Script **generated by TTT**

Title: Petter: Programmiersprachen (05.02.2020)

Date: Wed Feb 05 12:21:46 CET 2020

Duration: 94:14 min

Pages: 32

Stack-Backward Control Flow

Stack Traversal with longjmp

performing control flow jumps across procedure boundaries is the domain of *setjmp/longjmp* (FreeeBSD [4])

setjmp

```
;... signal blocking ...
movq      %rdi,%rcx
movq      0(%rsp),%rdx          ; return address
movq      %rdx, 0(%rcx)
movq      %rbx, 8(%rcx)
movq      %rsp,16(%rcx)
movq      %rbp,24(%rcx)
movq      %r12,32(%rcx)
movq      %r13,40(%rcx)
movq      %r14,48(%rcx)
movq      %r15,56(%rcx)
;... dealing with SSE / FPU
xorq      %rax,%rax
ret
```

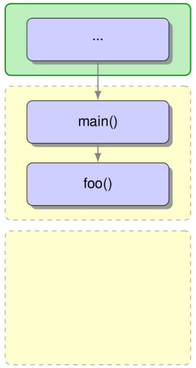
longjmp

```
;... signal blocking / dealing with SSE Registers....
    movq      %rsi,%rax      ; 2nd param -> return value
    movq      0(%rdx),%rcx
    movq      8(%rdx),%rbx
    movq      16(%rdx),%rsp
    movq      24(%rdx),%rbp
    movq      32(%rdx),%r12
    movq      40(%rdx),%r13
    movq      48(%rdx),%r14
    movq      56(%rdx),%r15
    fldcw     64(%rdx)
    testq     %rax,%rax
    jnz       1f
    incq      %rax
1:  movq      %rcx,0(%rsp)   ; setjmp's return address
    ret
```

- control transfer by manipulating stackpointer and instruction pointer

⇝ stack traversal only viable to enclosing stack frames, i.e. up the call hierarchy

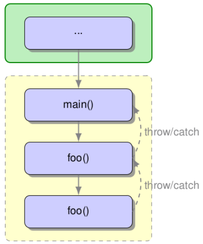
Stack Traversal with longjmp



⚠ heap objects might leak, after discarding several stack frames

Exceptions and Stack Unwinding [3]

```
#include <iostream>
using namespace std;
int foo(int p){
  if (p>3) throw "Error!";
  else return foo(p+1);
}
int main(){
  try {
      return foo(1);
  } catch(const char* s){
      cerr << " Caught\n";
  }
}
```



✓ The compiler appends after the method's body a table of exceptions this method can catch and a cleanup table

1. The unwinder checks for each function in the stack which exceptions can be caught.
   - No catch for exception is found ⇝ `std::terminate`
   - Otherwise, the unwinder restarts on the top of the stack.
2. Again, the unwinder goes through the stack to perform a cleanup for this method. A so called ⇝ *personality routine* will check the cleanup table on the current method.
   - To run cleanup actions, it swaps to the current stack frame. This will run the destructor for each object allocated at the current scope.
   - Reaching the frame in the stack that can handle the exception, the unwinder jumps into the proper catch statement.

Same-Level Control Flow

Stack Switching with makecontext and swapcontext [9]

makecontext

```
void makecontext(ucontext_t *ucp,
                 void (*func)(), int argc, ...);
```

- For preparation, the caller must
  - obtained a fresh context from a call to `getcontext()`
  - allocate a new stack for this context and assign its address to `ucp->uc_stack`
  - define a successor context and assign its address to `ucp->uc_link`
- `makecontext()` modifies the context pointed to by `ucp`
- On activation (using `swapcontext()`) the function `func` is called, and passed the `argc` many arguments of `int` type.
- When `func` returns, the successor context is activated. If the successor context pointer is NULL, the thread exits.

swapcontext

```
int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);
```

- `swapcontext()` saves the current context in `oucp`, and then activates `ucp`.

⇝ When successful, `swapcontext()` does not return. (But we may return later, in case `oucp` is activated, in which case it looks like `swapcontext()` returns 0.) On error, swapcontext() returns -1.

Stack Switching with makecontext and swapcontext

interleaved functions

```
#include <ucontext.h>
#include <stdio.h>
#include <stdlib.h>
static ucontext_t ctx_m, ctx_f1, ctx_f2;
#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

static void f1(void) {
  printf("f1: started\n");
  printf("f1  --swapcontext--> f2\n");
  if (swapcontext(&ctx_f1, &ctx_f2) == -1) handle_error("swap");
  printf("f1: returning\n");
}
static void f2(void) {
  printf("f2: started\n");
  printf("f2  --swapcontext--> f1\n");
  if (swapcontext(&ctx_f2, &ctx_f1) == -1) handle_error("swap");
  printf("f2: returning\n");
}
```

startup platform

```
int main(int argc, char *argv[]) {
   char f1_stack[16384];
   char f2_stack[16384];
   if (getcontext(&ctx_f1) == -1) handle_error("getcontext");
   ctx_f1.uc_stack.ss_sp = f1_stack;
   ctx_f1.uc_stack.ss_size = sizeof(f1_stack);
   ctx_f1.uc_link = &ctx_m;
   makecontext(&ctx_f1, f1, 0);
   if (getcontext(&ctx_f2) == -1) handle_error("getcontext");
   ctx_f2.uc_stack.ss_sp = f2_stack;
   ctx_f2.uc_stack.ss_size = sizeof(f2_stack);
   /* f2's successor context is f1(), unless argc > 1 */
   ctx_f2.uc_link = (argc > 1) ? NULL : &ctx_f1;
   makecontext(&ctx_f2, f2, 0);
   printf("main  --swapcontext--> f2\n");
   if (swapcontext(&ctx_m, &ctx_f2) == -1) handle_error("swap");
   printf("main: exiting\n");
   exit(EXIT_SUCCESS);
}
```

Stack Switching with makecontext and swapcontext

interleaved functions

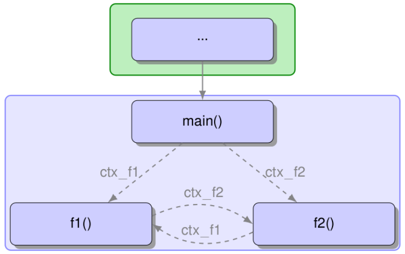
```c
#include <ucontext.h>
#include <stdio.h>
#include <stdlib.h>
static ucontext_t ctx_m, ctx_f1, ctx_f2;
#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

static void f1(void) {
  printf("f1: started\n");
  printf("f1  --swapcontext--> f2\n");
  if (swapcontext(&ctx_f1, &ctx_f2) == -1) handle_error("swap");
  printf("f1: returning\n");
}
static void f2(void) {
  printf("f2: started\n");
  printf("f2  --swapcontext--> f1\n");
  if (swapcontext(&ctx_f2, &ctx_f1) == -1) handle_error("swap");
  printf("f2: returning\n");
}
```

startup platform

```c
int main(int argc, char *argv[]) {
   char f1_stack[16384];
   char f2_stack[16384];
   if (getcontext(&ctx_f1) == -1) handle_error("getcontext");
   ctx_f1.uc_stack.ss_sp = f1_stack;
   ctx_f1.uc_stack.ss_size = sizeof(f1_stack);
   ctx_f1.uc_link = &ctx_m;
   makecontext(&ctx_f1, f1, 0);
   if (getcontext(&ctx_f2) == -1) handle_error("getcontext");
   ctx_f2.uc_stack.ss_sp = f2_stack;
   ctx_f2.uc_stack.ss_size = sizeof(f2_stack);
   /* f2's successor context is f1(), unless argc > 1 */
   ctx_f2.uc_link = (argc > 1) ? NULL : &ctx_f1;
   makecontext(&ctx_f2, f2, 0);
   printf("main  --swapcontext--> f2\n");
   if (swapcontext(&ctx_m, &ctx_f2) == -1) handle_error("swap");
   printf("main: exiting\n");
   exit(EXIT_SUCCESS);
}
```

```
main --swapcontext--> f2
f2: started
f2  --swapcontext--> f1
f1: started
f1  --swapcontext--> f2
f2: returning
f1: returning
main: exiting
```

Stack Switching with makecontext and swapcontext



⚠ stack frame for subcontext

- size has to be known
- has to be allocated manually
- has to be allocated by parent frame

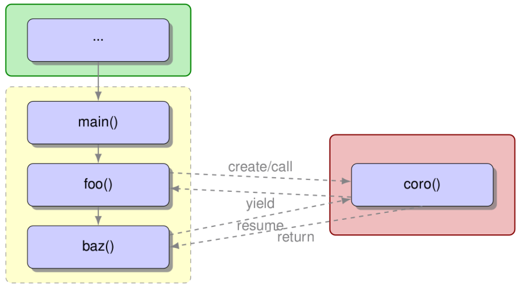
⚠ scheduling on termination depending on definition of a successor context

Stackless Coroutines

EcmaScript 6+:

```javascript
var genFn = function*(){
  var i = 0;
  while(true){
    yield i++;
  }
};
var gen = genFn();
while (true){
  var result = gen.next().value;
}
```

Stackless Coroutines

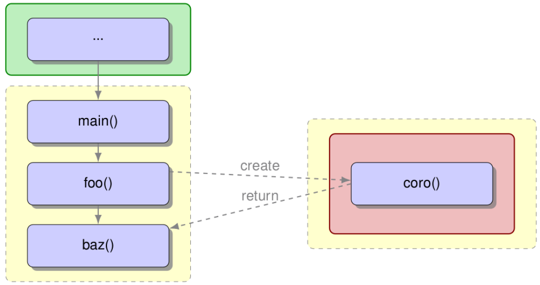


Stackful Coroutines

Lua:

```
function send (x)
  coroutine.yield(x)
end

local producer = coroutine.create(
  function ()
    while true do
      send(io.read())
    end
  end)
```

Stackful Coroutines



Abstracting Contexts

```
3 + 5*2 -1
3 + [5*2] -1
3 + [.] -1
```

- isolate the *context* of an expression within surrounding expression, i.e. `5*2`
- make the *context* a first level language construct

⇝ Continuations (Reynolds 1993[7])

Their counterpart

- is represented by already computed subexpressions
- is applicable to Continuations, yielding the final result

⇝ *Suspended Computations*

Continuation Passing Style (CPS) [8]

Transforming a function `f::a->b` into a CPS function `f'::a->((b->c)->c)` : `f'(k)`

- computes `f(k)` using only CPS styled functions and
- returns a function which, given a continuation `cont::b->c` returns `cont(f(k))`.

⇝ *suspended computation (*`::(b->c)->c`*)*

Direct style

```
square :: Int -> Int
square x = x * x

add :: Int -> Int -> Int
add x y = x + y

pythagoras :: Int -> Int -> Int
pythagoras x y = add (square x) (square y)
```

⇒

Continuation Passing Style

```
square_cps :: Int -> ((Int -> r) -> r)
square_cps x = \k -> k ((*) x x)

add_cps :: Int -> Int -> ((Int -> r) -> r)
add_cps x y = \k -> k ((+) x y)

pyth_cps :: Int -> Int -> ((Int -> r) -> r)
pyth_cps x y = \k ->
  square_cps x (\x_squared ->
   square_cps y (\y_squared ->
    add_cps x_squared y_squared (k)))
```

Continuation Passing Style (CPS)

Higher order functions, that receive CPS styled functions as parameters

Direct style

```
trip :: (a -> a) -> a -> a
trip f x = f (f (f x))
```

⇒

Continuation Passing Style

```
trip_cps :: (a -> ((a -> r) -> r)) -> a -> ((a -> r) -> r)
trip_cps f_cps x = \k ->
  f_cps x (\fx ->
    f_cps fx (\ffx ->
      f_cps ffx (k)))
```

Function Parameter Signature:
`(a->b)`

CPS Function Parameter Signature:
`(a->((b->r)->r))`

Depending on how you where raised as a programmer (⇝ *functional* vs. *iterative*), this might look horrible to you – ⚠ is it even efficient at all?

Continuation Passing Style (CPS) [8]

Transforming a function `f::a->b` into a CPS function `f'::a->((b->c)->c)`: `f'(k)`

- computes `f(k)` using only CPS styled functions and
- returns a function which, given a continuation `cont::b->c` returns `cont(f(k))`.

⇝ *suspended computation (`::(b->c)->c`)*

Direct style

```
square :: Int -> Int
square x = x * x

add :: Int -> Int -> Int
add x y = x + y

pythagoras :: Int -> Int -> Int
pythagoras x y = add (square x) (square y)
```

⇒

Continuation Passing Style

```
square_cps :: Int -> ((Int -> r) -> r)
square_cps x = \k -> k ((*) x x)

add_cps :: Int -> Int -> ((Int -> r) -> r)
add_cps x y = \k -> k ((+) x y)

pyth_cps :: Int -> Int -> ((Int -> r) -> r)
pyth_cps x y = \k ->
  square_cps x (\x_squared ->
    square_cps y (\y_squared ->
      add_cps x_squared y_squared (k)))
```


Continuation Passing Style (CPS)

Higher order functions, that receive CPS styled functions as parameters

Direct style

```
trip :: (a -> a) -> a -> a
trip f x = f (f (f x))
```

⇒

Continuation Passing Style

```
trip_cps :: (a -> ((a -> r) -> r)) -> a -> ((a -> r) -> r)
trip_cps f_cps x = \k ->
  f_cps x (\fx ->
    f_cps fx (\ffx ->
      f_cps ffx (k)))
```

Function Parameter Signature:
`(a->b)`

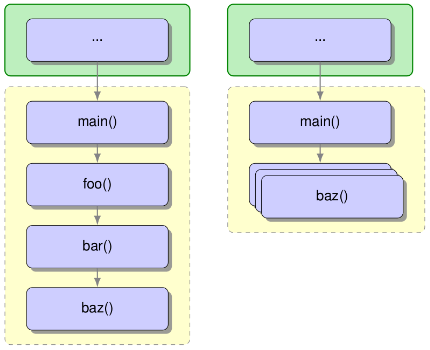
CPS Function Parameter Signature:
`(a->((b->r)->r))`

Depending on how you where raised as a programmer (⇝ *functional* vs. *iterative*), this might look horrible to you – ⚠ is it even efficient at all?

Tail Call Optimization

Steele 1977 [6]

```
main :: IO
main = do
  print (foo(5))

foo :: Int -> Int
foo f = bar(10)

bar :: Int -> Int
bar b = baz(100)

baz :: Int -> Int
baz z = 100 + 100
```



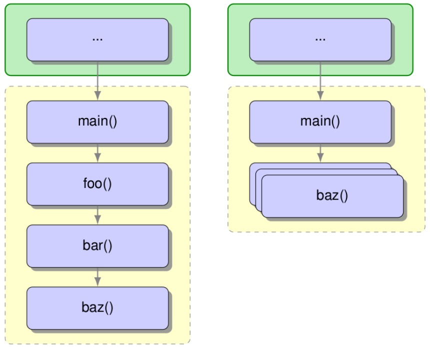

```
main :: IO
main = do
  print (foo(5))

foo :: Int -> Int
foo f = 100+100
```

- Potentially generate new closure
- Reuse the existing stackframe
- Potentially shift actual parameters on stack
- *Jump* to called function

Tail Call Optimization

Steele 1977 [6]

```
main :: IO
main = do
  print (foo(5))
foo :: Int -> Int
foo f = bar(10)

bar :: Int -> Int
bar b = baz(100)

baz :: Int -> Int
baz z = 100 + 100
```




```
main :: IO
main = do
  print (foo(5))
foo :: Int -> Int
foo f = 100+100
```

- Potentially generate new closure
- Reuse the existing stackframe
- Potentially shift actual parameters on stack
- *Jump* to called function

Composing Code by Continuations

Provide a function `compose`, that

- takes a suspended computation `s::(a->r)->r`
- takes a function in CPS style `f::a->((b->r)->r)`
- returns a composition of `f` to `s`, in form of another suspended computation `::(b->r)->r`

applying a CPS function to a *suspended computation*

```
compose :: ((a -> r) -> r) -> (a -> ((b -> r) -> r)) -> ((b -> r) -> r)
compose s f = \k -> s (\x -> f x (k))
```

The Cont Type Constructor

Data Constructor `Cont` represents suspended computations as a polymorphic Haskell data type, along with the functions:

- ⇝ `cont :: ((a -> r) -> r) -> Cont r a` creating a suspended computation
- ⇝ `runCont :: Cont r a -> (a -> r) -> r` computes the suspended computation with a given final function

Step by step introduce `Cont` into `compose`

```
compose' :: Cont r a -> (a -> Cont r b) -> Cont r b
compose' s f = cont (\k -> runCont s (\x -> runCont (f x) (k)))
```

⇕

Monadic bind: `(>>=) :: Monad m => m a -> (a -> m b) -> m b`

Excursion: Monads

Essentials of Monads (Wadler 92 [10])

A monad is a ⇝ *type class* for arbitrary type constructors, defining at least a function called return, and a combinator function called bind or >>=

```
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

Syntactic sugar: *do-notation*: allows to write monadic computations in a pseudo-imperative style

```
mothersPaternalGrandfather s =
 mother s >>= (\m ->
  father m >>= (\gf ->
   father gf))
```

⇒

```
mothersPaternalGrandfather s = do
  m  <- mother s
  gf <- father m
  father gf
```

Continuation Passing Style Monad

the Cont Monad

```
instance Monad (Cont r) where
    return x = cont (\k -> k x)
    s >>= f  = cont (\k -> runCont s (\x -> runCont (f x) k))
```

Continuation Passing Style

```
add_cps :: Int -> Int -> ((Int -> r) -> r)
add_cps x y = \k -> k (add x y)

square_cps :: Int -> ((Int -> r) -> r)
square_cps x = \k -> k (square x)

pyth_cps :: Int -> Int -> ((Int -> r) -> r)
pyth_cps x y = \k ->
 square_cps x (\x_squared ->
   square_cps y (\y_squared ->
     add_cps x_squared y_squared (k)))
```

Continuation Passing Style Monad

the Cont Monad

```
instance Monad (Cont r) where
    return x = cont (\k -> k x)
    s >>= f  = cont (\k -> runCont s (\x -> runCont (f x) k))
```

Continuation Passing Style

```
add_cps :: Int -> Int -> ((Int -> r) -> r)
add_cps x y = \k -> k (add x y)

square_cps :: Int -> ((Int -> r) -> r)
square_cps x = \k -> k (square x)

pyth_cps :: Int -> Int -> ((Int -> r) -> r)
pyth_cps x y = \k ->
 square_cps x (\x_squared ->
   square_cps y (\y_squared ->
     add_cps x_squared y_squared (k)))
```

Cont Monad Style

```
add_cont :: Int -> Int -> Cont r Int
add_cont x y = return (add x y)

square_cont :: Int -> Cont r Int
square_cont x = return (square x)

pythagoras_cont :: Int -> Int -> Cont r Int
pythagoras_cont x y = do
    x_squared <- square_cont x
    y_squared <- square_cont y
    add_cont x_squared y_squared
```

Call with Current Continuation

First implementation in Scheme

> *call/cc takes as an argument an abstraction and passes to the abstraction another abstraction, that takes the role of a continuation. When this continuation abstraction is applied, it sends its argument to the continuation of the call/cc.*

Clinger et. al 1986[2]

callcc in CPS

```
callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a
callCC f = cont (\h -> runCont (f (\a -> cont (\_ -> h a)) h ))

callCC' :: ((a->((b->r)->r)) -> ((a->r)->r)) -> ((a->r)->r)
callCC' f = (\h ->
  f (\a -> (\_ -> h a)) h
  )
```

Example: Control Structures with Call/CC

Loops with callcc

```
import Control.Monad.Trans.Class
import Control.Monad.Trans.Cont

main = flip runContT return $ do
  lift $ putStrLn "A"
  (k, num) <- callCC ( \c -> let f x = c (f, x)
                             in return (f, 0) )

  lift $ putStrLn "B"
  lift $ putStrLn "C"

  if num < 5
      then k (num + 1) >> return ()
      else lift $ print num
```

- Getting access to continuations may need a little monad trickery (⇝ lifting to Cont Monad)
- callCC now grants access to continuations

Example: Control Structures with Call/CC

Loops with callcc

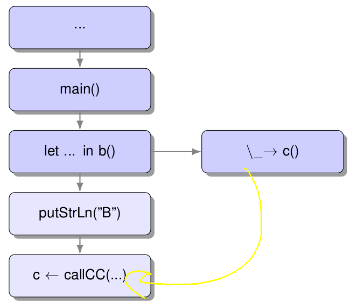
```
import Control.Monad.Trans.Class
import Control.Monad.Trans.Cont

main = flip runContT return $ do
  lift $ putStrLn "A"
  (k, num) <- callCC ( \c -> let f x = c (f, x)
                              in return (f, 0) )


  lift $ putStrLn "B"
  lift $ putStrLn "C"

  if num < 5
      then k (num + 1) >> return ()
      else lift $ print num
```

```
A
B
C
B
C
B
C
B
C
B
C
B
C
5
```

- Getting access to continuations may need a little monad trickery (⇝ lifting to Cont Monad)
- callCC now grants access to continuations
- Continuations in Haskell via callCC are *Multi-Shot Continuations*

Implementation of Continuations [5]

```
main = flip runContT return $ do
  lift $ putStrLn "A"
  c <- callCC ( \k ->
        let f _ = k (f)
        in return (f)  )
  lift $ putStrLn "B"
  let b = \_ -> c() in b ()
```



- Continuations, returned from callcc may *escape the current context*/function frame
- calling continuations restarts execution at the original callcc site and function frame
- *Multi-Shot Continuations* may return to the same callcc site multiple times
- ⚠ traditional stack based frame management discards and overwrites old function frames

Roundup

Applications of call/cc

- Standard Control Structures
- Exception Handling
- Coroutines
- Backtracking
- ...

Lessons Learned

1. Simple Gotos
2. Longjumps
3. Set-/Swapcontext
4. Exception Handling
5. Stackful/-less Coroutines
6. Single-/Multishot Continuations

Further Topics

```
s = \f -> (\g -> (\x -> f x (g x)))
k = \x -> (\y -> x)
i = \x -> x
```

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

[1] K. Asai and O. Kiselyov.
Introduction to programming with shift and reset.
In *ACM SIGPLAN Continuation Workshop*, 2011.

[2] W. Clinger, D. P. Friedman, and M. Wand.
A Scheme for a Higher-Level Semantic Algebra, page 237–250.
Cambridge University Press, USA, 1986.

[3] Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI.
Itanium C++ ABI: Exception Handling.
`https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html`.

[4] FreeBSD.
setjmp implementation.
`https://github.com/freebsd/freebsd/blob/master/lib/libc/amd64/gen/setjmp.S`.

[5] R. Hieb, R. K. Dybvig, and C. Bruggeman.
Representing control in the presence of first-class continuations.
In B. N. Fischer, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 66–77. ACM, 1990.

[6] G. L. S. Jr.
Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, LAMBDA: the ultimate GOTO.
In J. S. Ketchel, H. Z. Kriloff, H. B. Burner, P. E. Crockett, R. G. Herriot, G. B. Houston, and C. S. Kitto, editors, *Proceedings of the 1977 annual conference, ACM '77, Seattle, Washington, USA, October 16-19, 1977*, pages 153–162. ACM, 1977.

[7] J. C. Reynolds.
The discoveries of continuations.
Lisp and Symbolic Computation, 6(3-4):233–248, 1993.

[8] G. J. Sussman and G. L. Steele Jr.
Ai memo no. 349 december 1975.
contract, 14(75-C):0643.
`http://www.laputan.org/pub/papers/aim-349.pdf`.

[9] The IEEE and The Open Group.
The Open Group Base Specifications Issue 6 – IEEE Std 1003.1, 2004 Edition.
IEEE, New York, NY, USA, 2004.
`https://pubs.opengroup.org/onlinepubs/009695399/functions/makecontext.html`.

[10] P. Wadler.
The essence of functional programming.
In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, page 1–14, New York, NY, USA, 1992. Association for Computing Machinery.