An Introduction to MEMOIR

Tommy M'Michelen

Advanced Topics in Compilers
Overview

Bird’s Eye View

MEMOIR is a
Overview

Bird’s Eye View

MEMOIR is a

Compiler Intermediate Representation
Overview

Bird’s Eye View

MEMOIR is a Compiler Intermediate Representation for Data Collections and Objects
Overview

Bird’s Eye View

MEMOIR is a Compiler Intermediate Representation for Data Collections and Objects in an SSA Form.
What is a Data Collection?
Outline

What is a Data Collection?

What is SSA?
Outline

What is a Data Collection?

What is SSA?

How can I analyze it?
Overview

Outline

What is a Data Collection?

What is SSA?

How can I analyze it?

How can I transform it?
Data Collection

A logical organization of data
Data Collections

Examples

List
Data Collections

Examples

List

Dense Array

Sparse Array
Data Collections

Examples

List

Dense Array

Sparse Array

Set
Data Collections

Examples

- List
- Set
- Dense Array
- Sparse Array
- Map
Data Collections

Examples

- List
- Set
- Tree
- Graph

Dense Array

Sparse Array

Map
Data Collections

Examples

Sequential
- Dense Array
- Sparse Array

Associative
- Set
- Map

Dense Array
- List

Sparse Array
- Tree
- Graph
Data Collections

Examples

**Sequential**
- List
- Dense Array
- Sparse Array

**Associative**
- Set
- Map

**Tree**
\[ id \rightarrow [2 \times id] \]

**Graph**
\[ id \rightarrow \{id\} \]
Data Collections

Examples

Sequential
- List
- Dense Array
- Sparse Array

Associative
- Set
- Map
- Composition

Composition
- Tree
- Graph

Examples of Data Collections:
- List
- Set
- Map
- Tree
- Graph

Specialization:
- Dense Array
- Sparse Array

Composition:
- id → { id }
Data Collections

Representation

Index-Value Mapping

Index Space
Data Collections

Operations on the Index-Value Mapping

Index-Value Mapping

Index Space

write!
Operations on the *Index-Value Mapping*

Diagram: 
- **Index Space**
- **Index-Value Mapping**
- **swap!**
Data Collections

Operations on the *Index Space*

Index-Value Mapping

remove!
Operations on the Index Space

Index-Value Mapping

Data Collections
A language constraint, where each variable has a single definition in the static program.
Why SSA?

Referential Transparency
Replacing a subexpression with an equivalent one produces an equivalent expression.
Referential Transparency

She lives in Chicago

≈

She lives in the largest city in Illinois
Referential Opacity

‘Chicago’ contains seven letters

‘The largest city in Illinois’ contains seven letters
Benefits of Referential Transparency

A variable’s value is *independent of its position in the program*
Benefits of Referential Transparency

A variable’s value is independent of its position in the program

Information attached to the definition of a variable is true for all uses of the variable
So, what about data collections?

\[ \text{write!}(c, i, v) \]

**Semantics:** Following this operation, \( c[i] = v \)
So, what about data collections?

\[
\text{write!(c, i, v)}
\]

Semantics: Following this operation, \(c[i] = v\)

But, is \(\text{read}(c, i) \approx v\)?

*Not necessarily!* Depends on its position in the program.
So, what about data collections?

\[ \text{write!(c, i, v)} \]

\[ \text{read(c, i)} \]

_Yep!_ The use of \( c \) is dominated by the \texttt{write!} and is not dominated by any other \texttt{write!} to \( c \)
So, what about data collections?

\[
\text{write!}(c, i, v) \\
\text{write!}(c, i, w) \\
\text{read}(c, i)
\]

Nope! The use of \(c\) is dominated by the \text{write!}, but is dominated by another \text{write!} to \(c\)
So, what about data collections?

read\((c, i)\)

write!\((c, i, v)\)

*Nope!* The use of \(c\) is not dominated by the write!
So, how do we fix this? SSA Construction.

c' = write(c, i, v)
read(c', i)
So, what about MEMOIR?

Each operation on a collection produces a new collection (except for read)

$$\text{write!}(c, i, v) \rightarrow c' = \text{write}(c, i, v)$$
So, what about MEMOIR?

Each operation on a collection produces a new collection (except for read)

\[
\text{write!}(c, i, v) \quad \rightarrow \quad c' = \text{write}(c, i, v)
\]
\[
\text{swap!}(c, i, j) \quad \rightarrow \quad c' = \text{swap}(c, i, j)
\]
So, what about MEMOIR?

Each operation on a collection produces a new collection (except for read)

\[
\begin{align*}
\text{write!}(c, i, v) & \rightarrow c' = \text{write}(c, i, v) \\
\text{swap!}(c, i, j) & \rightarrow c' = \text{swap}(c, i, j) \\
\text{remove!}(c, i) & \rightarrow c' = \text{remove}(c, i) \\
\text{insert!}(c, i, v) & \rightarrow c' = \text{insert}(c, i, v)
\end{align*}
\]
SSA

So, what about MEMOIR?

Each operation on a collection produces a new collection (except for read)

\[
\begin{align*}
\text{write}!(c, i, v) & \rightarrow c' = \text{write}(c, i, v) \\
\text{swap}!(c, i, j) & \rightarrow c' = \text{swap}(c, i, j) \\
\text{remove}!(c, i) & \rightarrow c' = \text{remove}(c, i) \\
\text{insert}!(c, i, v) & \rightarrow c' = \text{insert}(c, i, v) \\
\text{read}(c, i) & \rightarrow v = \text{read}(c, i)
\end{align*}
\]
So, what about MEMOIR?

Other, useful query operations can be easily performed:

\[ n = \text{size}(c) \quad \triangleright \quad \text{# of elements in } c \]
\[ h = \text{has}(c, i) \quad \triangleright \quad \text{does } c \text{ have index } i? \]
\[ ks = \text{keys}(c) \quad \triangleright \quad \text{sequence of keys in } c \]
The MEMOIR Compiler
Step-by-step instructions are available

Writing a pass:

mcmichen.cc/memoir-docs/user/writing_a_pass

Writing a program:

mcmichen.cc/memoir-docs/user/writing_a_program
Conclusion

Additional Resources

Developer manual:

mcmichen.cc/memoir-docs

Doxygen:

mcmichen.cc/memoir-doxygen

The CGO’24 paper:

mcmichen.cc/files/MEMOIR_CGO_2024.pdf
Yippee!

Live Coding Time