A Formal C Memory Model
Supporting Integer-Pointer Casts

Jeehoon Kang (Seoul National University)
Chung-Kil Hur (Seoul National University)
William Mansky (UPenn)
Dmitri Garbuzov (UPenn)
Steve Zdancewic (UPenn)
Viktor Vafeiadis (MPI-SWS)
Motivation

• Integer-pointer cast is an **important feature** of C.
  + used in Linux kernel, Java HotSpot VM

• Pointers being integers **invalidates** optimizations.
  + e.g. constant propagation

• Want to support **integer-pointer casts & optimizations**
Integer-Pointer Casts: Importance in Practice

• Example 1: Pointers as hash keys

```c
void hash_put(void* key, Data value);
Data hash_get(void* key);
```

• Example 2: Pointer compression in Java HotSpot VM

```c
int32_t compress(void*); // 64bit -> 32bit
void* decompress(int32_t); // 32bit -> 64bit
```
Identifying Pointers with Integers: Invalidates Constant Propagation

- Anyone can access any address.

```c
extern void g();

c char f() {
    char a = '0';
    g();
    g();
    return a; // -> return '0'
}
```
Identifying Pointers with Integers: Invalidates Constant Propagation

Anyone can access any address.

```c
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> return '0'
}
```
Identifying Pointers with Integers: Invalidates Constant Propagation

- Anyone can access any address.

```c
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *(p) = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> return '0'
}
```

<table>
<thead>
<tr>
<th>expr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;a</td>
<td>0x120</td>
</tr>
</tbody>
</table>

0x100  
0x120  
'0'    
...
Identifying Pointers with Integers: Invalidates Constant Propagation

• Anyone can access any address.

```c
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a;  // -> return '0'
}
```
Identifying Pointers with Integers: Invalidates Constant Propagation

- Anyone can access any address.

```c
void g() {
  char b = '2';
  char* p = &b + 0x20;
  *p = '1';
}

char f() {
  char a = '0';
  g();
  return a; // -> return '0'
}
```

<table>
<thead>
<tr>
<th>expr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;a</td>
<td>0x120</td>
</tr>
<tr>
<td>&amp;b</td>
<td>0x100</td>
</tr>
</tbody>
</table>
Identifying Pointers with Integers: Validates Constant Propagation

• Anyone can access any address.

```c
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> return '0'
}
```

<table>
<thead>
<tr>
<th>:expr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;a</td>
<td>0x120</td>
</tr>
<tr>
<td>&amp;b</td>
<td>0x100</td>
</tr>
<tr>
<td>&amp;b+0x20</td>
<td>0x120</td>
</tr>
</tbody>
</table>
Identifying Pointers with Integers: Invalidates Constant Propagation

- Anyone can access any address.

```c
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}
char f() {
    char a = '0';
    g();
    return a;  // -> return '0'
}
```

### Validations:

- `&a` is at `0x120`
- `&b` is at `0x100`
- `&b+0x20` is also at `0x120`

<table>
<thead>
<tr>
<th>expr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;a</code></td>
<td>0x120</td>
</tr>
<tr>
<td><code>&amp;b</code></td>
<td>0x100</td>
</tr>
<tr>
<td><code>&amp;b+0x20</code></td>
<td>0x120</td>
</tr>
</tbody>
</table>
Identifying Pointers with Integers: Invalidates Constant Propagation

- Anyone can access any address.

```c
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> return '0'
}
```

<table>
<thead>
<tr>
<th>Expr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;a</td>
<td>0x120</td>
</tr>
<tr>
<td>&amp;b</td>
<td>0x100</td>
</tr>
<tr>
<td>&amp;b + 0x20</td>
<td>0x120</td>
</tr>
</tbody>
</table>
Goal of Memory Model

• To validate common optimizations by disallowing problematic memory accesses

• To allow integer-pointer casts
Outline

Supporting Int-Ptr Casts

Validating Optimizations
Outline

Invalidates Most Opt.

Validating Optimizations

Supporting Int-Ptr Casts

Naive
Validating Optimizations

Supporting Int-Ptr Casts

Invalidate Most Opt.

Naive → C11

Outline
Invalidates Most Opt.

Supporting Int-Ptr Casts

Naive → C11

Validating Optimizations

CompCert
Outline

- Supporting Int-Ptr Casts
- Validating Optimizations
- Invalidates Most Opt.
- Naive → C11
- Ours
- CompCert

Outline:
- C11
- Ours
- Naive
- Invalidates Most Opt.

Diagram:
- Supporting Int-Ptr Casts
- Validating Optimizations
- Naive → C11
- Ours
- CompCert
Validating Optimizations

Supporting Int-Ptr Casts

Naive → C11

Ours

CompCert

Invalidates Most Opt.
C11 Model: High-Level Idea

- Integers & pointers are tagged with permission.

<table>
<thead>
<tr>
<th>Integers</th>
<th>Pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>0x120</td>
</tr>
<tr>
<td>0x100</td>
<td>0x100</td>
</tr>
</tbody>
</table>
C11 Model:

High-Level Idea

- Integers & pointers are tagged with permission.

<table>
<thead>
<tr>
<th>Integers</th>
<th>Pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120 {&amp;a}</td>
<td>0x120 {&amp;a}</td>
</tr>
<tr>
<td>0x100 {&amp;b}</td>
<td>0x100 {&amp;b}</td>
</tr>
</tbody>
</table>
• Integers & pointers are tagged with permission.

Integers

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>{&amp;a}</td>
</tr>
<tr>
<td>0x100</td>
<td>{&amp;b}</td>
</tr>
</tbody>
</table>

Pointers

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>{&amp;a}</td>
</tr>
<tr>
<td>0x100</td>
<td>{&amp;b}</td>
</tr>
</tbody>
</table>
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}

cchar f() {
    char a = '0';
    g();
    return a; // -> '0'
}
```c
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
```

<table>
<thead>
<tr>
<th>expr</th>
<th>value {perm.}</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;a</td>
<td>0x120 {&amp;a}</td>
</tr>
<tr>
<td>&amp;b</td>
<td>0x100</td>
</tr>
<tr>
<td>&amp;b+0x20</td>
<td>0x120</td>
</tr>
</tbody>
</table>
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
void g() {
    char b = '2';
    char* p = &b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
g();
    return a;  // -> '0'
}
C11 Model’s Problems (1/2):
Too Complex Semantics

- **Integers also need to carry permission.**
  Since integer-pointer casts should preserve permission.

- **Operations need to properly calculate permission.**

  ```
  int y = x - x;  // -> int y = 0;
  ```

<table>
<thead>
<tr>
<th>expr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0x100 {&amp;a}</td>
</tr>
<tr>
<td>x - x</td>
<td>0x000 {?}</td>
</tr>
</tbody>
</table>
  ```
• **Integers also need to carry permission.**
  Since integer-pointer casts should preserve permission.

• **Operations need to properly calculate permission.**

```c
int y = x - x;  // -> int y = 0;
```

<table>
<thead>
<tr>
<th>expr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>0x100 {&amp;a}</td>
</tr>
<tr>
<td>(x - x)</td>
<td>0x000 {&amp;a} (\rightarrow) 0x000 {}</td>
</tr>
</tbody>
</table>
```
Integers also need to carry permission. Since integer-pointer casts should preserve permission.

Operations need to properly calculate permission.

\[
\text{int } y = x - x; \quad // \quad \rightarrow \quad \text{int } y = 0;
\]

<table>
<thead>
<tr>
<th>expr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>(0x100 {&amp;a})</td>
</tr>
<tr>
<td>(x - x)</td>
<td>(0x000 {})  (\rightarrow) (0x000 {})</td>
</tr>
</tbody>
</table>
• **Integers also need to carry permission.**
  Since integer-pointer casts should preserve permission.

• **Operations need to properly calculate permission.**

  \[
  \text{int } y = x - x; \quad // \quad \rightarrow \text{int } y = 0;
  \]

<table>
<thead>
<tr>
<th>expr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0x100 {&amp;a}</td>
</tr>
<tr>
<td>x - x</td>
<td>0x000 {}</td>
</tr>
<tr>
<td>x XOR x</td>
<td>0x000 {}</td>
</tr>
<tr>
<td>2 * x</td>
<td>0x200 {?}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

...
C11 Model’s Problems (2/2): Invalidates Some Optimizations

- A useful code motion is not allowed.

```
int a, b;
...
if (a != b) {
    a = b;
}
```

```
int a, b;
...
if (a != b) {
}
a = b;
```
C11 Model’s Problems (2/2): Invalidates Some Optimizations

- A useful code motion is not allowed.

```
int a, b;
...
if (a != b) {
    a = b;
}
```

<table>
<thead>
<tr>
<th>expr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0x100 {&amp;x} -&gt; 0x100 {}</td>
</tr>
<tr>
<td>b</td>
<td>0x100 {}</td>
</tr>
</tbody>
</table>
C11 Model’s Problems (2/2): Invalidates Some Optimizations

- We found a real GCC bug.

```c
void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t)&x;
    int* p = (int*)xi;
    *p = 1;
    printf("%d\n", x); } // prints 1
```

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752
C11 Model’s Problems (2/2): Invalidates Some Optimizations

- We found a real GCC bug.

```c
void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t) &x;
    int* p = (int*) xi;
    *p = 1;
    printf("%d\n", x); } // prints 1
```

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752

- Integer type for pointers

- We found a real GCC bug.

```c
void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t) &x;
    int* p = (int*) xi;
    *p = 1;
    printf("%d\n", x); } // prints 1
```

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752
void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t) &x;
    int* p = (int*) xi;
    *p = 1;
    printf("%d\n", x); } // prints 1

We found a real GCC bug.

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752
void main() {
  int x = 0;
  uintptr_t xi = (uintptr_t)&x;
  int* p = (int*) xi;
  *p = 1;
  printf("%d\n", x); } // prints 1

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752
C11 Model’s Problems (2/2): Invalidates Some Optimizations

- We found a real GCC bug.

```c
void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t) &x;
    uintptr_t i;
    for (i = 0; i < xi; ++i) {}  // prints 0
    if (xi != i) {
        printf("unreachable\n");
        xi = i;
    }
    int* p = (int*) xi;
    *p = 1;
    printf("%d\n", x); } // prints 0
```

[https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752](https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752)
C11 Model’s Problems (2/2): Invalidates Some Optimizations

- We found a real GCC bug.

```c
void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t)&x;
    uintptr_t i;
    for (i = 0; i < xi; ++i) {} // code motion
    if (xi != i) {
        printf("unreachable\n");
    }
    xi = i;
    int* p = (int*) xi;
    *p = 1;
    printf("%d\n", x); } // prints 0
```

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752
We found a real GCC bug.

```c
void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t) &x;
    uintptr_t i;
    for (i = 0; i < xi; ++i) {}  // DEAD CODE
    if (xi != i) {
        printf("unreachable\n");
    }
    xi = i; // code motion
    int* p = (int*) xi;
    *p = 1;
    printf("%d\n", x); // prints 0
} // prints 0
```

Integer type for pointers

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752
We found a real GCC bug.

```c
void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t) &x;
    uintptr_t i;
    for (i = 0; i < xi; ++i) {}
    if (xi != i) {
        printf("unreachable\n");
    }
    xi = i; // code motion
    int* p = (int*) xi;
    *p = 1;
    printf("%d\n", 0); } // constant propagation x -> 0
```

Integer type for pointers

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752
Invalidates Most Opt.

Invalidates Some Opt.

Complex Semantics

Supporting Int-Ptr Casts

Naive

C11

Ours

Validating Optimizations

CompCert

12 / 25
Invalidate Most Opt.

Supporting Int-Ptr Casts

Complex Semantics Invalidates Some Opt.

Validating Optimizations

Naive

C11

Ours

CompCert

CompCert
CompCert Model: High-Level Idea

- Pointers are **different from** integers.

<table>
<thead>
<tr>
<th>Integers</th>
<th>Pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>
void g() {
    char b[2]={'2','3'};
    char* p = b + 0x20;
    *p = '1';
}
char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
void g() {
    char b[2]={'2','3'};
    char* p = b + 0x20;
    *p = '1';
}
char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
void g() {
    char b[2]={'2','3'};
    char* p = b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
void g() {
    char b[2]={'2','3'};
    char* p = b + 0x20;
    *p = '1';
}
char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
void g() {
    char b[2]={'2','3'};
    char* p = b + 0x20;
    *p = '1';
}

char f() {
    char a = '0';
g();
    return a; // -> '0'
}
void g() {
    char b[2]={'2','3'};
    char* p = b + 0x20;
    *p = '1';
}
char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
Invalidates Most Opt.

Complex Semantics
Invalidates Some Opt.

Supporting Int-Ptr Casts

Naive

C11

Ours

No Int-Ptr Casts

Validating Optimizations

CompCert

C11

CompCert

Validating Optimizations

Ours

Invalidates Most Opt.

Supporting Int-Ptr Casts

Validating Optimizations

Ours

No Int-Ptr Casts

CompCert

C11

Naive
Our Model: High-Level Idea

- Pointers become integers only when casted.

<table>
<thead>
<tr>
<th>Integers</th>
<th>Pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>
• Pointers become integers only when casted.
Our Model: 
Realizes at Casting to Integer

```c
char a[2] = {'0','1'};
char b[3] = {'2','3','4'};
uintptr_t bi = (uintptr_t) b;
char* p1 = (char*) 0x101;
char* p2 = (char*) 0x120;
```
Our Model: Realizes at Casting to Integer

```c
char a[2] = {'0', '1'};
char b[3] = {'2', '3', '4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;
```

```
<p>| '1' |</p>
<table>
<thead>
<tr>
<th>'0'</th>
</tr>
</thead>
</table>
```

a

\( (l_1, 0) \)
Our Model: Realizes at Casting to Integer

```c
char a[2] = {'0','1'};
char b[3] = {'2','3','4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;
```
Our Model:
Realizes at Casting to Integer

```c
char a[2] = {'0','1'};
char b[3] = {'2','3','4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;
```
char a[2] = {'0', '1'};
char b[3] = {'2', '3', '4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;

Our Model: Realizes at Casting to Integer
char a[2] = {'0', '1'};
char b[3] = {'2', '3', '4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;

Our Model: Realizes at Casting to Integer

\[
\begin{array}{c}
\text{char } a[2] = \{'0', '1'\}; \\
\text{char } b[3] = \{'2', '3', '4'\}; \\
\text{bi} = \text{(uintptr\_t)} b; \\
p1 = \text{(char\_*) } 0x101; \\
p2 = \text{(char\_*) } 0x120;
\end{array}
\]
Our Model: Realizes at Casting to Integer

```c
char a[2] = {'0', '1'};
char b[3] = {'2', '3', '4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;
```

Table:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(l₁, 0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>(l₂, 0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bi</td>
<td>0x100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(char*) 0x101</td>
<td>(l₂, 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(char*) 0x120</td>
<td>FAIL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Our Model:

- Realizes blocks when casting to integer
- Casts back to corresponding blocks

```c
char a[2] = {'0', '1'};
char b[3] = {'2', '3', '4'};
uintptr_t bi = (uintptr_t) b;
char* p1 = (char*) 0x101;
char* p2 = (char*) 0x120;
```
void g() {
    char b[2]={'2','3'};
    char* p = b + 0x20;
    *p = '1';
}
char f() {
    char a = '0';
g();
    return a; // -> '0'
}
Benefits of Our Model (1/6): Still Validates Optimizations

```
void g() {
    char b[2]={'2','3'};
    //char* p = b + 0x20;
    bi = (uintptr_t) b;
    p = (char*) (bi+0x20);
    *p = '1';
}
char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
```
Benefits of Our Model (1/6): Still Validates Optimizations

```c
void g() {
    char b[2]= {'2','3'};
    char* p = b + 0x20;
    bi = (uintptr_t) b;
    p = (char*) (bi+0x20);
    *p = '1';
}

char f() {
    char a = '0';
    g();
    return a; // -> '0'
}
```
void g() {
    char b[2]= {'2','3'};
    //char* p = b + 0x20;
    bi = (uintptr_t) b;
    p = (char*) (bi+0x20);
    *p = '1';
}
char f() {
    char a = '0';
g();
return a; // -> '0'
}
• Pointer-to-integer casts always succeed.

• Integer operations on casted pointers always succeed.
Benefits of Our Model (3/6): Simple Semantics

- Integer values are just integers w/o permission.
- Integer operations are just integer operations.

• Integer optimizations are allowed.

```java
int a = x - x; // -> int a = 0;
```

• The useful code motion is allowed.

```java
int a, b;
...
if (a != b) {
    a = b;
}
```
• Just treat “casted pointers as escaped”.

```c
void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t)&x;
    uintptr_t i;
    for (i = 0; i < xi; ++i) {}
    if (xi != i) {
        printf("unreachable\n");
    }
    xi = i; // code motion
    int* p = (int*) xi;
    *p = 1;
    printf("%d\n", x); } // prints 1
```
Benefits of Our Model (5/6): Easily Applicable to Compilers

- Just treat "casted pointers as escaped".

```c
void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t) &x;
    uintptr_t i;
    for (i = 0; i < xi; ++i) {}  // treated as escaped
    if (xi != i) {
        printf("unreachable\n");
    }
    xi = i; // code motion
    int* p = (int*) xi;
    *p = 1;
    printf("%d\n", x);  // prints 1
}
```
Benefits of Our Model (5/6): Easily Applicable to Compilers

- Just treat "casted pointers as escaped".

```c
void main() {
    int x = 0;
    uintptr_t xi = (uintptr_t) &x;
    uintptr_t i;
    for (i = 0; i < xi; ++i) {} // casted pointers as escaped
    if (xi != i) {
        printf("unreachable\n");
    }
    xi = i; // code motion
    int* p = (int*) xi;
    *p = 1;
    printf("%d\n", x); } // prints 1
}
```
Benefits of Our Model (6/6):
Little Performance Penalty

- **Insignificant**: performance degradation due to “casted pointers as escaped”
  
  + In practice, addresses casted to integers are **global addresses**.

  + Compilers **already** treat **global addresses as escaped**.

Invalidates Most Opt.

Supporting Int-Ptr Casts

Fully Supports Int-Ptr Casts
Validates Most Opt.
Simple Semantics
Easily Applicable to Compilers
Little Performance Penalty

Supporting Int-Ptr Casts

Ours

No Int-Ptr Casts

CompCert

Validating Optimizations

Naive

C11

Ours

CompCert
What Else is in the Paper?

• Formal definition of our memory model
• Reasoning principles for compiler verification
• Verification of other optimization examples
  + Dead code elim., dead allocation elim.,
    arithmetic optimizations, alias analysis, etc.
• Comparison with other possible models

Fully formalized in Coq