TIL: A Type-Directed Optimizing Compiler for ML

Stefan Knudsen

McGill University

stefan.knudsen@mail.mcgill.ca

November 10, 2016

Motivation

SML is a functional programming language like OCaml, F♯, and Haskell

```
val map : ('a -> 'b) -> ['a] -> ['b]
fun map f ls =
case ls of
[] => []
| x :: xs = f x :: map f xs ;

map (fn x => x+1) [1,2,3];
-> [2,3,4];
```

Problem

To handle type variables, SML compilers tend to use a universal representation for values whose type they can't infer.

- Objects small enough - put in a tagged machine word
- Objects too large - tagged object on the heap referenced by a pointer

For garbage collection, SML uses tags to keep information about the type of the object (number, pointer, closure, $\cdots$).
If we can't be sure of the type, this seems reasonable.
Do we have to incur this cost otherwise? No.

Typed Intermediate Language

This paper introduces an approach to compiling functional languages using:

- Conventional functional language optimization
- Loop optimization
- Intensional polymorphism
- Nearly tag-free garbage collection

Typed Intermediate Language

This approach performs worse than before when we can't infer types :

- At run time, we have to construct types and pass them to polymorphic functions

It's good to transform as many polymorphic and higher-order functions as possible

- Can be done with uncurrying and inlining

Nearly tag-free garbage collection

Record representation information at compile time

When garbage collecting, use this to decide if values are pointers

Question 1: Why nearly?

No tags for values in registers, in data structures, or on the stack

There are still tags on heap-allocated data structures

At compile time, a table is made to keep track of the layout of registers, stack frames, and their corresponding call sites

The table is checked before garbage collection

Use intensional polymorphism to make tags when needed

Intensional Polymorphism

With intensional polymorphism, you can treat types like values

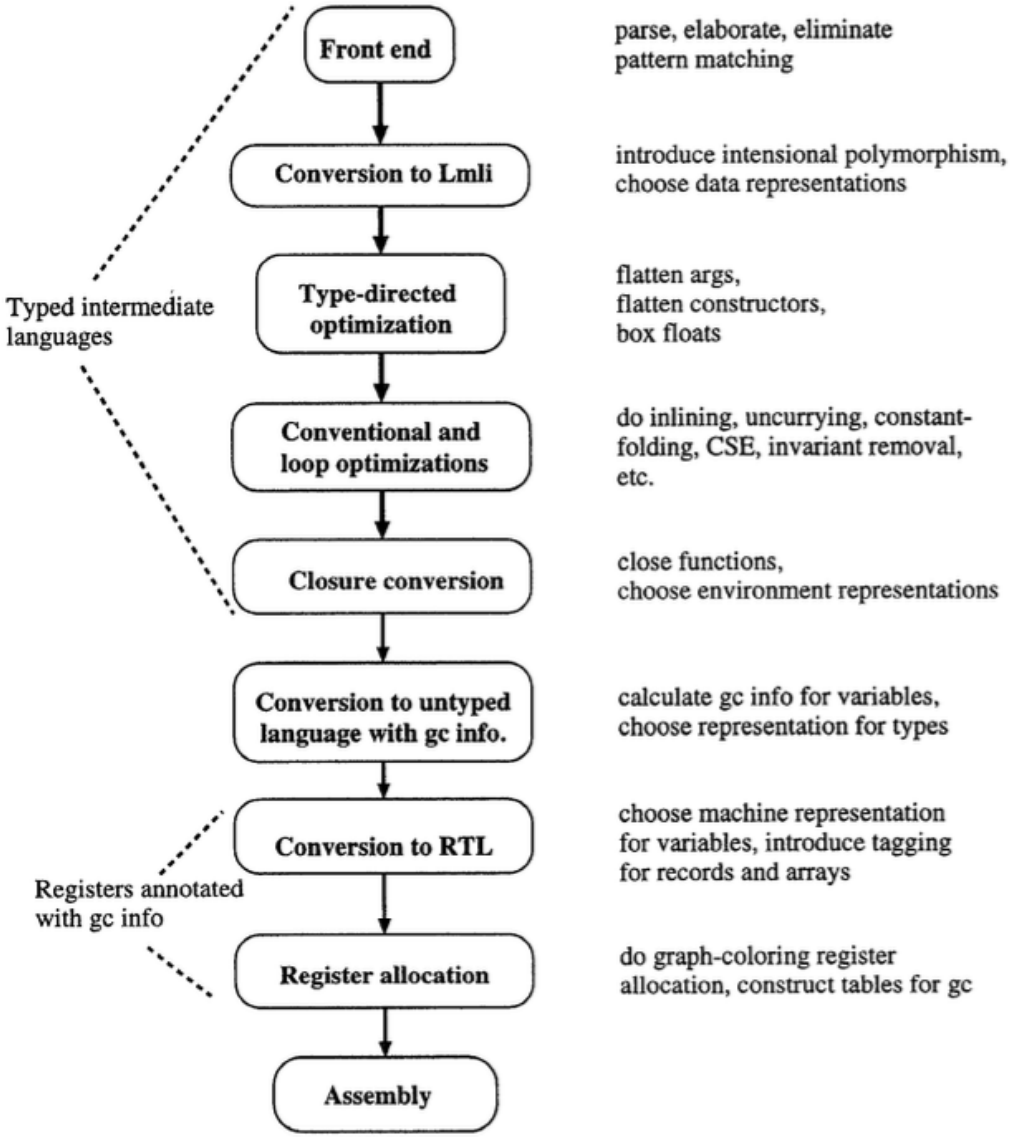
```
fun sub['a](x:'a array, i:int) =
typecase 'a of
int => intsub(x,i)
| float => floatsub(x,i)
| ptr('t) => ptrsub(x,i)
;
```

Polymorphic functions are big and slow, since types are made and passed at run time

For the type information to be helpful here or for GC, types are required at each level of the compiler

Thus the name TIL - Typed Intermediate Languages

Compilation Phases



Compilation Phases

Front end $\Rightarrow$ Lmli $\Rightarrow$ Optimizations $\Rightarrow \cdots$

Using ML Kit, compiles to Lambda, a explicitly-typed core language

Then to Lmli (λ_i^{ML}), an intensionally polymorphic language.

Compilation Phases

Front end $\Rightarrow$ Lmli $\Rightarrow$ Optimizations $\Rightarrow \cdots$

Using ML Kit, compiles to Lambda, a explicitly-typed core language

Then to Lmli (λ_i^{ML}), an intensionally polymorphic language.

Compilation Phases

Front end $\Rightarrow$ Lmli $\Rightarrow$ Optimizations $\Rightarrow \cdots$

Using ML Kit, compiles to Lambda, a explicitly-typed core language

Then to Lmli (λ_i^{ML}), an intensionally polymorphic language.

Optimizations

Alpha-conversion

Rename all variables so that they're unique

Dead-code elimination

Remove unevaluated code

Constant folding

Arithmetic operations, cases, and record projections are simplified where possible

Sinking

Pure expressions used only once in a case analysis are moved down

Inlining switch continuations

let x = **if** y **then** e2 **else raise** e3 **in** e4 **end**

$\Longrightarrow$

if y **then let** x = e2 **in** e4 **end else raise** e3

This helps for optimizations like CSE

Optimizations

Uncurrying

Unlike in Haskell, it's much better to uncurry functions

$\alpha \to \beta \to \alpha$ becomes $\alpha * \beta \to \alpha$

f x y = x becomes f(x,y) = x

Inlining

For functions called only once, we replace the call with the function body

Minimizing fix

Mutually-recursive functions are grouped into strongly connected sets. Question 2: Why?

By separating recursive and non-recursive functions, it helps for the inlined and dead code elimination optimizations

Loop Optimizations

Common subexpression elimination if e1 is pure,

```
let x = e1 in e2 end
⟹ [e1/x]e2
```

Eliminating redundant switches

```
let x = if z then
let y = if z then e1 else e2 in ...
⟹
let x = if z then
let y = e1 in ...
```

Eliminating redundant comparisons

If $x < y$ and $y < z$, we don't need to check if $x < z$

Hoisting

Constant expressions are hoisted to the top of the program

Compilation Phases

$\cdots \Rightarrow$ Closure conversion $\Rightarrow$ Ubform $\Rightarrow$ RTL $\Rightarrow$ Assembly

Converts Lmli-Bform to Lmli-Closure, built around closures and environments

Converts to an untyped language, typecase $\Rightarrow$ switch

Converts to a register-transfer language which has unlimited registers

Graph coloring for register allocation

Compilation Phases

$\cdots \Rightarrow$ Closure conversion $\Rightarrow$ Ubform $\Rightarrow$ RTL $\Rightarrow$ Assembly

Converts Lmli-Bform to Lmli-Closure, built around closures and environments

Converts to an untyped language, typecase $\Rightarrow$ switch

Converts to a register-transfer language which has unlimited registers

Graph coloring for register allocation

Compilation Phases

$\cdots \Rightarrow$ Closure conversion $\Rightarrow$ Ubform $\Rightarrow$ RTL $\Rightarrow$ Assembly

Converts Lmli-Bform to Lmli-Closure, built around closures and environments

Converts to an untyped language, typecase $\Rightarrow$ switch

Converts to a register-transfer language which has unlimited registers

Graph coloring for register allocation

Compilation Phases

$\cdots \Rightarrow$ Closure conversion $\Rightarrow$ Ubform $\Rightarrow$ RTL $\Rightarrow$ Assembly

Converts Lmli-Bform to Lmli-Closure, built around closures and environments

Converts to an untyped language, typecase $\Rightarrow$ switch

Converts to a register-transfer language which has unlimited registers

Graph coloring for register allocation

An Example

```
fun dot(cnt,sum) =
if cnt < bound then
let val sum' = sum +
sub2(A,i,cnt) * sub2(B,cnt,j)
in dot(cnt+1,sum')
else sum
```

An Example

```
val sub2 : 'a array2 * int * int -> 'a

fun sub2({columns,rows,v}, s:int, t:int) =
if s < 0
orelse s >= rows
orelse t < 0
orelse t >= columns 0
then raise Subscript
else unsafe_sub1(v, s * columns + t)
```

A few clarifications (post presentation)

The previous program dots two vectors, as in the inner loop of a matrix multiplication

unsafe_sub1(v, s $*$ columns + t) is an access to matrix v stored in memory at position (s,t)

In the next slides, Λt is a function parametrized by a type (intensional polymorphism)

There's no tagging in this example

```
sub2 =
   let fix f = Λty.
        let fix g = λarg.
            let a = (#0 arg)
                s = (#1 arg)
                t = (#2 arg)
                columns = (#0 a)
                rows = (#1 a)
                v = (#2 a)
                check =
                  let test1 = plst_i(s,0)
                  in Switch_enum test1 of
                       1 => λ.enum(1)
                     | 0 => λ.
                        let test2 = pgte_i(s,rows)
                        in Switch_enum test2 of
                             1 => λ.enum(1)
                           | 0 => λ.
                             let test3 = plst_i(t,0)
                             in  Switch_enum test3 of
                                   1 => λ.enum(1)
                                 | 0 => λ.pgte_i(t,columns)
                             end
                        end
                  end
            in Switch_enum check of
                   1 => λ.raise Subscript
                 | 0 => λ.unsafe_sub1 [ty] {v,t + s * columns}
            end
        in g
   in f
   end

fix dot=
    λi.let cnt = (#0 i)
           sum = (#1 i)
           d = plst_i(cnt,bound)
        in  Switch_enum d
            of 1 => λ.let sum' = sum +
                           ((sub2 [int]) {A,i,cnt}) *
                          ((sub2 [int]) {B,cnt,j})
                       in  dot{cnt+1,sum'}
                       end
             | 0 => λ.sum
        end
```

```
sub2 = ...
fix dot = λcnt,sum.
        let test = plst_i(cnt, bound)
            r =
             Switch_enum test of
                 1 => λ.
                  let a = sub2[int]
                      b = a(A,i,cnt)
                      c = sub2[int]
                      d = c(B,cnt,j)
                      e = b*d
                      f = sum+e
                      g = cnt+1
                      h = dot(g,f)
                  in h
                  end
               | 0 => λ.sum
        in  r
        end
```

```
sub2 = ...
fix dot = λcnt,sum.
          let test = plst_i(cnt, bound)
              r =
               Switch_enum test of
                   1 => λ.
                    let a = sub2[int]
                        b = a(A,i,cnt)
                        c = sub2[int]
                        d = c(B,cnt,j)
                        e = b*d
                        f = sum+e
                        g = cnt+1
                        h = dot(g,f)
                    in h
                    end
                 | 0 => λ.sum
          in  r
          end
```

Figure 3: Lmli-Bform before optimization

```
fix dot =
     λcnt,sum.
      let test = plst_i(cnt,bound)
          r = Switch_enum test of
                 1 =>
                 λ.
                  let a = t1 + cnt
                      b = psub_ai(av,a)
                      c = columns * cnt
                      d = j + c
                      e = psub_ai(bv,d)
                      f = b*e
                      g = sum+f
                      h = 1+cnt
                      i = dot(h,g)
                  in  i
                  end
              |  0 => λ.sum
     in  r
     end
```

Figure 4: Lmli-Bform after optimization

```
fix dot =
     λcnt,sum.
      let test = plst_i(cnt,bound)
          r = Switch_enum test of
                 1 =>
                 λ.
                  let a = t1 + cnt
                      b = psub_ai(av,a)
                      c = columns * cnt
                      d = j + c
                      e = psub_ai(bv,d)
                      f = b*e
                      g = sum+f
                      h = 1+cnt
                      i = dot(h,g)
                  in  i
                  end
               |  0 => λ.sum
      in  r
      end
```

Figure 4: Lmli-Bform after optimization

```
fix dot =
 λbound:INT,columns:INT,bv:TRACE,av:TRACE,t1:INT,
   j:INT,cnt:INT,sum:INT.
          let test:INT = pgtt_i(bound,cnt)
              r:INT =
                Switchint test of
                  1 =>
                  let a:INT = t1 + cnt
                      b:INT = psub_ai(av,a)
                      c:INT = columns * cnt
                      d:INT = j + c
                      e:INT = psub_ai(bv,d)
                      f:INT = b*e
                      g:INT = sum+f
                      h:INT = 1+cnt
                      i:INT = dot(bound,columns,bv,
                                  av,t1,j,h,g)
                  in i
                  end
                | 0 => sum
          in  r
          end : INT
```

Figure 5: After conversion to Ubform

```
fix dot =
 λbound:INT,columns:INT,bv:TRACE,av:TRACE,t1:INT,
    j:INT,cnt:INT,sum:INT.
          let test:INT = pgtt_i(bound,cnt)
              r:INT =
                Switchint test of
                  1 =>
                  let a:INT = t1 + cnt
                      b:INT = psub_ai(av,a)
                      c:INT = columns * cnt
                      d:INT = j + c
                      e:INT = psub_ai(bv,d)
                      f:INT = b*e
                      g:INT = sum+f
                      h:INT = 1+cnt
                      i:INT = dot(bound,columns,bv,
                                  av,t1,j,h,g)
                  in i
                  end
                | 0 => sum
          in  r
          end : INT
```

Figure 5: After conversion to Ubform

```
dot(([bound(INT),columns(INT),bv(TRACE),
    av(TRACE),t1(INT),j(INT),cnt(INT),
    sum(INT)],[]))
{ L0:  pgt bound (INT) , cnt(INT) , test(INT)
       bne test(INT),L1
       mv sum(INT),result (INT)
       br L2
  L1:  addl t1(INT) , cnt(INT) , a(INT)
(*)    s4add a(INT) , av(TRACE) , t2(LOCATIVE)
(*)    ldl b(INT) , 0(t2(LOCATIVE))
       mull columns(INT) , cnt(INT) , c (INT)
       addl j(INT) , c(INT) , d (INT)
(*)    s4add d (INT) , bv(TRACE) , t3(LOCATIVE)
(*)    ldl e (INT) , 0(t3 (LOCATIVE))
       mull/v b (INT) , e (INT) , f(INT)
       addl/v sum(INT) , f (INT) , g (INT)
       addl/v cnt(INT) , 1 , h (INT)
       trapb
       mv h (INT),cnt(INT)
       mv g (INT),sum(INT)
       br L0
  L2:  return retreg(LABEL) }
```

Figure 6: After conversion to RTL

```
dot(([bound(INT),columns(INT),bv(TRACE),
    av(TRACE),t1(INT),j(INT),cnt(INT),
    sum(INT)],[]))
{ L0:  pgt bound (INT) , cnt(INT) , test(INT)
       bne test(INT),L1
       mv sum(INT),result (INT)
       br L2
  L1:  addl t1(INT) , cnt(INT) , a(INT)
(*)    s4add a(INT) , av(TRACE) , t2(LOCATIVE)
(*)    ldl b(INT) , 0(t2(LOCATIVE))
       mull columns(INT) , cnt(INT) , c (INT)
       addl j(INT) , c(INT) , d (INT)
(*)    s4add d (INT) , bv(TRACE) , t3(LOCATIVE)
(*)    ldl e (INT) , 0(t3 (LOCATIVE))
       mull/v b (INT) , e (INT) , f(INT)
       addl/v sum(INT) , f (INT) , g (INT)
       addl/v cnt(INT) , 1 , h (INT)
       trapb
       mv h (INT),cnt(INT)
       mv g (INT),sum(INT)
       br L0
  L2:  return retreg(LABEL) }
```

Figure 6: After conversion to RTL

```
        .ent Lv2851_dot_205955
 # arguments : [$bound,$0] [$columns,$1] [$bv,$2]
 #             [$av,$3] [$t1,$4] [$j,$5]
 #             [$cnt,$6] [$sum,$7]
 # results     : [$result,$0]
 # return addr  : [$retreg,$26]
 # destroys    : $0 $1 $2 $3 $4 $5 $6 $7 $27
Lv2851_dot_205955:
        .mask (1 << 26), -32
        .frame $sp, 32, $26
        .prologue 1
        ldgp        $gp, ($27)
        lda        $sp, -32($sp)
        stq        $26, ($sp)
        stq        $8, 8($sp)
        stq        $9, 16($sp)
        mov        $26, $27
L1:
        cmplt        $6, $0, $8
        bne        $8, L2
        mov        $7, $1
        br        $31, L3
L2:
        addl        $4, $6, $8
        s4addl        $8, $3, $8
        ldl        $8, ($8)
        mull        $1, $6, $9
        addl        $5, $9, $9
        s4addl        $9, $2, $9
        ldl        $9, ($9)
        mullv        $8, $9, $8
        addlv        $7, $8, $7
        addlv        $6, 1, $6
        trapb
        br        $31, L1
L3:
        mov        $1, $0
        mov        $27, $26
        ldq        $8, 8($sp)
        ldq        $9, 16($sp)
        lda        $sp, 32($sp)
        ret        $31, ($26), 1
        .end Lv2851_dot_205955
```

Figure 7: Actual DEC ALPHA assembly language

Results

Program	lines	Description
Checksum	241	Checksum fragment from the Foxnet [7], doing 5000 checksums on a 4096-byte array.
FFT	246	Fast fourier transform, multiplying polynomials up to degree 65,536
Knuth-Bendix	618	An implementation of the Knuth-Bendix completion algorithm.
Lexgen	1123	A lexical-analyzer generator [6], processing the lexical description of Standard ML.
Life	146	The game of Life implemented using lists [39].
Matmult	62	Integer matrix multiply, on 200x200 integer arrays.
PIA	2065	The Perspective Inversion Algorithm [47] deciding the location of an object in a perspective video image.
Simple	870	A spherical fluid-dynamics program [17], run for 4 iterations with grid size of 100.

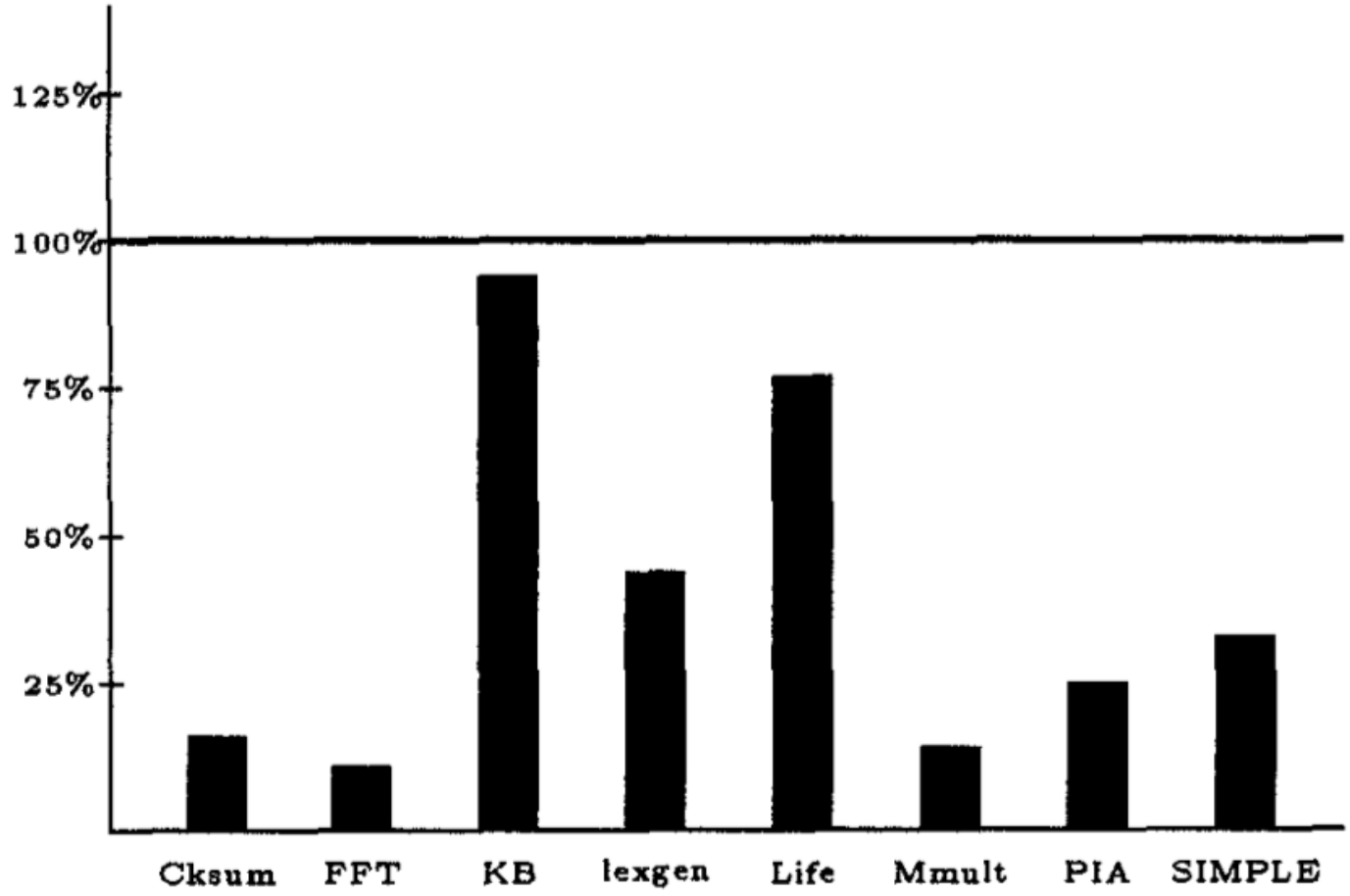


Figure 8: TIL Execution Time Relative to SML/NJ

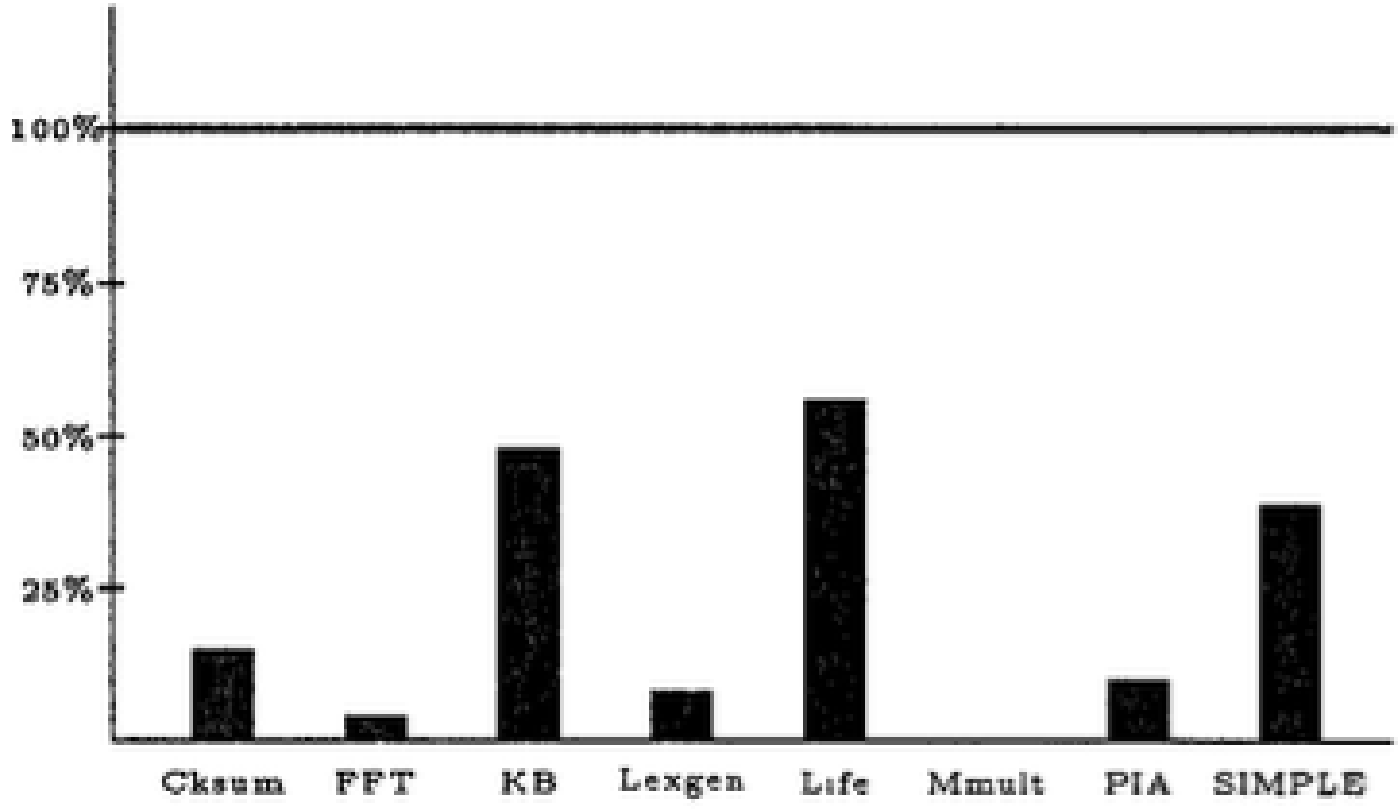


Figure 9: TIL Heap Allocation Relative to SML/NJ

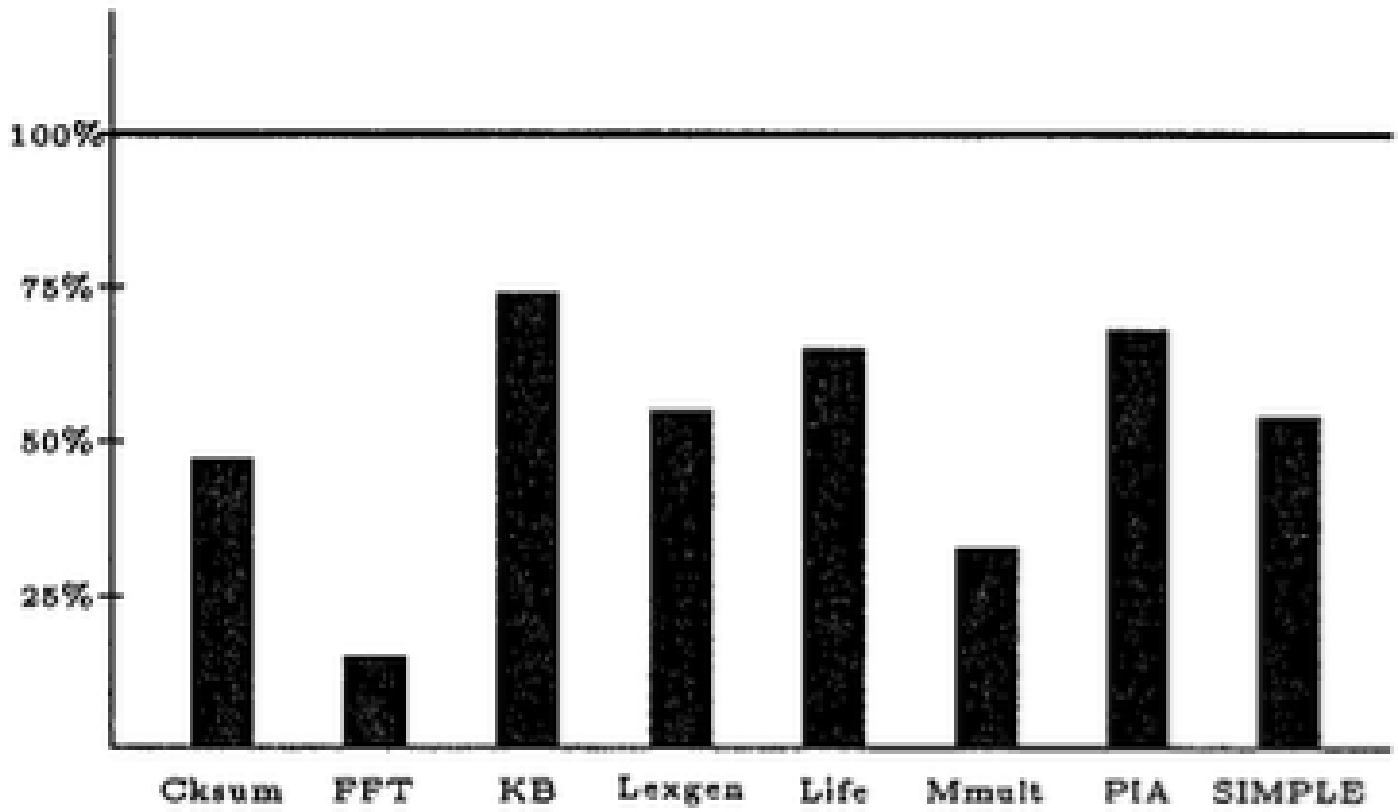


Figure 10: TIL Physical Memory Used Relative to SML/NJ

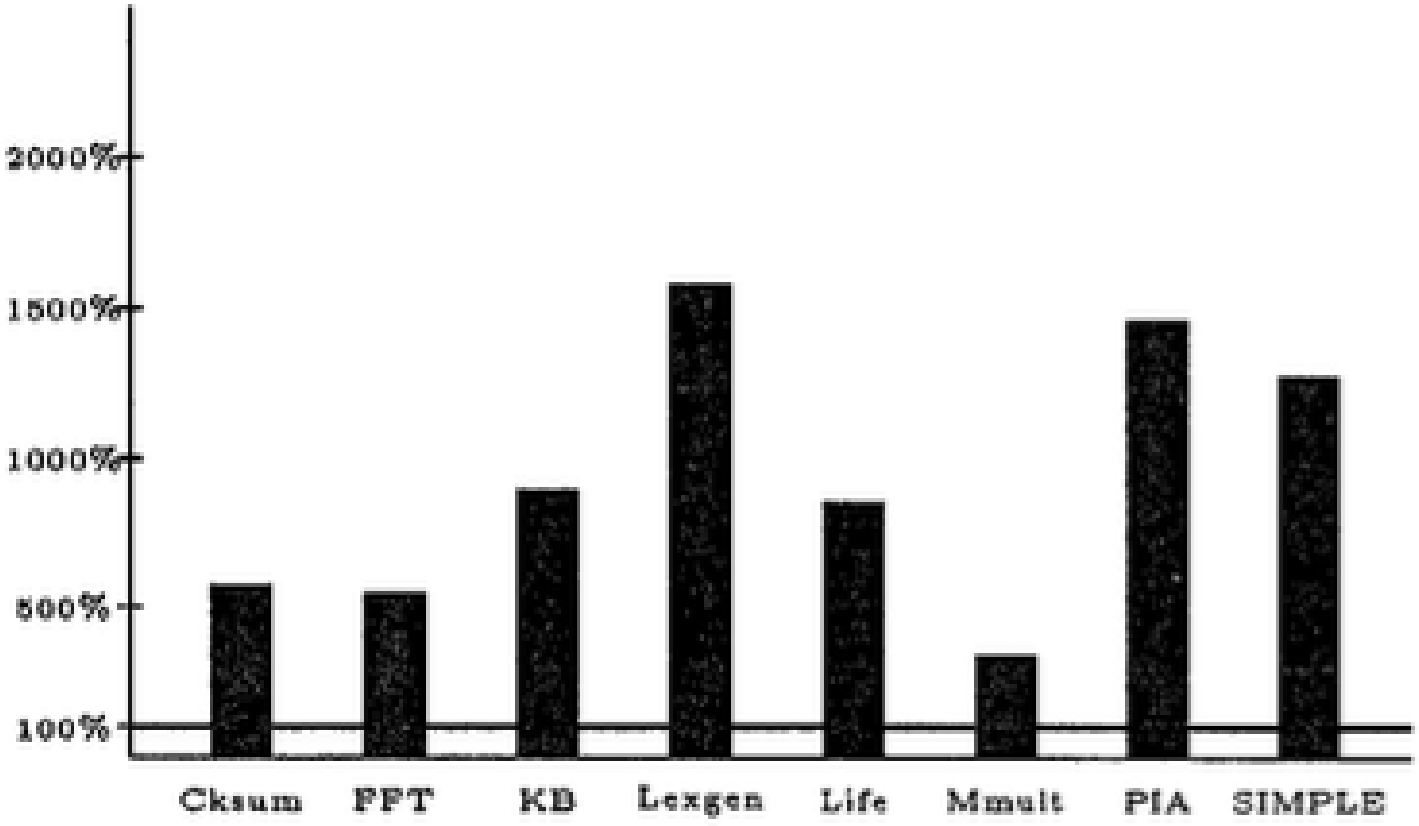


Figure 11: Til Compilation Time Relative to SML/NJ

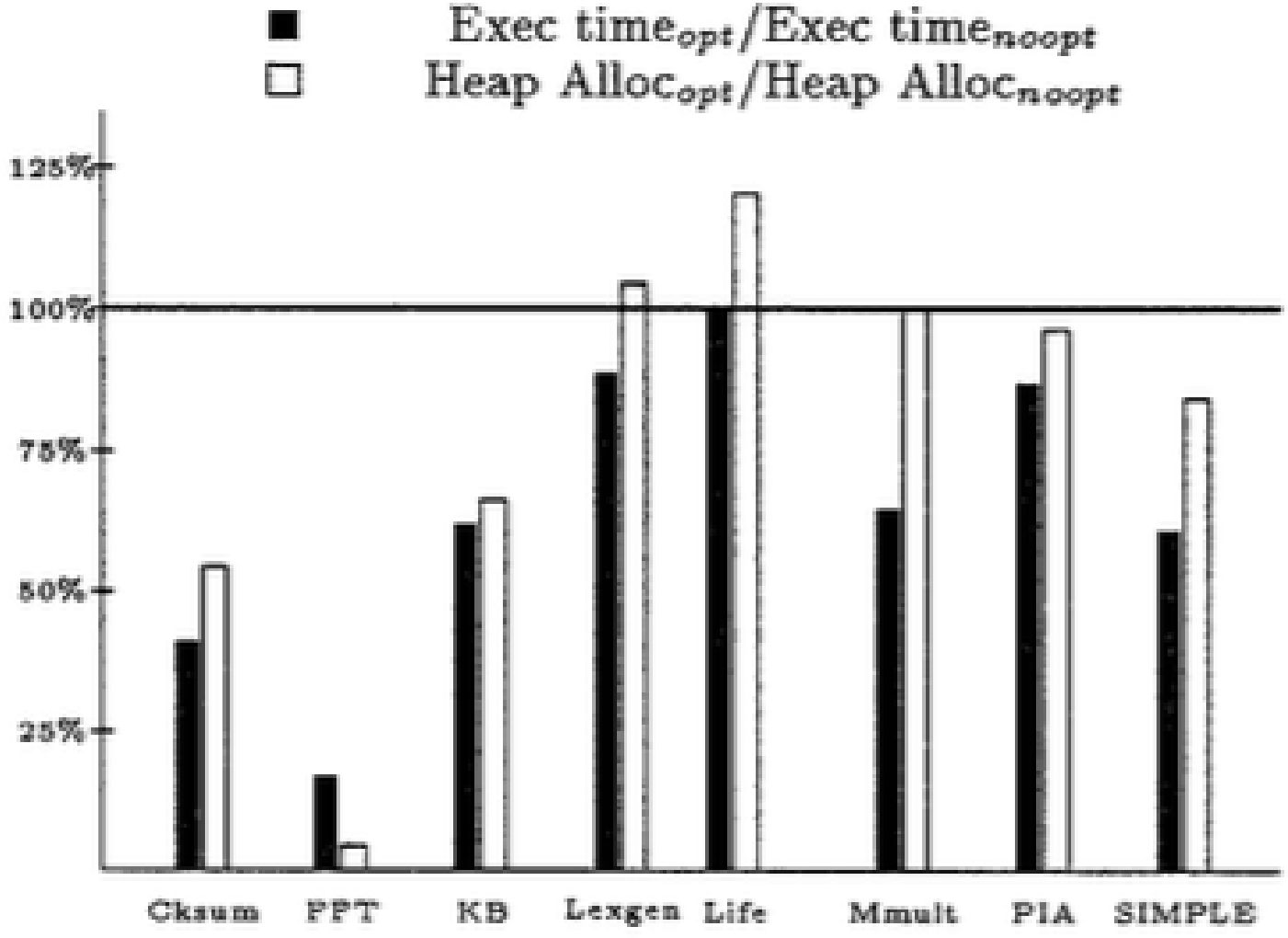


Figure 12: Effects of Loop Optimizations

Last question

How would you go about garbage-collection without any tags?

Conclusion

TIL is an effective optmizing compiler for SML

A few interesting approaches it takes are with nearly tag-free garbage collection and intensional polymorphism

Thanks for listening!


David Tarditi, J. Gregory Morrisett, Perry Cheng, Christopher A. Stone, Robert Harper, and Peter Lee.
TIL: A type-directed optimizing compiler for ML.
In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, May 21-24, 1996*, pages 181–192, 1996.