Compiling Parallel Algorithms to Memory Systems: Some Preliminary Results

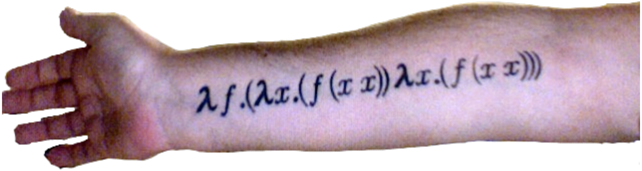
Stephen A. Edwards

Columbia University

CSL Group Seminar, March 25, 2013

$$(\lambda x.?)\, f = \mathrm{FPGA}$$

Functional Programs to FPGAs

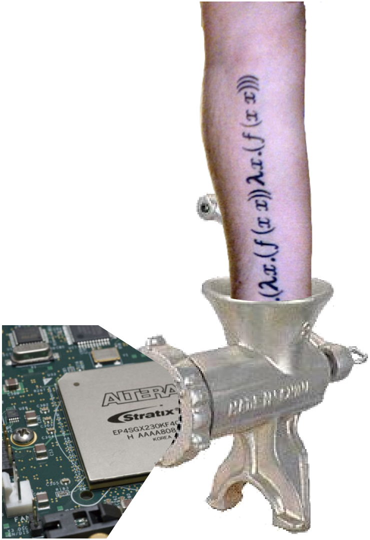


Functional Programs to FPGAs

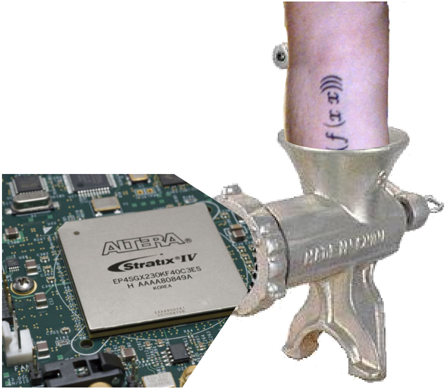
Functional Programs to FPGAs

Functional Programs to FPGAs

Functional Programs to FPGAs

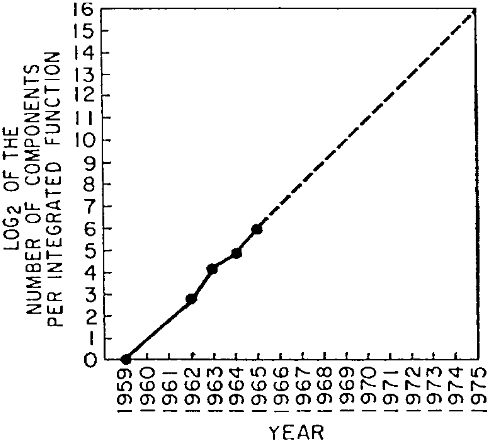


Functional Programs to FPGAs



Functional Programs to FPGAs

Moore's Law: Lots of Cheap Transistors...

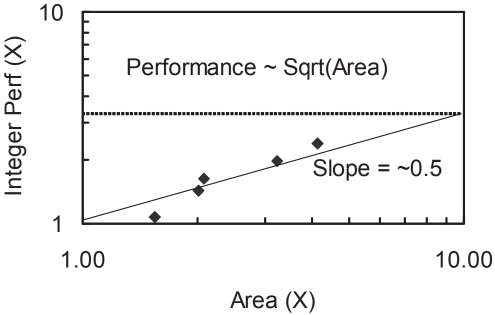


"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year."

Closer to every 24 months

Gordon Moore, *Cramming More Components onto Integrated Circuits,* Electronics, 38(8) April 19, 1965.

Pollack's Rule: ...Give Diminishing Returns for Processors

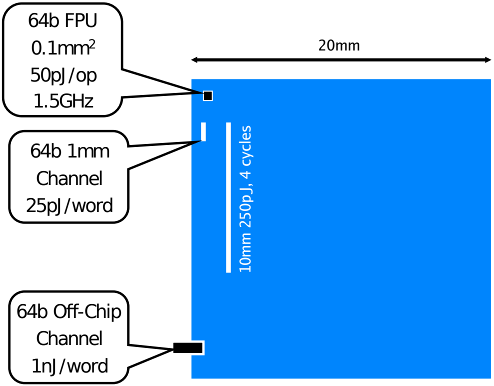


Single-core processor performance follows the square root of area.

It takes 4× the transistors to give 2× the performance.

Fred J. Pollack, MICRO 1999 keynote. Graph from Borkar, DAC 2007

Dally: Calculation is Cheap; Communication is Costly



"Chips are power limited and most power is spent moving data

Performance = Parallelism

Efficiency = Locality

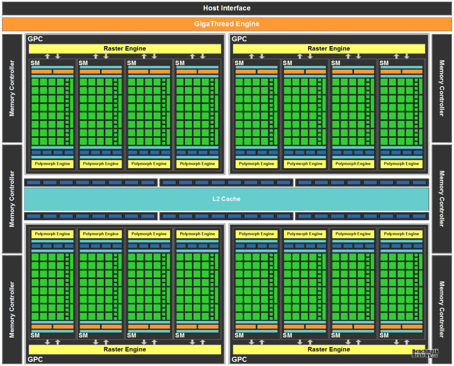
Bill Dally's 2009 DAC Keynote, *The End of Denial Architecture*

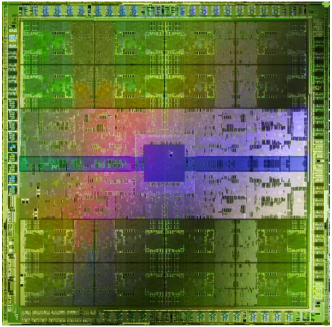
Parallelism for Performance and Locality for Efficiency

Dally: “Single-thread processors are in denial about these two facts”

We need
different programming paradigms
and
different architectures
on which to run them.

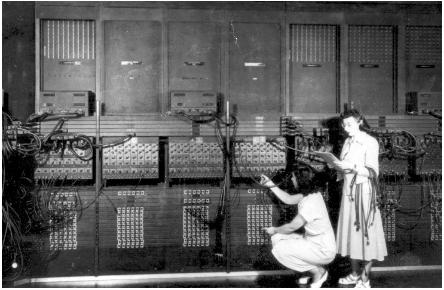
Massive On-Chip Parallelism is Here

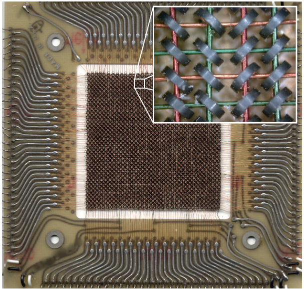




NVIDIA GeForce GTX-400/GF100/Fermi:
3 billion transistors, 512 CUDA cores, 16 geometry units, 64 texture units, 48 render output units, 384-bit GDDR5

The Future is Wires and Memory



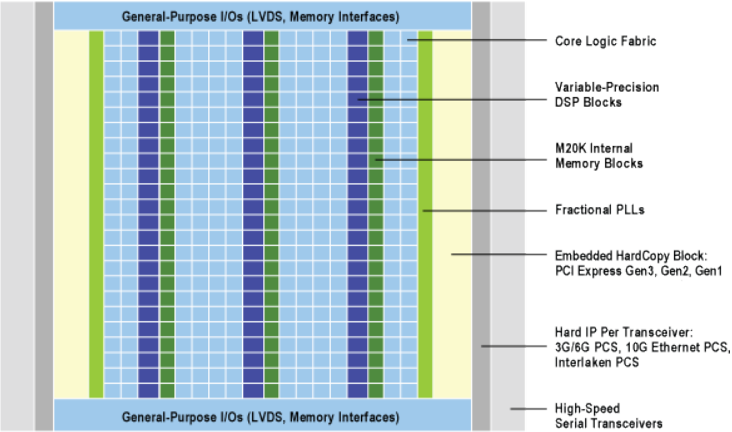


A Modern High-End FPGA: Altera's Stratix V

2500 dual-ported 2.5KB 600 MHz memory blocks; 6 Mb total

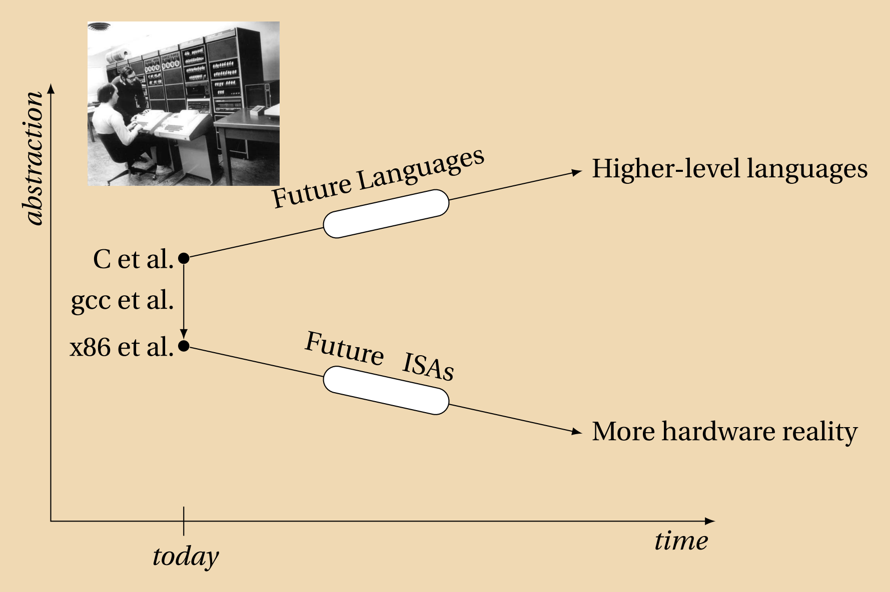
350 36-bit 500 MHz DSP blocks (MAC-oriented datapaths)

300000 6-input LUTs; 28 nm feature size

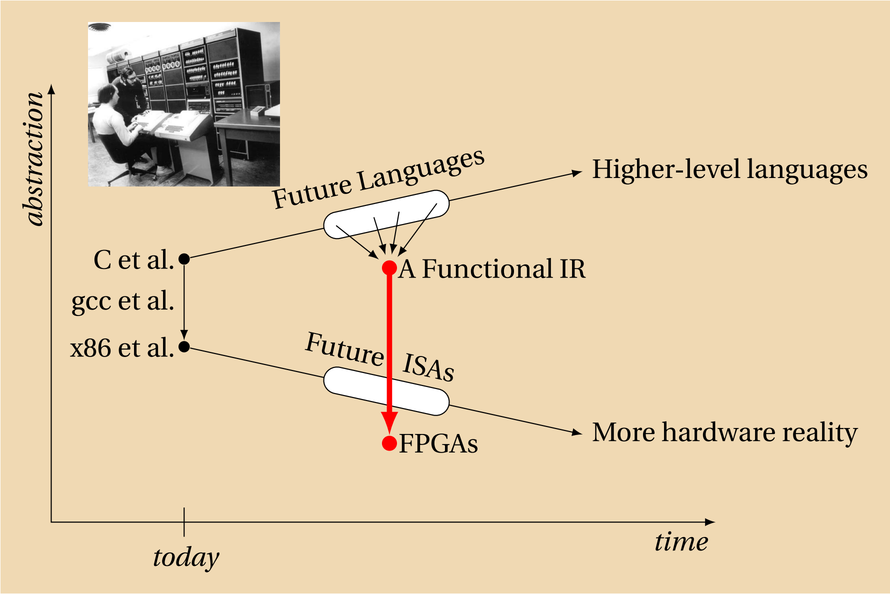


What We are Doing About It

What We are Doing About It

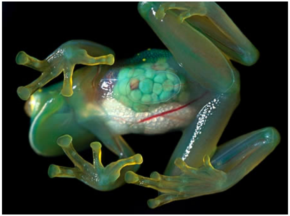


What We are Doing About It

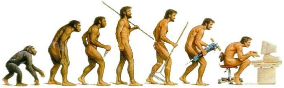


Why Functional Specifications?

- Referential transparency/side-effect freedom make formal reasoning about programs vastly easier
- Inherently concurrent and race-free (Thank Church and Rosser). If you want races and deadlocks, you need to add constructs.
- Immutable data structures makes it vastly easier to reason about memory in the presence of concurrency



Why FPGAs?



- We do not know the structure of future memory systems
  Homogeneous/Heterogeneous?
  Levels of Hierarchy?
  Communication Mechanisms?

- We do not know the architecture of future multi-cores
  Programmable in Assembly/C?
  Single- or multi-threaded?

Use FPGAs as a surrogate. Ultimately too flexible, but representative of the long-term solution.

The Practical Question

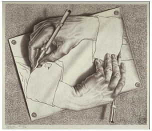
How do we synthesize hardware from pure functional languages for FPGAs?

Control and datapath are easy; the memory system is interesting.

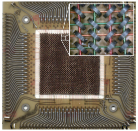
To Implement Real Algorithms in Hardware, We Need

Structured, recursive data types

Recursion to handle recursive data types



Memories



Memory Hierarchy

The Type System: Algebraic Data Types

Types are primitive (Boolean, Integer, etc.) or other ADTs:

$$\begin{array}{rcll} \mathit{type} & ::= & \mathit{Type} & \text{Named type/primitive} \\ & | & \mathit{Constr}\ \mathit{Type}^* \mid \cdots \mid \mathit{Constr}\ \mathit{Type}^* & \text{Tagged union} \end{array}$$

Subsume C structs, unions, and enums

Comparable power to C++ objects with virtual methods

“Algebraic” because they are sum-of-product types.

The Type System: Algebraic Data Types

Types are primitive (Boolean, Integer, etc.) or other ADTs:

$\mathit{type} ::= \mathit{Type}$	Named type/primitive
$\quad \mid\ \mathit{Constr}\ \mathit{Type}^* \mid \cdots \mid \mathit{Constr}\ \mathit{Type}^*$	Tagged union

Examples:

data Intlist = Nil — *Linked list of integers*
| Cons **Int** Intlist

data Bintree = Leaf **Int** — *Binary tree w/ integer leaves*
| Branch BinTree Bintree

data Expr = Literal **Int** — *Arithmetic expression*
| Var **String**
| Binop Expr Op Expr

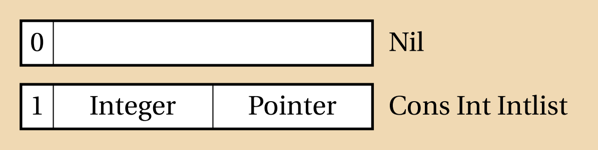
data Op = Add | Sub | Mult | Div

Representing Recursive Algebraic Data Types

Consider a list of integers:

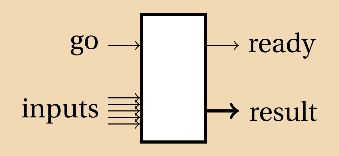
$$\begin{array}{lll} \textbf{data}\ \text{Intlist} & = & \text{Nil} \\ & \mid & \text{Cons}\ \textbf{Int}\ \text{Intlist} \end{array}$$

An obvious representation:

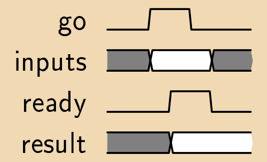


- Usual byte-alignment unnecessary & wasteful in hardware
- Naturally stored & managed in a custom integer-list memory
- Width of pointer can depend on integer-list memory size

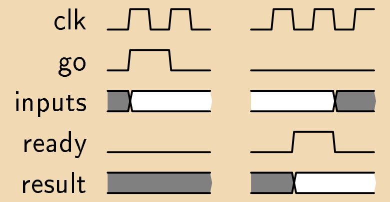
Syntax-Directed Translation of Expressions to Hardware



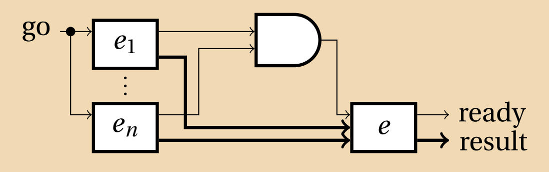
Combinational functions:



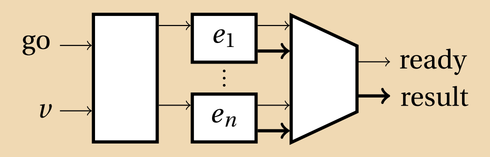
Sequential functions:



Translating Let and Case



Let makes all new variables available to its body.



Case invokes one of its sub-expressions, then synchronizes.

Removing Recursion: Recursive Fibonacci Example

```
fib 1 = 1                  -- Base case
fib 2 = 1                  -- Base case
fib n = fib (n−1) + fib (n−2) -- Recurse twice and sum results
```

Transform to Continuation-Passing Style

```
fib' 1 k = k 1                     -- Base case
fib' 2 k = k 1                     -- Base case
fib' n k = fib' (n−1)              -- First recursive call
    (λn1 → fib' (n−2)              -- Second recursive call
      (λn2 → k (n1 + n2)))         -- Sum results

fib n = fib' n (λx → x)
```

Name intermediate results (e.g., call to fib′ (n−1)). Pass them as arguments to λ terms.

Well-known technique; e.g., Appel et al.; SML/NJ compiler.

Name Lamba Terms; Capture Free Variables

```
call  1  k       = k 1                      -- Base case (return)
call  2  k       = k 1                      -- Base case (return)
call  n  k       = call (n-1) (c1 n k)      -- First recursive call (call)
c1    n  k n1    = call (n-2) (c2 n1 k)     -- Second recursive call (call)
c2    n1 k n2    = k (n1 + n2)              -- Sum Results (return)
c3    x          = x                        -- Return final result

fib n = call n c3
```

Each lambda term becomes its own function.

Represent Continuations with a Type; Merge Functions

```
fib' (Call           1  k)   = fib' (Cont k 1)
fib' (Call           2  k)   = fib' (Cont k 1)
fib' (Call           n  k)   = fib' (Call (n−1) (C1 n k))
fib' (Cont (C1 n k)  n1)     = fib' (Call (n−2) (C2 n1 k))
fib' (Cont (C2 n1 k) n2)     = fib' (Cont k (n1 + n2))
fib' (Cont (C3)         x)   = x

fib n = fib' (Call n C3)

data Continuation  = C1 Word8 Continuation
                   | C2 Word32 Continuation
                   | C3

data Call = Call Word8 Continuation
          | Cont Continuation Word32
```

Replace Type Recursion with Pointers

Before:

```
data Continuation  = C1 Word8 Continuation
                   | C2 Word32 Continuation
                   | C3
```

After:

```
type ContPtr = Word8 -- Pointer to a Continuation object

type ContRef = (ContPtr, ContMem)

data Continuation = C1 Word8 ContRef
                  | C2 Word32 ContRef
                  | C3
```

An Explicit "Store" Function

```
type ContMem = Array ContPtr ContBits -- Model of memory

data ContBits  = CB1 Word8    -- No need for "next" pointer
               | CB2 Word32   -- since these are on a stack
               | CB3

store :: Continuation → ContRef
store c = let (p, m, c′) = case c of
                              C1 n  (p, m) → (p, m, CB1 n)
                              C2 n1 (p, m) → (p, m, CB2 n1)
                              C3           → (0, emptyMem, CB3) in
          let p′ = p + 1 in       -- Place in next memory location
          (p′, m // [(p′, c′)])   -- Write memory
```

Store is more like a constructor: data in; address out.

An Explicit “Load” Function

$$
\begin{array}{l}
\text{load} :: \text{ContRef} \rightarrow \text{Continuation} \\
\text{load}\ (\text{p},\ \text{m}) = \mathbf{let}\ \text{p}' = \text{p} - 1\ \mathbf{in} \qquad \textit{-- Successor just below us} \\
\qquad\qquad\qquad \text{loadp}\ (\text{p}',\ \text{m},\ \text{m}\ !\ \text{p}) \qquad \textit{-- Read memory} \\
\\
\text{loadp} :: (\text{ContPtr}, \text{ContMem}, \text{ContBits}) \rightarrow \text{Continuation} \\
\text{loadp}\ (\text{p}',\ \text{m},\ \text{d}) = \mathbf{case}\ \text{d}\ \mathbf{of} \\
\qquad\qquad \text{CB1 n} \rightarrow \text{C1 n}\ (\text{p}',\ \text{m}) \qquad \textit{-- Reconstruct} \\
\qquad\qquad \text{CB2 n1} \rightarrow \text{C2 n1}\ (\text{p}',\ \text{m}) \\
\qquad\qquad \text{CB3} \rightarrow \text{C3}
\end{array}
$$

Broken into two functions to model synchronous RAM:

Load runs before the clock edge (prepare address)

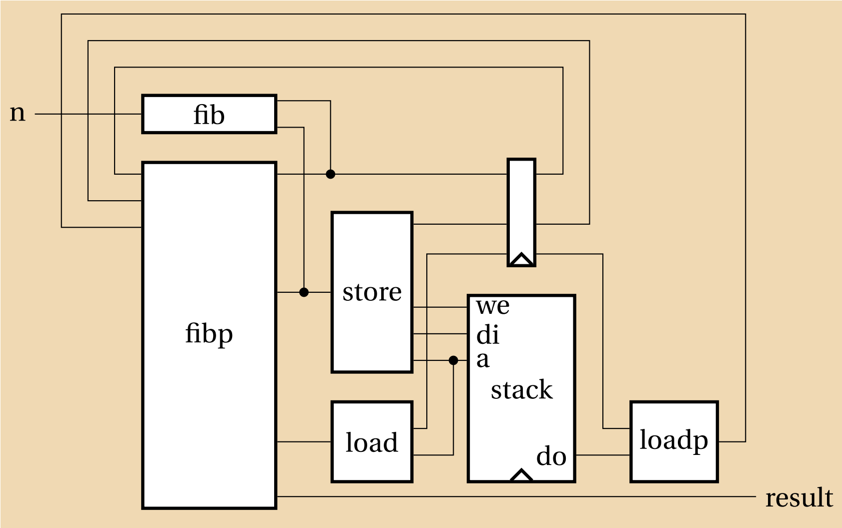
Loadp runs after the clock edge (handle returned data)

Version Suitable for Hardware Translation

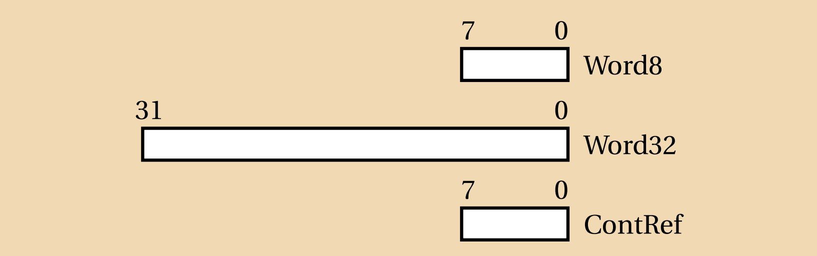
```
fibp (Call         1    kr)   = fibp (Cont (load kr) 1)
fibp (Call         2    kr)   = fibp (Cont (load kr) 1)
fibp (Call         n    kr)   = fibp (Call (n−1) (store (C1 n kr)))
fibp (Cont (C1 n kr) n1)      = fibp (Call (n−2) (store (C2 n1 kr)))
fibp (Cont (C2 n1 kr) n2)     = fibp (Cont (load kr) (n1 + n2))
fibp (Cont (C3)         x )   = x

fib n = fibp (Call n (store C3))
```

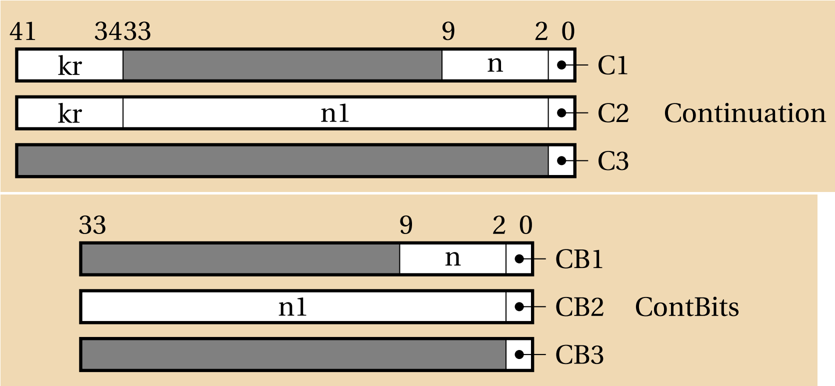
Block Diagram



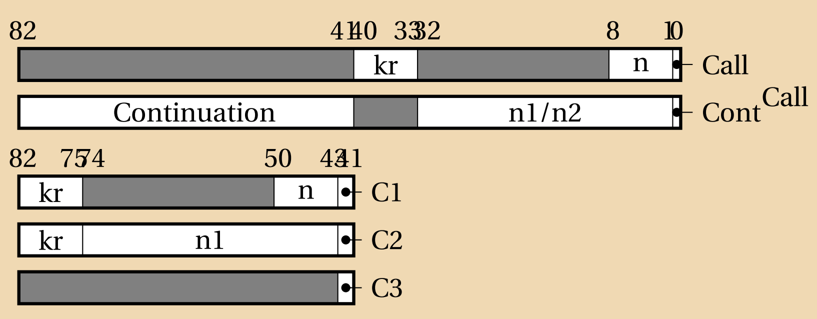
Concrete Representation of Types



Concrete Representation of Types



Concrete Representation of Types



Duplication for Performance

```
fib 0 = 0
fib 1 = 1
fib n = fib (n−1) + fib (n−2)
```

Duplication for Performance

```
fib 0 = 0
fib 1 = 1
fib n = fib (n−1) + fib (n−2)
```

After duplicating functions:

```
fib 0 = 0
fib 1 = 1
fib n = fib′ (n−1) + fib′′ (n−2)

fib′ 0 = 0
fib′ 1 = 1
fib′ n = fib′ (n−1) + fib′ (n−2)

fib′′ 0 = 0
fib′′ 1 = 1
fib′′ n = fib′′ (n−1) + fib′′ (n−2)
```

Here, *fib'* and *fib''* may run in parallel.

Unrolling Recursive Data Structures

Like a "blocking factor," but more general. Idea is to create larger memory blocks that can be operated on in parallel.

Original Huffman tree type:

data Htree = Branch Htree HTree | Leaf **Char**

Unrolled Huffman tree type:

data Htree = Branch Htree′ HTree′ | Leaf **Char**
data Htree′ = Branch′ Htree″ HTree″ | Leaf′ **Char**
data Htree″ = Branch″ Htree HTree | Leaf″ **Char**

Recursive instances must be pointers; others can be explicit.

Functions must be similarly modified to work with the new types.

Identifying Stacks

```
let xs = [1,2,3] in
let ys = 0:xs in
let zs = −1:ys in
ys
```

```
let xs = [1,2,3] in
let ys = 0:xs in
let zs = −1:xs in
ys
```

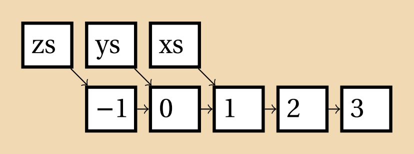
One of these has a list that behaves like a stack; the other does not.

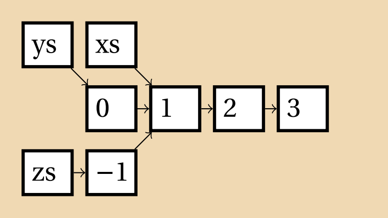
Identifying Stacks

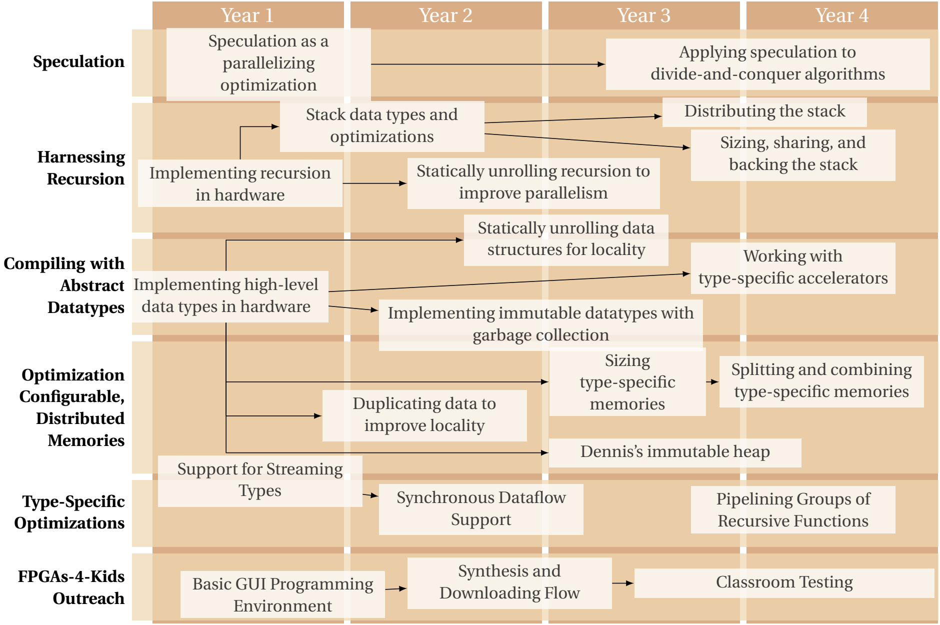
```
let xs = [1,2,3] in
let ys = 0:xs in
let zs = −1:ys in
ys
```

```
let xs = [1,2,3] in
let ys = 0:xs in
let zs = −1:xs in
ys
```

One of these has a list that behaves like a stack; the other does not.
Hint:






Year 1
Year 2
Year 3
Year 4
Speculation
Speculation as a parallelizing optimization
Applying speculation to divide-and-conquer algorithms
Harnessing Recursion
Stack data types and optimizations
Distributing the stack
Sizing, sharing, and backing the stack
Implementing recursion in hardware
Statically unrolling recursion to improve parallelism
Compiling with Abstract Datatypes
Statically unrolling data structures for locality
Working with type-specific accelerators
Implementing high-level data types in hardware
Implementing immutable datatypes with garbage collection
Optimization Configurable, Distributed Memories
Sizing type-specific memories
Splitting and combining type-specific memories
Duplicating data to improve locality
Dennis's immutable heap
Type-Specific Optimizations
Support for Streaming Types
Synchronous Dataflow Support
Pipelining Groups of Recursive Functions
FPGAs-4-Kids Outreach
Basic GUI Programming Environment
Synthesis and Downloading Flow
Classroom Testing