Verigraph
A tool for analysis of Graph Transformation

Leila Ribeiro, Rodrigo Machado, Andrei Costa, Jonas Bezerra, Guilherme Azzi, Leonardo Rodrigues

.Inf INSTITUTO DE INFORMÁTICA UFRGS

Motivation

- Why another tool?
  - Different types of graphs
  - Second-order GTs
  - Other approaches: SqPO, AGREE
  - Correctness

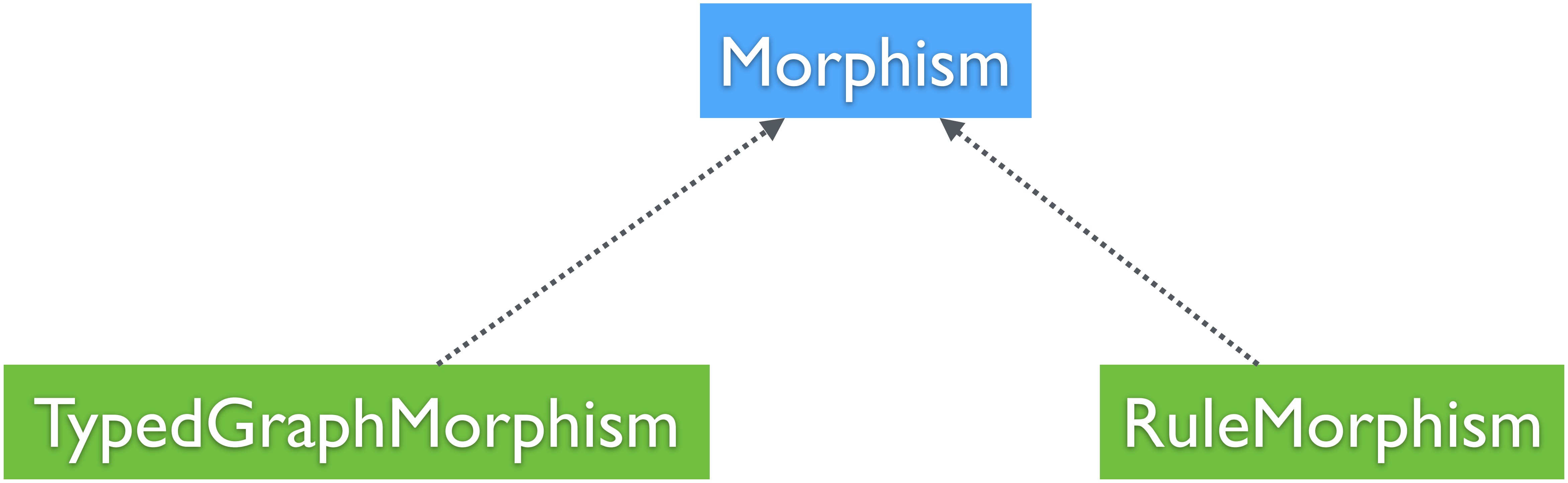
Verigraph

- A tool for **analysis** of GT
- Architected for **flexibility** and **proximity to theory**
- Written in **Haskell**
- Current features
  - Static analysis : *critical pairs/sequences, concurrent rules*
  - *Typed graphs*
  - *Second order rewriting/analysis*

.inf INSTITUTO DE INFORMÁTICA UFRGS

Architected for Extensibility

Interfaces based on the theory of adhesive HLRS



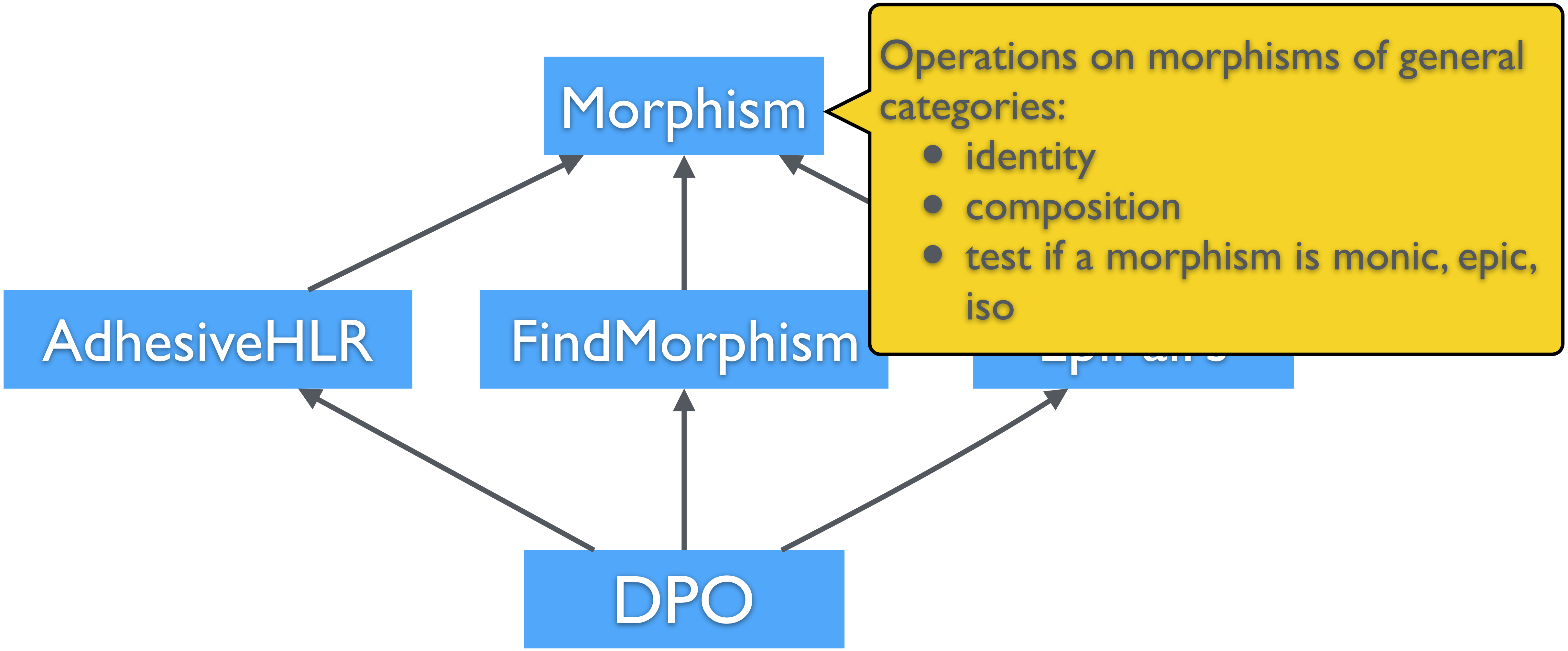
Morphism

TypedGraphMorphism

RuleMorphism

Architected for Extensibility

Interfaces based on the theory of adhesive HLRS

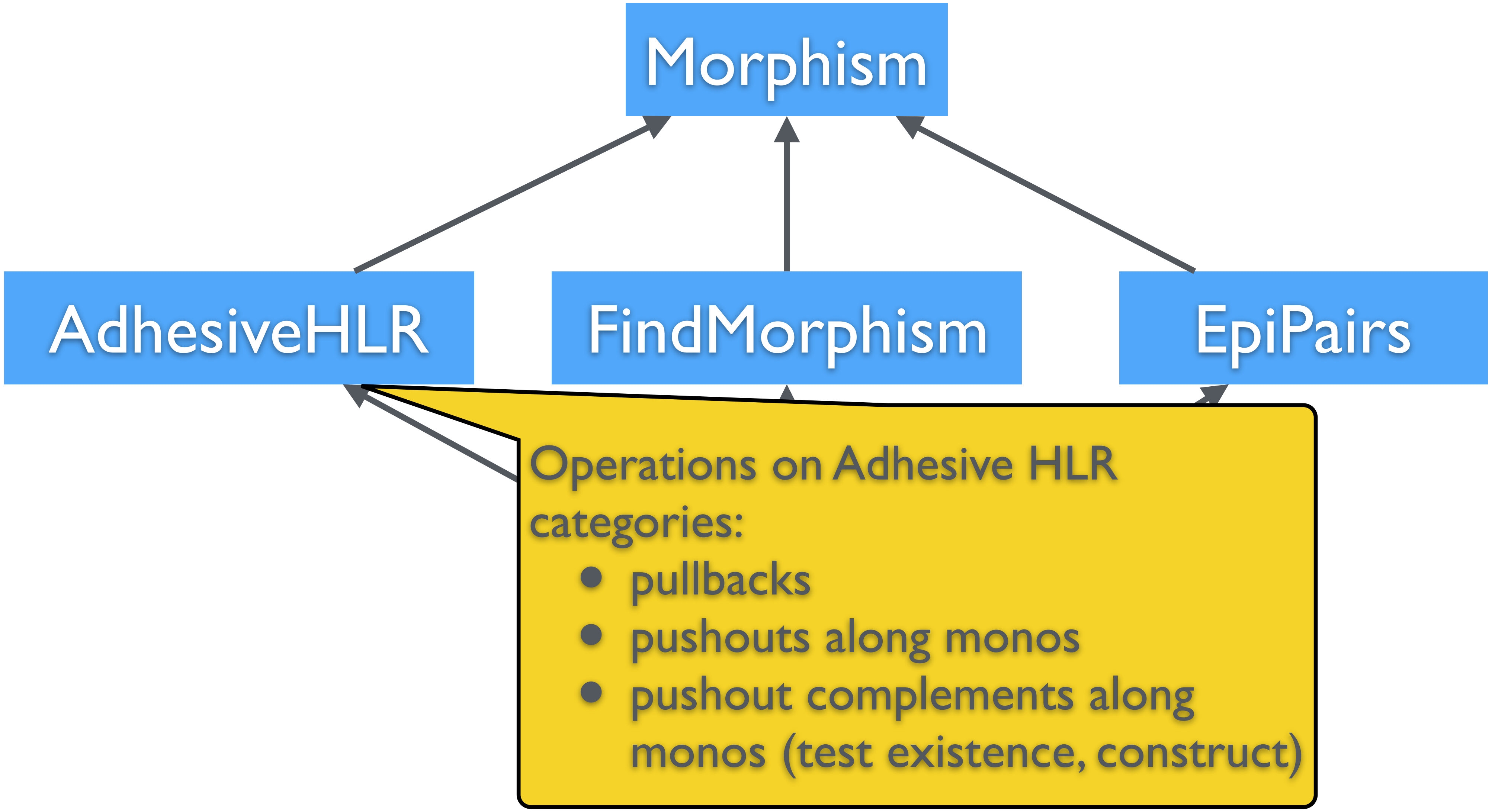


Operations on morphisms of general categories:

- identity
- composition
- test if a morphism is monic, epic, iso

Architected for Extensibility

Interfaces based on the theory of adhesive HLRS

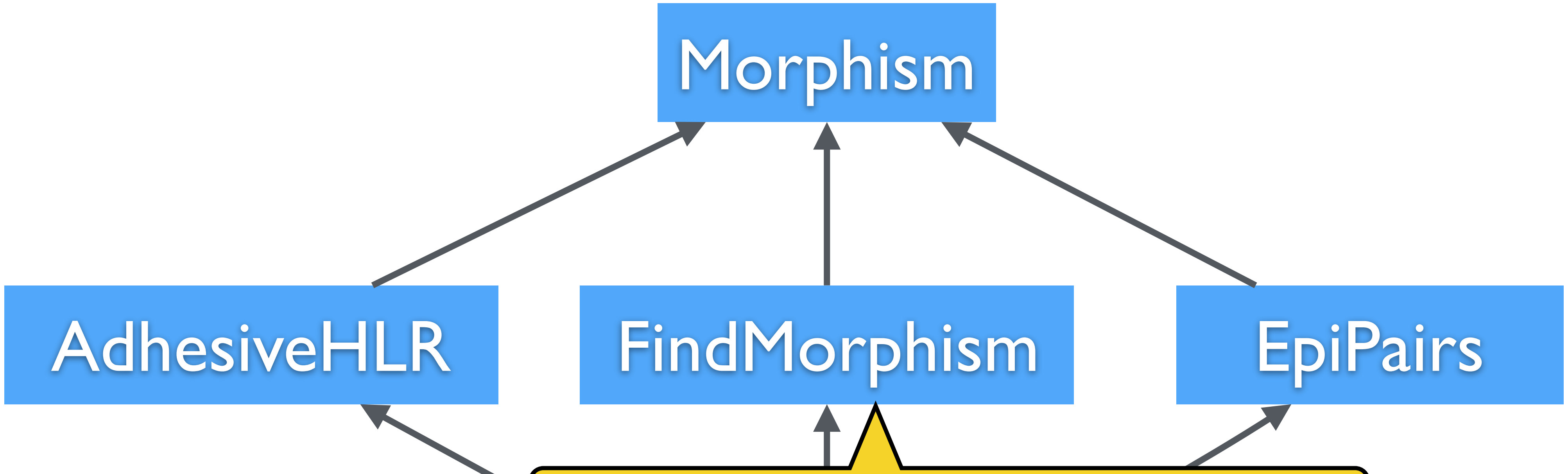


Operations on Adhesive HLR categories:

- pullbacks
- pushouts along monos
- pushout complements along monos (test existence, construct)

Architected for Extensibility

Interfaces based on the theory of adhesive HLRS

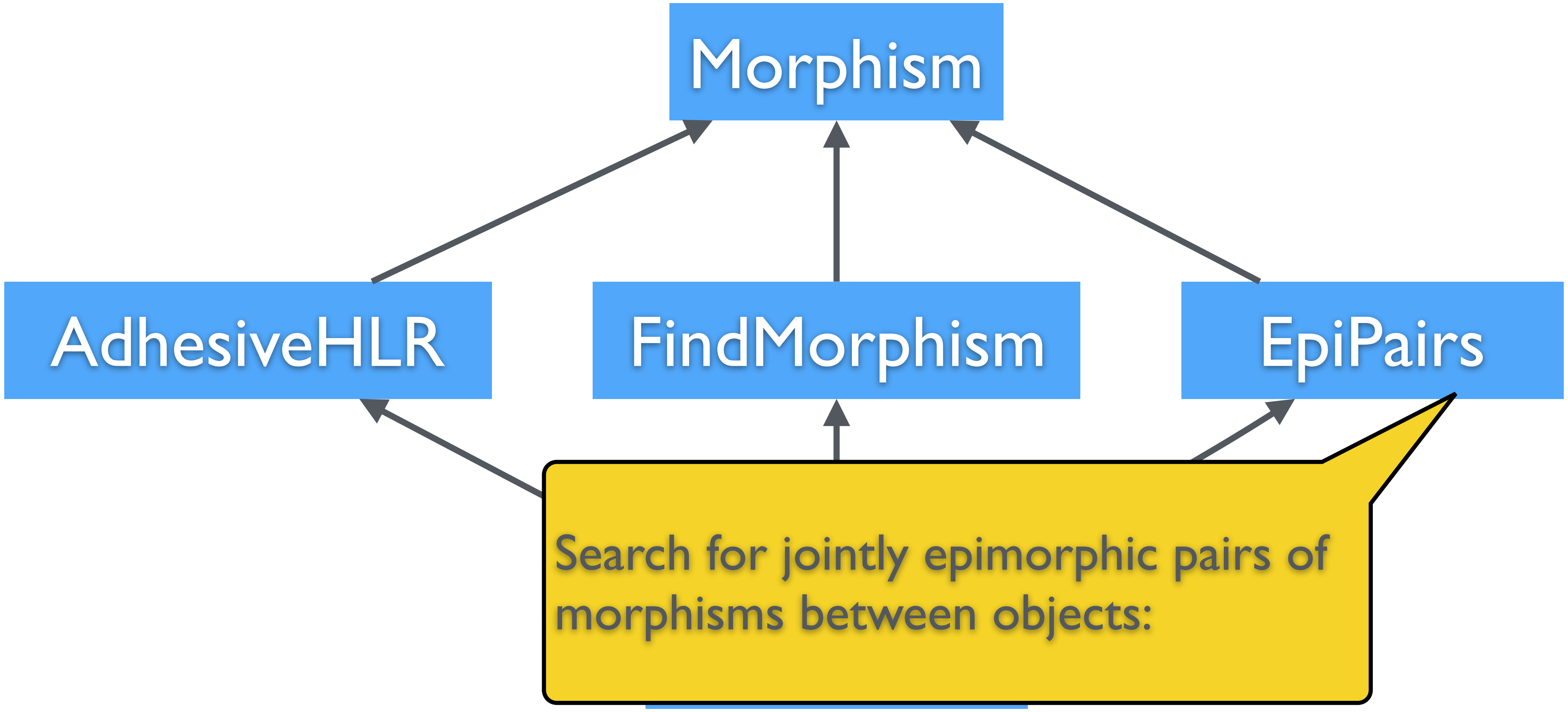


Search for morphisms between objects:

- monic
- epic
- iso
- arbitrary

Architected for Extensibility

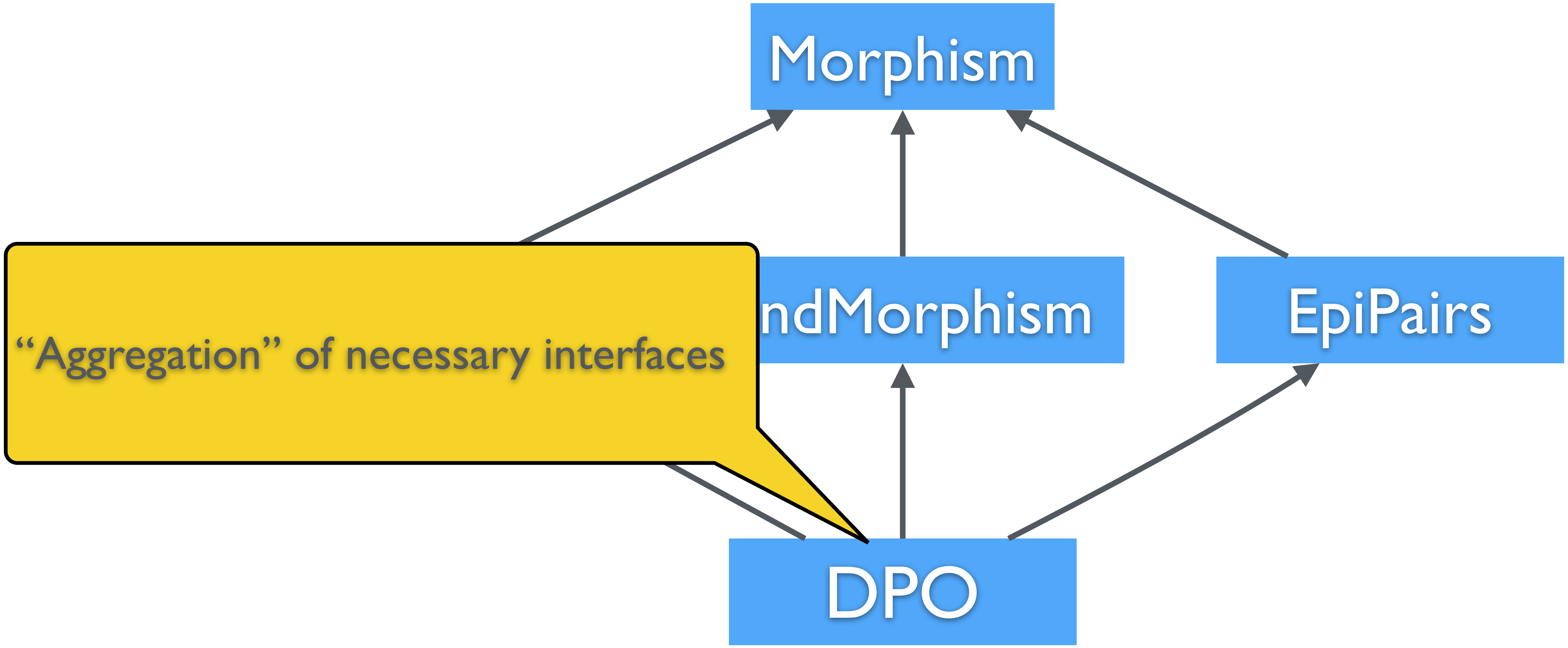
Interfaces based on the theory of adhesive HLRS



Search for jointly epimorphic pairs of morphisms between objects:

Architected for Extensibility

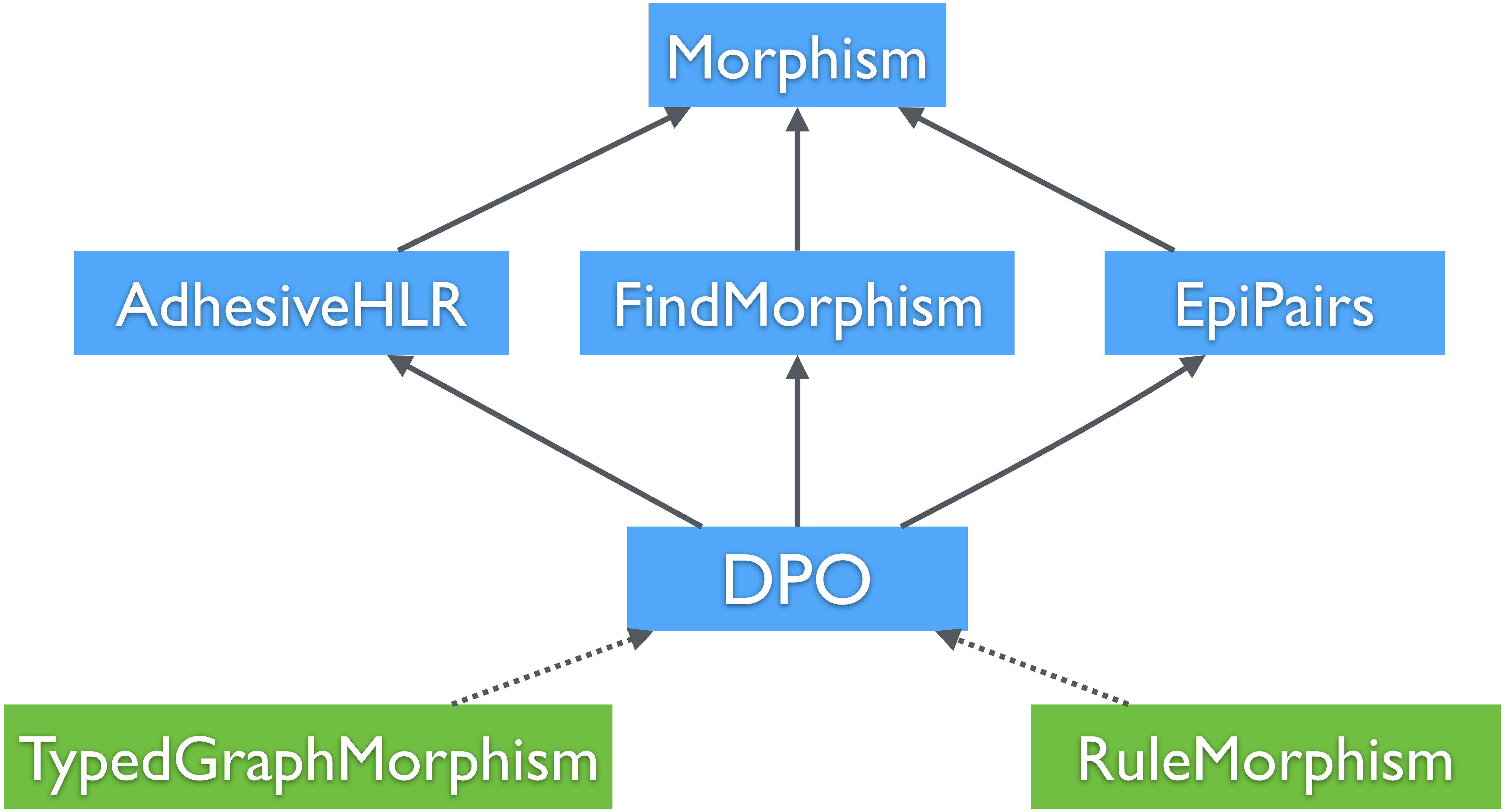
Interfaces based on the theory of adhesive HLRS



“Aggregation” of necessary interfaces

Architected for Extensibility

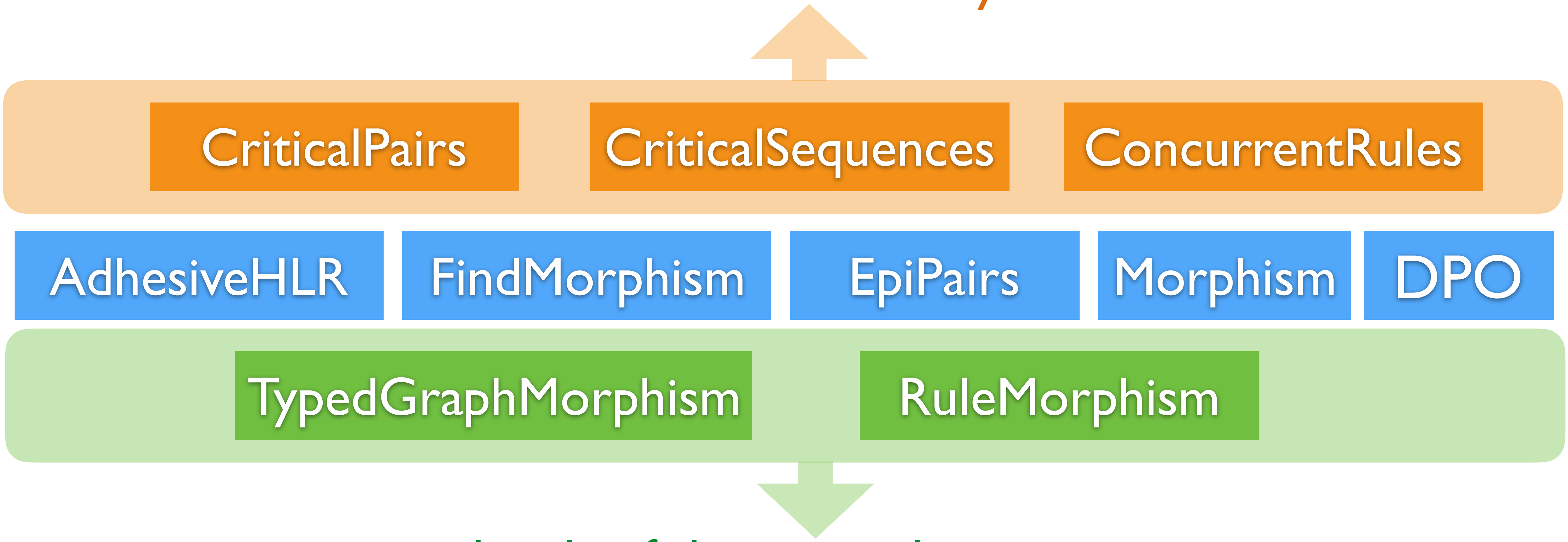
Interfaces based on the theory of adhesive HLRS



Architected for Flexibility

Clear separation between abstract and concrete layers

independent of the underlying category
close to theory



CriticalPairs
CriticalSequences
ConcurrentRules

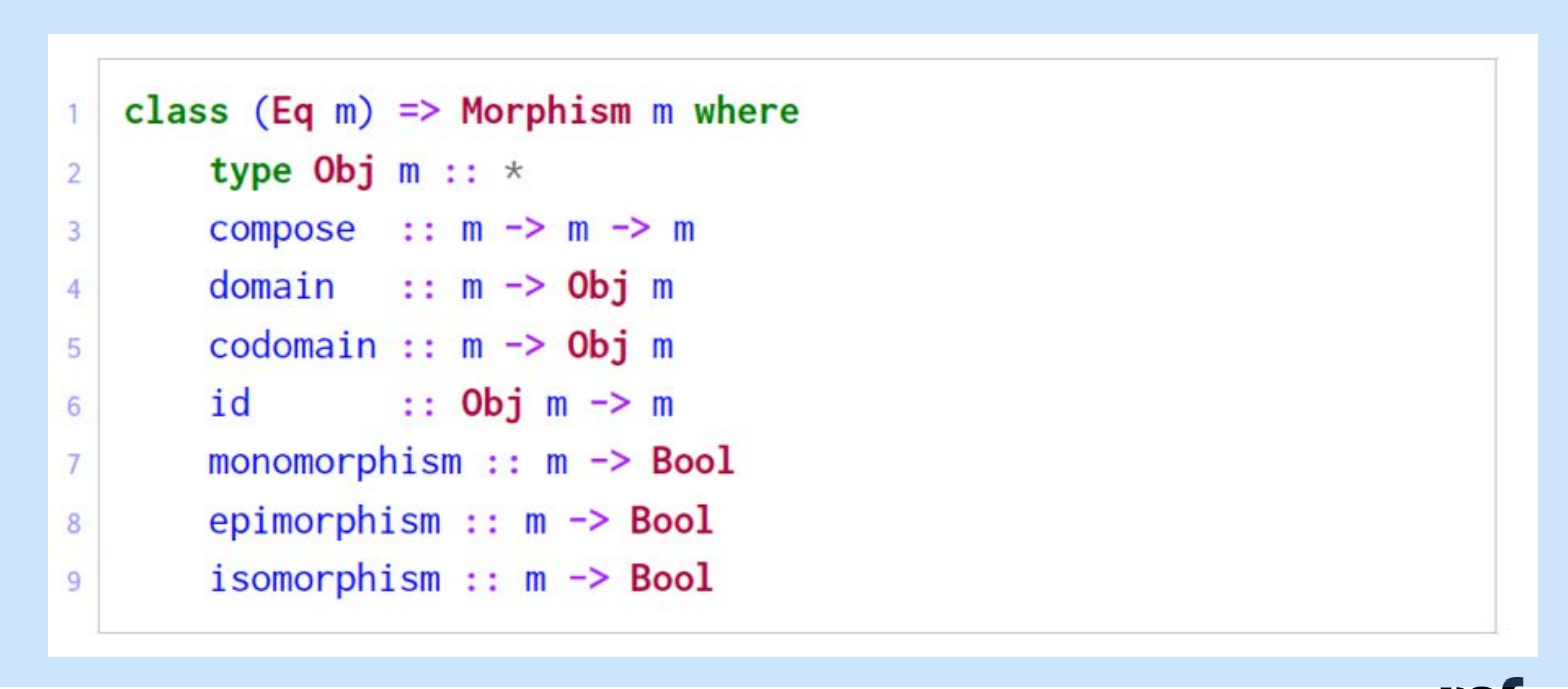
AdhesiveHLR
FindMorphism
EpiPairs
Morphism
DPO

TypedGraphMorphism
RuleMorphism

details of the particular category
optimization

Example: Interface Layer

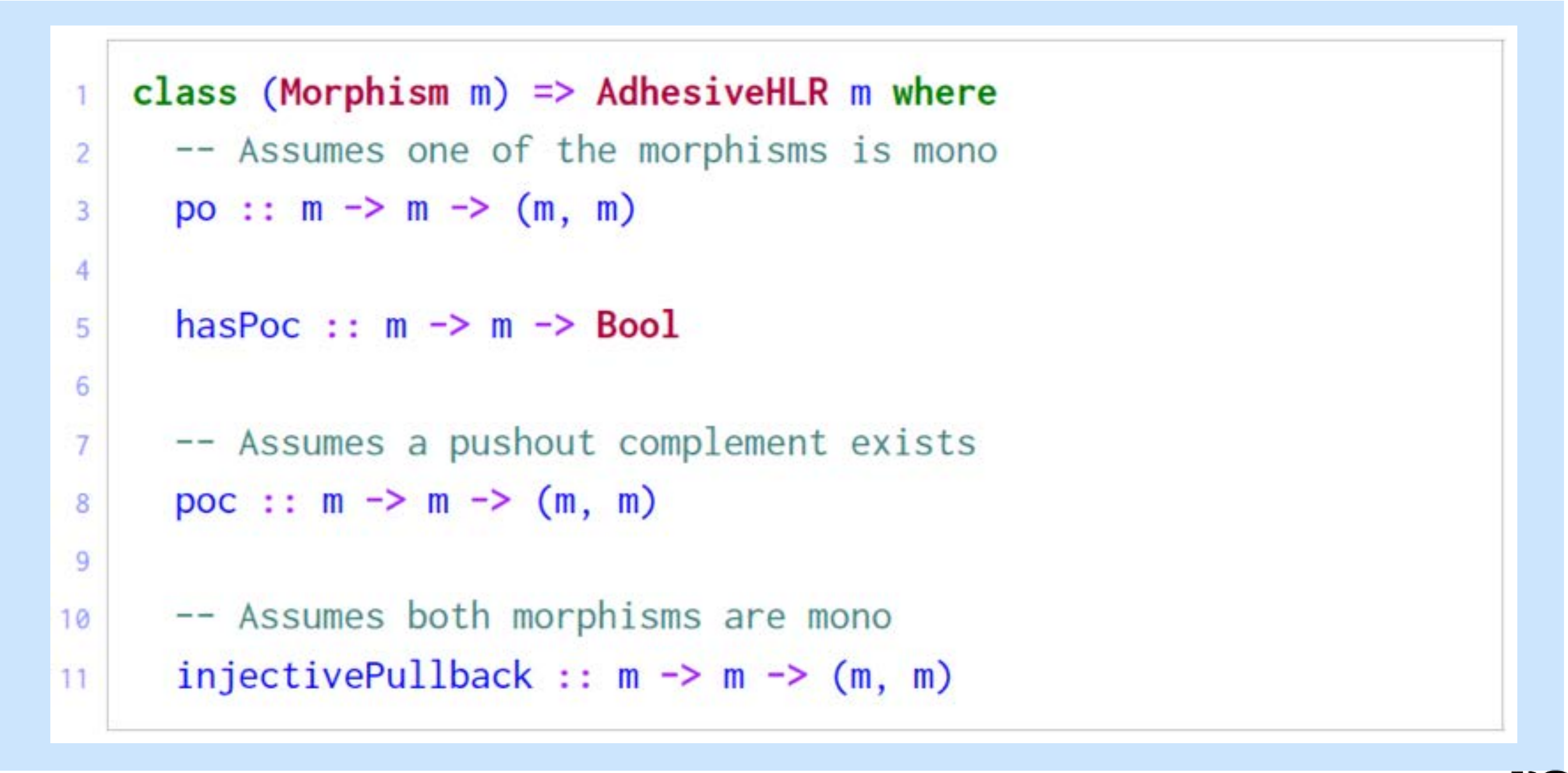
Type Class Morphism : class of morphisms of a category



```haskell
class (Eq m) => Morphism m where
    type Obj m :: *
    compose  :: m -> m -> m
    domain   :: m -> Obj m
    codomain :: m -> Obj m
    id       :: Obj m -> m
    monomorphism :: m -> Bool
    epimorphism :: m -> Bool
    isomorphism :: m -> Bool
```

Example: Interface Layer

Type Class AdhesiveHLR : morphisms of an adhesive category



```haskell
class (Morphism m) => AdhesiveHLR m where
  -- Assumes one of the morphisms is mono
  po :: m -> m -> (m, m)

  hasPoc :: m -> m -> Bool

  -- Assumes a pushout complement exists
  poc :: m -> m -> (m, m)

  -- Assumes both morphisms are mono
  injectivePullback :: m -> m -> (m, m)
```

Example: Abstract Layer

test for delete-use conflict

p1

$$\begin{tikzcd}
 & & N_1 & & N_2 & & \\
R_1 & K_1 \arrow[l, "r_1"'] \arrow[r, "l_1"] \arrow[d, "k_1"'] & L_1 \arrow[u, "n_1"] \arrow[dr, "m_1"'] & & L_2 \arrow[u, "n_2"] \arrow[dl, "m_2"] \arrow[dlll, dotted, bend right, "!h_{21}"'] & K_2 \arrow[l, "l_2"'] \arrow[r, "r_2"] & R_2 \\
 & D_1 \arrow[rr, "d_1"'] & & G & & &
\end{tikzcd}$$

```
isDeleteUse :: DPO m => Production m -> (m, m) -> Bool
isDeleteUse p1 (m1,m2) =
```

Example: Abstract Layer

test for delete-use conflict

p1

$$\begin{tikzcd}
& & N_1 & & N_2 & & \\
R_1 & K_1 \arrow[l, "r_1"'] \arrow[r, "l_1"] \arrow[d, "k_1"'] & L_1 \arrow[u, "n_1"] \arrow[dr, "m_1"] & & L_2 \arrow[u, "n_2"'] \arrow[dl, "m_2"] \arrow[dlll, dotted, bend right, "!h_{21}"'] & K_2 \arrow[l, "l_2"'] \arrow[r, "r_2"] & R_2 \\
& D_1 \arrow[rr, "d_1"'] & & G & & &
\end{tikzcd}$$

```
isDeleteUse :: DPO m => Production m -> (m, m) -> Bool
isDeleteUse p1 (m1,m2) =
 where
         (_,d1) = calculatePushoutComplement m1 (getLHS p1)
```

Example: Abstract Layer

test for delete-use conflict

p1

$$\begin{tikzcd}
 & & N_1 & & N_2 & & \\
R_1 & K_1 \arrow[l, "r_1"'] \arrow[r, "l_1"] \arrow[d, "k_1"'] & L_1 \arrow[u, "n_1"] \arrow[dr, "m_1"'] & & L_2 \arrow[u, "n_2"'] \arrow[dl, "m_2"] \arrow[dlll, dotted, bend right, "h_{21}"'] & K_2 \arrow[l, "l_2"'] \arrow[r, "r_2"] & R_2 \\
 & D_1 \arrow[rr, "d_1"'] & & G & & &
\end{tikzcd}$$

```
isDeleteUse :: DPO m => Production m -> (m, m) -> Bool
isDeleteUse p1 (m1,m2) =
 where
          (_,d1) = calculatePushoutComplement m1 (getLHS p1)
          candidates = findMorphisms (domain m2) (domain d1)
```

Example: Abstract Layer

test for delete-use conflict

p1

$$\begin{tikzcd}
 & & N_1 & & N_2 & & \\
R_1 & K_1 \arrow[l, "r_1"'] \arrow[r, "l_1"] \arrow[dd, "k_1"'] & L_1 \arrow[u, "n_1"] \arrow[dr, "m_1"] & & L_2 \arrow[u, "n_2"'] \arrow[dl, "m_2"'] \arrow[dlll, dotted, bend right, "h_{21}"'] & K_2 \arrow[l, "l_2"'] \arrow[r, "r_2"] & R_2 \\
 & & & G & & & \\
 & D_1 \arrow[urr, "d_1"'] & & & & &
\end{tikzcd}$$

=

```
isDeleteUse :: DPO m => Production m -> (m, m) -> Bool
isDeleteUse p1 (m1,m2) = null h21
 where
         (_,d1) = calculatePushoutComplement m1 (getLHS p1)
         candidates = findMorphisms (domain m2) (domain d1)
         h21 = filter (\x -> m2 == compose x d1) candidates
```

Example: Abstract Layer

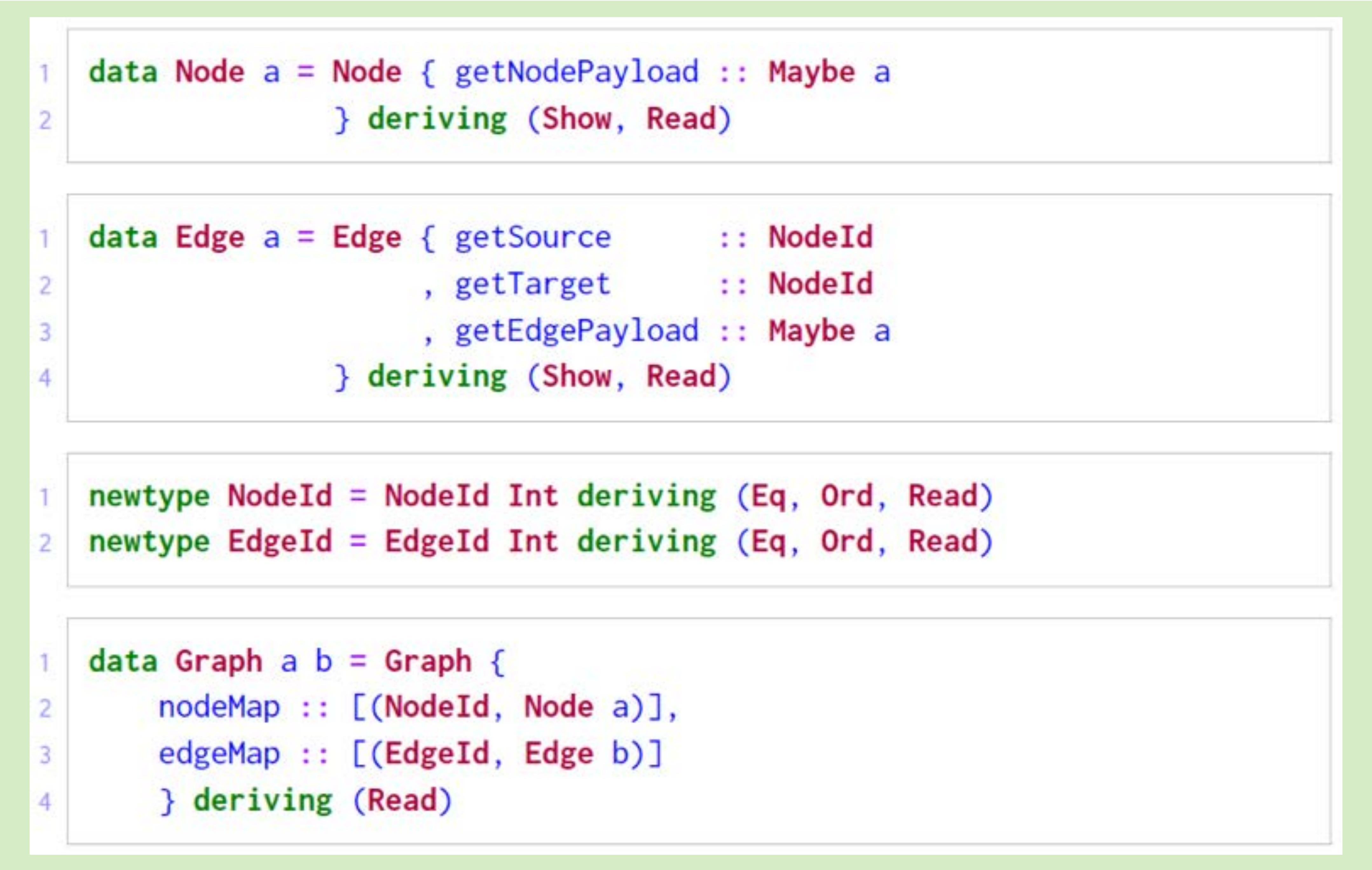
test for delete-use conflict

p1

$$\begin{tikzcd}
 & & N_1 & & N_2 & & \\
R_1 & K_1 \arrow[l, "r_1"'] \arrow[r, "l_1"] \arrow[d, "k_1"'] & L_1 \arrow[u, "n_1"] \arrow[dr, "m_1"'] & & L_2 \arrow[u, "n_2"'] \arrow[dl, "m_2"] \arrow[dlll, dotted, bend right, "!h_{21}"'] & K_2 \arrow[l, "l_2"'] \arrow[r, "r_2"] & R_2 \\
 & D_1 \arrow[rr, "d_1"'] & & G & & &
\end{tikzcd}$$

```
isDeleteUse :: DPO m => Production m -> (m, m) -> Bool
isDeleteUse p1 (m1,m2) = null h21
 where
          (_,d1) = calculatePushoutComplement m1 (getLHS p1)
          candidates = findMorphisms (domain m2) (domain d1)
          h21 = filter (\x -> m2 == compose x d1) candidates
```

Example: Concrete Layer



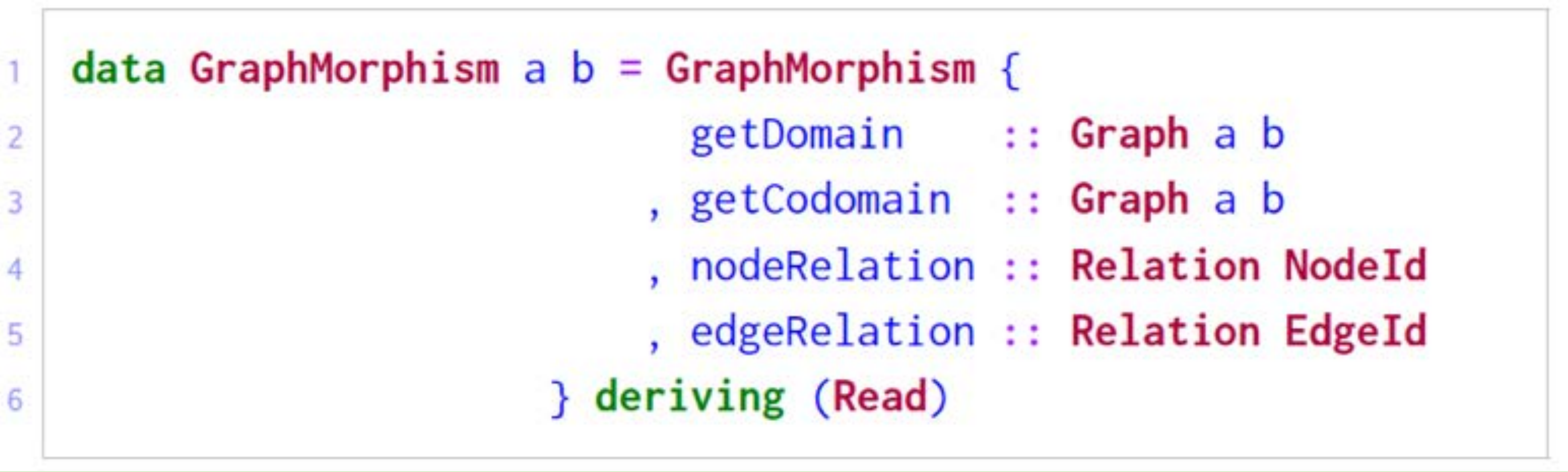
```haskell
data Node a = Node { getNodePayload :: Maybe a
              } deriving (Show, Read)
```

```haskell
data Edge a = Edge { getSource      :: NodeId
                   , getTarget      :: NodeId
                   , getEdgePayload :: Maybe a
              } deriving (Show, Read)
```

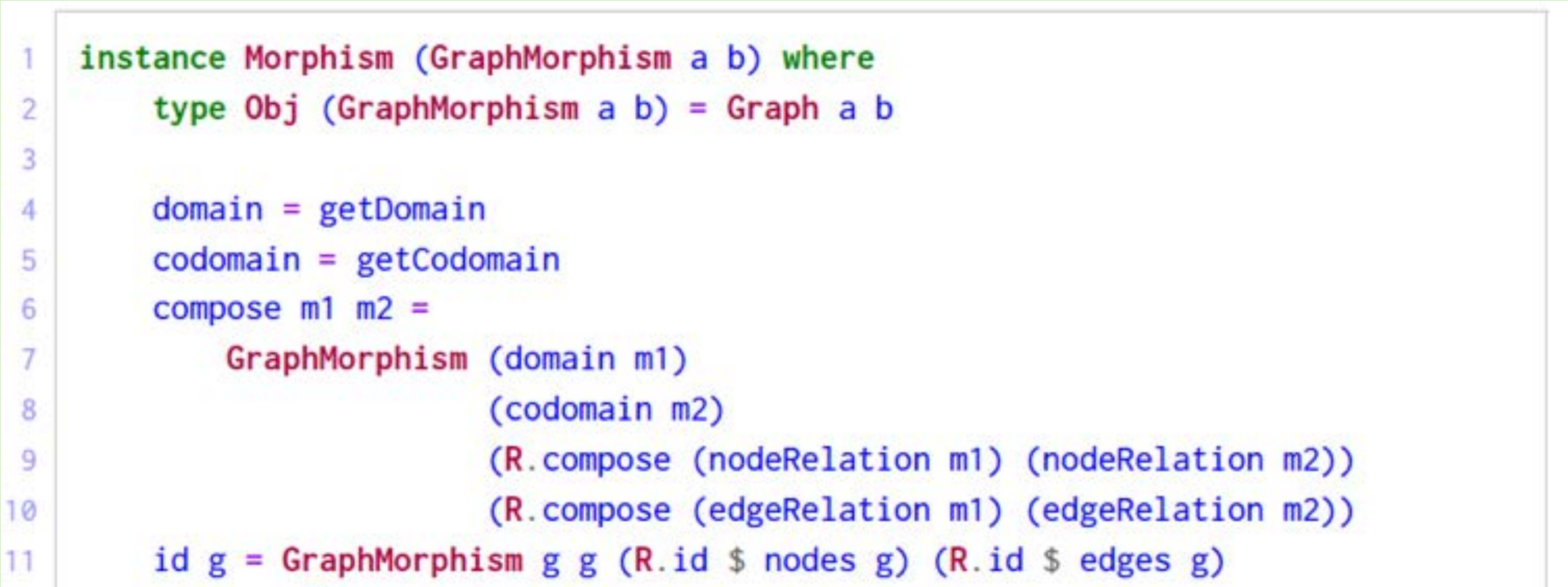
```haskell
newtype NodeId = NodeId Int deriving (Eq, Ord, Read)
newtype EdgeId = EdgeId Int deriving (Eq, Ord, Read)
```

```haskell
data Graph a b = Graph {
    nodeMap :: [(NodeId, Node a)],
    edgeMap :: [(EdgeId, Edge b)]
    } deriving (Read)
```

Example: Concrete Layer



```haskell
data GraphMorphism a b = GraphMorphism {
                             getDomain      :: Graph a b
                           , getCodomain    :: Graph a b
                           , nodeRelation   :: Relation NodeId
                           , edgeRelation   :: Relation EdgeId
                         } deriving (Read)
```



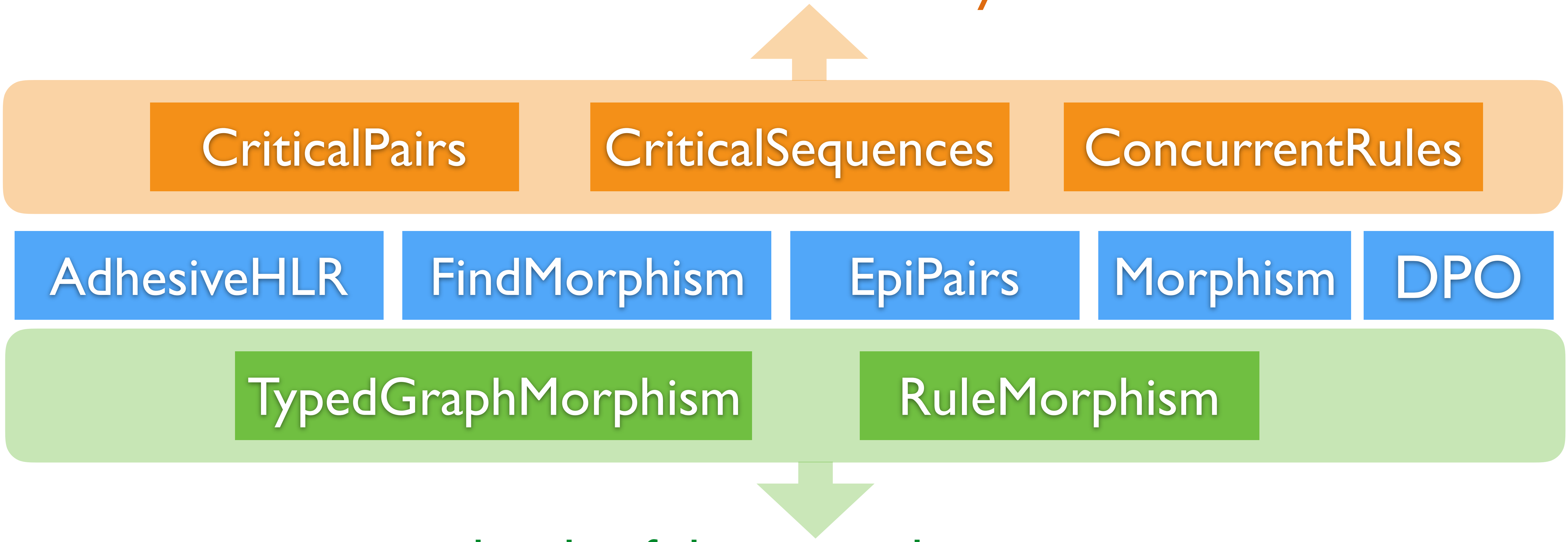
```haskell
instance Morphism (GraphMorphism a b) where
    type Obj (GraphMorphism a b) = Graph a b

    domain = getDomain
    codomain = getCodomain
    compose m1 m2 =
        GraphMorphism (domain m1)
                      (codomain m2)
                      (R.compose (nodeRelation m1) (nodeRelation m2))
                      (R.compose (edgeRelation m1) (edgeRelation m2))
    id g = GraphMorphism g g (R.id $ nodes g) (R.id $ edges g)
```

Architected for Flexibility

Clear separation between abstract and concrete layers

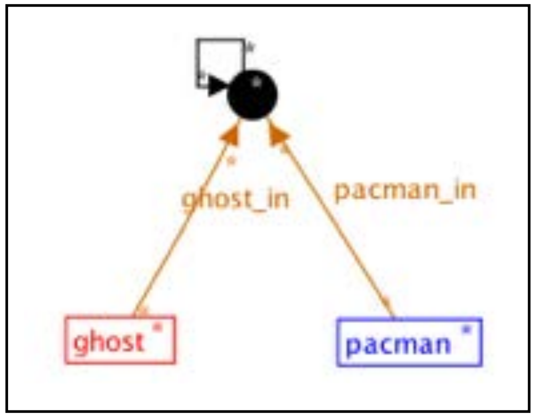
independent of the underlying category
close to theory

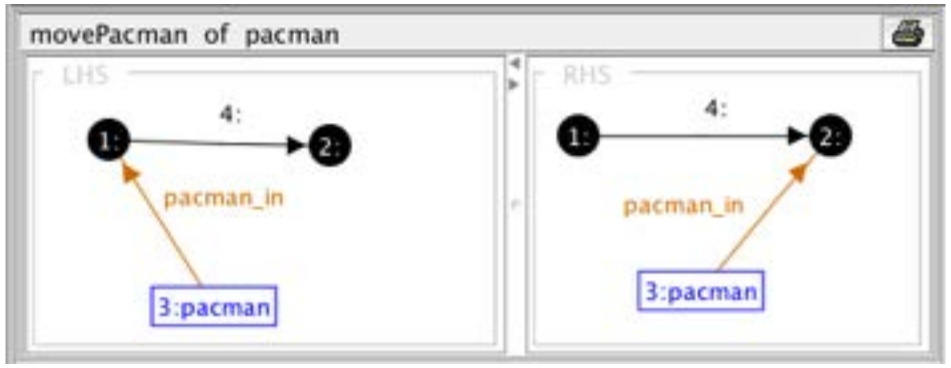


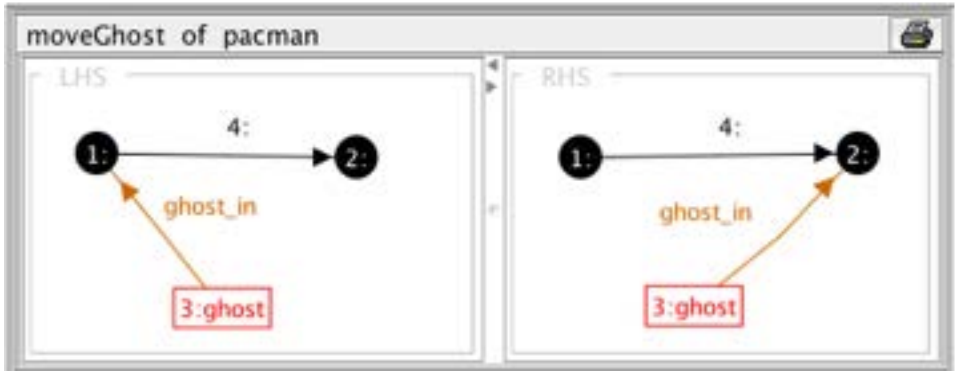
details of the particular category
optimization

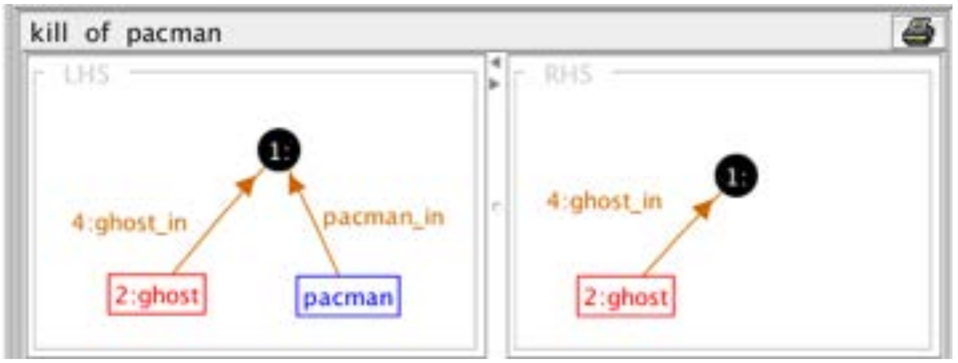
Second Order Example

Type Graph

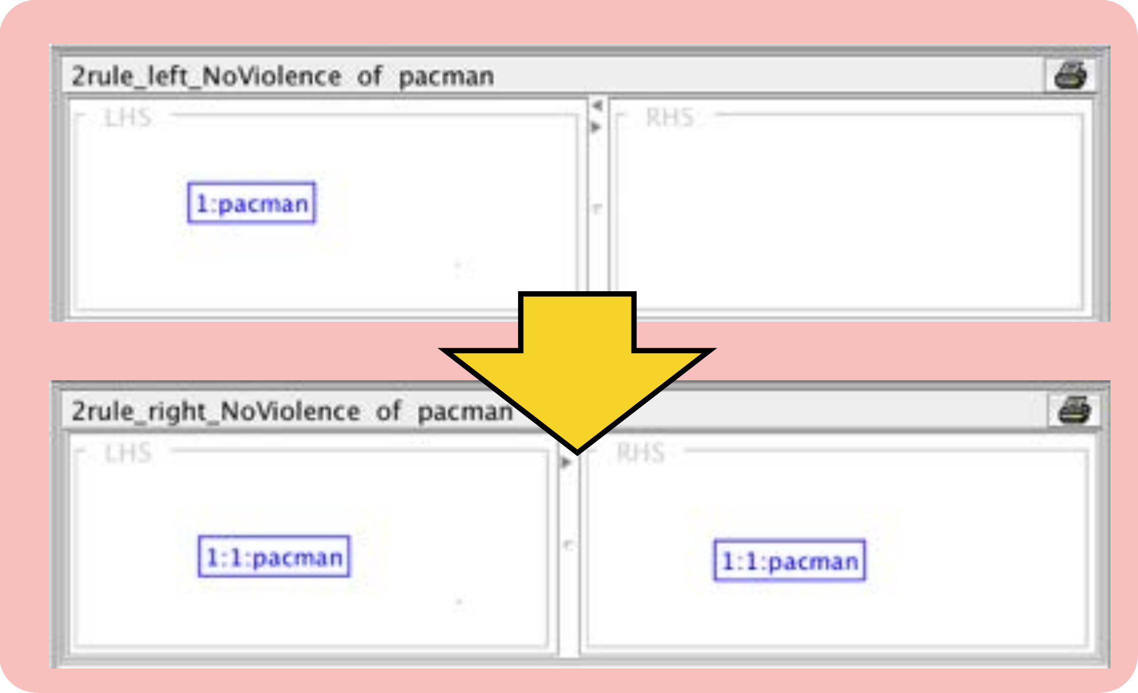








Second Order Example


2rule_left_NoViolence of pacman
LHS
RHS
1:pacman
2rule_right_NoViolence of pacman
LHS
RHS
1:1:pacman
1:1:pacman

LHS

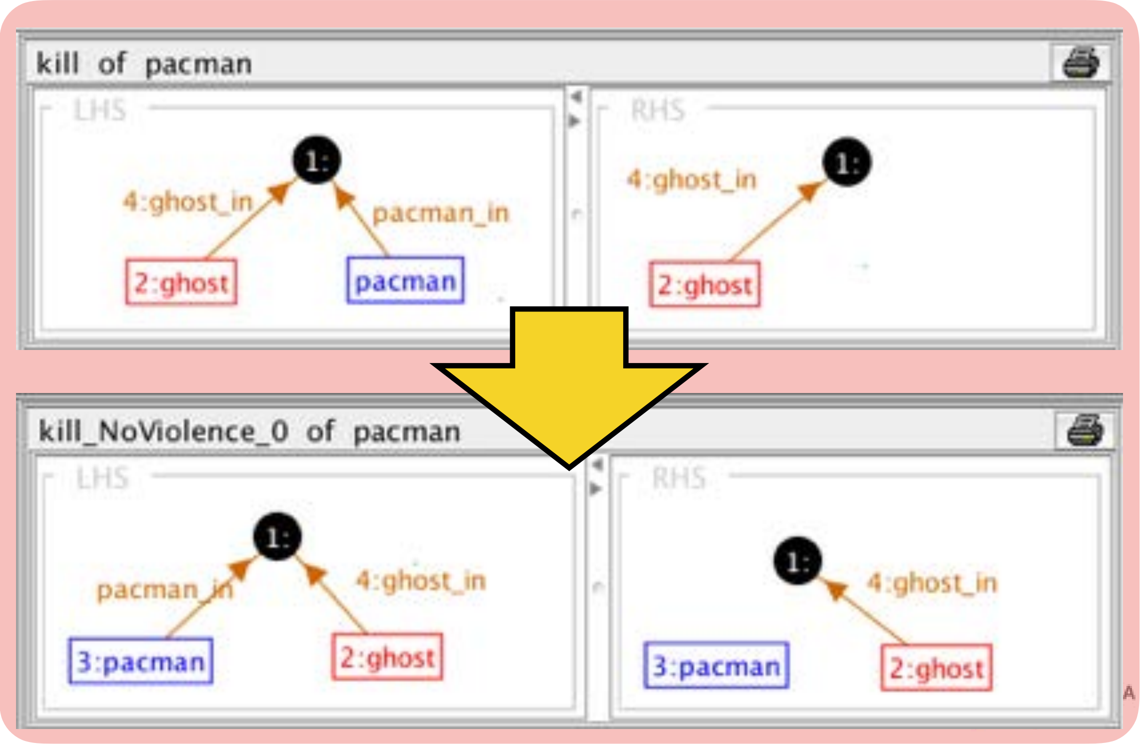
RHS

Second Order Rule

Before

Evolution

After



Demo…

Default: injective matches, flag “--all-matches” sets arbitrary

Critical Pairs/Sequences First Order (terminal and .cpx outputs)

```
verigraph analysis pacman.ggx
verigraph analysis -o out.cpx pacman.ggx
```

Critical Pairs/Sequences Second Order

```
verigraph analysis --snd-order pacman.ggx
```

Concurrent Rules, all epi pairs and only by dependency:

```
verigraph concurrent-rule --all-rules -o out.ggx pacman.ggx
verigraph concurrent-rule --all-rules --by-dependency -o out.ggx pacman.ggx
```

Applying Second Order transformations

```
verigraph snd-order -o out.ggx pacman.ggx
```

.inf INSTITUTO DE INFORMÁTICA UFRGS

Comparison: Performance

Tool	treeToList*		mutex*	
	Critical Pairs Time(s)	Critical Sequences Time(s)	Critical Pairs Time(s)	Critical Sequences Time(s)
AGG	1.704	6.156	10.874	47.717
Verigraph	0.822	3.489	1.036	3.224

* the grammars are in the Verigraph repository

Comparison: Features

Tool Feature	AGG	Verigraph
Rewriting	SPO / DPO simulation	DPO
Typed Graphs	✔	✔
Attributes	✔	✘
Subtyping	✔	✘
Second Order	✘	✔
Concurrent Rules	✔	✔
Critical Pairs/ Sequences	✔	✔
UI	GUI	CLI + import/export from AGG
Language	Java	Haskell

Ongoing/Future Work

- Graphical User Interface
- Graph constraints
- Attributes
- Graph processes
- AGREE/SqPO
- Evolve NACs with second order rules
- Model checking
- Theorem proving

Verigraph available at

- Source code: github.com/Verites/verigraph
- Tutorial: : ufrgs.br/verites/verigraph/verigraph-tutorial-v1.0-rc02
- Internal API docs: verites.github.io/verigraph

Thanks!

.Inf INSTITUTO DE INFORMÁTICA UFRGS