, 32]. Why? These languages provide an abstract view of memory to justify a wide range of “high-level” optimizations—often pretending that available memory is unbounded and that allocations yield disjoint blocks, where a pointer to one allocated block can

Authors' Contact Information: Calvin Beck, University of Pennsylvania, Philadelphia, PA, USA, [hobbes@seas.upenn.edu](mailto:hobbes@seas.upenn.edu); Irene Yoon, Inria, Paris, France, [eyoon@inria.fr](mailto:eyoon@inria.fr); Hanxi Chen, University of Pennsylvania, Philadelphia, PA, USA, [hanxi@seas.upenn.edu](mailto:hanxi@seas.upenn.edu); Yannick Zakowski, Inria, ENS de Lyon, CNRS, UCBL1, LIP, UMR 5668, France, [yannick.zakowski@inria.fr](mailto:yannick.zakowski@inria.fr); Steve Zdancewic, University of Pennsylvania, Philadelphia, PA, USA, [stevez@cis.upenn.edu](mailto:stevez@cis.upenn.edu).



This work is licensed under a Creative Commons Attribution 4.0 International License.  
© 2024 Copyright held by the owner/author(s).  
ACM 2475-1421/2024/8-ART263  
<https://doi.org/10.1145/3674652>

1980-01-01


## A Two-Phase Infinite/Finite Low-Level Memory Model

└ See the Paper

└ See the Paper For

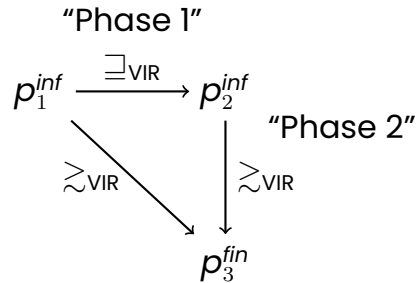
### See the Paper For

- Better handling of UB (time travel)
- Supporting undef values (quantum entagled bytes)
- Verified executable implementation
- Deeper comparison to related work
- More details about Coq development



# Conclusion

## Infinite/Finite Memory Model



1980-01-01

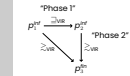
## A Two-Phase Infinite/Finite Low-Level Memory Model

└ See the Paper

└ Conclusion

### Conclusion

Infinite/Finite Memory Model

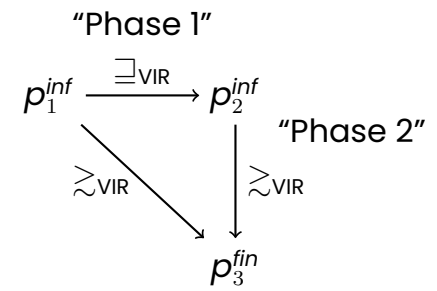


1. So, in conclusion we have implemented infinite and finite memory models and integrated these into an LLVM semantics
2. We have verified the connection between infinite and finite languages.

# Conclusion

Infinite/Finite Memory Model

Verified Optimizations



```
char *p = malloc(4);
p[0] = 0;
p = (char *) (int32_t) p;
char *q = malloc(2);
q[f(0)] = 3;
printf("%d\n", (int) p[0]);
```

→

```
printf("0\n");
```

1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

└ See the Paper

└ Conclusion

1. We are able to verify program transformations and optimization passes

### Conclusion

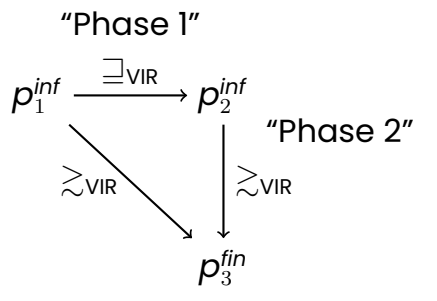
Infinite/Finite Memory Model

### Verified Optimizations

```
char *p = malloc(4);
p[0] = 0;
p = (char *) (int32_t) p;
char *q = malloc(2);
q[f(0)] = 3;
printf("%d\n", (int) p[0]);
```

→

```
printf("0\n");
```



```
char *p = malloc(4);
p[0] = 0;
p = (char*)(int32_t)p;
char *q = malloc(2);
q[f(0)] = 3;
printf("%d\n", (int)p[0]);
```

→ printf("0\n");

Model	Casts?	Dead Alloca?	Aliasing?
Finite	✓	✗	✗
Infinite	✗ → ✓	✓	✓

1. And our infinite memory model can support optimizations in the presence of pointer to integer casts

"Phase 1"

$p_1^{inf} \xrightarrow{\sqsubseteq_{VIR}} p_2^{inf}$

"Phase 2"

$p_1^{inf} \xrightarrow{\approx_{VIR}} p_3^{fin}$

$p_2^{inf} \xrightarrow{\approx_{VIR}} p_3^{fin}$

```






char *p = malloc(4);
p[0] = 0;
p = (char *) (int32_t) p;
char *q = malloc(2);
q[f(0)] = 3;
printf("%d\n", (int) p[0]);
    
```

Model	Casts?	Dead Alloca?	Aliasing?
Finite	✓	✗	✗
Infinite	✗ → ✓	✓	✓

1. Finally, artifacts are available and you can find our work on GitHub!
2. Thank you, are there any questions?

# References I

[TODO: armsvg, C image, x86 image]

-  **RISC-V Foundation.** URL: <https://commons.wikimedia.org/wiki/File:RISC-V-logo.svg>.
-  **Rust Foundation.** URL: <https://github.com/rust-lang/rust-artwork/blob/master/logo/rust-logo-blk.svg>.
-  **Haskell.** URL: [https://commons.wikimedia.org/wiki/File:Logo\\_of\\_the\\_Haskell\\_programming\\_language.svg](https://commons.wikimedia.org/wiki/File:Logo_of_the_Haskell_programming_language.svg).
-  **Caleb Stanford.** URL: <https://calebstanford.com/2019/01/15/coq-vector-image/>.
-  **Kenrick Turner.** URL: <https://www.flickr.com/photos/kenrickturner/11083895853>.

1980-01-01

## A Two-Phase Infinite/Finite Low-Level Memory Model

### References

#### References I

- [TODO: armsvg, C image, x86 image]
-  **RISC-V Foundation.** URL: <https://commons.wikimedia.org/wiki/File:RISC-V-logo.svg>.
  -  **Rust Foundation.** URL: <https://github.com/rust-lang/rust-artwork/blob/master/logo/rust-logo-blk.svg>.
  -  **Haskell.** URL: [https://commons.wikimedia.org/wiki/File:Logo\\_of\\_the\\_Haskell\\_programming\\_language.svg](https://commons.wikimedia.org/wiki/File:Logo_of_the_Haskell_programming_language.svg).
  -  **Caleb Stanford.** URL: <https://calebstanford.com/2019/01/15/coq-vector-image/>.
  -  **Kenrick Turner.** URL: <https://www.flickr.com/photos/kenrickturner/11083895853>.