

Small-step operational semantics and SML

Lecture 3

MPhil ACS & Part III course, Functional Programming:
Implementation, Specification and Verification

Magnus Myreen
Michaelmas term, 2013

Non-termination

From last lecture:

Big-step operational semantics

- inductive relation describes terminating evaluations
- “big-step” because term-to-result evaluation is described by a single transition

Non-termination

From last lecture:

Big-step operational semantics

- inductive relation describes terminating evaluations
- “big-step” because term-to-result evaluation is described by a single transition

can't directly model non-termination

Non-termination

From last lecture:

Big-step operational semantics

- inductive relation describes terminating evaluations
- “big-step” because term-to-result evaluation is described by a single transition

can't directly model non-termination

This lecture:

Small-step operational semantics

- models evaluation in “small steps”
- possibly infinite sequence of steps

Non-termination

From last lecture:

Big-step operational semantics

- inductive relation describes terminating evaluations
- “big-step” because term-to-result evaluation is described by a single transition

can't directly model non-termination

This lecture:

Small-step operational semantics

- models evaluation in “small steps”
- possibly infinite sequence of steps

can model non-termination

Non-termination

From last lecture:

Big-step operational semantics

can't directly model non-termination

- inductive relation describes terminating evaluations
- “big-step” because term-to-result evaluation is described by a single transition

This lecture:

Small-step operational semantics

can model non-termination

- models evaluation in “small steps”
- possibly infinite sequence of steps

Also in this lecture:

Sketch of a formal semantics for a real FP language (SML)

Evaluation as small steps

The steps execute an [abstract machine](#).

Evaluation as small steps

The steps execute an **abstract machine**.

Abstract machines:

- **Turing machine** for general computation

Evaluation as small steps

The steps execute an **abstract machine**.

Abstract machines:

- **Turing machine** for general computation
- Landin's **SECD machine** for idealised FP implementation (evaluates lambda calculus)

SECD = Stack, Environment, Code and Dump

Evaluation as small steps

The steps execute an **abstract machine**.

Abstract machines:

- **Turing machine** for general computation
- Landin's **SECD machine** for idealised FP implementation (evaluates lambda calculus)

SECD = Stack, Environment, Code and Dump

- We will use a modern version of SECD, a Felleisen and Friedman's **CEK machine**

CEK = Control, Environment and (K)ontinuation

sometimes called CESK, where S = store

A simple expression language

We start with a **very simple** language:

`val ::= num`

`exp ::= Const num`
`| Add exp exp`

A simple expression language

We start with a **very simple** language:

$\text{val} ::= \text{num}$

$\text{exp} ::= \text{Const num}$
 $\quad \mid \text{Add exp exp}$

A **big-step** semantics for this:

$$\frac{}{(\text{Const } n, env) \Downarrow_{ev} n} \quad \frac{(e_1, env) \Downarrow_{ev} s_1 \quad (e_2, env) \Downarrow_{ev} s_2}{(\text{Add } e_1 \ e_2, env) \Downarrow_{ev} (s_1 + s_2)}$$

A simple expression language

We start with a **very simple** language:

$\text{val} ::= \text{num}$

$\text{exp} ::= \text{Const num}$
 $\quad \mid \text{Add exp exp}$

A **big-step** semantics for this:

$$\frac{}{(\text{Const } n, env) \Downarrow_{ev} n} \quad \frac{(e_1, env) \Downarrow_{ev} s_1 \quad (e_2, env) \Downarrow_{ev} s_2}{(\text{Add } e_1 \ e_2, env) \Downarrow_{ev} (s_1 + s_2)}$$

a single step

Leaving holes

How to evaluate in steps?

Add e_1 e_2

Leaving holes

How to evaluate in steps?

Add e_1 e_2

A small-step machine should:

- evaluate e_1 first, but
- remember to come back and evaluate e_2 and Add

Leaving holes

How to evaluate in steps?

Add e_1 e_2

A small-step machine should:

- evaluate e_1 first, but
- remember to come back and evaluate e_2 and Add

Solution: keep a 'TO-DO list' (continuation) with holes in exps

Sketch of the idea

exp: Add (Add 1 2) 3 continuation:

Sketch of the idea

exp: Add (Add 1 2) 3

continuation:

exp: Add 1 2

continuation: Add ?? 3

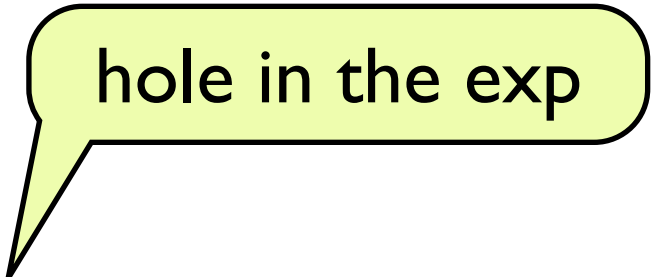
Sketch of the idea

exp: Add (Add 1 2) 3

continuation:

exp: Add 1 2

continuation: Add ?? 3



hole in the exp

Sketch of the idea

exp: Add (Add 1 2) 3

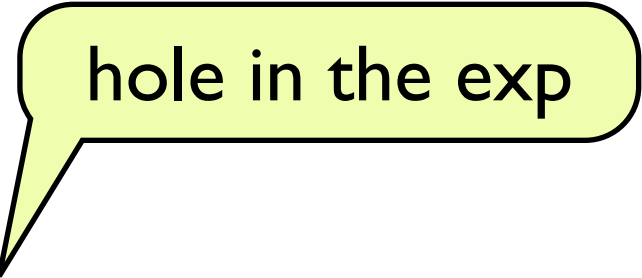
continuation:

exp: Add 1 2

continuation: Add ?? 3

exp: 1

continuation: Add ?? 2, Add ?? 3



hole in the exp

Sketch of the idea

exp: Add (Add 1 2) 3

continuation:

exp: Add 1 2

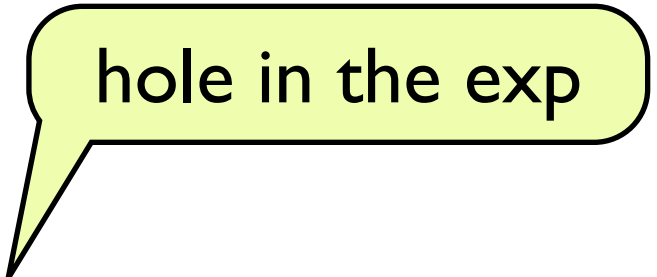
continuation: Add ?? 3

exp: 1

continuation: Add ?? 2, Add ?? 3

val: 1

continuation: Add ?? 2, Add ?? 3



hole in the exp

Sketch of the idea

exp: Add (Add 1 2) 3

continuation:

exp: Add 1 2

continuation: Add ?? 3

exp: 1

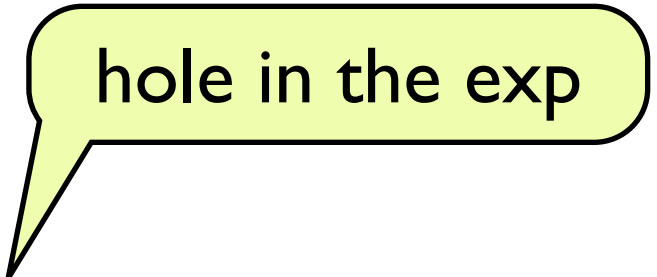
continuation: Add ?? 2, Add ?? 3

val: 1

continuation: Add ?? 2, Add ?? 3

exp: 2

continuation: Add 1 ??, Add ?? 3



hole in the exp

Sketch of the idea

exp: Add (Add 1 2) 3

continuation:

hole in the exp

exp: Add 1 2

continuation: Add ?? 3

exp: 1

continuation: Add ?? 2, Add ?? 3

val: 1

continuation: Add ? 2, Add ?? 3

val in the exp

exp: 2

continuation: Add 1 ??, Add ?? 3

Sketch of the idea

exp: Add (Add 1 2) 3

continuation:

hole in the exp

exp: Add 1 2

continuation: Add ?? 3

exp: 1

continuation: Add ?? 2, Add ?? 3

val: 1

continuation: Add ? 2, Add ?? 3

val in the exp

exp: 2

continuation: Add 1 ??, Add ?? 3

val: 2

continuation: Add 1 ??, Add ?? 3

Sketch of the idea

exp: Add (Add 1 2) 3

continuation:

hole in the exp

exp: Add 1 2

continuation: Add ?? 3

exp: 1

continuation: Add ?? 2, Add ?? 3

val: 1

continuation: Add ? 2, Add ?? 3

val in the exp

exp: 2

continuation: Add 1 ??, Add ?? 3

val: 2

continuation: Add 1 ??, Add ?? 3

val: 3

continuation: Add ?? 3

Sketch of the idea

exp: Add (Add 1 2) 3

continuation:

hole in the exp

exp: Add 1 2

continuation: Add ?? 3

exp: 1

continuation: Add ?? 2, Add ?? 3

val: 1

continuation: Add ? 2, Add ?? 3

val in the exp

exp: 2

continuation: Add 1 ??, Add ?? 3

val: 2

continuation: Add 1 ??, Add ?? 3

val: 3

continuation: Add ?? 3

exp: 3

continuation: Add 3 ??

Sketch of the idea

exp: Add (Add 1 2) 3

continuation:

hole in the exp

exp: Add 1 2

continuation: Add ?? 3

exp: 1

continuation: Add ?? 2, Add ?? 3

val: 1

continuation: Add ? 2, Add ?? 3

val in the exp

exp: 2

continuation: Add 1 ??, Add ?? 3

val: 2

continuation: Add 1 ??, Add ?? 3

val: 3

continuation: Add ?? 3

exp: 3

continuation: Add 3 ??

val: 3

continuation: Add 3 ??

Sketch of the idea

exp: Add (Add 1 2) 3

continuation:

hole in the exp

exp: Add 1 2

continuation: Add ?? 3

exp: 1

continuation: Add ?? 2, Add ?? 3

val in the exp

val: 1

continuation: Add ? 2, Add ?? 3

exp: 2

continuation: Add 1 ??, Add ?? 3

val: 2

continuation: Add 1 ??, Add ?? 3

val: 3

continuation: Add ?? 3

exp: 3

continuation: Add 3 ??

val: 3

continuation: Add 3 ??

val: 6

continuation:

Defining a step

We define what expressions with holes and values means:

```
ctxt ::= CAdd1 hole exp  
      | CAdd2 val hole
```

!

Defining a step

We define what expressions with holes and values means:

```
ctxt ::= CAdd1 hole exp
      | CAdd2 val hole
```

The state consists of env, exp-or-val and a list of ctxt (continuation):

```
state = (env, exp_or_val, (ctxt * env) list)
```

```
exp_or_val ::= Exp exp | Val val
```

Defining a step

We define what expressions with holes and values means:

```
ctxt ::= CAdd1 hole exp
      | CAdd2 val hole
```

The state consists of env, exp-or-val and a list of ctxt (continuation):

```
state = (env, exp_or_val, (ctxt * env) list)
```

```
exp_or_val ::= Exp exp | Val val
```

A step can either return or get stuck:

```
result = OK state | STUCK
```

The step function

```
step (env, e_or_v, c) =  
  case e_or_v of  
  | Val v => continue v c  
  | Exp (Const n) => return env n c  
  | Exp (Add e1 e2) => push env e1 (CAdd1 () e2) c
```

```
push env e c1 c = OK (env, Exp e, (c1, env) :: c)
```

```
return env v c = OK (env, Val v, c)
```

```
continue v c =  
  case c of  
  | [] => STUCK  
  | ((CAdd1 () e2, env) :: cs) => OK (env, Exp e2, (CAdd2 v ()) :: cs)  
  | ((CAdd2 v1 (), env) :: cs) => OK (env, Val (v1 + v), cs)
```


A few more functions

A function which recognises a final state:

```
is_final state =  
  case state of  
  | (env, Val v, []) => true  
  | _ => false
```

A function that can be used to run the step function:

```
steps state =  
  if is_final state then [OK state] else  
  case step state of  
  | STUCK => [STUCK]  
  | OK new_state => OK state :: steps new_state
```

NB: `steps` cannot be defined in HOL because it might not terminate.

Sample run

```
steps (env, Exp (Add (Const 1) (Const 2)), []) =
  [ OK (env, Exp (Add (Const 1) (Const 2)), []),
    OK (env, Exp (Const 1), [CAdd1 () (Const 2)]),
    OK (env, Val 1, [CAdd1 () (Const 2)]),
    OK (env, Exp (Const 2), [CAdd2 1 ()]),
    OK (env, Val 2, [CAdd2 1 ()]),
    OK (env, Val 3, [] ) ]
```

Formal model

We define a **relation** describing a good step:

$$s_1 \rightarrow s_2 \equiv (\text{step } s_1 = \text{OK } s_2)$$

Formal model

We define a **relation** describing a good step:

$$s_1 \rightarrow s_2 \equiv (\text{step } s_1 = \text{OK } s_2)$$

Using this we can define:

$$\text{terminating_eval } env \ e \ v \equiv (env, \text{Exp } e, []) \rightarrow^* (-, v, [])$$

Formal model

We define a **relation** describing a good step:

$$s_1 \rightarrow s_2 \equiv (\text{step } s_1 = \text{OK } s_2)$$

Using this we can define:

$$\text{terminating_eval } env \ e \ v \equiv (env, \text{Exp } e, []) \rightarrow^* (-, v, [])$$

reflexive-transitive closure

Formal model

We define a **relation** describing a good step:

$$s_1 \rightarrow s_2 \equiv (\text{step } s_1 = \text{OK } s_2)$$

Using this we can define:

reflexive-transitive closure

$$\text{terminating_eval } env \ e \ v \equiv (env, \text{Exp } e, []) \rightarrow^* (-, v, [])$$

$$\text{non_terminating_eval } env \ e \ v \equiv \\ (\forall s. (env, \text{Exp } e, []) \rightarrow^* s \implies \exists t. s \rightarrow t)$$

Extending the language

val ::= num

exp ::= Const num

| Var string

| Add exp exp

| If exp exp exp

| Let string exp exp

ctxt ::= CAdd1 hole exp

| CAdd2 val hole

| CIf hole exp exp

| CLet string hole exp

Extending the functions

```
step (env, e_or_v, c) =  
  case e_or_v of  
    ...  
  | Exp (Var n) => return env (lookup n env) c  
  | Exp (If e1 e2 e3) => push env e1 (CIf () e2 e3) c  
  | Exp (Let name e1 e2) => push env e1 (CLet name () e2) c
```

```
continue v c =  
  case c of  
    ...  
  | ((CIf () e2 e3, env)::cs) =>  
    OK (env, Exp (if isTrue v then e2 else e3), cs)  
  | ((CLet name () body, env)::cs) =>  
    OK (env[name := v], Exp body, cs)
```


Adding support for functions


```
val ::= Num num  
      | Closure string env exp
```

```
exp ::= Const num  
      | Var string  
      | Add exp exp  
      | If exp exp exp  
      | Let string exp exp  
      | App exp exp  
      | Fn string exp
```

```
ctxt ::= CAdd1 hole exp  
       | CAdd2 val hole  
       | CIf hole exp exp  
       | CLet string hole exp  
       | CApp1 hole exp  
       | CApp2 val hole
```

Adding support for functions

val ::= Num num
| Closure string env exp



function value

exp ::= Const num
| Var string
| Add exp exp
| If exp exp exp
| Let string exp exp
| App exp exp
| Fn string exp

ctxt ::= CAdd1 hole exp
| CAdd2 val hole
| CIf hole exp exp
| CLet string hole exp
| CApp1 hole exp
| CApp2 val hole

Adding support for functions

val ::= Num num
| Closure string env exp

function value

exp ::= Const num
| Var string
| Add exp exp
| If exp exp exp
| Let string exp exp
| App exp exp
| Fn string exp

function application

ctxt ::= CAdd1 hole exp
| CAdd2 val hole
| CIf hole exp exp
| CLet string hole exp
| CApp1 hole exp
| CApp2 val hole

Adding support for functions

val ::= Num num

| Closure string env exp

function value

exp ::= Const num

| Var string

| Add exp exp

| If exp exp exp

| Let string exp exp

| App exp exp

function application

| Fn string exp

anonymous function

ctxt ::= CAdd1 hole exp

| CAdd2 val hole

| CIf hole exp exp

| CLet string hole exp

| CApp1 hole exp

| CApp2 val hole

Additions to functions

```
step (env, e_or_v, c) =  
  case e_or_v of  
    ...  
    | Exp (App e1 e2) => push env e1 (CApp1 () e2) c  
    | Exp (Fn name e1) => return env (Closure name env e1) c
```

```
do_app v1 v2 =  
  case v1 of  
    | Closure name env exp => SOME (env[name := v2], exp)  
    | _ => NONE
```

```
continue v c =  
  case c of  
    ...  
    | ((CApp1 () e2, env)::cs) =>  
      OK (env, Exp e2, CApp2 v () :: cs)  
    | ((CApp2 v1 (), env)::cs) =>  
      case do_app v1 v of  
        | SOME (env, e) => OK (env, Exp e, cs)  
        | NONE => STUCK
```

Additions to functions

```
step (env, e_or_v, c) =  
  case e_or_v of
```

```
  ...
```

```
  | Exp (App e1 e2) => push env e1 (CApp1 () e2) c
```

```
  | Exp (Fn name e1) => return env (Closure name env e1) c
```



closure value created

```
do_app v1 v2 =
```

```
  case v1 of
```

```
  | Closure name env exp => SOME (env[name := v2], exp)
```

```
  | _ => NONE
```

```
continue v c =
```

```
  case c of
```

```
  ...
```

```
  | ((CApp1 () e2, env)::cs) =>  
    OK (env, Exp e2, CApp2 v () :: cs)
```

```
  | ((CApp2 v1 (), env)::cs) =>
```

```
    case do_app v1 v of
```

```
    | SOME (env, e) => OK (env, Exp e, cs)
```

```
    | NONE => STUCK
```

Additions to functions

```
step (env, e_or_v, c) =  
  case e_or_v of
```

```
  ...
```

```
  | Exp (App e1 e2) => push env e1 (CApp1 () e2) c
```

```
  | Exp (Fn name e1) => return env (Closure name env e1) c
```

closure value created

```
do_app v1 v2 =
```

```
  case v1 of
```

```
  | Closure name env exp => SOME (env[name := v2], exp)
```

```
  | _ => NONE
```

function application fails if value is not a function

```
continue v c =
```

```
  case c of
```

```
  ...
```

```
  | ((CApp1 () e2, env)::cs) =>
```

```
    OK (env, Exp e2, CApp2 v () :: cs)
```

```
  | ((CApp2 v1 (), env)::cs) =>
```

```
    case do_app v1 v of
```

```
    | SOME (env, e) => OK (env, Exp e, cs)
```

```
    | NONE => STUCK
```

Additions to functions

```
step (env, e_or_v, c) =  
  case e_or_v of
```

```
  ...
```

```
  | Exp (App e1 e2) => push env e1 (CApp1 () e2) c
```

```
  | Exp (Fn name e1) => return env (Closure name env e1) c
```

closure value created

```
do_app v1 v2 =
```

```
  case v1 of
```

```
  | Closure name env exp => SOME (env[name := v2], exp)
```

```
  | _ => NONE
```

function application fails if value is not a function

```
continue v c =
```

```
  case c of
```

```
  ...
```

```
  | ((CApp1 () e2, env)::cs) =>
```

```
    OK (env, Exp e2, CApp2 v () :: cs)
```

```
  | ((CApp2 v1 (), env)::cs) =>
```

```
    case do_app v1 v of
```

```
    | SOME (env, e) => OK (env, Exp e, cs)
```

```
    | NONE => STUCK
```

failed application causes STUCK state

Sample run

In ML syntax: `(fn n => n + 1) 2`

steps

```
([], Exp (App (Fn "n" (Add (Var "n") (Const 1))) (Const 2)), [])
```

=

```
[OK ([], Exp (App (Fn "n" (Add (Var "n") (Const 1))) (Const 2)), []),  
OK ([], Exp (Fn "n" (Add (Var "n") (Const 1))), [CApp1 () (Const 2)]),  
OK ([], Val (Closure "n" [] (Add (Var "n") (Const 1))), [CApp1 () (Const 2)]),  
OK ([], Exp (Const 2), [CApp2 (Closure "n" [] (Add (Var "n") (Const 1))) ()]),  
OK ([], Val (Num 2), [CApp2 (Closure "n" [] (Add (Var "n") (Const 1))) ()]),  
OK ([["n", Num 2], Exp (Add (Var "n") (Const 1)), []),  
OK ([["n", Num 2], Exp (Var "n"), [CAdd1 () (Const 1)]),  
OK ([["n", Num 2], Val (Num 2), [CAdd1 () (Const 1)]),  
OK ([["n", Num 2], Exp (Const 1), [CAdd2 (Num 2) ()]),  
OK ([["n", Num 2], Val (Num 1), [CAdd2 (Num 2) ()]),  
OK ([["n", Num 2], Val (Num 3), [])]
```

Another sample run

In ML syntax: 1(2)

```
steps ([], Exp (App (Const 1) (Const 2)), [])
```

```
=
```

```
[OK ([], Exp (App (Const 1) (Const 2)), []),  
  OK ([], Exp (Const 1), [CApp1 () (Const 2)]),  
  OK ([], Val (Num 1), [CApp1 () (Const 2)]),  
  OK ([], Exp (Const 2), [CApp2 (Num 1) ()]),  
  STUCK]
```

Another sample run

In ML syntax: 1(2)

```
steps ([], Exp (App (Const 1) (Const 2)), [])  
=  
[OK ([], Exp (App (Const 1) (Const 2)), []),  
  OK ([], Exp (Const 1), [CApp1 () (Const 2)]),  
  OK ([], Val (Num 1), [CApp1 () (Const 2)]),  
  OK ([], Exp (Const 2), [CApp2 (Num 1) ()]),  
  STUCK]
```

Properly set up, we can **prove** that **well-typed** programs **do not get stuck**.
(later lecture)

Towards full SML

Full Standard ML (SML) includes:

- recursive functions
- datatypes (constructors and pattern-matching)
- user-declared exceptions (`raise`, `handle`)
- references (`ref`, `!`, `:=`)
- modules

Extending the val type

```
env = (string * val) list
```

```
val ::= Num num  
      | Bool bool  
      | Datatype string (val list)  
      | Closure string env exp  
      | Recclosure string string env exp  
      | Loc of num
```

Extending the val type

env = (string * val) list

val ::= Num num

| Bool bool

| Datatype string (val list)

| Closure string env exp

| Recclosure string string env exp

| Loc of num

datatype constructors

Extending the val type

env = (string * val) list

val ::= Num num
| Bool bool
| Datatype string (val list)
| Closure string env exp
| Recclosure string string env exp
| Loc of num

datatype constructors

In ML syntax: [1,2,3]

In model:

Datatype "cons" [Num 1,
Datatype "cons" [Num 2,
Datatype "cons" [Num 3,
Datatype "nil" []]]]

Extending the val type

env = (string * val) list

val ::= Num num

| Bool bool

| Datatype string (val list)

| Closure string env exp

| Recclosure string string env exp

| Loc of num

datatype constructors

value of a rec. closure

In ML syntax: [1,2,3]

In model:

Datatype "cons" [Num 1,

Datatype "cons" [Num 2,

Datatype "cons" [Num 3,

Datatype "nil" []]]

Extending the val type

env = (string * val) list

val ::= Num num

| Bool bool

| Datatype string (val list)

| Closure string env exp

| Recclosure string string env exp

| Loc of num

datatype constructors

value of a rec. closure

value of an ML ref, i.e. pointer into global store (imperative feature)

In ML syntax: [1,2,3]

In model:

Datatype "cons" [Num 1,

Datatype "cons" [Num 2,

Datatype "cons" [Num 3,

Datatype "nil" []]]]

Recursive functions

```
do_app v1 v2 =  
  case v1 of  
  | Closure name env exp => SOME (env[name := v2], exp)  
  | Recclosure f name env exp =>  
    SOME (env[name := v2, f := Recclosure f name env exp], exp)  
  | _ => NONE
```

Recursive functions

```
do_app v1 v2 =  
  case v1 of  
  | Closure name env exp => SOME (env[name := v2], exp)  
  | Recclosure f name env exp =>  
    SOME (env[name := v2, f := Recclosure f name env exp], exp)  
  | _ => NONE
```

body of rec. closure can refer to itself via this name

Recursive functions

```
do_app v1 v2 =  
  case v1 of  
  | Closure name env exp => SOME (env[name := v2], exp)  
  | Recclosure f name env exp =>  
    SOME (env[name := v2, f := Recclosure f name env exp], exp)  
  | _ => NONE
```

body of rec. closure can refer to itself via this name

Producing a rec. closure in SML:

```
let  
  fun fac n = if n = 0 then 1 else n * fac (n-1)  
in fac end
```

Exceptions

Example use of `raise` and `handle` in SML:

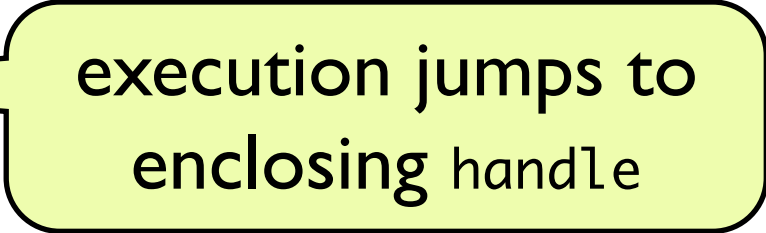
```
let
  val _ = print "Hello"
  val _ = raise (ErrorNumber 3)
  val _ = print " there!"
in 1 end
handle ErrorNumber n => n
```

This SML expression returns 3 and prints only Hello.

Exceptions

Example use of `raise` and `handle` in SML:

```
let
  val _ = print "Hello"
  val _ = raise (ErrorNumber 3)
  val _ = print " there!"
in 1 end
handle ErrorNumber n => n
```



execution jumps to
enclosing handle

This SML expression returns 3 and prints only Hello.

Exceptions

Example use of `raise` and `handle` in SML:

```
let
  val _ = print "Hello"
  val _ = raise (ErrorNumber 3)
  val _ = print " there!"
in 1 end
handle ErrorNumber n => n
```

execution jumps to
enclosing handle

execution continues here

This SML expression returns 3 and prints only Hello.

Semantics for exceptions

```
step (env, e_or_v, c) =  
  case e_or_v of  
  ...  
  | Exp (Raise e) => push env e (CRaise ()) c  
  | Exp (Handle e1 name e2) => push env e1 (CHandle () name e2) c
```

```
continue env v c =  
  case c of  
  ...  
  | ((CRaise (), env) :: cs) =>  
    case c of  
    | [] => UncaughtExc v  
    | ((CHandle () name e2) :: c) =>  
      OK (env[name := v], Exp e2, c)  
    | _ :: c => OK (env, Val v, ((CRaise (), env) :: c))  
  | ((CHandle () name e2, env) :: cs) => return env v c
```


Semantics for exceptions

```
step (env, e_or_v, c) =  
  case e_or_v of
```

```
  ...
```

```
  | Exp (Raise e) => push env e (CRaise ()) c
```

```
  | Exp (Handle e1 name e2) => push env e1 (CHandle () name e2) c
```

```
continue env v c =  
  case c of
```

```
  ...
```

```
  | ((CRaise (), env) :: cs) =>
```

```
    case c of
```

```
    | [] => UncaughtExc v
```

```
    | ((CHandle () name e2) :: c) =>
```

```
      OK (env[name := v], Exp e2, c)
```

```
    | _ :: c => OK (env, Val v, ((CRaise (), env) :: c))
```

```
  | ((CHandle () name e2, env) :: cs) => return env v c
```

raise looks for enclosing handle

Semantics for exceptions

```
step (env, e_or_v, c) =  
  case e_or_v of
```

```
  ...
```

```
  | Exp (Raise e) => push env e (CRaise ()) c
```

```
  | Exp (Handle e1 name e2) => push env e1 (CHandle () name e2) c
```

```
continue env v c =  
  case c of
```

```
  ...
```

```
  | ((CRaise (), env) :: cs) =>
```

```
    case c of
```

```
    | [] => UncaughtExc v
```

```
    | ((CHandle () name e2) :: c) =>
```

```
      OK (env[name := v], Exp e2, c)
```

```
    | _ :: c => OK (env, Val v, ((CRaise (), env) :: c))
```

```
  | ((CHandle () name e2, env) :: cs) => return env v c
```

raise looks for enclosing handle

handle just returns, i.e. transparent

Support for ML references

To support references (stateful), we need to pass around state:

```
step (env, e_or_v, c, state) =  
  case e_or_v of  
  | Val v => continue env v c state  
  | Exp (Const n) => return env n c state  
  | Exp (Add e1 e2) => push env e1 (CAdd1 () e2) c state
```

```
push env e c1 c state = OK (env, Exp e, (c1, env) :: c, state)
```

```
return env v c state = OK (env, Val v, c, state)
```

```
continue env v c state =
```

```
...
```

The state is a finite mapping from num to val.

Long definition ... help?

Use tool support!

Lem, a tool for lightweight executable mathematics

<http://www.cs.kent.ac.uk/people/staff/sao/lem/>

Ott, a tool for writing definitions of programming languages and calculi

<http://www.cl.cam.ac.uk/~pes20/ott/>

These tools are not examinable material.

Summary

Small-step operational semantics

- models evaluation in “small steps”
- possibly infinite sequence of steps
- can model non-termination
- handy for proof about type soundness (later lecture)

Summary

Small-step operational semantics

- models evaluation in “small steps”
- possibly infinite sequence of steps
- can model non-termination
- handy for proof about type soundness (later lecture)

Big-step operational semantics

- more convenient for certain proofs (e.g. compiler correctness, later lecture)

Summary

Small-step operational semantics

- models evaluation in “small steps”
- possibly infinite sequence of steps
- can model non-termination
- handy for proof about type soundness (later lecture)

Big-step operational semantics

- more convenient for certain proofs (e.g. compiler correctness, later lecture)

Semantics for SML

- function values modelled as closures (values containing code)
- for ‘real-world’ semantics: best use a specification tool