An Intermediate Language To Formally Justify Memory Access Reordering

Ralf Jung
Advisors: Prof. Dr. Sebastian Hack, Sigurd Schneider

Bachelor Thesis Talk

Department of Computer Science - Universität des Saarlandes

May 16th, 2013
Outline

1. Introduction
2. Memory Model
3. Type System
4. Limitations, Conclusion
Intermediate languages

- Abstract away from unnecessary details of source language
- Discard precise order of computations
- Program stored as directed graph
- Preserve relevant information: Which operation is performed on which operand
- All linearisations respecting the order are equivalent
- Optimisations can choose from all linearisations

\[
x = 3 + 5; \\
y = 2 \times 4; \\
z = x - y;
\]
Intermediate languages

- Abstract away from unnecessary details of source language
- Discard precise order of computations
- Program stored as directed graph
- Preserve relevant information: Which operation is performed on which operand
- All linearisations respecting the order are equivalent
- Optimisations can choose from all linearisations

\[
\begin{align*}
x &= 3 + 5; \\
y &= 2 \times 4; \\
z &= x - y;
\end{align*}
\]
Intermediate languages

- Abstract away from unnecessary details of source language
- Discard precise order of computations
- Program stored as directed graph
- Preserve relevant information: Which operation is performed on which operand
- All linearisations respecting the order are equivalent
- Optimisations can choose from all linearisations

\[
\begin{align*}
a &= 2 \times 4; \\
b &= 3 + 5; \\
c &= b - a;
\end{align*}
\]
Memory operations

This does not work well for memory operations:

\[
\begin{align*}
\text{store}(a, v); & \quad \text{store}(b, w); \\
\text{store}(b, w); & \quad \text{store}(a, v);
\end{align*}
\]

Without further knowledge, their order must be preserved.

However, if \(a\) and \(b\) never take the same value, the two programs are equivalent.
Memory operations

This does not work well for memory operations:

\[
\text{store}(a, v); \quad \text{store}(b, w);
\]

\[
\text{store}(b, w); \quad \text{store}(a, v);
\]

Without further knowledge, their order must be preserved.

However, if \(a\) and \(b\) never take the same value, the two programs are equivalent.
Memory operations

This does not work well for memory operations:

\[
\text{store}(a, v) ; \\
\text{store}(b, w) ; \\
\text{store}(b, w) ; \\
\text{store}(a, v) ;
\]

Without further knowledge, their order must be preserved.

However, if \( a \) and \( b \) never take the same value, the two programs are equivalent.
Contribution: IL/M

- Intermediate language based on IL/F which can express absence of dependencies between memory operations
- No memory safety
- Type system supporting proofs of correctness for transformations which de-linearise memory accesses
  - Based on knowledge about pointer values (alias information)
- Formal semantics and proof of correctness

Expected benefits

- Simplify analyses and transformations
- More opportunities for optimisation
Outline

1. Introduction
2. Memory Model
3. Type System
4. Limitations, Conclusion
Functional memory model

- Memory is an explicit object
- Immutable mapping of locations to values
- Memory operations manipulate memories similar to how integers are manipulated by arithmetic operations
- Effect of memory operations is completely described by operands

```
let m' = store m a v in
let m'' = store m' b w in
...  
```
Functional memory model

```
store(a, v);  
store(b, w);  
let m' = store m a v in
let m'' = store m' b w in
...  
```
Functional memory model

```
let m1' = store m1 a v in
let m2' = store m2 b w in
...```
Functional stores can express programs which cannot be directly simulated on real machines:

\[
\begin{align*}
\text{let } m' &= \text{store } m \ a \ v \\
\text{let } m'' &= \text{store } m \ a \ w \\
\text{let } x &= \text{load } m' \ a
\end{align*}
\]

Naïve translation: ignore memory argument

Resulting program is incorrect

Definition

A program permitting a naïve translation can be realised.
Realisability

- Functional stores can express programs which cannot be directly simulated on real machines:

\[
\text{let } m' = \text{store } m \text{ a} v \text{ in} \\
\text{let } m'' = \text{store } m \text{ a} w \text{ in} \\
\text{let } x = \text{load } m' \text{ a} \text{ in} \\
\ldots
\]

- Naïve translation: ignore memory argument

- Resulting program is incorrect

Definition

A program permitting a naïve translation can be \textit{realised}.
Outline

1 Introduction
2 Memory Model
3 Type System
4 Limitations, Conclusion
**Approach**

- Type system for memory objects
- Based on alias information
- Well-typed programs are realisable, i.e., they can easily be translated to machine code
- If a program is well-typed after de-linearising memory operations, it is semantically equivalent to the original program
let m' = store m a v in
let m'' = store m' b w in
let x = load m'' c in ...
\{a \not\approx b\} \text{ let } m' = \text{store } m \ a \ v \ \text{ in }
\{a \not\approx b\} \text{ let } m'' = \text{store } m' \ b \ w \ \text{ in }
\{a \not\approx b\} \text{ let } x = \text{load } m'' \ c \ \text{ in } \ldots
Example

\{a \not\equiv b\} \ let m1, m2 = \text{split} \ m \ \{a\} \ \text{in}
\{a \not\equiv b\} \ let m1' = \text{store} \ m1 \ a \ v \ \text{in}
\{a \not\equiv b\} \ let m2' = \text{store} \ m2 \ b \ w \ \text{in}
\{a \not\equiv b\} \ let m' = \text{merge} \ m1' \ m2' \ \text{in}
\{a \not\equiv b\} \ let x = \text{load} \ m' \ c \ \text{in} \ \ldots
Memory types

- Keep track of variables **split** to a separate memory
  - These variables form the *focus*
  - The memories containing these variables are called *focus memories*
  - Type: Set of variables used to create it
- All the other locations remain in the *panorama memory*
  - Real-world alias information is incomplete, so there can be locations we know nothing about
  - There is always exactly one panorama memory
  - Type: $\top$
- Memories may not be used again after *store, split, merge* to keep available memories pairwise disjoint
Example

<table>
<thead>
<tr>
<th>m</th>
<th>focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊤</td>
<td>{}</td>
</tr>
</tbody>
</table>

{a \not\approx b} \text{ let } m_1, m_2 = \text{split } m \{a\} \text{ in }

<table>
<thead>
<tr>
<th>m1</th>
<th>m2</th>
<th>focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a}</td>
<td>⊤</td>
<td>{a}</td>
</tr>
</tbody>
</table>

{a \not\approx b} \text{ let } m'_1 = \text{store } m_1 a v \text{ in }

{a \not\approx b} \text{ let } m'_2 = \text{store } m_2 b w \text{ in }

<table>
<thead>
<tr>
<th>m1'</th>
<th>m2'</th>
<th>focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a}</td>
<td>⊤</td>
<td>{a}</td>
</tr>
</tbody>
</table>

{a \not\approx b} \text{ let } m' = \text{merge } m'_1 m'_2 \text{ in }

<table>
<thead>
<tr>
<th>m'</th>
<th>focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊤</td>
<td>{}</td>
</tr>
</tbody>
</table>

{a \not\approx b} \text{ let } x = \text{load } m' c \text{ in } \ldots
Example

\[
\begin{array}{|c|c|}
\hline
m & \text{focus} \\
\hline
\top & \{\} \\
\hline
\end{array}
\]

\{a \not\approx b\} \text{ let } m_1, m_2 = \text{split } m \{a\} \text{ in }

\[
\begin{array}{|c|c|c|}
\hline
m_1 & m_2 & \text{focus} \\
\hline
\{a\} & \top & \{a\} \\
\hline
\end{array}
\]

\{a \not\approx b\} \text{ let } m_1' = \text{store } m_1 \text{ a v in }
\{a \not\approx b\} \text{ let } m_2' = \text{store } m_2 \text{ b w in }

\[
\begin{array}{|c|c|c|}
\hline
m_1' & m_2' & \text{focus} \\
\hline
\{a\} & \top & \{a\} \\
\hline
\end{array}
\]

\{a \not\approx b\} \text{ let } m' = \text{merge } m_1' \text{ m_2' in }

\[
\begin{array}{|c|c|}
\hline
m' & \text{focus} \\
\hline
\top & \{\} \\
\hline
\end{array}
\]

\{a \not\approx b\} \text{ let } x = \text{load } m' \text{ c in } \ldots
Example

\[\{a \not\in b\} \text{ let } m_1, m_2 = \text{split } m \{a\} \text{ in}\]

<table>
<thead>
<tr>
<th>m1</th>
<th>m2</th>
<th>focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a}</td>
<td>⊤</td>
<td>{a}</td>
</tr>
</tbody>
</table>

\[\{a \not\in b\} \text{ let } m'_1 = \text{store } m_1 a v \text{ in}\]

\[\{a \not\in b\} \text{ let } m'_2 = \text{store } m_2 b w \text{ in}\]

<table>
<thead>
<tr>
<th>m1'</th>
<th>m2'</th>
<th>focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a}</td>
<td>⊤</td>
<td>{a}</td>
</tr>
</tbody>
</table>

\[\{a \not\in b\} \text{ let } m' = \text{merge } m'_1 m'_2 \text{ in}\]

<table>
<thead>
<tr>
<th>m'</th>
<th>focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊤</td>
<td>{}</td>
</tr>
</tbody>
</table>

\[\{a \not\in b\} \text{ let } x = \text{load } m' c \text{ in } \ldots\]
Example

<table>
<thead>
<tr>
<th>m</th>
<th>focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊤</td>
<td>{}</td>
</tr>
</tbody>
</table>

{a \not\equiv b} let m1, m2 = split m {a} in

<table>
<thead>
<tr>
<th>m1</th>
<th>m2</th>
<th>focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a}</td>
<td>⊤</td>
<td>{a}</td>
</tr>
</tbody>
</table>

{a \not\equiv b} let m1’ = store m1 a v in

{a \not\equiv b} let m2’ = store m2 b w in

<table>
<thead>
<tr>
<th>m1’</th>
<th>m2’</th>
<th>focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a}</td>
<td>⊤</td>
<td>{a}</td>
</tr>
</tbody>
</table>

{a \not\equiv b} let m’ = merge m1’ m2’ in

<table>
<thead>
<tr>
<th>m’</th>
<th>focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊤</td>
<td>{}</td>
</tr>
</tbody>
</table>

{a \not\equiv b} let x = load m’ c in ...

Diagram:

```
  m
     |       |
     | split |
     |       |
  store  store
     |       |
     | merge |
     |       |
  load
```
Restrictions on memory accesses to provide semantic guarantees

- **store** and **load** require proofs that the affected location is *accessible* in the given memory
  - Accessibility is defined based on the type of the memory
  - To access focus memory: Prove equality to one variable from memory domain
  - To access panorama memory: Prove inequality to all focus variables
  - Proofs must be derived from alias annotation
  - Only if the (in)equality can be statically derived, the access is well-typed
Example: Accessibility

\{a \not\in\ b\} \ let \ m1, m2 = \texttt{split} \ m \ \{a\} \ \text{in}
\{a \not\in\ b\} \ let \ m1' = \texttt{store} \ m1 \ a \ v \ \text{in}

Access to a in memory of type \{a\}:
\ a \preceq a \ \text{trivially holds}

\{a \not\in\ b\} \ let \ m2' = \texttt{store} \ m2 \ b \ w \ \text{in}

Access to b in panorama, focus is \{a\}:
\ a \not\in b \ \text{holds by annotation}

\{a \not\in b\} \ let \ m' = \texttt{merge} \ m1' \ m2' \ \text{in}
\{a \not\in b\} \ let \ x = \texttt{load} \ m' \ c \ \text{in} \ldots

Access to c in panorama, focus is {}:
Nothing to show
Example: Accessibility

\{a \not\approx b\} \text{ let } m_1, m_2 = \text{split } m \{a\} \text{ in }
\begin{aligned}
{a \not\approx b} \text{ let } m_1' = \text{store } m_1 a v \text{ in }
\end{aligned}

Access to \(a\) in memory of type \(\{a\}\):  
\(a \approx_\Delta a\) trivially holds

\begin{aligned}
{a \not\approx b} \text{ let } m_2' = \text{store } m_2 b w \text{ in }
\end{aligned}

Access to \(b\) in panorama, focus is \(\{a\}\):  
\(a \not\approx b\) holds by annotation

\begin{aligned}
{a \not\approx b} \text{ let } m' = \text{merge } m_1' m_2' \text{ in }
\end{aligned}

\begin{aligned}
{a \not\approx b} \text{ let } x = \text{load } m' c \text{ in } ...
\end{aligned}

Access to \(c\) in panorama, focus is \(\{\}\):  
Nothing to show
Example: Accessibility

\{a \not\approx b\} \text{ let } m1, m2 = \text{split} \ m \ \{a\} \ \text{in}

\{a \not\approx b\} \text{ let } m1’ = \text{store} \ m1 \ a \ v \ \text{in}

\begin{itemize}
  \item Access to \ a \ \text{in memory of type} \ \{a\}:
    \begin{align*}
      a \approx a \ \text{trivially holds}
    \end{align*}
\end{itemize}

\{a \not\approx b\} \text{ let } m2’ = \text{store} \ m2 \ b \ w \ \text{in}

\begin{itemize}
  \item Access to \ b \ \text{in panorama, focus is} \ \{a\}:
    \begin{align*}
      a \not\approx b \ \text{holds by annotation}
    \end{align*}
\end{itemize}

\{a \not\approx b\} \text{ let } m’ = \text{merge} \ m1’ \ m2’ \ \text{in}

\{a \not\approx b\} \text{ let } x = \text{load} \ m’ \ c \ \text{in} \ldots

\begin{itemize}
  \item Access to \ c \ \text{in panorama, focus is} \ {}:
    \begin{align*}
      \text{Nothing to show}
    \end{align*}
\end{itemize}
Normalisation

- Remove all `split` and `merge` from the program
- Replace all memory variables by some fixed `m`

```ml
let m1, m2 = split m {a} in
let m1' = store m1 a v in
let m2' = store m2 b w in
let m' = merge m1' m2' in
let x = load m' c in ...
```

```ml
let m = store m a v in
let m = store m b w in
let x = load m c in ...
```
Normalisation

- Remove all `split` and `merge` from the program
- Replace all memory variables by some fixed $m$

```ocaml
let m1, m2 = split m {a} in
let m1' = store m1 a v in
let m2' = store m2 b w in
let m' = merge m1' m2' in
let x = load m' c in ...  
```

```ocaml
let m = store m a v in
let m = store m b w in
let x = load m c in ...  
```
Normalisation

- Remove all \texttt{split} and \texttt{merge} from the program
- Replace all memory variables by some fixed $m$

\begin{verbatim}
let m1, m2 = split m {a} in 
let m1' = store m1 a v in 
let m2' = store m2 b w in 
let m' = merge m1' m2' in 
let x = load m' c in ...
\end{verbatim}

\begin{verbatim}
let m = store m a v in 
let m = store m b w in 
let x = load m c in ...
\end{verbatim}
Normalisation

- Remove all `split` and `merge` from the program
- Replace all memory variables by some fixed `m`

```
let m1, m2 = split m {a} in
let m1' = store m1 a v in
let m2' = store m2 b w in
let m' = merge m1' m2' in
let x = load m' c in ...
```

```
let m' = store m a v in
let m'' = store m' b w in
let x = load m'' c in ...
```
Core Theorem

Normalisation preserves semantics

Every well-typed program is semantically equivalent to its normalisation.

- Every well-typed program is realisable
- Proof of correctness for transformations which change memory dependencies, but not normalisation of a program
Normalisation preserves semantics

Every well-typed program is semantically equivalent to its normalisation.

- Every well-typed program is realisable
- Proof of correctness for transformations which change memory dependencies, but not normalisation of a program
Core Theorem

Normalisation preserves semantics

Every well-typed program is semantically equivalent to its normalisation.

- Every well-typed program is realisable
- Proof of correctness for transformations which change memory dependencies, but not normalisation of a program
Outline

1 Introduction

2 Memory Model

3 Type System

4 Limitations, Conclusion
Limitations

- Functions can only take one memory variable as argument: the panorama memory
  - Need to merge all memories before calling a function
- No support for compound data types
- No support for pointer arithmetic
## Summary

### Contribution

- Intermediate language with explicit memory dependencies
- Reordering of independent memory operations inherent to the representation
  - Proof of correctness based on embedded alias information
- Realisability on a real machine guaranteed by typing relation
- Memory safety in source language *not* required
- Everything formalised and proven in Coq
Thank you very much for your attention!

Questions?

The thesis is available online at http://ralfj.de/cs/bachelor.pdf
IL/M Semantics

Three environments: Variables, Closures, Memories

let $x = e$ in $s$  
variable binding

let $m = \text{store } m a x$ in $s$  
memory store

let $x = \text{load } m a$ in $s$  
memory load

let $m = \text{free } m a$ in $s$  
memory deallocation

let $m, m = \text{split } m A$ in $s$  
splitting memory

let $m = \text{merge } m m$ in $s$  
merging memories
Three environments: Variables, Closures, Memories

\[
\text{fun } f \overline{x} m = s \text{ in } t \\
f \overline{x} m \\
x
\]

- function definition
- function application
- function return

No memory variables in closures
Three environments: Variables, Closures, Memories

\[ \text{if } x \text{ then } s \text{ else } t \quad \text{conditional} \]
IL/M Semantics

- Three environments: Variables, Closures, Memories

  \( \text{let } m, a = \text{alloc in } s \) memory allocation

- Needs to select a fresh address to keep memories disjoint
- Maintain set of allocated addresses in state
Separation Logic

- Separation Logic makes assertions about memory contents
- Central idea: *Separating conjunction* $\phi \ast \psi$ states that $\phi$ and $\psi$ apply to *disjoint parts* of the memory
- Seems to fit well to the concept of split
- However, the separating conjunction abstracts away from how the memory is split
  - split would be non-deterministic if the separating conjunction were used as specification
Separation Logic

- Separation Logic makes assertions about memory contents
- Central idea: *Separating conjunction* $\phi \ast \psi$ states that $\phi$ and $\psi$ apply to *disjoint parts* of the memory
- Seems to fit well to the concept of *split*
- However, the separating conjunction abstracts away from how the memory is split
  - *split* would be non-deterministic if the separating conjunction were used as specification
Assume a and b should be split into their own memory
- We don’t know whether they are equal or not
Which separation-logical formula describes this memory?
- a \not\rightarrow \neg denotes a memory which contains exactly a (singleton memory)
- Memory with a and b: (a \not\rightarrow \neg \ast b \not\rightarrow \neg) \lor (a \not\rightarrow \neg \land b \not\rightarrow \neg)

Combinatorial explosion!
Assume a and b should be split into their own memory
   - We don’t know whether they are equal or not
Which separation-logical formula describes this memory?
a \leftrightarrow \neg \cdot \ b \leftrightarrow \neg \cdot \ denotes a memory which contains exactly a (singleton memory)
Memory with a and b: \((a \leftrightarrow \neg \cdot \ b \leftrightarrow \neg) \lor (a \leftrightarrow \neg \cdot \ b \leftrightarrow \neg)\)

Combinatorial explosion!
Separation Logic: Representing alias information

- Fundamental structural difference
- Separation Logic is designed for a top-down view

\[ \phi \star \psi : \]

- Alias information is very local

- Enumerating all these local memories adds overhead for no visible benefit
**Separation Logic: Representing alias information**

- Fundamental structural difference
- Separation Logic is designed for a top-down view

\[
\phi \ast (\psi_1 \ast \psi_2):
\]

- Alias information is very local

- Enumerating all these local memories adds overhead for no visible benefit
Separation Logic: Representing alias information

- Fundamental structural difference
- Separation Logic is designed for a top-down view

\[ \phi \ast (\psi_1 \ast \psi_2) : \]

- Alias information is very local

- Enumerating all these local memories adds overhead for no visible benefit