Array Types for a Graph Processing Language

Mark Tullsen and Matt Sottile

Galois, Inc.

GABB, May 2016

Outline

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
- 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 bijections and views
- Sparse/associative arrays
- First class index spaces

HAL (Hierarchical Array Language)

- Declarative
- A very expressive type system
  - captures more structure
  - provides more laws
  - sums!
  - allows bijections and views
- Sparse/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 bijections and views
- Sparse/associative arrays
- First class index spaces

CAVEAT:

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

A simple matrix, *m*

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

$m :: \mathbb{Z}_4 \times \mathbb{Z}_4 \Rightarrow \mathbb{Z}$

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

m_1 = arr[$\mathbb{Z}_4 \times \mathbb{Z}_4$] [0 0 0 0
0 1 0 0
0 0 2 0
0 0 0 3]

$|m_1| = 1+4\times4$

m_2 = unnest(arr[$\mathbb{Z}_4$]
[arr[$\mathbb{Z}_4$] [0 0 0 0]
, arr[$\mathbb{Z}_4$] [0 1 0 0]
, arr[$\mathbb{Z}_4$] [0 0 2 0]
, arr[$\mathbb{Z}_4$] [0 0 0 3]
])

$|m_2| = 1+4\times4$

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

Upper triangular, all zeros

$$\begin{matrix} \emptyset & \emptyset & \emptyset \\ & \emptyset & \emptyset \\ & & \emptyset \end{matrix}$$

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

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

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

Upper triangular, all zeros

$$\begin{matrix} \emptyset & \emptyset & \emptyset \\ & \emptyset & \emptyset \\ & & \emptyset \end{matrix}$$

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

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

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

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

(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

$$\begin{matrix} \emptyset & \emptyset & \emptyset \\ & \emptyset & \emptyset \\ & & \emptyset \end{matrix}$$

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

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

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

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

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

A sequence on the diagonal

```
0
   1
      2
         3
```

```
d :: ℤ4 ⇒ ℤ4
```

```
d1 = arr[ℤ4] [0 1 2 3]
```

```
|d1| = 1+4
```

```
d2 = smart[ℤ4] id
```

```
|d2| = 1
```

Combining . . . to define *m* in HAL

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

```
m :: VEC²(ℤ₄) ⇒ ℤ
```

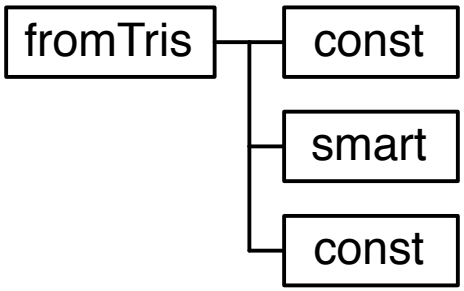
```
m₁ = arr[ℤ₄×ℤ₄] [0 0 0 0 0 1 ...]        |m₁| = 1+4×4

m₃ = fromTris ( const[SET²(ℤ₄)] 0        |m₃| = 1+1+1+1
              , smart[ℤ₄]       id
              , const[SET²(ℤ₄)] 0 )
```

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

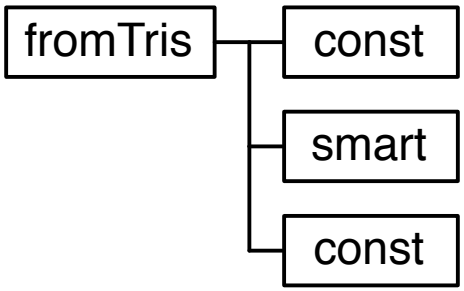
smart/const/... are represented by “tagged values”

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



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

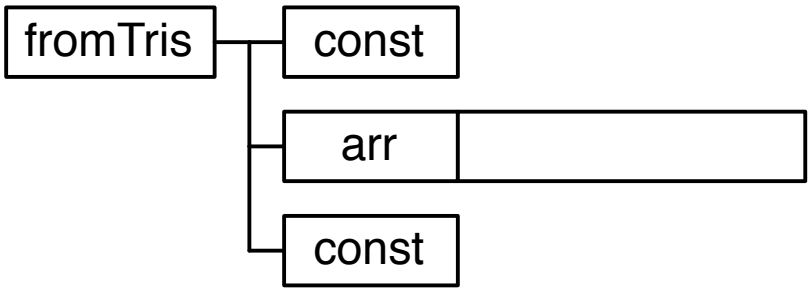
```
fromTris( const.., smart.., const..)          || = 1+1+1+1
```



- 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
```

Compact and Expanded Arrays, Mixed

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

```
m2 :: VEC²(ℤ₄) ⇒ ℤ
```

```
m2 = fromTris ( arr[SET²(ℤ₄)]    [5 9 1 2 5 3]
              , const[ℤ₄]        0
              , const[SET²(ℤ₄)] 0
              )
```

Compact and Expanded Arrays, Mixed

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

```
m2 :: VEC^2(Z_4) ⇒ Z
```

```
m2 = fromTris ( arr[SET^2(Z_4)]    [5 9 1 2 5 3]
              , const[Z_4]         0
              , const[SET^2(Z_4)]  0
              )
```

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

A Triangular Matrix

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

```
m3 :: SET^2(ℤ_4) ⇒ ℤ
```

```
m3 = arr[SET^2(ℤ_4)] [5 9 1 2 5 3]
```

```
|m3| = 1 + (4 choose 2) = 1 + 6
```

Partial Arrays

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

`{1: 2, 5: 3, ...}`

$$\begin{aligned} \texttt{a1} &:: \mathbb{Z}_{50} \rightharpoonup \mathbb{Z} \\ |\texttt{a1}| &= 1+2\times\texttt{nnz(a1)} \end{aligned}$$

`{1: 2, 5: 3, ...}` *DFLT* `0`

$$\begin{aligned} \texttt{a2} &:: \mathbb{Z}_{50} \Rightarrow \mathbb{Z} \\ |\texttt{a2}| &= 1+2\times\texttt{nnz(a1)} \end{aligned}$$

- `nnz` - number of non zeros, loosely

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)	(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)

A Very 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)

A Very 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

A Very 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

A Very 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

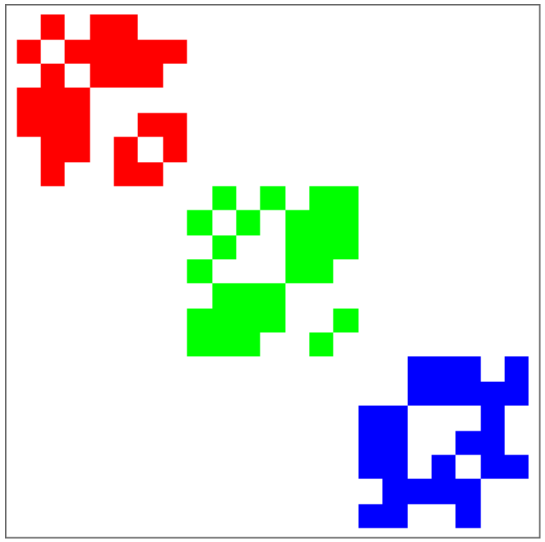
{0,3}	{0,2}	{0,1}	0
{1,3}	{1,2}	1	{0,1}
{2,3}	2	{1,2}	{0,2}
3	{2,3}	{1,3}	{0,3}

{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

A Diagonal Block Array

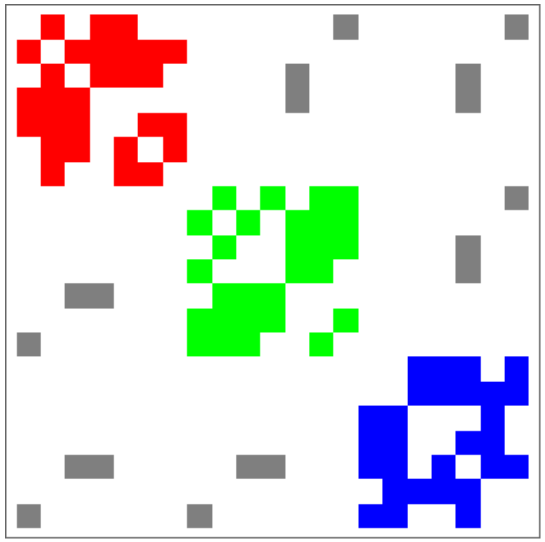


```
m1, m2, m3 :: Z7×Z7 ⇒ Z2

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
  - E.g., tree -> tree + tree
  - These help us write divide-and-conquer algorithms
- Graphs are not one of these
  - Though we do have some divide-and-conquer schemes, such as map-reduce.

One Method to Divide-And-Conquer Graphs

Incomplete Graphs:

One Method to Divide-And-Conquer Graphs

Incomplete Graphs:

- . . . *generalize* graphs

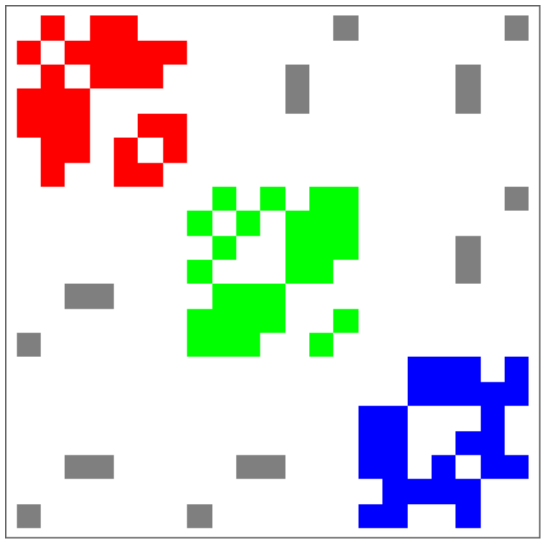
One Method to Divide-And-Conquer Graphs

Incomplete Graphs:

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

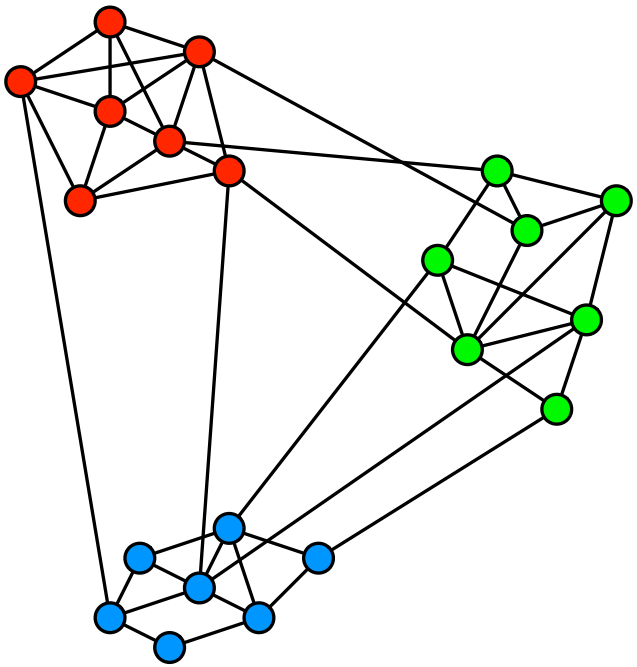
Incomplete Graphs Visualized

The adjacency matrix of graph:



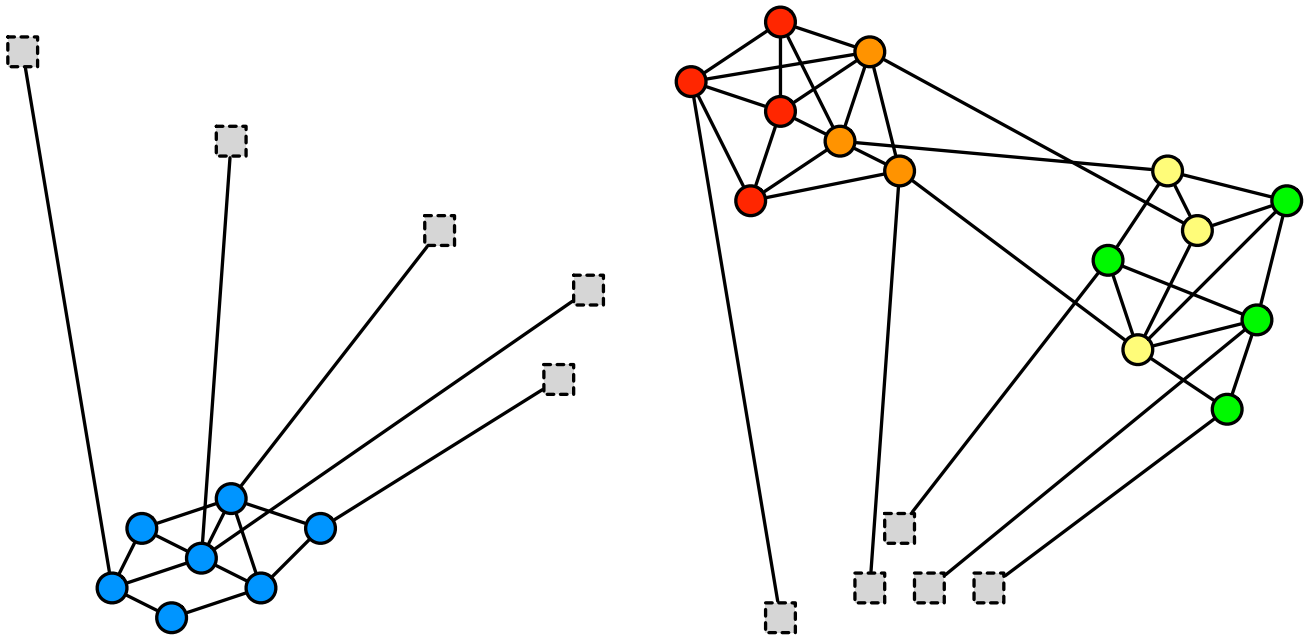
Incomplete Graphs Visualized

The graph:



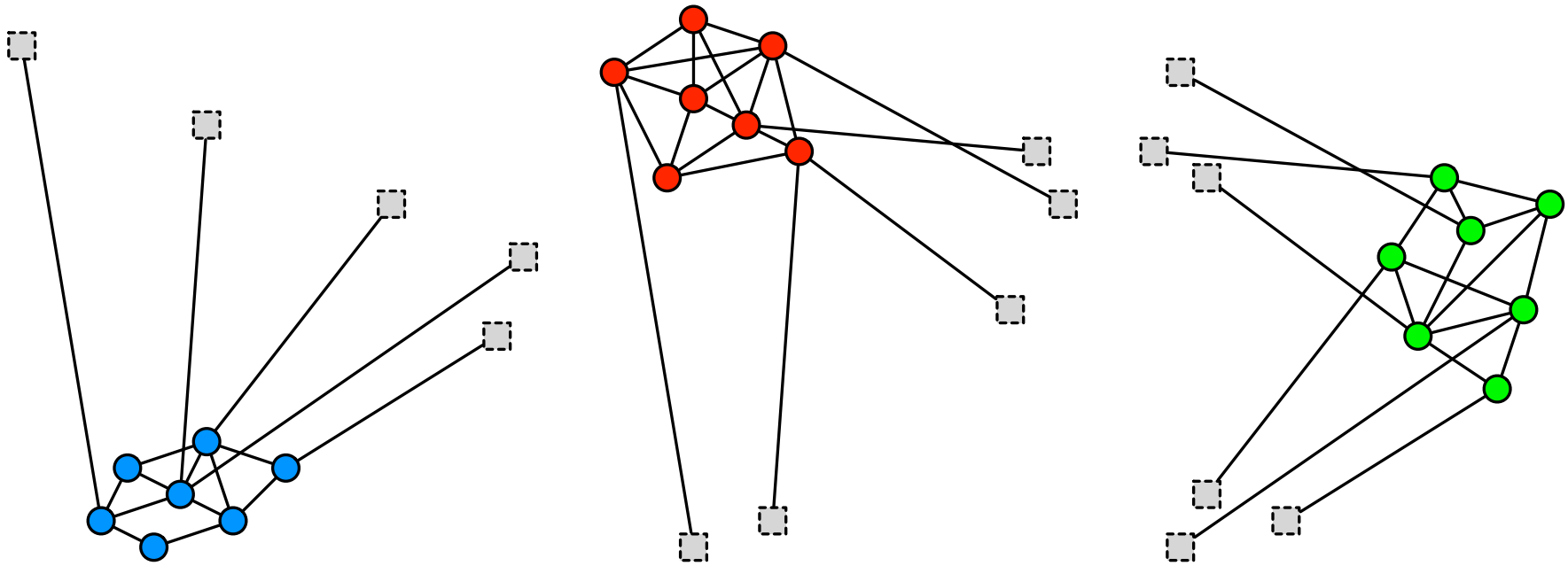
Incomplete Graphs Visualized

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



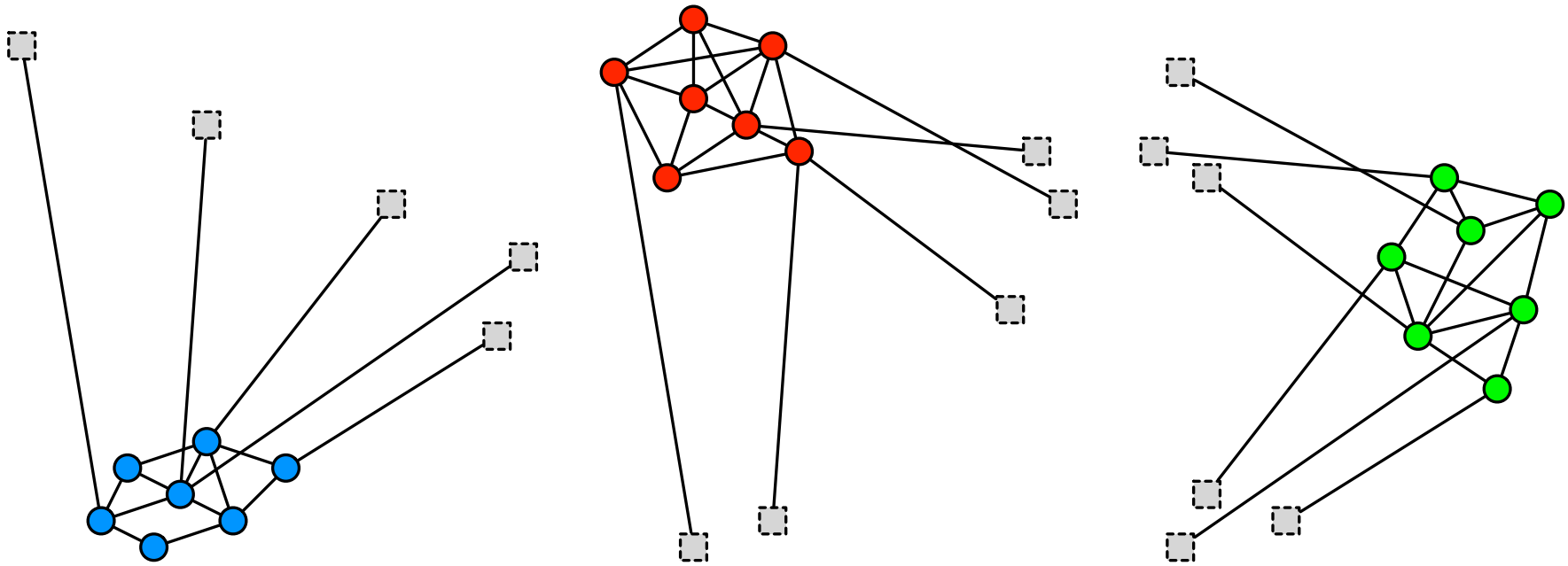
Incomplete Graphs Visualized

Extract green nodes, edges get “split in half”:



Incomplete Graphs Visualized

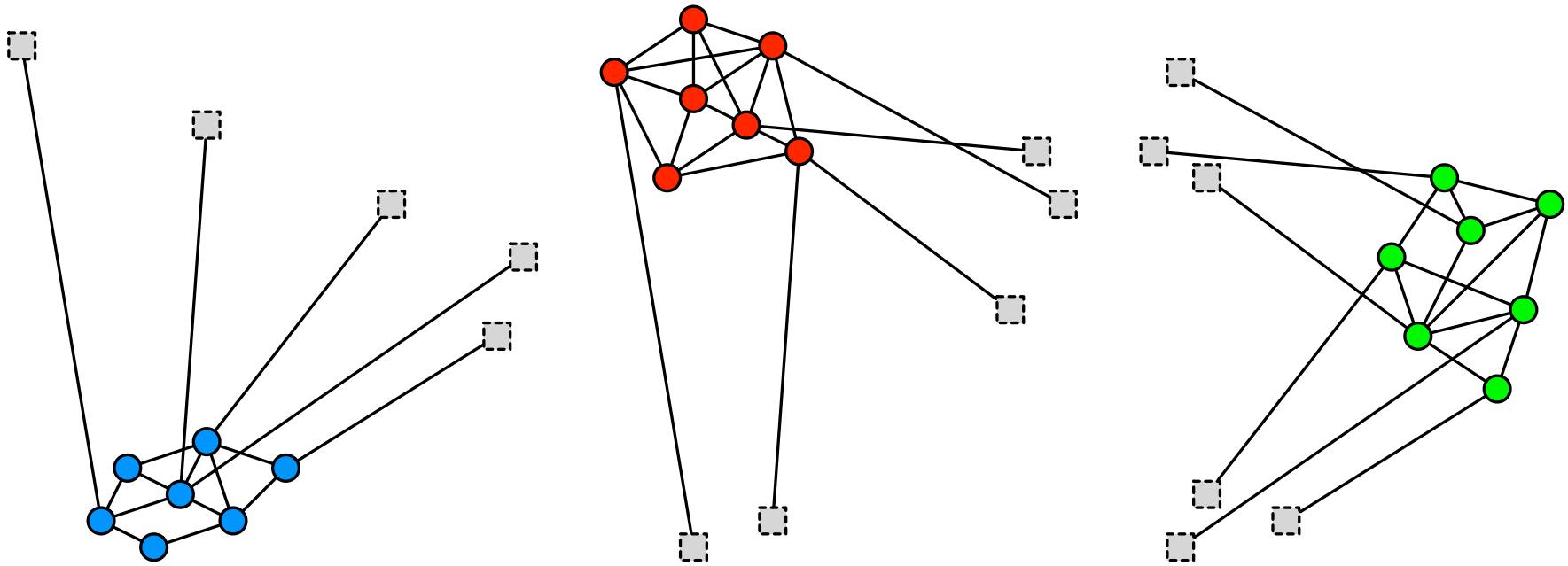
Extract green nodes, edges get “split in half”:



- 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

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
  - bijections give us views & in-place updates

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

$$\{1:\ 2,\ 5:\ 3,\ 7:\ 0,\ ...\}\ \underline{DFLT}\ 0 \quad :: \quad \mathbb{Z}_{50} \Rightarrow \mathbb{Z}\ ,\ ||=2\times nnz$$

Partial Arrays: representation and partiality are orthogonal

- sparse rep. of total arrays

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

- a flattened rep. of partial arrays

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

Partial Arrays: representation and partiality are orthogonal

- sparse rep. of total arrays

$$\{1:\ 2,\ 5:\ 3,\ 7:\ 0,\ ...\}\ \underline{DFLT}\ 0 \quad :: \quad \mathbb{Z}_{50} \Rightarrow \mathbb{Z}\ ,\ ||=2\times nnz$$

- a flattened rep. of partial arrays

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

$$\hat{}\hat{}\hat{}\hat{}\hat{}\hat{} \quad \mathbb{Z}_{50} \boxdot\!\!\Rightarrow \mathbb{Z} \quad \hat{}\hat{}\hat{}\hat{}\hat{}\hat{}\hat{}\hat{}$$

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, ...})` `::` $\mathbb{Z}_{50} \Rightarrow 1 + \mathbb{Z}$

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, ...})` `::` $\mathbb{Z}_{50} \boxdot\!\!\Rightarrow \mathbb{Z}$ `, ||=1+50`

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

What is a graph?

Graphs parameterized by vertex and edge:

```
type DG v e = VEC²(v) □⇒ e     -- directed
type UG v e = MSET²(v) □⇒ e    -- undirected
```

No edge data:

```
type DG' v = VEC²(v) ⇒ ℤ₂      -- directed
type UG' v = MSET²(v) ⇒ ℤ₂     -- 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)

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

compare (and other bijections on indices)

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

$$\texttt{less} :: \texttt{VEC}^2\texttt{(a)} \iff \texttt{LT: SET}^2\texttt{(a)} + \texttt{GTE: MSET}^2\texttt{(a)}$$

compare (and other bijections on indices)

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

$$\texttt{less} :: \texttt{VEC}^2\texttt{(a)} \iff \texttt{LT: SET}^2\texttt{(a)} + \texttt{GTE: MSET}^2\texttt{(a)}$$

$$\texttt{eq} :: \texttt{VEC}^2\texttt{(a)} \iff \texttt{EQ: a} + \texttt{NE: PERM}^2\texttt{(a)}$$

compare (and other bijections on indices)

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

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

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

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

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’

```
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