Backtracking and compilation with continuations
AMP, Spring 2009

Morten Rhiger (mir@itu.dk)
March 9, 2009
Today’s plan

- Backtracking using two continuations
- Compiling with continuations
With exceptions, expressions may generate zero or one result.

With backtracking, expressions may generate zero, one or multiple results.

$E_1 + E_2$ computes the sum of all combinations of results produced by $E_1$ and $E_2$.

For example,

\[
\{1\} + \{7\} = \{8\} \\
\{1\} + \{7, 8\} = \{8, 9\} \\
\{1, 4\} + \{7, 8\} = \{8, 9, 11, 12\} \\
\{\} + \{7, 8\} = \{\}
\]
The “CPS contract” for exceptions

Each expression takes two continuations, a success continuations (succ) and a failure continuation (fail).

- If an expression succeeds (completes normally) with a value \( V \), then it supplies its success continuation with \( V \).
- If an expression fails (completes abruptly), then it invokes its failure continuation (on no arguments).

(If needed, a store is passed around in a single threaded fashion.)

Types:

```ml
type ans
type fail = unit -> ans
type succ = int -> ans
val eval : exp -> ... -> succ -> fail -> ans
```
The “CPS contract” for backtracking

- Each expression takes two continuations, a success continuations (succ) and a failure continuation (fail).
  - If an expression succeeds (completes normally) with a value $V$, then it supplies its success continuation with $V$ and another continuation to invoke when this expression is probed for more results.
  - If an expression fails (completes abruptly), then it invokes its failure continuation (on no arguments).

(If needed, a store is passed around in a single threaded fashion.)

- Types:

  ```ml
  type ans
  type fail = unit -> ans
  type succ = int -> fail -> ans
  val eval : exp -> ... -> succ -> fail -> ans
  ```
A backtracking expression can be supplied with the following initial continuations to collect all results in an ML list:

\[
\text{fun run } e = e \ (\text{fn } v \Rightarrow \text{fn resume } \Rightarrow \ v :: \text{resume } () (\text{fn } () \Rightarrow []))
\]

Backtracking expressions producing zero, one, and two values:

\[
\begin{align*}
\text{fun none succ fail} & = \text{fail } () \\
\text{fun one } v \text{ succ fail} & = \text{succ } v \text{ fail} \\
\text{fun two } (v, w) \text{ succ fail} & = \text{succ } v (\text{fn } () \Rightarrow \text{succ } w \text{ fail})
\end{align*}
\]

Example

\[
\begin{align*}
\text{run } \text{none} & = [] \\
\text{run } (\text{one } 42) & = [42] \\
\text{run } (\text{two } (42, 37)) & = [42, 37]
\end{align*}
\]
Backtracking generator

- A backtracking expression generating a list of numbers
  
  ```ml
  fun to (a, b) succ fail = 
  let fun loop i = 
    if i <= b then 
      succ i (fn () => loop (i + 1)) 
    else 
      fail () 
    in loop a 
  end
  
  Example
  run (to (3,9)) = [3, 4, 5, 6, 7, 8, 9]
  ```
Backtracking arithmetic

An operation adding all combinations of its arguments

```haskell
fun add (g1, g2) succ fail =
  g1 (fn v1 => fn resume =>
    g2 (fn v2 => fn resume =>
      succ (v1 + v2) resume) resume) fail
```

Examples

- `run (add (one 1, one 4)) = [5]`
- `run (add (one 1, two (4, 8))) = [5,9]`
- `run (add (to (11, 13), two (4, 8))) = [15,19,16,20,17,21]`
Languages with backtracking

- Icon (Griswold & Griswold, 1983)
- Prolog
- We explain some aspects of C# generators in terms of backtracking
Backtracking interpreter
(An interpreter for backtracking expressions)

datatype exp
   = VAR of string
   | LET of string * exp * exp
   | INT of int
   | OPR of string * exp list
   | IF of exp * exp * exp

fun eval exp env succ fail = ...
fun eval (INT i) env succ fail = succ i fail

| eval (VAR x) env succ fail = succ (lookup env x) fail

: :
Backtracking interpreter
(Simple arithmetic)

| eval (OPR("+", [e1, e2])) env succ fail |
| eval e1 env (fn v1 => fn resume => |
| eval e2 env (fn v2 => fn resume => |
| succ (v1 + v2) resume) resume) fail |


The expression $E_1 \mid E_2$ produces (all values produced by) $E_1$ first, then (all values produced by) $E_2$.

$E_1 \& E_2$ produces (all values produced by) $E_2$ once foreach value produced by $E_1$. 
Backtracking interpreter
(Let bindings)

\[
\text{eval (LET (x, e1, e2)) env succ fail} = \\
\text{eval e1 env (fn v1 => fn resume =>} \\
\text{eval e2 (extend env x v1) succ resume) fail}
\]

(Let expressions are for loops!)
Backtracking interpreter

(Conditionals)

| eval (IF (e1, e2, e3)) env succ fail
  = eval e1 env
    (fn _ => fn _ => eval e2 env succ fail)
    (fn () => eval e3 env succ fail)

The then branch of an if expression is evaluated if the test produces at least one value. Otherwise the else branch is chosen.
Backtracking interpreter
(Other built-in operators)

\[
\begin{align*}
\text{eval} (\text{OPR} (\text{opr}, \text{es})) & \quad \text{env succ fail} \\
= & \quad \text{let fun eval_all } [] \quad \text{succ fail} \\
= & \quad \text{succ } [] \text{ fail} \\
| \text{eval_all } (e :: \text{es}) \text{ succ fail} \\
= & \quad \text{eval e env (fn } v \Rightarrow \text{ fn resume } \Rightarrow \text{ eval_all es (fn } vs \Rightarrow \text{ fn resume } \Rightarrow \text{ succ (v :: vs resume) resume) fail} \\
\text{in eval_all es (fn } vs \Rightarrow \text{ fn resume } \Rightarrow \text{ apply (opr, vs) succ resume) fail} \\
\end{align*}
\]

We evaluate all arguments, collect the values in a list, and delegate the operation to an auxiliary function apply.
fun apply ("="; [i, j]) succ fail
    = if i = j then succ i fail else fail ()
Backtracking interpreter
(Ranges)

| apply ("to", [i, j]) succ fail =
  let fun loop i
    = if i <= j then
        succ i (fn () => loop (i + 1))
      else
        fail ()
in  loop i
end
fun run e = eval e []
  (fn i => fn resume =>
   (print (Int.fromString i);
    print "\n";
    resume ()
   )
  (fn () =>
   print "no (more) results\n" )
}
Backtracking interpreter
(Examples in concrete syntax)

\[
\begin{align*}
42 & \rightarrow 42 \\
42 \mid 37 & \rightarrow 42, 37 \\
3 \text{ to } 9 & \rightarrow 3, 4, 5, 6, 7, 8, 9 \\
10 \times (3 \text{ to } 9) & \rightarrow 30, 40, 50, 60, 70, 80, 90 \\
(1 \text{ to } 3) \mid (12 \text{ to } 13) & \rightarrow 1, 2, 3, 12, 13 \\
(1 \text{ to } 3) \& (12 \text{ to } 13) & \rightarrow 12, 13, 12, 13, 12, 13 \\
\text{let val } x = 1 \text{ to } 3 \text{ in } 10 \times x \text{ end} & \rightarrow 10, 20, 30
\end{align*}
\]
A lexer and parser for backtracking expressions

- Running under Moscow ML
- Files needed:
  - Parser.grm
  - Lexer.lex
  - Ast.sml
  - Main.sml
  - Icon.sml
  - load
- To compile:
  - mosmlyac Parser.grm
  - mosmllex Lexer.lex
  - mosmlc -c Ast.sml
  - mosmlc -c Parser.sig
  - mosmlc -c Parser.sml
  - mosmlc -c Lexer.sml
  - mosmlc -c Main.sml
  - mosmlc -c Icon.sml
From interpreters to compilers

Start from an interpreter. Distinguish *binding times* (See Sestoft, PLC chapter 14); that is

- “which parts are static” (that is, which parts of the interpreter *can be* evaluated at compile time), from
- “which parts are dynamic” (that is, which parts of the interpreter *must be* evaluated at run time)

CPS compilers handle the generation of dynamic bindings particularly gracefully.
fun eval (INT i) env k = k i
| eval (VAR x) env k = k (lookup env x)
| eval (ADD (e1, e2)) env k
  = eval e1 env (fn v1 =>
    eval e2 env (fn v2 =>
      k (v1 + v2)))
| eval (LET (x, e1, e2)) env k
  = eval e1 env (fn v1 =>
    eval e2 (extend env x v1) k)

fun main e
  = eval e [] (fn v =>
    print (Int.toString v))
Implementing a “generating extension”

- Mark as dynamic the parts of the interpreter that *must be* evaluated at run time. (These includes memory allocations and primitive run-time operations.)
- Propagate this information throughout the interpreter (until a “fixed point” is reached).
- Implement the dynamic parts such that they *generate* code rather than execute code.
- (It may be necessary to mark some seemingly static operations as dynamic. It may also be necessary to slightly modify the interpreter so that as much as possible is marked as static.)
fun eval (INT i)  env k = k i
| eval (VAR x)  env k = k (lookup env x)
| eval (ADD (e1, e2))  env k = eval e1 env (fn v1 =>
  eval e2 env (fn v2 =>
    k (v1 + v2)))
| eval (LET (x, e1, e2)) env k = eval e1 env (fn v1 =>
  eval e2 (extend env x v1) k)

fun main e = eval e [] (fn v =>
  print (Int.toString v))

+ is a dynamic operation. lookup is a static operation that must produce a dynamic value.
Simple expression compiler

Compiling expression to C:

```
fun comp (INT i) env k = k (Int.toString i)
| comp (ADD (e1, e2)) env k
  = comp e1 env (fn c1 =>
                  comp e2 env (fn c2 =>
                                   k (c1 ^ "+" ^ c2)))
```

```
fun main e
  = comp e [] (fn c =>
               "print(" ~ c ~ ");")
```

This compiler simply maps an expression into its C counterpart. It does not reduce the problems of compiling expressions to machine code.
Expression compiler, target language

The target language is a subset of C. A program consists of sequences of statements each being either

- an assignment storing the sum of two simple expressions in a variable:
  
  ```
  int x_n = e_1 + e_2;
  
  ```
  
  or

- a statement printing the value of a simple expression

  ```
  print(e);
  ```

In abstract syntax:

```plaintext
datatype trg_exp = TRG_INT of int 
  | TRG_VAR of string 

datatype trg_stm = TRG_ADD of string * trg_exp * trg_exp
  | TRG_PRINT of trg_exp

datatype trg_cexp = TRG_CEXP of string

fun printc prog = ...
```

Morten Rhiger

Backtracking and compilation with continuations
Compiling expression to C, storing intermediate results in C variables:

- The continuation is passed a trivial C expression (a variable or an integer constant) that will produce the value of the expression being compiled.
- Compilation generates necessary bindings of C variables to (values of) serious expressions.
- We need fresh C variable numbers:
  ```ml
  val counter = ref 0
  fun gensym () =
    let val this = !counter
    in
      counter := this + 1;
      "x" ^ Int.toString this
    end
  ```
“CPS contract” for the expression compiler

- A continuation is applied to a trivial target expression (that is, a constant or a variable). The continuation generates (and returns) code that executes the remaining program.
- C variables needed by (the code generated by) the continuation must be defined “in front”. The application of a continuation may therefore be surrounded by target statements that bind the C variables necessary to evaluate the trivial expression.
- The environment maps source language variables to trivial C expressions (variables or constants).
fun comp (INT i) env k = k (TRG_INT i)
| comp (VAR x) env k = k (lookup env x)
| comp (ADD (e1, e2)) env k = comp e1 env (fn t1 =>
  comp e2 env (fn t2 =>
    let val x = gensym ()
    in TRG_ADD (x, t1, t2) :: k (TRG_VAR x)
    end))
| comp (LET (x, e1, e2)) env k = comp e1 env (fn t =>
  comp e2 (extend env x t) k)

fun main e
  = printc (comp e [] (fn t => [TRG_PRINT t]))
Compiling expressions to C

5

5 + 7

let y = 6 in y + y

5 + (let y = 6 in y + y)

5 + (let y = 6 + 7 in y + y)