Intel Labs Haskell Research Compiler

Hai (Paul) Liu
with Neal Glew, Leaf Peterson, Todd A. Anderson
Intel Labs. September 28, 2016

Intel Labs Haskell Research Compiler

An alternative Haskell compiler that:

- uses GHC as frontend;
- does whole program compilation;

Intel Labs Haskell Research Compiler

An alternative Haskell compiler that:

- uses GHC as frontend;
- does whole program compilation;
- achieves overall performance parity with GHC+LLVM;
- is significantly better for some programs;

Intel Labs Haskell Research Compiler

An alternative Haskell compiler that:

- uses GHC as frontend;
- does whole program compilation;
- achieves overall performance parity with GHC+LLVM;
- is significantly better for some programs;
- does automatic SIMD vectorization for Intel CPUs.

However...

The backend of HRC was not originally designed for Haskell.

However...

The backend of HRC was not originally designed for Haskell.

We re-purposed an existing FL compiler and runtime that:

- has a set of different design decisions;
- makes an interesting comparison to GHC.

However...

The backend of HRC was not originally designed for Haskell.

We re-purposed an existing FL compiler and runtime that:

- has a set of different design decisions;
- makes an interesting comparison to GHC.

Known Limitations:

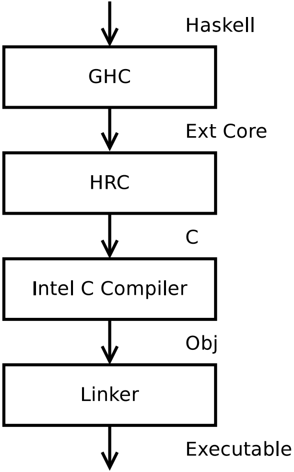
- No lightweight threads, sparks, or STM (easy);
- No exception re-throw for thunks (fixable);
- No asynchronous exceptions (hard).

Functionality

HRC is highly compatible to GHC:

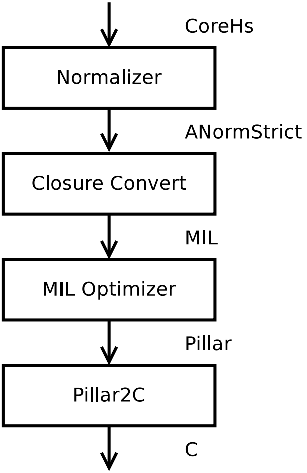
- Modified GHC and base libraries to handle differences;
- Modified Vector library to use initializing writes;
- Modified Cabal to compile for HRC;
- Little or no modifications to most other libraries.

Compilation Process



Inside HRC Pipeline

CoreHs	functional, non-strict
ANormStrict	ANF, strict explicit thunk/eval
MIL	CFG/SSA based high-level object
Pillar	inspired by C−− variant of C



Comparison to GHC

GHC		HRC	
Desugaring Type analysis Core-to-Core transformation			
STG	Functional, object memory model, optimized for currying and thunks	MIL	CFG/SSA based, object memory model, conventional
Cmm	CFG blocks, low-level types, and custom calling convention	Pillar	C types, C calling convention meta and GC support
LLVM or NCG	Portable LLVM bitcode, or direct assembly generation	Intel C/C++ Compiler	Portable C code compiled to assembly
Runtime and GC	Optimized for currying and thunks	Runtime and GC	Conventional

MIL

- High level object model with low level control flow
- Leveraging immutability and memory safety
- Data flow and control flow analysis
- Inter- and intra- procedural optimizations
- Representation/contification/loop/thunk optimizations
- SIMD auto vectorization

Immutable Array

GHC creates an immutable array by:

- creating a mutable array;
- writing to it;
- freezing the result.

Immutable Array

GHC creates an immutable array by:

- creating a mutable array;
- writing to it;
- freezing the result.

HRC separates array creation and initialization, with primitives to:

- create immutable array
- do initializing writes
- read

Immutable Array

GHC creates an immutable array by:

- creating a mutable array;
- writing to it;
- freezing the result.

HRC separates array creation and initialization, with primitives to:

- create immutable array
- do initializing writes
- read

Programmers must ensure:

- an array field is written to before it is read;
- a field is never written to more than once.

Walkthrough by Example

$$\begin{aligned}
&\text{odd, even} :: \text{Int} \rightarrow \text{Bool}\\
&\text{odd } \mathsf{0} = \text{False}\\
&\text{odd } \mathsf{n} = \text{even } (\mathsf{n} - \mathsf{1})\\
\\
&\text{even } \mathsf{0} = \text{True}\\
&\text{even } \mathsf{n} = \text{odd } (\mathsf{n} - \mathsf{1})
\end{aligned}$$

GHC Core

```
even :: (Int → Bool) =
  \ (n :: Int) →
      %case n %of (_ :: Int)
        {I# (n1 :: Int#) →
           %case n1 %of (n2 :: Int#)
             {(0 :: Int#) → True;
              %_ → odd (I# (n2 − (1 :: Int#)))}};

odd :: (Int → Bool) =
  \ (m :: Int) →
      %case m %of (_ :: Int)
        {I# (m1 :: Int#) →
           %case m1 %of (m2 :: Int#)
             {(0 :: Int#) → False;
              %_ → even (I# (m2 − (1 :: Int#)))}};
```

GHC Core

```
even :: (Int → Bool) =
  \ (n :: Int) →
      %case n %of (_ :: Int)
        {I# (n1 :: Int#) →
           %case n1 %of (n2 :: Int#)
             {(0 :: Int#) → True;
              %_ → odd (I# (n2 − (1 :: Int#)))}};

odd :: (Int → Bool) =
  \ (m :: Int) →
      %case m %of (_ :: Int)
        {I# (m1 :: Int#) →
           %case m1 %of (m2 :: Int#)
             {(0 :: Int#) → False;
              %_ → even (I# (m2 − (1 :: Int#)))}};
```

GHC Core

```
even :: (Int → Bool) =
  \ (n :: Int) →
      %case n %of (_ :: Int)
        {I# (n1 :: Int#) →
           %case n1 %of (n2 :: Int#)
             {(0 :: Int#) → True;
              %_ → odd (I# (n2 - (1 :: Int#)))}};

odd :: (Int → Bool) =
  \ (m :: Int) →
      %case m %of (_ :: Int)
        {I# (m1 :: Int#) →
           %case m1 %of (m2 :: Int#)
             {(0 :: Int#) → False;
              %_ → even (I# (m2 - (1 :: Int#)))}};
```

GHC Core

```
even :: (Int → Bool) =
  \ (n :: Int) →
      %case n %of (_ :: Int)
        {I# (n1 :: Int#) →
           %case n1 %of (n2 :: Int#)
             {(0 :: Int#) → True;
              %_ → odd (I# (n2 − (1 :: Int#)))}};

odd :: (Int → Bool) =
  \ (m :: Int) →
      %case m %of (_ :: Int)
        {I# (m1 :: Int#) →
           %case m1 %of (m2 :: Int#)
             {(0 :: Int#) → False;
              %_ → even (I# (m2 − (1 :: Int#)))}};
```

GHC Core

```
even :: (Int → Bool) =
  \ (n :: Int) →
      %case n %of (_ :: Int)
        {I# (n1 :: Int#) →
           %case n1 %of (n2 :: Int#)
             {(0 :: Int#) → True;
              %_ → odd (I# (n2 − (1 :: Int#)))}};

odd :: (Int → Bool) =
  \ (m :: Int) →
      %case m %of (_ :: Int)
        {I# (m1 :: Int#) →
           %case m1 %of (m2 :: Int#)
             {(0 :: Int#) → False;
              %_ → even (I# (m2 − (1 :: Int#)))}};
```

ANormStrict

```
even = %thunk
  %let f = \n →
    %let n0 = %eval n
    %in %case n0 of
        {I# n1 → %case n1 of
          {0 → %eval True;
           _ → %let u = %let v = %let w = 1
                                 %in n1 − w
                             i = %eval I#
                         %in i v
                     t = %thunk u
                     g = %eval odd
                 %in g t}}
  %in f;
```

ANormStrict

```
even = %thunk
  %let f = \n →
    %let n0 = %eval n
    %in %case n0 of
        {I# n1 → %case n1 of
          {0 → %eval True;
           _ → %let u = %let v = %let w = 1
                                 %in n1 − w
                             i = %eval I#
                         %in i v
                     t = %thunk u
                     g = %eval odd
               %in g t}}
  %in f;
```

ANormStrict

```
even = %thunk
  %let f = \n →
    %let n0 = %eval n
    %in %case n0 of
         {I# n1 → %case n1 of
           {0 → %eval True;
            _ → %let u = %let v = %let w = 1
                                  %in n1 − w
                              i = %eval I#
                          %in i v
                     t = %thunk u
                     g = %eval odd
                %in g t}}
  %in f;
```

ANormStrict (closure converted)

```
even = %thunk <I#, True, odd;>
  %let  f  = \<I#, True, odd; n> →
    %let  n0 = %eval n
    %in %case n0 of
        {I# n1 → %case n1 of
          {0 → %eval  True;
           _ → %let  u =  %let  v =  %let  w = 1
                                     %in  n1 − w
                                 i  = %eval I#
                            %in  i  v
                      t  = %thunk <u;> u
                      g  = %eval odd
                 %in  g  t}}
  %in  f;
```

ANormStrict (closure converted)

```
even = %thunk <I#, True, odd;>
  %let  f  = \<I#, True, odd; n> →
    %let  n0 = %eval n
    %in %case n0 of
        {I# n1 → %case n1 of
          {0 → %eval  True;
           _ → %let  u =  %let  v =  %let  w = 1
                                      %in n1 − w
                                i  = %eval I#
                          %in  i  v
                     t  = %thunk <u;> u
                     g  = %eval odd
                 %in  g  t}}
  %in  f;
```

MIL

```
f_code =
  Code(CcClosure(lv_I#, lv_True, lv_odd); n)
  {
     L0():     n0 ← Eval(n) ⇒ L1
     L1():     Case tagof(n0) { U32(0) ⇒ L2() }
     L2():     n1 = SumProj(n0.U32(0).0)
               Case n1 { S32(0) ⇒ L8() Default ⇒ L3() }
     L3():     v = SInt32Minus(n1, 1)
               i ← Eval(lv_I#) ⇒ L4
     L4():     u ← CallClos(i) (v) ⇒ L5
     L5():     t = ThunkMkVal(u)
               g ← Eval(lv_odd) ⇒ L6
     L6():     c ← CallClos(g) (t) ⇒ L7
     L7():     Goto L10(c)
     L8():     b ← Eval(lv_True) ⇒ L9
     L9():     Goto L10(b)
     L10(r):   Return(r)
  }
```

MIL

```
f_code =
  Code(CcClosure(lv_I#, lv_True, lv_odd); n)
  {
    L0():     n0 ← Eval(n) ⇒ L1
    L1():     Case tagof(n0) { U32(0) ⇒ L2() }
    L2():     n1 = SumProj(n0.U32(0).0)
              Case n1 { S32(0) ⇒ L8() Default ⇒ L3() }
    L3():     v = SInt32Minus(n1, 1)
              i ← Eval(lv_I#) ⇒ L4
    L4():     u ← CallClos(i) (v) ⇒ L5
    L5():     t = ThunkMkVal(u)
              g ← Eval(lv_odd) ⇒ L6
    L6():     c ← CallClos(g) (t) ⇒ L7
    L7():     Goto L10(c)
    L8():     b ← Eval(lv_True) ⇒ L9
    L9():     Goto L10(b)
    L10(r):   Return(r)
  }
```

MIL

```
f_code =
  Code(CcClosure(lv_I#, lv_True, lv_odd); n)
  {
    L0():     n0 ← Eval(n) ⇒ L1
    L1():     Case tagof(n0) { U32(0) ⇒ L2() }
    L2():     n1 = SumProj(n0.U32(0).0)
              Case n1 { S32(0) ⇒ L8() Default ⇒ L3() }
    L3():     v = SInt32Minus(n1, 1)
              i ← Eval(lv_I#) ⇒ L4
    L4():     u ← CallClos(i) (v) ⇒ L5
    L5():     t = ThunkMkVal(u)
              g ← Eval(lv_odd) ⇒ L6
    L6():     c ← CallClos(g) (t) ⇒ L7
    L7():     Goto L10(c)
    L8():     b ← Eval(lv_True) ⇒ L9
    L9():     Goto L10(b)
    L10(r):   Return(r)
  }
```

MIL

```
f_code =
  Code(CcClosure(lv_I#, lv_True, lv_odd); n)
  {
    L0():    n0 ← Eval(n) ⇒ L1
    L1():    Case tagof(n0) { U32(0) ⇒ L2() }
    L2():    n1 = SumProj(n0.U32(0).0)
             Case n1 { S32(0) ⇒ L8() Default ⇒ L3() }
    L3():    v = SInt32Minus(n1, 1)
             i ← Eval(lv_I#) ⇒ L4
    L4():    u ← CallClos(i) (v) ⇒ L5
    L5():    t = ThunkMkVal(u)
             g ← Eval(lv_odd) ⇒ L6
    L6():    c ← CallClos(g) (t) ⇒ L7
    L7():    Goto L10(c)
    L8():    b ← Eval(lv_True) ⇒ L9
    L9():    Goto L10(b)
    L10(r):  Return(r)
  }
```

MIL

```
f_code =
  Code(CcClosure(lv_I#, lv_True, lv_odd); n)
  {
    L0():    n0 ← Eval(n) ⇒ L1
    L1():    Case tagof(n0) { U32(0) ⇒ L2() }
    L2():    n1 = SumProj(n0.U32(0).0)
             Case n1 { S32(0) ⇒ L8() Default ⇒ L3() }
    L3():    v = SInt32Minus(n1, 1)
             i ← Eval(lv_I#) ⇒ L4
    L4():    u ← CallClos(i) (v) ⇒ L5
    L5():    t = ThunkMkVal(u)
             g ← Eval(lv_odd) ⇒ L6
    L6():    c ← CallClos(g) (t) ⇒ L7
    L7():    Goto L10(c)
    L8():    b ← Eval(lv_True) ⇒ L9
    L9():    Goto L10(b)
    L10(r):  Return(r)
  }
```

MIL

```
f_code =
  Code(CcClosure(lv_I#, lv_True, lv_odd); n)
  {
    L0():    n0 ← Eval(n) ⇒ L1
    L1():    Case tagof(n0) { U32(0) ⇒ L2() }
    L2():    n1 = SumProj(n0.U32(0).0)
             Case n1 { S32(0) ⇒ L8() Default ⇒ L3() }
    L3():    v = SInt32Minus(n1, 1)
             i ← Eval(lv_I#) ⇒ L4
    L4():    u ← CallClos(i) (v) ⇒ L5
    L5():    t = ThunkMkVal(u)
             g ← Eval(lv_odd) ⇒ L6
    L6():    c ← CallClos(g) (t) ⇒ L7
    L7():    Goto L10(c)
    L8():    b ← Eval(lv_True) ⇒ L9
    L9():    Goto L10(b)
    L10(r):  Return(r)
  }
```

MIL

```
f_code =
  Code(CcClosure(lv_I#, lv_True, lv_odd); n)
  {
    L0():     n0 ← Eval(n) ⇒ L1
    L1():     Case tagof(n0) { U32(0) ⇒ L2() }
    L2():     n1 = SumProj(n0.U32(0).0)
              Case n1 { S32(0) ⇒ L8() Default ⇒ L3() }
    L3():     v = SInt32Minus(n1, 1)
              i ← Eval(lv_I#) ⇒ L4
    L4():     u ← CallClos(i) (v) ⇒ L5
    L5():     t = ThunkMkVal(u)
              g ← Eval(lv_odd) ⇒ L6
    L6():     c ← CallClos(g) (t) ⇒ L7
    L7():     Goto L10(c)
    L8():     b ← Eval(lv_True) ⇒ L9
    L9():     Goto L10(b)
    L10(r):   Return(r)
  }
```

MIL (simplified)

```
f_code =
  Code(CcClosure(lv_I#, lv_True, lv_odd); n)
  {
    L0():   n0 ← Eval(n) ⇒ L1
    L1():   n1 = SumProj(n0.U32(0).0)
            e  = SInt32Eq (n1, 0)
            Case e { True ⇒ L6() False ⇒ L2() }
    L2():   v = SInt32Minus(n1, 1)
            i ← Eval(lv_I#) ⇒ L3
    L3():   u ← CallClos(i) (v) ⇒ L4
    L4():   t = ThunkMkVal(u)
            g ← Eval(lv_odd) ⇒ L5
    L5():   CallClos(g) (t) =|
    L6():   Return(gv_True)
  }
```

MIL (simplified)

```
f_code =
  Code(CcClosure(lv_I#, lv_True, lv_odd); n)
  {
    L0():   n0 ← Eval(n) ⇒ L1
    L1():   n1 = SumProj(n0.U32(0).0)
            e  = SInt32Eq (n1, 0)
            Case e { True ⇒ L6() False ⇒ L2() }
    L2():   v = SInt32Minus(n1, 1)
            i ← Eval(lv_I#) ⇒ L3
    L3():   u ← CallClos(i) (v) ⇒ L4
    L4():   t = ThunkMkVal(u)
            g ← Eval(lv_odd) ⇒ L5
    L5():   CallClos(g) (t) =|
    L6():   Return(gv_True)
  }
```

MIL (simplified)

```
f_code =
  Code(CcClosure(lv_I#, lv_True, lv_odd); n)
  {
    L0():    n0 ← Eval(n) ⇒ L1
    L1():    n1 = SumProj(n0.U32(0).0)
             e  = SInt32Eq (n1, 0)
             Case e { True ⇒ L6() False ⇒ L2() }
    L2():    v = SInt32Minus(n1, 1)
             i ← Eval(lv_I#) ⇒ L3
    L3():    u ← CallClos(i) (v) ⇒ L4
    L4():    t = ThunkMkVal(u)
             g ← Eval(lv_odd) ⇒ L5
    L5():    CallClos(g) (t) =|
    L6():    Return(gv_True)
  }
```

MIL (simplified)

```
f_code =
  Code(CcClosure(lv_I#, lv_True, lv_odd); n)
  {
    L0():   n0 ← Eval(n) ⇒ L1
    L1():   n1 = SumProj(n0.U32(0).0)
            e  = SInt32Eq (n1, 0)
            Case e { True ⇒ L6() False ⇒ L2() }
    L2():   v = SInt32Minus(n1, 1)
            i ← Eval(lv_I#) ⇒ L3
    L3():   u ← CallClos(i) (v) ⇒ L4
    L4():   t = ThunkMkVal(u)
            g ← Eval(lv_odd) ⇒ L5
    L5():   CallClos(g) (t) =|
    L6():   Return(gv_True)
  }
```

MIL (after Rep optimization)

```
f_code =
  Code(CcCode; n)
  {
    L0():   even = SInt32Eq (n, 0)
            Case even { True ⇒ L4() False ⇒ L1() }
    L1():   v = SInt32Minus(n, 1)
            odd = SInt32Eq(v, 0)
            Case odd  { True ⇒ L3() False ⇒ L2() }
    L2():   w  = SInt32Minus(v, 1)
            Call(f_code) (w) =|
    L3():   Return(gv_False)
    L4():   Return(gv_True)
  }
```

MIL (after Rep optimization)

```
f_code =
  Code(CcCode; n)
  {
    L0():    even = SInt32Eq (n, 0)
             Case even { True ⇒ L4() False ⇒ L1() }
    L1():    v = SInt32Minus(n, 1)
             odd = SInt32Eq(v, 0)
             Case odd { True ⇒ L3() False ⇒ L2() }
    L2():    w = SInt32Minus(v, 1)
             Call (f_code) (w) =|
    L3():    Return(gv_False)
    L4():    Return(gv_True)
  }
```

MIL (after Rep optimization)

```
f_code =
  Code(CcCode; n)
  {
    L0():    even = SInt32Eq (n, 0)
             Case even { True ⇒ L4() False ⇒ L1() }
    L1():    v = SInt32Minus(n, 1)
             odd = SInt32Eq(v, 0)
             Case odd  { True ⇒ L3() False ⇒ L2() }
    L2():    w  = SInt32Minus(v, 1)
             Call(f_code) (w) =|
    L3():    Return(gv_False)
    L4():    Return(gv_True)
  }
```

MIL (after contification)

```
f_code =
  Code(CcCode; n)
  {
    L0():   Goto L1(n)
    L1(u):  even = SInt32Eq(u, 0)
            Case even { True ⇒ L5() False ⇒ L2() }
    L2():   v = SInt32Minus(u, 1)
            odd = SInt32Eq(v, 0)
            Case odd  { True ⇒ L4() False ⇒ L3() }
    L3():   w = SInt32Minus(v, 1)
            Goto L1(w)
    L4():   Return(gv_False)
    L5():   Return(gv_True)
  }
```

MIL (after arithmetic simplification)

```
f_code =
  Code(CcCode; n)
  {
    L0():    Goto L1(n)
    L1(u):   even = SInt32Eq(u, 0)
             Case even { True ⇒ L5() False ⇒ L2() }
    L2():    odd = SInt32Eq(u, 1)
             Case odd  { True ⇒ L4() False ⇒ L3() }
    L3():    w = SInt32Minus(u, 2)
             Goto L1(w)
    L4():    Return(gv_False)
    L5():    Return(gv_True)
  }
```

Benchmarking

40+ benchmark programs:

- a mixed set of programs from nofib benchmark suite;
- performance oriented programs using array libraries.

Sequential performance is measured by compiling and running on 2.7GHz Xeon machine (32-bit Windows) with:

- standard GHC 7.6.1
- GHC 7.6.1 + LLVM 2.9
- HRC with modified GHC 7.6.1 + Intel C/C++ compiler.

Benchmark Result

HRC is at parity to GHC+LLVM, which is 10% faster than GHC.

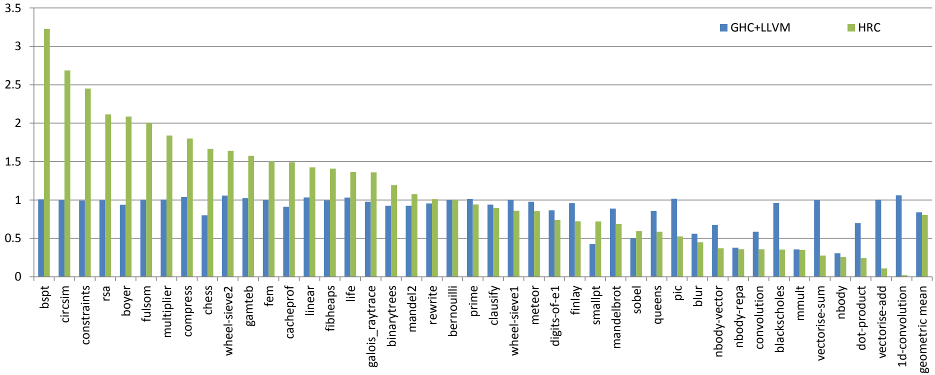


Figure: Kernel Execution Time Relative to GHC (smaller is better)

Benchmark Result (selected)

Performance oriented program with a numeric computation kernel using arrays.

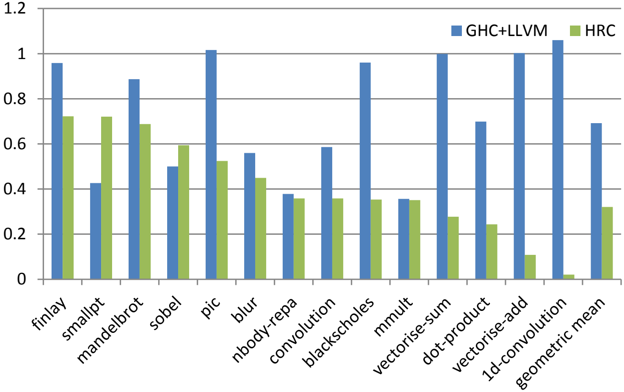


Figure: Kernel Execution Time Relative to GHC (smaller is better)

Performance

For GHC:

- GHC is better at executing lazy code and curried functions.
- GHC's object representation and GC are better suited to typical Haskell programs.

For HRC:

- HRC focuses on optimizing strict programs with hot loops.
- HRC benefits significantly from the elimination of thunks, boxes and branches.

Take Aways

- Reusing GHC is a big win (Core: easy, library: a bit work)
- Novel MIL design choices:
  - Low-level control with high-level object.
  - Immutable array with initializing writes.
- Eliminating thunks is critical to performance.
- Penalty of not using a specialized runtime.
- Compilation through C has overhead, but not as significant.

References

Automatic SIMD Vectorization for Haskell. Leaf Petersen and Dominic Orchard and Neal Glew. ICFP'13.

Measuring the Haskell Gap. Leaf Petersen, Todd A. Anderson, Hai Liu and Neal Glew. Presented at IFL'13.

A Multivalued Language with a Dependent Type System. Neal Glew, Tim Sweeney and Leaf Petersen. DTP'13.

Pillar: A Parallel Implementation Language. Anderson et al. LCPC'07.

Optimizations in a private nursery-based garbage collector. Todd A. Anderson. ISMM'10.