Type Inference

Relevance to Telescoping Languages

Arun Chauhan

Questions

- Why do we need type inference?
- Can we leverage the type inference work in the programming languages community?

Towards High-Level Systems for High-Performance Computing

Towards High-Level Systems for High-Performance Computing

- software engineering issues

Towards High-Level Systems for High-Performance Computing

- software engineering issues
  - bigger, more complicated, applications

Towards High-Level Systems for High-Performance Computing

- software engineering issues
  - bigger, more complicated, applications
  - fewer people to program in low-level languages

Towards High-Level Systems for High-Performance Computing

- software engineering issues
  - bigger, more complicated, applications
  - fewer people to program in low-level languages
- programmer productivity
  - shortage of programmers, in general
  - domain-specific libraries reduce effort

Towards High-Level Systems for High-Performance Computing

- software engineering issues
  - bigger, more complicated, applications
  - fewer people to program in low-level languages
- programmer productivity
  - shortage of programmers, in general
  - domain-specific libraries reduce effort
- several recent solutions
  - systems like POOMA, CCA, ROSE
  - languages like Matlab, S+

Telescoping Languages and Type Inference

- telescoping languages is a strategy for compiling high-level languages
- high-level languages are typically typeless or weakly typed
- type information is needed for efficient mapping onto hardware
- type information is needed for optimizations
  - users often implicitly intend multiple types
  - type information enables other optimizations

What is a Type?

What is a Type?

- the universe, V, is the set of all values

What is a Type?

- the universe, V, is the set of all values
- a subset, obeying certain properties, is an *ideal*

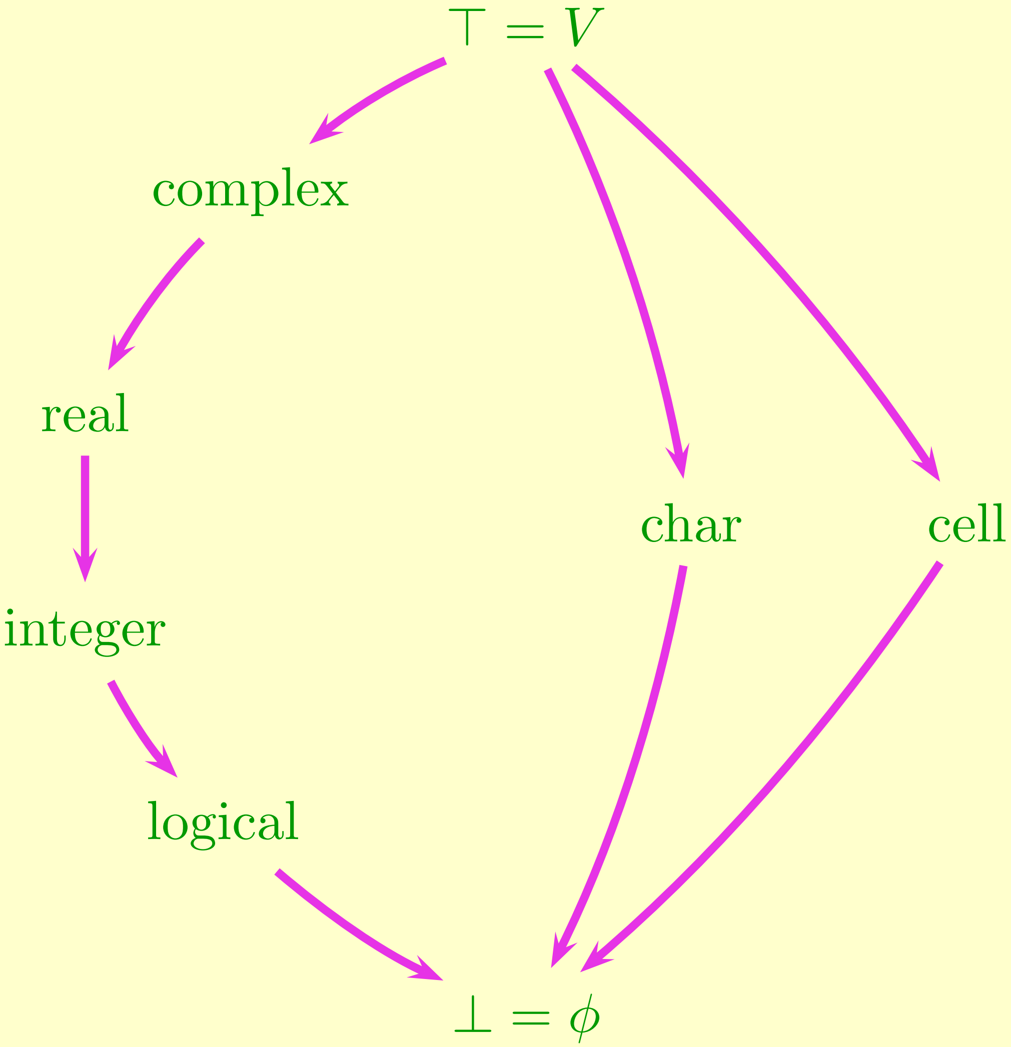
What is a Type?

- the universe, V, is the set of all values
- a subset, obeying certain properties, is an *ideal*
- a *type* is an *ideal*
    - there are other more complicated views of types as well

What is a Type?

- the universe, V, is the set of all values
- a subset, obeying certain properties, is an *ideal*
- a *type* is an *ideal*
  - there are other more complicated views of types as well
- the set of all types forms a lattice
  - $\top$ is the set of all values, V
  - $\perp$ is a singleton with the least element of V
  - elements are ordered by set inclusion, $\subset$

Example of a Simple System



$\top = V$

complex

real

integer

logical

char

cell

$\perp = \phi$

Defining Terms

having a type	membership in the appropriate set
type system	a small subset of all possible ideals
monomorphic type system	each value belongs to at most one type
polymorphic type system	some values may belong to more than one type
T_1 is a subtype of T_2	$T_1 \subseteq T_2$
untyped system	the type system consists of only one set, V

Defining Terms

having a type	membership in the appropriate set
type system	a small subset of all possible ideals
monomorphic type system	each value belongs to at most one type
polymorphic type system	some values may belong to more than one type
T_1 is a subtype of T_2	$T_1 \subseteq T_2$
untyped system	the type system consists of only one set, V

- language primitives allow constructing new types
  - function definitions create new function types
  - in an object-oriented language, class definitions create new data types

Basic Lambda Calculus

- akin to Turing Machine for programming languages

`e ::= x`	a variable is a λ expression
`e ::= λ(x) e`	functional abstraction of `e`
`e ::= e(e)`	operator `e` applied to operand `e`

Basic Lambda Calculus

- akin to Turing Machine for programming languages

`e ::= x`	a variable is a λ expression
`e ::= λ(x) e`	functional abstraction of `e`
`e ::= e(e)`	operator `e` applied to operand `e`

`id = λ(x) x`	identity function
`succ = λ(x) x+1`	successor function for integers

Typed λ-Calculus

```
succ = λ(x:Int) x+1
```

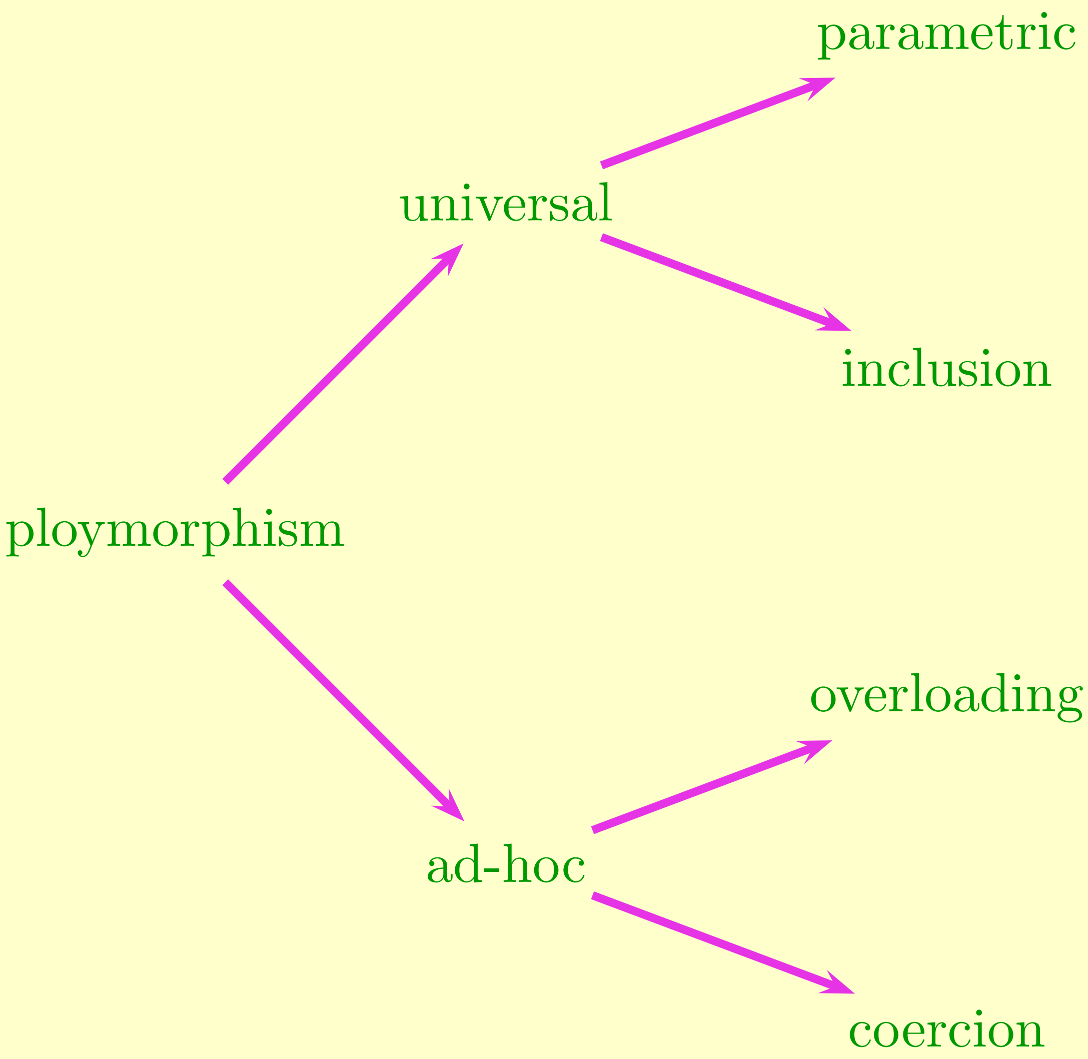
- the above definition has type $\text{Int} \rightarrow \text{Int}$
- this typed λ-calculus is sufficient to describe monomorphic type systems

Universal Quantification

```
∀a id = λ(x:a) x
```

- the above definition has type `a` → `a`
- universal quantification is needed to model polymorphic functions (or generic types)
- ML infers polymorphic function types (modeled by ∀)
- restricted universal quantification models **Hindley-Milner** type systems
- general universal quantification models **Girard-Reynolds** type systems

Types of Polymorphism



ploymorphism

universal

parametric

inclusion

ad-hoc

overloading

coercion

Existential Quantification

```
p:  ∃a.t(a)
```

- the above means that `p` has the type `t(a)` for some type `a`
- existential quantification enables modeling information hiding
  - e.g., private members of classes in object-oriented languages
- combining universal and existential quantification models parametric data abstraction

Bounded Quantification

$$\forall[\mathtt{a} \leq \mathtt{T}]\ \mathtt{e}$$

- the above means that `a` ranges over all subtypes of `T` in the scope of `e`
- this involves defining a $\leq$ relation among types, which models subtyping
- bounded quantification is necessary to model inheritance (inclusion polymorphism) adequately

Matlab Types for Tel. Languages

type $= (\tau, \rho, \sigma, \psi) =$ <*intrinsic type, rank, size, shape*>

- array size needed to eliminate dynamic resizing
- intrinsic type needed to minimize computation requirement
- shape useful in optimization
- all type information can be used to specialize procedures

Matlab Type Inference

Telescoping Languages Framework

- the Matlab type system just defined needs bounded quantification to be modeled
  - a procedure can always accept a larger array, thus, has inclusion polymorphism
- this makes type inference in telescoping languages context very hard
- even for straight line code, the problem is $\mathcal{NP}$-hard
- need to infer all possible valid types to trigger specialization

Matlab Type Inference

McCosh's propositional-logic approach

- static technique employing procedure-level annotations
- clique-based solution efficient under certain assumptions
- finds all valid type configurations

Matlab Type Inference

McCosh's propositional-logic approach

- static technique employing procedure-level annotations
- clique-based solution efficient under certain assumptions
- finds all valid type configurations
- imprecise for certain cases
  - does not handle data-dependent types precisely
  - type information not transferred across SSA ϕ-functions
  - needs extra support for dynamic inference

Set-based Type Inference

Cormac Flanagan, PhD, Rice 1997

- types are explicitly represented as sets of values
- a *specification* phase builds constraints on the sets of values for each expression in the program
- a *solution* phase solves the set constraints to compute the least solution
- implemented for Scheme, and subsequently for Java (MrSpidey)

Set-based Type Inference

Cormac Flanagan, PhD, Rice 1997

- types are explicitly represented as sets of values
- a *specification* phase builds constraints on the sets of values for each expression in the program
- a *solution* phase solves the set constraints to compute the least solution
- implemented for Scheme, and subsequently for Java (MrSpidey)
- cannot handle overloaded operators for type inference

Dependent Types for Array Sizes

Hongwei Xi, Frank Pfenning, PLDI 1998

- dependent types defined in terms of an index
  - e.g., a type can be defined as `int(2)`
- well-typed language (ML) and some annotations
- carry out the standard ML type inference
- then build constraints from indexed expressions
- constraints simplified to linear inqequalities to solve

Dependent Types for Array Sizes

Hongwei Xi, Frank Pfenning, PLDI 1998

- dependent types defined in terms of an index
  - e.g., a type can be defined as `int(2)`
- well-typed language (ML) and some annotations
- carry out the standard ML type inference
- then build constraints from indexed expressions
- constraints simplified to linear inqeualities to solve
- works in a limited context

Type Inference for Matlab

Luiz DeRose, PhD, UIUC 1995

- based on traditional standard dataflow techniques
- type inference mapped to a flow independent framework
- iterative solver used to arrive at a fixed point

Type Inference for Matlab

Luiz DeRose, PhD, UIUC 1995

- based on traditional standard dataflow techniques
- type inference mapped to a flow independent framework
- iterative solver used to arrive at a fixed point
- termination considerations affect the analysis
  - loops handled in an ad-hoc manner
  - backward flow of information limited to one step
- the approach is inadequate for inter-procedural analysis or recursion

Conclusion

- high-level programming systems rapidly becoming important for high-performance computing
- type inference necessary for effective compilation of high-level languages
- language theory provides useful understanding of issues related to type inference
- compiler writers must find engineering solutions for practical languages

References

1. Luca Cardelli, Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys* 17(4), 471–522, December 1985.

2. Luiz de Rose. *Compiler Techniques for MATLAB Programs*. PhD Thesis, University of Illinois at Urbana-Champaign, 1995.

3. Cormac Flanagan. *Effective Static Debugging via Componential Set-based Analysis*. PhD Thesis, Rice University, 1997.

4. Hongwei Xi, Frank Pfenning. Eliminating Array Bound Checking Through Dependent Types. In *Proceedings of the ACM SIGPLAN PLDI Conference*, pages 249–257, June 1998.

5. Cheryl McCosh. *Type-based Specialization in a Telescoping Compiler for Matlab*. MS Thesis, Rice University, 2002.

Extra Slides

Type Inference for Arrays

type $= (\tau, \rho, \sigma, \psi) =$ <*intrinsic type, rank, size, shape*>

```
function [A, F] = pisar (xt, sin_num)
 ...
 mcos = [];
 for n = 1:sin_num
   vcos = [];
   for i = 1:sin_num
     vcos = [vcos cos(n*w_est(i))];
   end
   mcos = [mcos; vcos]
 end
 ...
```

Type Inference for Arrays

type $= (\tau, \rho, \sigma, \psi) =$ *<intrinsic type, rank, size, shape>*

```
function [A, F] = pisar (xt, sin_num)
  ...
  mcos = [];
  for n = 1:sin_num
      vcos = [];
      for i = 1:sin_num
          vcos = [vcos cos(n*w_est(i))];
      end
      mcos = [mcos; vcos]
  end
  ...
```

size can grow around a loop

Another Way to Grow Arrays

```
A = zeros(1,N);
A(end+1) = x;
for i = 1:2*N
    A(i) = sqrt(i);
end
...
A(3, :) = [1:2*N];
...
A(:,:,2) = zeros(3, 2*N);
...
```

Example 1

```
A  = zeros(1, N);

y  = ...
A(y) = ...

x  = ...
A(x) = ...
```

Example 1

```
A  = zeros(1, N);
```

$\sigma^A = \langle N \rangle$

```
y  = ...
A(y) = ...
```

$\sigma^A = \mathtt{max}(\sigma^A, \langle y \rangle)$

```
x  = ...
A(x) = ...
```

$\sigma^A = \mathtt{max}(\sigma^A, \langle x \rangle)$

Example 1

$\texttt{A}_1 \texttt{ = zeros(1, N);}$

$\sigma_1^A \texttt{ = } <N>$

$\texttt{y}_1 \texttt{ = ...}$

$\texttt{A}_1(\texttt{y}_1) \texttt{ = ...}$

$\sigma_2^A \texttt{ = } \texttt{max}(\sigma_1^A, <y_1>)$

$\texttt{x}_1 \texttt{ = ...}$

$\texttt{A}_1(\texttt{x}_1) \texttt{ = ...}$

$\sigma_3^A \texttt{ = } \texttt{max}(\sigma_2^A, <x_1>)$

Example 1

$\mathtt{A}_1 = \mathtt{zeros(1,\ N);}$

$\Rightarrow \sigma^A_1 = {<}N{>}$

$\Rightarrow \mathtt{y}_1 = \mathtt{...}$

$\mathtt{A}_1(\mathtt{y}_1) = \mathtt{...}$

$\Rightarrow \sigma^A_2 = \mathtt{max}(\sigma^A_1, {<}y_1{>})$

$\Rightarrow \mathtt{x}_1 = \mathtt{...}$

$\mathtt{A}_1(\mathtt{x}_1) = \mathtt{...}$

$\Rightarrow \sigma^A_3 = \mathtt{max}(\sigma^A_2, {<}x_1{>})$

Example 1

```
⇒$\sigma_1^A$ = $<N>$
⇒$y_1$ = ...
⇒$\sigma_2^A$ = max$(\sigma_1^A, <y_1>)$
⇒$x_1$ = ...
⇒$\sigma_3^A$ = max$(\sigma_2^A, <x_1>)$
  allocate(A, $\sigma_3^A$);
  $A_1$ = zeros(1, N);
  $A_1$($y_1$) = ...
  $A_1$($x_1$) = ...
```

Slice Hoisting

- insert σ statements
- do SSA conversion
- identify the slice involved in computing the σ values
- *hoist* the slice before the first use of the array

Example 2

```
y  = ...
A (y ) = ...

c  = ...
if (c )
   ...
   B  = [ ... ];
   x  = min(B );
else
   ...
   x  = 10;
end

A (x ) = ...
```

Example 2

```
y  = ...
A (y ) = ...
$\sigma^A$ = $<y>$
c  = ...
if (c )
   ...
   B  = [ ... ];
   x  = min(B );
else
   ...
   x  = 10;
end

A (x ) = ...
$\sigma^A$ = max($\sigma^A$, $<x>$)
```

- insert σ functions

Example 2

```
$y_1$ = ...
$A_1(y_1)$ = ...
$\sigma_1^A = <y_1>$
$c_1$ = ...
if ($c_1$)
    ...
    $B_1$ = [ ... ];
    $x_1$ = min($B_1$);
else
    ...
    $x_2$ = 10;
end
$x_3 = \phi(x_1, x_2)$
$A_1(x_3)$ = ...
$\sigma_2^A = \mathtt{max}(\sigma_1^A, <x_3>)$
```

- insert σ functions
- do SSA

Example 2

```
⇒$y_1$ = ...
 $A_1(y_1)$ = ...
⇒$\sigma_1^A$ = $<y_1>$
⇒$c_1$ = ...
⇒if ($c_1$)
     ...
⇒    $B_1$ = [ ... ];
⇒    $x_1$ = min($B_1$);
⇒else
     ...
⇒    $x_2$ = 10;
⇒end
⇒$x_3$ = $\phi(x_1, x_2)$
 $A_1(x_3)$ = ...
⇒$\sigma_2^A$ = $\texttt{max}(\sigma_1^A, <x_3>)$
```

- insert σ functions
- do SSA
- identify slice

Example 2

```
⇒$y_1$ = ...
⇒$c_1$ = ...
⇒if ($c_1$)
⇒   $B_1$ = [ ... ];
⇒   $x_1$ = min($B_1$);
⇒else
⇒   $x_2$ = 10;
⇒end
⇒$x_3$ = $\phi$($x_1$, $x_2$)
⇒$\sigma_1^A$ = $<y_1>$
⇒$\sigma_2^A$ = max($\sigma_1^A$, $<x_3>$)
 allocate(A, $\sigma_3^A$);
 $A_1$($y_1$) = ...
 if ($c_1$)
    ...
 else
    ...
 end
 $A_1$($x_3$) = ...
```

- insert σ functions
- do SSA
- identify slice
- hoist slice

Example 3

```
A (x ) = ...

for i  = 1:N
   ...


   A  = [A  f(i )];

end
```

Example 3

```
A (x ) = ...
$\sigma^A$ = $<x>$
for i  = 1:N
    ...



    A  = [A  f(i )];
    $\sigma^A$ = $\sigma^A + <1>$
end
```

- insert σ functions

Example 3

```
$A_1$($x_1$) = ...
$\sigma_1^A$ = $<x_1>$
for $i_1$ = 1:N
    ...
    $A_2$ = $\phi$($A_1$, $A_3$)
    $\sigma_2^A$ = $\phi$($\sigma_1^A$, $\sigma_3^A$)
    $A_3$ = [$A_2$ f($i_1$)];
    $\sigma_3^A$ = $\sigma_2^A + <1>$
end
```

- insert σ functions
- do SSA

Example 3

```
  $A_1(x_1)$ = ...
⇒$\sigma_1^A$ = $<x_1>$
⇒for $i_1$ = 1:N
      ...
      $A_2$ = $\phi(A_1, A_3)$
⇒     $\sigma_2^A$ = $\phi(\sigma_1^A, \sigma_3^A)$
      $A_3$ = [$A_2$ f($i_1$)];
⇒     $\sigma_3^A$ = $\sigma_2^A + <1>$
⇒end
```

- insert σ functions
- do SSA
- identify slice

Example 3

```
⇒$\sigma_1^A$ = $<x_1>$
⇒for i$_1$ = 1:N
⇒    $\sigma_2^A$ = $\phi(\sigma_1^A$, $\sigma_3^A)$
⇒    $\sigma_3^A$ = $\sigma_2^A + <1>$
⇒end
  allocate(A, $\sigma_3^A$);
  A$_1$(x$_1$) = ...
  for i$_1$ = 1:N
      ...
      A$_2$ = $\phi$(A$_1$, A$_3$)
      A$_3$ = [A$_2$ f(i$_1$)];
  end
```

- insert σ functions
- do SSA
- identify slice
- hoist the slice

Advantages of the Approach

- very simple and fast
  - needs only SSA analysis and linear time
- it can leverage more advanced analyses, if available
  - symbolic analysis
  - dependence analysis
- subsumes inspector-executor style
- benefits from the telescoping languages framework
  - procedure specialization
  - procedure strength reduction
- handles most common cases

Dependences Can Raise Roadblocks

```
A(1) = ...

...
x = f(A)
A(x) = ...

...
```

Dependences Can Raise Roadblocks

`A(1) = ...`

σ^A = $<1>$

`...`

`x = f(A)`

`A(x) = ...`

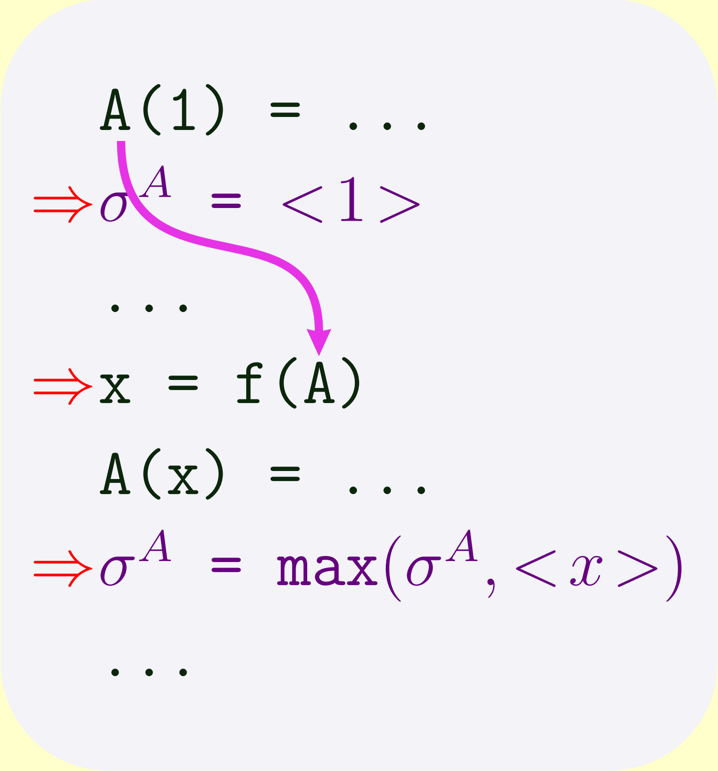
σ^A = $\texttt{max}(\sigma^A, <x>)$

`...`

Dependences Can Raise Roadblocks

```
  A(1) = ...
⇒$\sigma^A$ = $<1>$
  ...
⇒x = f(A)
  A(x) = ...
⇒$\sigma^A$ = max$(\sigma^A, <x>)$
  ...
```

Dependences Can Raise Roadblocks



`A(1) = ...`

$\Rightarrow \sigma^{A}$ = $<1>$

`...`

$\Rightarrow$ `x = f(A)`

`A(x) = ...`

$\Rightarrow \sigma^{A}$ = max$(\sigma^{A}, <x>)$

`...`

dependence blocks hoisting