On the Turing Completeness of Typed Feature Structure Unification

Guy Emerson

What I'll Cover

- Proof: any Turing Machine can be simulated as unification in a Delph-in type system

What I'll Cover

- Proof: any Turing Machine can be simulated as unification in a Delph-in type system
  - … using only two features

What I'll Cover

- Proof: any Turing Machine can be simulated as unification in a Delph-in type system
  - … using only two features
- Demo: untyped lambda calculus (Turing-complete)

What I'll Cover

- Proof: any Turing Machine can be simulated as unification in a Delph-in type system
  - … using only two features
- Demo: untyped lambda calculus (Turing-complete)
- Recipe: how to add "relational constraints" to a Delph-in grammar

What I'll Cover

- Proof: any Turing Machine can be simulated as unification in a Delph-in type system
  - … using only two features
- Demo: untyped lambda calculus (Turing-complete)
- Recipe: how to add "relational constraints" to a Delph-in grammar
- Recipe (experimental): how to add *nondeterministic* "relational constraints" to a Delph-in grammar

Didn’t we know about Turing completeness?

- Previous work demonstrated Turing completeness of unification

Didn’t we know about Turing completeness?

- Previous work demonstrated Turing completeness of unification
- BUT: Delph-in formalism more restricted
  - Copestake (2002): “the type inference system is essentially non-recursive”

Didn't we know about Turing completeness?

- Previous work demonstrated Turing completeness of unification
- BUT: Delph-in formalism more restricted
  - Copestake (2002): "the type inference system is essentially non-recursive"
- Different kind of recursion from previous work!

Didn’t you present this at the 2019 summit?

- More precisely, how Turing completeness comes in

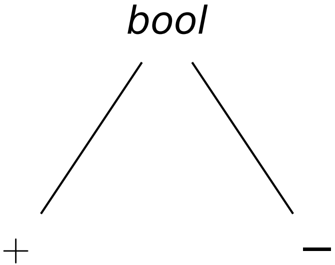
Didn’t you present this at the 2019 summit?

- More precisely, how Turing completeness comes in
- How to be more systematic (more user-friendly?)

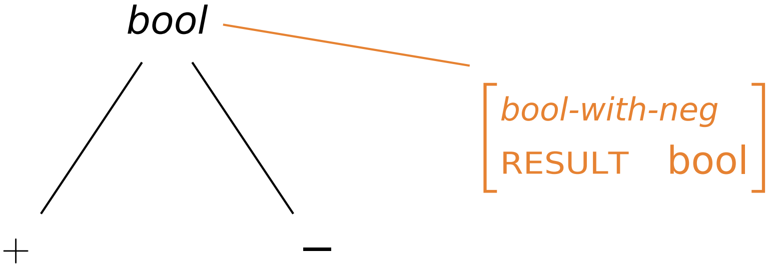
Path to Turing Completeness

- Computation types
- Recursive computation types
- Turing-complete computation types

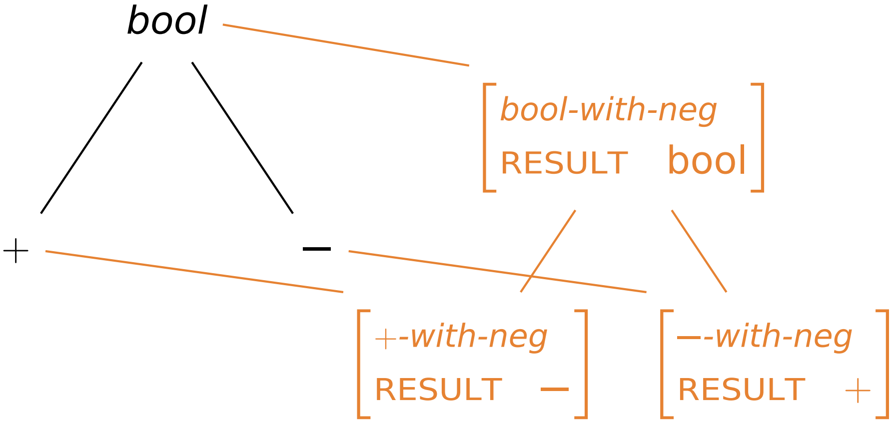
Computation Types



Computation Types



Computation Types



Computation Types

- Data type (e.g. *bool*)
- Computation type (e.g. *bool-with-neg*)
  - Subtype of data type
  - Additional feature (e.g. RESULT)
- Common subtypes define computation
  (e.g. $+$-*with-neg*, $-$-*with-neg*)

Recursive Computation Types

- Recursive data type (e.g. *list*)
- Recursive computation type (e.g. *list-with-diff-list*)
  - Subtype of data type
  - Additional feature (e.g. RESULT)
- Common subtypes define computation (e.g. *cons-with-diff-list*, *null-with-diff-list*)

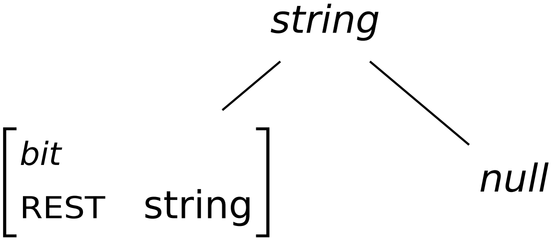
Recursive Data Types

list

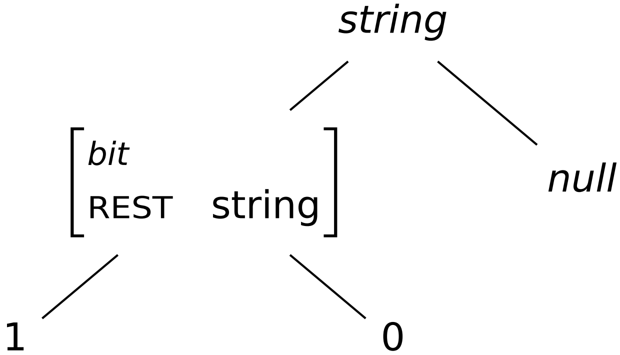
$$\begin{bmatrix} \textit{cons} & \\ \text{FIRST} & \text{*top*} \\ \text{REST} & \text{list} \end{bmatrix}$$

null

Recursive Data Types

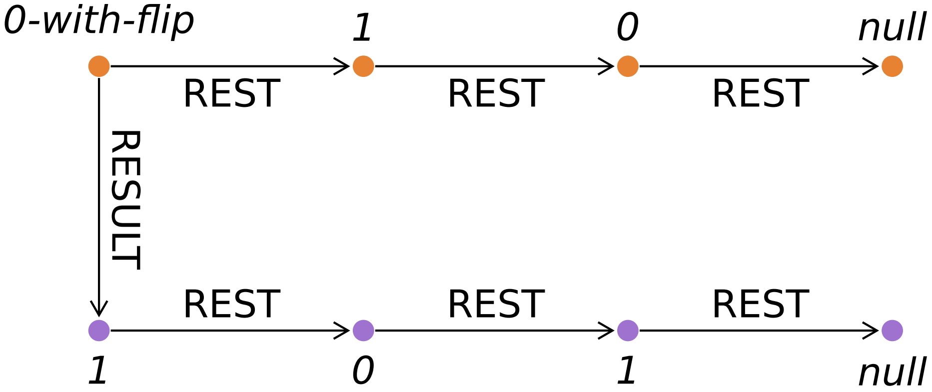


Recursive Data Types

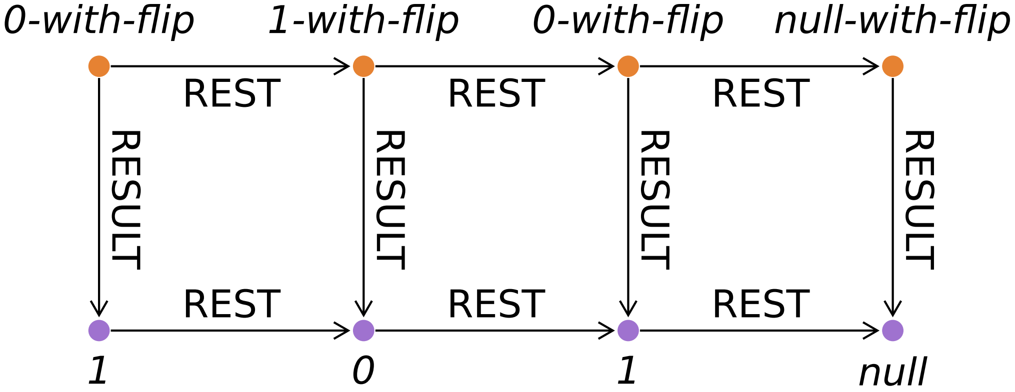


Recursive Data Types

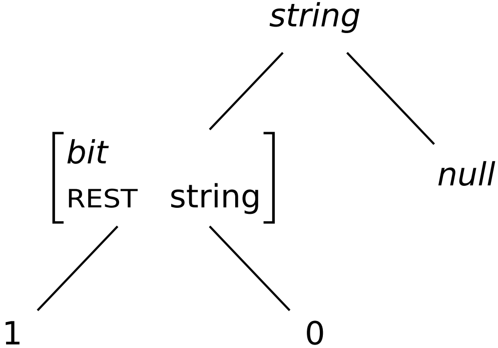
Recursive Computation Types



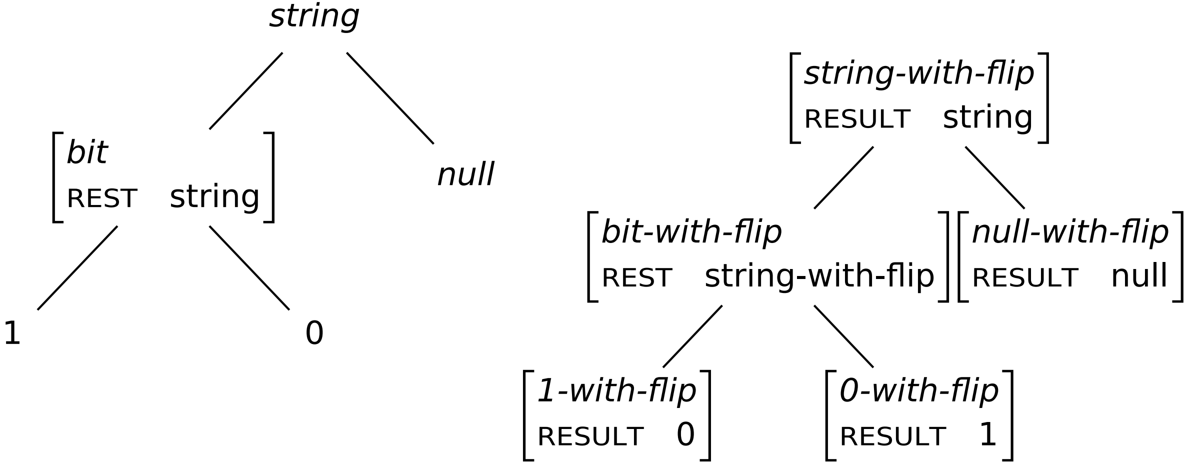
Recursive Computation Types



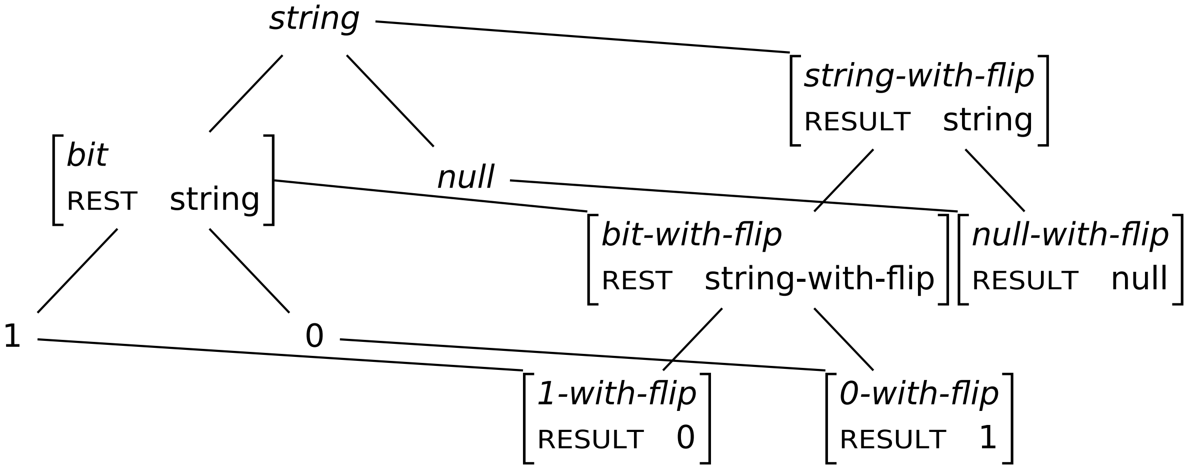
Recursive Computation Types



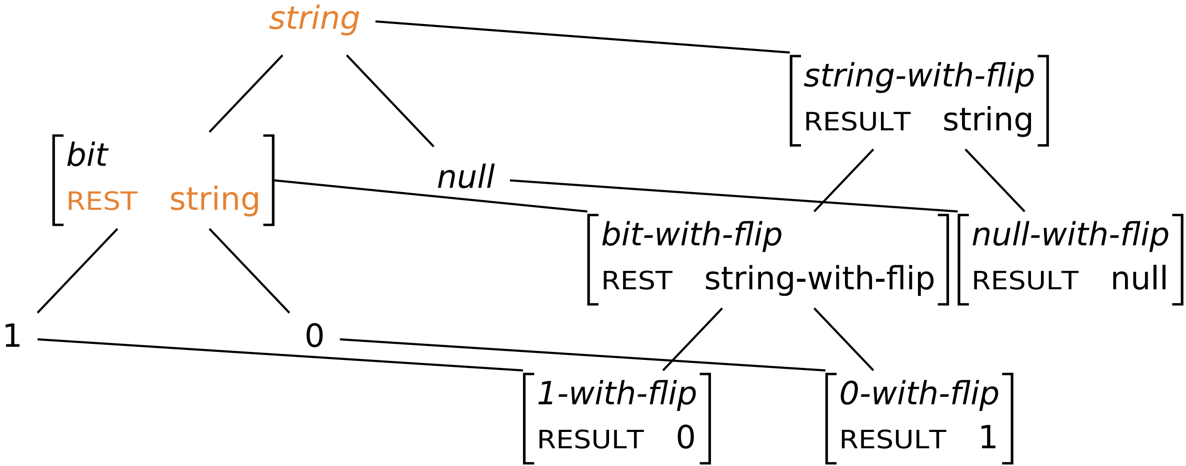
Recursive Computation Types



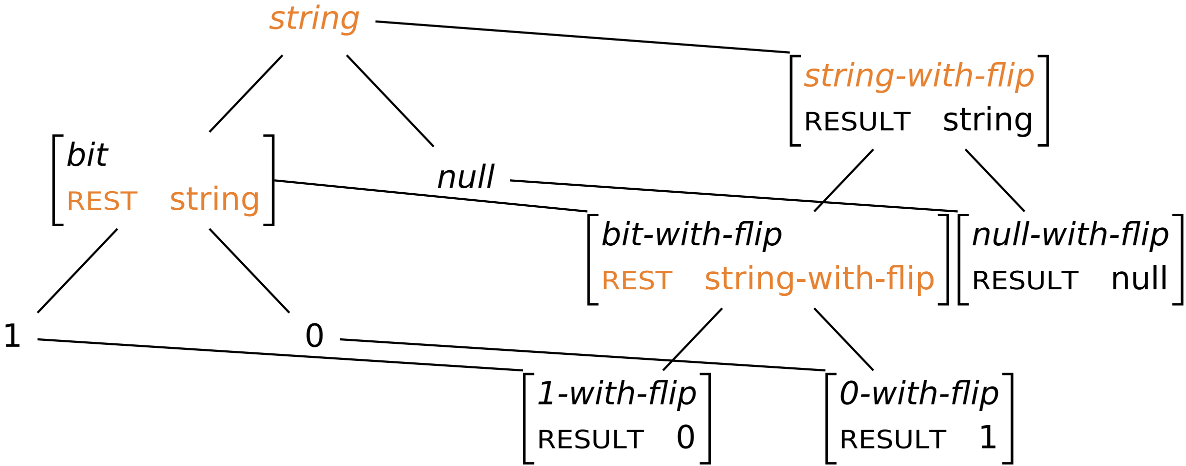
Recursive Computation Types



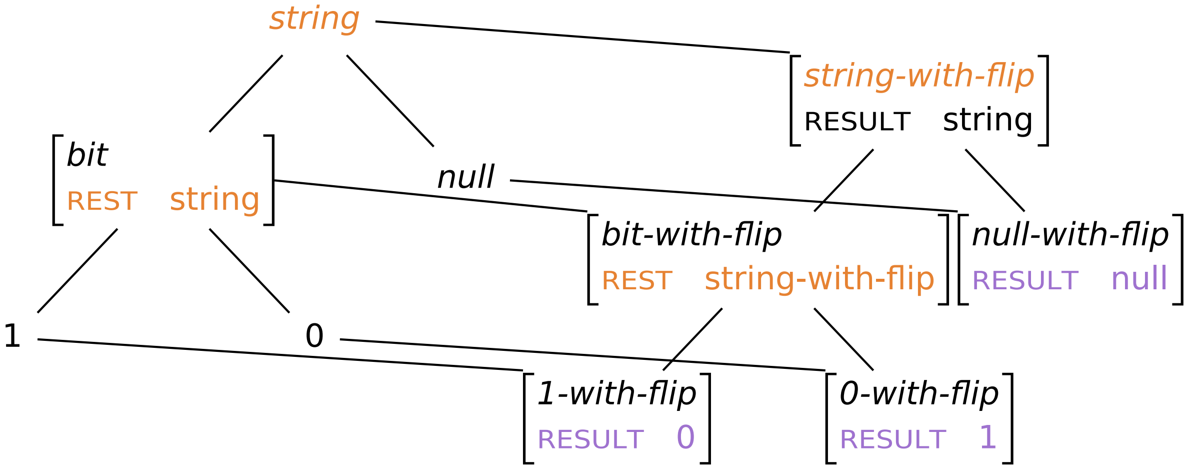
Recursive Computation Types



Recursive Computation Types

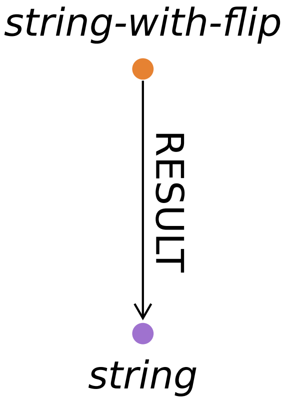


Recursive Computation Types

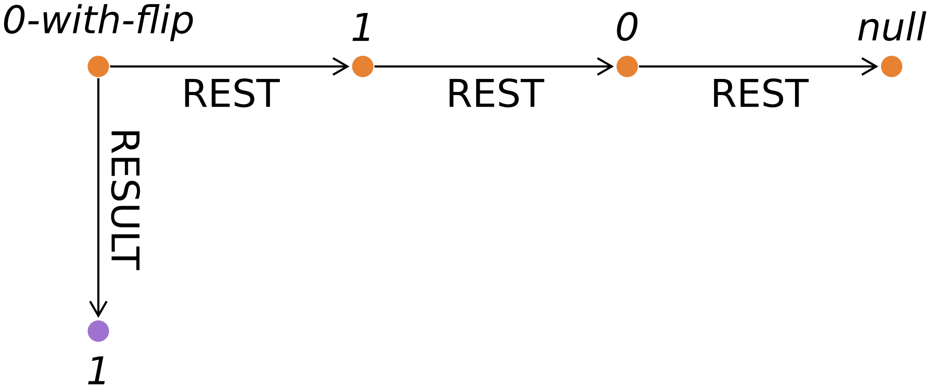


Unification can create paths

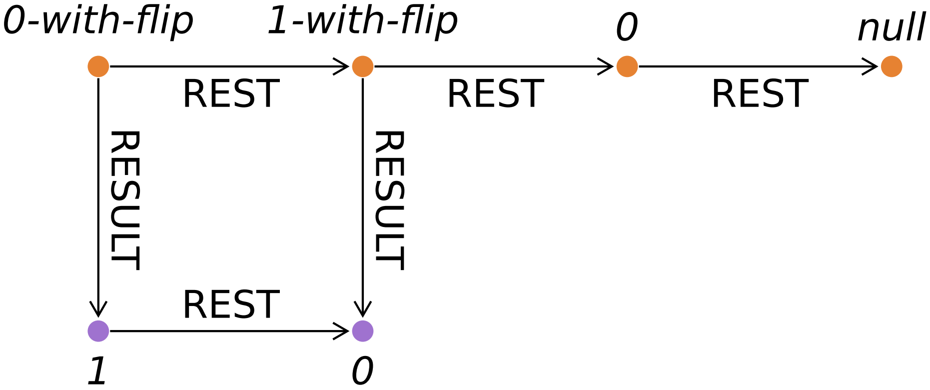
Unification can create paths



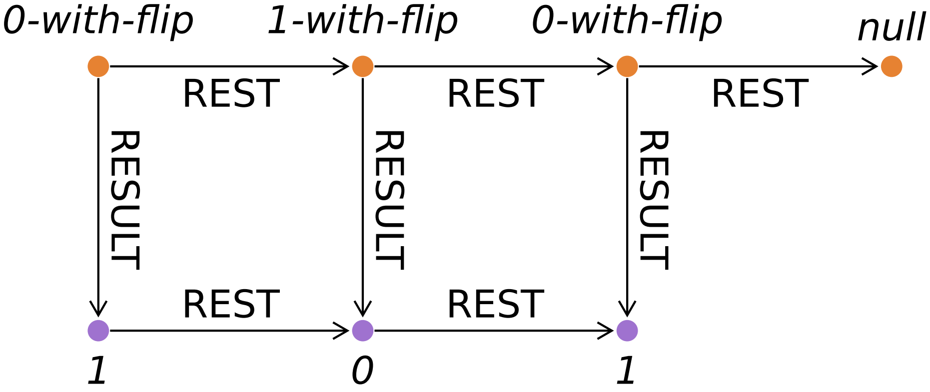
Unification can create paths



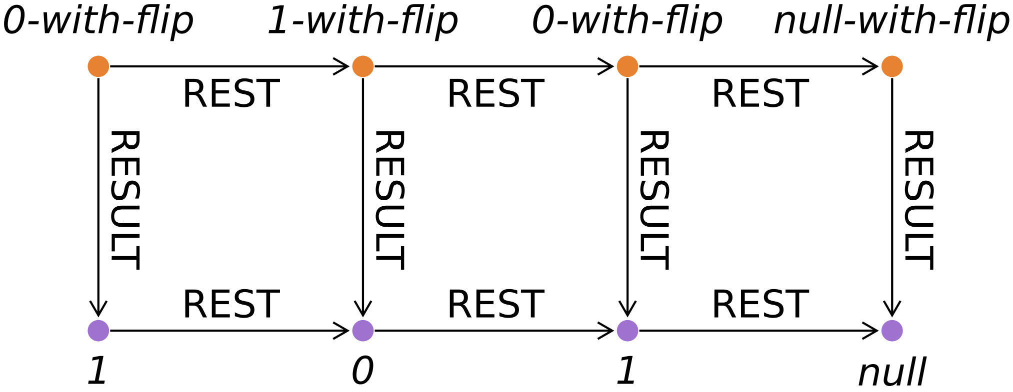
Unification can create paths



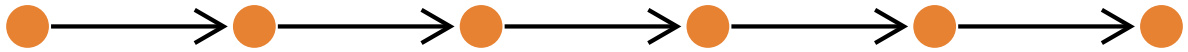
Unification can create paths



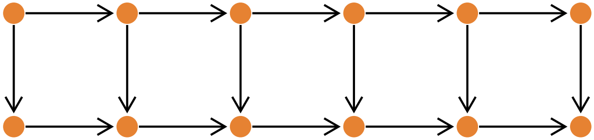
Unification can create paths



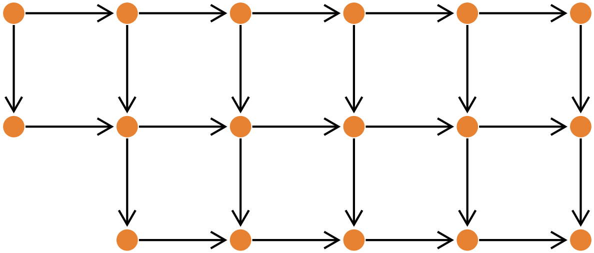
Doubly recursive computation



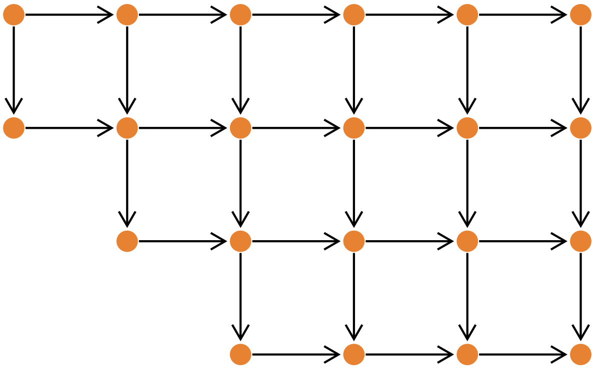
Doubly recursive computation



Doubly recursive computation



Doubly recursive computation



Turing-complete computation

- Recursion with one feature: finite state machine
- Recursion with two features: Turing machine

Turing-complete computation

- Recursion with one feature: finite state machine
- Recursion with two features: Turing machine
  - One feature for the tape
  - One feature for transitions

TDL Turing Machine

$$\begin{bmatrix} \textit{turing-machine} & \\ \text{TAPE-LEFT} & \text{string} \\ \text{TAPE-RIGHT} & \text{string} \end{bmatrix}$$

TDL Turing Machine

$$\begin{bmatrix} \textit{turing-machine} & \\ \text{TAPE-LEFT} & \text{string} \\ \text{TAPE-RIGHT} & \text{string} \end{bmatrix} \quad \begin{bmatrix} \textit{turing-machine-with-final-state} & \\ \text{TAPE-LEFT} & \text{string} \\ \text{TAPE-RIGHT} & \text{string} \\ \text{NEXT-RESULT} & \text{turing-machine} \\ \text{FINAL-RESULT} & \text{turing-machine} \end{bmatrix}$$

TDL Turing Machine

- Need to define:
  - Push and pop operations for tape
  - Transitions (combination of state and symbol)
  - Propagation of final state

TDL Turing Machine

- Need to define:
  - Push and pop operations for tape
  - Transitions (combination of state and symbol)
  - Propagation of final state
- Turing-complete unification:
  - Data type (initial state and tape)
  - Computation type (with final state and tape)

Two-feature TDL Turing Machine

- Combine features:
  - Interleave left tape and right tape
(push and pop operations are a little fiddly...)
  - Use RESULT for both next state and final state
(need to split the machine into two nodes)

Demo time!

Grammar Engineering

- Make it easy to understand code

- Make it easy to invoke computation types

Grammar Engineering

- Make it easy to understand code
  - Clear naming conventions
- Make it easy to invoke computation types

Grammar Engineering

- Make it easy to understand code
  - Clear naming conventions
- Make it easy to invoke computation types
  - Wrapper types

Naming conventions

- ```
  with-computation := *top* &
    [ RESULT *top* ].
  ```
  ```

Naming conventions

- ```
  with-computation := *top* &
     [ RESULT *top* ].
  ```

- ```
  input-with-func := input & with-computation &
     [ RESULT output ].
  ```

Unary function from `input` to `output`

- Define new types, matching `input` hierarchy
- Inherit from `with-computation`
- Specify output

Unary function from `input` to `output`

- Define new types, matching `input` hierarchy
- Inherit from `with-computation`
- Specify output
- (Pay attention to recursion, if present!)

Binary functions: Currying

- Curry binary function into pair of unary functions

```
input1-with-f1 := input1 & with-computation &
  [ RESULT input2-with-f2 ].

input2-with-f2 := input2 & with-computation &
  [ RESULT output ].
```

Binary functions: Currying

- Curry binary function into pair of unary functions

```
input1-with-f1 := input1 & with-computation &
  [ RESULT input2-with-f2 ].

input2-with-f2 := input2 & with-computation &
  [ RESULT output ].
```

- Will end up defining a type for each combination of input types (i.e. Cartesian product)

Binary functions: Currying

- Example in grammar: boolean logic
- Currying done once, to map to `bool-pair`
- Operations (and, or) defined on `bool-pair`

Wrapper Types

- A computation type can be defined once, and used in many rules

Wrapper Types

- A computation type can be defined once, and used in many rules
- Wrapper types: a little boilerplate once, but more intuitive in each use

Wrapper Types

- $\begin{bmatrix} \textit{bool-with-neg} & \\ \text{RESULT} & \text{bool} \end{bmatrix} \begin{bmatrix} +\textit{-with-neg} & \\ \text{RESULT} & - \end{bmatrix} \begin{bmatrix} -\textit{-with-neg} & \\ \text{RESULT} & + \end{bmatrix}$

Wrapper Types

- $\begin{bmatrix} \textit{bool-with-neg} & \\ \text{RESULT} & \text{bool} \end{bmatrix} \begin{bmatrix} +\textit{-with-neg} & \\ \text{RESULT} & - \end{bmatrix} \begin{bmatrix} -\textit{-with-neg} & \\ \text{RESULT} & + \end{bmatrix}$

- $\begin{bmatrix} \textit{bool-wrapper} & \\ \text{BOOL} & \text{bool} \end{bmatrix}$

Wrapper Types

- $\begin{bmatrix} \textit{bool-with-neg} & \\ \text{RESULT} & \text{bool} \end{bmatrix} \begin{bmatrix} \textit{+-with-neg} & \\ \text{RESULT} & - \end{bmatrix} \begin{bmatrix} \textit{−-with-neg} & \\ \text{RESULT} & + \end{bmatrix}$

- $\begin{bmatrix} \textit{neg-bool} & \\ \text{BOOL} & \boxed{1}\ \text{bool} \\ \text{NEG} & \begin{bmatrix} \text{BOOL} & \begin{bmatrix} \textit{bool-with-neg} \\ \text{RESULT}\ \boxed{1} \end{bmatrix} \end{bmatrix} \end{bmatrix}$

Wrapper Types

- Define a wrapper type for each data type, with a unique feature (e.g. BOOL)
- Define a subtype for each computation type, with a unique feature (e.g. NEG)

Wrapper Types

- Define a wrapper type for each data type, with a unique feature (e.g. BOOL)
- Define a subtype for each computation type, with a unique feature (e.g. NEG)
- “Call” a type using its feature

Wrapper Types

```
...
FEAT.AND < #1, #2 >,
DTRS < [ FEAT #1 ], [ FEAT #2 ] >
...
```

Relational Constraints

- Any deterministic relational constraint can be expressed as a computation type

Relational Constraints

- Any deterministic relational constraint can be expressed as a computation type

- What about nondeterministic constraints? (e.g. popping an arbitrary element from a list)

Relational Constraints

- Any deterministic relational constraint can be expressed as a computation type
- What about nondeterministic constraints? (e.g. popping an arbitrary element from a list)
  - Delph-in unification is deterministic

Relational Constraints

- Any deterministic relational constraint can be expressed as a computation type
- What about nondeterministic constraints?
(e.g. popping an arbitrary element from a list)
  - Delph-in unification is deterministic
  - Parsing is nondeterministic
(multiple parses for the same input)

Nondeterministic Constraints

- Idea: one edge per output

Nondeterministic Constraints

- Idea: one edge per output
- Use unary rules to perform computation

Nondeterministic Constraints

- Idea: one edge per output
- Use unary rules to perform computation
- Need a feature to keep track of computation (fiddly recursion)
- Need to encapsulate from rest of grammar (fiddly protection for all other rules)

Example: head-comp

```
basic-head-1st-comp-phrase := basic-head-comp-phrase &
  [ SYNSEM.LOCAL.CAT.VAL.COMPS #comps,
    HEAD-DTR.SYNSEM.LOCAL.CAT.VAL.COMPS < #synsem . #comps >,
    NON-HEAD-DTR.SYNSEM #synsem ].

basic-head-2nd-comp-phrase := basic-head-comp-phrase &
  [ SYNSEM.LOCAL.CAT.VAL.COMPS < #firstcomp . #othercomps >,
    HEAD-DTR.SYNSEM.LOCAL.CAT.VAL.COMPS [ FIRST #firstcomp,
                                          REST < #synsem . #othercomps > ],
    NON-HEAD-DTR.SYNSEM #synsem ].
```

Example: head-comp

```
basic-head-any-comp-phrase := basic-head-comp-phrase &
  [ SYNSEM.LOCAL.CAT.VAL.COMPS #new-comps,
    HEAD-DTR.SYNSEM.LOCAL.CAT.VAL.COMPS #old-comps,
    NON-HEAD-DTR.SYNSEM #synsem,
    NONDETERMINISTIC [ POP-INPUT #old-comps,
                       POP-OUTPUT-LIST #new-comps,
                       POP-OUTPUT-ITEM #synsem ] ].
```

Summary

- Delph-in unification is Turing-complete
  (with two recursive features, lots of re-entrancies)
- Computation types allow relational constraints
- Wrapper types allow readable code
- Unary rules allow non-deterministic constraints