Modeling, Constraint Solving and Model Manipulation sub: Code, lots of it!

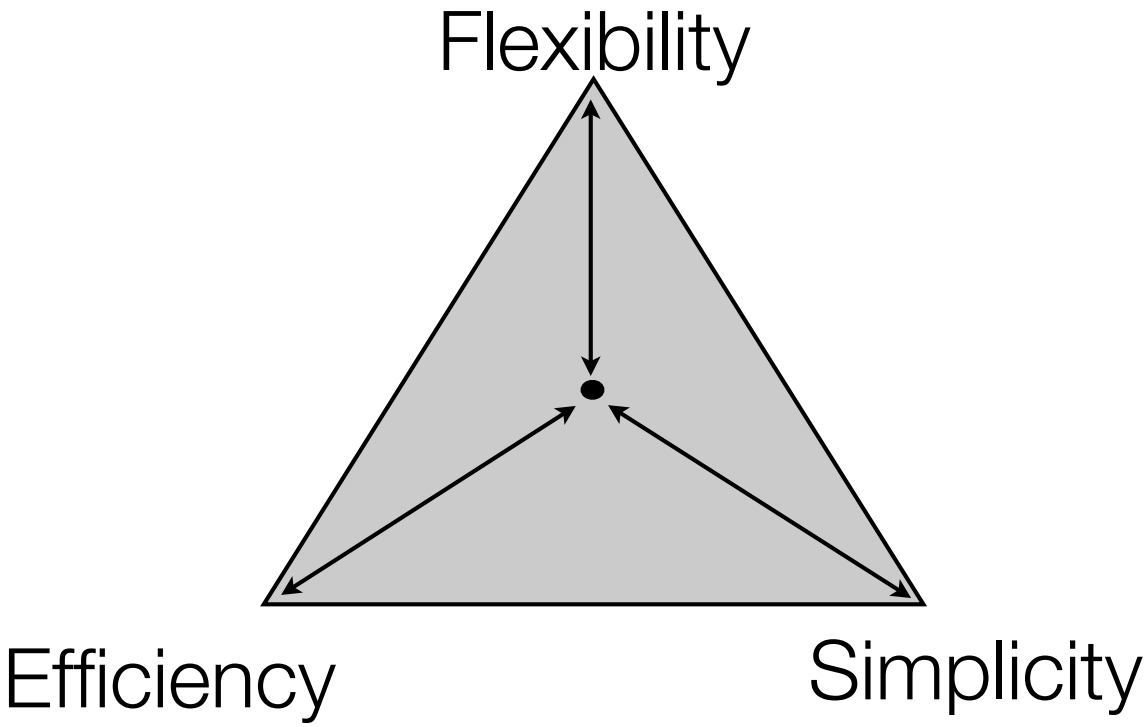
Laurent Michel[1], Pascal Van Hentenryck[2] Pierre Schauss[3]
[1]U. of Connecticut
[2]U. of Michigan
[3]U. of Louvain-La-Neuve

Overview

- Motivation
- Modeling Layer
  - Variables / Constraints
  - Transformations, reformulation and concretizations
  - Consistency handling
- Solving Layer
  - Microkernel
  - Events / Propagators and Propagation
  - Views
- Hybrids
- Search building blocks
- Parallel support

Motivation

- Solver Design Objective
  - Strike a balance….



Design Mantra

Optimization Program

=

Model + Solver + Search

Models

Optimization Program = Model + Solver + Search

- Models are first-class objects
  - models contain abstract objects, can be inspected/transformed
- The model expresses the structure explicitly
  - modeling is a key challenge in optimization
  - transforming models is a fundamental area of research
- At least the same expressivity as modeling languages
  - AMPL, OPL, Zinc

Solvers

Optimization Program = Model + Solver + Search

- Technology **specific**
  - Holds onto representation for variables, domains, propagators,…
- Provides inference capabilities
  - Through propagators
  - Through statistics (learning)
- Memory consideration
  - Concurrency
  - State restoration

Search

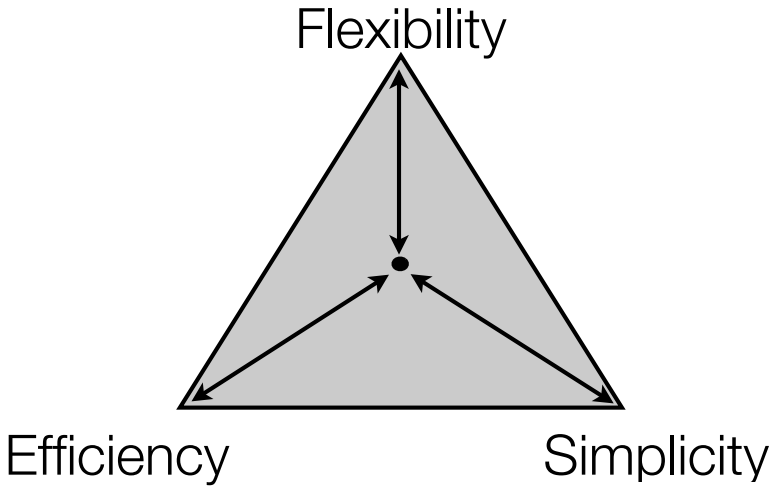
Optimization Program = Model + Solver + Search

- Technology **agnostic**
  - Adapts to LS / CP / Even (M)IP ? …
- Supports
  - Non-determinism
  - Strategies (non-chronological)
  - Instrumentation
  - Concurrent-friendly

Objective Today

- How to bring this **VISION** together
  - Clean separation of concerns (Model + Solver + Search)
  - Clean mapping from Declarative to Operational
    - Declarative abstractions
    - Concrete abstractions
    - Concretization process
  - Clean Search
    - Technology neutral
    - Concurrency neutral

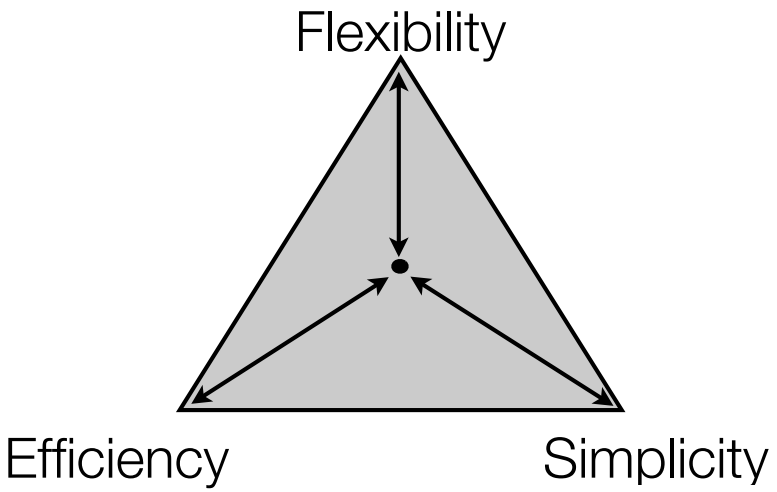
Benefits



- Flexibility
  - Reusable models
  - Technology switch easy
  - Hybrids techniques
- Simplicity
  - Microkernel maintainable
  - Microkernel adaptable
  - Easily and **transparently** support parallel computation
- Efficiency
  - Low computational overhead
  - Constant factor memory usage increase

Roadmap

- Look at architecture for
  - Abstract layer
  - Concrete layer
  - Concretization process
  - Reformulation process
- Focus on design decisions that reach the right balance



One Example

- Consider the classic magic series with two redundant

$$\forall i \in 0..n-1 : \sum_{k \in 0..n-1} (x_k = i) = x_i$$

$$\sum_{i \in 0..n-1} (x_i \cdot i) = n$$

$$\sum_{i \in 0..n-1} x_i \cdot (i-1) = 0$$

This is a model with no commitment to any technology

One Example

- As a Swift program

```
import ORProgram
autoreleasepool {
   print("magicSerie in swift!")
   let n = 14
   let m = ORFactory.createModel()
   let R = range(m,0...n)
   let x = ORFactory.intVarArray(m, range: R, domain: R)
   for i in 0..<n {
      m.add(sum(m, R: R) {k in x[k] == i} == x[i])
   }
   m.add(sum(m,R: R) {i in x[i] * i     } == n)
   m.add(sum(m,R: R) {i in x[i] * (i-1)} == 0)
}
```

Still no solver. Only a *declarative* model

One Example: Solving it!

```
import ORProgram

autoreleasepool {
    …

    let cp = ORFactory.createCPProgram(m)
    cp.search {
        firstFail(cp, x)
    }
    print("Number of solutions: \(cp.solutionPool().count())")
}
```

Solve with CP, sequentially.

One Example: Solving it again…

```
import ORProgram

autoreleasepool {
   …

   let cp = ORFactory.createCPParProgram(m, nbThreads : 2)
   cp.search {
      firstFail(cp, x)
   }
   print("Number of solutions: \(cp.solutionPool().count())")
}
```

Solve with CP, in parallel.

One Example: Solving it again…

```
import ORProgram

autoreleasepool {
    …

    let cp = ORFactory.createMIPProgram(m)
    cp.defaultSearch()


    print("Number of solutions: \(cp.solutionPool().count())")
}
```

Solve with MIP (Gurobi).

Let’s do this!

Overview

- Motivation
- Modeling Layer
  - Variables / Constraints
  - Transformations, reformulation and concretizations
  - Consistency handling
- Solving Layer
  - Microkernel
  - Events / Propagators and Propagation
  - Views
- Hybrids
- Search building blocks
- Parallel support

Variables [Objectice-CP, CP'13]

- Use multiple abstractions
  - An *Abstract* Modeling Variable
    - Purely Declarative
    - Only holds onto an identifier
    - Is serializable
  - Several *Concrete* Implementation Variables
    - Purely Operational
    - Holds data structures needed by the matching solver

When modeling…

- Only create the declarative abstraction
- Postpone the creation of operational objects

Modeling Layer

- Have an *explicit* model representation
  - Model *owns* variables (as well as data-structures and constraints)
  - Variable remembers its owner (model)
  - Variable only store an identifier and possibly bounds
- Use sub-type polymorphism
  - To support multiple *types of variables*
    - Integers
    - Floats
    - Real
    - Sets
    - …

Constraints

- Use multiple abstractions
  - Abstract *constraints* for modeling
    - Purely declarative
    - Expressions for algebraic, logical and reified constraints
    - Classes for global constraints
    - Hold an identifier each time
  - Concrete *propagators* for solving
    - Purely operational
    - Contain data structures and filtering algorithms

Expressions for Constraints

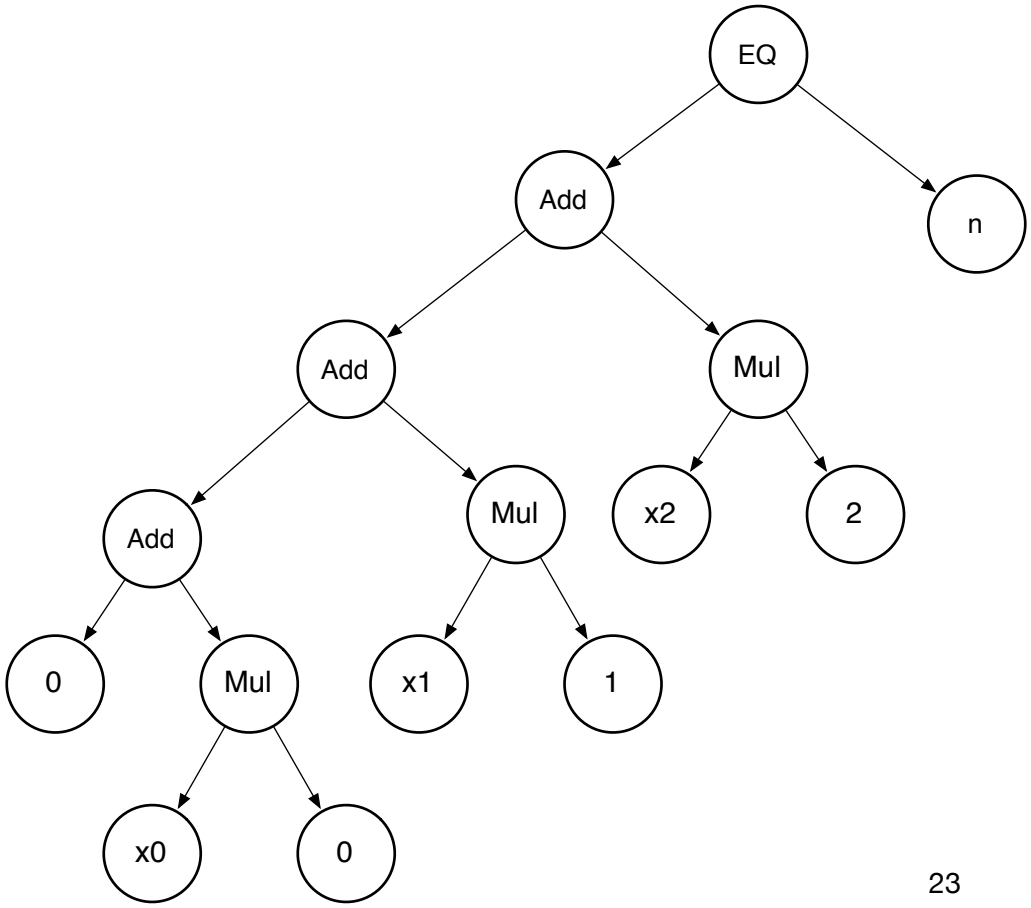
- Expressions language **E** defined inductively with a simple grammar

```
Rel    :== Rel {||,&&,=>} Rel
       :== !Rel
Rel    :== Expr {≤,≥,<,>,==,!=} Expr
Expr   :== Expr {+,-,*,/} Expr
       :== sum(id in Expr) Expr
       :== prod(id in Expr) Expr
       :== Expr [ integer ]
       :== Expr [ Expr ]
       :== Expr . id
       :== id
       :== integer
       :== abs(Expr)
```

Expression and Parse Trees

```
sum(m,R: R) {i in x[i] * i    } == n
```

- This code builds an expression for the language **E**
  - Assume n = 3 (for brevity)
  - Left associativity for +
  - Neutral of zero
  - Composite design pattern
- Introspection with tree traversal
  - Visitor design pattern



Classes for Constraints

- Useful for combinatorial structure
- Prototypical example

```
alldifferent(x : array[ORIntVar])
```

- Part of Sudoku Model

```
import ORProgram
autoreleasepool {
   let model = ORFactory.createModel()
   let R     = range(model, 1...9)
   let x     = ORFactory.intVarMatrix(model, range: R, R, domain:R)
   for i : ORInt in 1...9 {
      model.add(ORFactory.alldifferent(all(model,R) { j in x[i,j] }))
      model.add(ORFactory.alldifferent(all(model,R) { j in x[j,i] }))
   }
   …
}
```

Model Introspection

- Key ability
  - Must be able to traverse / inspect all the entities in the model
  - For *serialization*
  - For *reformulation* [Combinators,CP'13,Lagrangian,CP'14,Scheduling,CPAIOR'16]
  - For *concretization*

Reformulation ?

- It's a compilation / rewriting of
  - From an abstract model M_0 $\langle X, D, C \rangle$
  - To a abstract model M_1 $\langle X', D', C' \rangle$

- Formally

$$\langle X', D', C' \rangle = \tau(\langle X, D, C \rangle)$$

Reformulation

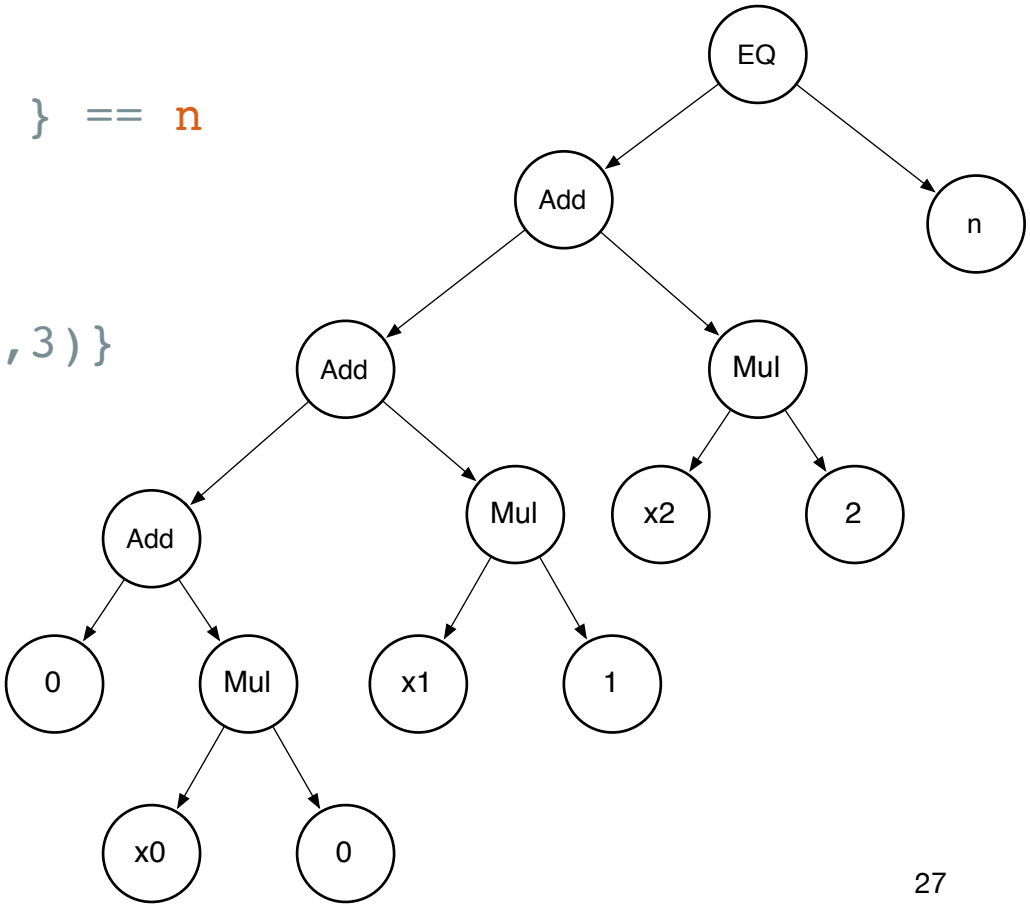
- Simple example
  - Decompose expressions into basic constraints
- Input: an expression:

```
sum(m,R: R) {i in x[i] * i } == n
```

- Output : A set of constraints

```
{LinearEQ([x[1],x[2]],[1,2],3)}
```

- Notes
  - No new variables
  - A single output constraint



Specification

- First
  - Let a term $t \in \mathcal{T}$ represent a linear form

$$t = \sum_{i \in S} a_i \cdot x_i + c$$

- Then define relations over expression in language **E**

$$\mathcal{R}_e : Rel \rightarrow 2^C \times 2^X$$

$$\mathcal{L}_e : Expr \rightarrow \mathcal{T} \times 2^C \times 2^X$$

Abridged Definitions

- R relation for equality

$$\mathcal{R}_e(e_1 == e_2) = \langle C_1 \cup C_2 \cup \{\text{LinearEQ}(t_3, 0)\}, X_1 \cup X_2 \rangle$$

  - Where

$$\mathcal{L}_e(e_1) = \langle t_1, C_1, X_1 \rangle$$
$$\mathcal{L}_e(e_2) = \langle t_2, C_2, X_2 \rangle$$

  - In

$$t_3 = t_1 \ominus t_2$$

Abridged Definitions

- R relation for equality to a constant

$$\mathcal{R}_e(e_1 == c) = \langle C_1 \cup \{t_1 - c = 0\}, X_1 \rangle$$

  - Where

$$\mathcal{L}_e(e_1) = \langle t_1, C_1, X_1 \rangle$$

Abridged Definition

- L relations (literal, variable, addition, multiplication by constant)

$$\mathcal{L}_e(c) = \langle c, \emptyset, \emptyset \rangle$$

$$\mathcal{L}_e(x) = \langle 1 \cdot x + 0, \emptyset, \{x\} \rangle$$

$$\mathcal{L}_e(e_1 + e_2) = \langle t_1 \oplus t_2, C_1 \cup C_2, X_1 \cup X_2 \rangle$$

$$\text{where } \begin{cases} \mathcal{L}_e(e_1) & = & \langle t_1, C_1, X_1 \rangle \\ \mathcal{L}_e(e_2) & = & \langle t_2, C_2, X_2 \rangle \end{cases}$$

$$\mathcal{L}_e(e_1 \cdot c) = \langle t_0, C_1, X_1 \rangle$$

$$\text{where } \mathcal{L}_e(e_1) = \langle t_1, C_1, X_1 \rangle$$

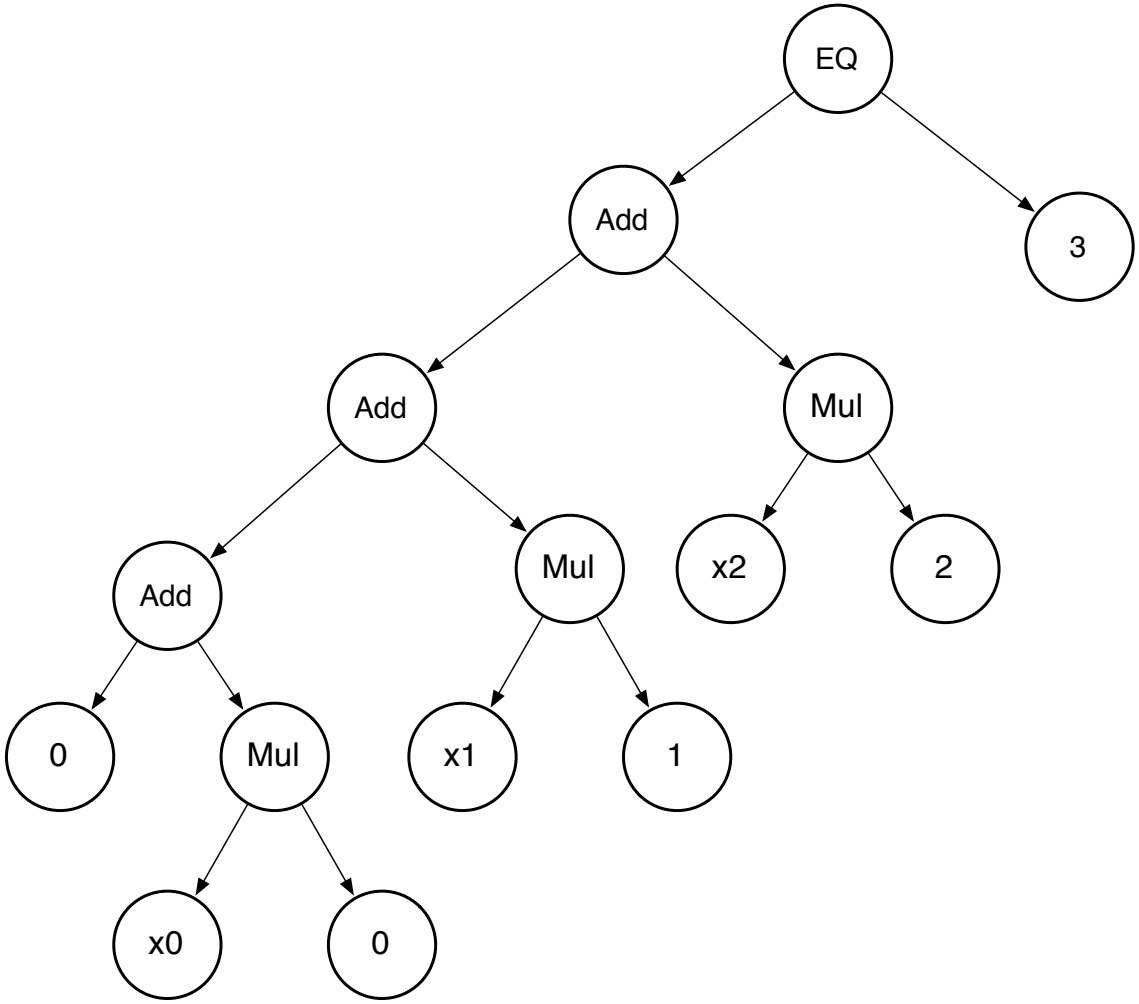
$$\text{in } t_0 = \sum_{i \in S(t_1)} a_{1i} \cdot c \cdot x_i + c_1 \cdot c$$

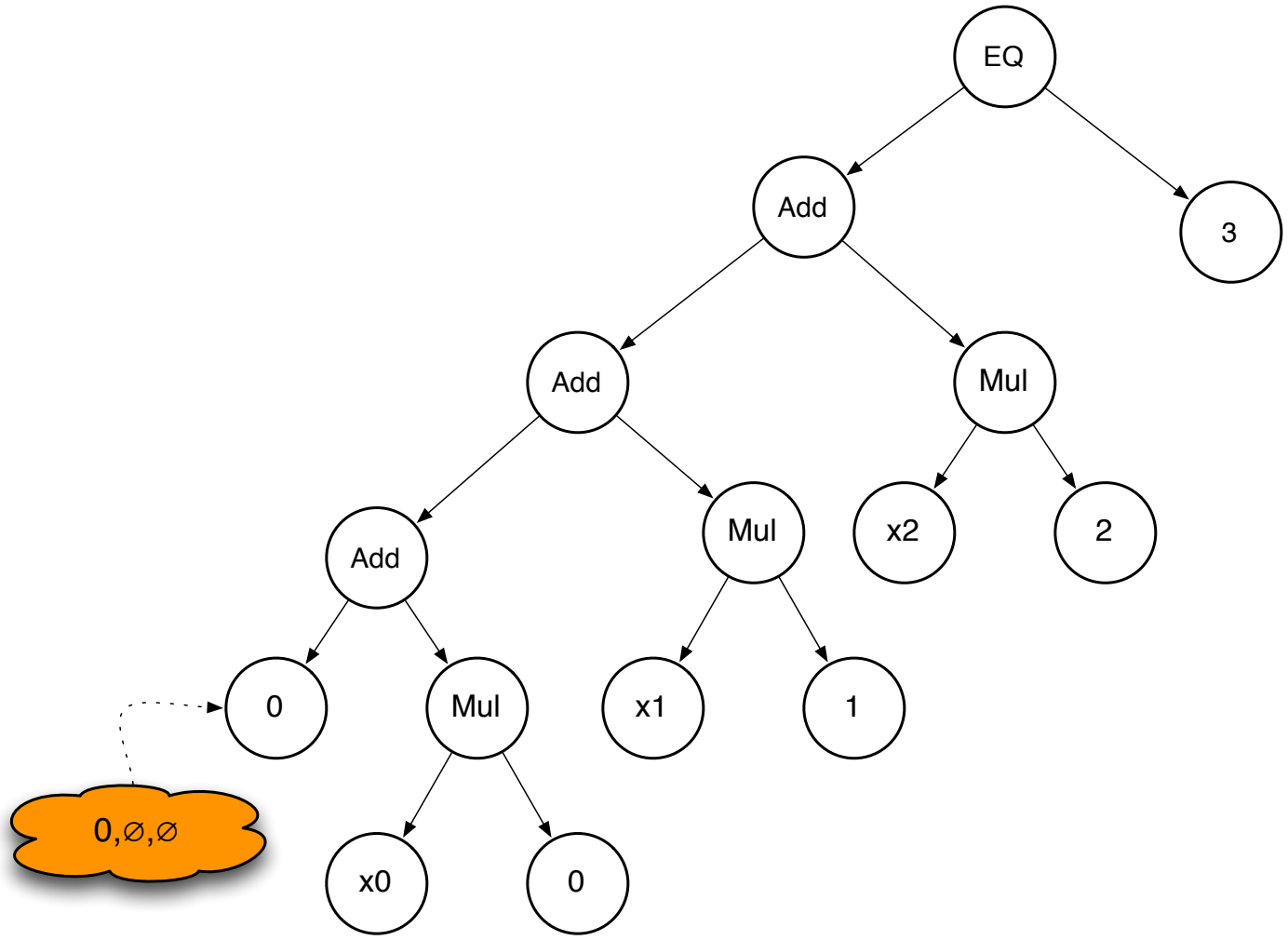
Abridged Definition

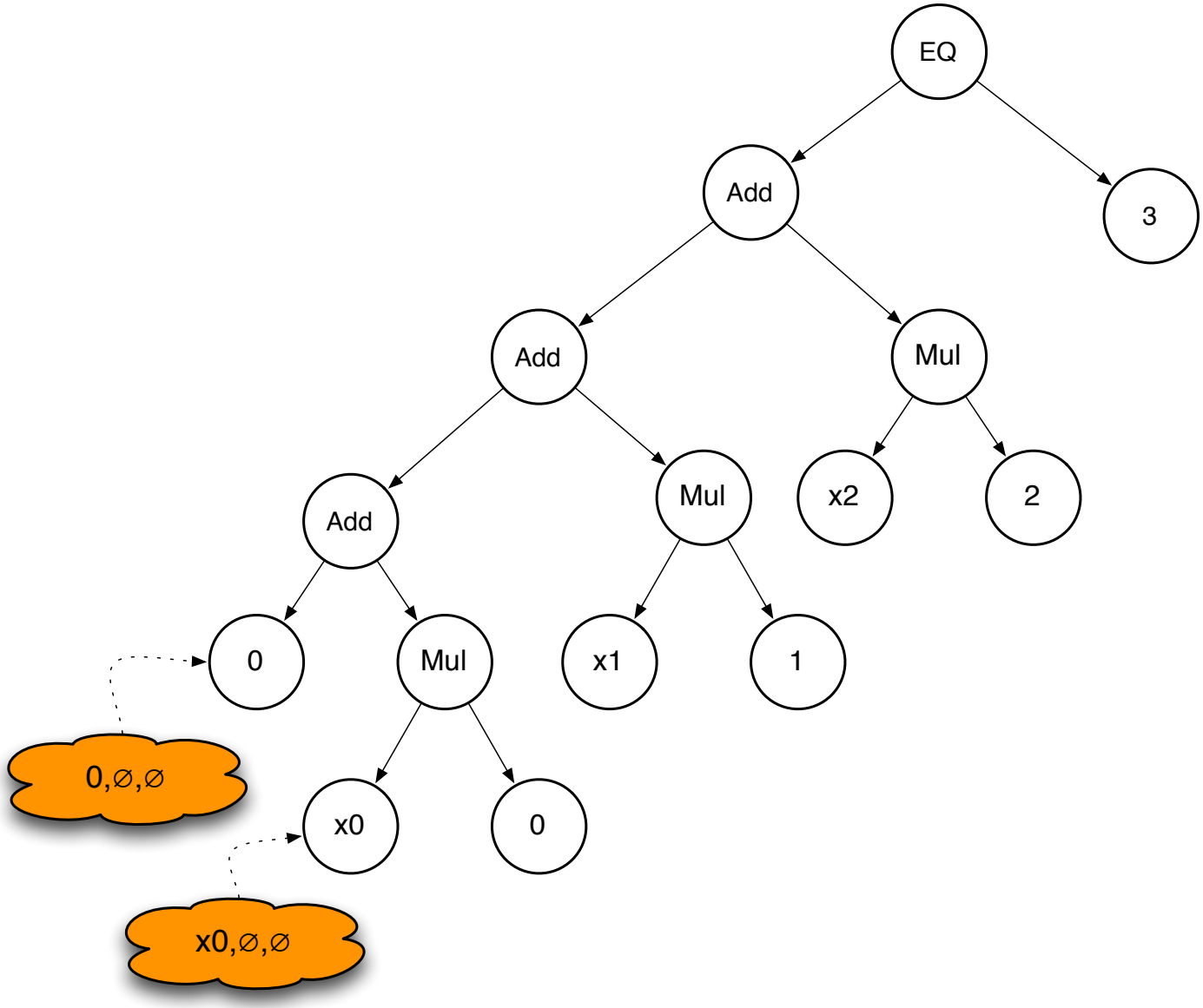
- L relation (reified equality)

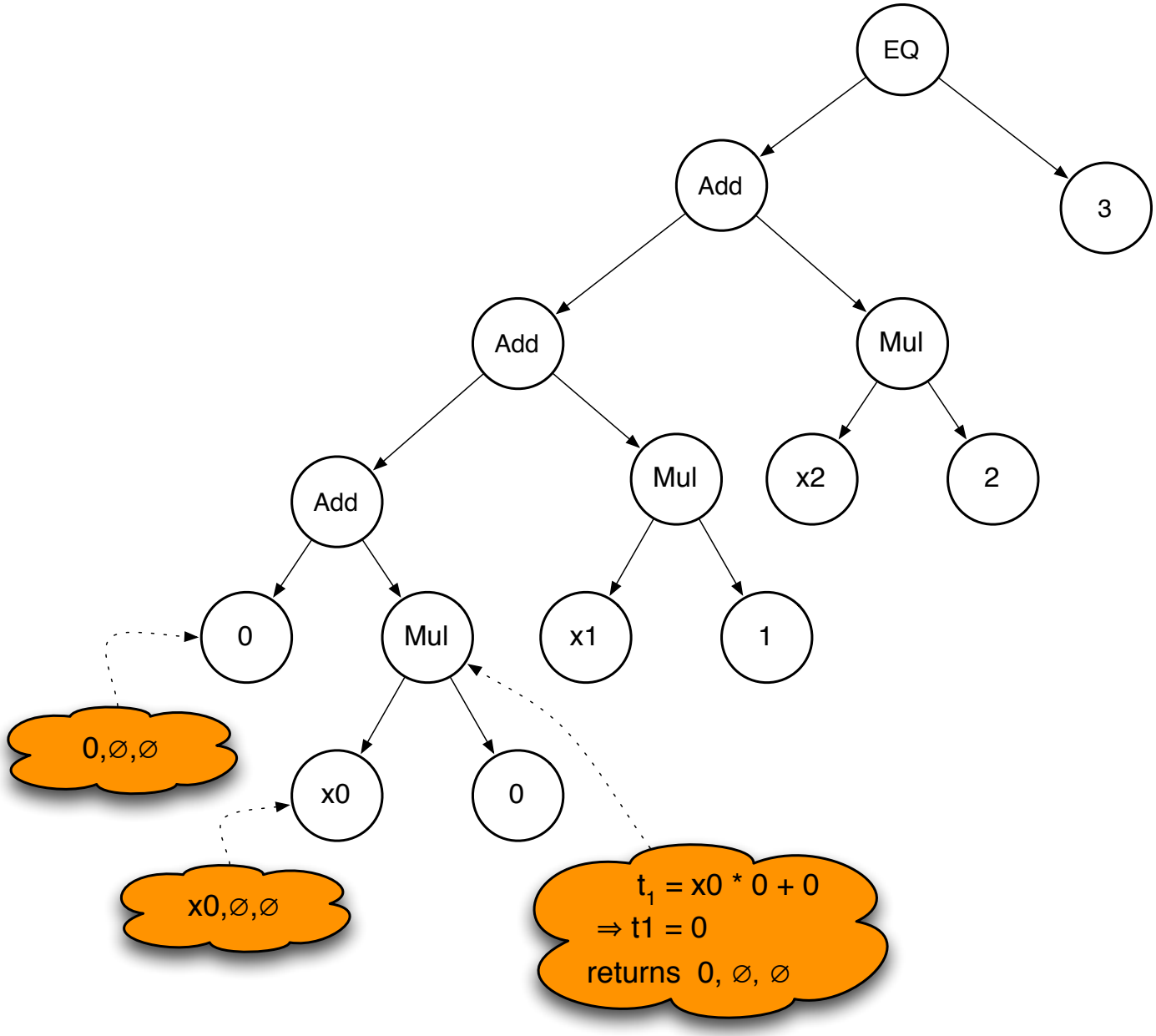
$$\mathcal{L}_e(e_1{==}c) = \langle \alpha_1, C_1 \cup C_2, X_1 \cup \{\alpha_0, \alpha_1\} \rangle$$

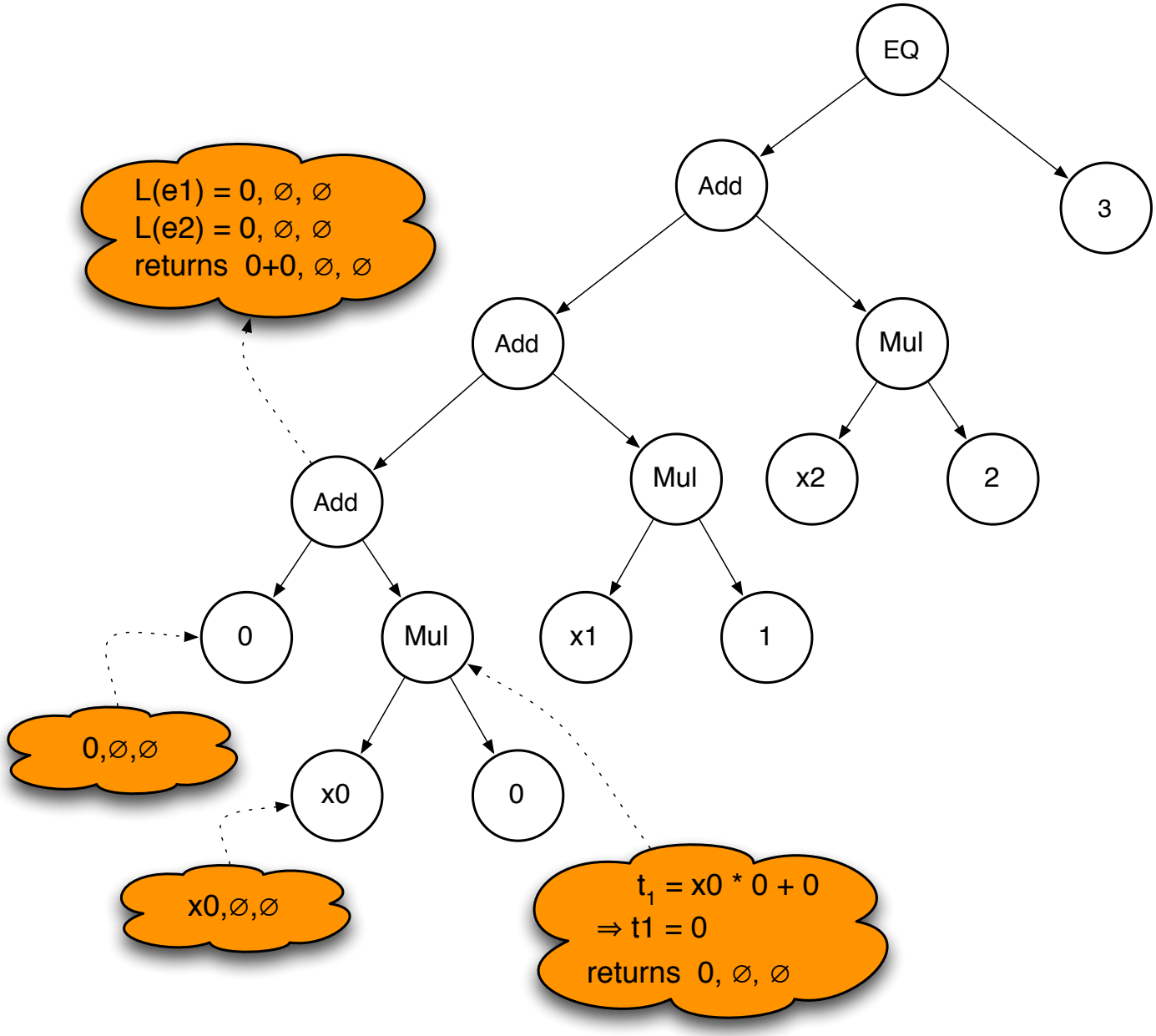
$$\text{where} \quad \begin{array}{lcl} \mathcal{L}_e(e_1) & = & \langle t_1, C_1, X_1 \rangle \\ D(\alpha_0) & = & lower(t_1)..upper(t_1) \\ D(\alpha_1) & = & 0..1 \\ C_2 & = & \{\alpha_0 = t_1, \text{reifyEQ}(\alpha_1, \alpha_0, c)\} \end{array}$$

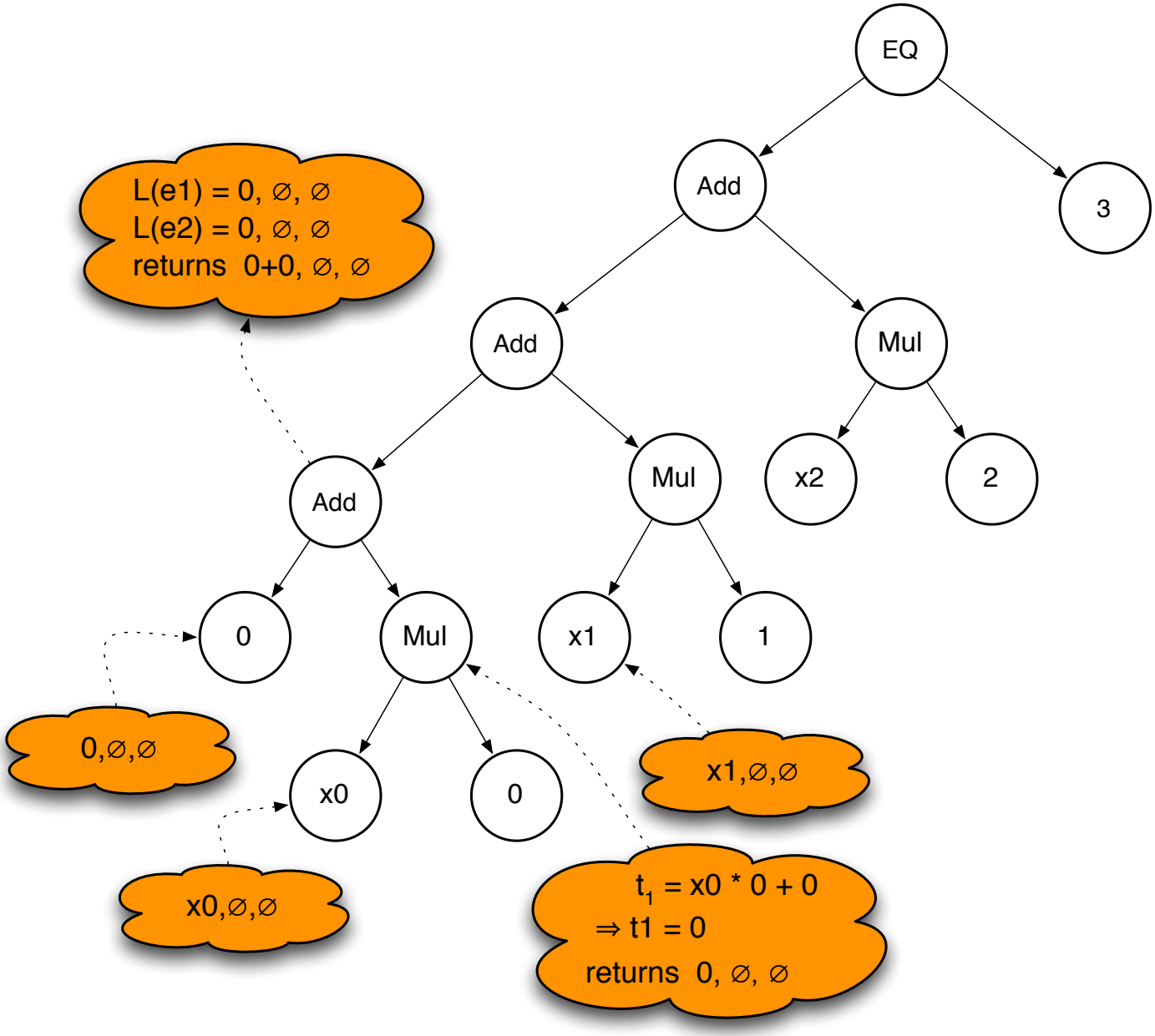

EQ
Add
3
Add
Mul
Add
Mul
x2
2
0
Mul
x1
1
x0
0

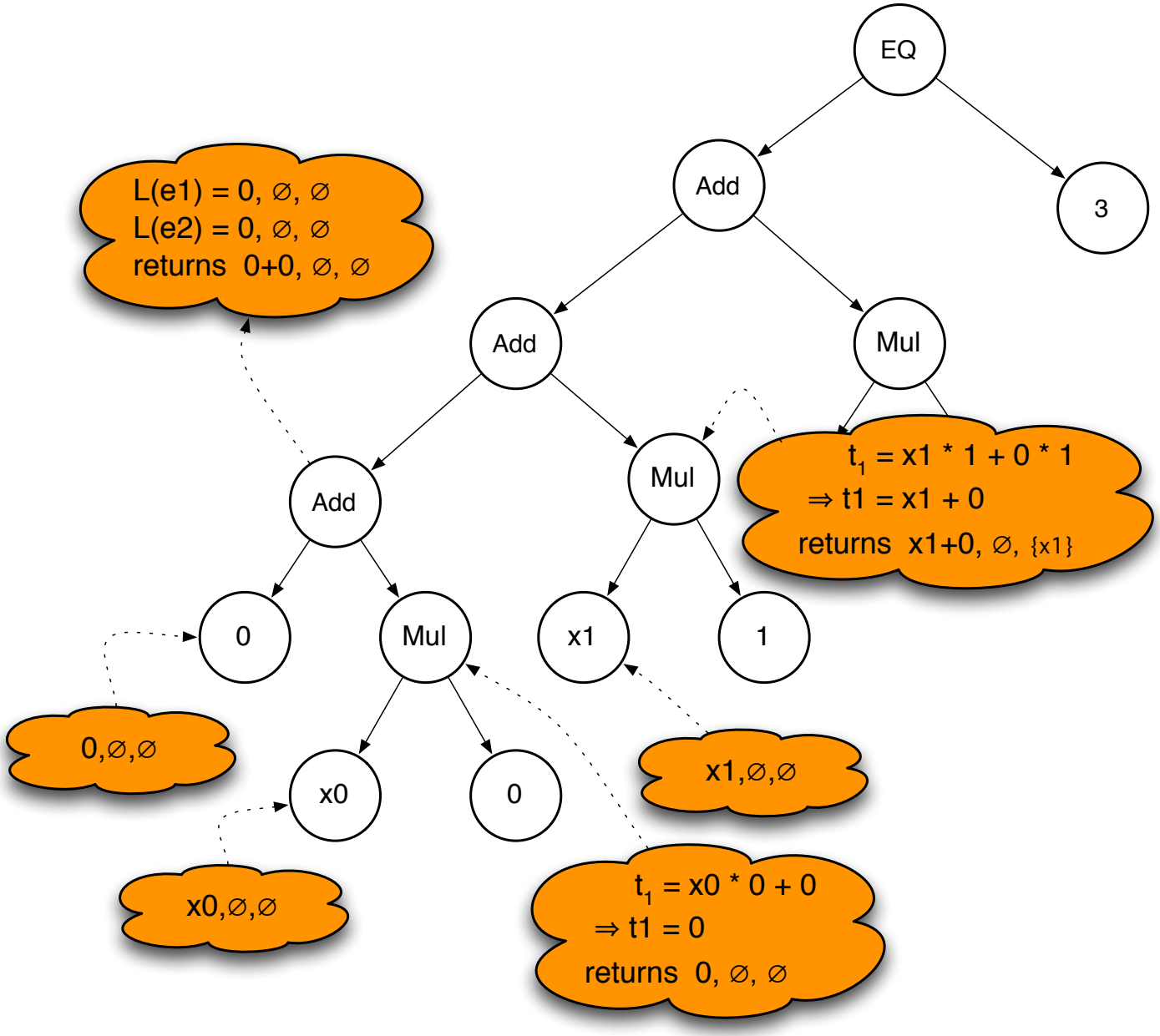

EQ
Add
3
Add
Mul
Add
Mul
x2
2
Add
Mul
0
x1
1
0,∅,∅
x0
0

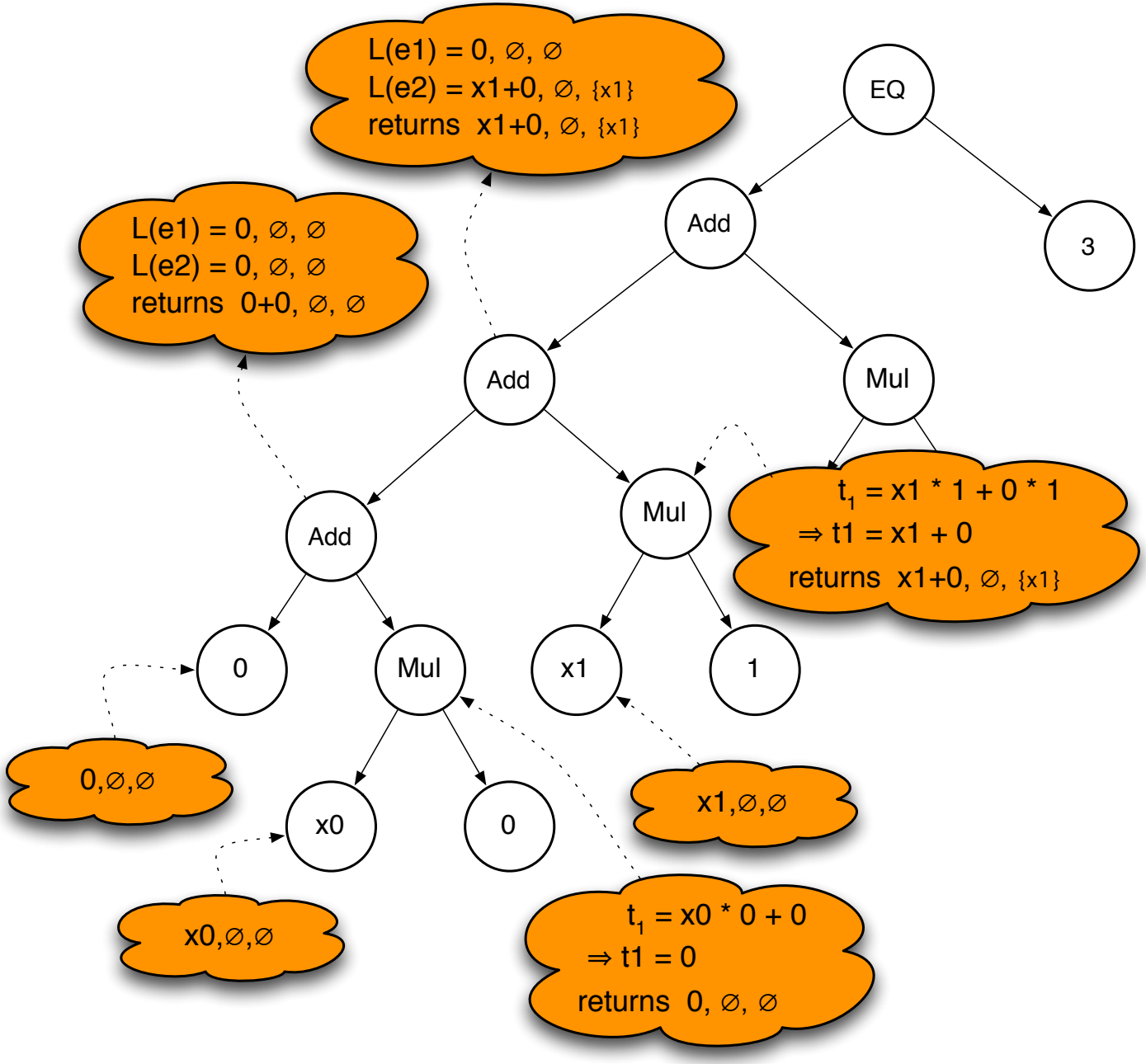

EQ
Add
3
Add
Mul
Add
Mul
x2
2
0
Mul
x1
1
0,∅,∅
x0
0
x0,∅,∅

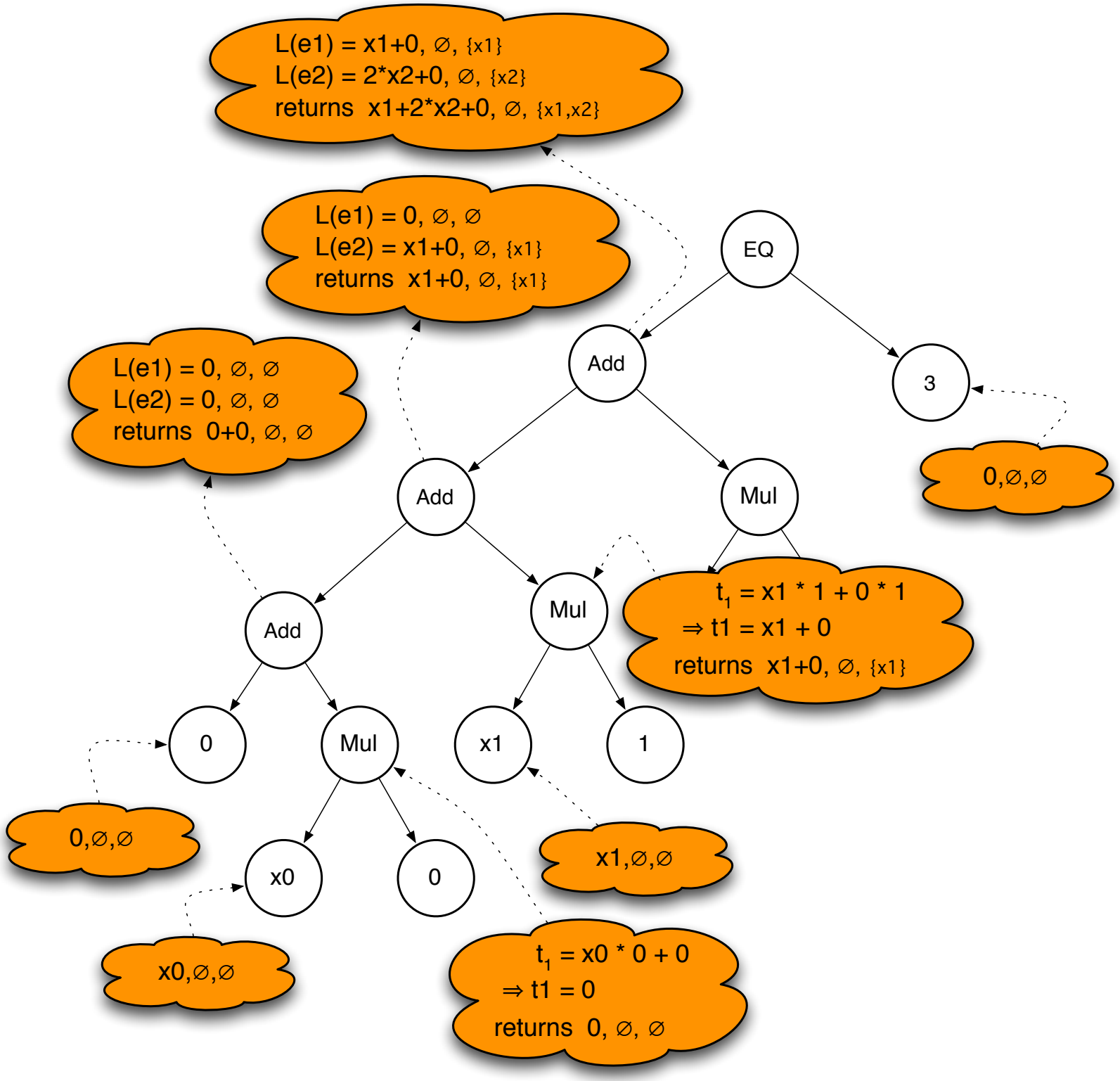

EQ
Add
3
Add
Mul
Add
Mul
x2
2
Add
Mul
0
Mul
x1
1
0,∅,∅
x0
0
x0,∅,∅
t1 = x0 * 0 + 0
⇒ t1 = 0
returns 0, ∅, ∅

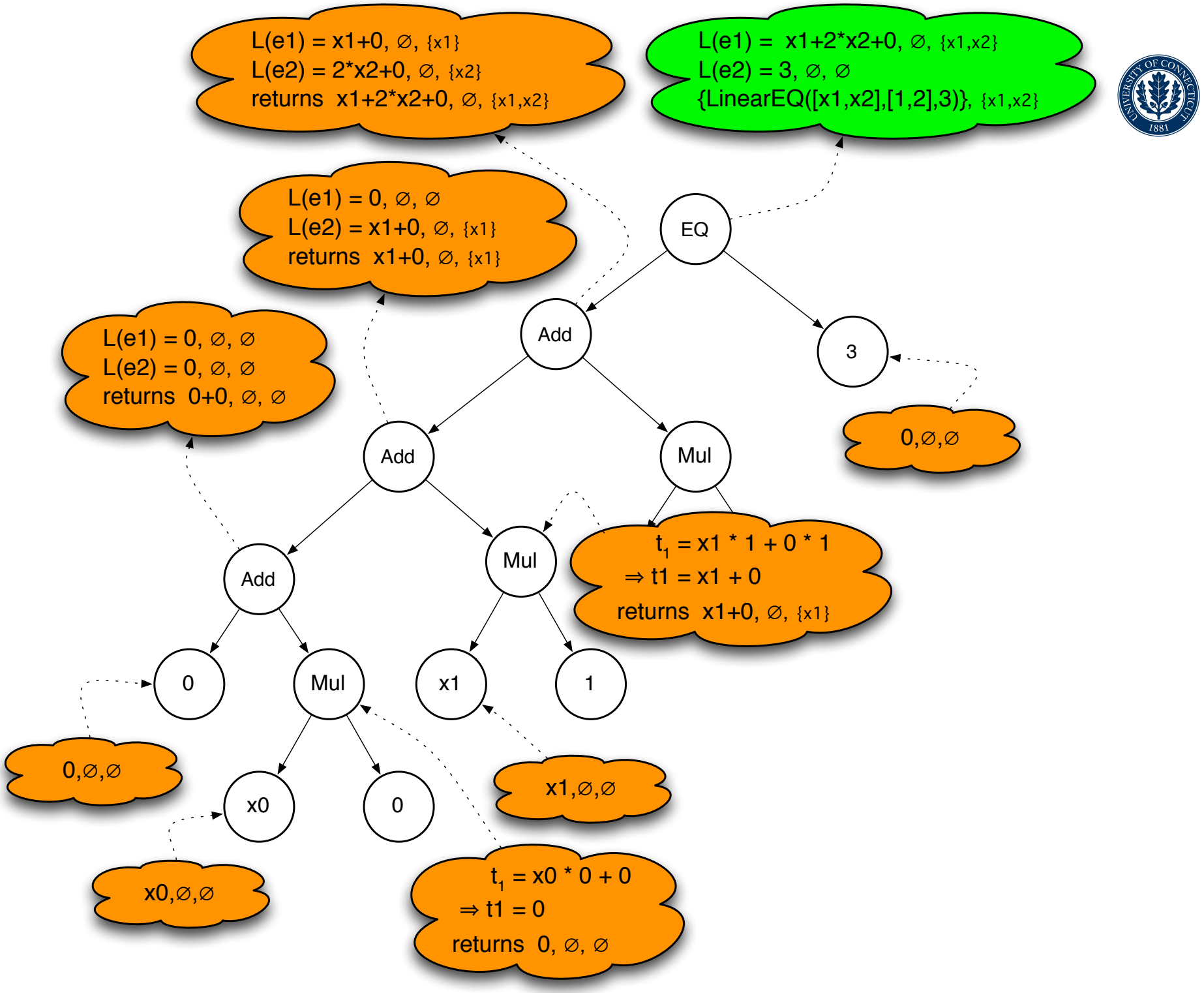

EQ
Add
3
Add
Mul
Add
Mul
x2
2
0
Mul
x1
1
x0
0
L(e1) = 0, ∅, ∅
L(e2) = 0, ∅, ∅
returns 0+0, ∅, ∅
0,∅,∅
x0,∅,∅
t1 = x0 * 0 + 0
⇒ t1 = 0
returns 0, ∅, ∅


EQ
Add
3
Add
Mul
Add
Mul
x2
2
0
Mul
x1
1
x0
0
L(e1) = 0, ∅, ∅
L(e2) = 0, ∅, ∅
returns 0+0, ∅, ∅
0,∅,∅
x0,∅,∅
x1,∅,∅
t1 = x0 * 0 + 0
⇒ t1 = 0
returns 0, ∅, ∅


EQ
Add
3
Add
Mul
Add
Mul
0
Mul
x1
1
x0
0
L(e1) = 0, ∅, ∅
L(e2) = 0, ∅, ∅
returns 0+0, ∅, ∅
t1 = x1 * 1 + 0 * 1
⇒ t1 = x1 + 0
returns x1+0, ∅, {x1}
0,∅,∅
x1,∅,∅
x0,∅,∅
t1 = x0 * 0 + 0
⇒ t1 = 0
returns 0, ∅, ∅


L(e1) = 0, ∅, ∅
L(e2) = x1+0, ∅, {x1}
returns x1+0, ∅, {x1}
L(e1) = 0, ∅, ∅
L(e2) = 0, ∅, ∅
returns 0+0, ∅, ∅
EQ
Add
3
Add
Mul
Add
Mul
t_1 = x1 * 1 + 0 * 1
⇒ t1 = x1 + 0
returns x1+0, ∅, {x1}
0
Mul
x1
1
0,∅,∅
x0
0
x1,∅,∅
x0,∅,∅
t_1 = x0 * 0 + 0
⇒ t1 = 0
returns 0, ∅, ∅


L(e1) = x1+0, ∅, {x1}
L(e2) = 2*x2+0, ∅, {x2}
returns x1+2*x2+0, ∅, {x1,x2}
L(e1) = 0, ∅, ∅
L(e2) = x1+0, ∅, {x1}
returns x1+0, ∅, {x1}
L(e1) = 0, ∅, ∅
L(e2) = 0, ∅, ∅
returns 0+0, ∅, ∅
EQ
Add
3
Add
Mul
0,∅,∅
Add
Mul
t1 = x1 * 1 + 0 * 1
⇒ t1 = x1 + 0
returns x1+0, ∅, {x1}
0
Mul
x1
1
0,∅,∅
x0
0
x1,∅,∅
x0,∅,∅
t1 = x0 * 0 + 0
⇒ t1 = 0
returns 0, ∅, ∅


L(e1) = x1+0, ∅, {x1}
L(e2) = 2*x2+0, ∅, {x2}
returns x1+2*x2+0, ∅, {x1,x2}
L(e1) = x1+2*x2+0, ∅, {x1,x2}
L(e2) = 3, ∅, ∅
{LinearEQ([x1,x2],[1,2],3)}, {x1,x2}
L(e1) = 0, ∅, ∅
L(e2) = x1+0, ∅, {x1}
returns x1+0, ∅, {x1}
L(e1) = 0, ∅, ∅
L(e2) = 0, ∅, ∅
returns 0+0, ∅, ∅
EQ
Add
3
Add
Mul
0,∅,∅
Add
Mul
t1 = x1 * 1 + 0 * 1
⇒ t1 = x1 + 0
returns x1+0, ∅, {x1}
0
Mul
x1
1
0,∅,∅
x0
0
x1,∅,∅
x0,∅,∅
t1 = x0 * 0 + 0
⇒ t1 = 0
returns 0, ∅, ∅

Abridged Definitions

- R relation for equality

$$\mathcal{R}_e(e_1 == e_2) = \langle C_1 \cup C_2 \cup \{\text{LinearEQ}(t_3, 0)\}, X_1 \cup X_2 \rangle$$

  - Where

$$\mathcal{L}_e(e_1) = \langle t_1, C_1, X_1 \rangle$$
$$\mathcal{L}_e(e_2) = \langle t_2, C_2, X_2 \rangle$$

  - In

$$t_3 = t_1 \ominus t_2$$

Transformation Maps

- Easy to do with identifiers
  - Array of objects indexed by source identifier.
- When

$$y = \tau(x)$$

- Create an array and…

$$\tau[x.id] \leftarrow y$$

Summary

- Decomposition is a Reformulation
  - It rewrites expressions and breaks them down
  - It produces a set of *simpler* constraints
  - It can introduce auxiliary variables
- Bottom line
  - The resulting model is still technology neutral
  - The model *transformation* tracks the rewriting in a mapping τ

Concretization ?

- It's a compilation of
  - An abstract model $\langle X, D, C \rangle$
  - To a concrete representation $\langle X', D', C' \rangle$

- Formally

$$\langle X', D', C' \rangle = \gamma(\langle X, D, C \rangle)$$

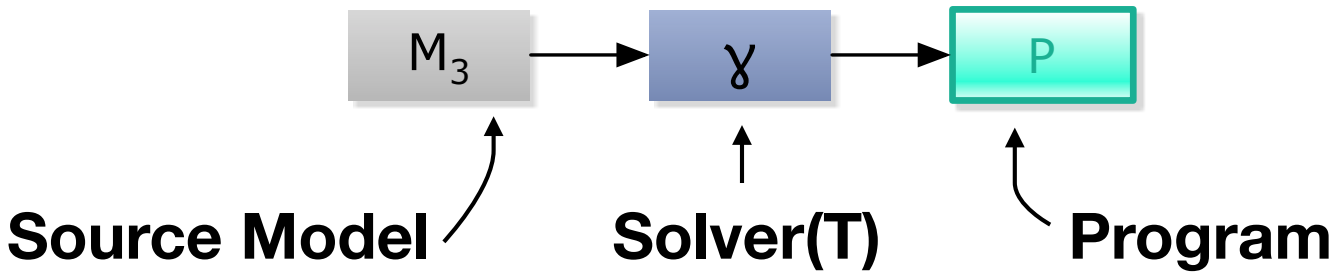
- More details: Next talk!

Concretization

- Purpose
  - *maps* variables
  - *adds* auxiliary variables
  - *maps* abstract constraints into concrete propagators
  - *builds mappings* to
    - Remember the relationships
  - Mappings enable the *translation* of
    - abstract operations on the declarative model
    - into concrete operations on the operational model

Concretization Process

- Introspection on model
  - Inductive definition on the constructions in **L**
  - Build the concrete objects and add them
    - To the solver
    - To the concretization map ɣ
- Output: A Program
  - Computational capabilities
  - Only need to add the search



Consistency

- Purpose
  - State the level of consistency desired for a constraint
- How to do it
  - Provide the ability to give an annotation for a model constraint
  - Annotation can be anything.
  - To be interpreted by the solver during the concretization process
  - Annotation stored in a map: Constraint → Note

Usage Example

- From a Swift Model

```
autoreleasepool {
  let n = 50
   let m = ORFactory.createModel()
   let an = ORFactory.annotation()
   let R = range(m,0...n)
   let x = ORFactory.intVarArray(m, range: R, domain: R)
   for i in 0..<n {
      an.dc(m.add(sum(m, R: R) {k in x[k] == i} == x[i]))
   }
   m.add(sum(m,R: R) {i in x[i] * i    } == n)
   m.add(sum(m,R: R) {i in x[i] * (i-1)} == 0)

   let cp = ORFactory.createCPProgram(m, annotation: an)
   cp.search { firstFail(cp, x) }
   print("Number of solutions: \(cp.solutionPool().count())")
}
```

Define Annotation Map

Ask for DC

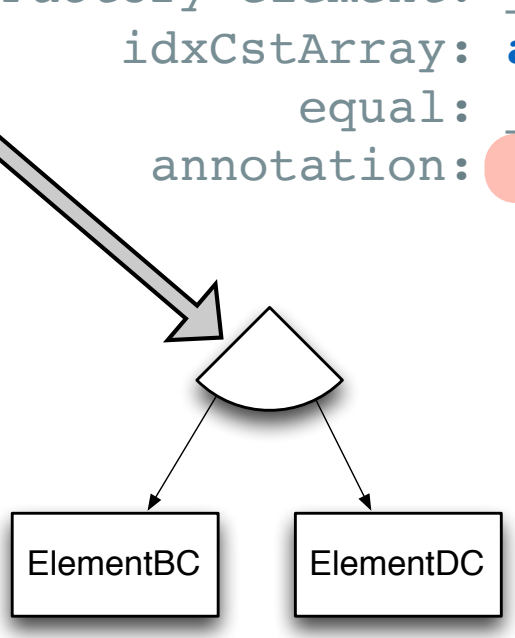
Use Annotation Map

Inside the Implementation

- Close look at one concretization case Element: res = array[idx]
  - Objective-C implementation!

```
-(void) visitElementCst: (id<ORElementCst>) cstr {
   if (_gamma[cstr.getId] == NULL) {
     [cstr.array visit: self];
     [cstr.idx visit: self];
     [cstr.res visit: self];
     id<CPPropagator> cc = [CPFactory element: _gamma[idx.getId]
                                      idxCstArray: array
                                            equal: _gamma[res.getId]
                                       annotation: [_notes levelFor:cstr]
                                ];
     [_engine add: cc];
     _gamma[cstr.getId] = cc;
   }
}
```

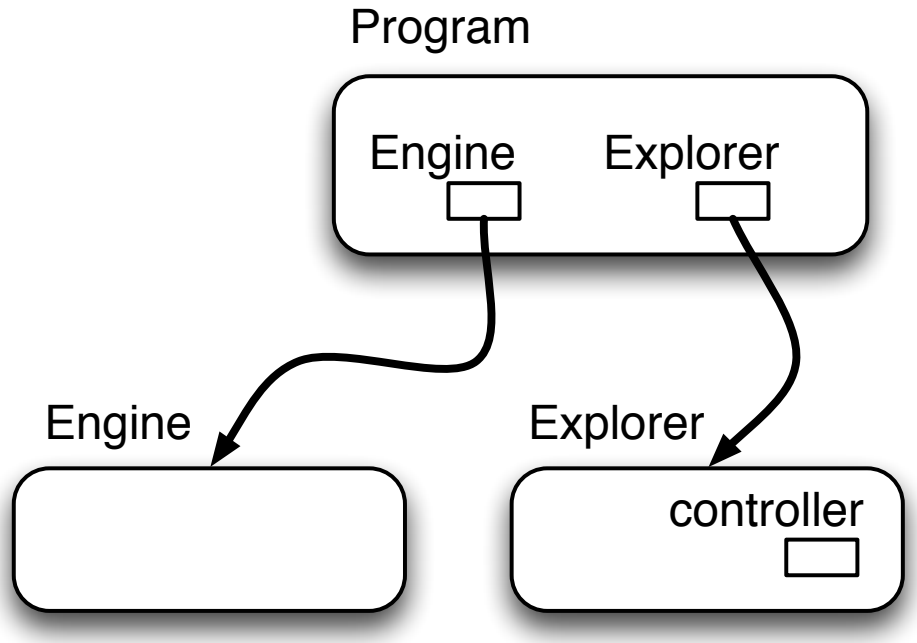
Use Annotation Map



Overview

- Motivation
- Modeling Layer
  - Variables / Constraints
  - Transformations, reformulation and concretizations
  - Consistency handling
- Solving Layer
  - Microkernel
  - Events / Propagators and Propagation
  - Views
- Hybrids
- Search building blocks
- Parallel support

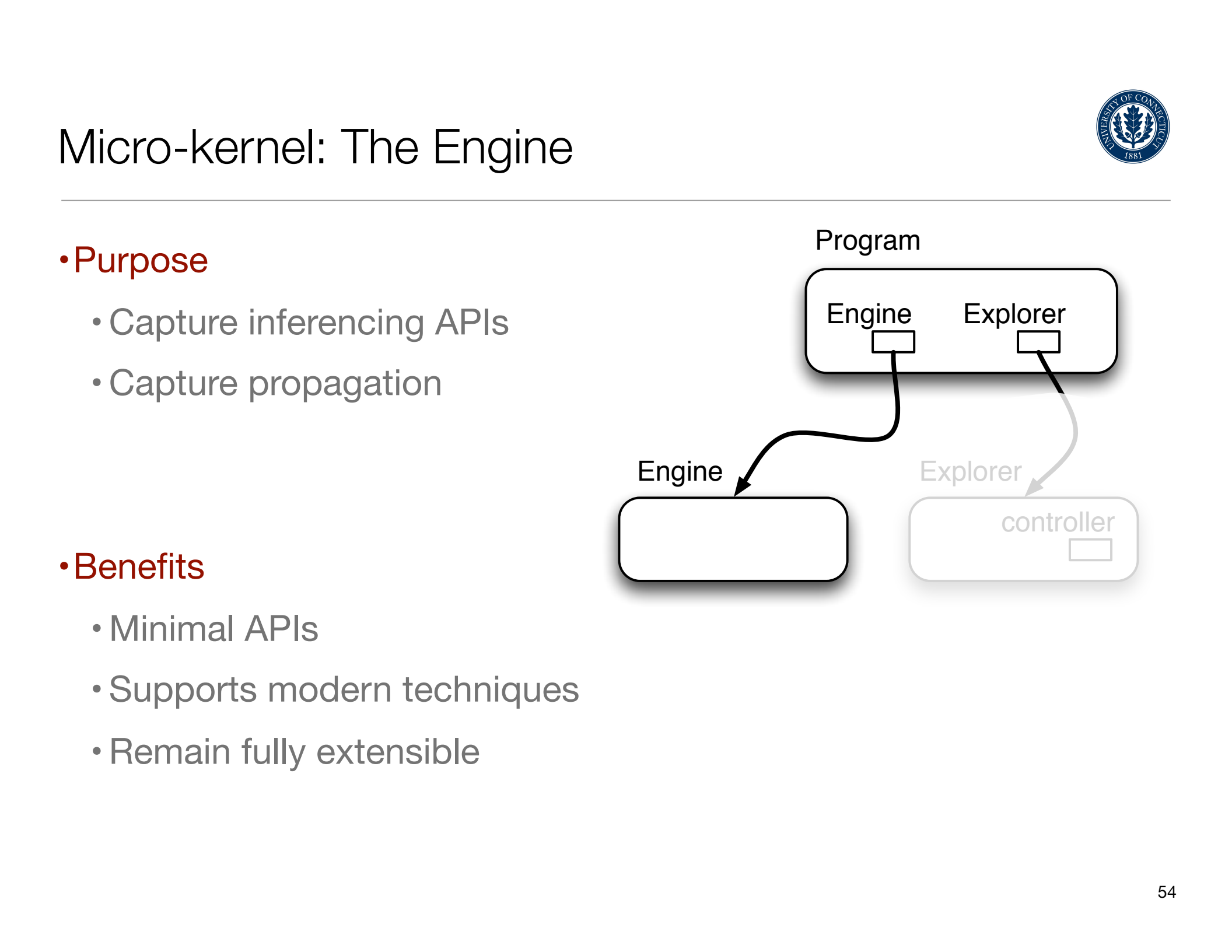
Programs with Micro-kernel Architecture



- Purpose of Program
  - Bundle
    - Computational engine
    - Search capabilities
- Separation of concerns
  - Engine : inference (e.g., propagation)
  - Explorer: search
- Program ?
  - Mostly delegation

[Microkernel,CP'17]

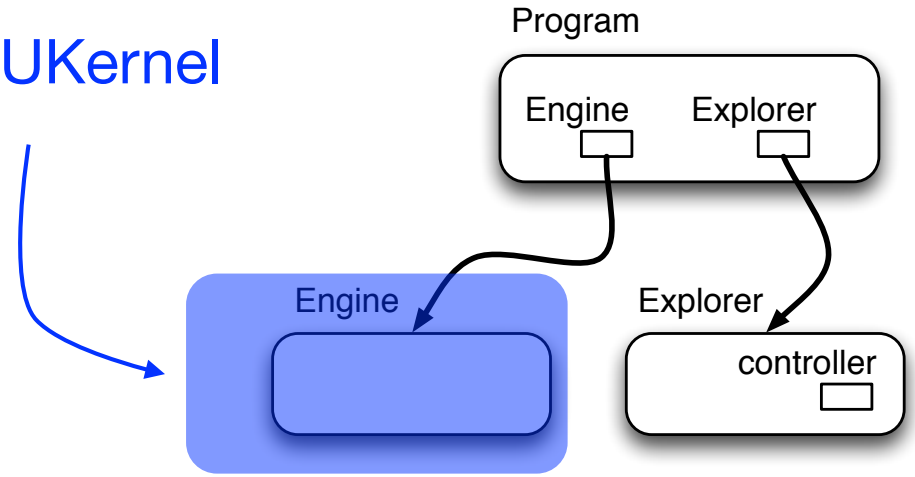
Micro-kernel: The Engine



- Purpose
  - Capture inferencing APIs
  - Capture propagation
- Benefits
  - Minimal APIs
  - Supports modern techniques
  - Remain fully extensible

Bird's eye view

- µKernel APIs (Objective-C protocols)
  - Model loading — CPEngine
  - Propagation — CPUKernel

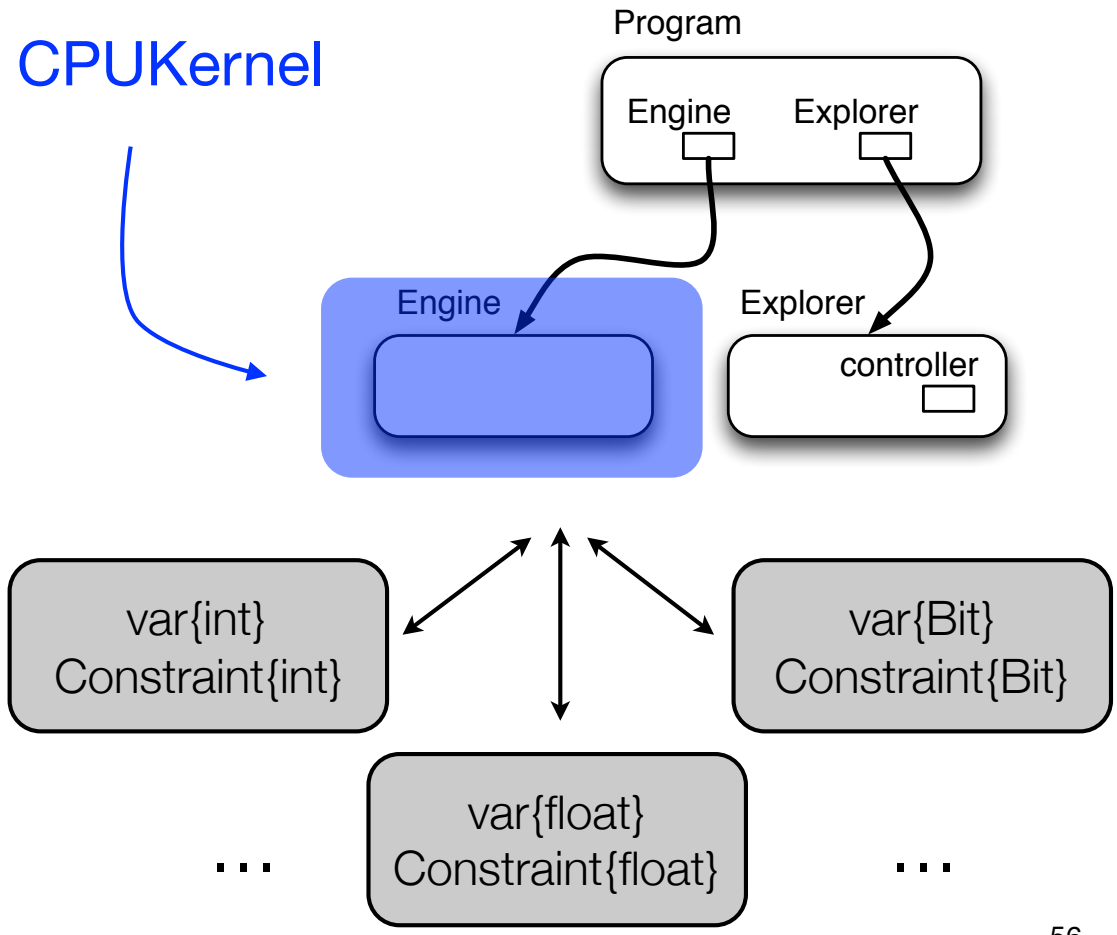


Bird's eye view

- μKernel APIs (interfaces)
  - Model loading
  - Propagation
- What remains **outside**
  - Domains
  - Variables
  - Constraints

CPEngine

CPUKernel



Sole requirement

- Abstract Propagator Concept

```
interface CPPropagator {
  ORUInt getId();
  void post();
}
```

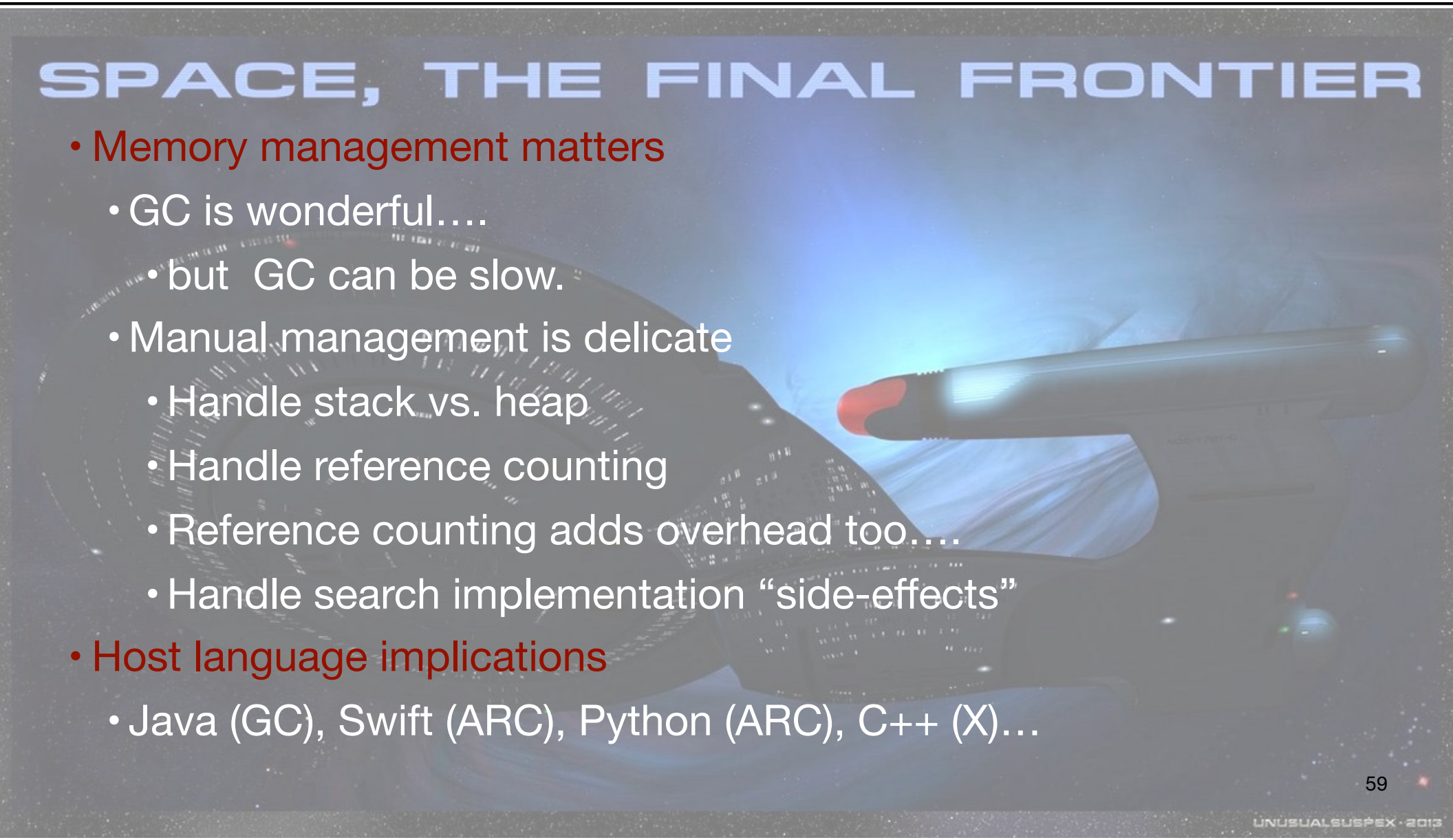
Numbering

Registration

What is **outside** the µKernel ?

- All variable types
  - int
  - float / real
  - bit vectors
  - sets
  - graphs …
- All propagators for each type of variables
- Illustration with
  - Finite Domain Integer Variables
  - Finite Domain Propagators

Storage Medium



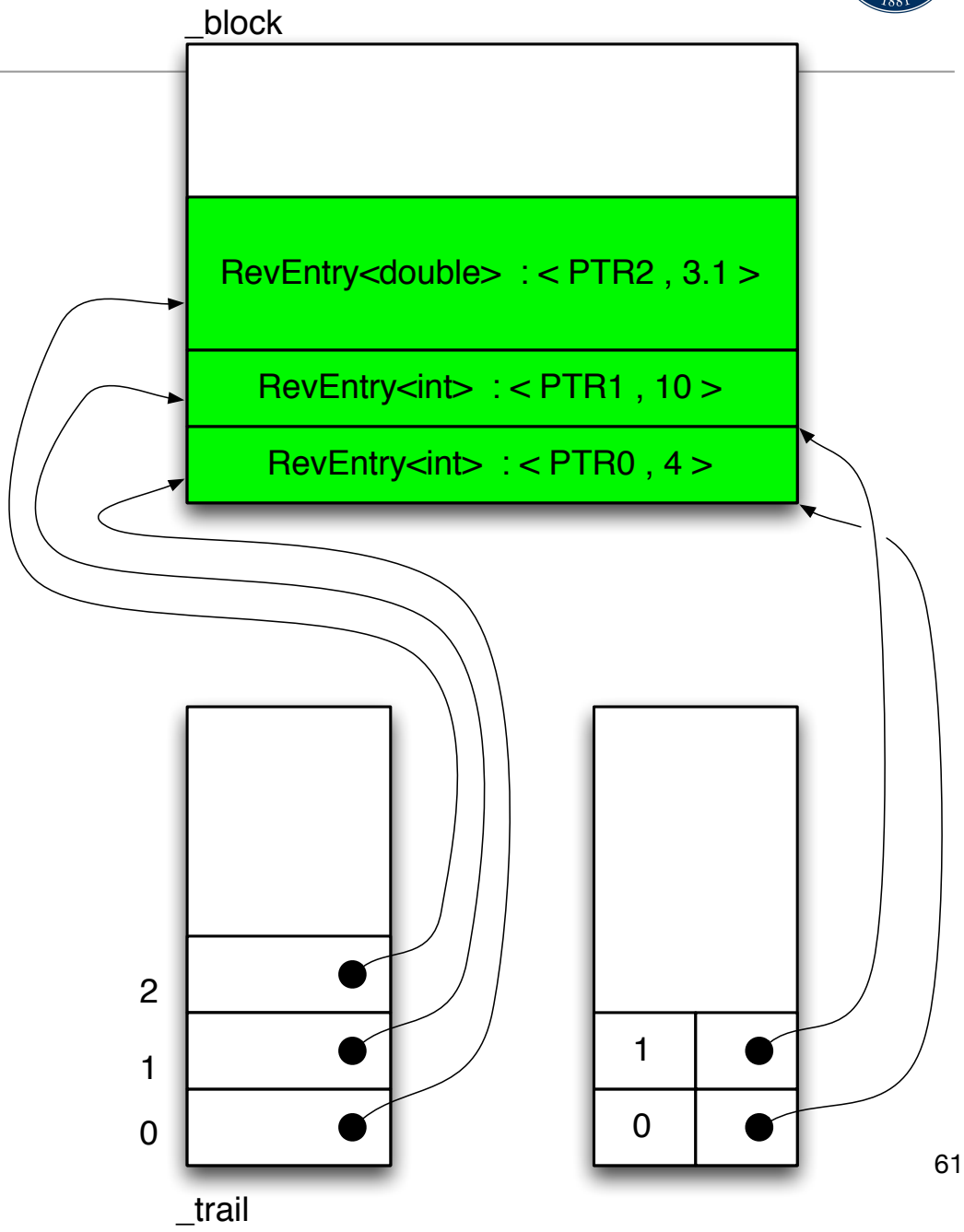
- Memory management matters
  - GC is wonderful….
    - but GC can be slow.
  - Manual management is delicate
    - Handle stack vs. heap
    - Handle reference counting
    - Reference counting adds overhead too….
    - Handle search implementation “side-effects”
- Host language implications
  - Java (GC), Swift (ARC), Python (ARC), C++ (X)…

State Management / Backtracking

- Done with a trail (Inheritance from CLP systems)
  - Keeps track of changes over time
  - **Undo** changes upon backtracking
    - Can be state undo (Address + old value)
    - Can be action undo (Closure!)
  - Timestamped trailing
- Note
  - There are alternatives (State replication : Gecode)

Context

- Two stacks
  - _trail : pointers to reversible entries
  - _tops : pairs with
    - size of trail
    - top of mem. pool
- Trailing records a change
  - Allocate reversible entry
  - Adds entry on top of stack
- Backtracking
  - **Undo** reversible changes



Context

```
struct Entry {
   virtual void restore() = 0;
};

class Context {
   std::stack<Entry*>                        _trail;
   std::stack<std::tuple<int,std::size_t>>  _tops;
   mutable int      _magic;
   char*            _block;
   std::size_t        _bsz;
   std::size_t       _btop;
public:
    Context();
   ~Context();
    typedef std::shared_ptr<Context> Ptr;
    void trail(Entry* e) { _trail.push(e);}
    int magic() const    { return _magic;}
    void incMagic()      { _magic++;}
    void push();
    void pop();
    friend void* operator new(std::size_t sz,Context::Ptr& e);
};
```

State Management in this talk

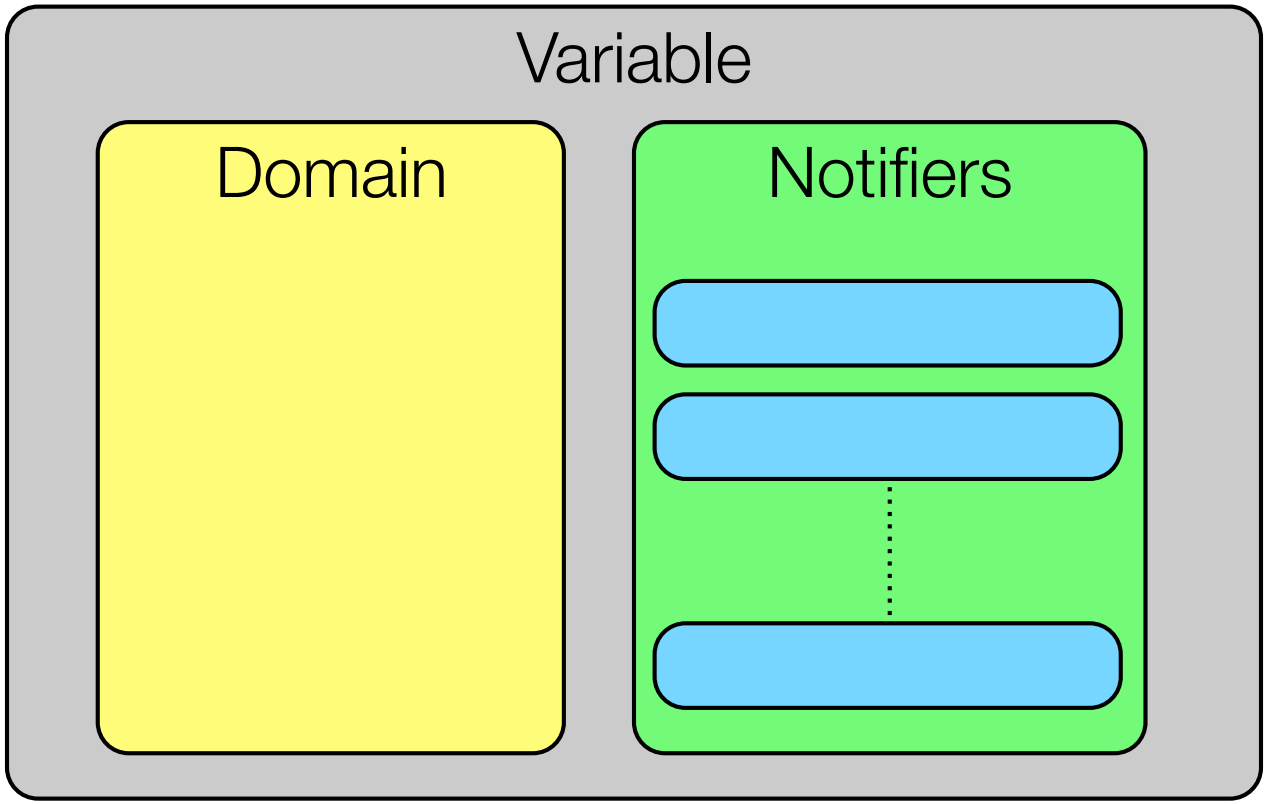
- State restoration supported through trailing [MiniCP,17]

```
template<class T> class rev {
    Context::Ptr  _ctx;
    int          _magic;
    T              _val;
pu  inline void* operator new(std::size_t sz,Context::Ptr& e) {
      char* ptr = e->_block + e->_btop;
      e->_btop += sz;
      return ptr;
    }
        T*  _at;
        T  _old;
    public:
        RevEntry(T* at) : _at(at),_old(*at) {}
        void restore() { *_at = _old;}
    };
    void trail(int nm) {
        _magic = nm;
        Entry* entry = new (_ctx) RevEntry(&_val);
        _ctx->trail(entry);
    }
};
```

Placement new

Finite-Domain Integer **Variable**

- Coalesce
  - Domain representation
  - Notifiers to domain events: **Closures!**



Abstract Concrete Var

- Yes!
  - Super class for all concrete vars
  - It only offers an API to “number” them!

```
class AVar {
protected:
    virtual void setId(int id) = 0;
    friend class CPSolver;
public:
    typedef handle_ptr<AVar> Ptr;
    AVar() {}
    virtual ~AVar() {}
};
```

Domain Events

- API to notify the occurrences of “Events”

```
struct IntNotifier   {
    virtual void bindEvt() = 0;
    virtual void domEvt(int sz) = 0;
    virtual void updateMinEvt(int sz) = 0;
    virtual void updateMaxEvt(int sz) = 0;
};
```

Domain (C++ Style)

- BitSet representation (other choices possible!)

```
class BitDomain {
    Engine::Ptr                 _ctx;
    std::vector<rev<int>>       _dom;
    rev<int>                    _min,_max,_sz;
    const int                   _imin,_imax;
public:
    typedef std::unique_ptr<BitDomain>  Ptr;
    BitDomain(Engine::Ptr ctx,int min,int max);
    int getMin() const;
    int getMax() const;
    int getSize() const;
    bool isBound() const;
    bool member(int v) const;
    void bind(int v,IntNotifier& x);
    void remove(int v,IntNotifier& x);
    void updateMin(int newMin,IntNotifier& x);
    void updateMax(int newMax,IntNotifier& x);
};
```

Reversible State

Concrete FD Variable Representation

```
template<typename T> class var {};


template<> class var<int> :public AVar, public IntNotifier {
   std::weak_ptr<CPSolver> _solver;
   BitDomain::Ptr          _dom;
   int                     _id;
   revList<std::function<void(void)>> _onBindList;
   revList<std::function<void(void)>> _onBoundsList;
protected:
    void setId(int id) override { _id = id;}
public:
    typedef handle_ptr<var<int>> Ptr;
    var<int>(CPSolver::Ptr& cps,int min,int max);
    ~var<int>();
};
```

Owner & Representation

Notifiers

Concrete FD Variable API

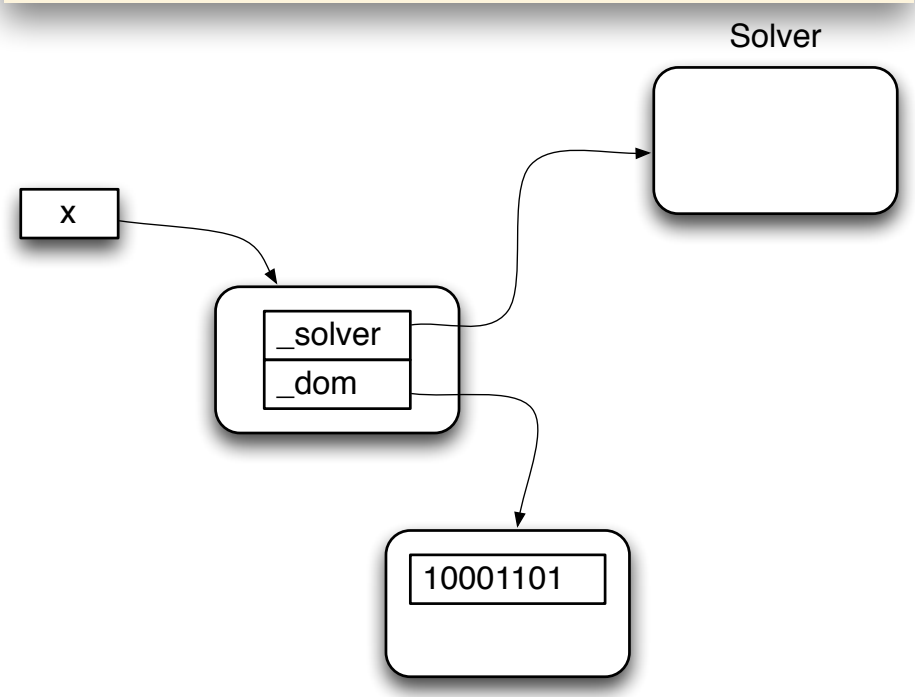
```
template<> class var<int> :public AVar, public IntNotifier {
    …
public:
    int getMin() const { return _dom->getMin();}
    int getMax() const { return _dom->getMax();}
    int getSize() const { return _dom->getSize();}
    bool isBound() const { return _dom->isBound();}
    bool contains(int v) const { return _dom->member(v);}
    void bind(int v);
    void remove(int v);
    void updateMin(int newMin);
    void updateMax(int newMax);
    void updateBounds(int newMin,int newMax);
    void bindEvt() override;
    void domEvt(int sz)  override;
    void updateMinEvt(int sz) override;
    void updateMaxEvt(int sz) override;
    auto whenBind(std::function<void(void)>&& f);
    auto whenChangeBounds(std::function<void(void)>&& f);
};
```

Updates

Registration

Data Flow on update (e.g., bind)

```
static inline void failNow()
{
    throw Failure;
}
```



x.bind(5)

```
void var<int>::bind(int v)
{
    _dom->bind(v,*this);
}
```

```
void BitDomain::bind(int v,IntNotifier& x)
{
    if (_sz == 1 && v == _min)
        return;
    if (v < _min || v > _max || !GETBIT(v))
        failNow();
    _min = v;
    _max = v;
    _sz  = 1;
    x.bindEvt();
}
```

```
void var<int>::bindEvt()
{
    CPSolver::Ptr solver = _solver.lock();
    for(auto& f :  _onBindList)
        solver->schedule(f);
    for(auto& f :  _onBoundsList)
        solver->schedule(f);
}
```

What are these f thingies ?

Event handling through Closures

- Responding to an *event* is done with a *closure.*
  - When something happens, we schedule the closure for execution
  - Scheduling can use priorities if we wish
- Revisit the variable definition!

```
template<> class var<int> :public AVar, public IntNotifier {
    std::weak_ptr<CPSolver> _solver;
    BitDomain::Ptr                _dom;
    int                            _id;
    revList<std::function<void(void)>> _onBindList;
    revList<std::function<void(void)>> _onBoundsList;
```

Notifiers

Reversible list of anonymous functions void → void

Priorities ?

- Easy to handle
  - Make it a list of pairs < int, void → void >
  - Use the integer as a priority when scheduling the closure
- Data structure change

```
revList<std::tuple<int,std::function<void(void)>>> _onBindList;
```

Events ?

- It's your call!
- Obvious choices for FD variables
  - min Increase
  - max Decrease
  - both bound change
  - variable bound
  - arbitrary domain change
  - value loss (with the value being lost)

Key Observations

- The micro-kernel does not know what is scheduled!
  - This can be variable-centric
  - This can be constraint-centric
  - It's up to you based on what the closure does.
- This supports multiple AC-style algorithms, e.g.,
  - AC3
  - AC5
- Closure receiving an argument (e.g., value lost) can….
  - Wrap the argument in a temporary polymorphic object
  - e.g., Value class to accommodate typing

Propagator example x != y + c

- Class definition for bound-consistent propagator

```
class NEQBinBC : public Propagator { // x != y + c
  var<int>::Ptr _x,_y;
  int _c;
public:
  NEQBinBC(var<int>::Ptr& x,var<int>::Ptr& y,int c)
     : _x(x),_y(y),_c(c) {}
  void post() override;
};
```

Propagator example x != y + c

- post method *does everything!*

```
void NEQBinBC::post() {
    if (_x->isBound())
        _y->remove(_x->getMin() - _c);
    else if (_y->isBound())
        _x->remove(_y->getMin() + _c);
    else {
        _x->whenBind([this] {
            _y->remove(_x->getMin() - _c);
        });
        _y->whenBind([this] {
            _x->remove(_y->getMin() + _c);
        });
    }
}
```

Disabling ?

```
class NEQBinBC : public Propagator { // x != y + c
  var<int>::Ptr _x,_y;
```

```
void NEQBinBC::post() {
    if (_x->isBound())
        _y->remove(_x->getMin() - _c);
    else if (_y->isBound())
        _x->remove(_y->getMin() + _c);
    else {
       hdl[0] = _x->whenBind([this] {
              _y->remove(_x->getMin() - _c);
              hdl[0]->detach();
              hdl[1]->detach();
          });
       hdl[1] = _y->whenBind([this] {
              _x->remove(_y->getMin() + _c);
              hdl[0]->detach();
              hdl[1]->detach();
          });
    }
}
```

Supporting Value Events (AC5)

```
void NEQBinDC::post() {
  if (_x->isBound())
      _y->remove(_x->getMin() - _c);
  else if (_y->isBound())
      _x->remove(_y->getMin() + _c);
  else {
     _x->updateMinAndMax(_y->getMin() + _c,_y->getMax() + _c);
     _y->updateMinAndMax(_x->getMin() - _c,_x->getMax() - _c);
     for(auto v = _x->getMin(); v <= _x->getMax(); v++)
        if (!_x->member(v)) _y->remove(v - _c);
     for(auto v = _y->getMin(); v <= _y->getMax(); v++)
        if (!_y->member(v)) _x->remove(v + _c);
     _x->whenLoseValue([this](const Value& v) { _y->remove(v - _c);});
     _y->whenLoseValue([this](const Value& v) { _x->remove(v + _c);});
  }
}
void var<int>::loseValEvt(int v) {
    CPSolver::Ptr solver = _solver.lock();
    for(auto& f : _onLoseValList))
      solver->scheduleValue(f,Value(v));
}
```

Propagation Engine

- Purpose
  - Track the concrete variables
  - Track the concrete propagators
  - Provide the propagation logic
  - Maintain basic statistics

Propagation Engine class

```
typedef std::reference_wrapper<std::function<void(void)>> Closure;

class Engine {                                  // # microkernel
   Context::Ptr                      _ctx;      // # backtracking context
   std::list<AVar::Ptr>              _iVars;    // # all concrete variables
   std::list<Propagator::Ptr> _iCstr;           // # all propagators
   std::deque<Closure>               _queue;    // # propagation queue
   bool                             _closed;    // # model posted?
   int                                 _nbc;    // # choices
   int                                 _nbf;    // # fails
   int                                 _nbs;    // # solutions
   rev<Status>                          _cs;
public:
    …
};
```

Propagation Engine class

```
typedef std::reference_wrapper<std::function<void(void)>> Closure;

class Engine {
   …
public:
   template<typename T> friend class var;
   typedef std::shared_ptr<CPSolver> Ptr;
   CPSolver();
   ~CPSolver();
   void registerVar(AVar::Ptr avar);
   void schedule(std::function<void(void)>& cb);
   Status propagate();
   Status add(Propagator::Ptr c);
   void close();
   Context::Ptr context() { return _ctx;}
   Status status() const  { return _cs;}
   void incrNbChoices()   { _nbc += 1;}
   void incrNbSol()       { _nbs += 1;}
};
```

Scheduling

- Trivial implementation
  - Stores the given closure into the queue
  - Note that queue holds a *reference* to the closure (which is held in a variable somewhere….)

```
void schedule(std::function<void(void)>& cb)
{
    _queue.emplace_back(cb);
}
```

Propagation

- Minor simplification (not showing priorities)

```
Status CPSolver::propagate() {
    try {
        while (!_queue.empty()) {
            auto cb = _queue.front();
            _queue.pop_front();
            cb();
        }
       return _cs = Suspend;
    } catch(Status x) {
        _queue.clear();
        _nbf += 1;
        return _cs = Failure;
    }
}
```

```
static inline void failNow()
{
    throw Failure;
}
```

Overview

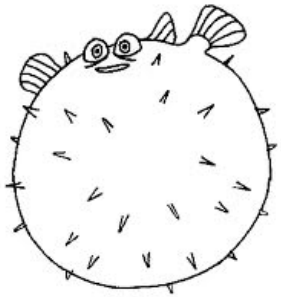
- Motivation
- Modeling Layer
  - Variables / Constraints
  - Transformations, reformulation and concretizations
  - Consistency handling
- Solving Layer
  - Microkernel
  - Events / Propagators and Propagation
  - Views
- Hybrids
- Search building blocks
- Parallel support

Views [Michel,14]

- Views are useful (in order)
  - To reduce software complexity and “variants” of propagators
  - To cut back on storage for simple arithmetic relation
  - To speed up simple constraint propagation
- Views can be less than straightforward
  - Sometimes carry restrictions on what can be captured
  - Introduce unexpected side-effects
  - Break idempotence assumptions

Typical Example

- Consider the queens problem with globals
  - 3 constraints over variable array **x**
    - alldifferent(**x**)
    - alldifferent(**all(i in D) x[i] + i**)
    - alldifferent(**all(i in D) x[i] - i**)
  - Constraint signatures?
    - alldifferent(**var**<CP>{int} [] array)
    - alldifferent(**expr**<CP>{int}[] array)
  - Thus **two** different constraints and implementations!



Decomposition…

- It's not pretty but it can solve the problem
- It can be automated with reformulations

```
Model m();
var<CP>{int} x[i in D](D);

m.post(alldiffere
m.post(alldiffere
m.post(alldiffere
```

Yet… Huge Overheads

in Time

in Space

```
Model m();
var<CP>{int} x[i in D](D);
 xp[i in D];
 xn[i in D];

))
(p[i] == x[i] + i);
))
(n[i] == x[i] - i);
fferent(x));
fferent(xp));
fferent(xn));
```

A Better Solution

- Produce a design mechanism that...
  - Delivers the ***illusion of variety***
  - While retaining a unique lean implementation

Variable Adapter a.k.a. a **View**

"Virtual" variable
Adapter uses a "mapping" function
to translate for the "real" variable

Related Work

- The idea is certainly not new
  - Indexicals — Prolog-style languages
    - [Carlsson97]
    - [VHT92]
  - Views — Ilog Solver,Gecode, COMET
- Gecode views enable *derived* propagators
  - [Schulte08: Perfect Derived Propagators]
  - [Schulte13: View-based propagator derivation]
- CASPER supports general views but limited to bound-consistency
  - [Correia13: View-based propagation of decomposable constraints]

Gecode Views

- Template-based
  - But essence is general and could rely on polymorphism
- Strike a specific balance between
  - Expressiveness
  - Flexibility
- Limitation?
  - injective context
- Henceforth

Variable Views

Domain Views

- Offers a new way to support views
  - Using sub-type polymorphism
  - But it can be done just as well with templates!
- Generalization
  - To support injective use [like variable-views]
  - To support **non-injective use** [new!]
  - To fully support **domain-consistency** [new!]
  - At virtually no cost [w.r.t. variable-views]

Domain Views

Variable

- First, the interface

```
interface Variable

    bool member(V v);
    bool remove(V v);

    void watch(C c);
    void watchValue(C c);

    void wake();
    void wakeValue(V v);
```

Query & update

Registration

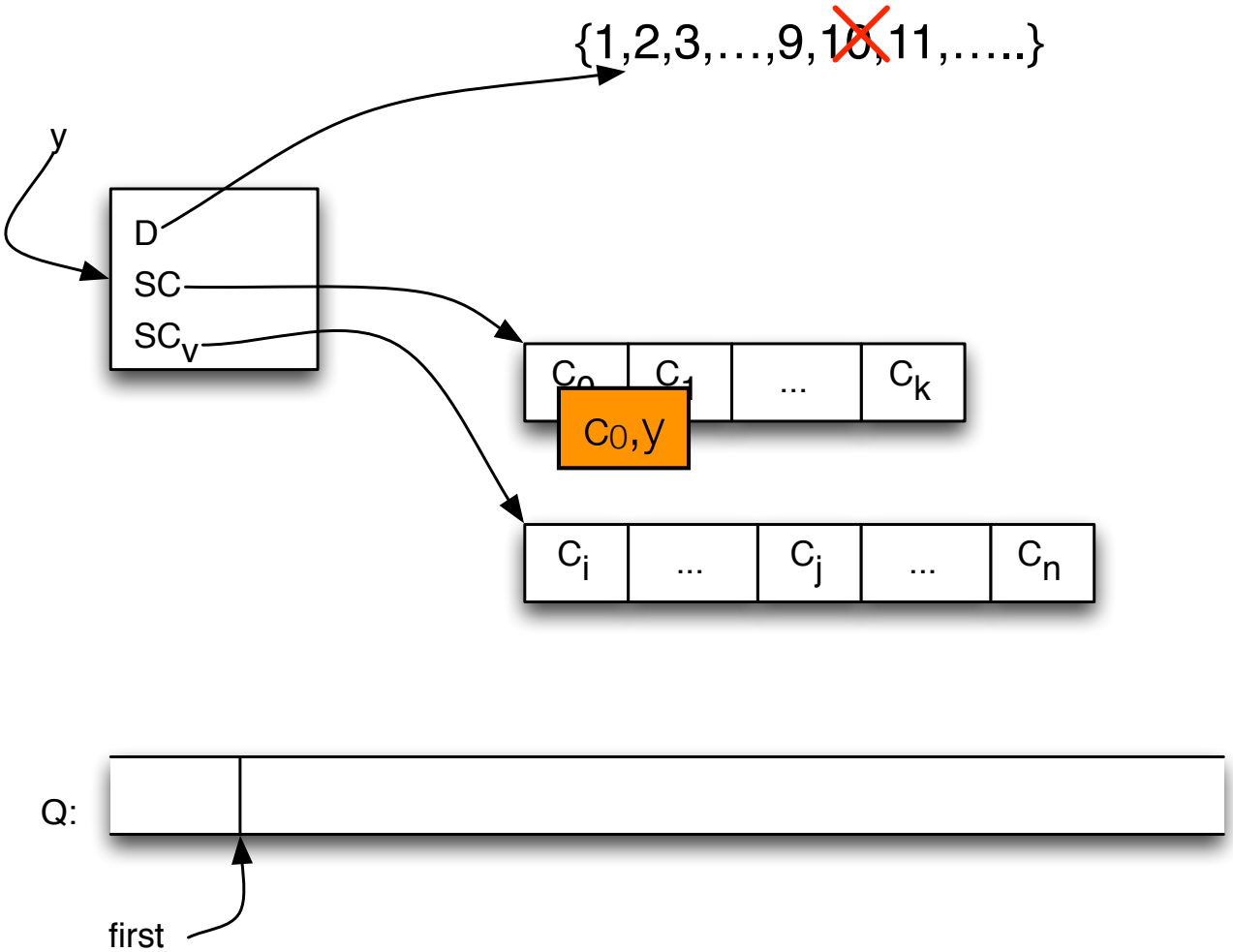
Response notification

Classic Domain Variable

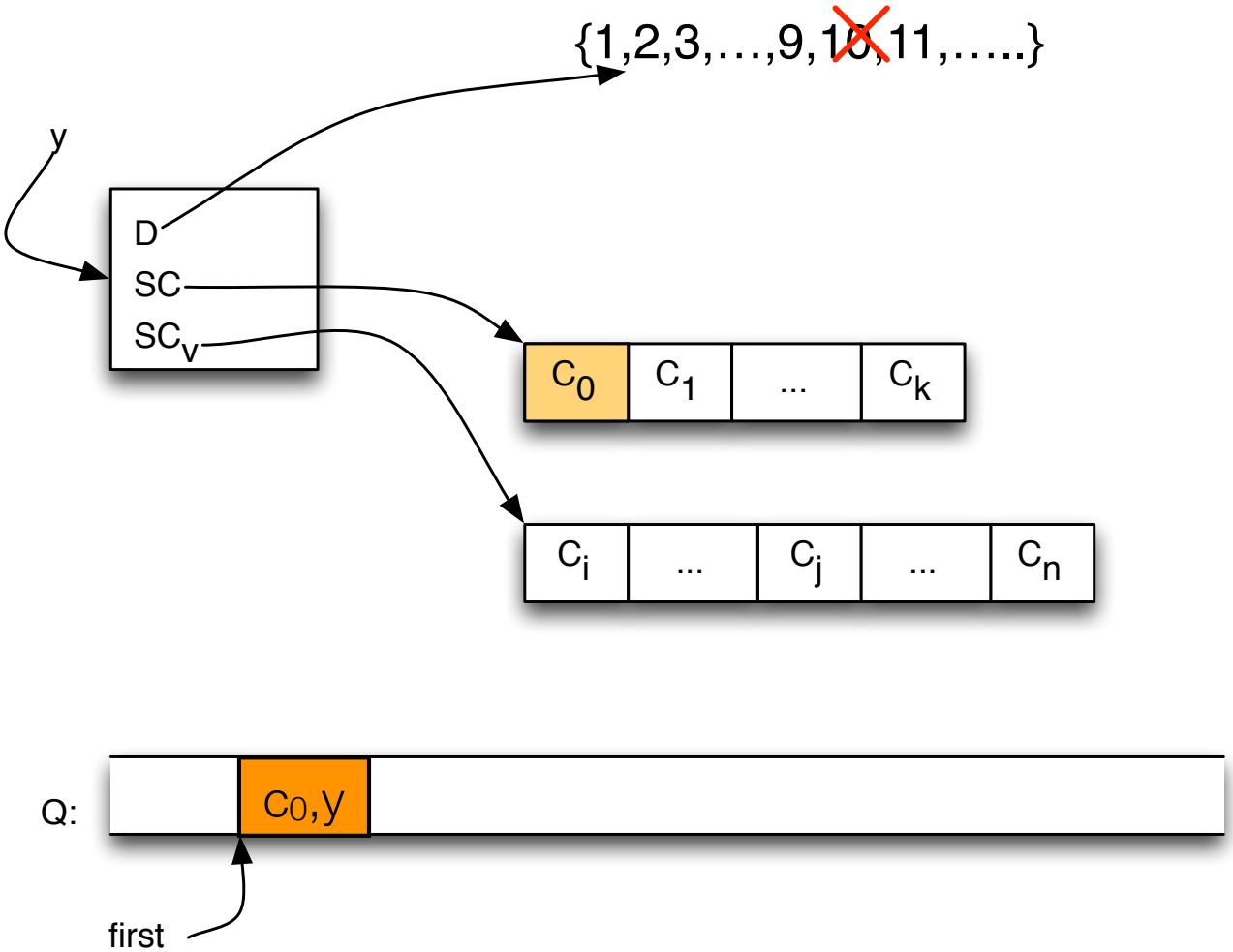
```
implementation DomainVariable
    {V} D;
    {C} SC;
    {C} SCv;

    DomainVariable({V} Do)  { D=Do;SC=∅;SCv=∅;}
    bool member(V v)        { return v ∈ D;}
    bool remove(V v)        {
        if (v ∈ D)
            D = D \ {v};
            wake();
            wakeValue(v);
    }
    void watch(C c)         { SC := SC    ∪ {c};}
    void watchValue(C c)    { SCv := SCv  ∪ {c};}
    void wake()             { Q := Q ∪ { <c,this>   : c ∈ SC;}
    void wakeValue(V v)     { Q := Q ∪ { <c,this,v> : c ∈ SCv;}
```

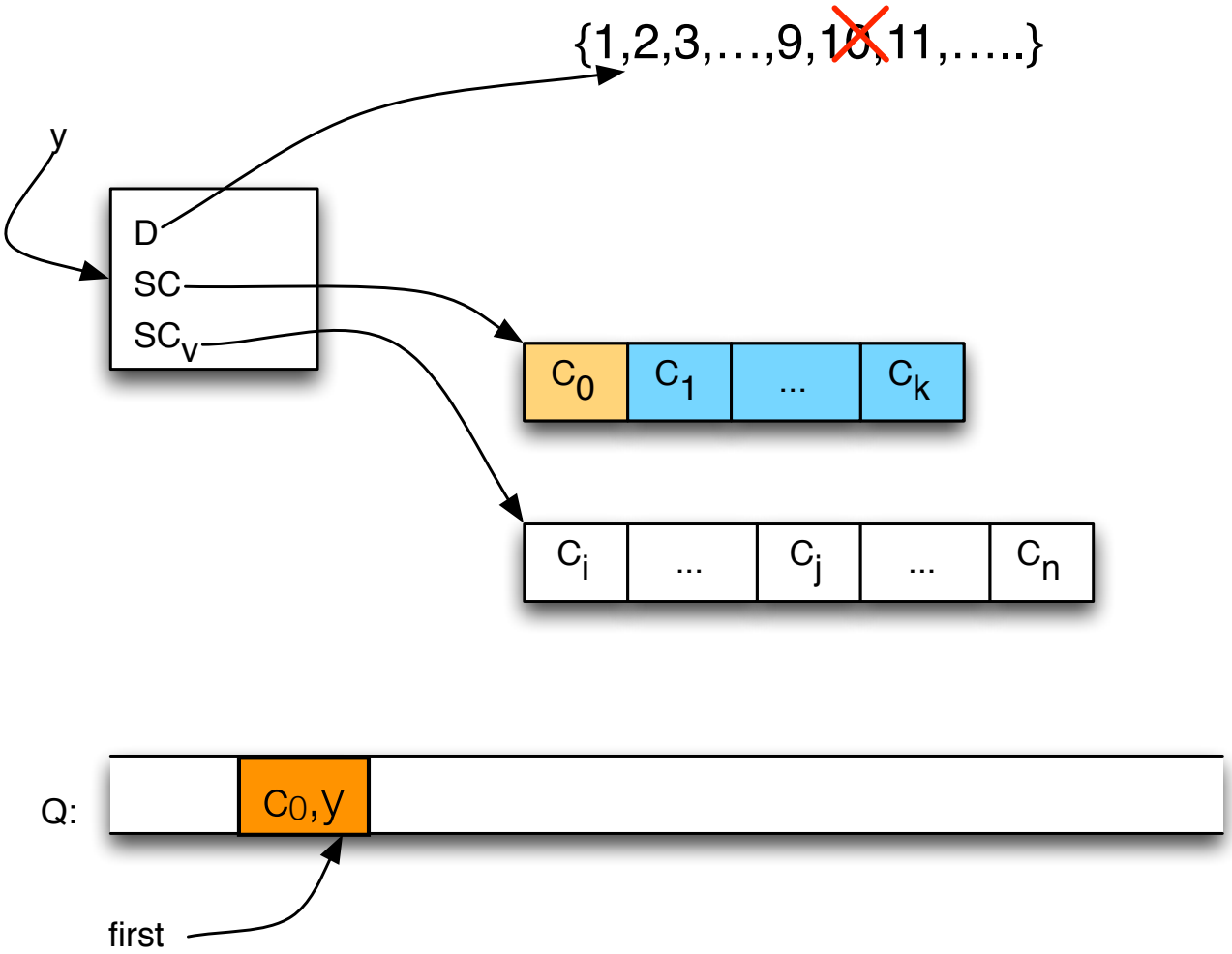
Animation



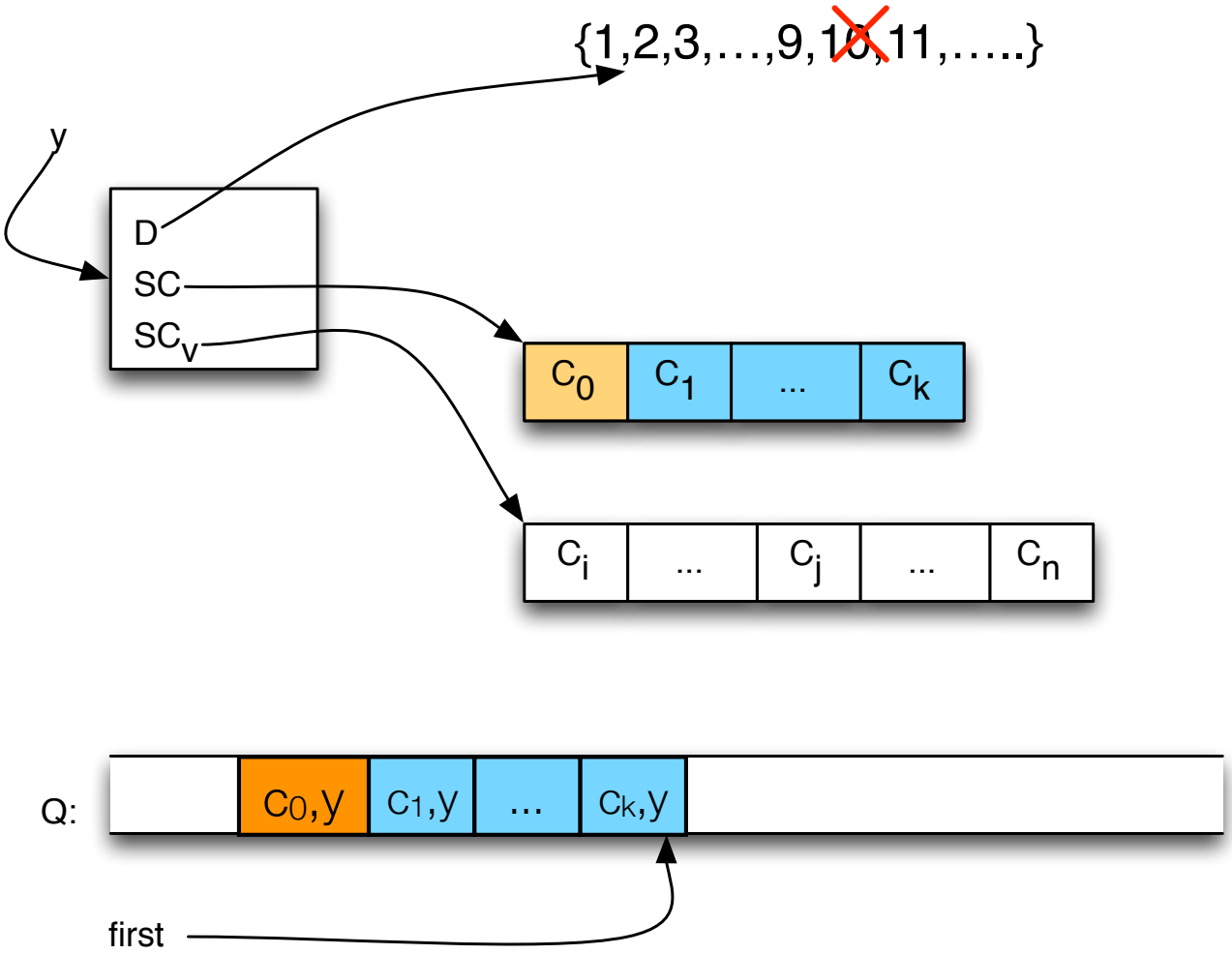
Animation



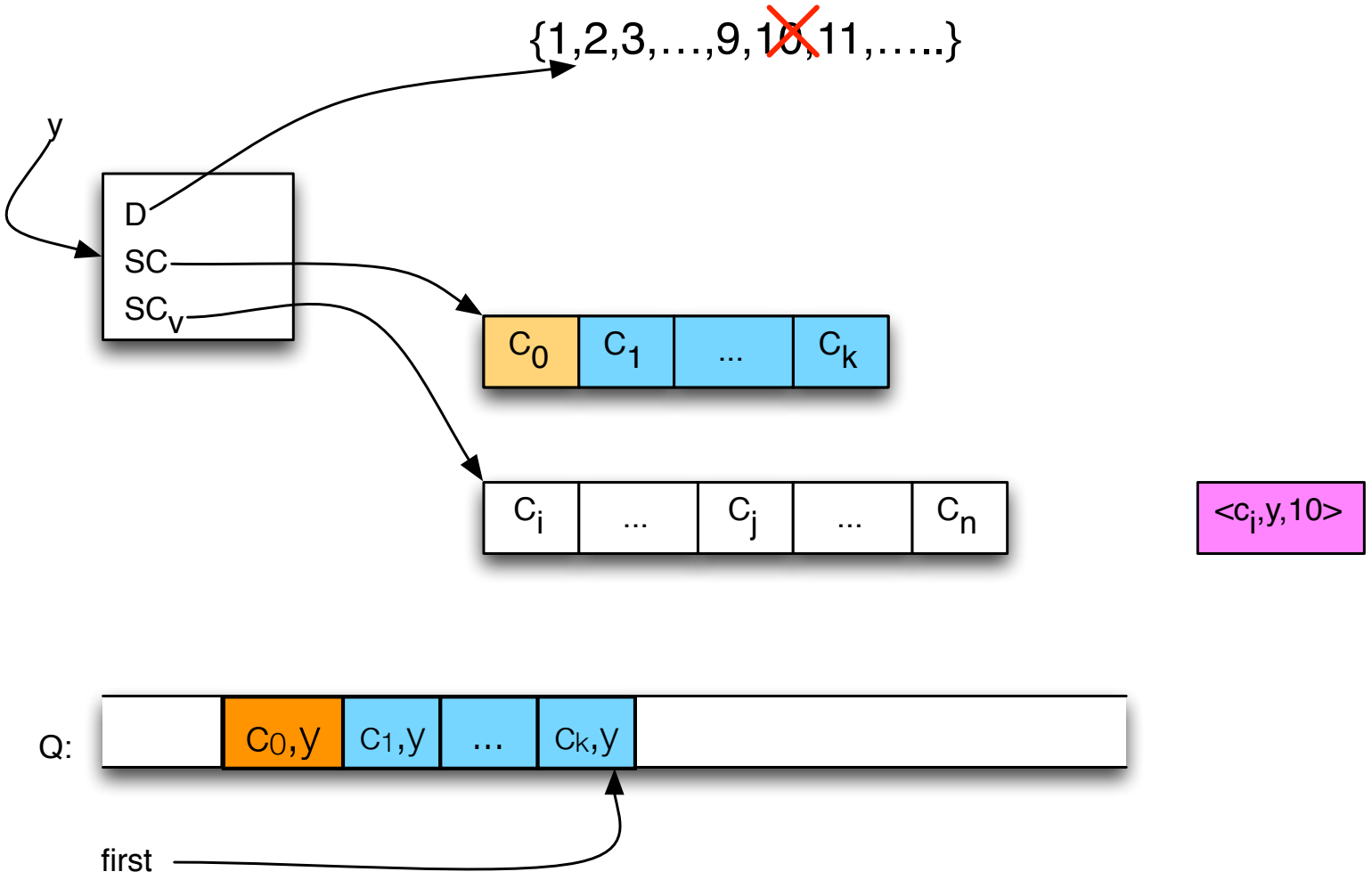
Animation



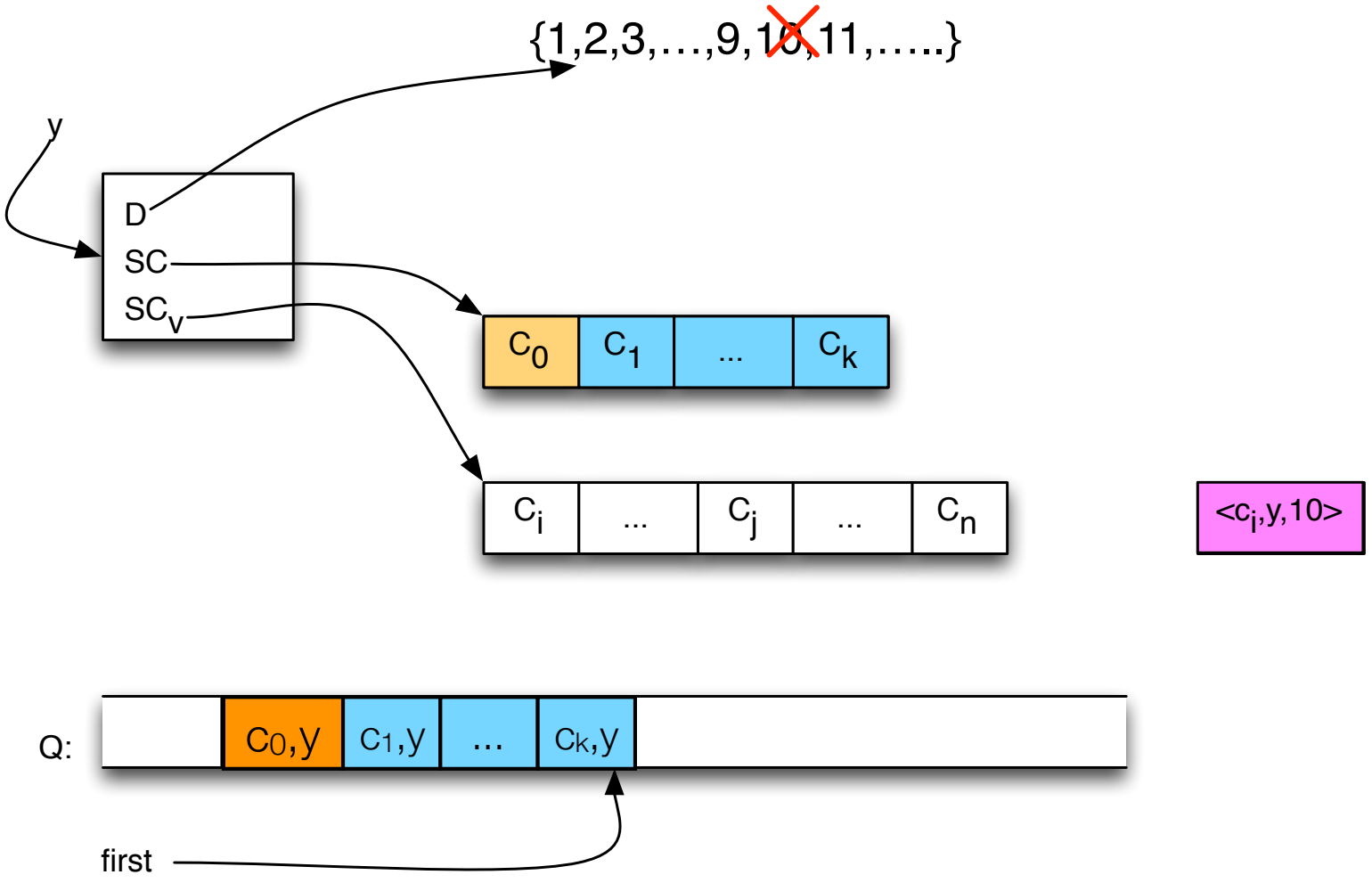
Animation



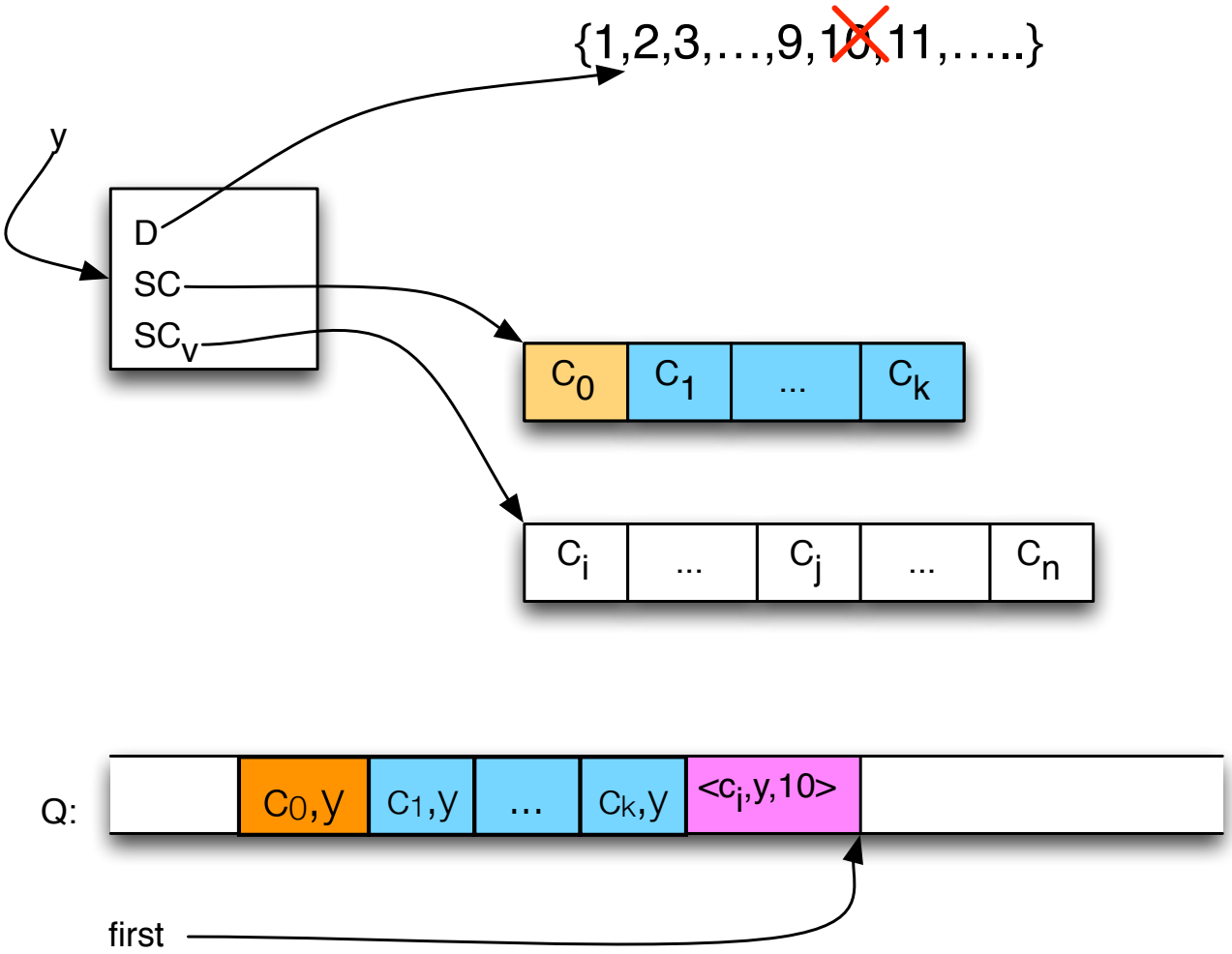
Animation



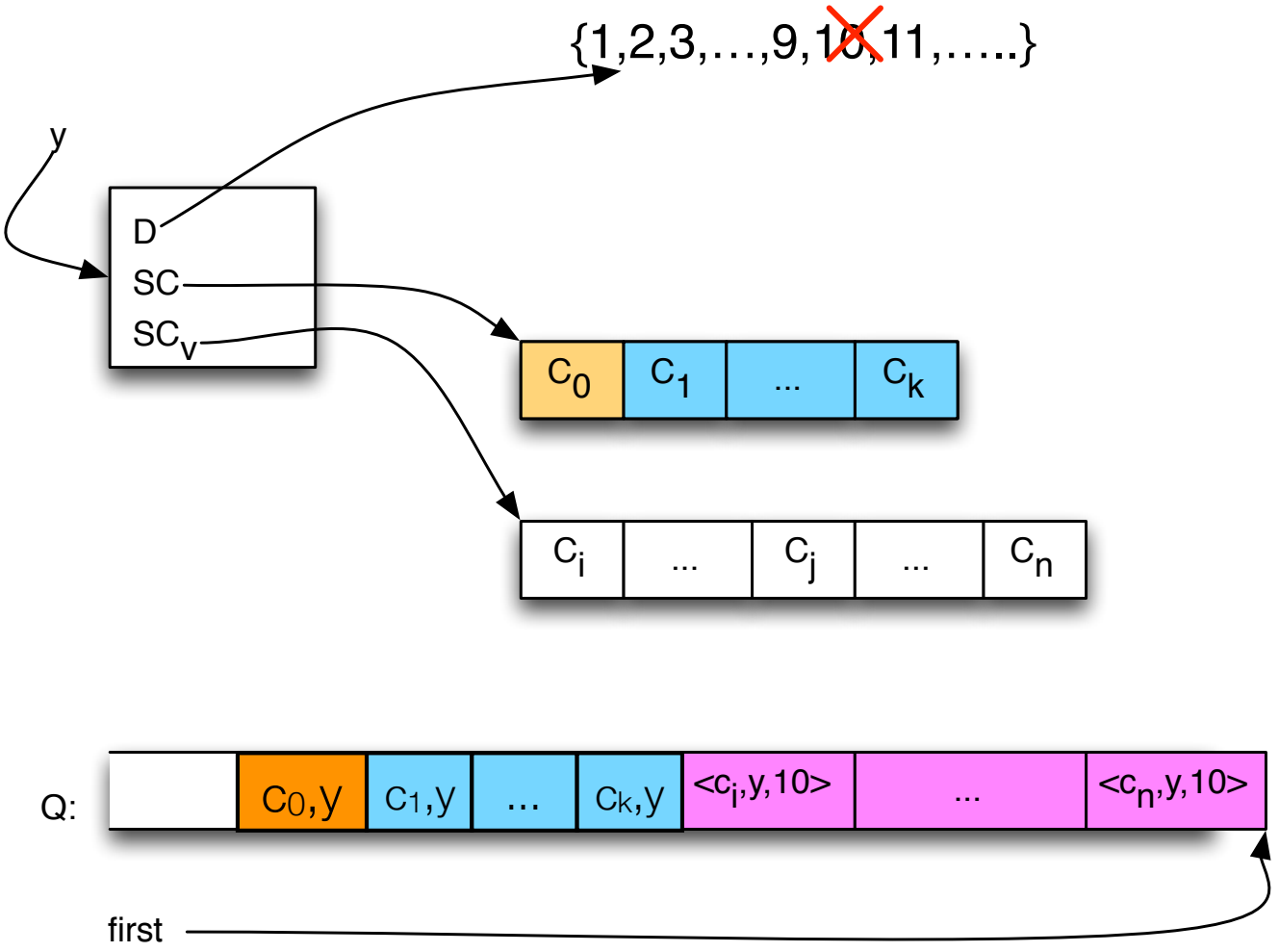
Animation



Animation



Animation


{1,2,3,…,9,10,11,…..}
y
D
SC
SCv
C0
C1
...
Ck
Ci
...
Cj
...
Cn
Q:
C0,y
C1,y
…
Ck,y
<ci,y,10>
...
<cn,y,10>
first

Propagation

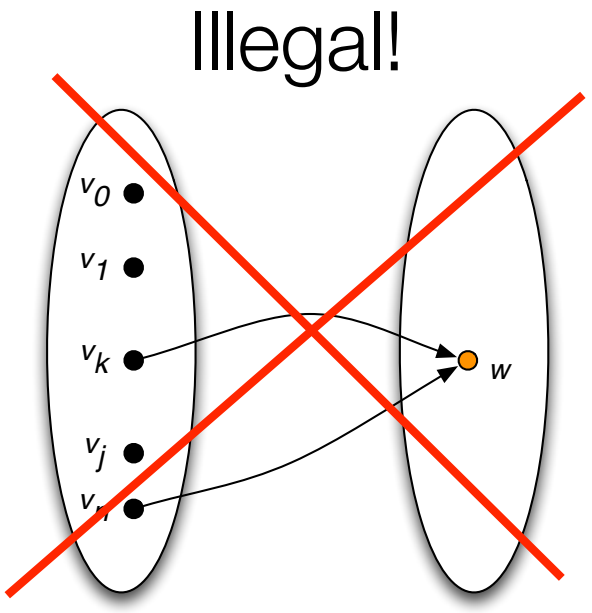
- Imagine C_n is a Domain-Consistent equality x == y + c
  - When popping <C_n,y,10>
    - Notify C_n that variable y lost value 10: C_n.valueLoss(y,10)

Propagation

```
implementation EqualDC                          // x == y + c
    {X} _x,_y;
    int    _c;
    EqualDC({X} x, {X} y,int c) { _x = x;_y = y;_c =c;}
    void post()                 {
        for(k in D(_y))
            if (k + _c ∉ D(_x))  y.remove(k);
        for(k in D(_x))
            if (k - _c ∉ D(_y))  x.remove(k);
        if (!bound(_x)) _x.watchValue(self);
        if (!bound(_y)) _y.watchValue(self);
        _x.whenValueLost([](int theValue) {_y.remove(theValue - _c);});
        _y.whenValueLost([](int theValue) {_x.remove(theValue + _c);});
    }
```

Views: ψ

- Based on injective function



- Classic examples
  - Shift view $y \leftarrow x + c$
  - Scale view $y \leftarrow c * x$
  - Affine view $y \leftarrow a * x + b$

- Inverse easy to define
  - ψ^{-1}
  - *Example: if* $\psi(v) = v + c$ *then* $\psi^{-1}(v) = v - c$

Standard Domain Variable (Again!)

- Abstract Implementation

```
implementation DomainVariable
    {V} D;
    {C} SC;
    {C} SCv;

    DomainVariable({V} Do)      { D=Do;SC=∅;SCv=∅;}
    bool member(V v)            { return v ∈ D;}
    bool remove(V v)            {
        if (v ∈ D)
            D = D \ {v};
            wake();
            wakeValue(v);
    }
    void watch(C c)                     { SC := SC   ∪ {c};}
    void watchValue(C c)                { SCv := SCv ∪ {c};}



    void wake()                         { Q := Q ∪ { <c,this>: c ∈ SC;}
    void wakeValue(V v)                 { Q := Q ∪ { <c,this,v>   : c ∈ SCv;}
```

Refining Standard Variables

- Revise the watching API!

```
implementation DomainVariable
     {V}           D;
     {<C,X>}       SC;
     {<C,X,F>}     SCv;

     DomainVariable({V} Do)     { D=Do;SC=∅;SCv=∅;}
     bool member(V v)           { return v ∈ D;}
     bool remove(V v)           {
          if (v ∈ D)
               D = D \ {v};
               wake();
               wakeValue(v);
     }
     void watch(C c,X y)                  { SC := SC   ∪ {<c,y>};}
     void watchValue(C c,X y,F ψ)         { SCv := SCv ∪ {<c,y,ψ>};}
     void watch(C c)                      { this.watch(c,this);}
     void watchValue(C c)                 { this.watch(c,this,λk.k);}

     void wake()                          { Q := Q ∪ SC; };
     void wakeValue(V v)                  { Q := Q ∪ { <c,x,ψ(v)>   : <c,x,ψ> ∈ SCv;};}
```

Classic Variable View<ψ>

```
implementation VariableView<ψ>    // this ← ψ(x)
    X    x;

    VariableView(X theVar)    { x = theVar;}
    bool member(V v)          {
        if ψ⁻¹(v) ≠ ⊥ return x.member(ψ⁻¹(v));else return NO;
    }
    bool remove(V v)          {
        if ψ⁻¹(v) ≠ ⊥ return x.remove(ψ⁻¹(v));else return YES;
    }
    void watch(C c,X y)                 { x.watch(c,y);}
    void watchValue(C c,X y,F ϕ)        { x.watchValue(c,y,ϕ ∘ ψ);}

    void watch(C c)                     { x.watch(c,this);}
    void watchValue(C c)                { x.watch(c,this,ψ);}

    void wake()                         { Q := Q ∪ SC;}
    void wakeValue(V v)                 { Q := Q ∪ { <c,x,ψ(v)>   : <c,x,ψ> ∈ SCv;};}
```

Bottom Line

- Works only for Injective Views
- Can be made to support AC-5
- Save the injective function to remap values on wakeup
- Must compose injective function for views on views
- Nota bene
  - One can optimize this away…
  - But the solution clutters the variable API with a map function.

Domain Views?

- Change the organization
- Relax the limitation of variable views
  - Full support for non-injective views.
- Restore a simpler state management in variables
- Key insights
  - Variable is aware of views defined on it.
  - Responders to notifications are back into the view

Key Solution

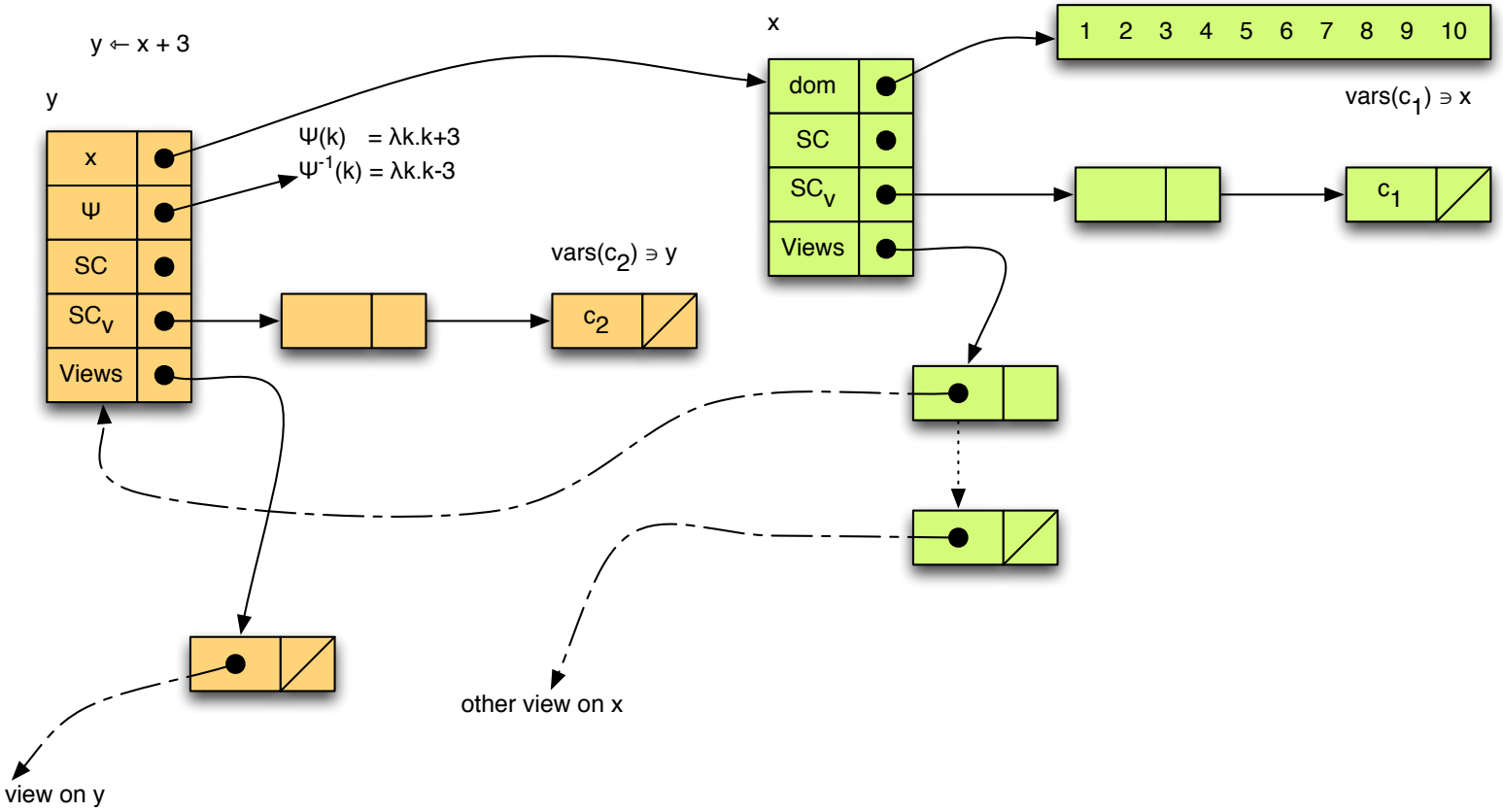
```
implementation DomainVariable
     {V} D;
     {C} SC;
     {C} SCv;
     {X} Views;
     DomainVariable({V} Do)     { D=Do;SC=∅;SCv=∅;Views = ∅;}
     void addView(X x)          { Views := Views ∪ {x};}
     bool member(V v)           { return v ∈ D;}
     bool remove(V v)           {
          if (v ∈ D)
               D = D \ {v};
               wake();
               wakeValue(v);
               forall y ∈ Views do
                    y.wake();
                    y.wakeValue(v);
     }
     void watch(C c)                    { SC := SC   ∪ {c};}
     void watchValue(C c)               { SCv := SCv ∪ {c};}
     void wake()                        { Q := Q ∪ { <c,this>     : c ∈ SC;}
     void wakeValue(V v)                { Q := Q ∪ { <c,this,v>   : c ∈ SCv;}
```

DomainView<ψ>

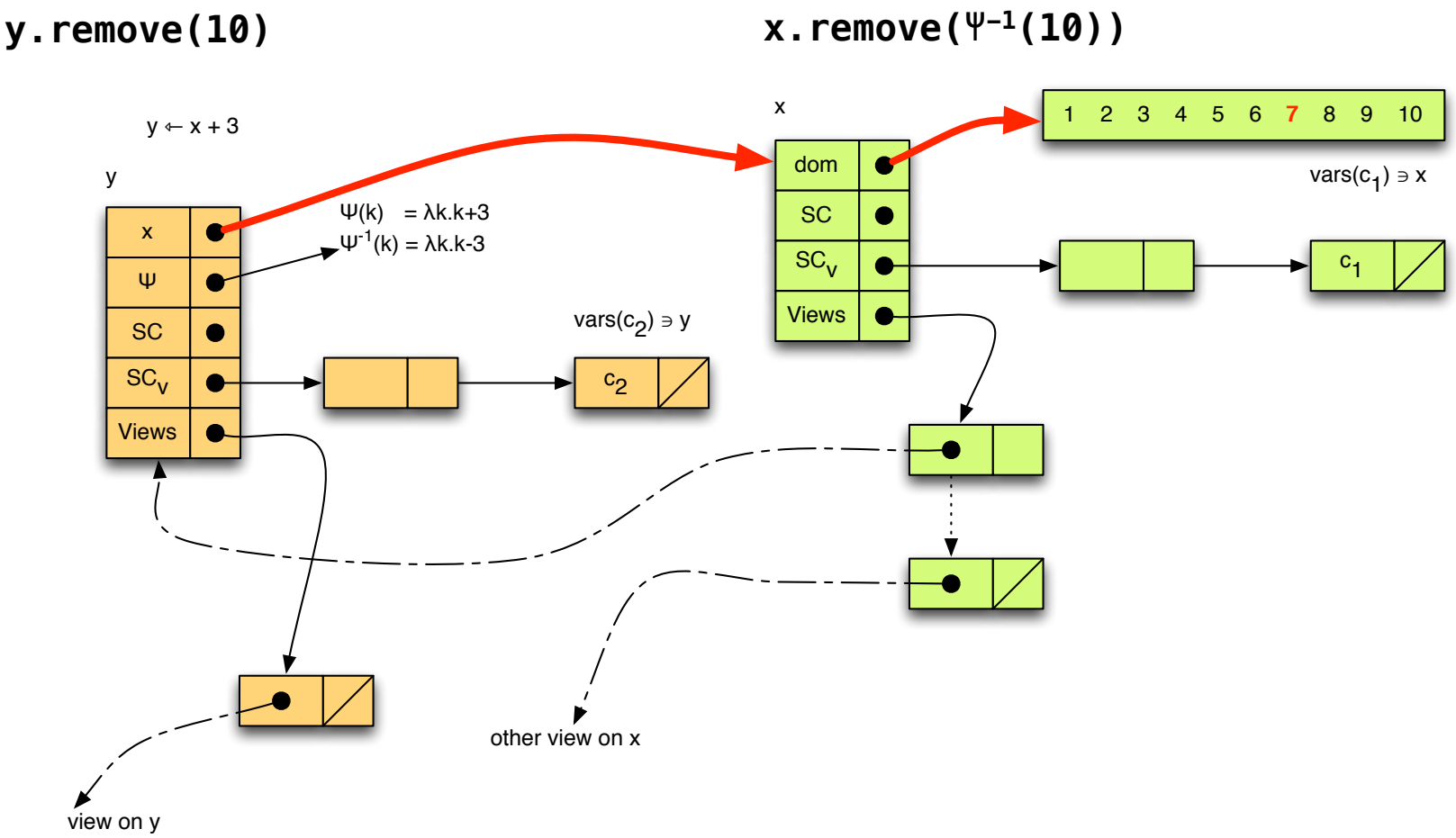
```
implementation DomainView<ψ>      // this ← ψ(x)
    X    x;
    {C}  SC;
    {C}  SCv;
    {X}  Views;
    DomainView(X theVar)      { x = theVar;SC=∅;SCv=∅;Views = ∅;}
    void addView(X x)         { Views := Views ∪ {x};}
    bool member(V v)          {
        if ψ-1(v) ≠ ⊥ return x.member(ψ-1(v));else return NO;
    }
    bool remove(V v)          {
        if ψ-1(v) ≠ ⊥ return x.remove(ψ-1(v));else return YES;
    }
    void watch(C c)                   { SC := SC   ∪ {c};}
    void watchValue(C c)              { SCv := SCv ∪ {c};}
    void wake()  {
        Q := Q ∪ { <c,this> : c ∈ SC};
        forall(y in Views) y.wake();
    }
    void wakeValue(V v)  {                          // v is a value from D(x)
        Q := Q ∪ { <c,this,ψ(v)> : c ∈ SCv;};
        forall(y in Views) y.wakeValue(ψ(v));
    }
```

Illustration

`y.remove(10)`

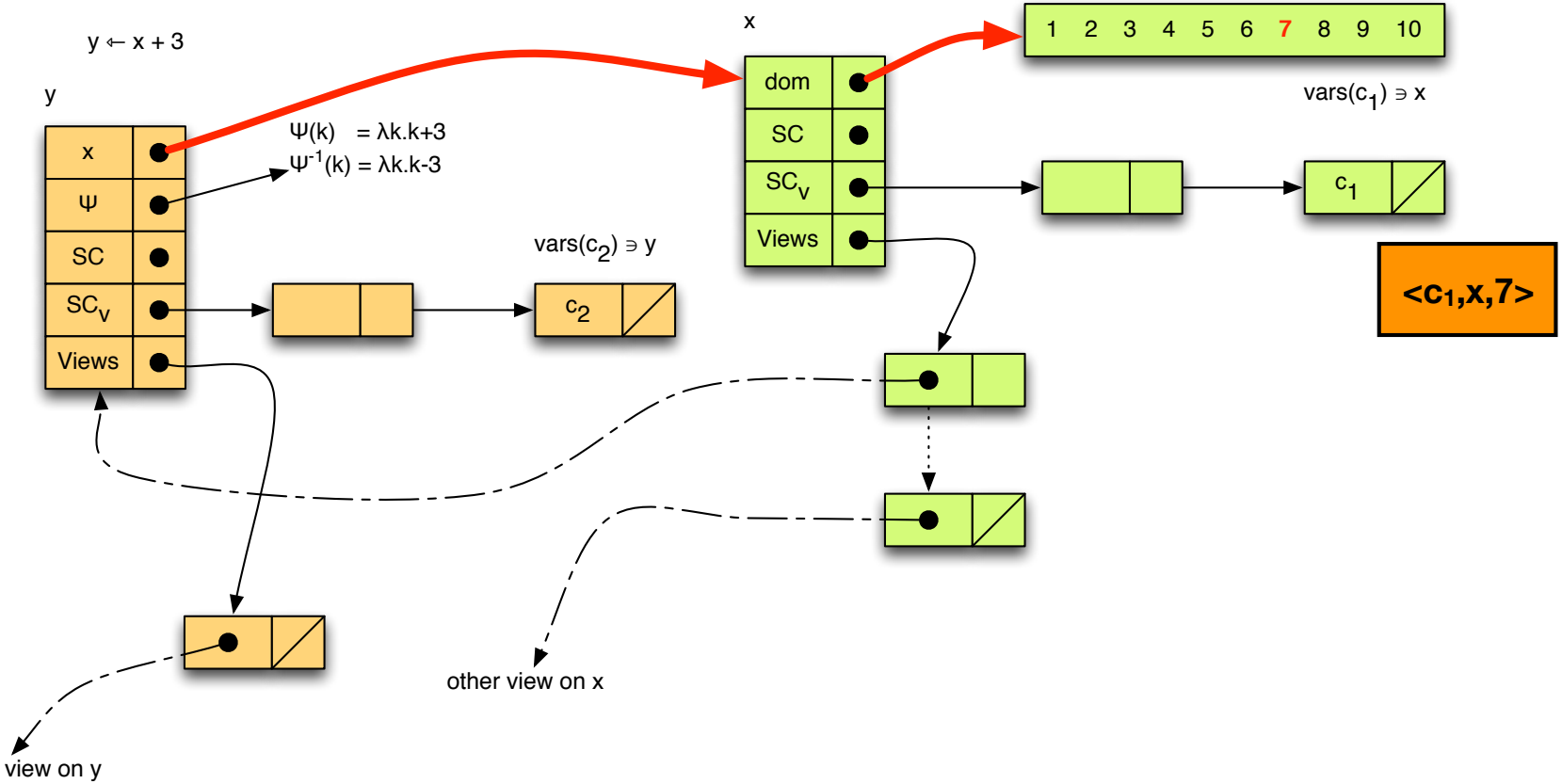


Illustration



Illustration

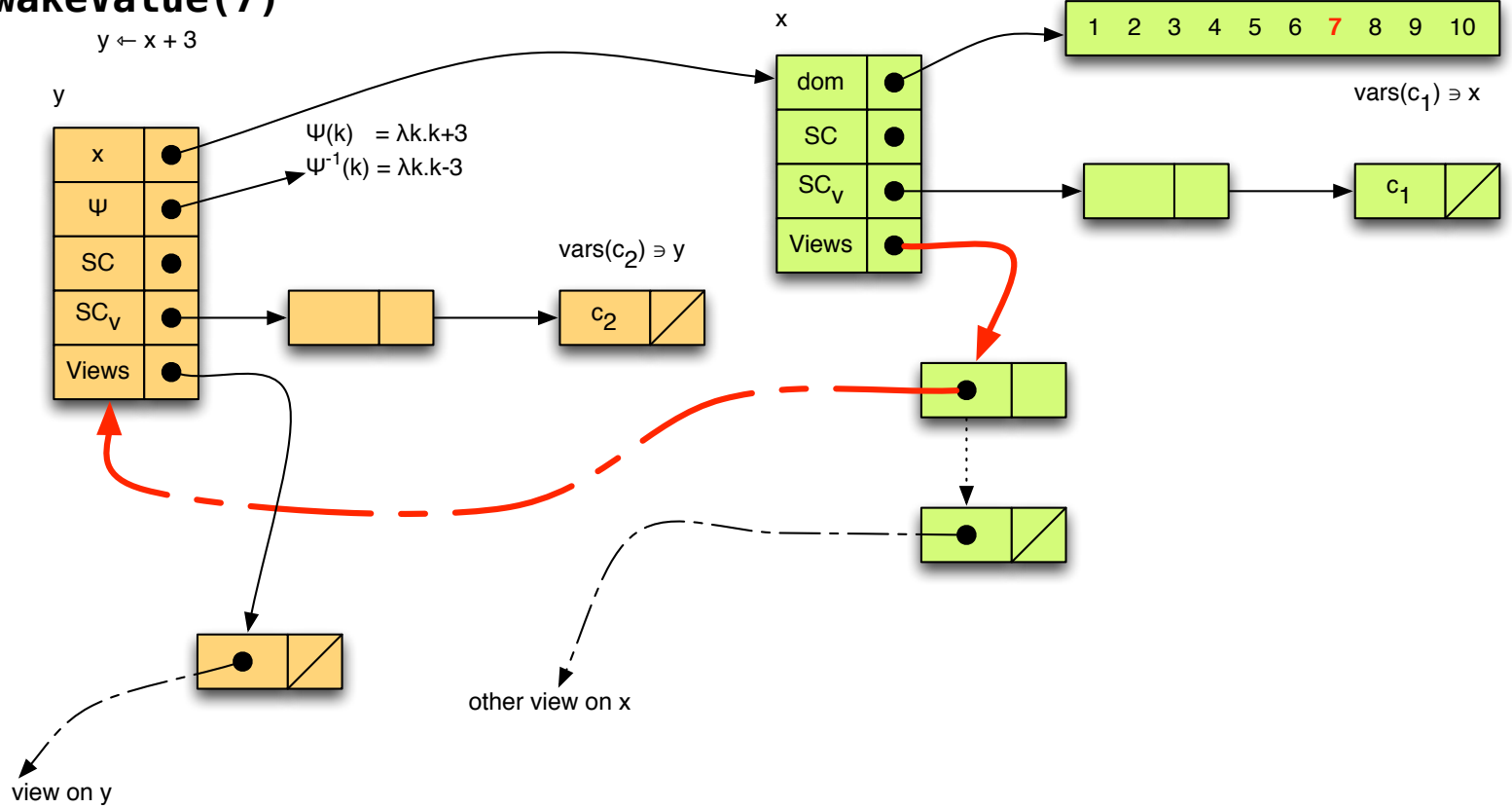
`y.remove(10)`



Q

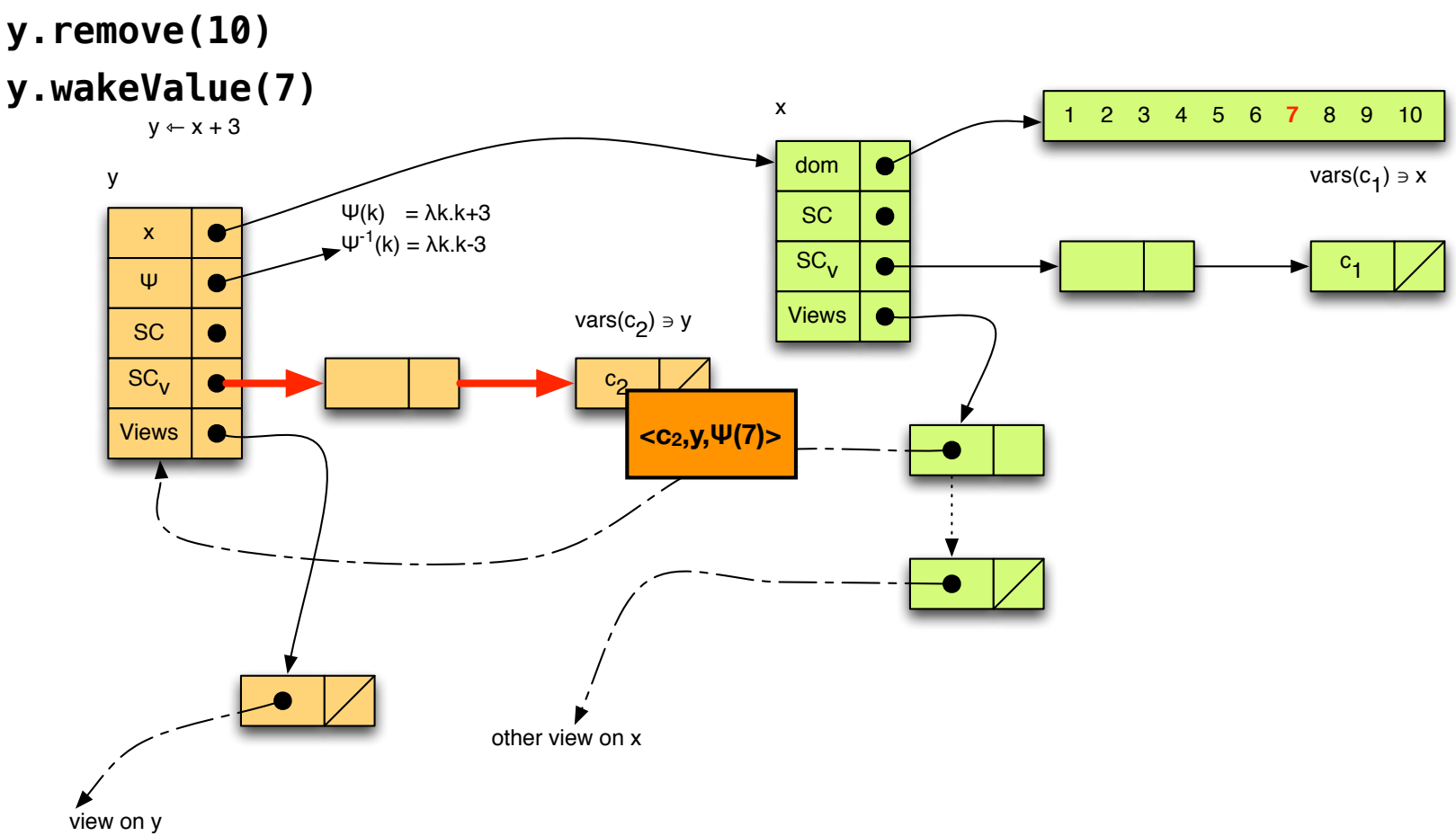
Illustration

```
y.remove(10)
y.wakeValue(7)
```



Q <c1,x,7>

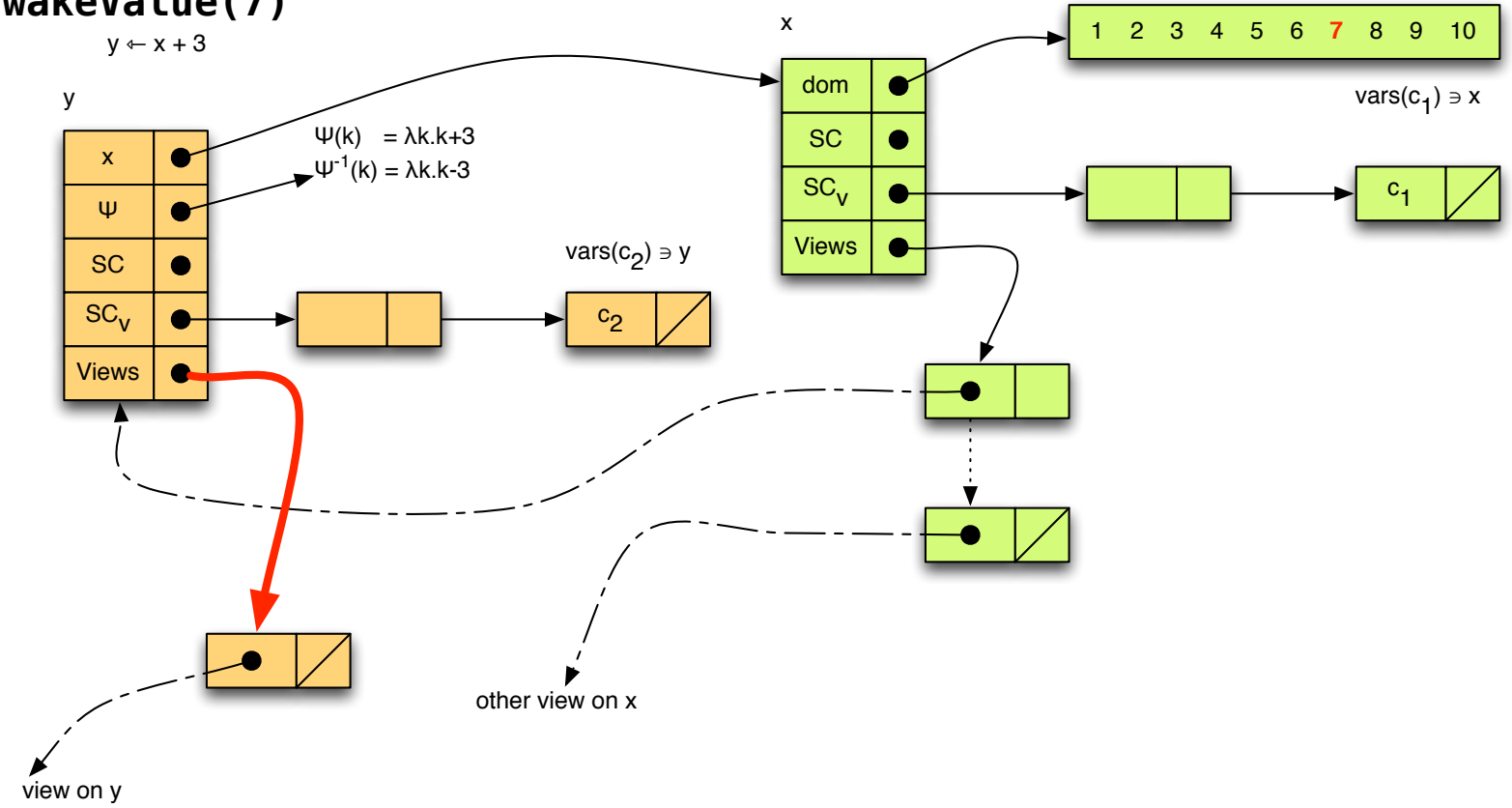
Illustration



Illustration

y.remove(10)

y.wakeValue(7)



z.wakeValue(10)

Q

Immediate Benefits

- Keeps constraints ***on*** variables ***with*** variables
- Modularizes Ψ inside the variable
  - Composition of Ψ functions via delegation
- Handles AC-5 events, one value at a time
  - No delta set
  - No approximation with intervals
  - **Exact representation of losses**
- Almost no space overhead
- Instrumentation points
  - remove/member/wake/wakeValue

Idempotence

- Important assumption used in many solvers

Beware!

It interferes with views

Example

- Consider the following linear constraint [$X_1, X_2, X_3 \in \{0,1\}$]

$$5 * x_1 + 3 * x_2 + 3 * x_3 \geq 7$$

- The Linear Inequality propagator will…
  - Deduce that to reach 7, x_1 **must** be 1
  - Once x_1 is 1, either x_2 or x_3 can be used to get to 7. No deductions
  - Idempotence tells us we can safely stop propagation after checking each variable once in this constraint.

However…

- Consider the presence of a view: $x_2 = 1 - x_1$
- What is the impact ?

$$5 * x_1 + 3 * x_2 + 3 * x_3 \geq 7$$

- The Linear Inequality propagator will…
  - Deduce that x1 must be 1 (as before of course)
  - The view *immediately deduces* $x_2 = 0$
  - But the linear equality does not know that and did not realize the new upper bound of x_2. So it stops as before….
  - And x_3 is never tightened to 1. [propagation is incorrect]

Lesson ?

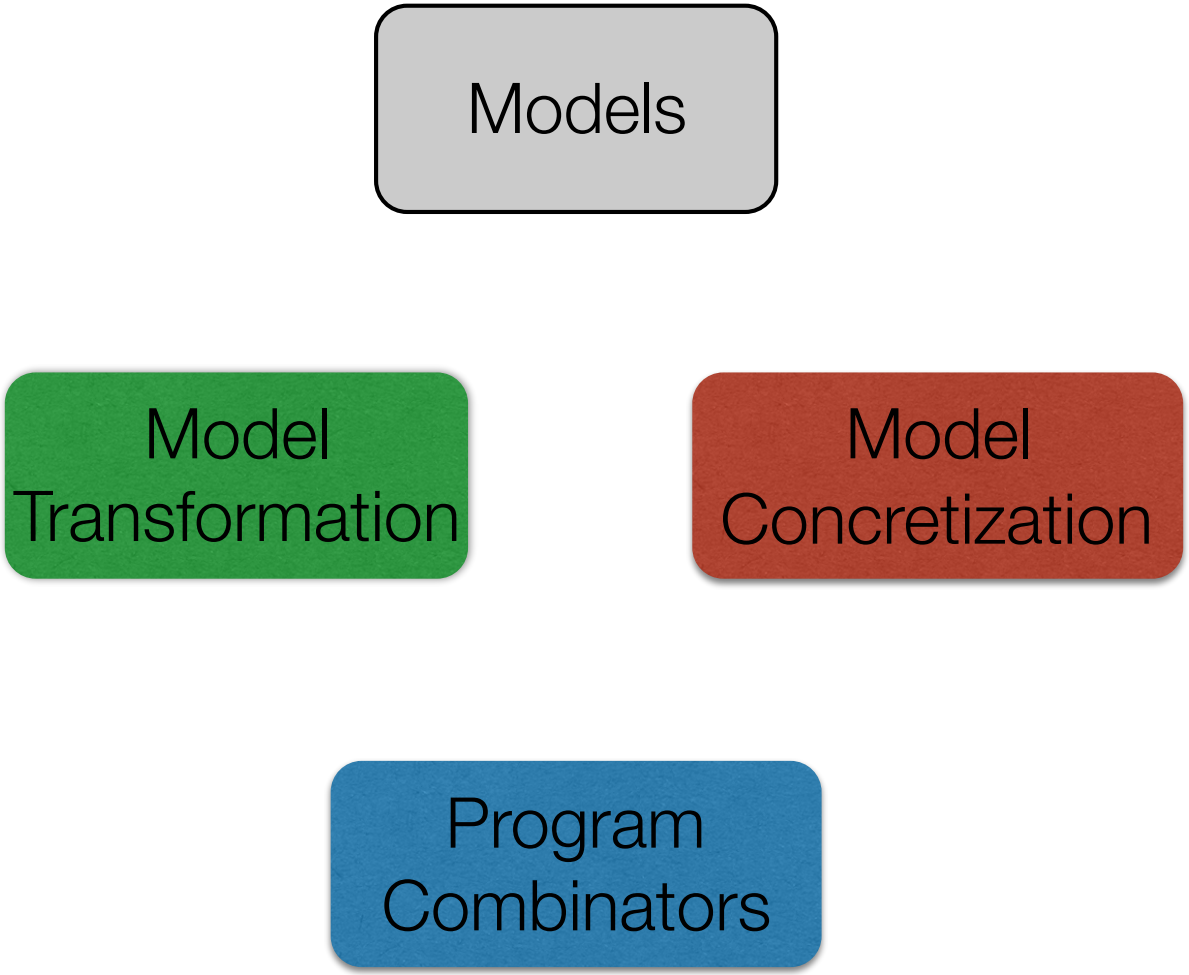
- While views are very useful….
- One can easily get bitten!
  - Views are side-channel communications
  - They invalidate idempotence assumptions
- To be correct…
  - It is necessary to analyze such channels to use suitable propagators!

Overview

- Motivation
- Modeling Layer
  - Variables / Constraints
  - Transformations, reformulation and concretizations
  - Consistency handling
- Solving Layer
  - Microkernel
  - Events / Propagators and Propagation
  - Views
- Hybrids
- Search building blocks
- Parallel support

Modeling

- Ingredients

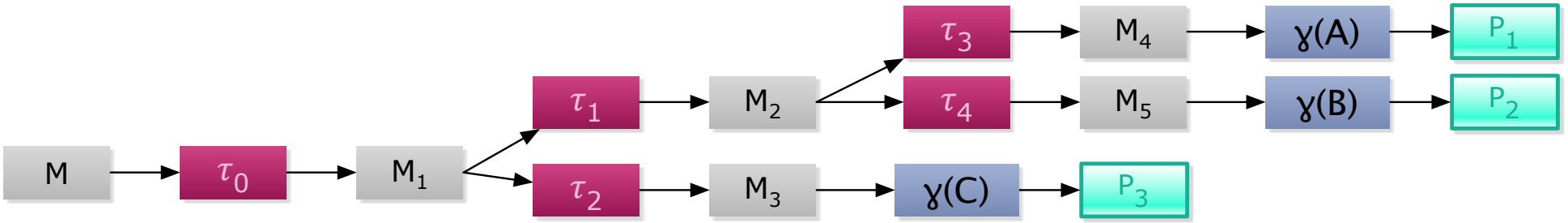


Available technologies

- Currently
  - CP
  - LP
  - IP
- Being developed
  - Routing, Scheduling [CPAIOR'16], MINLP, OptPower
  - Same with explanations
- (radically new) designed
  - CBLS

Overall

- Given 3 technologies A,B,C
  - Rewrite models from M
  - Apply the 3 concretizations
  - Obtain 3 ***Programs***

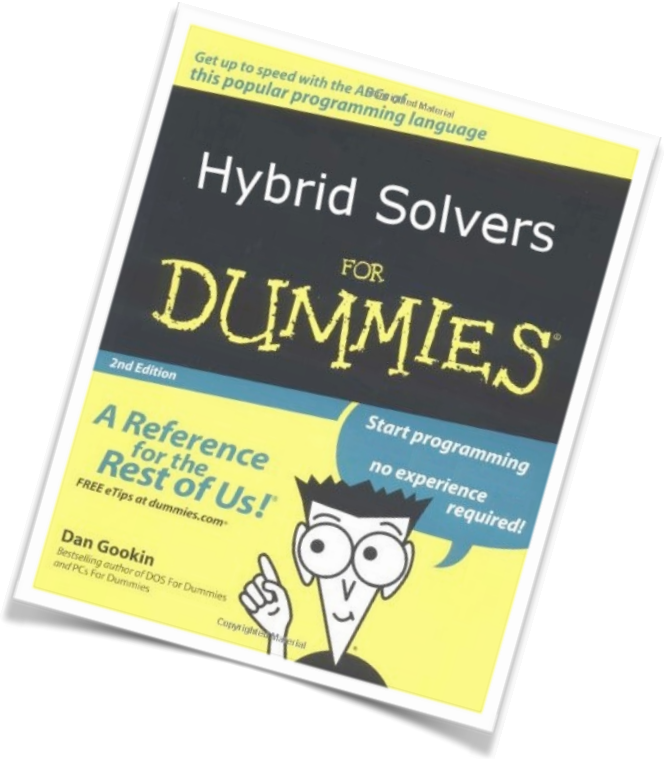


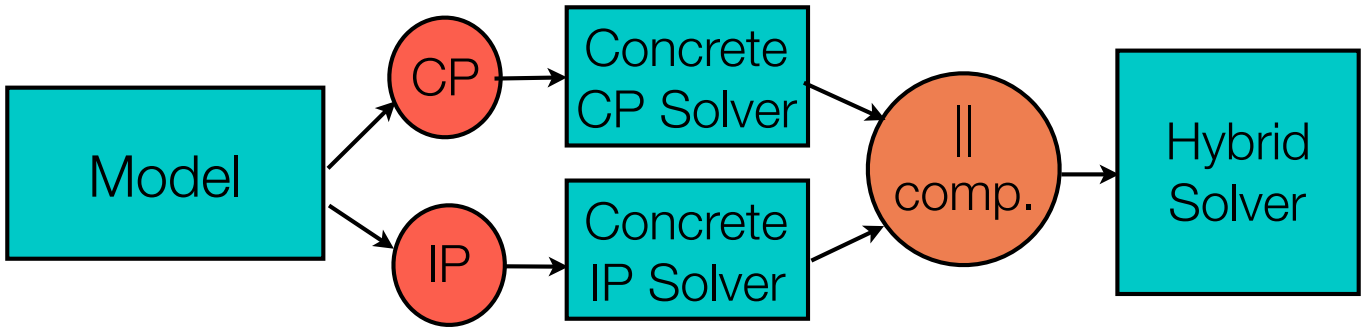
- Then
  - Compose the programs

[Combinators,CP'13]

Purpose?

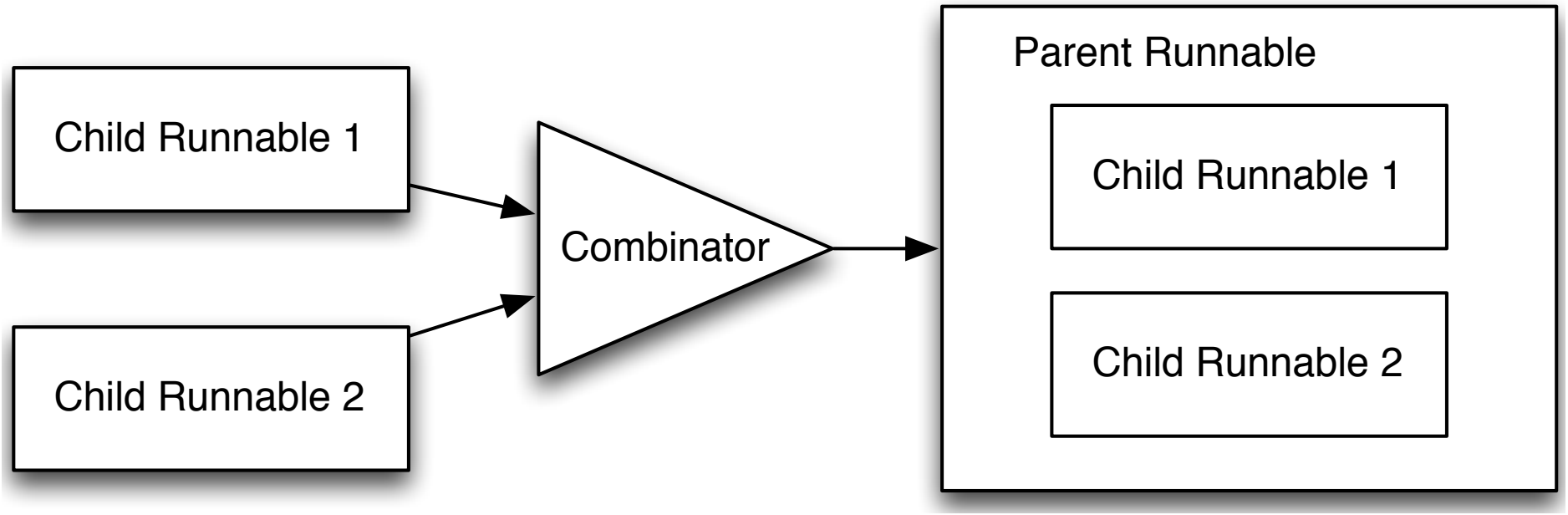
- Facilitate complex optimization *programs*
  - *Multi-technology (CP/IP/LS)* *[CP'13]*
    - *Sequential / Parallel*
    - *Column generation*
    - *Benders*
  - *LR* *[CP'14]*
  - *Scheduling* *[CPAIOR'16]*





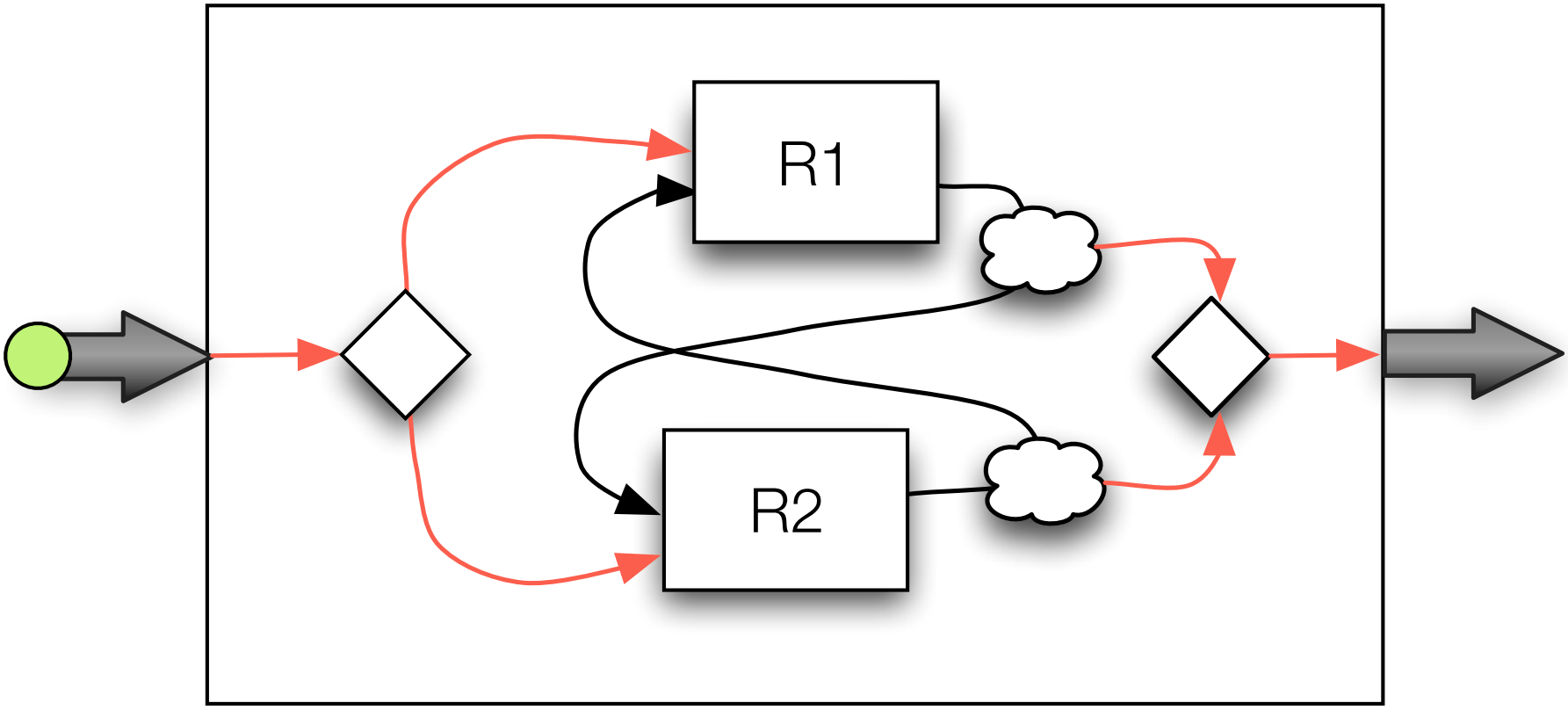
Combinators

- Take input runnables and…
  - generate output runnable, R = C(R1, R2)



Combinators Example (Parallel)

- Piping rules between R, R1, R2
  - Internal Piping
  - External Piping



Code Example (GLR)

```
1 id<ORModel>  P = [ORFactory createModel];
2 ...
3 id<ORIdArray>  H  = ... // array of hard constraints in P
4 id<ORModel>    L  = [ORFactory lagrangianRelax: P relaxingConstraints: H];
5 id<ORProgram>  O  = [ORFactory createMIPProgram: L];
6 id<ORRunnable> r  = [ORFactory subgradient: O];
7 [r run];
```

- Start with abstract **model**
- Create a relaxed model using LR and constraint set to soften
- Setup a **program (MIP)**
- Use a sub gradient scheme on top of that program

Applied to Scheduling ?

High-level Modeling

UCONN

```
1 id<ORModel> m = [ORFactory createModel];
2 // data setup ...
3 id<ORIntRange> J         = RANGE(m,0,nbJobs-1);
4 id<ORIntRange> M         = RANGE(m,0,nbMach-1);
5 id<ORIntMatrix> D        = [ORFactory intMatrix: m range: J : M];
6 id<ORIntMatrix> resource = [ORFactory intMatrix: m range: J : M];
7 // variables
8 id<ORTaskVarMatrix> task = [ORFactory tvMatrix:m range:J:M horizon:H duration:D];
9 id<ORIntVar> makespan    = [ORFactory intVar: m domain: RANGE(m,0,totalDur)];
10 id<ORTaskDisjunctiveArray> disjunctive = [ORFactory disjunctiveArray:m range: M];
11 // model
12 [m minimize: makespan];
13 for(ORInt i = J.low; i <= J.up; i++)
14    for(ORInt j = M.low; j < M.up; j++)
15       [m add: [[task at: i : j] precedes: [ task at: i : j+1]]];
16 for(ORInt i = J.low; i <= J.up; i++)
17    [m add: [[task at: i : Machines.up] isFinishedBy: makespan]];
18 for(ORInt i = J.low; i <= J.up; i++)
19    for(ORInt j = M.low; j <= M.up; j++)
20       [disjunctive[[resource at: i : j]] add: [task at: i : j]];
21 for(ORInt i=M.low; i <= M.up; i++)
22    [m add: disjunctive[i]];
```

High-level Modeling

UCONN

```
1  id<ORModel> m = [ORFactory createModel];
2  // data setup ...
3  id<ORIntRange> J          = RANGE(m,0,nbJobs-1);
4  id<ORIntRange> M          = RANGE(m,0,nbMach-1);
5  id<ORIntMatrix> D         = [ORFactory intMatrix: m range: J : M];
6  id<ORIntMatrix> resource = [ORFactory intMatrix: m range: J : M];
7  // variables
8  id<ORTaskVarMatrix> task = [ORFactory tvMatrix:m range:J:M horizon:H duration:D];
9  id<ORIntVar> makespan    = [ORFactory intVar: m domain: RANGE(m,0,totalDur)];
10 id<ORTaskDisjunctiveArray> disjunctive = [ORFactory disjunctiveArray:m range: M];
11 // model
12 [m minimize: makespan];
13 for(ORInt i = J.low; i <= J.up; i++)
14    for(ORInt j = M.low; j < M.up; j++)
15       [m add: [[task at: i : j] precedes: [ task at: i : j+1]]];
16 for(ORInt i = J.low; i <= J.up; i++)
17    [m add: [[task at: i : Machines.up] isFinishedBy: makespan]];
18 for(ORInt i = J.low; i <= J.up; i++)
19    for(ORInt j = M.low; j <= M.up; j++)
20       [disjunctive[[resource at: i : j]] add: [task at: i : j]];
21 for(ORInt i=M.low; i <= M.up; i++)
22    [m add: disjunctive[i]];
```

High-level model

linearize

τ_0

$\mathcal{M}_{\text{LINEAR}}$

γ

$\text{CP}_{\text{concretize}}$

γ

$\text{MIP}_{\text{concretize}}$

$$\texttt{min } makespan$$
$$\texttt{s.t.} \begin{cases} precedes(task_{i,j}, task_{i+1,j}) & \forall i \in M, \forall j \in J \\ finished\_by(task_{m,j}, makespan) & \forall j \in J \\ disjunctive(\{task_{\sigma^j_k,j} \mid j \in J, k \in 1 \dots m, \sigma^j_k = r\}) & \forall r \in M \end{cases}$$

$$\texttt{min } makespan$$
$$\texttt{s.t.} \begin{cases} x_{i,j} \geq 0 & \forall j \in J, \forall i \in M \\ x_{\sigma^j_h,j} \geq x_{\sigma^j_{h-1},j} + p_{\sigma^j_{h-1},j} & \forall j \in J, h \in 2, \dots, m \\ x_{i,j} \geq x_{i,k} + p_{i,k} - z_{i,j,k} * V & \forall j,k \in J, k < j, i \in M \\ x_{i,k} \geq x_{i,j} + p_{i,j} - (1 - z_{i,j,k}) * V & \forall j,k \in J, k < j, i \in M \\ makespan \geq x_{\sigma^j_m,j} + p_{\sigma^j_m,j} & \forall j \in J \\ z_{i,j,k} \in \{0,1\} & \forall i \in M, \forall j \in J, \forall k \in J \end{cases}$$

High-level Modeling

UCONN

global constraint formulation:

$$\texttt{min } makespan$$
$$\texttt{s.t.} \begin{cases} precedes(task_{i,j}, task_{i+1,j}) & \forall i \in M, \forall j \in J \\ finished\_by(task_{m,j}, makespan) & \forall j \in J \\ disjunctive(\{task_{\sigma_k^j,j} \mid j \in J, k \in 1 \dots m, \sigma_k^j = r\}) & \forall r \in M \end{cases}$$

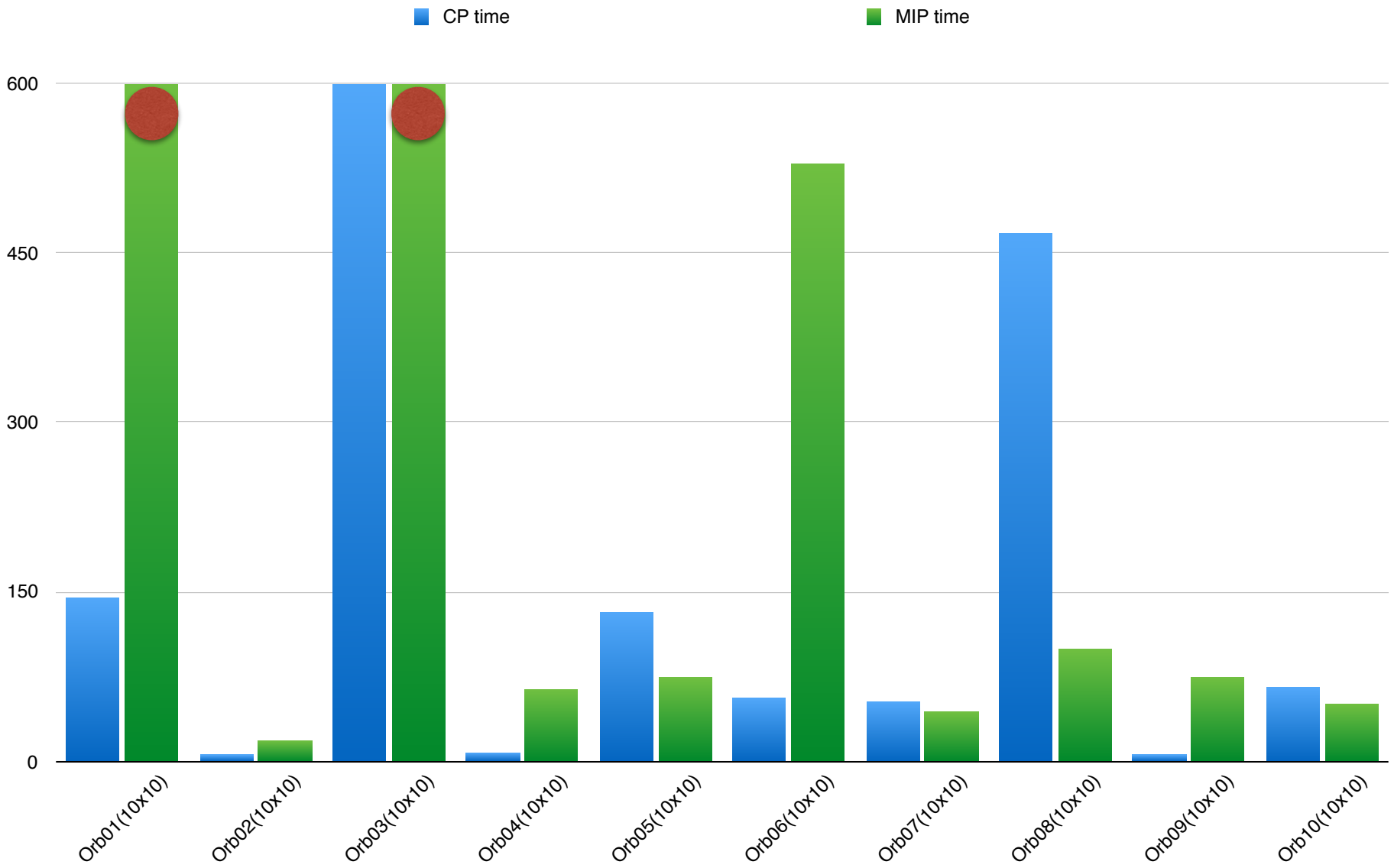
disjunctive linear formulation:

$$\texttt{min } makespan$$
$$\texttt{s.t.} \begin{cases} x_{i,j} \geq 0 & \forall j \in J, \forall i \in M \\ x_{\sigma_h^j,j} \geq x_{\sigma_{h-1}^j,j} + p_{\sigma_{h-1}^j,j} & \forall j \in J, h \in 2, \dots, m \\ x_{i,j} \geq x_{i,k} + p_{i,k} - z_{i,j,k} * V & \forall j, k \in J, k < j, i \in M \\ x_{i,k} \geq x_{i,j} + p_{i,j} - (1 - z_{i,j,k}) * V & \forall j, k \in J, k < j, i \in M \\ makespan \geq x_{\sigma_m^j,j} + p_{\sigma_m^j,j} & \forall j \in J \\ z_{i,j,k} \in \{0, 1\} & \forall i \in M, \forall j \in J, \forall k \in J \end{cases}$$

Creating the Parallel Runnable

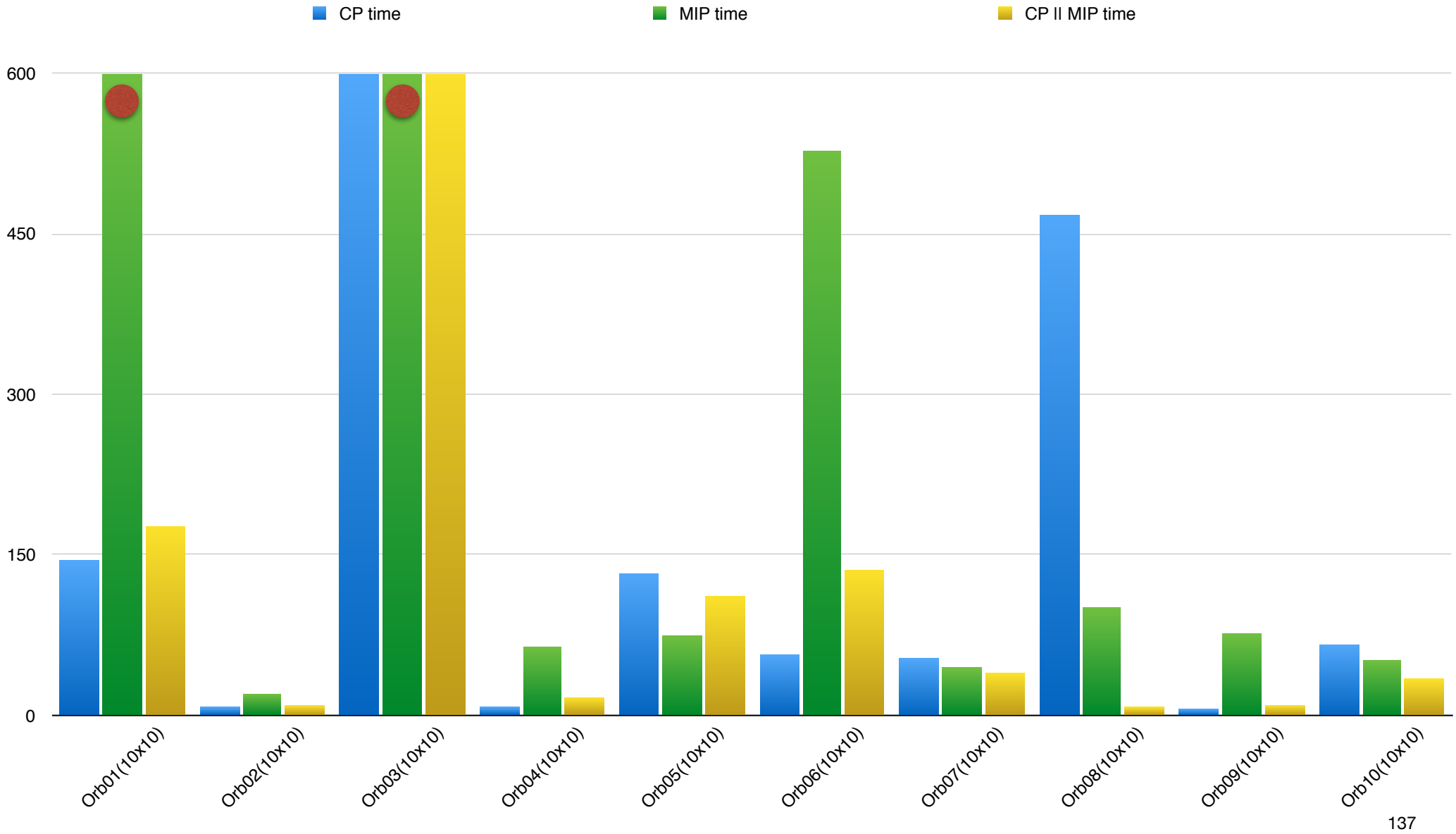

```
1 id<ORModel> m = ... // Def. of Jobshop Model
2 id<ORModel> LinearModel = [ORFactory linearize:m encoding:Disjunctive];
3 id<ORRunnable> r0 = [ORFactory CPRunnable: m solve: search ];
4 id<ORRunnable> r1 = [ORFactory MIPRunnable: LinearModel];
5 id<ORRunnable> parallel = [ORCombinator completeParallel: r0 with: r1];
6 [parallel run];
```

Pure Solvers

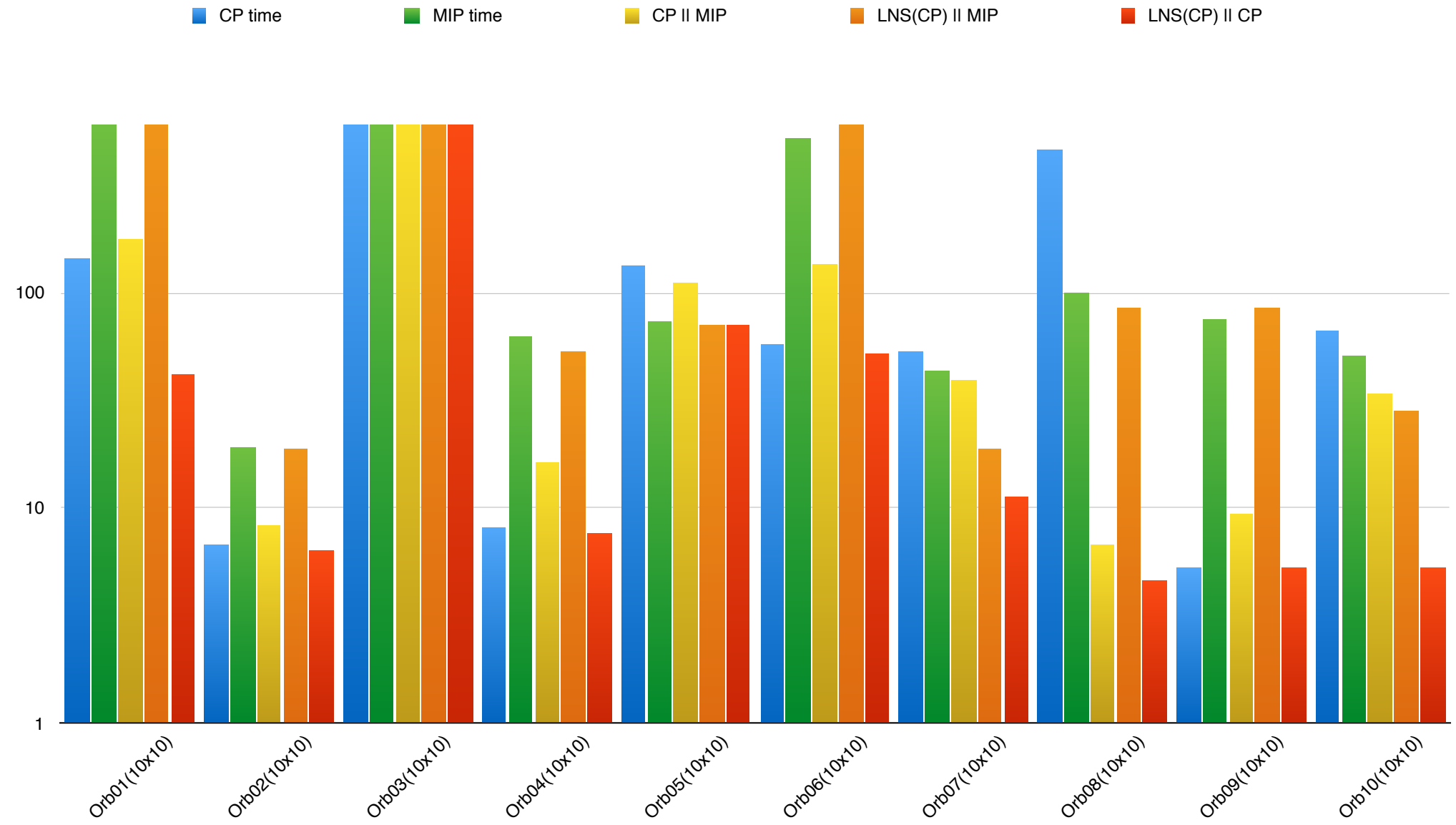


Adding Plain Parallel

UCONN



Adding Parallel with LNS



Results

UCONN

Instances	CP		MIP		$CP \parallel MIP$		$LNS_{CP} \parallel MIP$		$LNS_{CP} \parallel CP$	
	time	ub	time	ub	time	ub	time	ub	time	ub
Orb01(10 × 10)	145.38	1059*	600.0	1072	176.12	1059*	600.0	1071	**41.96**	**1059***
Orb02(10 × 10)	6.80	888*	19.06	888*	8.36	888*	18.97	888*	**6.33**	**888***
Orb03(10 × 10)	**600.0**	**1015**	600.0	1021	**600.0**	**1015**	**600.0**	**1005**	600.0	1015
Orb04(10 × 10)	8.17	1005*	63.07	1005*	16.33	1005*	53.33	1005*	**7.67**	**1005***
Orb05(10 × 10)	132.46	887*	74.20	887*	110.82	887*	70.92	887*	**70.35**	**887***
Orb06(10 × 10)	**57.37**	**1010***	528.22	1010*	135.53	1010*	600.0	1010**	52.05	1010
Orb07(10 × 10)	53.22	397*	43.64	397*	39.15	397*	18.65	397*	**11.23**	**397***
Orb08(10 × 10)	467.19	899*	99.86	899*	6.82	899*	84.41	899*	**4.57**	**899***
Orb09(10 × 10)	**5.31**	**934***	75.36	934*	9.41	934*	85.55	934*	**5.31**	**934***
Orb10(10 × 10)	66.24	944*	51.20	944*	33.87	944*	28.34	944*	**5.31**	**944***
la31(30 × 10)	600.0	2801	600.0	2003	600.0	2109	30.82	1784*	**17.23**	**1784***
la36(15 × 15)	600.0	2059	600.0	1292	600.0	1297	600.0	1281	**136.96**	**1268***
la37(15 × 15)	600.0	1855	600.0	1454	600.0	1478	**13.62**	**1397***	13.97	1397*
la38(15× 15)	600.0	1633	600.0	1230	600.0	1243	**600.0**	**1196**	600.0	1255
la21(15× 10)	600.0	1129	600.0	1079	600.0	1097	600.0	1058	**600.0**	**1046**

Overview

- Motivation
- Modeling Layer
  - Variables / Constraints
  - Transformations, reformulation and concretizations
  - Consistency handling
- Solving Layer
  - Microkernel
  - Events / Propagators and Propagation
  - Views
- Hybrids
- Search building blocks
- Parallel support

Search building blocks

- How to build a search facility ?

- Challenges
  - Search is non-deterministic
  - CP shines brightest when *users* exploit semantics through search
  - Black-box (closed) search is easy to support
  - Glass-box (open) search is a lot harder
  - Parallel support should be orthogonal

Dealing with non-determinism

- One possible answer

Continuations

[Constraints,06 ; CP'06]

- Rationale ?
  - Preserves control through the host language
  - No nested "search interpreter"
  - No compromise: fully retains the ability to write custom search.
- Challenge
  - Deal with subtleties of memory models.

Continuations in a Nutshell

- What is it ?
  - An abstraction to ***capture*** and ***restore*** control flow
- Support ?
  - Some languages have it natively (Scheme/ML)
  - Can be implemented in C/C++/Objective-C/Swift/….

Capture Control Flow

- What it captures
  - Processor state (all registers)
  - Instruction pointer (where we are)
  - Runtime system stack
- What is left alone
  - Any heap allocated structure
  - Any static structure

Restore Control Flow

- What is restored
  - Processor state
  - Instruction pointer
  - System stack
- Effects
  - It abandons the current execution control flow
  - It resumes a saved execution control flow
  - The resumed control flow ***knows*** it is a restoration.

Benefits

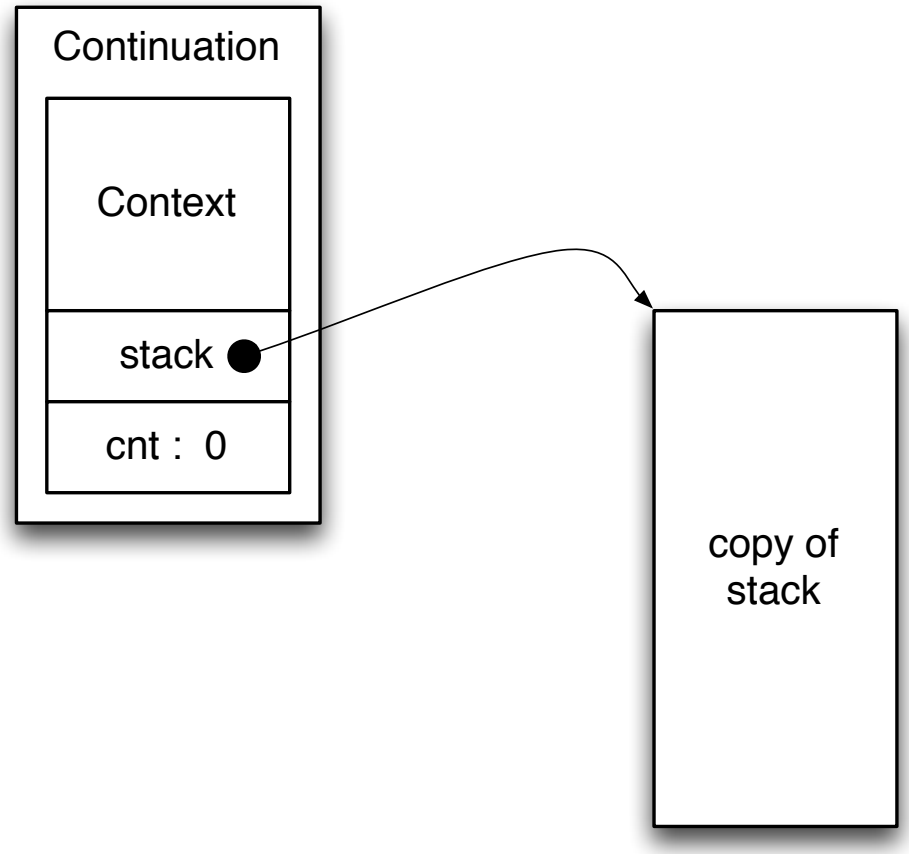
- There are no interpreters
  - All code executes natively at full speed
  - All the facilities of the host language still work
    - Breakpointing
    - State inspection
    - …
  - All the native control flow instruction work unmodified.

Cost

- Expressed in two ways
  - Time to save / restore the system stack
  - Space used to preserve the system stack
- How significant is this ?
  - Time-wise
    - Continuation code is faster than hard-coded DFS!
    - Memory copying is *fast* on Intel architectures
  - Space-wise
    - Relatively small (typically 1Kb per continuation)
    - Can be made incremental (space-time tradeoff)

Pictorially

- Processor context
  - A simple structure with all the registers
- Stack
  - A pointer to a memory block with the copy of the state
- Count
  - How many times it has been called so far [0 ..



Doing it in C++

- Processor Context (x86)

```
class Cont;
struct Ctx64 {
   long rax,rbx,rcx,rdx,rdi,rsi,rbp,rsp;
   long r8,r9,r10,r11,r12,r13,r14,r15,rip;
   unsigned int   pad; // alignment padding.
   unsigned int   mxcsr;
   double xmm0[2],xmm1[2],xmm2[2],xmm3[2],xmm4[2],xmm5[2];
   double xmm6[2],xmm7[2],xmm8[2],xmm9[2],xmm10[2],xmm11[2];
   double xmm12[2],xmm13[2],xmm14[2],xmm15[2];
   char   fpu[108];
};
__attribute__((noinline)) Cont* saveCtx(struct Ctx64* ctx,Cont* k);
__attribute__((noinline)) Cont* restoreCtx(struct Ctx64* ctx,
                                           char* start,
                                           char* data,
                                           size_t length);
```

Context

```
static __thread char* baseStack = 0;
__attribute__((noinline)) Cont* saveCtx(struct Ctx64* ctx,Cont* k) {
    Cont* var = 0;
    char* sp;
    asm volatile("movq %%rsp , %%rax;" // load rax with SP
                 :"=a"(sp));           // write rax into output var sp
    size_t len = baseStack - sp;       // length of stack suffix
    k->saveStack(len,sp);              // save it (memory copy)
    asm volatile("movq %%rbx,8(%%rax);\n\t"
                 "movq %%rcx,16(%%rax);\n\t"
                 …
                 "jmp   resume;\n\t"
                 "goon: popq %%rbx;\n\t"
                 "      movq %%rbx, 128(%%rax);\n\t"
                 "      xor %%rax,%%rax;\n\t"
                 "      jmp end;\n\t"
                 "resume: call goon;\n\t"
                    "end: nop;\n\t"
                    :"=a"(var) :"a"(ctx));
       return var;
}
```

Context

```
__attribute__((noinline)) Cont* restoreCtx(struct Ctx64* ctx,
                                           char* start,char* data,
                                           size_t length) {
   Cont* rv = 0;
   // ctx in rdi, start in rsi, data in rdx, length in ecx
   asm volatile("copystack: cmp $0x0,%%ecx    ; \n\t"
                …
                "             jmp copystack    ; \n\t" //go to top
                "donecopy:  mov %%rdi,%%rax    ; \n\t" // context in rax
                "movq 8(%%rax),%%rbx           ; \n\t" // restore state
                "movq 16(%%rax),%%rcx;\n\t"
                …
                "frstor 400(%%rax)             ;\n\t"  // restore FP
                "movq 128(%%rax),%%rdi         ;\n\t"  // rdi <- return
                "movq (%%rax),%%rax            ;\n\t"  // restore rax
                "jmp *%%rdi                    ;\n\t"  // jump to end!!!
                 :"=a"(rv) :"D"(ctx));
  return rv;
}
```

Now, it's easy!

- Continuation is a class

```
class Cont {
   struct Ctx64   _target __attribute__ ((aligned(16)));
   size_t _length;
   char* _start, *_data;
   int _used,_cnt;
public:
   Cont();
   ~Cont();
   void saveStack(size_t len,void* s);
   void call();
   int nbCalls() const { return _used;}
};

Cont* takeContinuation();
void initContinuationLibrary(int *base);
void shutdown();
```

Key methods

```
void Cont::call() {
   struct Ctx64* ctx = &_target;
   ctx->rax = (long)this;
   restoreCtx(ctx,_start,_data,_length);
}

Cont* Cont::takeContinuation() {
   Cont* k = new Cont;
   struct Ctx64* ctx = &k->_target;
   Cont* resume = saveCtx(ctx,k);
   if (resume != 0) {
      resume->_used++;
      return resume;
   } else return k;
}
```

A simple non-deterministic binary choice

- Use closures to capture the branches

```
template <class Body1,class Body2>
void CPSolver::tryBin(Body1 left,Body2 right) {
   Cont* k = takeContinuation();
   if (k->nbCalls()==0) {
      _nbc++;
      _stack.push(k);
      left();
   } else {
      delete k;
      _nbc++;
      right();
   }
}
```

There is a minor *simplification*
Should use *Controllers* to be general
See next talk with PVH.

Usage Demo

- Uses standard C/C++ loops, STL and any API we wish
- Uses closures for the two branches (x = c and x ≠ c)
- Does non-determinism

```
cp->solveAll([] {
      for(int i=0;i < n;i++) {
         withVarDo(q,min_dom(q),[cp](auto x) {
               while(!x->isBound()) {
                  int c = x->getMin();
                  cp->tryBin([=] { cp->add(x == c);},
                             [=] { cp->add(x != c);});
               }
            });
      }
      cp->incrNbSol();
      *nbSol += 1;
   });
```

Revisiting Memory

- If you use ARC
  - Objective-CP / Swift / C++
  - shared_ptr<T> , unique_ptr<T>
- Be mindful
  - You can exit the same closure multiple times (backtracking!)
- Therefore…
  - Destructor of the smart pointer called several times!
  - You can’t hold smart pointers on the stack / in closure
  - You should have only *reference to those* or not use them
  - Alternatively, use vanilla *pointer handles*

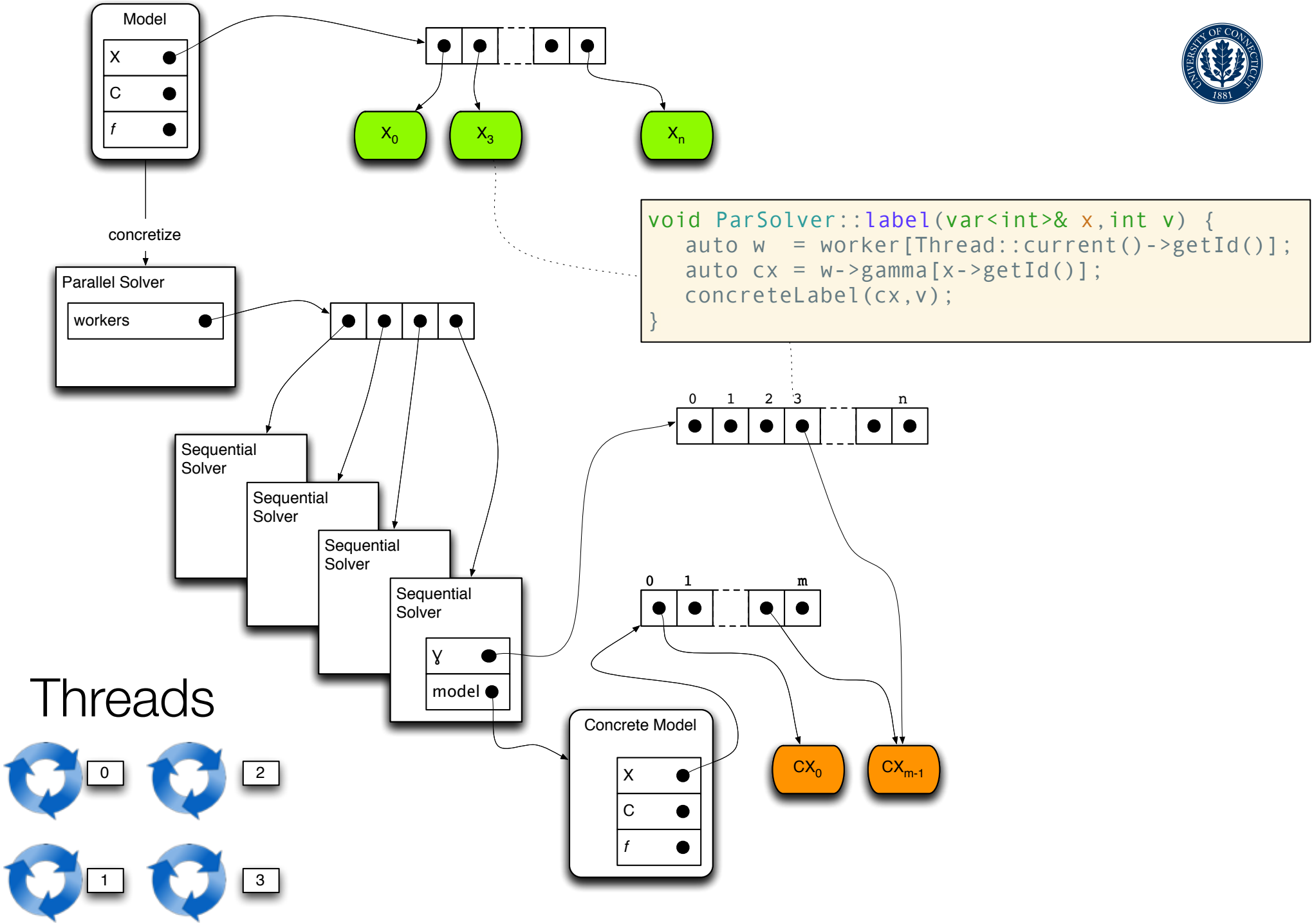
Overview

- Motivation
- Modeling Layer
  - Variables / Constraints
  - Transformations, reformulation and concretizations
  - Consistency handling
- Solving Layer
  - Microkernel
  - Events / Propagators and Propagation
  - Views
- Hybrids
- Search building blocks
- Parallel support

Parallel Support

- Very little needs to be done thanks to Gamma's existence!
  - Look at the picture
- Key ingredients
  - Gamma
  - Virtual binding
  - Delegation API

[DAMP'10 ; Computers & OR,'09 ; JOC'09]


Model
X
C
f
concretize
Parallel Solver
workers
X_0
X_3
X_n
void ParSolver::label(var<int>& x,int v) {
auto w = worker[Thread::current()->getId()];
auto cx = w->gamma[x->getId()];
concreteLabel(cx,v);
}
0 1 2 3 n
Sequential Solver
Sequential Solver
Sequential Solver
Sequential Solver
Y
model
0 1 m
Concrete Model
X
C
f
CX_0
CX_{m-1}
Threads
0
2
1
3


THE END