Aggregate Update Problem for Multi-clocked Dataflow Languages

Hannes Kallwies Martin Leucker Torben Scheffel Malte Schmitz
Daniel Thoma

Institute for Software Engineering and Programming Languages,
University of Lübeck, Lübeck, Germany

International Symposium on Code Generation and Optimization, April 2022

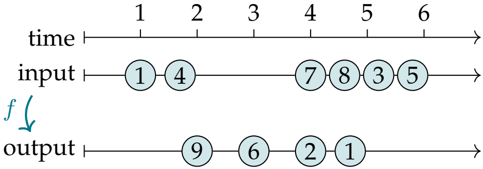
Dataflow Programming

Programming paradigm.

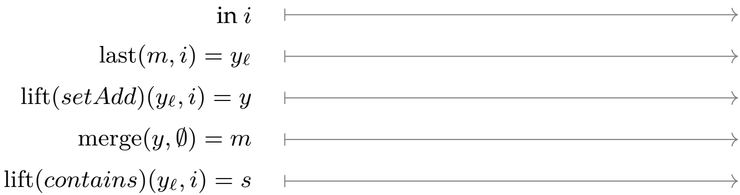
Basic concept: Data streams are combined with operators to generate output streams.

Popular dataflow languages:

- Lustre
- Lucid Synchrone
- SIGNAL
- Esterell
- LabView
- LOLA
- Striver
- TeSSLa



Running Example



$\text{in } i$

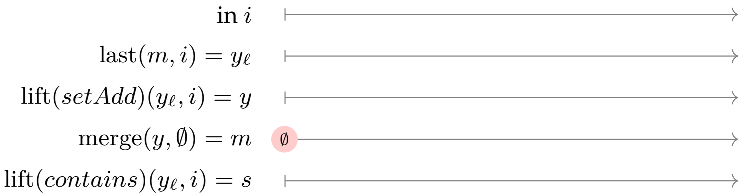
$\text{last}(m, i) = y_\ell$

$\text{lift}(\mathit{setAdd})(y_\ell, i) = y$

$\text{merge}(y, \emptyset) = m$

$\text{lift}(\mathit{contains})(y_\ell, i) = s$

Running Example



$\text{in } i$

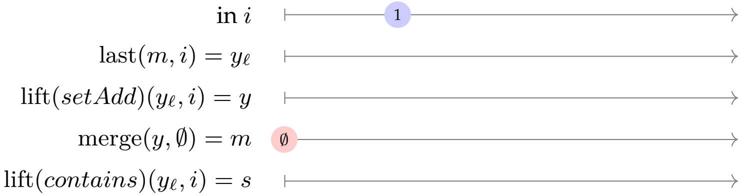
$\text{last}(m, i) = y_\ell$

$\text{lift}(\mathit{setAdd})(y_\ell, i) = y$

$\text{merge}(y, \emptyset) = m$

$\text{lift}(\mathit{contains})(y_\ell, i) = s$

Running Example



$\text{in } i$

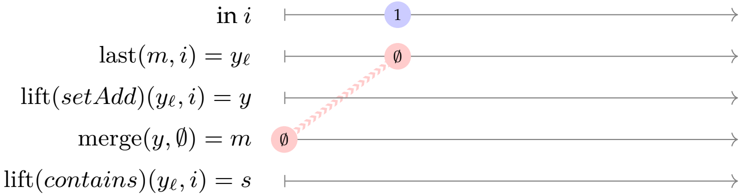
$\text{last}(m, i) = y_\ell$

$\text{lift}(\mathit{setAdd})(y_\ell, i) = y$

$\text{merge}(y, \emptyset) = m$

$\text{lift}(\mathit{contains})(y_\ell, i) = s$

Running Example



$\text{in } i$

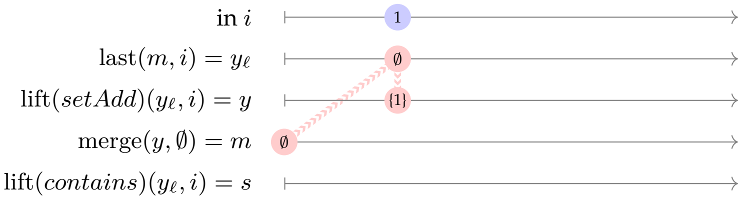
$\text{last}(m, i) = y_\ell$

$\text{lift}(\mathit{setAdd})(y_\ell, i) = y$

$\text{merge}(y, \emptyset) = m$

$\text{lift}(\mathit{contains})(y_\ell, i) = s$

Running Example



$\text{in } i$

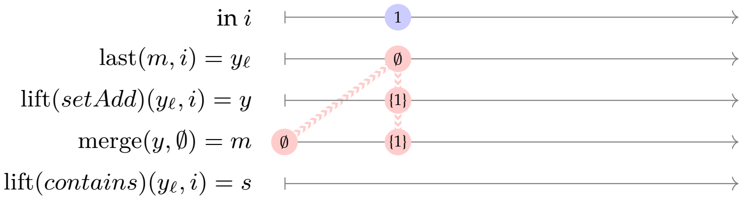
$\text{last}(m, i) = y_\ell$

$\text{lift}(setAdd)(y_\ell, i) = y$

$\text{merge}(y, \emptyset) = m$

$\text{lift}(contains)(y_\ell, i) = s$

Running Example



in i

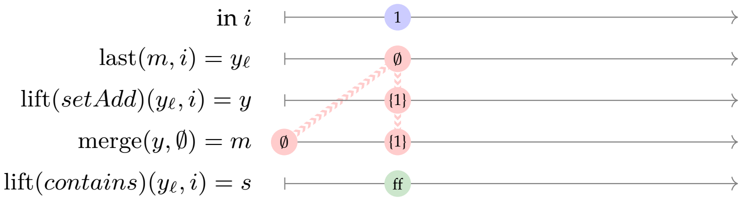
$\text{last}(m, i) = y_\ell$

$\text{lift}(setAdd)(y_\ell, i) = y$

$\text{merge}(y, \emptyset) = m$

$\text{lift}(contains)(y_\ell, i) = s$

Running Example



in i

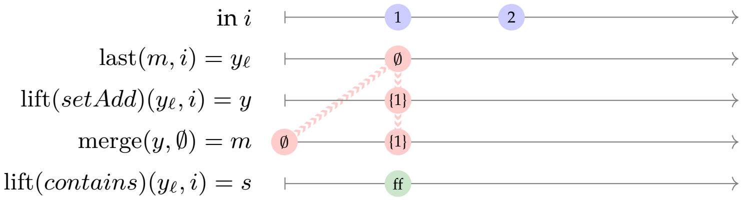
$\text{last}(m, i) = y_\ell$

$\text{lift}(setAdd)(y_\ell, i) = y$

$\text{merge}(y, \emptyset) = m$

$\text{lift}(contains)(y_\ell, i) = s$

Running Example



in i

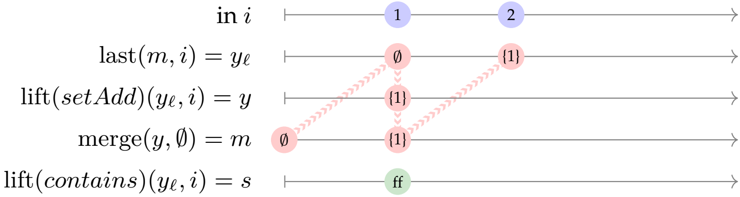
$\text{last}(m, i) = y_\ell$

$\text{lift}(\mathit{setAdd})(y_\ell, i) = y$

$\text{merge}(y, \emptyset) = m$

$\text{lift}(\mathit{contains})(y_\ell, i) = s$

Running Example



$\text{in } i$

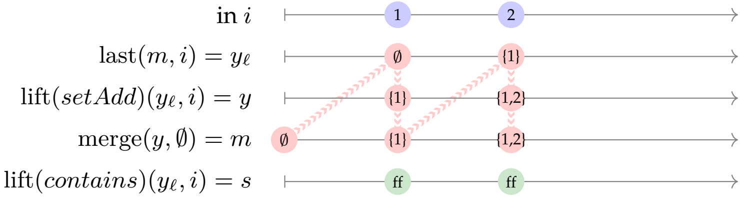
$\text{last}(m, i) = y_\ell$

$\text{lift}(setAdd)(y_\ell, i) = y$

$\text{merge}(y, \emptyset) = m$

$\text{lift}(contains)(y_\ell, i) = s$

Running Example



$\text{in } i$

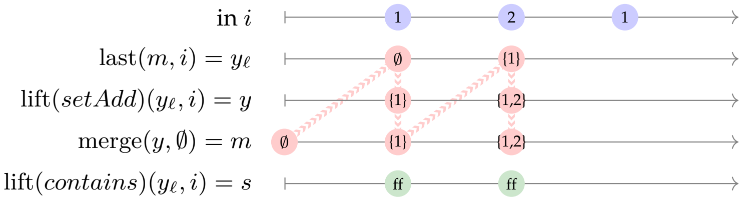
$\text{last}(m, i) = y_\ell$

$\text{lift}(\mathit{setAdd})(y_\ell, i) = y$

$\text{merge}(y, \emptyset) = m$

$\text{lift}(\mathit{contains})(y_\ell, i) = s$

Running Example



in i

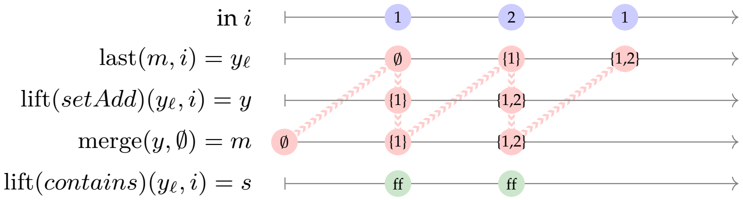
$\text{last}(m, i) = y_\ell$

$\text{lift}(\mathit{setAdd})(y_\ell, i) = y$

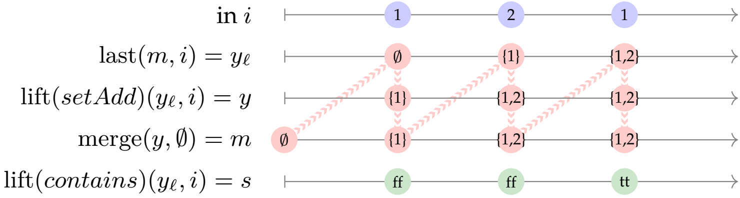
$\text{merge}(y, \emptyset) = m$

$\text{lift}(\mathit{contains})(y_\ell, i) = s$

Running Example

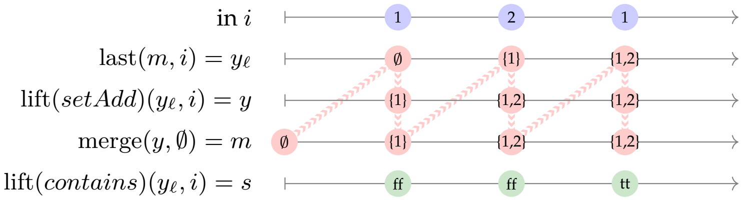

in i
1
2
1
$\mathrm{last}(m, i) = y_\ell$
∅
{1}
{1,2}
$\mathrm{lift}(setAdd)(y_\ell, i) = y$
{1}
{1,2}
$\mathrm{merge}(y, \emptyset) = m$
∅
{1}
{1,2}
$\mathrm{lift}(contains)(y_\ell, i) = s$
ff
ff

Running Example



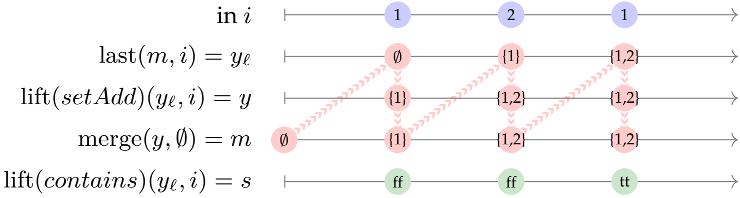
in i		1	2	1
$\text{last}(m, i) = y_\ell$		∅	{1}	{1,2}
$\text{lift}(setAdd)(y_\ell, i) = y$		{1}	{1,2}	{1,2}
$\text{merge}(y, \emptyset) = m$	∅	{1}	{1,2}	{1,2}
$\text{lift}(contains)(y_\ell, i) = s$		ff	ff	tt

Running Example


in i — 1, 2, 1
$\text{last}(m, i) = y_\ell$ — ∅, {1}, {1,2}
$\text{lift}(\mathit{setAdd})(y_\ell, i) = y$ — {1}, {1,2}, {1,2}
$\text{merge}(y, \emptyset) = m$ — ∅, {1}, {1,2}, {1,2}
$\text{lift}(\mathit{contains})(y_\ell, i) = s$ — ff, ff, tt

Evaluation of dataflow languages follows a basic scheme:

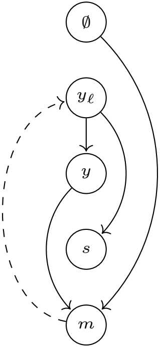
Running Example



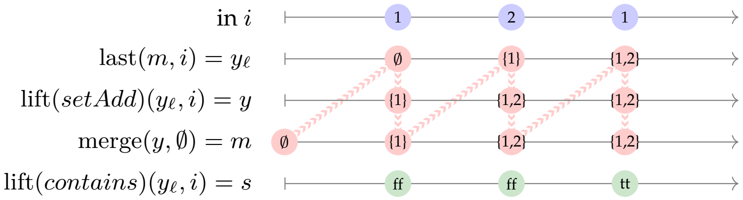
in i
1
2
1
$\text{last}(m, i) = y_\ell$
∅
{1}
{1,2}
$\text{lift}(setAdd)(y_\ell, i) = y$
{1}
{1,2}
{1,2}
$\text{merge}(y, \emptyset) = m$
∅
{1}
{1,2}
{1,2}
$\text{lift}(contains)(y_\ell, i) = s$
ff
ff
tt

Evaluation of dataflow languages follows a basic scheme:

- Construct dependency graph

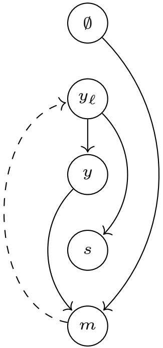


Running Example

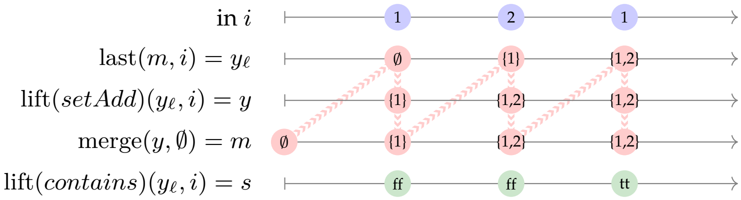

in i
$\text{last}(m, i) = y_\ell$
$\text{lift}(setAdd)(y_\ell, i) = y$
$\text{merge}(y, \emptyset) = m$
$\text{lift}(contains)(y_\ell, i) = s$
1
2
1
∅
{1}
{1,2}
{1}
{1,2}
{1,2}
∅
{1}
{1,2}
{1,2}
ff
ff
tt

Evaluation of dataflow languages follows a basic scheme:

- Construct dependency graph
- Find linear ordering of graph

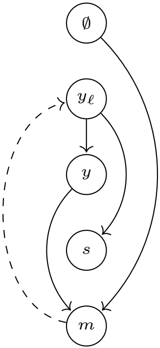


Running Example

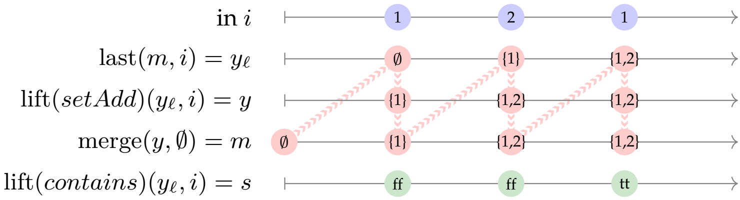

in i
1
2
1
$\mathrm{last}(m, i) = y_\ell$
∅
{1}
{1,2}
$\mathrm{lift}(\mathit{setAdd})(y_\ell, i) = y$
{1}
{1,2}
{1,2}
$\mathrm{merge}(y, \emptyset) = m$
∅
{1}
{1,2}
{1,2}
$\mathrm{lift}(\mathit{contains})(y_\ell, i) = s$
ff
ff
tt

Evaluation of dataflow languages follows a basic scheme:

- Construct dependency graph
- Find linear ordering of graph
- Continuously read inputs and calculate stream values in the given order

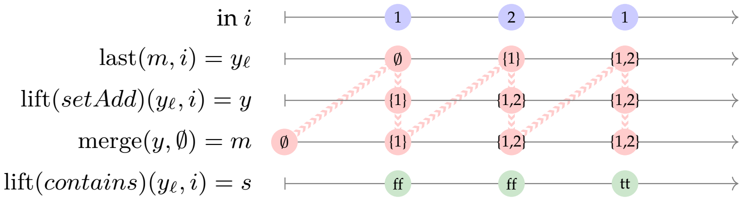


Aggregate Update Problem


in i
$\text{last}(m, i) = y_\ell$
$\text{lift}(\mathit{setAdd})(y_\ell, i) = y$
$\text{merge}(y, \emptyset) = m$
$\text{lift}(\mathit{contains})(y_\ell, i) = s$
1
2
1
∅
{1}
{1,2}
{1}
{1,2}
{1,2}
∅
{1}
{1,2}
{1,2}
ff
ff
tt

Dataflow languages have immutable semantics: After applying *setAdd* to the events from stream y_ℓ these (old) events may still be accessed.

Aggregate Update Problem

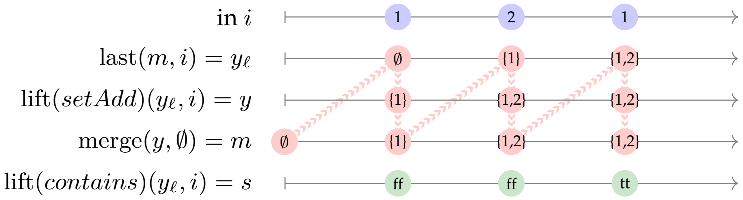


$\text{in } i$		1	2	1
$\text{last}(m, i) = y_\ell$		∅	{1}	{1,2}
$\text{lift}(\mathit{setAdd})(y_\ell, i) = y$		{1}	{1,2}	{1,2}
$\text{merge}(y, \emptyset) = m$	∅	{1}	{1,2}	{1,2}
$\text{lift}(\mathit{contains})(y_\ell, i) = s$		ff	ff	tt

Dataflow languages have immutable semantics: After applying *setAdd* to the events from stream y_ℓ these (old) events may still be accessed.

⇒ During evaluation every data structure must be copied before it is modified.

Aggregate Update Problem



in i
$\text{last}(m, i) = y_\ell$
$\text{lift}(setAdd)(y_\ell, i) = y$
$\text{merge}(y, \emptyset) = m$
$\text{lift}(contains)(y_\ell, i) = s$
1
2
1
∅
{1}
{1,2}
{1}
{1,2}
{1,2}
∅
{1}
{1,2}
{1,2}
ff
ff
tt

Dataflow languages have immutable semantics: After applying *setAdd* to the events from stream y_ℓ these (old) events may still be accessed.

⇒ During evaluation every data structure must be copied before it is modified.

But: In the concrete example the data structure fronm y_ℓ could be updated in-place iff stream s is calculated before y.

Aggregate Update Problem

The problem of finding the maximum number of data structures in a program that can be modified in-place is called **Aggregate Update Problem**.

Aggregate Update Problem

The problem of finding the maximum number of data structures in a program that can be modified in-place is called **Aggregate Update Problem**.

- Well studied in the field of functional languages.
- The number of variables that can be modified in place is dependent on the scheduling of the calculations.

Aggregate Update Problem

The problem of finding the maximum number of data structures in a program that can be modified in-place is called **Aggregate Update Problem**.

- Well studied in the field of functional languages.
- The number of variables that can be modified in place is dependent on the scheduling of the calculations.

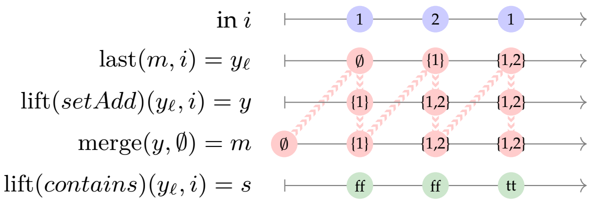
Our approach:

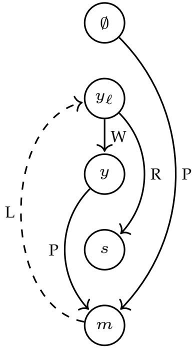
1. Finding the optimal translation order, s.t. as many data structures as possible can be modified in place.
2. Using **mutable** data structures for those that can be updated in place and **persistent** data structures for the other ones.

The optimization algorithm for TeSSLa

1. Classification of the edges in the usage graph:
Read, Write, Pass, Last edges

Example


in i
$\text{last}(m, i) = y_\ell$
$\text{lift}(setAdd)(y_\ell, i) = y$
$\text{merge}(y, \emptyset) = m$
$\text{lift}(contains)(y_\ell, i) = s$
1
2
1
∅
{1}
{1,2}
{1}
{1,2}
{1,2}
∅
{1}
{1,2}
{1,2}
ff
ff
tt



The optimization algorithm for TeSSLa

2. Finding potentially aliasing variables

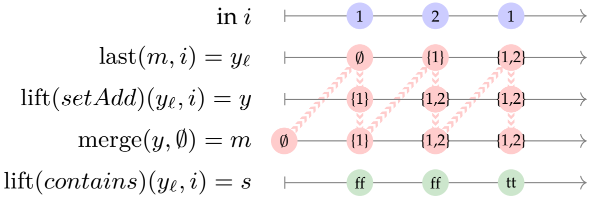
We call streams a and b potential aliases ($a \simeq b$), if we cannot prove them to be aliasing safe.

The optimization algorithm for TeSSLa

2. Finding potentially aliasing variables

We call streams a and b potential aliases ($a \simeq b$), if we cannot prove them to be aliasing safe.

Example:



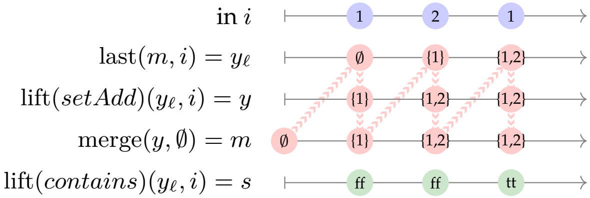
in i		1	2	1
$\text{last}(m, i) = y_\ell$		∅	{1}	{1,2}
$\text{lift}(\mathit{setAdd})(y_\ell, i) = y$		{1}	{1,2}	{1,2}
$\text{merge}(y, \emptyset) = m$	∅	{1}	{1,2}	{1,2}
$\text{lift}(\mathit{contains})(y_\ell, i) = s$		ff	ff	tt

The optimization algorithm for TeSSLa

2. Finding potentially aliasing variables

We call streams a and b potential aliases ($a \simeq b$), if we cannot prove them to be aliasing safe.

Example:



in i: 1, 2, 1

$\mathrm{last}(m, i) = y_\ell$: ∅, {1}, {1,2}

$\mathrm{lift}(setAdd)(y_\ell, i) = y$: {1}, {1,2}, {1,2}

$\mathrm{merge}(y, \emptyset) = m$: ∅, {1}, {1,2}, {1,2}

$\mathrm{lift}(contains)(y_\ell, i) = s$: ff, ff, tt

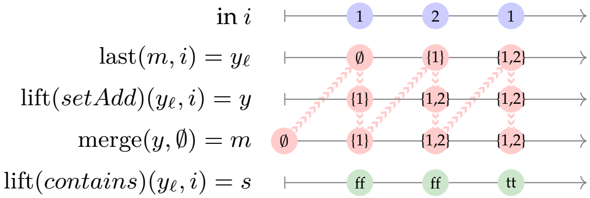
- $y_\ell \not\simeq m$: y_ℓ and m cannot have the same event at the same time

The optimization algorithm for TeSSLa

2. Finding potentially aliasing variables

We call streams a and b potential aliases ($a \simeq b$), if we cannot prove them to be aliasing safe.

Example:



in i
last$(m, i) = y_\ell$
lift$(setAdd)(y_\ell, i) = y$
merge$(y, \emptyset) = m$
lift$(contains)(y_\ell, i) = s$
1
2
1
∅
{1}
{1,2}
{1}
{1,2}
{1,2}
∅
{1}
{1,2}
{1,2}
ff
ff
tt

- $y_\ell \not\simeq m$: y_ℓ and m cannot have the same event at the same time
- $y \simeq m$: y and m may have the same event at the same time

The optimization algorithm for TeSSLa

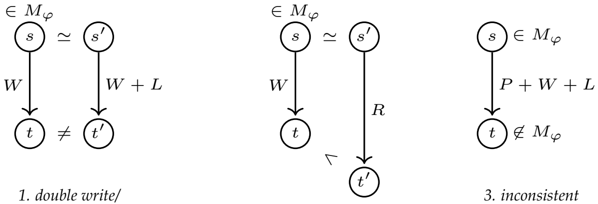
3. Criteria for mutable variables

A stream s may be implemented with mutable data structures ($s \in M_\varphi$), if none of the following patterns matches

The optimization algorithm for TeSSLa

3. Criteria for mutable variables

A stream s may be implemented with mutable data structures ($s \in M_\varphi$), if none of the following patterns matches



1. double write/ reproduction

2. read after write

3. inconsistent mutability

The optimization algorithm for TeSSLa

4. Algorithm for determination of the maximum set of mutable variables

The optimization algorithm for TeSSLa

4. Algorithm for determination of the maximum set of mutable variables

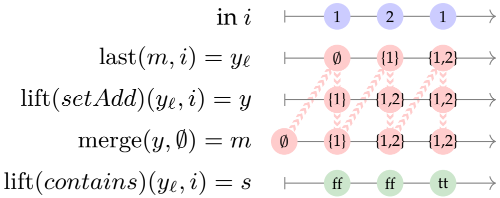
- Find variable families which can be all mutable or all persistent (rule 3).

The optimization algorithm for TeSSLa

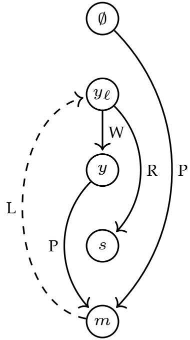
4. Algorithm for determination of the maximum set of mutable variables

- Find variable families which can be all mutable or all persistent (rule 3).

Example:



in i
$\text{last}(m, i) = y_\ell$
$\text{lift}(setAdd)(y_\ell, i) = y$
$\text{merge}(y, \emptyset) = m$
$\text{lift}(contains)(y_\ell, i) = s$
1
2
1
∅
{1}
{1,2}
{1}
{1,2}
{1,2}
∅
{1}
{1,2}
{1,2}
ff
ff
tt

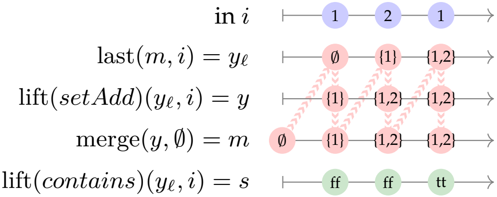


The optimization algorithm for TeSSLa

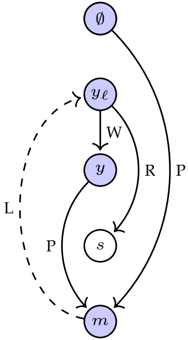
4. Algorithm for determination of the maximum set of mutable variables

- Find variable families which can be all mutable or all persistent (rule 3).

Example:



in i
1
2
1
$\text{last}(m, i) = y_\ell$
∅
{1}
{1,2}
$\text{lift}(setAdd)(y_\ell, i) = y$
{1}
{1,2}
{1,2}
$\text{merge}(y, \emptyset) = m$
∅
{1}
{1,2}
{1,2}
$\text{lift}(contains)(y_\ell, i) = s$
ff
ff
tt

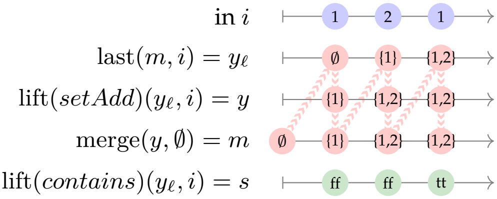


The optimization algorithm for TeSSLa

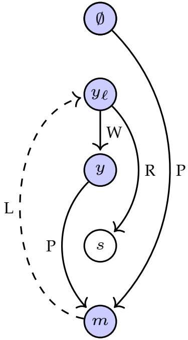
4. Algorithm for determination of the maximum set of mutable variables

- Climb up from Write nodes and search for aliases.

Example:



in i		1	2	1
$\text{last}(m, i) = y_\ell$		∅	{1}	{1,2}
$\text{lift}(\mathit{setAdd})(y_\ell, i) = y$		{1}	{1,2}	{1,2}
$\text{merge}(y, \emptyset) = m$	∅	{1}	{1,2}	{1,2}
$\text{lift}(\mathit{contains})(y_\ell, i) = s$		ff	ff	tt

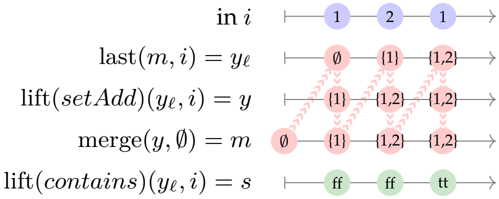


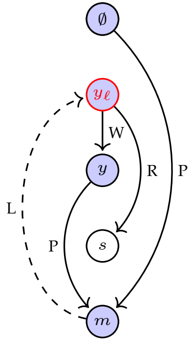
The optimization algorithm for TeSSLa

4. Algorithm for determination of the maximum set of mutable variables

- Climb up from Write nodes and search for aliases.

Example:


in i
1 2 1
$\text{last}(m, i) = y_\ell$
∅ {1} {1,2}
$\text{lift}(\mathit{setAdd})(y_\ell, i) = y$
{1} {1,2} {1,2}
$\text{merge}(y, \emptyset) = m$
∅ {1} {1,2} {1,2}
$\text{lift}(\mathit{contains})(y_\ell, i) = s$
ff ff tt

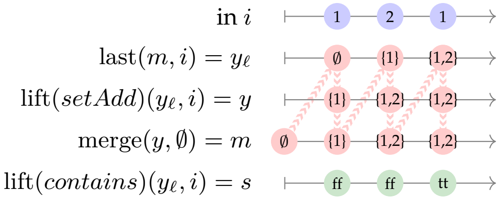


The optimization algorithm for TeSSLa

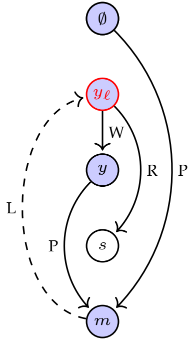
4. Algorithm for determination of the maximum set of mutable variables

- Make variable family persistent if rule 1 (double write/replicate) is breached.

Example:



in i: 1, 2, 1
$\text{last}(m, i) = y_\ell$: ∅, {1}, {1,2}
$\text{lift}(setAdd)(y_\ell, i) = y$: {1}, {1,2}, {1,2}
$\text{merge}(y, \emptyset) = m$: ∅, {1}, {1,2}, {1,2}
$\text{lift}(contains)(y_\ell, i) = s$: ff, ff, tt

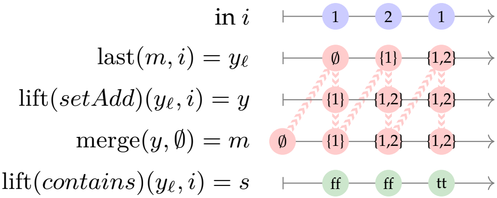


The optimization algorithm for TeSSLa

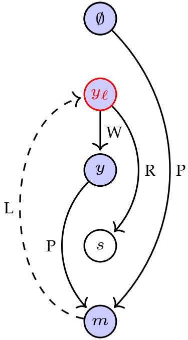
4. Algorithm for determination of the maximum set of mutable variables

- Include edges for Read-Before-Write dependencies (rule 2) in usage graph.

Example:



in i
1
2
1
$\mathrm{last}(m, i) = y_\ell$
∅
{1}
{1,2}
$\mathrm{lift}(setAdd)(y_\ell, i) = y$
{1}
{1,2}
{1,2}
$\mathrm{merge}(y, \emptyset) = m$
∅
{1}
{1,2}
{1,2}
$\mathrm{lift}(contains)(y_\ell, i) = s$
ff
ff
tt

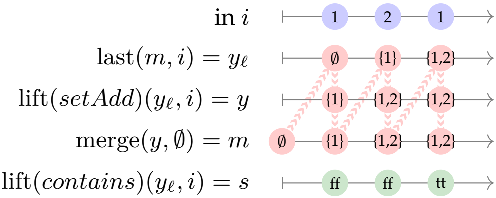


The optimization algorithm for TeSSLa

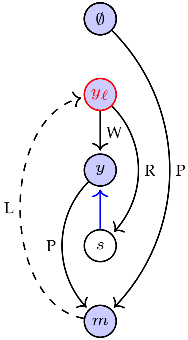
4. Algorithm for determination of the maximum set of mutable variables

- Include edges for Read-Before-Write dependencies (rule 2) in usage graph.

Example:



in i
1
2
1
$\text{last}(m, i) = y_\ell$
∅
{1}
{1,2}
$\text{lift}(setAdd)(y_\ell, i) = y$
{1}
{1,2}
{1,2}
$\text{merge}(y, \emptyset) = m$
∅
{1}
{1,2}
{1,2}
$\text{lift}(contains)(y_\ell, i) = s$
ff
ff
tt

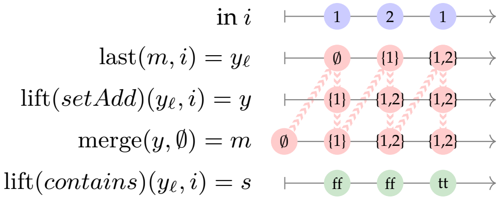


The optimization algorithm for TeSSLa

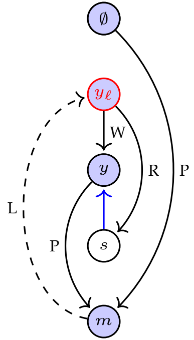
4. Algorithm for determination of the maximum set of mutable variables

- Find optimal translation order of usage graph (NP-complete).

Example:



in i
1
2
1
$\text{last}(m, i) = y_\ell$
∅
{1}
{1,2}
$\text{lift}(setAdd)(y_\ell, i) = y$
{1}
{1,2}
{1,2}
$\text{merge}(y, \emptyset) = m$
∅
{1}
{1,2}
{1,2}
$\text{lift}(contains)(y_\ell, i) = s$
ff
ff
tt

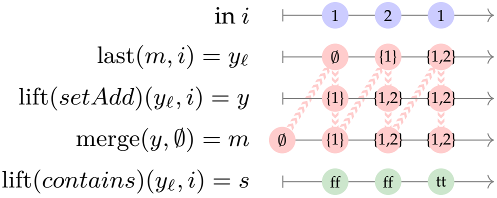


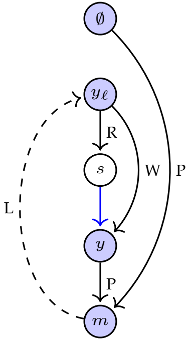
The optimization algorithm for TeSSLa

4. Algorithm for determination of the maximum set of mutable variables

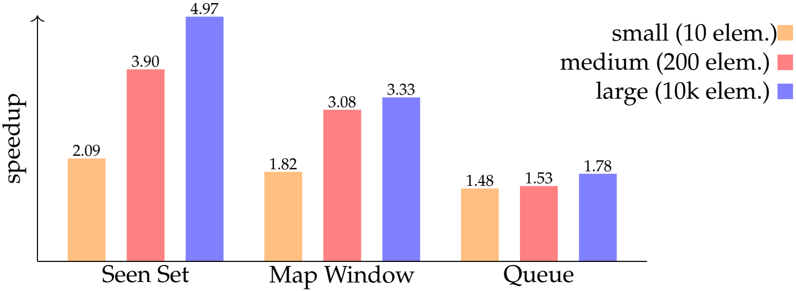
- Find optimal translation order of usage graph (NP-complete).

Example:


in i
1
2
1
$\text{last}(m, i) = y_\ell$
∅
{1}
{1,2}
$\text{lift}(\mathit{setAdd})(y_\ell, i) = y$
{1}
{1,2}
{1,2}
$\text{merge}(y, \emptyset) = m$
∅
{1}
{1,2}
{1,2}
$\text{lift}(\mathit{contains})(y_\ell, i) = s$
ff
ff
tt



Evaluation of the approach: Synthetic examples



Speedups compared to the non-optimized (fully persistent) implementation (10^9 input events).

Evaluation of the approach: Real-World examples

Specification	**Optimized**	**Non-optimized**	**Speedup**
DBTimeCons.	171 s	216 s	1.3
DBAccessCons.(full)	233 s	> 1 h	> 15.5
DBAccessCons.(33 %)	59.2 s	127 s	2.1
PeakDetection	7.56 s	14.0 s	1.9
SpectrumCalc.	1.04 s	2.07 s	2.0

Conclusion

- Dataflow languages can be evaluated by iteratively calculating stream events in the correct order.
- The Aggregate Update Problem deals with the question which data-structures can be updated in place.
- We presented a solution for finding the perfect ordering to maximize in place updates.
- The evaluation showed significant speedups.

Contact information

Contact: kallwies@isp.uni-luebeck.de