Array Types for a Graph Processing Language

Mark Tullsen and Matt Sottile

Galois, Inc.

GABB, May 2016

Outline

1. Introduction
2. Capturing Structure
3. Representations
4. “Partial Arrays” (I.e., Maps, Dictionaries, Etc.)
5. Index Spaces
6. Larger Examples
7. Conclusion

1 Introduction

2 Capturing Structure

3 Representations

4 “Partial Arrays” (I.e., Maps, Dictionaries, Etc.)

5 Index Spaces

6 Larger Examples

7 Conclusion

LPG and HAL

The context of HAL: designing & implementing LPG

LPG (Language for Processing Graphs):

- A declarative DSL for medium-size[1] graph processing.
- Goal: Architecture Neutral
- Goal: Maximize implicit parallelism

HAL (Hierarchical Array Language):

- a declarative array language,
- the primary abstraction between our graph algorithms and the parallel architecture.

[1] Fit on single-host.

Why an Intermediate Array Language?

Why an Intermediate Array Language?

- Graph algorithms
  - Need many data structures, not just graphs
  - Often look like array-processing
    - Existing work on parallelizing array languages
- Very natural when thinking in terms of adjacency matrices
  - Classes of Graphs have analogs in HAL's various array structures
- We expect this will enable more efficient code
  - More parallelism
  - Have a richer set of laws (for transformation)

HAL (Hierarchical Array Language)

- Declarative
- A very expressive type system
  - captures more structure
  - provides more laws
  - sums!
  - allows us to provide bijections and views
- Standard and “partial” (associative) arrays
- First class index spaces

HAL (Hierarchical Array Language)

- Declarative
- A very expressive type system
  - captures more structure
  - provides more laws
  - sums!
  - allows us to provide bijections and views
- Standard and “partial” (associative) arrays
- First class index spaces

CAVEAT:

HAL (Hierarchical Array Language)

- Declarative
- A very expressive type system
  - captures more structure
  - provides more laws
  - sums!
  - allows us to provide bijections and views
- Standard and "partial" (associative) arrays
- First class index spaces

CAVEAT:

- HAL is in the design & prototype stage: no compiler and no performance figures yet.

1 Introduction

2 Capturing Structure

3 Representations

4 “Partial Arrays” (I.e., Maps, Dictionaries, Etc.)

5 Index Spaces

6 Larger Examples

7 Conclusion

A simple matrix, *m*

```
0 0 0 0
0 1 0 0
0 0 2 0
0 0 0 3
```

$$m :: \mathbb{Z}_4 \times \mathbb{Z}_4 \Rightarrow \mathbb{Z}$$
$$m :: \mathrm{VEC}^2(\mathbb{Z}_4) \Rightarrow \mathbb{Z}$$

```
m₁ = arr[ℤ4×ℤ4] [0 0 0 0
                  0 1 0 0
                  0 0 2 0
                  0 0 0 3]
```

$|m_1| = 1{+}4{\times}4$

```
m₂ = unnest(arr[ℤ4]
              [ arr[ℤ4] [0 0 0 0]
              , arr[ℤ4] [0 1 0 0]
              , arr[ℤ4] [0 0 2 0]
              , arr[ℤ4] [0 0 0 3]
              ])
```

$|m_2| = 1{+}4{\times}4$

|m| = storage requirements of m, in machine words

Upper triangular, all zeros

$$\begin{array}{ccc} \emptyset & \emptyset & \emptyset \\ & \emptyset & \emptyset \\ & & \emptyset \end{array}$$

$$u :: \mathrm{SET}^2(\mathbb{Z}_4) \Rightarrow \mathbb{Z}$$

$$u_1 = \mathrm{arr}[\mathrm{SET}^2(\mathbb{Z}_4)]\ [\emptyset\ \emptyset\ \emptyset\ \emptyset\ \emptyset\ \emptyset] \qquad |u_1| = 1+6$$

$$u_2 = \mathrm{const}[\mathrm{SET}^2(\mathbb{Z}_4)]\ \emptyset \qquad |u_2| = 1$$

Upper triangular, all zeros

$$\begin{array}{ccc} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ & \mathbf{0} & \mathbf{0} \\ & & \mathbf{0} \end{array}$$

$u :: \mathtt{SET}^2(\mathbb{Z}_4) \Rightarrow \mathbb{Z}$

$u_1 = \mathtt{arr}[\mathtt{SET}^2(\mathbb{Z}_4)]\ [\mathbf{0}\ \mathbf{0}\ \mathbf{0}\ \mathbf{0}\ \mathbf{0}\ \mathbf{0}]$ $\qquad |u_1| = 1{+}6$

$u_2 = \mathtt{const}[\mathtt{SET}^2(\mathbb{Z}_4)]\ \mathbf{0}$ $\qquad |u_2| = 1$

$\mathtt{VEC}^2(\mathbb{Z}_4)$

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

Upper triangular, all zeros

```
0 0 0
  0 0
    0
```

$\mathtt{u} :: \mathrm{SET}^2(\mathbb{Z}_4) \Rightarrow \mathbb{Z}$

$\mathtt{u}_1 = \mathtt{arr}[\mathrm{SET}^2(\mathbb{Z}_4)]\ [0\ 0\ 0\ 0\ 0\ 0]$ $\qquad |\mathtt{u}_1| = 1+6$

$\mathtt{u}_2 = \mathtt{const}[\mathrm{SET}^2(\mathbb{Z}_4)]\ 0$ $\qquad |\mathtt{u}_2| = 1$

$\mathrm{SET}^2(\mathbb{Z}_4)$

	{0,1}	{0,2}	{0,3}
		{1,2}	{1,3}
			{2,3}

A sequence on the diagonal

```
0
   1
      2
         3
```

$\mathtt{d} :: \mathbb{Z}_4 \Rightarrow \mathbb{Z}_4$

$\mathtt{d}_1 = \mathtt{arr}[\mathbb{Z}_4]\ [\mathtt{0\ 1\ 2\ 3}]$ $\qquad |\mathtt{d}_1| = \mathtt{1{+}4}$

$\mathtt{d}_2 = \mathtt{smart}[\mathbb{Z}_4]\ \mathtt{id}$ $\qquad |\mathtt{d}_2| = \mathtt{1}$

Combining . . . to define *m* in HAL

```
0  0  0  0
0  1  0  0
0  0  2  0
0  0  0  3
```

$\mathtt{m} :: \mathtt{VEC}^2(\mathbb{Z}_4) \Rightarrow \mathbb{Z}$

$\mathtt{m}_1 = \mathtt{arr}[\mathbb{Z}_4\times\mathbb{Z}_4]\ [\mathtt{0\ 0\ 0\ 0\ 0\ 1\ ...}]$ $\qquad |\mathtt{m}_1| = 1+4\times 4$

$\mathtt{m}_3 = \mathtt{fromTris}\ (\ \mathtt{const}[\mathtt{SET}^2(\mathbb{Z}_4)]\ \mathtt{0}$ $\qquad |\mathtt{m}_3| = 1+1+1+1$

$\qquad\qquad\qquad ,\ \mathtt{smart}[\mathbb{Z}_4]\qquad \mathtt{id}$

$\qquad\qquad\qquad ,\ \mathtt{const}[\mathtt{SET}^2(\mathbb{Z}_4)]\ \mathtt{0}\ )$

- same type, same interface
- `const` & `smart` provide
  - correctness by construction
  - smaller representations

1 Introduction

2 Capturing Structure

3 Representations

4 “Partial Arrays” (I.e., Maps, Dictionaries, Etc.)

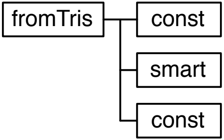
5 Index Spaces

6 Larger Examples

7 Conclusion

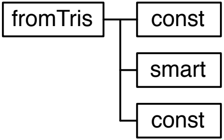
smart/const/... are represented by "tagged values"

```
fromTris( const.., smart.., const..)        || = 1+1+1+1
```



smart/const/... are represented by “tagged values”

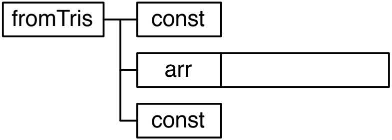
```
fromTris( const.., smart.., const..)        || = 1+1+1+1
```



- Could nest arbitrarily, thus “Hierarchical Array Language”

Converting to Expanded Representations

```
fromTris( const.., @smart.., const..)        || = 1+1+4+1
```



- The `@` operator expands our const/smart constructors to `arr`-like, flat representations. Thus, we can define things semantically then get either
  - compact representation
  - flattened representation

Converting to Expanded Representations (2)

```
@fromTris( const.., smart.., const..)        || = 1+16
```

arr	

Compact and Expanded Arrays, Mixed

```
0 5 9 1
0 0 2 5
0 0 0 3
0 0 0 0
```

$\texttt{m2 ::} \ \mathtt{VEC}^2(\mathbb{Z}_4) \Rightarrow \mathbb{Z}$

$$
\begin{array}{llll}
\texttt{m2 = fromTris} & (& \texttt{arr}[\mathtt{SET}^2(\mathbb{Z}_4)] & [5\ 9\ 1\ 2\ 5\ 3] \\
 & , & \texttt{const}[\mathbb{Z}_4] & 0 \\
 & , & \texttt{const}[\mathtt{SET}^2(\mathbb{Z}_4)] & 0 \\
 &) & &
\end{array}
$$

Compact and Expanded Arrays, Mixed

```
0  5  9  1
0  0  2  5
0  0  0  3
0  0  0  0
```

$\texttt{m2 :: VEC}^2(\mathbb{Z}_4) \Rightarrow \mathbb{Z}$

$$
\begin{array}{llll}
\texttt{m2 = fromTris} & \texttt{(} & \texttt{arr[SET}^2(\mathbb{Z}_4)\texttt{]} & \texttt{[5 9 1 2 5 3]} \\
 & \texttt{,} & \texttt{const[}\mathbb{Z}_4\texttt{]} & \texttt{0} \\
 & \texttt{,} & \texttt{const[SET}^2(\mathbb{Z}_4)\texttt{]} & \texttt{0} \\
 & \texttt{)} & &
\end{array}
$$

- NOTE: This is not a substitute for explicitly defining symmetric/triangular matrices.

A Triangular Matrix

$$\begin{array}{ccc} 5 & 9 & 1 \\ & 2 & 5 \\ & & 3 \end{array}$$

$\mathtt{m3} :: \mathtt{SET}^2(\mathbb{Z}_4) \Rightarrow \mathbb{Z}$

$\mathtt{m3} = \mathtt{arr}[\mathtt{SET}^2(\mathbb{Z}_4)]\ [5\ 9\ 1\ 2\ 5\ 3]$

$|\mathtt{m3}| = 1 + (4\ \mathtt{choose}\ 2) = 1 + 6$

1 Introduction

2 Capturing Structure

3 Representations

4 “Partial Arrays” (I.e., Maps, Dictionaries, Etc.)

5 Index Spaces

6 Larger Examples

7 Conclusion

Partial Arrays

- *Partial Array* = map, dictionary, associative array, ...
- Examples:

`{1: 2, 5: 3, ...}`	$a1 :: \mathbb{Z}_{50} \mapsto \mathbb{Z}$ $\|a1\| = 1{+}2{\times}nnz(a1)$
`{1: 2, 5: 3, ...}` *DFLT* `0`	$a2 :: \mathbb{Z}_{50} \Rightarrow \mathbb{Z}$ $\|a2\| = 1{+}2{\times}nnz(a1)$
`@({1: 2, 5: 3, ...}` *DFLT* `0)`	$a3 :: \mathbb{Z}_{50} \Rightarrow \mathbb{Z}$ $\|a3\| = 1{+}50$

- `nnz` - number of non zeros, loosely

Aside: Regarding Sums

- A sum type

  ```
  a + b + c
  ```

- `a` or `b` or `c` (but tagged to distinguish)
- Like 'tagged' unions in C, but *safe*
- Haskell's "`Maybe a`" type (ML's "`option a`") in HAL:

  $\mathbb{Z}_1$ + a

Partial Arrays: Semantics

- Semantically we can understand partial arrays through this (provided) bijection between partial and total arrays:

$$\mathtt{a} \;\square\!\!\Rightarrow \mathtt{b} \qquad \Longleftrightarrow \qquad \mathtt{a} \;\Rightarrow\; \mathbb{Z}_1 + \mathtt{b}$$

- Operationally
  - Uses a sparse representation, by default
  - Can iterate over in time proportional to `nnz(parray)`
- Semantic partiality is orthogonal to sparse *representations*
  - can have sparse representations for (total) arrays (*DFLT*)
  - can have non-sparse representations for partial arrays

1 Introduction

2 Capturing Structure

3 Representations

4 “Partial Arrays” (I.e., Maps, Dictionaries, Etc.)

5 Index Spaces

6 Larger Examples

7 Conclusion

Index Space Transformation + Bijection

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

(0,0)	(0,1)	(0,0)	(0,1)
(1,0)	(1,1)	(1,0)	(1,1)
(0,0)	(0,1)	(0,0)	(0,1)
(1,0)	(1,1)	(1,0)	(1,1)

(0,0)	(0,1)
(1,0)	(1,1)

(0,0)	(0,1)
(1,0)	(1,1)

(0,0)	(0,1)
(1,0)	(1,1)

(0,0)	(0,1)
(1,0)	(1,1)

An Extremely Useful Bijection

(0,0)	(0,1)	(0,0)	(0,1)
(1,0)	(1,1)	(1,0)	(1,1)
(0,0)	(0,1)	(0,0)	(0,1)
(1,0)	(1,1)	(1,0)	(1,1)

0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

(0,0)	(0,0)	(0,1)	(0,1)
(1,0)	(1,0)	(1,1)	(1,1)
(2,0)	(2,0)	(2,1)	(2,1)
(3,0)	(3,0)	(3,1)	(3,1)

An Extremely Useful Bijection

(0,0)	(0,1)	(0,0)	(0,1)
(1,0)	(1,1)	(1,0)	(1,1)
(0,0)	(0,1)	(0,0)	(0,1)
(1,0)	(1,1)	(1,0)	(1,1)

0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

(0,0)	(0,0)	(0,1)	(0,1)
(1,0)	(1,0)	(1,1)	(1,1)
(2,0)	(2,0)	(2,1)	(2,1)
(3,0)	(3,0)	(3,1)	(3,1)

- I.e., if we can partition the index-space (via a bijection)
  - we can decompose the array
- This allows for
  - processor partitioning
  - divide-and-conquer algorithms
  - nice interactions with the smart/const constructors

An Extremely Useful Bijection

(0,0)	(0,1)	(0,0)	(0,1)
(1,0)	(1,1)	(1,0)	(1,1)
(0,0)	(0,1)	(0,0)	(0,1)
(1,0)	(1,1)	(1,0)	(1,1)

0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

(0,0)	(0,0)	(0,1)	(0,1)
(1,0)	(1,0)	(1,1)	(1,1)
(2,0)	(2,0)	(2,1)	(2,1)
(3,0)	(3,0)	(3,1)	(3,1)

0	{0,1}	{0,2}	{0,3}
{0,1}	1	{1,2}	{1,3}
{0,2}	{1,2}	2	{2,3}
{0,3}	{1,3}	{2,3}	3

{0,3}	{0,2}	{0,1}	0
{1,3}	{1,2}	1	{0,1}
{2,3}	2	{1,2}	{0,2}
3	{2,3}	{1,3}	{0,3}

{0,0}	{0,1}	{0,2}	{0,3}
{0,1}	{1,1}	{1,2}	{1,3}
{0,2}	{1,2}	{2,2}	{2,3}
{0,3}	{1,3}	{2,3}	{3,3}

0	(0,1)	(0,2)	(0,3)
(1,0)	1	(1,2)	(1,3)
(2,0)	(2,1)	2	(2,3)
(3,0)	(3,1)	(3,2)	3

An Extremely Useful Bijection

(0,0)	(0,1)	(0,0)	(0,1)
(1,0)	(1,1)	(1,0)	(1,1)
(0,0)	(0,1)	(0,0)	(0,1)
(1,0)	(1,1)	(1,0)	(1,1)

0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

(0,0)	(0,0)	(0,1)	(0,1)
(1,0)	(1,0)	(1,1)	(1,1)
(2,0)	(2,0)	(2,1)	(2,1)
(3,0)	(3,0)	(3,1)	(3,1)

0	{0,1}	{0,2}	{0,3}
{0,1}	1	{1,2}	{1,3}
{0,2}	{1,2}	2	{2,3}
{0,3}	{1,3}	{2,3}	3

{0,3}	{0,2}	{0,1}	0
{1,3}	{1,2}	1	{0,1}
{2,3}	2	{1,2}	{0,2}
3	{2,3}	{1,3}	{0,3}

{0,0}	{0,1}	{0,2}	{0,3}
{0,1}	{1,1}	{1,2}	{1,3}
{0,2}	{1,2}	{2,2}	{2,3}
{0,3}	{1,3}	{2,3}	{3,3}

0	(0,1)	(0,2)	(0,3)
(1,0)	1	(1,2)	(1,3)
(2,0)	(2,1)	2	(2,3)
(3,0)	(3,1)	(3,2)	3

- Without sums, we'd only be able to partition into equal sized parts.
    - The indices remain unique and form a valid type

1 Introduction

2 Capturing Structure

3 Representations

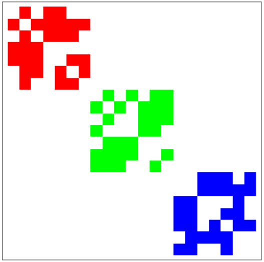
4 “Partial Arrays” (I.e., Maps, Dictionaries, Etc.)

5 Index Spaces

6 Larger Examples

7 Conclusion

A Diagonal Block Array

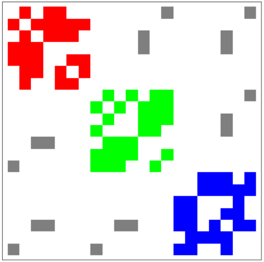


```
m1, m2, m3 :: ℤ7×ℤ7 ⇒ ℤ2

dba :: ℤ21×ℤ21 ⇒ ℤ2
dba =
 unblock
   (fromTris
     ( const[SET²(ℤ3)] (const[ℤ7×ℤ7] 0)
     , arr[ℤ3] [m1, m2, m3]
     , const[SET²(ℤ3)] (const[ℤ7×ℤ7] 0)
     )
```

- using $\mathbb{Z}_2$ to represent booleans (1 signifies edge between)

A Diagonal Block Array, with Sparse Off-Diagonals



```
m1, m2, m3 :: ℤ7×ℤ7 ⇒ ℤ2

dba' :: ℤ21×ℤ21 ⇒ ℤ2
dba' = unblock (arr [ℤ3×ℤ3]
  [       m1       , {..} DFLT 0, {..} DFLT 0
  , {..} DFLT 0,       m2       , {..} DFLT 0
  , {..} DFLT 0, {..} DFLT 0,       m3
  ])
```

- if adjacency matrix of graph, 3 graphs with sparse connectivity between

An Unfortunate Situation

- Many data-structures have “clean” decompositions
  - These support divide-and-conquer algorithms
  - E.g., lists (head & tail, split, ...), queues, binary-trees (left & right), etc.
- Graphs are not one of these
  - Though we do have some divide-and-conquer schemes, such as map-reduce.
- How might we decompose the graph just shown?
  - Has “obvious locality”

One Method to Divide-And-Conquer Graphs

Incomplete Graphs:

One Method to Divide-And-Conquer Graphs

Incomplete Graphs:

- . . . *generalize* graphs

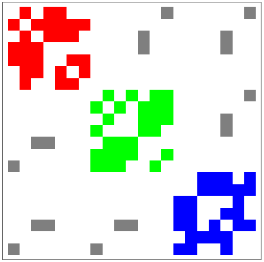
One Method to Divide-And-Conquer Graphs

Incomplete Graphs:

- . . . *generalize* graphs
- . . . allow us to group vertices (similar to super-vertices)

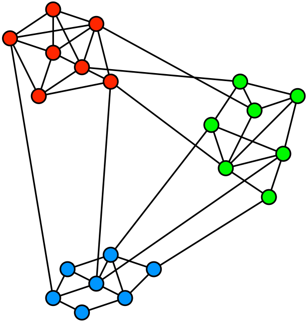
Incomplete Graphs Visualized

The adjacency matrix of graph:



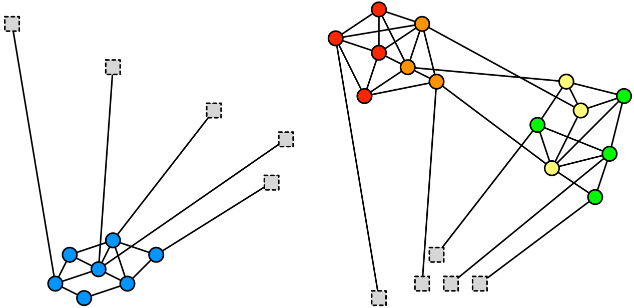
Incomplete Graphs Visualized

The graph:



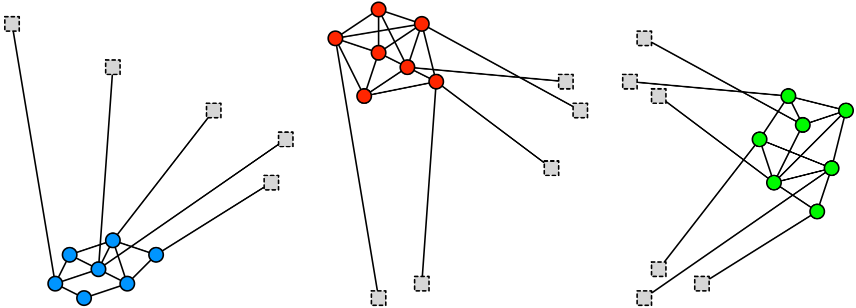
Incomplete Graphs Visualized

Extract blue nodes; edges get “split in half”:



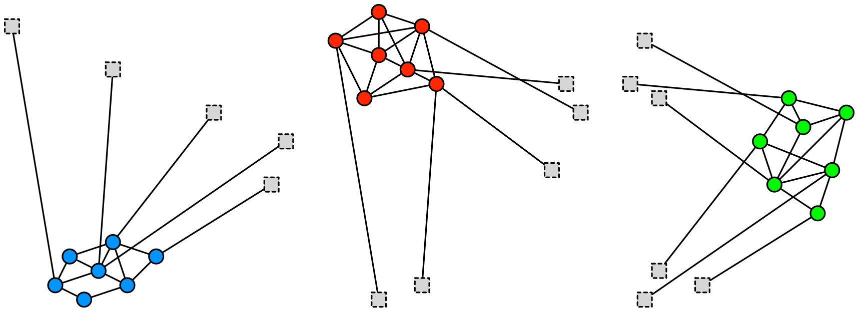
Incomplete Graphs Visualized

Extract green nodes, edges get "split in half":



Incomplete Graphs Visualized

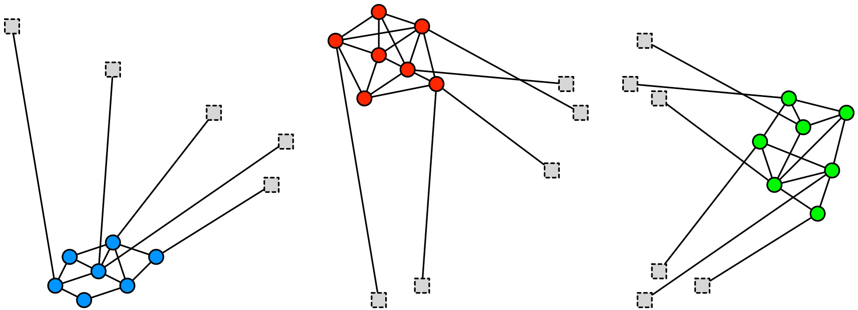
Extract green nodes, edges get "split in half":



- Can merge IGs in any order (associative and commutative)

Incomplete Graphs Visualized

Extract green nodes, edges get "split in half":



- Can merge IGs in any order (associative and commutative)
- Can do computations on IGs, computations updated when we merge

1 Introduction

2 Capturing Structure

3 Representations

4 “Partial Arrays” (I.e., Maps, Dictionaries, Etc.)

5 Index Spaces

6 Larger Examples

7 Conclusion

Observations

- Multiple features are working synergistically:
  - smart/const arrays (with a default tagged rep.)
  - bijections
  - expressive types
  - `@`, the expansion operator
- Novel features
  - powerful index space transformations
  - use of sums in an array language
  - type system (more expressive than most)
- Results
  - separation of interface and representation
  - expressive, high level array transformations
  - get views, and in-place updates, when we program with bijections

Conclusion

- In the paper, we discuss many more aspects of the language
  - types
  - functors (map-like functions)
  - program laws and transformations
  - the four combinatorial collections: `SET`, `VEC`, `PERM`, `MSET`
- Current status of project
  - A proof of concept embedded in Haskell
  - A few graph algorithms (Borůvka, Triangle Counting)
  - See the github project: `https://github.com/GaloisInc/lpg`
- Future
  - Need to “test drive” on more algorithms
  - Exploring yet more expressiveness in the type system
  - ...

Thank You

Supplementary Slides

Partial Arrays: representation and partiality are orthogonal

- sparse rep. of total arrays

  `{1: 2, 5: 3, 7: 0, ...}` *DFLT* `0` $::$ $\mathbb{Z}_{50} \Rightarrow \mathbb{Z}$, `||=2×nnz`

Partial Arrays: representation and partiality are orthogonal

- sparse rep. of total arrays

  `{1: 2, 5: 3, 7: 0, ...}` ___DFLT___ `0` $::$ $\mathbb{Z}_{50} \Rightarrow \mathbb{Z}$, `||=2×nnz`

- a flattened rep. of partial arrays

  `t2p(@p2t{1: 2, 5: 3, 7: 0, ...})` $::$ $\mathbb{Z}_{50}$ $\square\!\!\Rightarrow \mathbb{Z}$, `||=1+50`

Partial Arrays: representation and partiality are orthogonal

- sparse rep. of total arrays

  `{1: 2, 5: 3, 7: 0, ...}` <u>*DFLT*</u> `0` `::` $\mathbb{Z}_{50} \Rightarrow \mathbb{Z}$, `||=2×nnz`

- a flattened rep. of partial arrays

  `t2p(@p2t{1: 2, 5: 3, 7: 0, ...})` `::` $\mathbb{Z}_{50}$ ⇰ $\mathbb{Z}$, `||=1+50`

  `^^^^^^` $\mathbb{Z}_{50}$ ⇰ $\mathbb{Z}$ `^^^^^^^^`

Partial Arrays: representation and partiality are orthogonal

- sparse rep. of total arrays

  `{1: 2, 5: 3, 7: 0, ...}` <u>*DFLT*</u> `0` `::` $\mathbb{Z}_{50} \Rightarrow \mathbb{Z}$, `||=2×nnz`

- a flattened rep. of partial arrays

  `t2p(@p2t{1: 2, 5: 3, 7: 0, ...})` `::` $\mathbb{Z}_{50}$ □⇒ $\mathbb{Z}$, `||=1+50`

  `^^^^^^^^` $\mathbb{Z}_{50} \Rightarrow 1 + \mathbb{Z}$ `^^^^^^^^`

Partial Arrays: representation and partiality are orthogonal

- sparse rep. of total arrays

  `{1: 2, 5: 3, 7: 0, ...}` <u>*DFLT*</u> `0` `::` $\mathbb{Z}_{50} \Rightarrow \mathbb{Z}$ `, ||=2×nnz`

- a flattened rep. of partial arrays

  `t2p(@p2t{1: 2, 5: 3, 7: 0, ...})` `::` $\mathbb{Z}_{50} \boxed{} \!\Rightarrow \mathbb{Z}$ `, ||=1+50`

  `^^^^^^^^` $\mathbb{Z}_{50} \boxed{} \!\Rightarrow \mathbb{Z}$ `^^^^^^^^^^^^^^^`

What is a graph?

Graphs parameterized by vertex and edge:

$$\texttt{type DG v e} = \texttt{VEC}^2\texttt{(v)} \mathrel{\square\!\!\Rightarrow} \texttt{e} \qquad \texttt{-- directed}$$
$$\texttt{type UG v e} = \texttt{MSET}^2\texttt{(v)} \mathrel{\square\!\!\Rightarrow} \texttt{e} \qquad \texttt{-- undirected}$$

No edge data:

$$\texttt{type DG' v} = \texttt{VEC}^2\texttt{(v)} \Rightarrow \mathbb{Z}_2 \qquad \texttt{-- directed}$$
$$\texttt{type UG' v} = \texttt{MSET}^2\texttt{(v)} \Rightarrow \mathbb{Z}_2 \qquad \texttt{-- undirected}$$

An aside: transformations

```
mapRes compare (smart[t] id)
```

An aside: transformations

```
  mapRes compare (smart[t] id)

= smart[t] (compare . id)
```

An aside: transformations

```
  mapRes compare (smart[t] id)

= smart[t] (compare . id)

= smart[t] compare
```

An aside: transformations

```
  mapRes compare (smart[t] id)

= smart[t] (compare . id)

= smart[t] compare

= smart[t] (id . compare)
```

An aside: transformations

```
  mapRes compare (smart[t] id)

= smart[t] (compare . id)

= smart[t] compare

= smart[t] (id . compare)

= mapIdx compare (smart[t] id)
```

compare (and other bijections on indices)

$$
\texttt{compare :: VEC}^2\texttt{(a)} \iff \begin{array}{l} \ \texttt{LT: SET}^2\texttt{(a)} \\ +\ \texttt{EQ: a} \\ +\ \texttt{GT: SET}^2\texttt{(a)} \end{array} = \texttt{COMPARE(a)}
$$

compare (and other bijections on indices)

$$
\begin{array}{lcl}
\texttt{compare} :: \mathtt{VEC}^2(\mathtt{a}) & \Longleftrightarrow & \begin{array}{l} \ \mathtt{LT}:\ \mathtt{SET}^2(\mathtt{a}) \\ +\ \mathtt{EQ}:\ \mathtt{a} \\ +\ \mathtt{GT}:\ \mathtt{SET}^2(\mathtt{a}) \end{array} = \mathtt{COMPARE}(\mathtt{a})
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{less} :: \mathtt{VEC}^2(\mathtt{a}) & \Longleftrightarrow & \begin{array}{l} \ \mathtt{LT}:\ \mathtt{SET}^2(\mathtt{a}) \\ +\ \mathtt{GTE}:\ \mathtt{MSET}^2(\mathtt{a}) \end{array}
\end{array}
$$

compare (and other bijections on indices)

```
compare :: VEC²(a)  ⟺   LT: SET²(a)     =  COMPARE(a)
                       + EQ: a
                       + GT: SET²(a)

less    :: VEC²(a)  ⟺   LT:  SET²(a)
                      + GTE: MSET²(a)

eq      :: VEC²(a)  ⟺   EQ: a
                       + NE: PERM²(a)
```

compare (and other bijections on indices)

$$
\begin{array}{llll}
\texttt{compare} :: \texttt{VEC}^2(\texttt{a}) & \Longleftrightarrow & \ \texttt{LT}: \texttt{SET}^2(\texttt{a}) & = \texttt{COMPARE(a)} \\
& & +\ \texttt{EQ}: \texttt{a} & \\
& & +\ \texttt{GT}: \texttt{SET}^2(\texttt{a}) &
\end{array}
$$

$$
\begin{array}{lll}
\texttt{less} :: \texttt{VEC}^2(\texttt{a}) & \Longleftrightarrow & \ \texttt{LT}: \texttt{SET}^2(\texttt{a}) \\
& & +\ \texttt{GTE}: \texttt{MSET}^2(\texttt{a})
\end{array}
$$

$$
\begin{array}{lll}
\texttt{eq} :: \texttt{VEC}^2(\texttt{a}) & \Longleftrightarrow & \ \texttt{EQ}: \texttt{a} \\
& & +\ \texttt{NE}: \texttt{PERM}^2(\texttt{a})
\end{array}
$$

$$\mathbb{Z}(\texttt{n}\times\texttt{m}) \Longleftrightarrow \mathbb{Z}_n \times \mathbb{Z}_m$$

Generalizing ‘fromTris’

```
smart[ℤ4] id   :: ℤ4 ⇒ ℤ4

  (0,0) (0,1) (0,2) (0,3)
  (1,0) (1,1) (1,2) (1,3)
  (2,0) (2,1) (2,2) (2,3)
  (3,0) (3,1) (3,2) (3,3)
```

Generalizing 'fromTris'

```
smart[ℤ4] id  :: ℤ4 ⇒ ℤ4

  (0,0) (0,1) (0,2) (0,3)
  (1,0) (1,1) (1,2) (1,3)
  (2,0) (2,1) (2,2) (2,3)
  (3,0) (3,1) (3,2) (3,3)

mapRes compare' (smart[ℤ4] id) :: ℤ4 ⇒ LT+EQ+GT

  EQ    LT    LT    LT
  GT    EQ    LT    LT
  GT    GT    EQ    LT
  GT    GT    GT    EQ
```

Generalizing 'fromTris'

```
mapRes compare' (smart[ℤ4] id) :: ℤ4 ⇒ LT+EQ+GT

  EQ    LT    LT    LT
  GT    EQ    LT    LT
  GT    GT    EQ    LT
  GT    GT    GT    EQ

mapRes compare (smart[ℤ4] id)  :: ℤ4 ⇒ COMPARE(ℤ4)

  EQ:0       LT:(0,1)  LT:(0,2)  LT:(0,3)
  GT:(0,1)   EQ:1      LT:(1,2)  LT:(1,3)
  GT:(0,2)   GT:(1,2)  EQ:2      LT:(2,3)
  GT:(0,3)   GT:(1,3)  GT:(2,3)  EQ:3
```

Generalizing ‘fromTris’

$\texttt{mapRes compare (smart[}\mathbb{Z}_4\texttt{] id)} :: \mathbb{Z}_4 \Rightarrow \texttt{COMPARE(}\mathbb{Z}_4\texttt{)}$

```
EQ:0      LT:(0,1)  LT:(0,2)  LT:(0,3)
GT:(0,1)  EQ:1      LT:(1,2)  LT:(1,3)
GT:(0,2)  GT:(1,2)  EQ:2      LT:(2,3)
GT:(0,3)  GT:(1,3)  GT:(2,3)  EQ:3
```