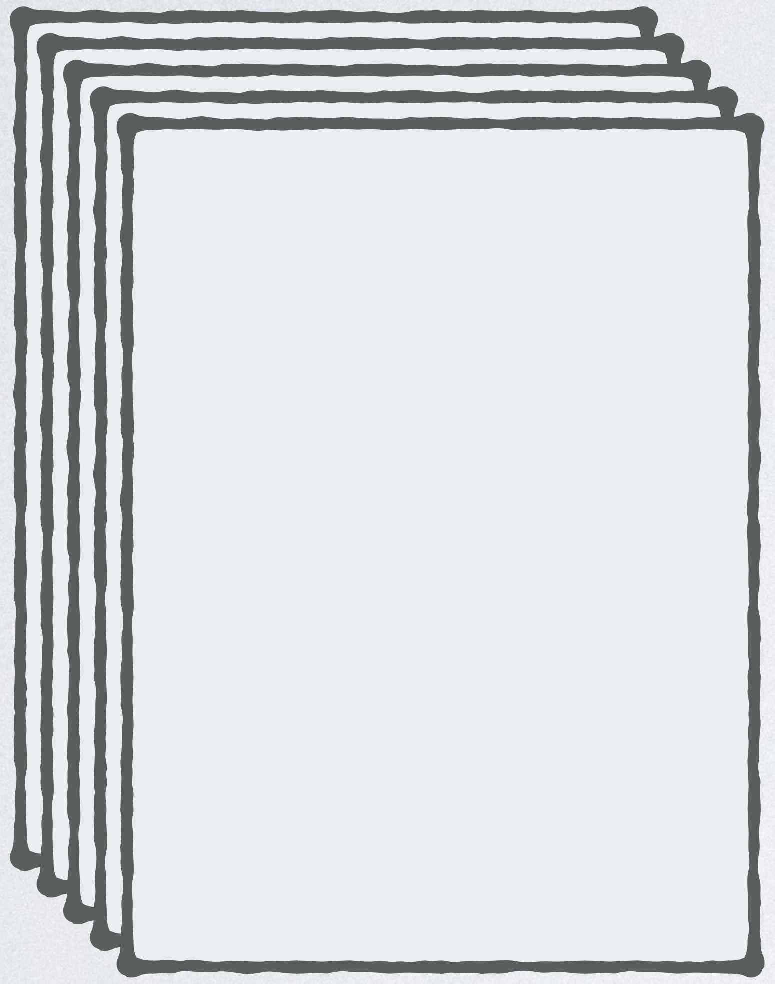
Liquid Resource Types

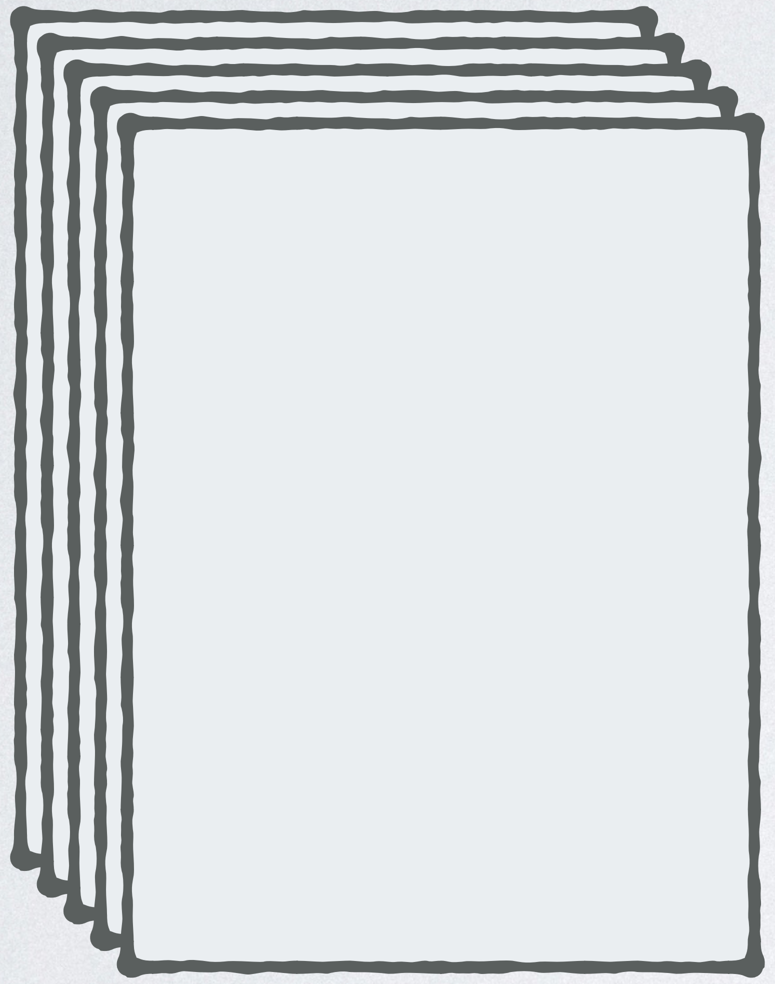
Tristan Knoth[1], **Di Wang**[2], Adam Reynolds[1], Jan Hoffmann[2], Nadia Polikarpova[1]
[1] University of California, San Diego
[2] Carnegie Mellon University

RESOURCE ANALYSIS



Programs

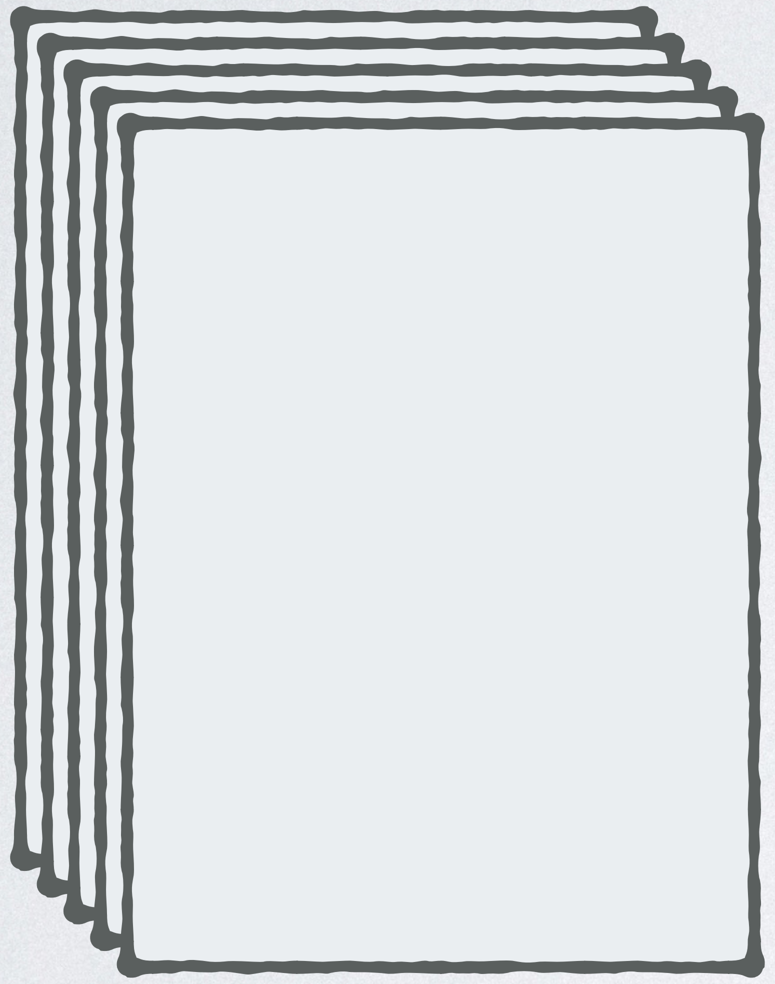
Resource Analysis



Programs

Performance

RESOURCE ANALYSIS

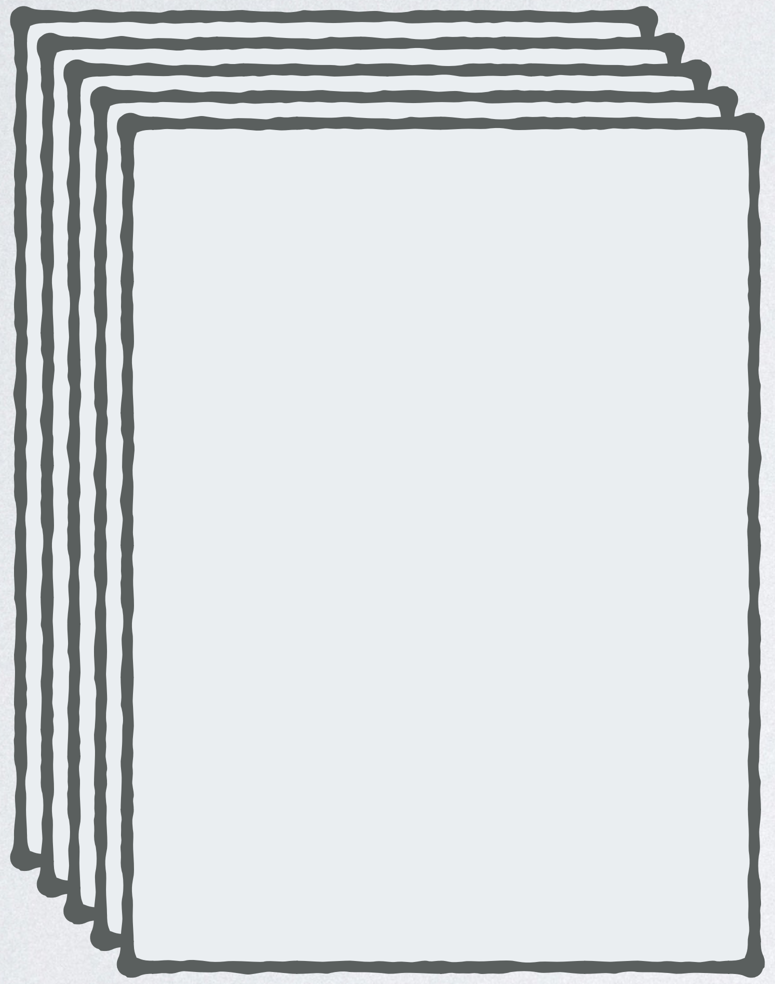


- Time
- Memory
- Power
- …

Programs

Performance

RESOURCE ANALYSIS



Programs

- Time
- Memory
- Power
- …

Performance

Scenarios

- Algorithmic complexity
- Gas usage in blockchains
- Side-channel detection

Example: Insertion Sort

Quick Sort

Insertion Sort

Example: Insertion Sort

Quick Sort	Insertion Sort

- These two are functionally equivalent. Which one performs better?

EXAMPLE: INSERTION SORT

Quick Sort

Insertion Sort

- These two are functionally equivalent. Which one performs better?
- Both run in quadratic time in the worst case.

EXAMPLE: INSERTION SORT

Quick Sort	Insertion Sort

- These two are functionally equivalent. Which one performs better?
- Both run in quadratic time in the worst case.
- But insertion sort can be linear-time on nearly-sorted data.

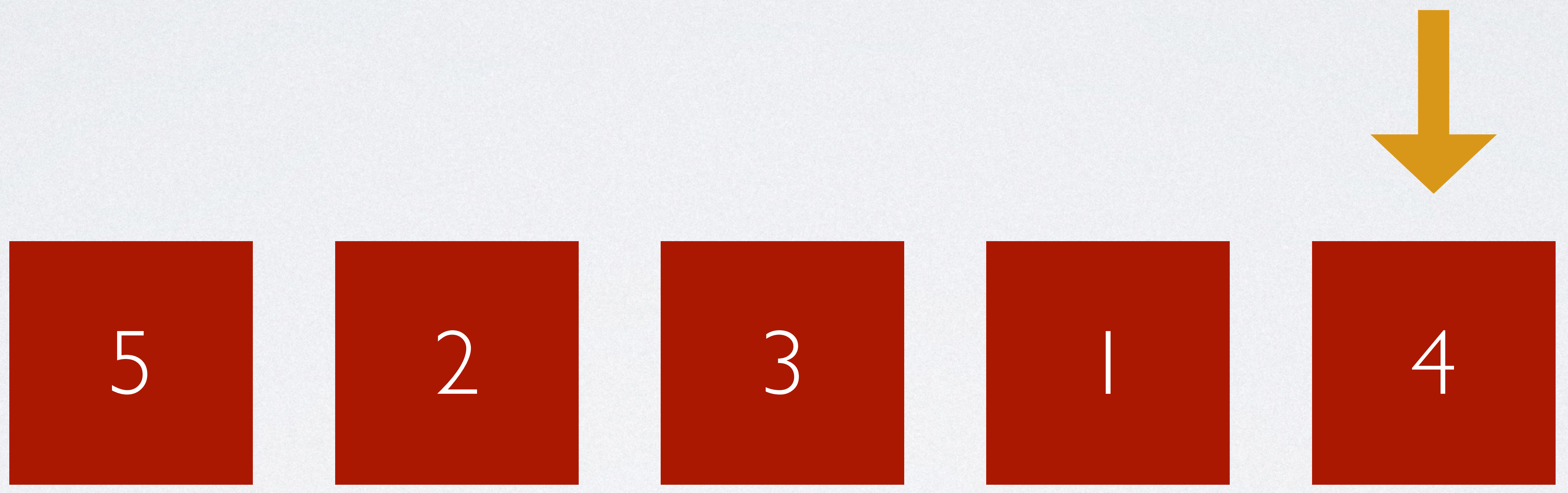
EXAMPLE: INSERTION SORT

Insertion Sort

5 2 3 1 4

Example: Insertion Sort

Insertion Sort



Example: Insertion Sort

Insertion Sort

How many **swaps** does the algorithm need?

1	2	3	4	5

Example: Insertion Sort

Insertion Sort

How many **swaps** does the algorithm need?

1	2	3	4	5

- #swaps is proportional to #out-of-order-pairs in the input.

Example: Insertion Sort

Insertion Sort

How many **swaps** does the algorithm need?

1	2	3	4	5

- #swaps is proportional to #out-of-order-pairs in the input.
- **Challenge:** Express and automatically verify such a complex bound.

Liquid Types and Resources

[1] P. M. Rondon, M. Kawaguchi, and R. Jhala. 2008. Liquid Types. In *PLDI'08*.
[2] T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. 2019. Resource-Guided Program Synthesis. In *PLDI'19*.

LIQUID TYPES AND RESOURCES

Liquid Types[1]

"A function returns the absolute value of the input integer"

[1] P. M. Rondon, M. Kawaguchi, and R. Jhala. 2008. Liquid Types. In *PLDI'08*.
[2] T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. 2019. Resource-Guided Program Synthesis. In *PLDI'19*.

LIQUID TYPES AND RESOURCES

Liquid Types[1]	"A function returns the absolute value of the input integer"
RESYN[2]	"A list where each element carries one unit of potential"

[1] P. M. Rondon, M. Kawaguchi, and R. Jhala. 2008. Liquid Types. In *PLDI'08*.
[2] T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. 2019. Resource-Guided Program Synthesis. In *PLDI'19*.

LIQUID TYPES AND RESOURCES

Liquid Types[1] — "A function returns the absolute value of the input integer"

only linear bounds

RESYN[2] — "A list where each element carries one unit of potential"

[1] P. M. Rondon, M. Kawaguchi, and R. Jhala. 2008. Liquid Types. In *PLDI'08*.
[2] T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. 2019. Resource-Guided Program Synthesis. In *PLDI'19*.

LIQUID TYPES AND RESOURCES

Liquid Types[1] "A function returns the absolute value of the input integer"

only linear bounds

RESYN[2] "A list where each element carries one unit of potential"

Liquid Resource Types (This Work)

[1] P. M. Rondon, M. Kawaguchi, and R. Jhala. 2008. Liquid Types. In *PLDI'08*.
[2] T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. 2019. Resource-Guided Program Synthesis. In *PLDI'19*.

LIQUID TYPES AND RESOURCES

Liquid Types[1] "A function returns the absolute value of the input integer"

only linear bounds

RESYN[2] "A list where each element carries one unit of potential"

Liquid Resource Types (This Work) "Potentials can be inductively specified from the formation of data structures themselves"

[1] P. M. Rondon, M. Kawaguchi, and R. Jhala. 2008. Liquid Types. In *PLDI'08*.
[2] T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. 2019. Resource-Guided Program Synthesis. In *PLDI'19*.

Liquid Types and Resources

Liquid Types[1] "A function returns the absolute value of the input integer"

only linear bounds

ReSyn[2] "A list where each element carries one unit of potential"

Liquid Resource Types (This Work) "Potentials can be **inductively** specified from the formation of data structures themselves"

can be used to express the bound of insertion sort

[1] P. M. Rondon, M. Kawaguchi, and R. Jhala. 2008. Liquid Types. In *PLDI'08*.
[2] T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. 2019. Resource-Guided Program Synthesis. In *PLDI'19*.

Contributions

CONTRIBUTIONS

- Liquid resource **types** for verifying super-linear value-dependent bounds

Contributions

- Liquid **resource** **types** for verifying super-linear value-dependent bounds
- Proof of type **soundness** w.r.t. a cost semantics

Contributions

- Liquid resource **types** for verifying super-linear value-dependent bounds
- Proof of type **soundness** w.r.t. a cost semantics
- Prototype **implementation** and **evaluation**

Outline

- [x] Motivation
- [ ] Background: Liquid Types and the Potential Method
- [ ] Liquid Resource Types
- [ ] Evaluation

Liquid Types

Liquid Types

$\{ B \mid \psi \}$

A value of type B that satisfies ψ

LIQUID TYPES

`{ B	ψ }`	A value of type B that satisfies ψ
`{ Int	v ≥ 0 }`	A non-negative integer

Liquid Types

`{ B | ψ }` A value of type B that satisfies ψ

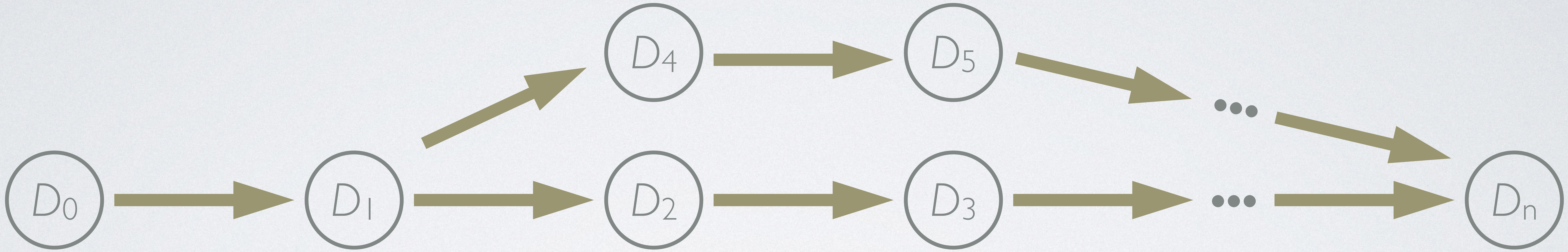
`{ Int | v ≥ 0 }` A non-negative integer

```
(xs : List a) -> { List a | len(v) = len(xs) + 1 }
```

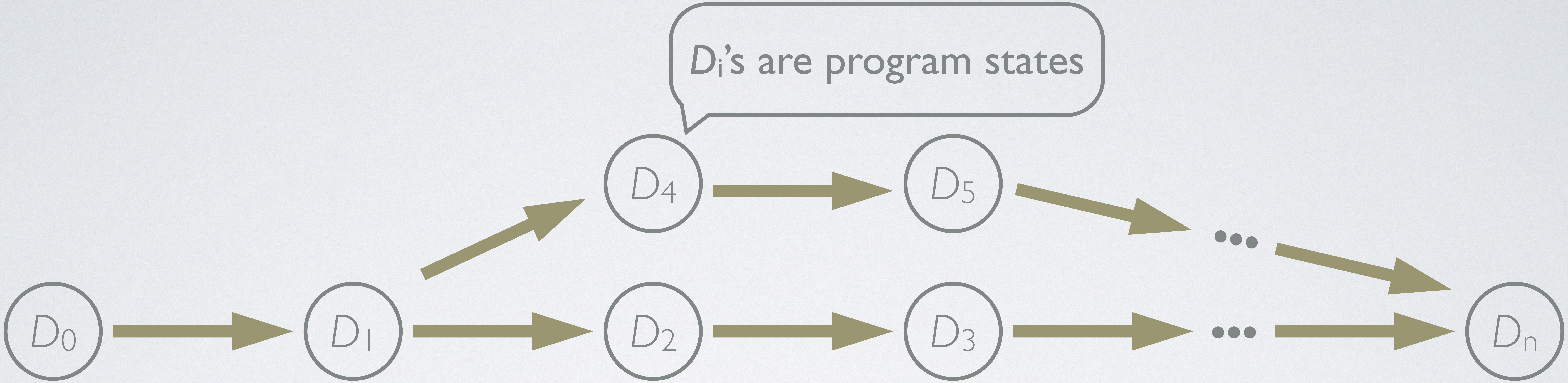
A function that returns a list whose length is one plus the length of its input

The Potential Method

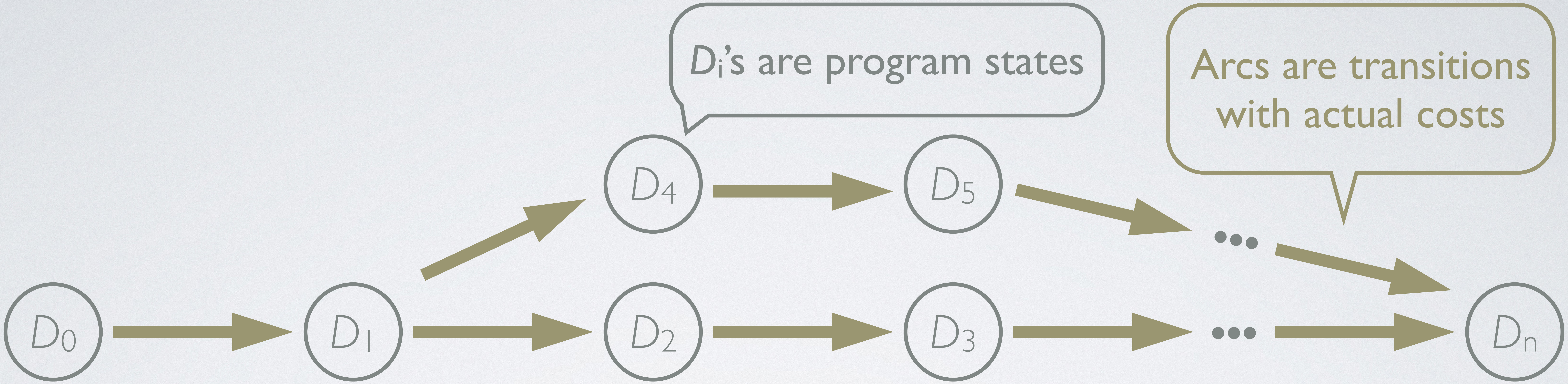
THE POTENTIAL METHOD



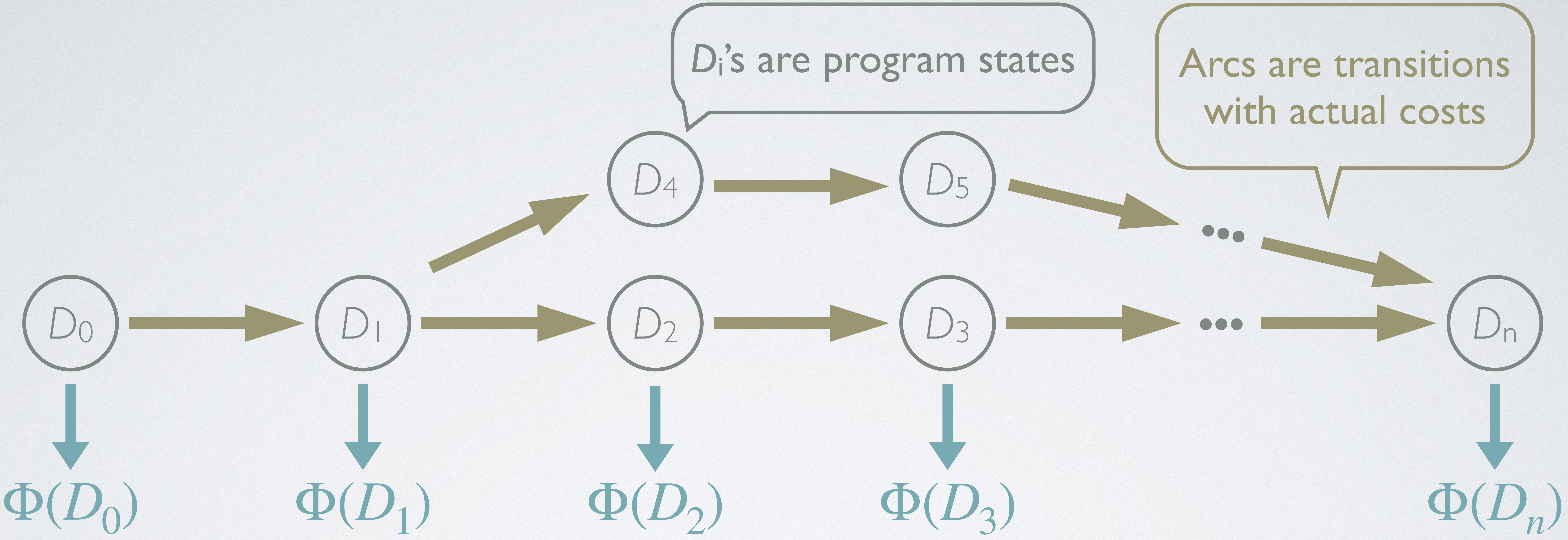
THE POTENTIAL METHOD



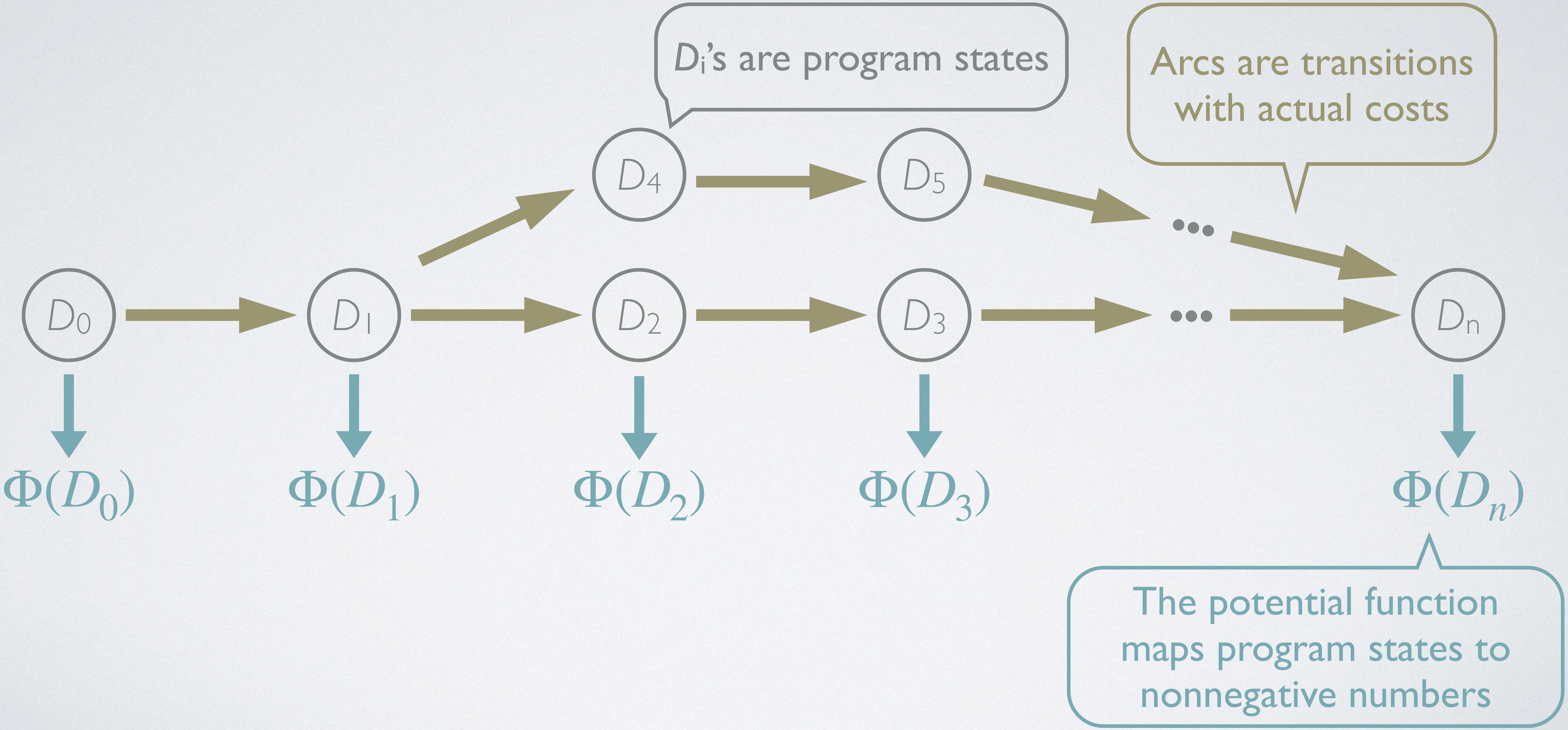
The Potential Method



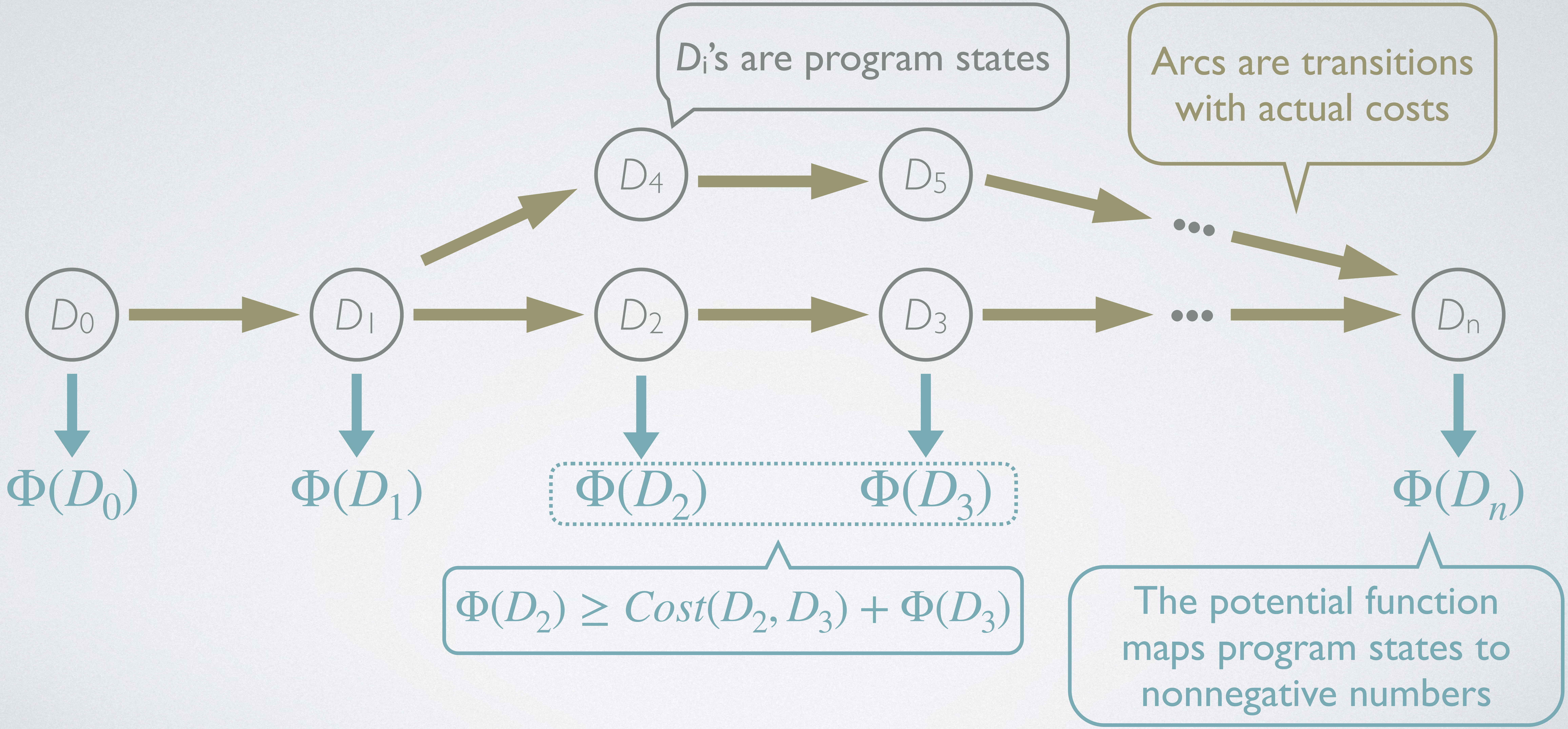
The Potential Method



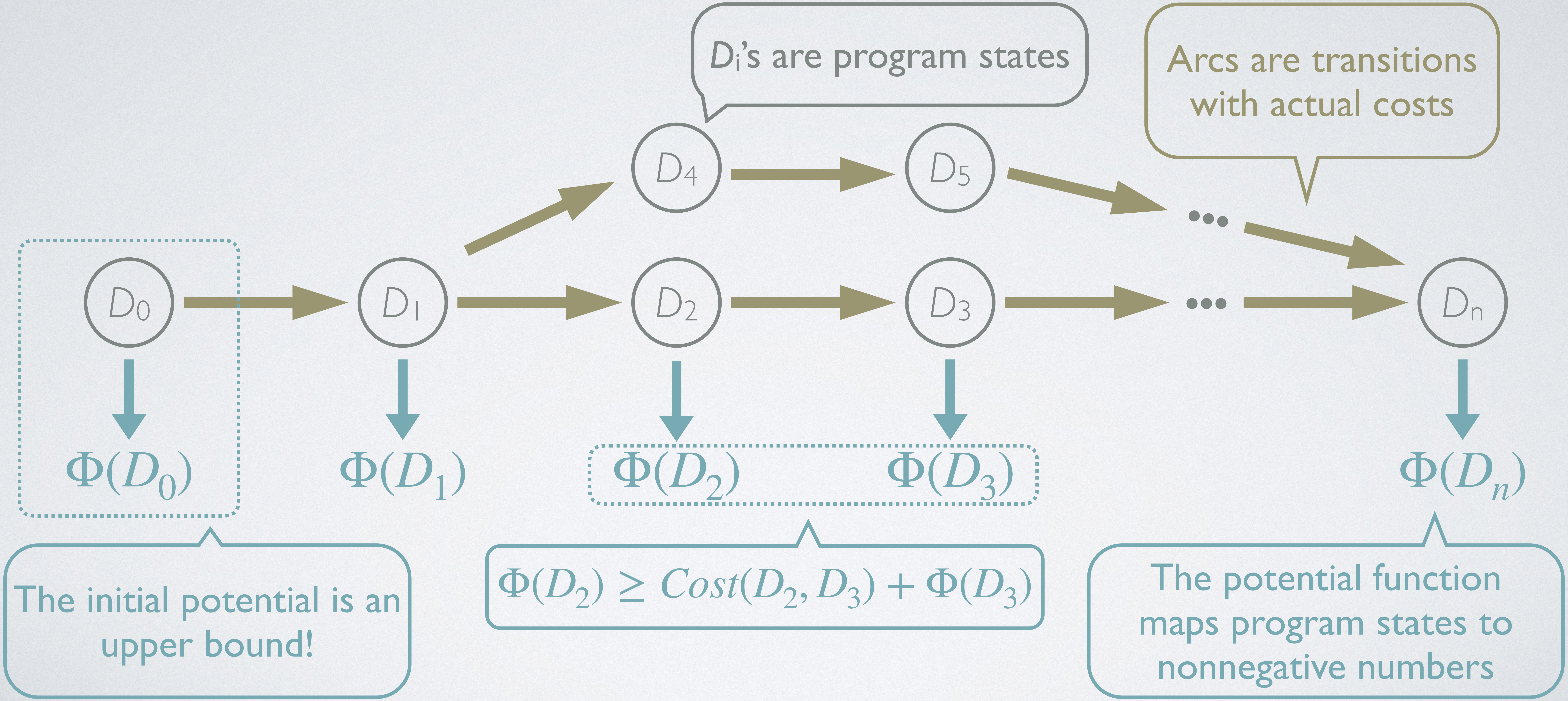
The Potential Method



The Potential Method



The Potential Method



ReSyn: Liquid Types + Linear Potentials

Potential: numeric

$$\{\ B \ \mid\ \psi\ \}^{\varphi}$$

Refinement: boolean

ReSyn: Liquid Types + Linear Potentials

Potential: numeric

$$\{\ B \mid \psi\ \}^{\varphi}$$

Refinement: boolean

$$\{\ \texttt{Int} \mid v \geq 0\ \}^{5 \cdot v}$$

A non-negative integer carrying potential equal to 5 times of its value

ReSyn: Liquid Types + Linear Potentials

Potential: numeric

$$\{\ B \mid \psi\ \}^{\varphi}$$

Refinement: boolean

$$\{\ \mathtt{Int} \mid v \geq 0\ \}^{5 \cdot v}$$

A non-negative integer carrying potential equal to 5 times of its value

$$\mathtt{List\ a}^{\mathtt{ite}(v \geq 0,1,0)}$$

A list of numbers carrying potential equal to #non-negative elements in it

RESYN: LIQUID TYPES + LINEAR POTENTIALS

```
insert = λx. λxs.
 match xs with
   Nil -> Cons x xs
   Cons hd tl -> if hd < x
     then Cons hd (tick 1 (insert x tl))
     else Cons x (Cons hd tl)
```

ReSyn: Liquid Types + Linear Potentials

```
insert = λx. λxs.
 match xs with
   Nil -> Cons x xs
   Cons hd tl -> if hd < x
     then Cons hd (tick 1 (insert x tl))
     else Cons x (Cons hd tl)
```

insert :: (x : a^{0}) -> (xs : List $a^{ite(x>v,1,0)}$) -> List a^{0}

ReSyn: Liquid Types + Linear Potentials

```
insert = λx. λxs.
  match xs with
    Nil -> Cons x xs
    Cons hd tl -> if hd < x
      then Cons hd (tick 1 (insert x tl))
      else Cons x (Cons hd tl)
```

```
insert :: (x : a^0) -> (xs : List a^ite(x>v,1,0)) -> List a^0
```

Each element that is less than **x** carries **one unit** of potential

ReSyn: Liquid Types + Linear Potentials

```
insert = λx. λxs.
  match xs with
    Nil -> Cons x xs
    Cons hd tl -> if hd < x
      then Cons hd (tick 1 (insert x tl))
      else Cons x (Cons hd tl)
```

```
insert :: (x : a^0) -> (xs : List a^ite(x>v,1,0)) -> List a^0
```

Each element that is less than **x** carries **one unit** of potential

- In ReSyn, you **cannot** express a type for insertion sort.

ReSyn: Liquid Types + Linear Potentials

```
insert = λx. λxs.
 match xs with
   Nil -> Cons x xs
   Cons hd tl -> if hd < x
     then Cons hd (tick 1 (insert x tl))
     else Cons x (Cons hd tl)
```

```
insert :: (x : a^0) -> (xs : List a^ite(x>v,1,0)) -> List a^0
```

Each element that is less than **x** carries **one unit** of potential

- In ReSyn, you **cannot** express a type for insertion sort.
- Because you can only distribute potential **uniformly** within a list.

OUTLINE

- [x] Motivation
- [x] Background: Liquid Types and the Potential Method
- [ ] Liquid Resource Types
- [ ] Evaluation

Inductive Potentials

```
data QList a where
  QNil :: QList a
  QCons :: (x : a) -> QList a¹ -> QList a
```

INDUCTIVE POTENTIALS

```
data QList a where
  QNil :: QList a
  QCons :: (x : a) -> QList a¹ -> QList a
```

the tail of the list carries **one** more unit of potential in each element than the head

INDUCTIVE POTENTIALS

```
data QList a where
  QNil :: QList a
  QCons :: (x : a) -> QList a¹ -> QList a
```

the tail of the list carries **one** more unit of potential in each element than the head

What is the potential in $L = [v_1, v_2, \cdots, v_n]$ of type `QList T`?

Inductive Potentials

```
data QList a where
  QNil :: QList a
  QCons :: (x : a) -> QList a¹ -> QList a
```

the tail of the list carries **one** more unit of potential in each element than the head

What is the potential in $L = [v_1, v_2, \cdots, v_n]$ of type `QList T`?

$$\begin{aligned}\Phi(L) &= \sum_i p + \sum_i \sum_{j>i} 1 \\ &= np + \sum_i (n-i) \\ &= \frac{n(n+2p-1)}{2}\end{aligned}$$

p is the potential of type T

Inductive Potentials

```
data QList a where
  QNil :: QList a
  QCons :: (x : a) -> QList a¹ -> QList a
```

the tail of the list carries **one** more unit of potential in each element than the head

What is the potential in $L = [v_1, v_2, \cdots, v_n]$ of type `QList T`?

$$\Phi(L) = \sum_i p + \sum_i \sum_{j>i} 1$$

$$= np + \sum_i (n - i)$$

$$= \boxed{\frac{n(n + 2p - 1)}{2}}$$

Quadratic

p is the potential of type T

INDUCTIVE POTENTIALS

```
data ISList a where
  ISNil :: ISList a
  ISCons :: (x : a) -> ISList a^ite(x>v,1,0) -> ISList a
```

INDUCTIVE POTENTIALS

```
data ISList a where
  ISNil :: ISList a
  ISCons :: (x : a) -> ISList a^ite(x>v,1,0) -> ISList a
```

the elements in the tail of the list only carries the one extra unit of potential
when their value is less than the head

INDUCTIVE POTENTIALS

```
data ISList a where
  ISNil :: ISList a
  ISCons :: (x : a) -> ISList a^ite(x>v,1,0) -> ISList a
```

the elements in the tail of the list only carries the **one** extra unit of potential
when their value is less than the head

↓

The potential in L is **#out-of-order-pairs** in it!

INDUCTIVE POTENTIALS

```
data ISList a where
  ISNil :: ISList a
  ISCons :: (x : a) -> ISList a^{ite(x>v,1,0)} -> ISList a
```

the elements in the tail of the list only carries the **one** extra unit of potential **when their value is less than the head**

The potential in L is **#out-of-order-pairs** in it!

☑ Can be used to express the bound of insertion sort

Abstract Potentials

```haskell
data QList a where
  QNil :: QList a
  QCons :: a -> QList a^1 -> QList a
```

```haskell
data ISList a where
  ISNil :: ISList a
  ISCons :: (x : a) -> ISList a^ite(x>v,1,0)
    -> ISList a
```

ABSTRACT POTENTIALS

```
data QList a where
  QNil :: QList a
  QCons :: a -> QList a^1 -> QList a
```

```
data ISList a where
  ISNil :: ISList a
  ISCons :: (x : a) -> ISList a^ite(x>v,1,0)
    -> ISList a
```

Baked-in potential annotations are **not reusable**

ABSTRACT POTENTIALS

```
data QList a where
  QNil :: QList a
  QCons :: a -> QList a¹ -> QList a
```

```
data ISList a where
  ISNil :: ISList a
  ISCons :: (x : a) -> ISList aite(x>v,1,0)
    -> ISList a
```

```
data List t <q::(t*t)->Nat> where
  Nil :: List t <q>
  Cons :: (x : t) -> List tq(x,v) <q>
    -> List t <q>
```

Baked-in potential annotations are **not reusable**

Abstract Potentials

```
data QList a where
  QNil :: QList a
  QCons :: a -> QList a¹ -> QList a
```

```
data ISList a where
  ISNil :: ISList a
  ISCons :: (x : a) -> ISList aite(x>v,1,0)
    -> ISList a
```

Baked-in potential annotations are **not reusable**

parameterized by a potential extractor

```
data List t <q::(t*t)->Nat> where
  Nil :: List t <q>
  Cons :: (x : t) -> List tq(x,v) <q>
    -> List t <q>
```

ABSTRACT POTENTIALS

```
data QList a where
  QNil :: QList a
  QCons :: a -> QList a¹ -> QList a
```

```
data ISList a where
  ISNil :: ISList a
  ISCons :: (x : a) -> ISList a^ite(x>v,1,0)
    -> ISList a
```

Baked-in potential annotations are **not reusable**

parameterized by a potential extractor

```
data List t <q::(t*t)->Nat> where
  Nil :: List t <q>
  Cons :: (x : t) -> List t^q(x,v) <q>
    -> List t <q>
```

In the `Cons` constructor, the value `q(x,v)` is **added** to the potential annotation of every element in the tail

Abstract Potentials

```
data QList a where
  QNil :: QList a
  QCons :: a -> QList a¹ -> QList a
```

```
data ISList a where
  ISNil :: ISList a
  ISCons :: (x : a) -> ISList a^ite(x>v,1,0)
    -> ISList a
```

Baked-in potential annotations are **not reusable**

parameterized by a potential extractor

```
data List t <q::(t*t)->Nat> where
  Nil :: List t <q>
  Cons :: (x : t) -> List t^q(x,v) <q>
    -> List t <q>
```

In the `Cons` constructor, the value `q(x,v)` is **added** to the potential annotation of every element in the tail

```
QList a = List a <λ_.1>
ISList a = List a <λ(x1,x2).ite(x1>x2,1,0)>
```

Constraint-Based Type Checking

```
insert :: (x : a^0) -> (xs : List a^{ite(x>v,1,0)} <λ_.0>) -> List a^0 <λ_.0>

sort :: (xs : List a^1 <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^0 <λ_.0>
```

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

CONSTRAINT-BASED TYPE CHECKING

```
insert :: (x : a^0) -> (xs : List a^ite(x>v,1,0) <λ_.0>) -> List a^0 <λ_.0>

sort :: (xs : List a^1 <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^0 <λ_.0>
```

```
sort = λxs.
 match xs with
   Nil -> Nil
   Cons hd tl ->
     insert hd
       (tick 1
         (sort tl))
```

```
[xs: List a^1 <λ(x1,x2).ite(x1>x2,1,0)>]
```

Constraint-Based Type Checking

```
insert :: (x : a^0) -> (xs : List a^ite(x>v,1,0) <λ_.0>) -> List a^0 <λ_.0>

sort :: (xs : List a^1 <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^0 <λ_.0>
```

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

```
[xs: List a^1 <λ(x1,x2).ite(x1>x2,1,0)>]
[]
```

CONSTRAINT-BASED TYPE CHECKING

```
insert :: (x : a^0) -> (xs : List a^{ite(x>v,1,0)} <λ_.0>) -> List a^0 <λ_.0>

sort :: (xs : List a^1 <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^0 <λ_.0>
```

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

```
Cons :: (hd : a^1) -> List a^{1+ite(hd>v,1,0)} <λ(x1,x2).ite(x1>x2,1,0)>
  -> List a^1 <λ(x1,x2).ite(x1>x2,1,0)>
```

CONSTRAINT-BASED TYPE CHECKING

insert :: (x : a^{0}) -> (xs : List a$^{ite(x>v,1,0)}$ <λ_.0>) -> List a^{0} <λ_.0>

sort :: (xs : List a^{1} <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^{0} <λ_.0>

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

Cons :: (hd : a^{1}) -> List a$^{1+ite(hd>v,1,0)}$ <λ(x1,x2).ite(x1>x2,1,0)>
  -> List a^{1} <λ(x1,x2).ite(x1>x2,1,0)>

Constraint-Based Type Checking

```
insert :: (x : a^{0}) -> (xs : List a^{ite(x>v,1,0)} <λ_.0>) -> List a^{0} <λ_.0>

sort :: (xs : List a^{1} <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^{0} <λ_.0>
```

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

```
Cons :: (hd : a^{1}) -> List a^{1+ite(hd>v,1,0)} <λ(x1,x2).ite(x1>x2,1,0)>
  -> List a^{1} <λ(x1,x2).ite(x1>x2,1,0)>
```

Constraint-Based Type Checking

```
insert :: (x : a^0) -> (xs : List a^{ite(x>v,1,0)} <λ_.0>) -> List a^0 <λ_.0>

sort :: (xs : List a^1 <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^0 <λ_.0>
```

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

```
[xs: List a^1 <λ(x1,x2).ite(x1>x2,1,0)>]
[]
```

Constraint-Based Type Checking

```
insert :: (x : a^0) -> (xs : List a^{ite(x>v,1,0)} <λ_.0>) -> List a^0 <λ_.0>

sort :: (xs : List a^1 <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^0 <λ_.0>
```

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

```
[xs: List a^1 <λ(x1,x2).ite(x1>x2,1,0)>]
[]
[hd: a^1, tl: List a^{1+ite(hd>v,1,0)} <λ(x1,x2).ite(x1>x2,1,0)>]
```

Constraint-Based Type Checking

```
insert :: (x : a^0) -> (xs : List a^ite(x>v,1,0) <λ_.0>) -> List a^0 <λ_.0>

sort :: (xs : List a^1 <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^0 <λ_.0>
```

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

```
[xs: List a^1 <λ(x1,x2).ite(x1>x2,1,0)>]
[]
[hd: a^1, tl: List a^1+ite(hd>v,1,0) <λ(x1,x2).ite(x1>x2,1,0)>]
[hd: a^p1, tl: List a^q1(hd,v) <q1>]
```

Constraint-Based Type Checking

```
insert :: (x : a^0) -> (xs : List a^{ite(x>v,1,0)} <λ_.0>) -> List a^0 <λ_.0>

sort :: (xs : List a^1 <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^0 <λ_.0>
```

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

```
[xs: List a^1 <λ(x1,x2).ite(x1>x2,1,0)>]
[]
[hd: a^1, tl: List a^{1+ite(hd>v,1,0)} <λ(x1,x2).ite(x1>x2,1,0)>]
[hd: a^{p1}, tl: List a^{q1(hd,v)} <q1>]
[hd: a^{p2}, tl: List a^{q2(hd,v)} <q2>]
```

Constraint-Based Type Checking

```
insert :: (x : a^{0}) -> (xs : List a^{ite(x>v,1,0)} <λ_.0>) -> List a^{0} <λ_.0>

sort :: (xs : List a^{1} <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^{0} <λ_.0>
```

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

```
[xs: List a^{1} <λ(x1,x2).ite(x1>x2,1,0)>]
[]
[hd: a^{1}, tl: List a^{1+ite(hd>v,1,0)} <λ(x1,x2).ite(x1>x2,1,0)>]
[hd: a^{p1}, tl: List a^{q1(hd,v)} <q1>]
[hd: a^{p2}, tl: List a^{q2(hd,v)} <q2>]
```

Constraint-Based Type Checking

```
insert :: (x : a^0) -> (xs : List a^{ite(x>v,1,0)} <λ_.0>) -> List a^0 <λ_.0>

sort :: (xs : List a^1 <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^0 <λ_.0>
```

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

```
[xs: List a^1 <λ(x1,x2).ite(x1>x2,1,0)>]
[]
[hd: a^1, tl: List a^{1+ite(hd>v,1,0)} <λ(x1,x2).ite(x1>x2,1,0)>]
[hd: a^{p1}, tl: List a^{q1(hd,v)} <q1>]
[hd: a^{p2}, tl: List a^{q2(hd,v)} <q2>]
[hd: a^{p2-1}, tl: List a^{q2(hd,v)} <q2>]
```

CONSTRAINT-BASED TYPE CHECKING

```
insert :: (x : a^0) -> (xs : List a^ite(x>v,1,0) <λ_.0>) -> List a^0 <λ_.0>

sort :: (xs : List a^1 <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^0 <λ_.0>
```

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

```
[xs: List a^1 <λ(x1,x2).ite(x1>x2,1,0)>]
[]
[hd: a^1, tl: List a^(1+ite(hd>v,1,0)) <λ(x1,x2).ite(x1>x2,1,0)>]
[hd: a^p1, tl: List a^q1(hd,v) <q1>]
[hd: a^p2, tl: List a^q2(hd,v) <q2>]
[hd: a^(p2-1), tl: List a^q2(hd,v) <q2>]
```

$$\exists p_1, p_2, q_1, q_2, s \,.\, \forall hd, v \,.$$

CONSTRAINT-BASED TYPE CHECKING

$\texttt{insert :: (x : a}^{0}\texttt{) -> (xs : List a}^{\texttt{ite(x>v,1,0)}}\ \texttt{<}\lambda\texttt{\_.0>) -> List a}^{0}\ \texttt{<}\lambda\texttt{\_.0>}$

$\texttt{sort :: (xs : List a}^{1}\ \texttt{<}\lambda\texttt{(x1,x2).ite(x1>x2,1,0)>) -> List a}^{0}\ \texttt{<}\lambda\texttt{\_.0>}$

```
sort = λxs.
 match xs with
   Nil -> Nil
   Cons hd tl ->
     insert hd
       (tick 1
          (sort tl))
```

$\texttt{[xs: List a}^{1}\ \texttt{<}\lambda\texttt{(x1,x2).ite(x1>x2,1,0)>]}$

$\texttt{[]}$

$\texttt{[hd: a}^{1}\texttt{, tl: List a}^{\texttt{1+ite(hd>v,1,0)}}\ \texttt{<}\lambda\texttt{(x1,x2).ite(x1>x2,1,0)>]}$

$\texttt{[hd: a}^{\texttt{p1}}\texttt{, tl: List a}^{\texttt{q1(hd,v)}}\ \texttt{<q1>]}$

$\texttt{[hd: a}^{\texttt{p2}}\texttt{, tl: List a}^{\texttt{q2(hd,v)}}\ \texttt{<q2>]}$

$\texttt{[hd: a}^{\texttt{p2-1}}\texttt{, tl: List a}^{\texttt{q2(hd,v)}}\ \texttt{<q2>]}$

$$\exists p_1, p_2, q_1, q_2, s \,.\, \forall hd, v \,. \quad p_1 + p_2 = 1 \land q_1(hd, v) + q_2(hd, v) = 1 + \textbf{ite}(hd > v, 1, 0)$$

Constraint-Based Type Checking

```
insert :: (x : a^0) -> (xs : List a^{ite(x>v,1,0)} <λ_.0>) -> List a^0 <λ_.0>

sort :: (xs : List a^1 <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^0 <λ_.0>
```

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

```
[xs: List a^1 <λ(x1,x2).ite(x1>x2,1,0)>]
[]
[hd: a^1, tl: List a^{1+ite(hd>v,1,0)} <λ(x1,x2).ite(x1>x2,1,0)>]
[hd: a^{p1}, tl: List a^{q1(hd,v)} <q1>]
[hd: a^{p2}, tl: List a^{q2(hd,v)} <q2>]
[hd: a^{p2-1}, tl: List a^{q2(hd,v)} <q2>]
```

$$\exists p_1, p_2, q_1, q_2, s \,.\, \forall hd, v \,.$$

CONSTRAINT-BASED TYPE CHECKING

```
insert :: (x : a^0) -> (xs : List a^ite(x>v,1,0) <λ_.0>) -> List a^0 <λ_.0>

sort :: (xs : List a^1 <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^0 <λ_.0>
```

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

```
[xs: List a^1 <λ(x1,x2).ite(x1>x2,1,0)>]
[]
[hd: a^1, tl: List a^{1+ite(hd>v,1,0)} <λ(x1,x2).ite(x1>x2,1,0)>]
[hd: a^p1, tl: List a^{q1(hd,v)} <q1>]
[hd: a^p2, tl: List a^{q2(hd,v)} <q2>]
[hd: a^{p2-1}, tl: List a^{q2(hd,v)} <q2>]
```

$\exists p_1, p_2, q_1, q_2, s \,.\, \forall hd, v \,.$ $\qquad p_2 - 1 \geq 0$

Constraint-Based Type Checking

```
insert :: (x : a^{0}) -> (xs : List a^{ite(x>v,1,0)} <λ_.0>) -> List a^{0} <λ_.0>

sort :: (xs : List a^{1} <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^{0} <λ_.0>
```

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

```
[xs: List a^{1} <λ(x1,x2).ite(x1>x2,1,0)>]
[]
[hd: a^{1}, tl: List a^{1+ite(hd>v,1,0)} <λ(x1,x2).ite(x1>x2,1,0)>]
[hd: a^{p1}, tl: List a^{q1(hd,v)} <q1>]
[hd: a^{p2}, tl: List a^{q2(hd,v)} <q2>]
[hd: a^{p2-1}, tl: List a^{q2(hd,v)} <q2>]
```

$$\exists p_1, p_2, q_1, q_2, s \,.\, \forall hd, v \,.$$

Constraint-Based Type Checking

```
insert :: (x : a^0) -> (xs : List a^ite(x>v,1,0) <λ_.0>) -> List a^0 <λ_.0>
sort :: (xs : List a^1+s <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^0+s <λ_.0>
```

```
sort = λxs.
 match xs with
   Nil -> Nil
   Cons hd tl ->
     insert hd
       (tick 1
          (sort tl))
```

```
[xs: List a^1 <λ(x1,x2).ite(x1>x2,1,0)>]
[]
[hd: a^1, tl: List a^1+ite(hd>v,1,0) <λ(x1,x2).ite(x1>x2,1,0)>]
[hd: a^p1, tl: List a^q1(hd,v) <q1>]
[hd: a^p2, tl: List a^q2(hd,v) <q2>]
[hd: a^p2-1, tl: List a^q2(hd,v) <q2>]
```

$\exists p_1, p_2, q_1, q_2, s \,.\, \forall hd, v \,.$ $\quad q_2(hd, v) \geq 1 + s(hd, v)$

Constraint-Based Type Checking

```
insert :: (x : a^{0}) -> (xs : List a^{ite(x>v,1,0)} <λ_.0>) -> List a^{0} <λ_.0>
sort :: (xs : List a^{1+s} <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^{0+s} <λ_.0>
```

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

```
[xs: List a^{1} <λ(x1,x2).ite(x1>x2,1,0)>]
[]
[hd: a^{1}, tl: List a^{1+ite(hd>v,1,0)} <λ(x1,x2).ite(x1>x2,1,0)>]
[hd: a^{p1}, tl: List a^{q1(hd,v)} <q1>]
[hd: a^{p2}, tl: List a^{q2(hd,v)} <q2>]
[hd: a^{p2-1}, tl: List a^{q2(hd,v)} <q2>]
```

$$\exists p_1, p_2, q_1, q_2, s \,.\, \forall hd, v \,.$$

Constraint-Based Type Checking

```
insert :: (x : a^0) -> (xs : List a^ite(x>v,1,0) <λ_.0>) -> List a^0 <λ_.0>
sort :: (xs : List a^1+s <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^0+s <λ_.0>
```

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

```
[xs: List a^1 <λ(x1,x2).ite(x1>x2,1,0)>]
[]
[hd: a^1, tl: List a^1+ite(hd>v,1,0) <λ(x1,x2).ite(x1>x2,1,0)>]
[hd: a^p1, tl: List a^q1(hd,v) <q1>]
[hd: a^p2, tl: List a^q2(hd,v) <q2>]
[hd: a^p2-1, tl: List a^q2(hd,v) <q2>]
```

$\exists p_1, p_2, q_1, q_2, s \,.\, \forall hd, v \,.$ $\quad 0 + s(hd, v) \geq \mathbf{ite}(hd > v, 1, 0)$

Constraint-Based Type Checking

```
insert :: (x : a^0) -> (xs : List a^{ite(x>v,1,0)} <λ_.0>) -> List a^0 <λ_.0>
sort :: (xs : List a^{1+s} <λ(x1,x2).ite(x1>x2,1,0)>) -> List a^{0+s} <λ_.0>
```

```
sort = λxs.
  match xs with
    Nil -> Nil
    Cons hd tl ->
      insert hd
        (tick 1
          (sort tl))
```

```
[xs: List a^1 <λ(x1,x2).ite(x1>x2,1,0)>]
[]
[hd: a^1, tl: List a^{1+ite(hd>v,1,0)} <λ(x1,x2).ite(x1>x2,1,0)>]
[hd: a^{p1}, tl: List a^{q1(hd,v)} <q1>]
[hd: a^{p2}, tl: List a^{q2(hd,v)} <q2>]
[hd: a^{p2-1}, tl: List a^{q2(hd,v)} <q2>]
```

Second-Order Conditional Linear Arithmetic Constraints

Type Soundness

Type Soundness

If a closed program ***E*** of type ***T*** is well-typed with ***Q*** units of initial potential, then the evaluation of ***E*** with ***Q*** units of initial resource will not get stuck, and when ***E*** evaluates to a value ***V***, ***V*** will satisfy the constraints specified by ***T***.

OUTLINE

- ☑ Motivation
- ☑ Background: Liquid Types and the Potential Method
- ☑ Liquid Resource Types
- ☐ Evaluation

Reusable Datatypes

Datatype	Type	Potential

Reusable Datatypes

Datatype	Type	Potential

List	data List t <q::t->t->Nat> where
	  Nil :: List t <q>
	  Cons :: (x : t) -> List t$^{q(x,v)}$ <q> -> List t <q>
		Quadratic
		$\sum_{i<j} q(v_i, v_j)$

REUSABLE DATATYPES

Datatype	Type	Potential

List	`data List t <q::t->t->Nat> where`
	`  Nil :: List t <q>`
	`  Cons :: (x : t) -> List t^{q(x,v)} <q> -> List t <q>`
		Quadratic
		$\sum_{i<j} q(v_i, v_j)$

List	`data EList t <q::Nat> where`
	`  Nil :: EList t <q>`
	`  Cons :: (x : t^q) -> EList t <2*q> -> EList t <q>`
		Exponential
		$q \cdot (2^n - 1)$

REUSABLE DATATYPES

Datatype	Type	Potential

List	data List t <q::t->t->Nat> where
	  Nil :: List t <q>
	  Cons :: (x : t) -> List t$^{q(x,v)}$ <q> -> List t <q>
		Quadratic
		$\sum_{i<j} q(v_i, v_j)$

List	data EList t <q::Nat> where
	  Nil :: EList t <q>
	  Cons :: (x : t^{q}) -> EList t <2*q> -> EList t <q>
		Exponential
		$q \cdot (2^n - 1)$

Binary tree	data LTree t <q::Nat> where
	  Leaf :: (x : t) -> LTree t <q>
	  Node :: LTree t^{q} <q> -> LTree t^{q} <q> -> Ltree t <q>
		Size * Height
		$\approx q \cdot n \log_2 n$

Reusable Datatypes

Datatype	Type	Potential

List	`data List t <q::t->t->Nat> where`
	`  Nil :: List t <q>`
	`  Cons :: (x : t) -> List t^q(x,v) <q> -> List t <q>`
		Quadratic
		$\sum_{i<j} q(v_i, v_j)$

List	`data EList t <q::Nat> where`
	`  Nil :: EList t <q>`
	`  Cons :: (x : t^q) -> EList t <2*q> -> EList t <q>`
		Exponential
		$q \cdot (2^n - 1)$

Binary tree	`data LTree t <q::Nat> where`
	`  Leaf :: (x : t) -> LTree t <q>`
	`  Node :: LTree t^q <q> -> LTree t^q <q> -> Ltree t <q>`
		Size * Height
		$\approx q \cdot n \log_2 n$

Pathed potential tree	`data PTree t <p::t->Bool, q::Nat> where`
	`  Leaf :: PTree t <p, q>`
	`  Node :: (x : t^q) -> PTree t <p, ite(p(x),q,0)>`
	`    -> PTree t <p, ite(p(x),0,q)> -> PTree t <p, q>`
		Parameterized by a specific path
		$q \cdot \lvert \ell \rvert$

Benchmark Programs

Kind	Description	Bound on #recursive-calls	Time (in sec.)
Polynomial Quadratic Potential	All Ordered Pairs	n^2+n	0.5
	List Reverse (Slow)	$0.5n^2+1.5n$	0.4
	List Remove Duplicates	$0.5n^2+1.5n$	0.4
	Insertion Sort (Coarse)	$0.5n^2+1.5n$	0.6
	Selection Sort	$1.5n^2+2.5n$	0.5
	Quick Sort	$1.5n^2+1.5n$	1.0
	Merge Sort	n^2+n	0.9
Non-Polynomial Potential	Subset Sum	$2(2^n-1)$	0.3
	Merge Sort Flatten	$(n+1)*h$	0.9
Value-Dependent Potential	Insertion Sort (Fine)	#out-of-order-pairs	5.4
	BST Insert	|insertion path|	2.4
	BST Member	|search path|	6.0

LIQUID RESOURCE TYPES

```
data List t <q::t->t->Nat> where
  Nil :: List t <q>
  Cons :: (x : t) -> List tq(x,v) <q> -> List t <q>
```

LIQUID RESOURCE TYPES

```
data List t <q::t->t->Nat> where
  Nil :: List t <q>
  Cons :: (x : t) -> List t^q(x,v) <q> -> List t <q>
```

Contributions:

- Verification of value-dependent super-linear resource bounds
- Soundness proof of the type system
- Effective prototype implementation

Liquid Resource Types

```
data List t <q::t->t->Nat> where
  Nil :: List t <q>
  Cons :: (x : t) -> List t^q(x,v) <q> -> List t <q>
```

Contributions:

- Verification of value-dependent super-linear resource bounds
- Soundness proof of the type system
- Effective prototype implementation

Limitations:

- Only inductively defined potentials
- Only univariate bounds
- No bound inference