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

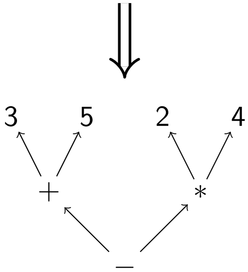
- Abstract away from unnecessary details of source language
- Discard precise order of computations
- Program stored as directed graph
- Preserve relevant information: Which operation is performed on which operand
- All linearisations respecting the order are equivalent
- Optimisations can choose from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

```
x = 3+5;
y = 2*4;
z = x-y;
```

Intermediate languages

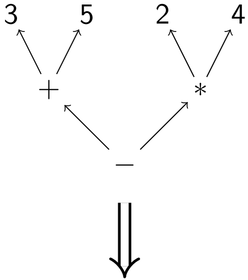
- Abstract away from unnecessary details of source language
- Discard precise order of computations
- Program stored as directed graph
- Preserve relevant information: Which operation is performed on which operand
- All linearisations respecting the order are equivalent
- Optimisations can choose from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

```
x = 3+5;
y = 2*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




```
a = 2*4;
b = 3+5;
c = b-a;
```

Memory operations

This does not work well for memory operations:

```
store(a, v);          store(b, w);
store(b, w);          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.

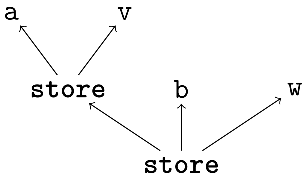
Memory operations

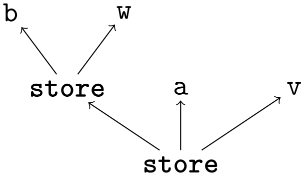
This does not work well for memory operations:

```
store(a, v);
store(b, w);
```

```
store(b, w);
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.

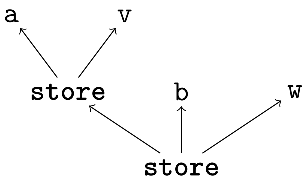
Memory operations

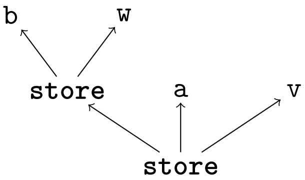
This does not work well for memory operations:

```
store(a, v);
store(b, w);
```

```
store(b, w);
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

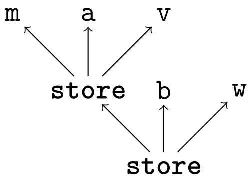
2 Memory Model

3 Type System

4 Limitations, Conclusion

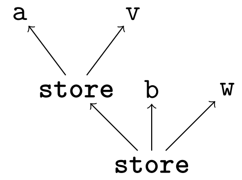
Functional memory model

- 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

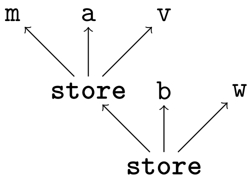



```
let m' = store m a v in
let m'' = store m' b w in
...
```

Functional memory model

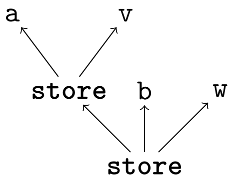



```
store(a, v);
store(b, w);
```

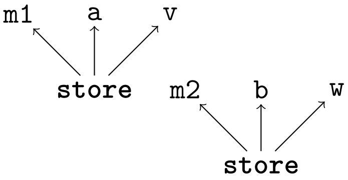



```
let m' = store m a v in
let m'' = store m' b w in
...
```

Functional memory model



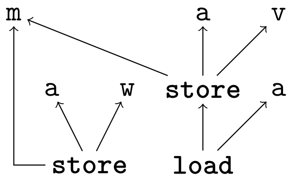

```
store(a, v);
store(b, w);
```




```
let m1' = store m1 a v in
let m2' = store m2 b w in
...
```

Realisability

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




```
let m' = store m a v in
let m'' = store m a w in
let x = load m' a in
...
```

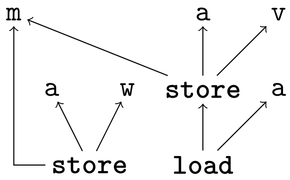
- Naïve translation: ignore memory argument
- Resulting program is incorrect

Definition

A program permitting a naïve translation can be *realised*.

Realisability

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




```
let m' = store m a v in
let m'' = store m a w in
let x = load m' a in
...
```

- Naïve translation: ignore memory argument
- Resulting program is incorrect

Definition

A program permitting a naïve translation can be *realised*.

Outline

1 Introduction

2 Memory Model

3 Type System

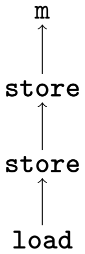
4 Limitations, Conclusion

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

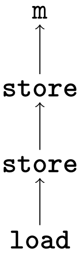
Example

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



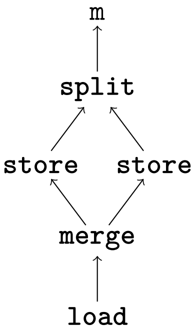
Example

```
{a ≇ b} let m' = store m a v in
{a ≇ b} let m'' = store m' b w in
{a ≇ b} let x = load m'' c in ...
```



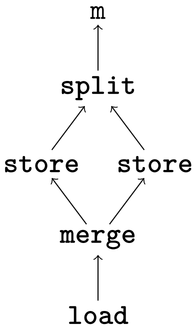
Example

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



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

m	focus
⊤	{}

```
{a ≢ b} let m1, m2 = split m {a} in
```

m1	m2	focus
{a}	⊤	{a}

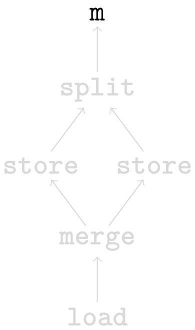
```
{a ≢ b} let m1' = store m1 a v in
{a ≢ b} let m2' = store m2 b w in
```

m1'	m2'	focus
{a}	⊤	{a}

```
{a ≢ b} let m' = merge m1' m2' in
```

m'	focus
⊤	{}

```
{a ≢ b} let x = load m' c in ...
```



Example

m	focus
⊤	{}

```
{a ≉ b} let m1, m2 = split m {a} in
```

m1	m2	focus
{a}	⊤	{a}

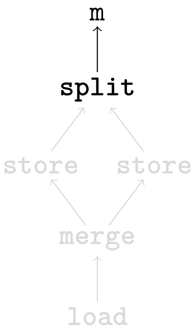
```
{a ≉ b} let m1' = store m1 a v in
{a ≉ b} let m2' = store m2 b w in
```

m1'	m2'	focus
{a}	⊤	{a}

```
{a ≉ b} let m' = merge m1' m2' in
```

m'	focus
⊤	{}

```
{a ≉ b} let x = load m' c in ...
```



Example

m	focus
⊤	{}

```
{a ≇ b} let m1, m2 = split m {a} in
```

m1	m2	focus
{a}	⊤	{a}

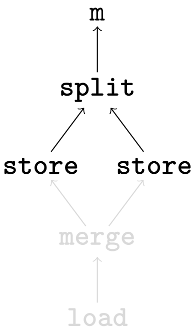
```
{a ≇ b} let m1' = store m1 a v in
{a ≇ b} let m2' = store m2 b w in
```

m1'	m2'	focus
{a}	⊤	{a}

```
{a ≇ b} let m' = merge m1' m2' in
```

m'	focus
⊤	{}

```
{a ≇ b} let x = load m' c in ...
```



Example

m	focus
⊤	{}

```
{a ≇ b} let m1, m2 = split m {a} in
```

m1	m2	focus
{a}	⊤	{a}

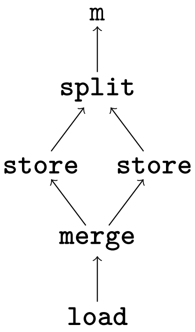
```
{a ≇ b} let m1' = store m1 a v in
{a ≇ b} let m2' = store m2 b w in
```

m1'	m2'	focus
{a}	⊤	{a}

```
{a ≇ b} let m' = merge m1' m2' in
```

m'	focus
⊤	{}

```
{a ≇ b} let x = load m' c in ...
```



Accessibility

- 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 ≇ b} let m1, m2 = split m {a} in
{a ≇ b} let m1' = store m1 a v in
```

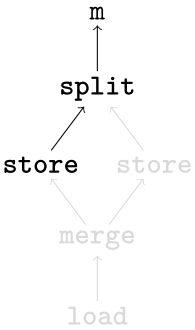
Access to a in memory of type {a}:
a ≅ a trivially holds

```
{a ≇ b} let m2' = store m2 b w in
```

Access to b in panorama, focus is {a}:
a ≇ b holds by annotation

```
{a ≇ b} let m' = merge m1' m2' in
{a ≇ b} let x = load m' c in ...
```

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



Example: Accessibility

```
{a ≇ b} let m1, m2 = split m {a} in
{a ≇ b} let m1' = store m1 a v in
```

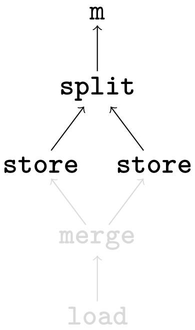
Access to a in memory of type $\{a\}$:
$a \cong a$ trivially holds

```
{a ≇ b} let m2' = store m2 b w in
```

Access to b in panorama, focus is $\{a\}$:
$a \ncong b$ holds by annotation

```
{a ≇ b} let m' = merge m1' m2' in
{a ≇ b} let x = load m' c in ...
```

Access to c in panorama, focus is $\{\}$:
Nothing to show



Example: Accessibility

```
{a ≇ b} let m1, m2 = split m {a} in
{a ≇ b} let m1' = store m1 a v in
```

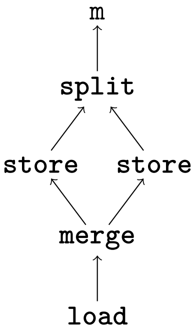
Access to a in memory of type {a}:
a ≅ a trivially holds

```
{a ≇ b} let m2' = store m2 b w in
```

Access to b in panorama, focus is {a}:
a ≇ b holds by annotation

```
{a ≇ b} let m' = merge m1' m2' in
{a ≇ b} let x = load m' c in ...
```

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



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 ...
```

$$\Downarrow$$

```
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*

```
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 ...
```

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 ...
```

$$\Downarrow$$

```
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*

```
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

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

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

$\texttt{let}\ x = e\ \texttt{in}\ s$	variable binding
$\texttt{let}\ m = \texttt{store}\ m\ a\ x\ \texttt{in}\ s$	memory store
$\texttt{let}\ x = \texttt{load}\ m\ a\ \texttt{in}\ s$	memory load
$\texttt{let}\ m = \texttt{free}\ m\ a\ \texttt{in}\ s$	memory deallocation
$\texttt{let}\ m, m = \texttt{split}\ m\ A\ \texttt{in}\ s$	splitting memory
$\texttt{let}\ m = \texttt{merge}\ m\ m\ \texttt{in}\ s$	merging memories

IL/M Semantics

- Three environments: Variables, Closures, Memories

$\texttt{fun}\ f\ \overline{x}\ m = s\ \texttt{in}\ t$	function definition
$f\ \overline{x}\ m$	function application
x	function return

- No memory variables in closures

IL/M Semantics

- Three environments: Variables, Closures, Memories

if x **then** s **else** t	conditional

IL/M Semantics

- Three environments: Variables, Closures, Memories

$\mathtt{let}\ m, a = \mathtt{alloc}\ \mathtt{in}\ s$ 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* $\phi * \psi$ states that ϕ and ψ 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* $\phi * \psi$ states that ϕ and ψ 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: Describing memory domains

- 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?
- $\mathtt{a} \mapsto -$ denotes a memory which contains exactly a (singleton memory)
- Memory with a and b: $(\mathtt{a} \mapsto - * \mathtt{b} \mapsto -) \vee (\mathtt{a} \mapsto - \wedge \mathtt{b} \mapsto -)$

Combinatorial explosion!

Separation Logic: Describing memory domains

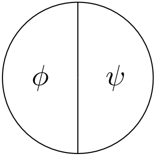
- Assume `a` and `b` should be split into their own memory
  - We don't know whether they are equal or not
- Which separation-logical formula describes this memory?
- $\mathtt{a} \mapsto -$ denotes a memory which contains exactly `a` (singleton memory)
- Memory with `a` and `b`: $(\mathtt{a} \mapsto - * \mathtt{b} \mapsto -) \vee (\mathtt{a} \mapsto - \wedge \mathtt{b} \mapsto -)$

Combinatorial explosion!

Separation Logic: Representing alias information

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

$\phi * \psi$:



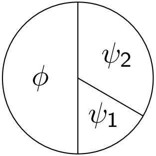
- Alias information is very local

- 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

$\phi * (\psi_1 * \psi_2)$:



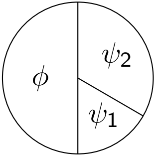
- Alias information is very local

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

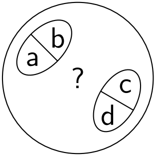
Separation Logic: Representing alias information

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

$\phi * (\psi_1 * \psi_2)$:



- Alias information is very local



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