Chapter 6: Graph Reduction, Interpretation, Abstract Interpretation and Type Inference

Learning targets of this chapter:

1. Compilation paradigm: graph reduction
2. Interpretation: syntax, denotational semantics, environment
3. Abstract interpretation with examples
4. Strictness analysis and its uses
5. Type inference in the Hindley-Milner type system

*Source reference: Grune et al.: *Modern Compiler Design*, Wiley, 2000, Chap. 7

Graph Reduction

- Idea: reduction of a call graph to normal form
- Challenge: choice of the subgraph to be substituted (*redex*)
- Replacement strategies: (weak-head normal form)

  normal order: function application before parameter evaluation (lazy)

  applicative order: parameter evaluation before function application (eager)
- Two examples:

```
let f x = 0          let const c x = c
    g x = g x
in  f g              in  const 1 (1/0)
```

normal order:	returns 0	returns 1
applicative order:	non-termination	error message

Church-Rosser Theorems

- All reduction orders reach the same normal form (if they terminate).
- The strategy of *normal order* yields maximal termination.
- Observation:

  with the use of higher-order functions unary functions suffice (*currying*).

Normal-Order Reduction Strategy of Haskell

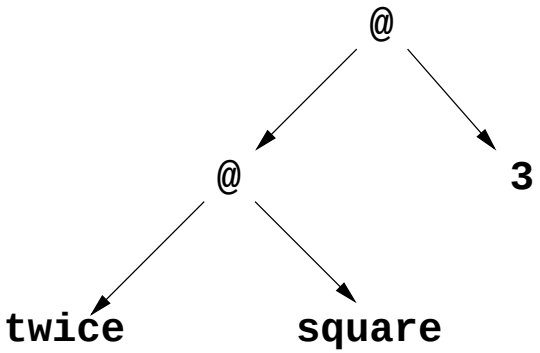
Idea: look for the next essential redex (*unwinding*)

- Start with the root of the entire call tree.
- If it is not an application node, return it and terminate.
- Otherwise proceed to the function to be called (the left leaf)
- If the number of application nodes traversed on the way does not match the currying degree of the function, return the graph as is.
- Otherwise check whether the function is user-defined.
- If so, the root node of the graph is the next redex.
- Certain predefined functions can only be reduced if their arguments are values (strict evaluation).
- In this case, the root node of a strict argument not yet evaluated is the next redex.

Graph Reduction: Example

```
let twice f x = f (f x)

    square n  = n * n

in  twice square 3
```

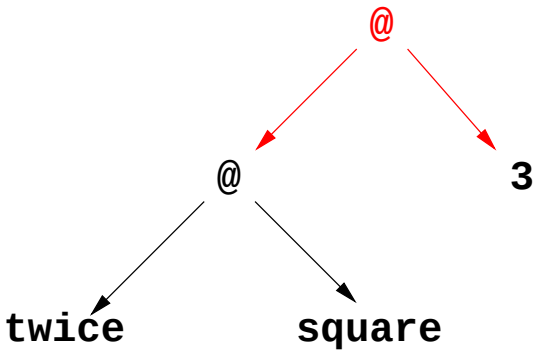
source tree



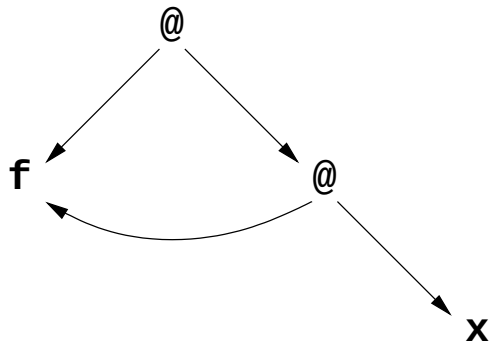
Graph Reduction: Example

```
let twice f x = f (f x)

    square n  = n * n

in  twice square 3
```

first redex: call of `twice`



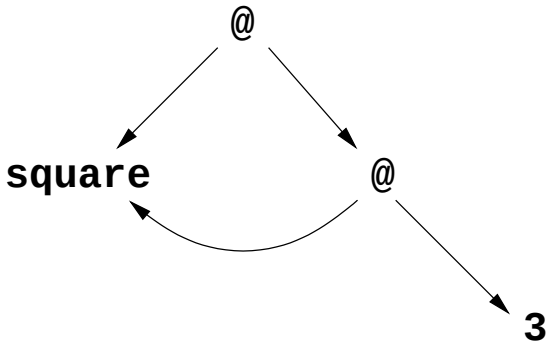
`twice f x = f (f x)`



Graph Reduction: Example

```
let twice f x = f (f x)

    square n  = n * n

in  twice square 3
```

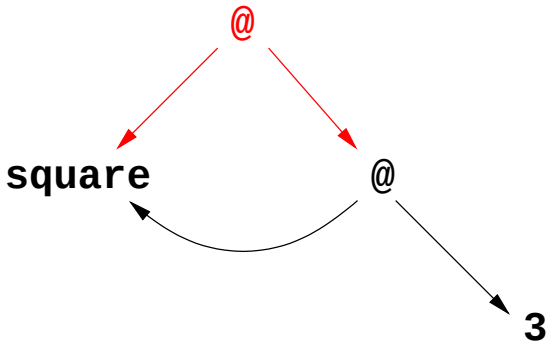
first step: expansion of the call of `twice`



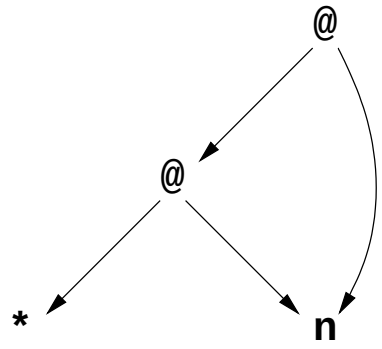
Graph Reduction: Example

```
let twice f x = f (f x)

    square n  = n * n

in  twice square 3
```

second redex: call of `square`



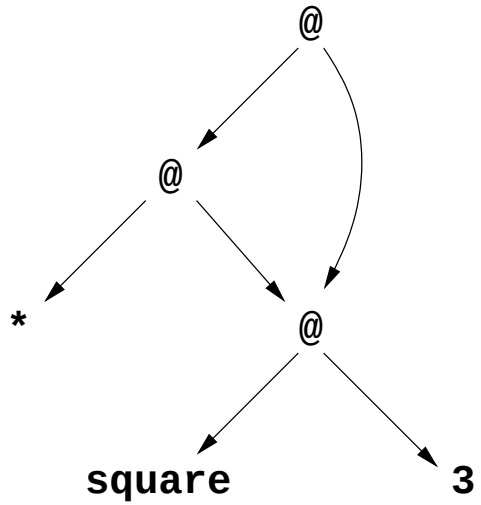
`square n = n * n`



Graph Reduction: Example

```
let twice f x = f (f x)

    square n  = n * n

in  twice square 3
```

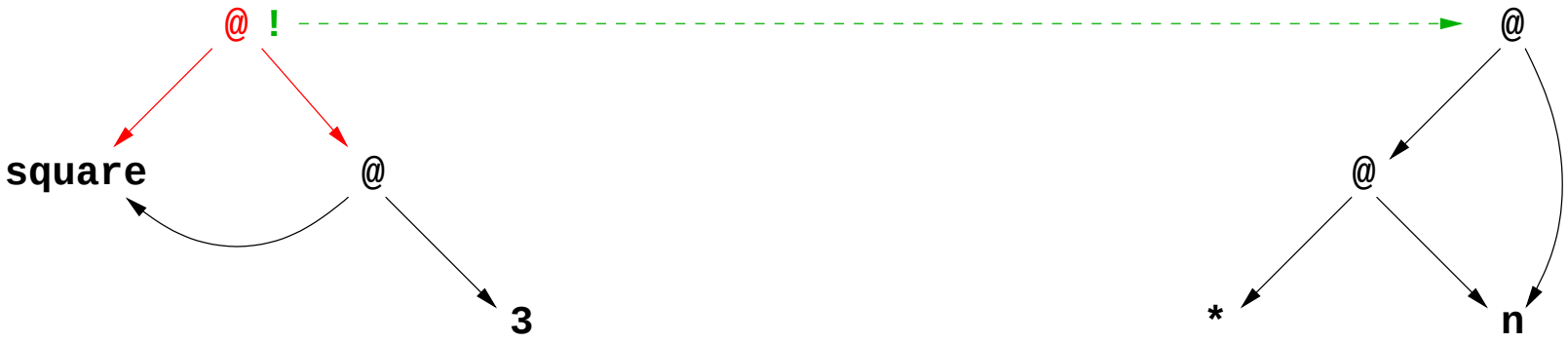
second step: expansion of the call of `square`



Graph Reduction: Example

```
let twice f x = f (f x)

    square n  = n * n

in  twice square 3
```

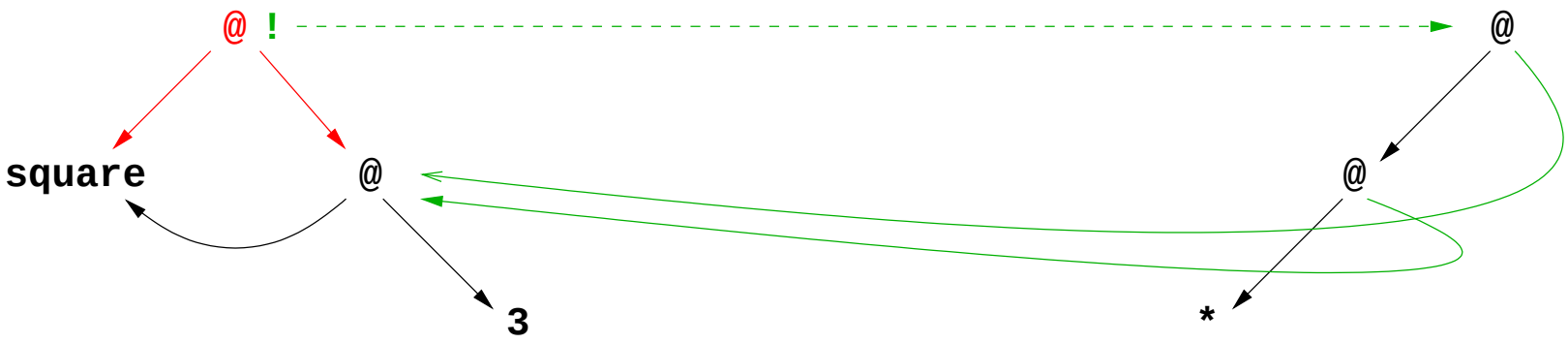
second step in computational steps: link of both trees



Graph Reduction: Example

```
let twice f x = f (f x)

    square n  = n * n

in  twice square 3
```

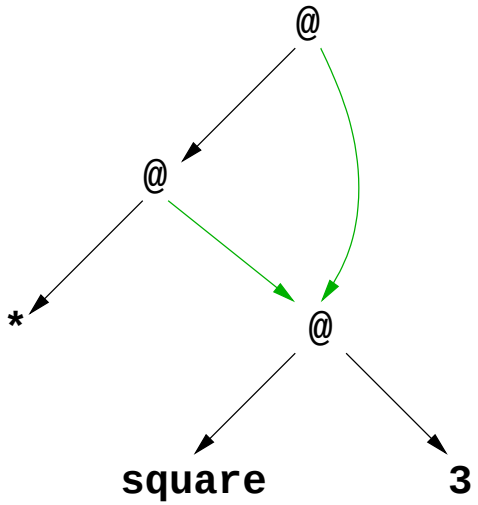
second step in computational steps: link of both arguments



Graph Reduction: Example

```
let twice f x = f (f x)

    square n  = n * n

in  twice square 3
```

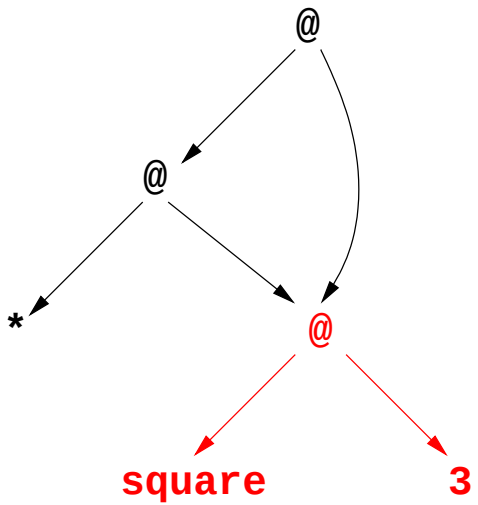
second step in computational steps: merge of both trees



Graph Reduction: Example

```
let twice f x = f (f x)

    square n  = n * n

in  twice square 3
```

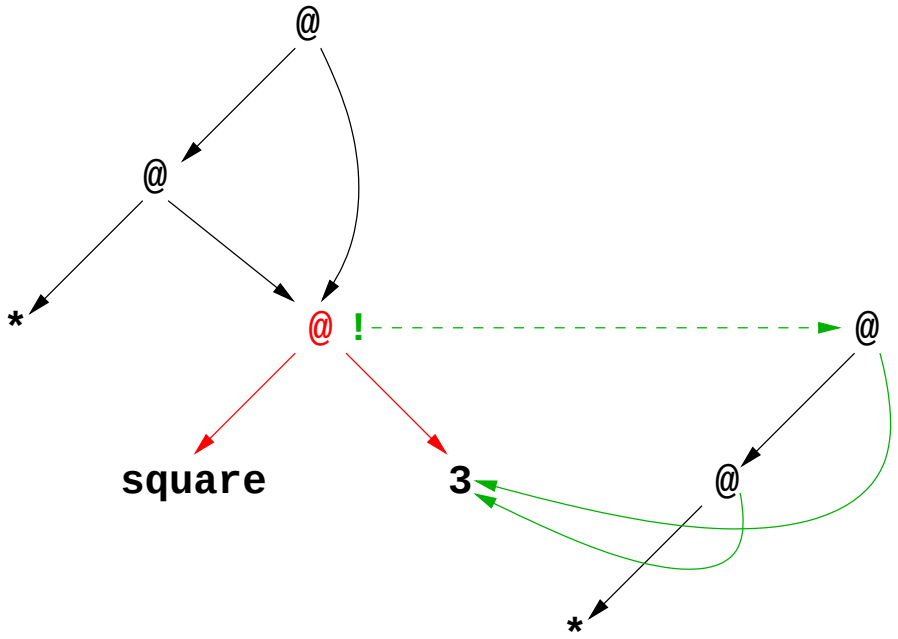
third redex: remaining call of `square`



Graph Reduction: Example

```
let twice f x = f (f x)

    square n  = n * n

in  twice square 3
```

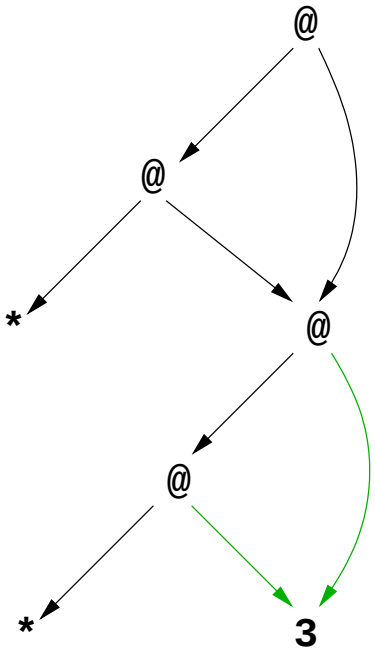
third step: linking of the arguments



Graph Reduction: Example

```
let twice f x = f (f x)

    square n  = n * n

in  twice square 3
```

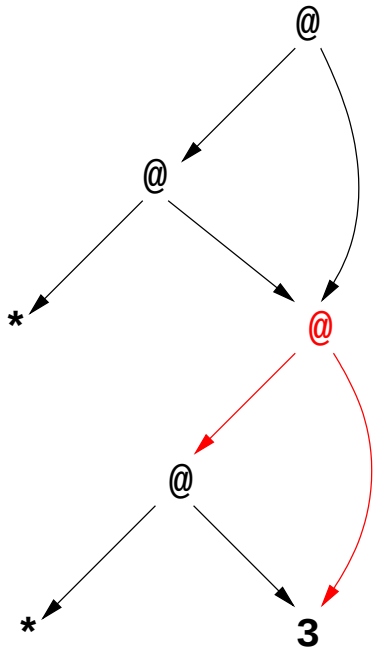
third step: merge of both trees



Graph Reduction: Example

```
let twice f x = f (f x)

    square n  = n * n

in  twice square 3
```

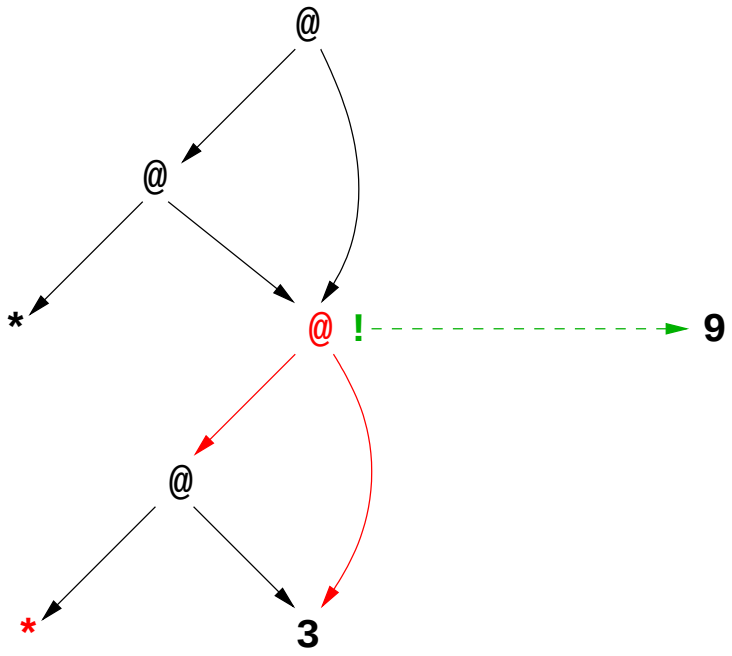
fourth redex: inner multiplication



Graph Reduction: Example

```
let twice f x = f (f x)

    square n  = n * n

in  twice square 3
```

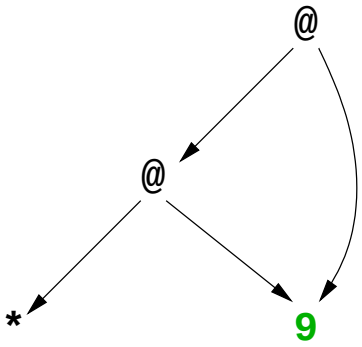
fourth step: link to the result



Graph Reduction: Example

```
let twice f x = f (f x)

    square n  = n * n

in  twice square 3
```

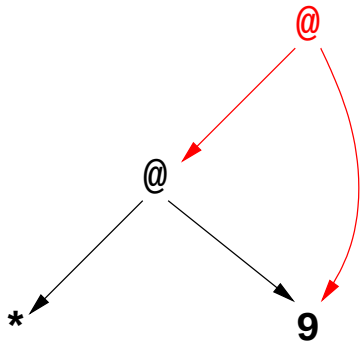
fourth step: substitution of the result



Graph Reduction: Example

```
let twice f x = f (f x)

    square n  = n * n

in  twice square 3
```

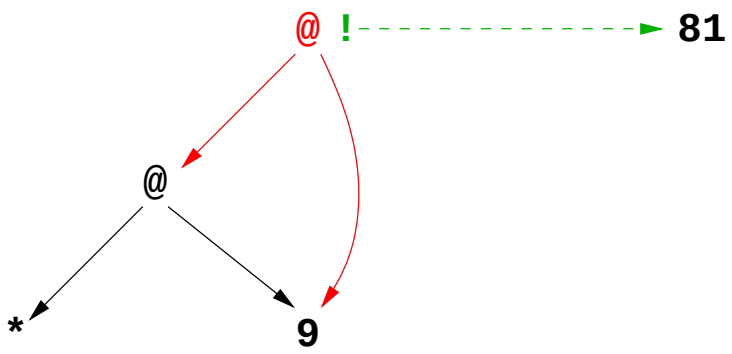
fifth redex: remaining multiplication



Graph Reduction: Example

```
let twice f x = f (f x)

    square n  = n * n

in  twice square 3
```

fifth step: link to the result



Graph Reduction: Example

```
let twice f x = f (f x)

    square n  = n * n

in  twice square 3
```

fifth step: substitution of the result

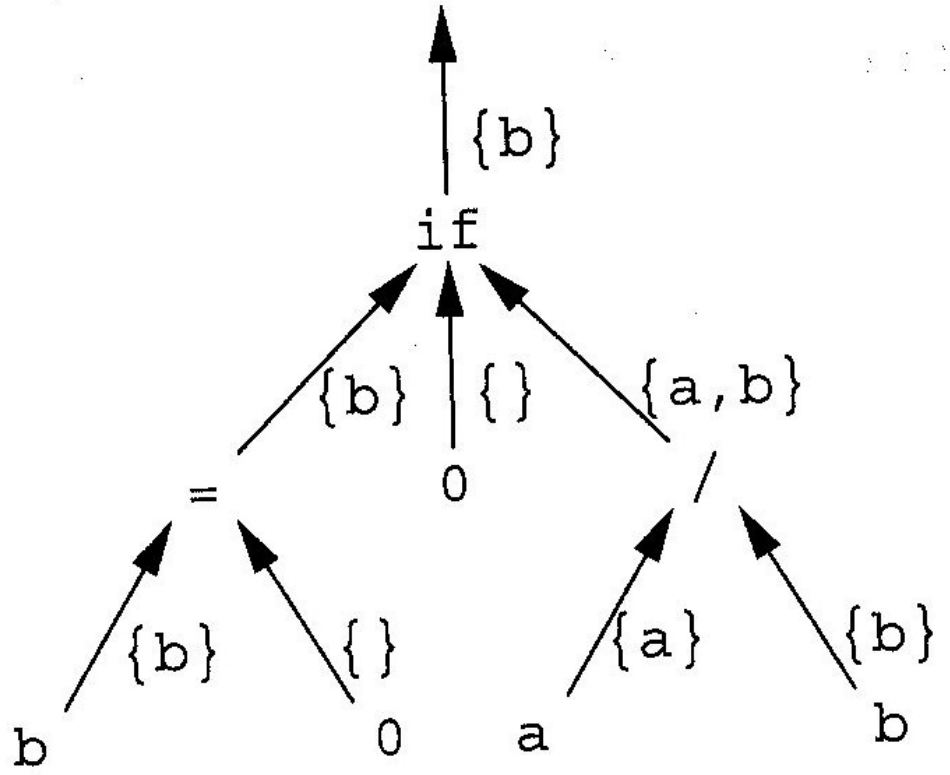
81

Strictness Analysis

- Problem:
  - for all user-defined functions, determine the arguments that will be evaluated immediately in a call
  - generate target code for the evaluation of these parameters
  - avoid the construction and reduction of the respective call graph this way
- Idea:
  - propagate strictness info from the leaves of the abstract syntax tree (of the primitive functions) towards the root
- Source reference:
  - Grune, Bal, Jacobs, Langendoen: *Modern Compiler Design*, Wiley & Sons, 2000, Kap. 7.7.1

Example: "Safe" Division

```
safe_div a b = if b==0 then 0 else a/b
```



Edge label: set of arguments that must be strictly evaluated in the computation of the expression at the source

Strictness Rules

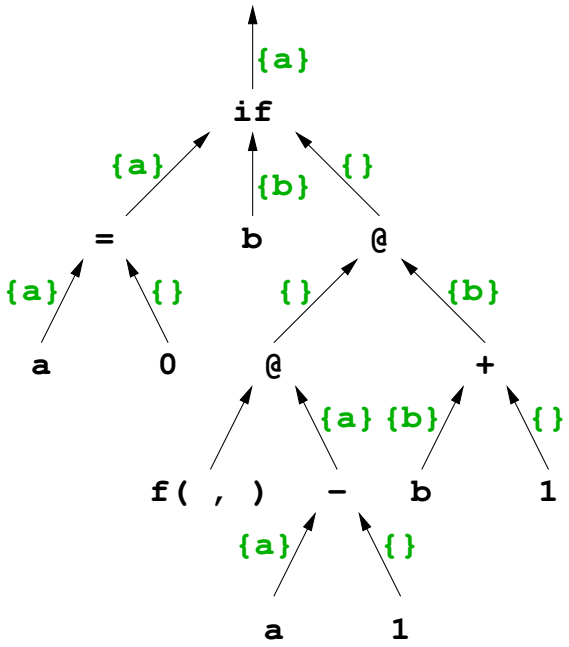
- Arithmetic operators: all operands are strict
- Alternation:
  - strictness information of the condition
  - Strictness conditions that apply to *both* alternatives
- Function calls:
  - function with a global name:
    - available strictness information on the arguments
    - collect all strict arguments
  - non-globally declared function:
    - in general, no strictness information on the arguments available
    - propagate just the information on the function expression
- Local variables: `let` v `=` V `in` E
  - if v does not occur in E, propagate just the information on E
  - otherwise replace the old strictness information on v with that on V

Strictness Analysis of Recursive Functions

- Example 1: `f a b = if a==0 then b else f (a-1) (b+1)`
- Strictness analysis:

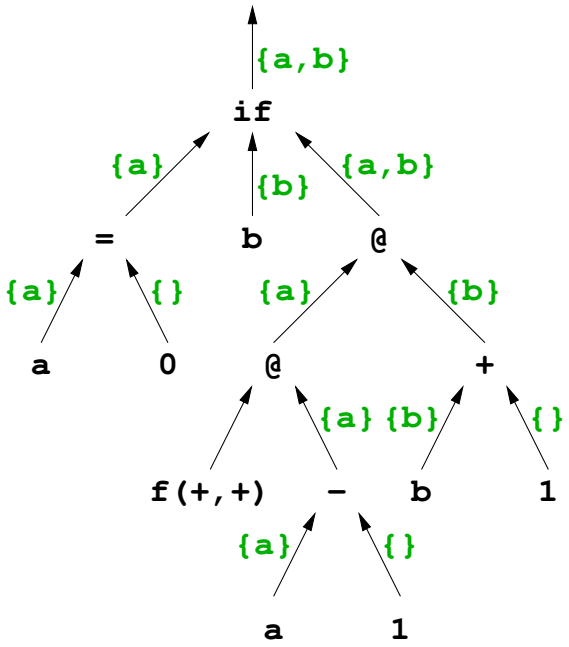
conservative assumption:

arguments not strict



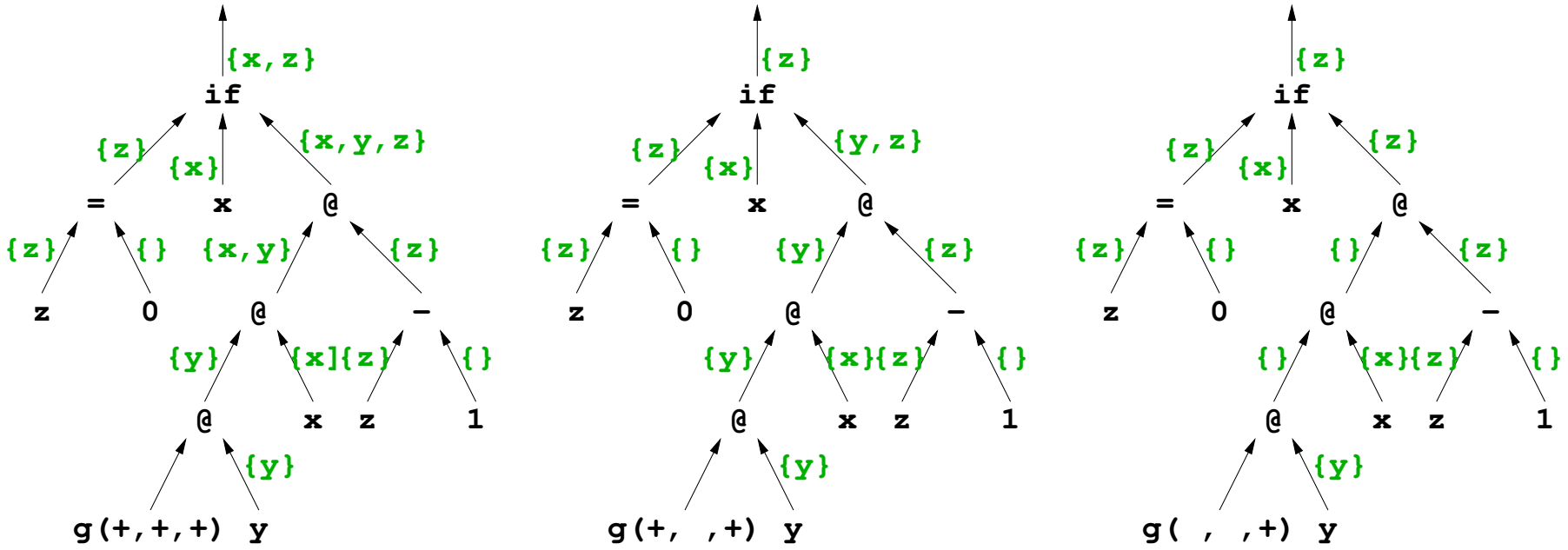
optimistic assumption:

arguments strict



Strictness Analysis of Recursive Functions

- Example 2: `g x y z = if z==0 then x else g y x (z-1)`
- Strictness analysis:
  - first optimistic analysis here too optimistic
  - successive analysis steps restrict the result
  - incorrect analysis for non-terminating functions (`f x = f x`)



Interpretation

- Interpreter :: syntax → semantics
  - syntax: data structure (syntax tree)
  - semantics: Haskell function: input → output
- Abstract interpretation
  - reduced semantics: abstract input → abstract output
    - integer coset arithmetic
    - strictness analysis
- Type inference: can be viewed as interpretation

Interpreter

1. Defines the denotational semantics of a syntactic expression
2. Transforms input data to output data

Recommended structure: inductively defined semantics function

- following the inductive structure of the syntax tree
- based on the semantics of the subexpressions

Treatment of bound variables

- β-reduction: (syntactic) substitution
- semantics function: deposit of variables (their values) in an environment

Semantics Function of the Simply Typed λ-Calculus

Abstract Syntax:

```
data LExp = V String              -- variable
          | LExp :@: LExp         -- application
          | L String LExp         -- abstraction
```

Semantics function $\mathcal{S}$ (ϵ environment):

1. $\mathcal{S}\,[\![\texttt{V}\ x]\!]\,\epsilon = \epsilon\,[\![x]\!]$
2. $\mathcal{S}\,[\![f\ \texttt{:@:}\ x]\!]\,\epsilon = (\mathcal{S}\,[\![f]\!]\,\epsilon)\ (\mathcal{S}\,[\![x]\!]\,\epsilon)$
3. $\mathcal{S}\,[\![\texttt{L}\ a\ v]\!]\,\epsilon = \lambda x\,.\,(\mathcal{S}\,[\![v]\!]\,\epsilon')$,
   where x new name, $\epsilon'\,[\![a]\!] = x$ and $\epsilon'\,[\![y]\!] = \epsilon\,[\![y]\!]$ if $y \neq a$.

Implementation in Haskell : Auxiliary Functions

```
newtype Env a = Env { content::[(String,a)] }    -- def. of environment

value :: Env a -> String -> a                    -- look up a variable's value
value env key = case lookup key (content env) of
                  Nothing -> error ("unknown value of: " ++ key)
                  Just x  -> x

instance Show a => Show (Value a) where                  -- display a value
   show (Ground x) = show x
   show (Funct x)  = "<function>"

addenv :: (String,a) -> Env a -> Env a   -- new entry (variable, value)
addenv  x (Env xs) = Env (x:xs)    -- possible occlusion of an old name
```

Implementation in Haskell: Semantics Function

```
data Value a = Ground a                              -- ground type
             | Funct (Value a -> Value a)            -- function type

sem :: LExp -> Env (Value a) -> Value a
sem (V x)     env = value env x                      -- extract a value
sem (f :@: x) env = let (Funct h) = sem f env        -- f must be a function
                        y         = sem x env
                    in h y                 -- here, y is still unevaluated!
sem (L a v)   env = Funct (\ x -> sem v (addenv (a,x) env))
```

```
Lambda> sem ((L "a" (V "f" :@: V "a")) :@: V "b")
            (Env [("b",Ground (5::Int)),
                  ("f",Funct (\(Ground x) -> Ground (x*x)))])
25
```

Abstract Interpretation

- Principle: evaluation using only abstract information
- Main goal: automatic program analysis
- Examples:
  - sign rule for numbers
  - coset arithmetic
  - strictness analysis
  - data dependence analysis
  - size analysis
  - type inference
- Challenge: analysis of recursive functions

  (treatment of fixed points)

Sign Rule for Multiplication

- Concrete structure: $\bar{K} = (\mathbb{Z}, *)$, abstract structure: $\bar{A} = (\{\boxdot, \boxminus, \boxplus\}, \circledast)$
- Abstraction function: $a : \mathbb{Z} \to \{\boxdot, \boxminus, \boxplus\}$

$$a\ x = \begin{cases} \boxminus & , \text{ if } x < 0 \\ \boxdot & , \text{ if } x = 0 \\ \boxplus & , \text{ if } x > 0 \end{cases}$$

- Abstraction of $*$: $\circledast$ with

$\circledast$	$\boxdot$	$\boxminus$	$\boxplus$
$\boxdot$	$\boxdot$	$\boxdot$	$\boxdot$
$\boxminus$	$\boxdot$	$\boxplus$	$\boxminus$
$\boxplus$	$\boxdot$	$\boxminus$	$\boxplus$

- Thus, we have: $a\ (x * y) = a\ x\ \circledast\ a\ y$

Cosets as Abstract Information

```
data Rem10007 = R10007 Int  deriving (Eq,Show)

instance Num Rem10007 where
   R10007 x + R10007 y = R10007 ((x+y) `mod` 10007)
   R10007 x - R10007 y = R10007 ((x-y) `mod` 10007)
   R10007 x * R10007 y = R10007 ((x*y) `mod` 10007)
   fromInteger x = R10007 (fromInteger (x `mod` 10007))

evalPoly :: Num a => [a] -> a -> a
evalPoly cs x = foldl (\ v c -> x*v+c) 0 cs
```

```
   *Remainder> evalPoly [1..100000] 2 `mod` 10007
   1006
   (42.57 secs, 1593189860 bytes)
   *Remainder> evalPoly (map R10007 [1..100000]) (R10007 2)
   R10007 1006
   (1.77 secs, 25315500 bytes)
```

Type Inference as Abstract Interpretation

- Type information of the application

$$\frac{f :: \alpha \to \beta \mid x :: \alpha'}{f\ x\ :: \beta'}$$

where $\beta' = \sigma(\beta)$, $\sigma = \text{mgu}(\alpha, \alpha')$, *most general unifier* (substitution)

Ex.: $f :: \gamma \to \delta \to \gamma$, $x :: (Int \to Int)$,

$\sigma = [\gamma := Int \to Int]$, $f\ x :: (Int \to Int) \to \delta \to (Int \to Int)$

- The *most general unifier* is the most general specialization σ for which:

$$\sigma(\alpha) = \sigma(\alpha')$$

Unification

Unification is a generalization of assignment. Assignment solves a defining equation by evaluating the right-hand side and assigning it to the left. The left-hand side is syntactically restricted to one variable (or, in a *multiple assignment*, to several variables).

In unification, both sides of the equation can be arbitrary expressions. Unification identifies the most general specialization that makes both sides equal. In the subsequent computation, the values of variables in the expressions are specialized accordingly.

Unification does not necessarily lead to a full evaluation of the expressions. In the specialization, variables are allowed to remain unevaluated.

Unification: One More Example

Two unifiers of the following two expressions:

$$((\alpha_1 \to \alpha_2), [\alpha_3]) \to [\alpha_2]$$
$$((\alpha_3 \to \alpha_4), [\alpha_3]) \to \alpha_5$$

are:

x	$S(x)$	$S'(x)$
α_1	α_3	α_1
α_2	α_2	α_1
α_3	α_3	α_1
α_4	α_2	α_1
α_5	$[\alpha_2]$	$[\alpha_1]$

S is the most general unifier.

Ex.: Type Inference of Function map (1)

```
map = Y (\ map' f xs
            -> if null xs
                  then []
                  else f (head xs) : map' f (tail xs)
        )
```

$$\frac{\texttt{xs} :: \tau_1 \;\mid\; \texttt{null} :: [\tau_2] \to \texttt{Bool}}{\texttt{null xs} :: \texttt{Bool}} \; \{\tau_1 = [\tau_2]\}$$

$$\frac{\texttt{xs} :: [\tau_2] \;\mid\; \texttt{head} :: [\tau_3] \to \tau_3}{\texttt{head xs} :: \tau_2} \; \{\tau_3 = \tau_2\}$$

$$\frac{\texttt{f} :: \tau_4 \;\mid\; \texttt{head xs} :: \tau_2}{\texttt{f (head xs)} :: \tau_5} \; \{\tau_4 = \tau_2 \to \tau_5\}$$

Ex.: Type Inference of Function `map` (2)

$$\frac{\texttt{xs} :: [\tau_2] \mid \texttt{tail} :: [\tau_6] \to [\tau_6]}{\texttt{tail xs} :: [\tau_2]} \{\tau_6 = \tau_2\}$$

$$\frac{\texttt{map'} :: \tau_7 \mid \texttt{f} :: \tau_2 \to \tau_5}{\texttt{map' f} :: \tau_8} \{\tau_7 = (\tau_2 \to \tau_5) \to \tau_8\}$$

$$\frac{\texttt{map' f} :: \tau_8 \mid \texttt{tail xs} :: [\tau_2]}{\texttt{map' f (tail xs)} :: \tau_9} \{\tau_8 = [\tau_2] \to \tau_9\}$$

$$\frac{\texttt{(:)} :: \tau_{10} \to [\tau_{10}] \to [\tau_{10}] \mid \texttt{f (head xs)} :: \tau_5}{\texttt{(:) (f (head xs))} :: [\tau_5] \to [\tau_5]} \{\tau_{10} = \tau_5\}$$

$$\frac{\texttt{(:)(f (head xs))} :: [\tau_5] \to [\tau_5] \mid \texttt{map' f (tail xs)} :: \tau_9}{\texttt{f (head xs) : map' f (tail xs)} :: [\tau_5]} \{\tau_9 = [\tau_5]\}$$

Ex.: Type Inference of Function `map` (3)

$$\frac{\begin{array}{l}\texttt{null xs} :: \texttt{Bool} \\ \mid \texttt{[]} :: [\tau_{11}] \\ \mid \texttt{f (head xs) : map' f (tail xs)} :: [\tau_5]\end{array}}{\begin{array}{l}\texttt{if null xs} \\ \quad \texttt{then []} \\ \quad \texttt{else f (head xs) : map' f (tail xs)} \\ :: [\tau_5]\end{array}} \{\tau_{11} = \tau_5\}$$

Ex.: Type Inference of Function `map` (4)

Shorthand: `map = Y (\ map' -> \ f xs ->` R`)`

$$\frac{R :: [\tau_5] \mid \texttt{xs} :: [\tau_2]}{(\backslash \texttt{xs} \texttt{->} R) :: [\tau_2] \to [\tau_5]}$$

$$\frac{\texttt{f} :: \tau_2 \to \tau_5 \mid (\backslash \texttt{xs} \texttt{->} R) :: [\tau_2] \to [\tau_5]}{(\backslash \texttt{f xs} \texttt{->} R) :: \sigma} \{\sigma = (\tau_2 \to \tau_5) \to [\tau_2] \to [\tau_5]\}$$

$$\frac{\texttt{map'} :: \tau_7 \mid (\backslash \texttt{f xs} \texttt{->} R) :: \sigma}{(\texttt{Y} (\backslash \texttt{map' -> } \backslash \texttt{f xs} \texttt{->} R)) :: \sigma} \{\tau_7 = \sigma\}$$

Result after generalization: $\boxed{\texttt{map} :: \forall\, \alpha\, \beta\, .\, (\alpha \to \beta) \to [\alpha] \to [\beta]}$

Generic Type Variables

- Not typable in the Hindley-Milner system:
  `(\f -> (f 3, f True)) id`
  - Reason: `f` :: $(\alpha \to \alpha)$, but α cannot be simultaneously of type `Integer` and `Bool`
  - Rank-1 polymorphism!
- Typable: `let f = id in (f 3, f True)`
  - Reason: `f` :: $\underline{\forall\alpha.}(\alpha \to \alpha)$
  - α is a *generic* type variable
- Not typable: `(\g -> let f = g in (f 3, f True)) id`
  - Reason: $\forall\alpha$ is resolved via λ-binding of `g`

Hindley-Milner Type System (1)

Variable $$A \uplus \{x :: \tau\} \vdash x :: \tau$$

Condition $$\frac{A \vdash c :: \texttt{Bool} \mid A \vdash e_1 :: \tau \mid A \vdash e_2 :: \tau}{A \vdash (\texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2) :: \tau}$$

Abstraction $$\frac{A \uplus (x :: \sigma) \vdash e :: \tau}{A \vdash (\backslash x \texttt{->} e) :: (\sigma \to \tau)}$$

Application $$\frac{A \vdash f :: (\sigma \to \tau) \mid A \vdash x :: \sigma}{A \vdash (f\ x) :: \tau}$$

Hindley-Milner Type System (2)

`let`-Expression

$$\frac{A \vdash y :: \sigma \mid A \uplus \{x :: \sigma\} \vdash e :: \tau}{A \vdash (\texttt{let}\ x\ \texttt{=}\ y\ \texttt{in}\ e) :: \tau}$$

Fixpoint

$$\frac{A \uplus \{x :: \tau\} \vdash e :: \tau}{A \vdash (\texttt{Y}\ (\backslash x\ \texttt{->}\ e)) :: \tau}$$

Generalization

$$\frac{A \vdash e :: \tau}{A \vdash e :: \forall \alpha.\tau}\ (\alpha \text{ not free in } A)$$

Specialization

$$\frac{A \vdash e :: \forall \alpha.\tau}{A \vdash e :: (\tau[\alpha := \sigma])}$$

Milner's $\mathcal{W}$-Algorithm [Milner, 1978]

- Assignment of a unique, most general type to every syntactically correct (ML-)expression, or report of a type error.
- The $\mathcal{W}$-algorithm is guaranteed to terminate; type inference is computable (in contrast to the type inference of certain Haskell extensions).
- Only *shallow types* are computed, i.e., no (universal) quantifiers in type expressions (rank-1 polymorphism).
- Solution of type equations via unification with the Robinson algorithm [J. Alan Robinson, 1965]; computation of the most general unifier (substitution).
- The unsolvability of $\alpha = \alpha \to \beta$ with finite terms is the reason for the failure of the type inference of $(x\ x)$ in $\mathcal{W}$ (the rules of the Hindley-Milner system would permit this).