# CS653: Functional Programming
# 2017-18 $II^{nd}$ Semester

# G Machine

Amey Karkare
karkare@cse.iitk.ac.in

http://www.cse.iitk.ac.in/~karkare/cs653
Department of CSE, IIT Kanpur

# Agenda

- Full Laziness
- G Machine

# Maximal Free Expressions

- **Free Subexpression:** A subexpression E of a $\lambda$ abstraction L is free in L if all variables in E are free in L.
- **Maximal Free Subexpression (MFE):** A MFE of L is a free expression which is not a *proper* subexpression of another free subexpression of L.

# $\lambda$-lifting using Maximal Free Expressions

Instead of $\lambda$-lifting a free variable, abstract out free subexpressions as parameters

```
f = g 4
g x y = y+(sqrt x)
(f 1) + (f 2)
```

```
$g1 sqrtx y = + y sqrtx
$g x = $g1 (sqrt x)
$f = $g 4
$Prog = + ($f 1) ($f 2)
```

When evaluating the expression, we get

```
   $Prog
→  + ( • 1) ( • 2)
                             ($g 4)
→  + ( • 1) ( • 2)
                             ($g1 (sqrt 4))
→  + ( • 1) (+ 2 • )
                             ($g1 (sqrt 4))
→  + ( • 1) (+ 2 • )
                             ($g1 2)
→  + ( • 1) 4
                             ($g1 2)
→  + (+ 1 • ) 4
                             2
→  + 3 4
→  7
```

# Fully Lazy $\lambda$-lifting and Letrecs

```
let
    f = λx.letrec fac = λn.(...)
           in x + (fac 1000)
in (f 3) + (f 4)
```

Without fully lazy $\lambda$-lifting:

```
$fac fac n = (...)
f x =letrec fac = $fac fac
     in + x (fac 1000)
$Prog = + (f 3) (f 4)
```

# Fully Lazy $\lambda$-lifting and Letrecs

'Float' free letrec-s outward

```
letrec
       fac = λn.(...)
in let
       f = λx.x + (fac 1000)
in (f 3) + (f 4)
```

Now, we can have fully lazy $\lambda$-lifting:

```
$fac n = (...)
$fac1000 = $fac 1000
$f x = + x $fac1000
$Prog = + ($f 3) ($f 4)
```

$\boxed{\text{G - Machine}}$

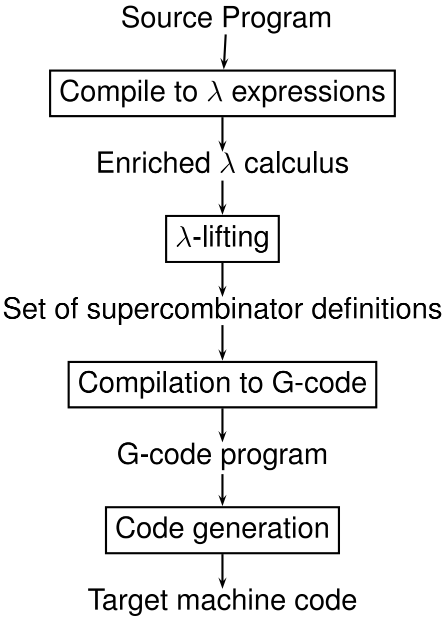
# G-machine

- Heart of functional programming paradigm is function application
- Need an efficient way to apply function body to arguments
- i.e., efficient graph reduction
- G-machine is an extremely fast implementation of graph reduction based on supercombinator compilation.
  - developed at Chalmers Institute of Technology, Sweden
  - by Johnsson and Augustsson

# G-code

- An *intermediate* code
- code for an *abstract sequential* machine
- Machine code for a specific target is generated from G-code

# Structure of G-machine Compiler



# Example of G-machine Execution

- Consider the following program in supercombinator form:

```
$from n = n :$from ($succ n)
$succ n = n + 1
$Prog = $from ($succ 0)
```

- We shall look at the code along with snapshots of execution.

# Startup code

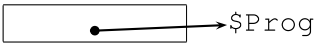
| | |
|---|---|
| `begin` | ◀ Begining of program |
| `pushglobal $Prog` | Push $Prog onto stack |
| `eval` | Evaluate it |
| `print` | Print the result |
| `end` | End of program |

# Startup code

| | |
|---|---|
| `begin` | Begining of program |
| `pushglobal $Prog` | ◀ Push $Prog onto stack |
| `eval` | Evaluate it |
| `print` | Print the result |
| `end` | End of program |

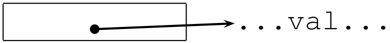
# Startup code

| | | |
|---|---|---|
| `begin` | | Begining of program |
| `pushglobal $Prog` | | Push $Prog onto stack |
| `eval` | ◀ | Evaluate it |
| `print` | | Print the result |
| `end` | | End of program |



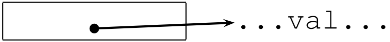
# Startup code

| | |
|---|---|
| `begin` | Begining of program |
| `pushglobal $Prog` | Push $Prog onto stack |
| `eval` | Evaluate it |
| `print` | Print the result |
| `end` | End of program |



# Startup code

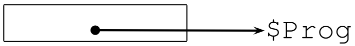
| | |
|---|---|
| `begin` | Begining of program |
| `pushglobal $Prog` | Push $Prog onto stack |
| `eval` | Evaluate it |
| `print` | Print the result |
| `end` | ◀ End of program |



# $Prog code

```
$Prog = $from ($succ 0)
```

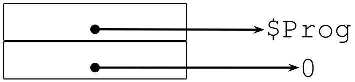
| | |
|---|---|
| `pushint 0` | ◀ Push constant 0 |
| `pushglobal $succ` | Push function $succ |
| `mkapp` | Construct ($succ 0) |
| `pushglobal $from` | Push function $from |
| `mkapp` | Construct ($from ($succ 0)) |
| `update 1` | Update the root of the redex |
| `unwind` | Initiate next reduction |



# $Prog code

```
$Prog = $from ($succ 0)
```

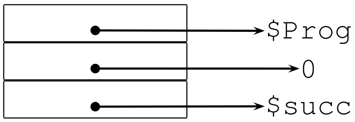
| | | |
|---|---|---|
| `pushint 0` | | Push constant 0 |
| `pushglobal $succ` | ◀ | Push function $succ |
| `mkapp` | | Construct ($succ 0) |
| `pushglobal $from` | | Push function $from |
| `mkapp` | | Construct ($from ($succ 0)) |
| `update 1` | | Update the root of the redex |
| `unwind` | | Initiate next reduction |



# $Prog code

```
$Prog = $from ($succ 0)
```

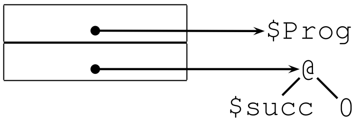
| | |
|---|---|
| `pushint 0` | Push constant 0 |
| `pushglobal $succ` | Push function $succ |
| `mkapp` | ◀ Construct ($succ 0) |
| `pushglobal $from` | Push function $from |
| `mkapp` | Construct ($from ($succ 0)) |
| `update 1` | Update the root of the redex |
| `unwind` | Initiate next reduction |



# $Prog code

```
$Prog = $from ($succ 0)
```

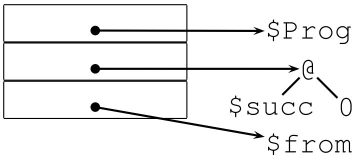
| | | |
|---|---|---|
| `pushint 0` | | Push constant 0 |
| `pushglobal $succ` | | Push function $succ |
| `mkapp` | | Construct ($succ 0) |
| `pushglobal $from` | ◀ | Push function $from |
| `mkapp` | | Construct ($from ($succ 0)) |
| `update 1` | | Update the root of the redex |
| `unwind` | | Initiate next reduction |



# $Prog code

```
$Prog = $from ($succ 0)
```

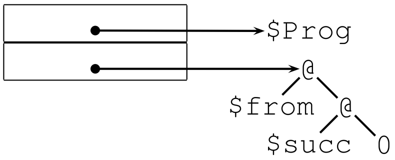
| | |
|---|---|
| `pushint 0` | Push constant 0 |
| `pushglobal $succ` | Push function $succ |
| `mkapp` | Construct ($succ 0) |
| `pushglobal $from` | Push function $from |
| `mkapp` | ◀ Construct ($from ($succ 0)) |
| `update 1` | Update the root of the redex |
| `unwind` | Initiate next reduction |



# $Prog code

```
$Prog = $from ($succ 0)
```

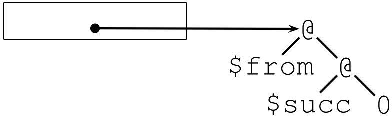
| | | |
|---|---|---|
| `pushint 0` | | Push constant 0 |
| `pushglobal $succ` | | Push function $succ |
| `mkapp` | | Construct ($succ 0) |
| `pushglobal $from` | | Push function $from |
| `mkapp` | | Construct ($from ($succ 0)) |
| `update 1` | ◀ | Update the root of the redex |
| `unwind` | | Initiate next reduction |



# $Prog code

```
$Prog = $from ($succ 0)
```

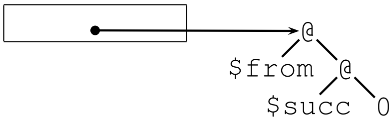
| | |
|---|---|
| `pushint 0` | Push constant 0 |
| `pushglobal $succ` | Push function $succ |
| `mkapp` | Construct ($succ 0) |
| `pushglobal $from` | Push function $from |
| `mkapp` | Construct ($from ($succ 0)) |
| `update 1` | Update the root of the redex |
| `unwind` | Initiate next reduction |



# $Prog code

```
$Prog = $from ($succ 0)
```

| | |
|---|---|
| `pushint 0` | Push constant 0 |
| `pushglobal $succ` | Push function $succ |
| `mkapp` | Construct ($succ 0) |
| `pushglobal $from` | Push function $from |
| `mkapp` | Construct ($from ($succ 0)) |
| `update 1` | Update the root of the redex |
| `unwind` | Initiate next reduction |

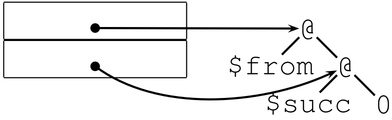



travels down the spine, identifies the tip, manipulates pointers so that they start pointing to the arguments, jumps to the code of $from.

# $Prog code

```
$Prog = $from ($succ 0)
```

| | |
|---|---|
| `pushint 0` | Push constant 0 |
| `pushglobal $succ` | Push function $succ |
| `mkapp` | Construct ($succ 0) |
| `pushglobal $from` | Push function $from |
| `mkapp` | Construct ($from ($succ 0)) |
| `update 1` | Update the root of the redex |
| `unwind` | Initiate next reduction |

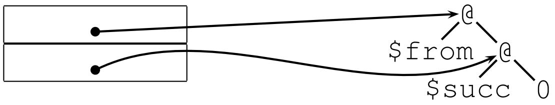



travels down the spine, identifies the tip, manipulates pointers so that they start pointing to the arguments, jumps to the code of $from.

# $from code

```
$from = n:$from ($succ 0)
```

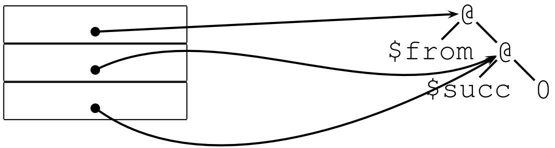
| | |
|---|---|
| `push 0` | ◀ Push n |
| `pushglobal $succ` | Push function $succ |
| `mkapp` | Construct ($succ n) |
| `pushglobal $from` | Push function $from |
| `mkapp` | Construct ($from ($succ n)) |
| `push 1` | Push n |
| `cons` | Construct (n:($from($succ n))) |
| `update 2` | Update the root of the redex |
| `pop 1` | Pop the parameter n |
| `unwind` | Initiate next reduction |



# $from code

```
$from = n:$from ($succ 0)
```

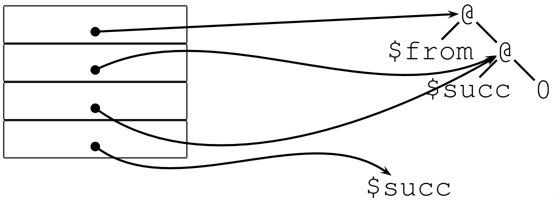
| | |
|---|---|
| `push 0` | Push n |
| `pushglobal $succ` | Push function $succ |
| `mkapp` | Construct ($succ n) |
| `pushglobal $from` | Push function $from |
| `mkapp` | Construct ($from ($succ n)) |
| `push 1` | Push n |
| `cons` | Construct (n:($from($succ n))) |
| `update 2` | Update the root of the redex |
| `pop 1` | Pop the parameter n |
| `unwind` | Initiate next reduction |



# $from code

```
$from = n:$from ($succ 0)
```

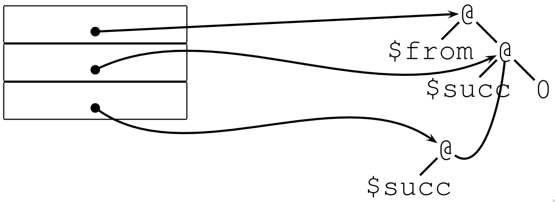
| | |
|---|---|
| `push 0` | Push n |
| `pushglobal $succ` | Push function $succ |
| `mkapp` | ◀ Construct ($succ n) |
| `pushglobal $from` | Push function $from |
| `mkapp` | Construct ($from ($succ n)) |
| `push 1` | Push n |
| `cons` | Construct (n:($from($succ n))) |
| `update 2` | Update the root of the redex |
| `pop 1` | Pop the parameter n |
| `unwind` | Initiate next reduction |



# $from code

```
$from = n:$from ($succ 0)
```

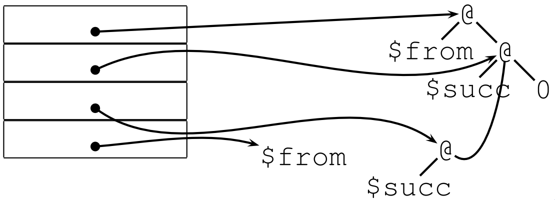
| | |
|---|---|
| `push 0` | Push n |
| `pushglobal $succ` | Push function $succ |
| `mkapp` | Construct ($succ n) |
| `pushglobal $from` | ◀ Push function $from |
| `mkapp` | Construct ($from ($succ n)) |
| `push 1` | Push n |
| `cons` | Construct (n:($from($succ n))) |
| `update 2` | Update the root of the redex |
| `pop 1` | Pop the parameter n |
| `unwind` | Initiate next reduction |



# $from code

```
$from = n:$from ($succ 0)
```

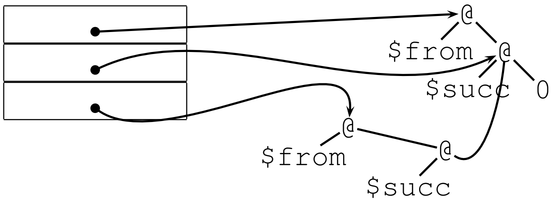
| | |
|---|---|
| `push 0` | Push n |
| `pushglobal $succ` | Push function $succ |
| `mkapp` | Construct ($succ n) |
| `pushglobal $from` | Push function $from |
| `mkapp` | ◀ Construct ($from ($succ n)) |
| `push 1` | Push n |
| `cons` | Construct (n:($from($succ n))) |
| `update 2` | Update the root of the redex |
| `pop 1` | Pop the parameter n |
| `unwind` | Initiate next reduction |



# $from code

```
$from = n:$from ($succ 0)
```

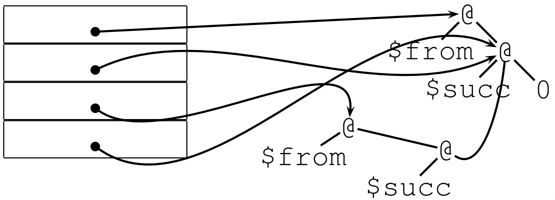
| | |
|---|---|
| `push 0` | Push n |
| `pushglobal $succ` | Push function $succ |
| `mkapp` | Construct ($succ n) |
| `pushglobal $from` | Push function $from |
| `mkapp` | Construct ($from ($succ n)) |
| `push 1` | Push n |
| `cons` | Construct (n:($from($succ n))) |
| `update 2` | Update the root of the redex |
| `pop 1` | Pop the parameter n |
| `unwind` | Initiate next reduction |



# $from code

```
$from = n:$from ($succ 0)
```

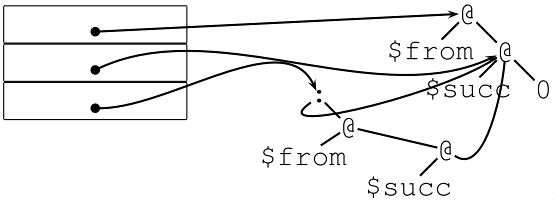
| | |
|---|---|
| `push 0` | Push n |
| `pushglobal $succ` | Push function $succ |
| `mkapp` | Construct ($succ n) |
| `pushglobal $from` | Push function $from |
| `mkapp` | Construct ($from ($succ n)) |
| `push 1` | Push n |
| `cons` | ◀ Construct (n:($from($succ n))) |
| `update 2` | Update the root of the redex |
| `pop 1` | Pop the parameter n |
| `unwind` | Initiate next reduction |



# $from code

```
$from = n:$from ($succ 0)
```

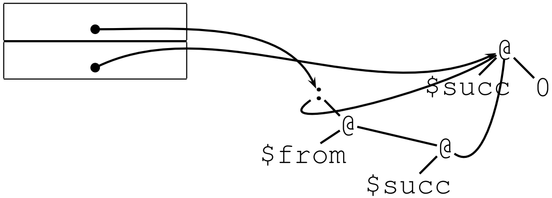
| | |
|---|---|
| `push 0` | Push n |
| `pushglobal $succ` | Push function $succ |
| `mkapp` | Construct ($succ n) |
| `pushglobal $from` | Push function $from |
| `mkapp` | Construct ($from ($succ n)) |
| `push 1` | Push n |
| `cons` | Construct (n:($from($succ n))) |
| `update 2` | ◀ Update the root of the redex |
| `pop 1` | Pop the parameter n |
| `unwind` | Initiate next reduction |



# $from code

```
$from = n:$from ($succ 0)
```

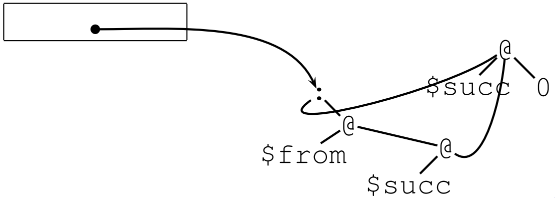
| | |
|---|---|
| `push 0` | Push n |
| `pushglobal $succ` | Push function $succ |
| `mkapp` | Construct ($succ n) |
| `pushglobal $from` | Push function $from |
| `mkapp` | Construct ($from ($succ n)) |
| `push 1` | Push n |
| `cons` | Construct (n:($from($succ n))) |
| `update 2` | Update the root of the redex |
| `pop 1` | ◀ Pop the parameter n |
| `unwind` | Initiate next reduction |



# $from code

```
$from = n:$from ($succ 0)
```

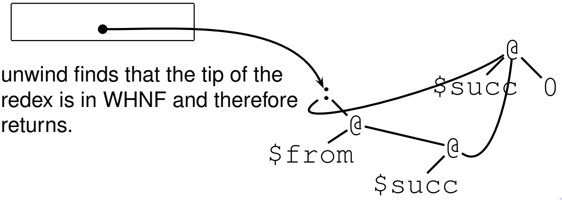
| | |
|---|---|
| `push 0` | Push n |
| `pushglobal $succ` | Push function $succ |
| `mkapp` | Construct ($succ n) |
| `pushglobal $from` | Push function $from |
| `mkapp` | Construct ($from ($succ n)) |
| `push 1` | Push n |
| `cons` | Construct (n:($from($succ n))) |
| `update 2` | Update the root of the redex |
| `pop 1` | Pop the parameter n |
| `unwind` | ◀ Initiate next reduction |



# $from code

```
$from = n:$from ($succ 0)
```

| | |
|---|---|
| `push 0` | Push n |
| `pushglobal $succ` | Push function $succ |
| `mkapp` | Construct ($succ n) |
| `pushglobal $from` | Push function $from |
| `mkapp` | Construct ($from ($succ n)) |
| `push 1` | Push n |
| `cons` | Construct (n:($from($succ n))) |
| `update 2` | Update the root of the redex |
| `pop 1` | Pop the parameter n |
| `unwind` | Initiate next reduction |




unwind finds that the tip of the redex is in WHNF and therefore returns.

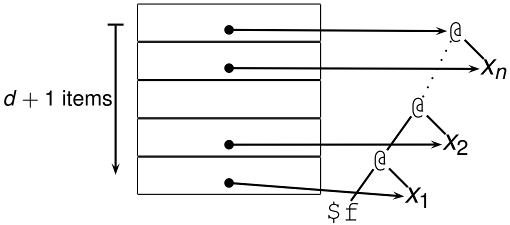
# The General Compilation Scheme

- The startup code is always

```
begin
pushglobal $Prog
eval
print
end
```

# Code for Supercombinator

- The code for supercombinator is given by $\mathbb{F}$ compilation scheme
  $\mathbb{F}[\![\ \$F\ x_1\ \ x_2 \ldots x_n\ ]\!] = \ldots$ G-code for $\$F \ldots$
- To define $\mathbb{F}$, note the situation just before entering a supercombinator body



# Context

- The distance from the top of the stack to the root of the redex is called the *context*
- Its length is denoted as $d + 1$
- The initial value of $d$ is $n$

  and changes as the execution of the supercombinator body proceeds.

# Environment Function $\rho$

- The distance between $i^{th}$ arg and the root of the redex is given by an environment function $\rho$
- $\rho\ x_i = n + 1 - i$
- Thus, the distance of an argument $i$ from the top of the stack is $d - (\rho\ x_i)$.
- This is how we keep track of the position of an argument from the top of the stack.
- Note that $\rho$ is constant for a supercombinator, but $d$ has to be tracked as compilation proceeds.

# $\mathbb{F}$ Compilation Scheme

$$\begin{aligned}\mathbb{F}[\![\ \$F\ x_1\ x_2 \ldots x_n\ ]\!] = {} & \text{globstart } \$F, n \\ & \mathbb{R}[\![\ E\ ]\!]\ [x_1 = n, x_2 = n-1, \ldots x_n = 1]\ n\end{aligned}$$

# $\mathbb{R}$ Compilation Scheme

$$
\begin{aligned}
\mathbb{R}[\![\ E\ ]\!]\ \rho\ d = {} & \mathbb{C}[\![\ E\ ]\!]\ \rho\ d \\
& \text{update } (d+1) \\
& \text{pop } d \\
& \text{unwind}
\end{aligned}
$$

# $\mathbb{C}$ Compilation Scheme

- $\mathbb{C}[\![ E ]\!] \rho d$
- The definition of $\mathbb{C}[\![ \ ]\!]$ is given through case by case basis.

# $E$ is a constant

- $E$ is an integer $i$ (or Boolean etc.)

$$\mathbb{C}[\![\ i\ ]\!]\ \rho\ d = \text{pushint}\ i$$

- $E$ is a supercombinator or built-in function, $f$

$$\mathbb{C}[\![\ f\ ]\!]\ \rho\ d = \text{pushglobal}\ f$$

# $E$ is a variable

- The value of a variable $x$ is in stack
- At offset $(d - \rho\ x)$ from the top
- We would like to bring it to the top
- The G-code is

$$\mathbb{C}[\![\ x\ ]\!]\ \rho\ d = \text{push}\ (d - \rho\ x)$$

# $E$ is an application

- Application $(E_1 \; E_2)$
- First construct an instance of $E_2$
- Leave a pointer to the $E_2$ instance at the top of the stack
- Construct an instance of $E_1$ - but adjust for the $E_2$'s instance in the context ($d$)
- Make an application Cell on the top of the stack

$$
\begin{aligned}
\mathbb{C}[\![ \; E_1 \; E_2 \; ]\!] \; \rho \; d = \; & \mathbb{C}[\![ \; E_2 \; ]\!] \; \rho \; d \\
& \mathbb{C}[\![ \; E_1 \; ]\!] \; \rho \; (d + 1) \\
& \text{mkap}
\end{aligned}
$$

# $E$ is a let-expression

- $E$ is let $x = E_x$ in $E_b$
- Construct instance of $E_x$, leave pointer to it at the stack top
- Augment $\rho$ to say that $x$ is at offset $d + 1$ from the base of the context
- Construct instance of $E_b$, using updated $\rho$ and $d$
- Now $E_b$ instance is at the top, $E_x$ instance is below it. Since we do not want $E_x$ any more, clean up.

  $$\begin{aligned} \mathbb{C}[\![ \text{ let } x = E_x \text{ in } E_b ]\!] \; \rho \; d = \; & \mathbb{C}[\![ E_x ]\!] \; \rho \; d \\ & \mathbb{C}[\![ E_b ]\!] \; \rho[x = d + 1] \; (d + 1) \\ & \text{slide } 1 \end{aligned}$$