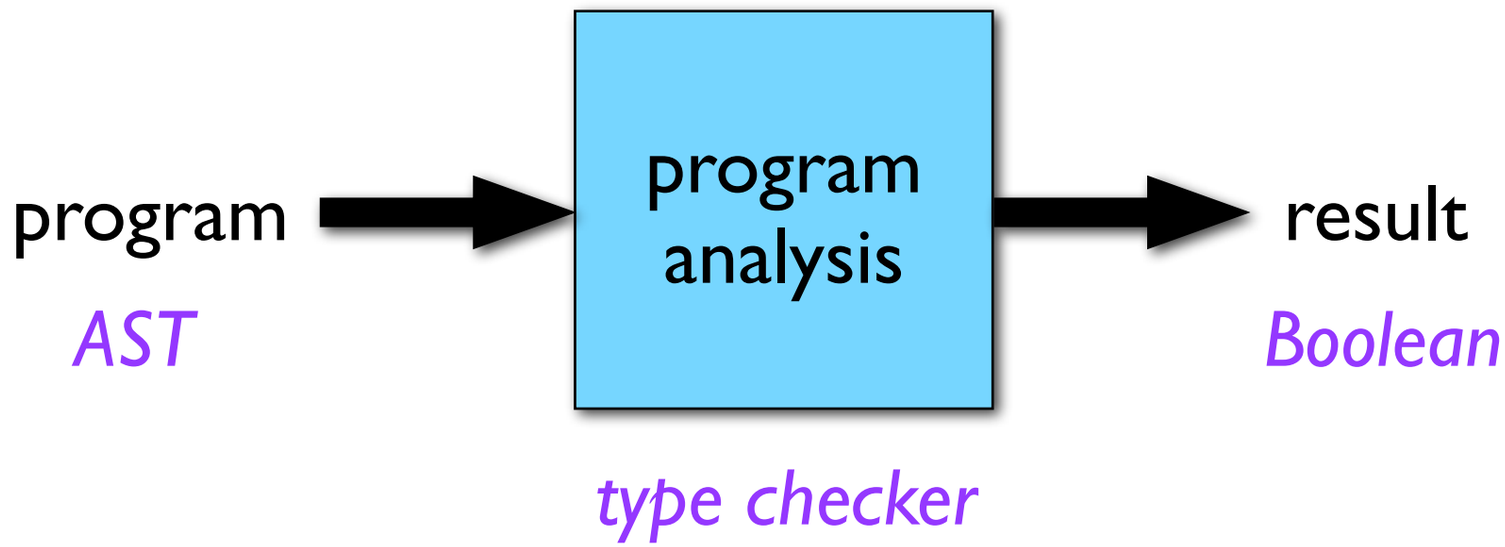
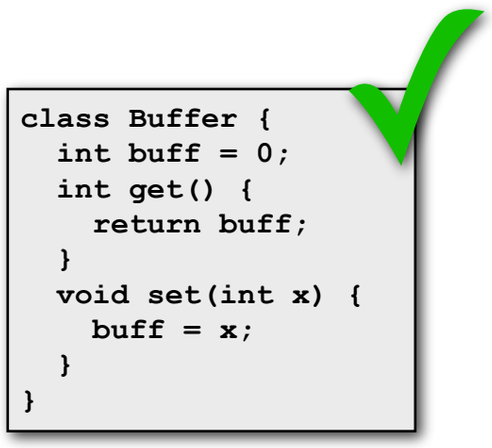
Toward Variational Data Structures

Eric Walkingshaw
University of Marburg

Program analysis



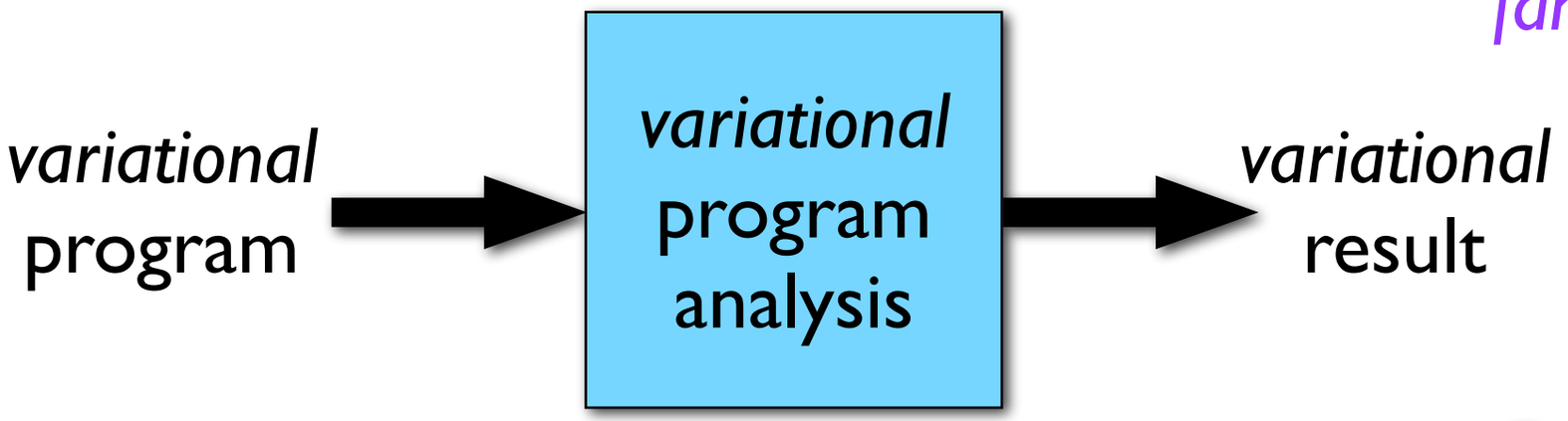


```
class Buffer {
  int buff = 0;
  int get() {
    return buff;
  }
  void set(int x) {
    buff = x;
  }
}
```

Variational program analyses

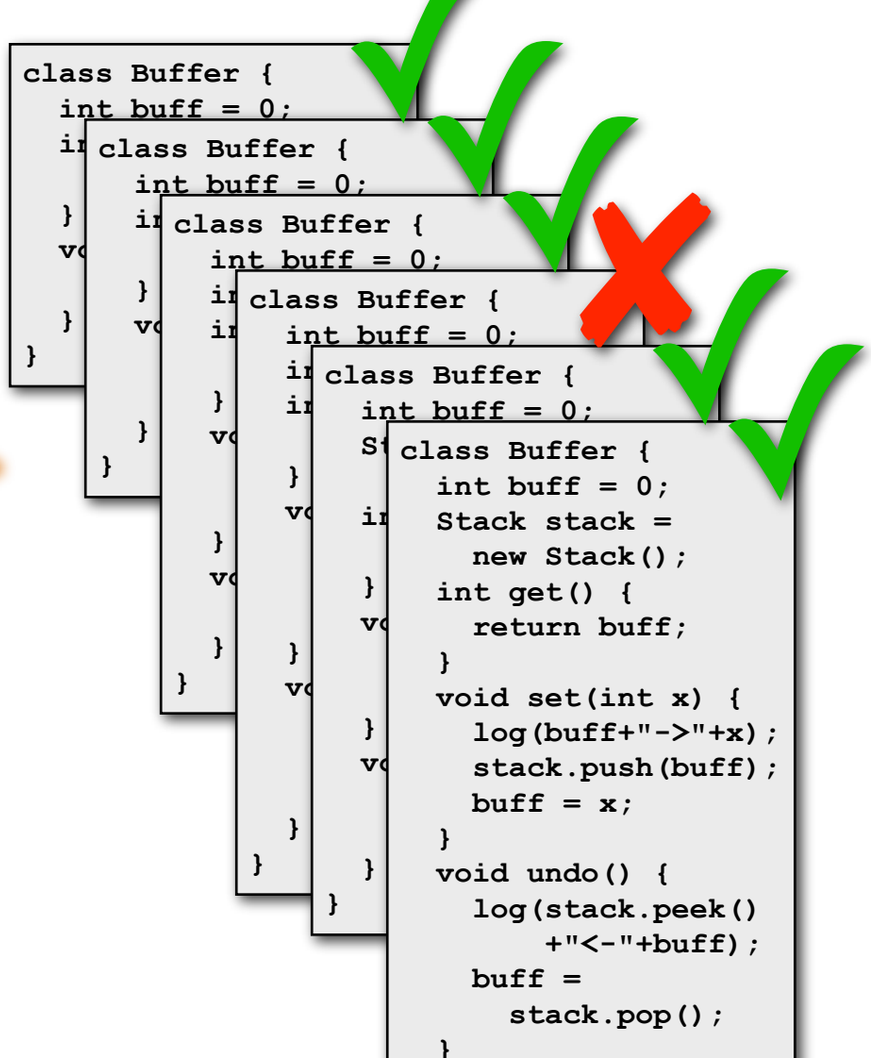
"variability-aware"

"family-based"




```
class Buffer {
  int buff = 0;
#ifdef UndoOne
  int back = 0;
#endif
#ifdef UndoMany
  Stack stack =
    new Stack();
#endif
  int get() {
    return buff;
  }
  void set(int x) {
#ifdef Logging
    log(buff+"->"+x);
#endif
#ifdef UndoOne
    back = buff;
#endif
#ifdef UndoMany
    stack.push(buff);
#endif
    buff = x;
  }
```

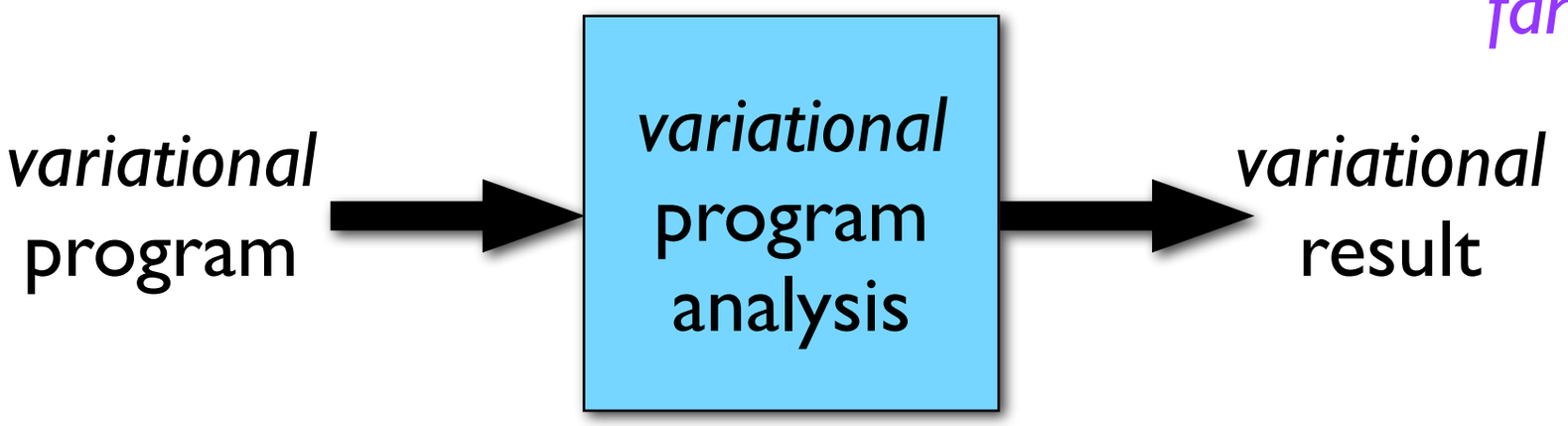
```
#ifdef UndoOne
  void undo() {
#ifdef Logging
    log(back+"<-"+bug);
#endif
    buff = back;
  }
#endif
#ifdef UndoMany
  void undo() {
#ifdef Logging
    log(stack.peek()
        +"<-"+buff);
#endif
    buff =
      stack.pop();
  }
#endif
}
```



Variational program analyses

"variability-aware"

"family-based"



How do we implement these things efficiently?

Goal of this talk

Promote foundational research
on *variational data structures*

Develop variational program analyses systematically:

- identify where variation occurs in underlying data types
- identify/prioritize operations
- analyze trade-offs between variational implementations

Many other applications too!

Collaborators

Christian Kästner
Carnegie Mellon

Sven Apel
Passau

Martin Erwig
Oregon State

Eric Bodden
Darmstadt

Variational …

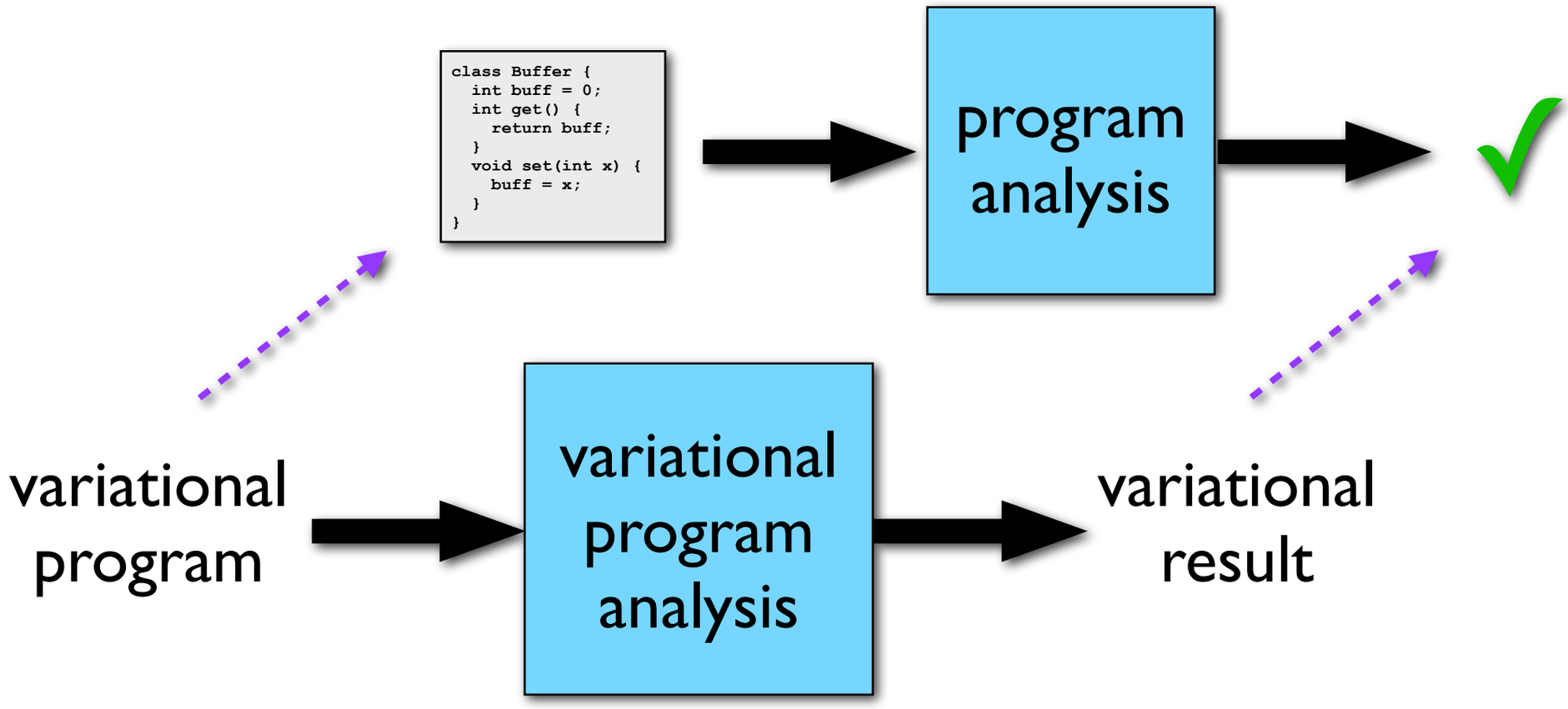
- parsing in TypeChef
- type checking in FFJ_{PL}
- type checking in TypeChef
- type inference in VLC
- programming in Haskell DSL
- data-flow analysis in SPL^{LIFT}

Outline of talk

1. Introduction
2. Functional view of variation
3. Variational data types
4. Variational lists
5. Conclusion

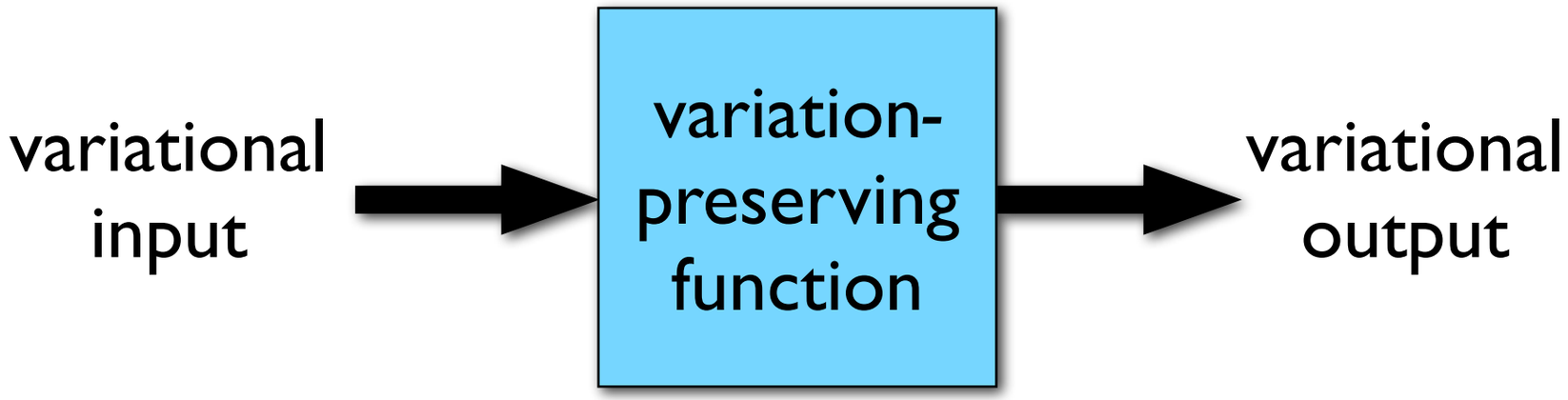
Variation preservation

Variability in input must be *preserved* in output (correspondence between variants)



Variation-preserving functions

Can generalize beyond variational program analyses



Supports *computing with variational data*

e.g. variational test execution

Supports *simultaneous exploration of alternatives*

e.g. route/travel planning

Variation-preserving functions

Plain function

$f : (A_1, \ldots, A_n) \rightarrow B$

Variation-preserving function

$f_V : (V[A_1], \ldots, V[A_n]) \rightarrow V[B]$

may choose some inputs to be non-variational

Promoting static variation to input parameters

```
String format(String s) {
#if COLOR
   ...
#else
   ...
#endif
}
```

```
String format(Boolean color, String l) {
  if (color) {
     ...
  } else {
     ...
  }
}
```

$\texttt{format}_V$: (V[Boolean], String) → V[String]

Outline of talk

1. Introduction
2. Functional view of variation
3. Variational data types
4. Variational lists
5. Conclusion

Variational data types

Configuration space

$C = \{c_1, \ldots, c_n\}$

configurations

Variational data type

$V[A] : C \rightarrow A$

mapping from configurations to values of type A

$\{(c_1, a_1), \ldots, (c_n, a_n)\}$

explicit representation
(doesn't scale, many duplicates)

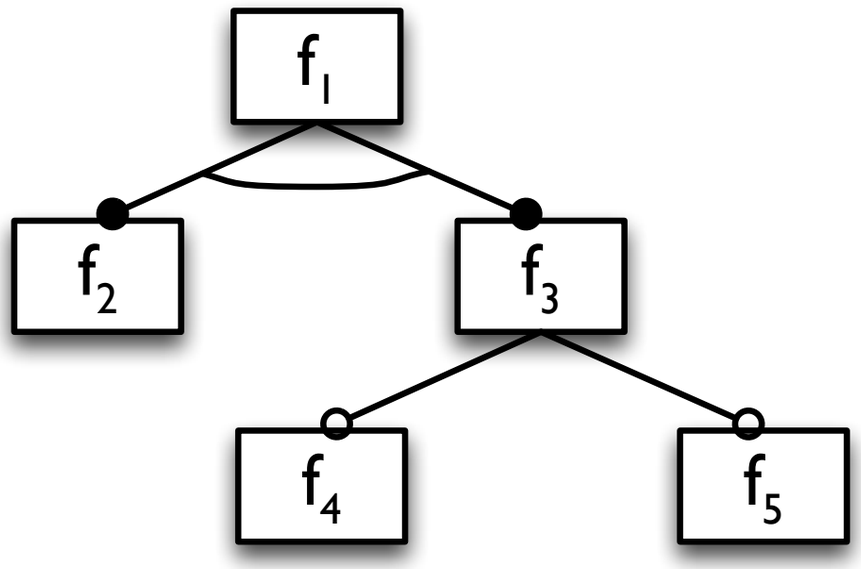
Variational data types

Configuration space

$C = \{c_1, \ldots, c_n\}$

Variational data type

$V[A] : C \rightarrow A$



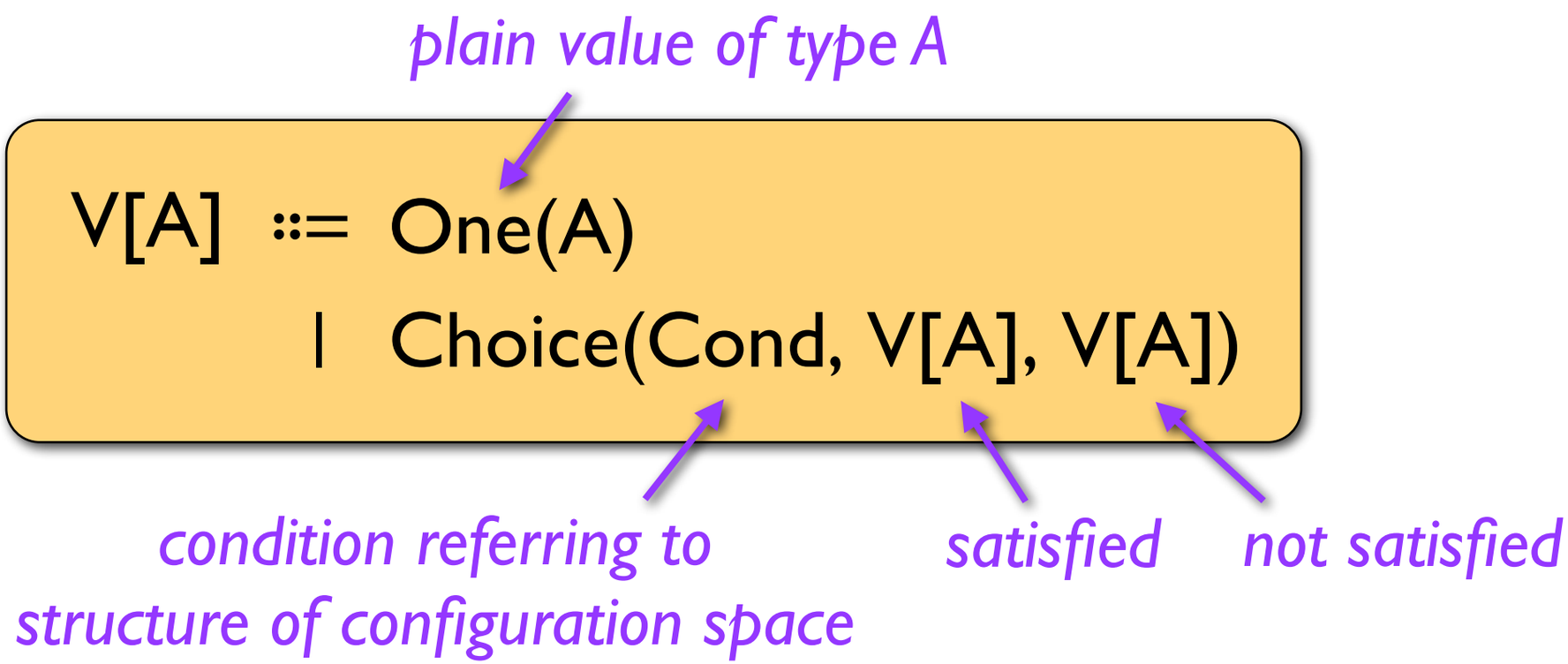
Can exploit structure in configuration space

$\{ (f_4 \wedge f_5, a_1), (\neg f_4 \vee \neg f_5, a_2) \}$

Labels must partition the configuration space

Implementing variational data types

Option 1: "Choice trees"



Choice(C_1, One(a_1),
Choice(C_2, One(a_2), One(a_3))

$C_1\langle a_1, C_2\langle a_2, a_3\rangle\rangle$ *syntactic sugar*

Implementing variational data types

Option 1: “Choice trees”

```
V[A]  ::=  One(A)
       |   Choice(Cond, V[A], V[A])
```

If conditions are:

- simple references ≡ choice calculus
- boolean formulas ≡ TypeChef

Implementing variational data types

Option 2: “Formula maps”

existing map data structure

V[A] ::= Map[A, Formula]

Map({ $a_1 \Rightarrow f_4 \wedge f_5$,
$a_2 \Rightarrow \neg f_4 \vee \neg f_5$ })

Implementation trade-offs

Partition invariant:

- choice trees – *by construction*
- formula maps – *explicitly maintained*

exactly one value per configuration

Space minimality:

- choice trees – *not maintained*
- formula maps – *by construction*

each variant appears once

	add	select
choice tree	log n	log n
formula map	n	n

(assumes SAT-solving is very fast)

Computing with variational data types

Can systematically "lift" many operations on A to V[A]

map : (A → B) → V[A] → V[B]
flatMap : (A → V[B]) → V[A] → V[B]

V[·] *is a monad*

v = C⟨13, D⟨42, 67⟩⟩ : V[Integer]

map(even, v) = C⟨false, D⟨true, false⟩⟩ : V[Boolean]

Outline of talk

1. Introduction
2. Functional view of variation
3. Variational data types
4. Variational lists
5. Conclusion

Need for specialized variational data structures

List[A]

```
nil    : List[A]
cons   : (A, List[A])   → List[A]
get    : (List[A], Int) → A
length : List[A]        → Int
```

List #1	List #2	V[List[A]]	List[V[A]]
[1, 2, 3]	[1, 4, 3]	D⟨[1, 2, 3], [1, 4, 3]⟩	[1, D⟨2, 4⟩, 3]
[5, 6]	[5, 6, 7]	D⟨[5, 6], [5, 6, 7]⟩	*impossible*
		no sharing	*not expressive*

Variational tail list

Option 1: Add variational constructor

TList[A]

```
nil    : TList[A]
cons   : (A, TList[A])   → TList[A]
vlist  : V[TList[A]]     → TList[A]
get    : (TList[A], Int) → V[Opt[A]]
length : TList[A]        → V[Int]
```

can be null

x = cons(5, cons(6, vlist(D⟨nil, cons(7, nil)⟩))) D⟨[5, 6], [5, 6, 7]⟩

length(x) = D⟨2, 3⟩

get(x, 3) = D⟨null, 7⟩

- maximally expressive
- generic technique for algebraic data types (GTTSE'12)

Variational tail list

Option 1: Add variational constructor

TList[A]

```
nil    : TList[A]
cons   : (A, TList[A])   → TList[A]
vlist  : V[TList[A]]     → TList[A]
get    : (TList[A], Int) → V[Opt[A]]
length : TList[A]        → V[Int]
```

y = cons(1, vlist(D⟨cons(2, cons(3, nil)),
cons(4, cons(3, nil))⟩))

D⟨[1, 2, 3], [1, 4, 3]⟩

only supports prefix sharing

Variational, optional element list

Option 2: Make head variational and optional

OList[A]

```
nil    : OList[A]
cons   : (V[Opt[A]], OList[A])  → OList[A]
get    : (OList[A], Int)        → V[Opt[A]]
length : OList[A]               → V[Int]
```

x = cons(One(5), cons(One(6), cons(D⟨One(null), One(7)⟩, nil)))

D⟨[5, 6], [5, 6, 7]⟩

y = cons(One(1), cons(D⟨One(2), One(4)⟩, cons(3, nil)))

D⟨[1, 2, 3], [1, 4, 3]⟩

Outline of talk

1. Introduction
2. Functional view of variation
3. Variational data types
4. Variational lists
5. Conclusion

Summary

Functional view of computing with variation

- variation in inputs *preserved* in output

Variational data types – V[·]

- choice trees vs. formula maps (time vs. space)
- systematic lifting of plain functions

Variational data structures

- limitations of V[·]
- some example variational list data structures

Conclusion

Goal of this talk

Promote foundational research
on *variational data structures*

Opportunity for us as a community:

- many external applications
- we're the experts on coping with variation
- surface is barely scratched …