Programming in Manticore, a Heterogenous Parallel Functional Language

Matthew Fluet

Toyota Technological Institute at Chicago

May 25, 2009

Part I

Introduction and Overview

Background and Motivation

The Manticore Project is our effort to address the programming needs of commodity applications running on the commodity hardware of 2014.

- hardware supports concurrency and parallelism at multiple levels
- software exhibits concurrency and parallelism at multiple levels
- to maximize productivity and performance, languages should support concurrency and parallelism at multiple levels

`http://manticore.cs.uchicago.edu`

Background and Motivation

The Manticore Project is our effort to address the programming needs of commodity applications running on the commodity hardware of 2014.

- hardware supports concurrency and parallelism at multiple levels
- software exhibits concurrency and parallelism at multiple levels
- to maximize productivity and performance, languages should support concurrency and parallelism at multiple levels

`http://manticore.cs.uchicago.edu`

Manticore is a *research* project.

People and Acknowledgements

The Manticore Project is a joint project between the University of Chicago and the Toyota Technological Institute at Chicago:

- Lars Bergstrom — University of Chicago
- Matthew Fluet — Toyota Technological Institute at Chicago
- Mike Rainey — University of Chicago
- John Reppy — University of Chicago
- Adam Shaw — University of Chicago
- Yingqi Xiao — University of Chicago

and supported (in part) by the

- National Science Foundation

Concurrency and Parallelism in Hardware

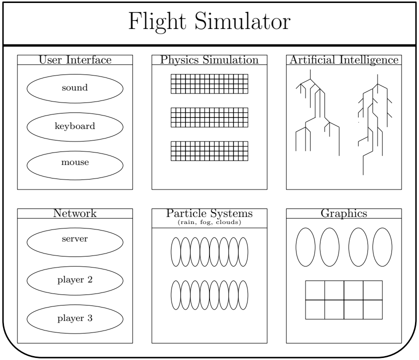
Hardware supports concurrency and parallelism at multiple levels:

- single instruction, multiple data (SIMD) instructions
- simultaneous multithreading executions
- multicore processors
- multiprocessor systems

Concurrency and Parallelism in Software

Software exhibits concurrency and parallelism at multiple levels.

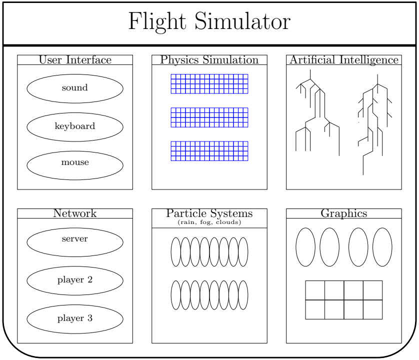
Consider a networked flight simulator:



Concurrency and Parallelism in Software

Software exhibits concurrency and parallelism at multiple levels.

Consider a networked flight simulator:

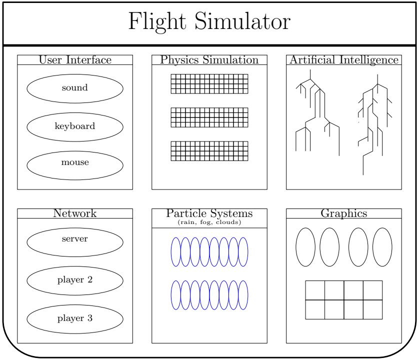


- SIMD parallelism for physics simulation

Concurrency and Parallelism in Software

Software exhibits concurrency and parallelism at multiple levels.

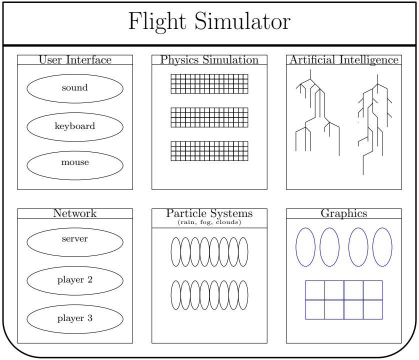
Consider a networked flight simulator:



- data-parallel computations for particle systems
  to model natural phenomena (*e.g.*, rain, fog, and clouds)

Concurrency and Parallelism in Software

Software exhibits concurrency and parallelism at multiple levels.
Consider a networked flight simulator:

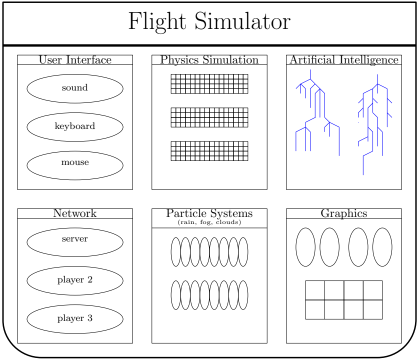


- parallel threads for preloading terrain and computing level-of-detail refinements

Concurrency and Parallelism in Software

Software exhibits concurrency and parallelism at multiple levels.

Consider a networked flight simulator:

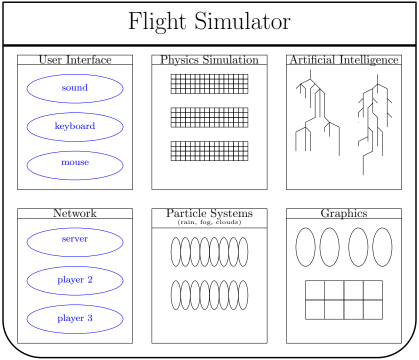


- speculative search for artificial intelligence

Concurrency and Parallelism in Software

Software exhibits concurrency and parallelism at multiple levels.

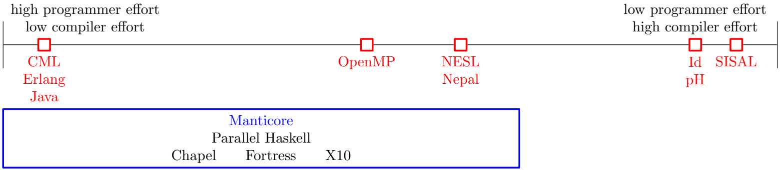
Consider a networked flight simulator:



- concurrent threads for user interface and network components

Manticore: A Heterogeneous Parallel Language

- An effort to design and implement
  a new parallel functional programming language
  supporting heterogeneous parallelism:
  - commodity applications with multiple levels of software parallelism
  - commodity hardware with multiple levels of hardware parallelism



Manticore

A long-range project with two major aspects:

- Language design for heterogeneous parallel programming.
- Language implementation for heterogeneous parallelism.

Manticore: Language Design

Combination of three distinct, but synergistic, sub-languages:

- A mutation-free subset of Standard ML
- Language mechanisms for *explicitly-threaded* concurrency
  - programmer explicitly spawns threads
  - coordinate via synchronous message-passing
- Language mechanisms for *implicitly-threaded* parallelism
  - programmer annotates fine-grained parallel computations
  - compiler and runtime map onto parallel threads

Manticore: Language Implementation

Unified runtime framework:

- Handle demands of various heterogeneous parallelism mechanisms exposed by high-level language design
- Capable of supporting a diverse mix of scheduling policies

Implemented with compiler and runtime-system features:

- small core of primitive scheduling mechanisms
- minimal, light-weight representations for computational tasks, borrowing from past work on *continuations*

Manticore: Language Design

Rooted in the family of *statically-typed*, *strict* functional languages, such as OCaml and Standard ML

- Functional languages emphasize a *value-oriented* and *mutation-free* programming model
  - avoids entanglements between separate concurrent computations
- Strict languages (rather than lazy or lenient languages) are easier to implement efficiently and accessible to a larger community of potential users

Manticore: Language Design

A mutation-free subset of Standard ML

- Strict evaluation
- Statically typed: polymorphism, type inference
- Higher-order functions
- Algebraic datatypes
- Exceptions
  - interesting implications for implicitly-threaded parallelism mechanisms, but useful for systems programming
- Module system (simplified)
  - omit functors and sophisticated type sharing
- No mutable data
  - omit references cells and (mutable) arrays

Manticore: Language Design

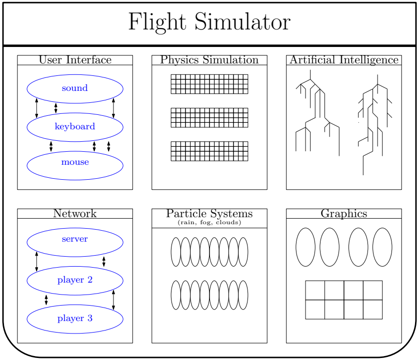
A mutation-free subset of Standard ML

- Strict evaluation
- Statically typed: polymorphism, type inference
- Higher-order functions
- Algebraic datatypes
- Exceptions
  - interesting implications for implicitly-threaded parallelism mechanisms, but useful for systems programming
- Module system (simplified)
  - omit functors and sophisticated type sharing
- No mutable data
  - omit references cells and (mutable) arrays

Manticore: Language Design

Language mechanisms for *explicitly-threaded* concurrency

- programmer explicitly spawns threads
- coordinate via synchronous message-passing



Manticore: Language Design

Language mechanisms for *explicitly-threaded* concurrency

- programmer explicitly spawns threads
- coordinate via synchronous message-passing

These explicit mechanisms serve two purposes:

- support concurrent programming
    - an important feature for systems programming
- support explicit-parallel programming
    - for additional programmer control

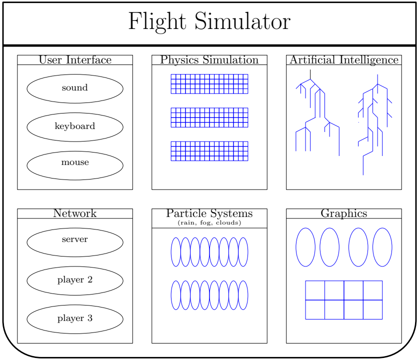
Programming-model based upon *first-class synchronous operations*

- provides a mechanism for building synchronization and communication abstractions

Manticore: Language Design

Language mechanisms for *implicitly-threaded* parallelism

- programmer annotates fine-grained parallel computations
- compiler and runtime map onto parallel threads



Manticore: Language Design

Language mechanisms for *implicitly-threaded* parallelism

- programmer annotates fine-grained parallel computations
- compiler and runtime map onto parallel threads

Manticore provides several light-weight syntactic forms for introducing implicitly-parallel computations.

These forms are *hints* to the compiler and runtime that a computation is a good candidate for parallel execution.

- *Parallel arrays*: fine-grain data-parallel computations over seqs
- *Parallel tuples*: basic fork-join parallel computation
- *Parallel bindings*: data-flow and work-stealing parallelism
- *Parallel case*: non-deterministic speculative parallelism
- *Cancellation*: unused/abandoned subcomputations

Part II

Explicit Concurrency in Manticore

Introduction

Concurrent programming

- programs consisting of multiple independent flows of sequential control (*threads*)
- execution viewed as an interleaving of the sequential executions of consitituent threads

Motivations for concurrent programming:

- improve performance by exploiting multiprocessors
- application domains with naturally concurrent structure:
  - interactive systems (e.g., graphical-user interfaces)
  - distributed systems

Introduction

The explicit-concurrency mechanisms of Manticore
are based on those of Concurrent ML (CML).

- dynamic creation of threads and typed channels
- rendezvous communication via synchronous message passing
- first-class synchronous operations, called events
- automatic reclamation of threads and channels
- pre-emptive scheduling of explicitly concurrent threads
- efficient implementation — both on uni- and multi-processors

Threads

Create a new independent flow of sequential control

```
spawn e
```

- `e` is of type `unit`
- `spawn e` is of type `tid` (the type of a thread identifier)
- the thread that evaluates `spawn e` is the *parent*
- the thread that evaluates `e` is the *child*

Thread executes until the evaluation of its expression is complete

- an uncaught exception completes the evaluation

Threads are preemptively scheduled

Program executes until all threads have terminated or are blocked

Channels

By themselves, multiple concurrent threads are not very useful

Need mechanisms for communication and synchronization

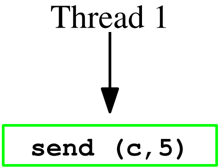
Synchronous message passing on *typed channels*

```
type 'a chan
val channel : unit -> 'a chan
val recv : 'a chan -> 'a
val send : 'a chan * 'a -> unit
```

Channels

Synchronous message passing on typed channels

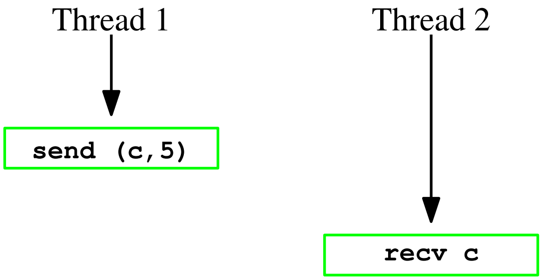
- a sender blocks until there is a matching receiver



Channels

Synchronous message passing on typed channels

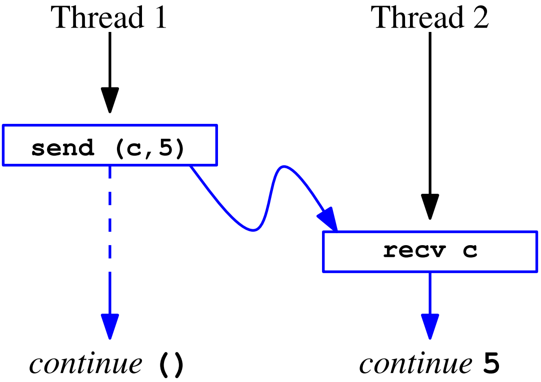
- a sender blocks until there is a matching receiver



Channels

Synchronous message passing on typed channels

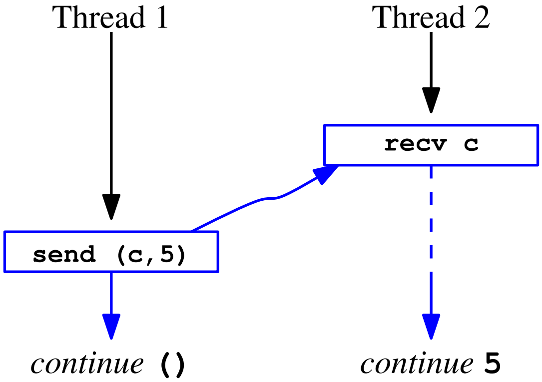
- a sender blocks until there is a matching receiver



Channels

Synchronous message passing on typed channels

- a receiver blocks until there is a matching sender



Channels

Synchronous message passing on typed channels:

- channels do not name the sender or receiver
- channels do not specify the direction of communication
- a channel may pass multiple values between multiple threads
- multiple threads may offer to `recv` or `send` on the same channel
- each `recv` is matched with exactly one `send`

Examples

Three examples

- Updatable storage cells
- Sieve of Eratosthenes (stream of primes)
- Fibonacci Series

Example: Updatable Storage Cells

Although mutable state make concurrent programming difficult, it is easy to give an implementation of updatable storage cells using threads and channels

Implementation is a prototypical example of the *client-server* style of concurrent programming

```
signature CELL =
  sig
    type 'a cell
    val cell : 'a -> 'a cell
    val get  : 'a cell -> 'a
    val put  : 'a cell * 'a -> unit
  end
```

Example: Updatable Storage Cells

```
structure Cell : CELL =
  sig
    datatype 'a req = GET of 'a chan | PUT of 'a
    datatype 'a cell = CELL of 'a req chan

    fun get (CELL reqCh) =
      let
        val replyCh = channel ()
      in
        send (reqCh, GET replyCh) ;
        recv replyCh
      end

    fun put (CELL reqCh, y) =
      send (reqCh, PUT y)
```

Example: Updatable Storage Cells

```
  fun cell z =
    let
      val reqCh = channel ()
      fun loop x =
        case recv reqCh of
           GET replyCh => (send (replyCh, x) ;
                           loop x)
          | PUT y => loop y
      val _ = spawn (loop z)
    in
      CELL reqCh
    end
end
```

Example: Sieve of Eratosthenes

Compute a *stream* of prime numbers

Implementation is a prototypical example of the *dataflow* style of concurrent programming

```
fun firstPrimes (n : int) : int list =
  let val primesCh = primes ()
      fun loop (i, acc) =
        if i = 0
          then rev acc
          else loop (i - 1, (recv primesCh)::acc)
  in  loop (n, [])
  end
```

Example: Sieve of Eratosthenes

```
fun forever (init : 'a) (f : 'a -> 'a) : unit =
  let fun loop s = loop (f s)
      val _ = spawn (loop init)
  in  ()
  end

fun succs (i : int) : int chan =
  let val succsCh = channel ()
      fun succsFn i = (send (succsCh, i) ; i + 1)
      val () = forever i succsFn
  in  succsCh
  end
```

Example: Sieve of Eratosthenes

```
fun filter (p: int, inCh : int chan) : int chan =
  let val outCh = channel ()
      fun filterFn () =
        let val i = recv inCh
        in  if (i mod p) <> 0 then send (outCh, i) else ()
        end
      val () = forever () filterFn
  in  outCh
  end

fun primes () : int chan =
  let val primesCh = channel ()
      fun primesFn ch =
        let val p = recvCh
        in  send (primesCh p) ; filter (p, ch)
        end
      val () = forever (succs 2) primesFn
  in  primesCh
  end
```

Example: Fibonacci Series

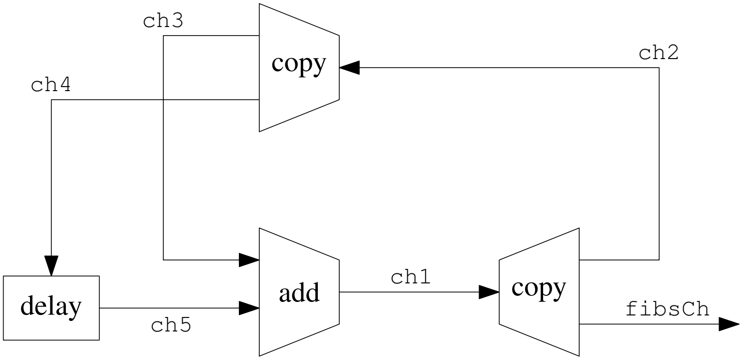
Compute a *stream* of Fibonacci numbers

$$\begin{aligned} fib_1 &= 1 \\ fib_2 &= 1 \\ fib_{i+2} &= fib_{i+1} + fib_i \end{aligned}$$

Example: Fibonacci Series

Compute a *stream* of Fibonacci numbers

$$\begin{aligned} fib_1 &= 1 \\ fib_2 &= 1 \\ fib_{i+2} &= fib_{i+1} + fib_i \end{aligned}$$



Example: Fibonacci Series

```
fun addStrms (inCh1, inCh2, outCh) =
  forever () (fn () =>
    send (outCh, (recv inCh1) + (recv inCh2)))

fun copyStrm (inCh, outCh1, outCh2) =
   forever () (fn () =>
     let val x = recv inCh
     in  send (outCh1, x) ; send (outCh2, x)
     end)

fun delayStrm first (inCh, outCh) =
   forever first (fn x =>
      (send (outCh, x) ; recv inCh))
```

Example: Fibonacci Series

```
fun fibs () : int chan =
  let val fibsCh = channel ()
      val ch1 = channel ()
      val ch2 = channel ()
      val ch3 = channel ()
      val ch4 = channel ()
      val ch5 = channel ()
   in
      copyStrm (ch1, ch2, fibsCh) ;
      copyStrm (ch2, ch3, ch4) ;
      delayStrm 0 (ch4, ch5) ;
      addStrms (ch3, ch5, ch1) ;
      send (ch1, 1) ;
      fibsCh
   end
```

Need for Selective Communication

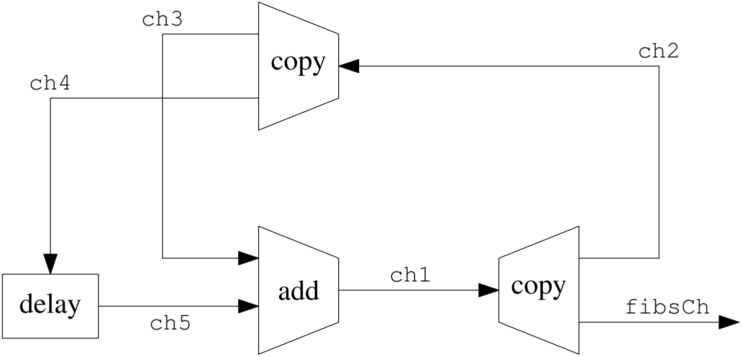
When programming with `recv` and `send` exclusively, there are limits to the kinds of concurrent programs that can be expressed.

- fragility in the implementation of concurrency abstractions

Need for Selective Communication

When programming with `recv` and `send` exclusively, there are limits to the kinds of concurrent programs that can be expressed.

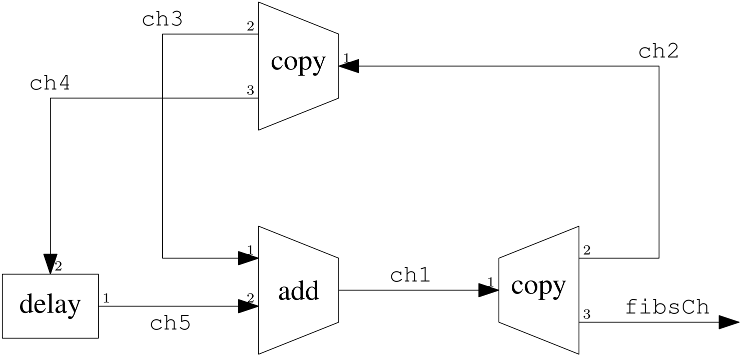
- fragility in the implementation of concurrency abstractions



Need for Selective Communication

When programming with `recv` and `send` exclusively, there are limits to the kinds of concurrent programs that can be expressed.

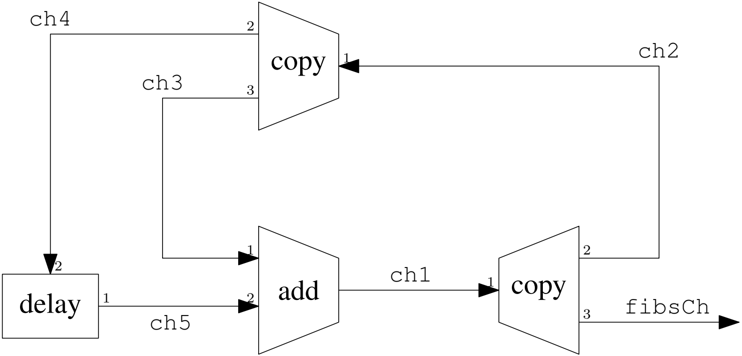
- fragility in the implementation of concurrency abstractions



Need for Selective Communication

When programming with `recv` and `send` exclusively, there are limits to the kinds of concurrent programs that can be expressed.

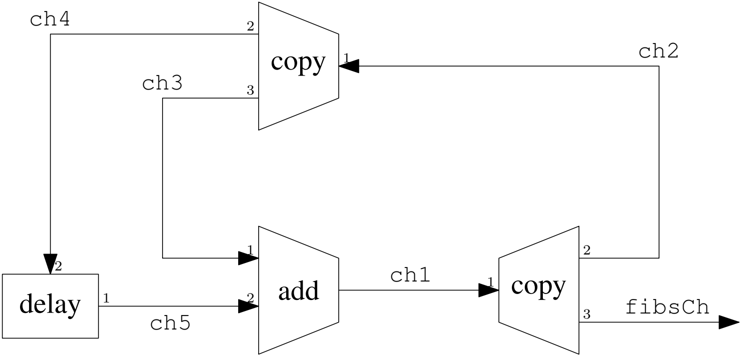
- fragility in the implementation of concurrency abstractions



Need for Selective Communication

When programming with `recv` and `send` exclusively, there are limits to the kinds of concurrent programs that can be expressed.

- fragility in the implementation of concurrency abstractions



- problem: *deadlock*
- solution: eliminate dependency on the order of blocking operations

Selective Communication

Selective communication

- allow a thread to block on a choice of several communications
- first communication that becomes *enabled* is chosen
- if two or more communications are simultaneously enabled, then one is chosen *nondeterministically*

Selective Communication vs. Abstraction

Selective communication vs. Abstraction

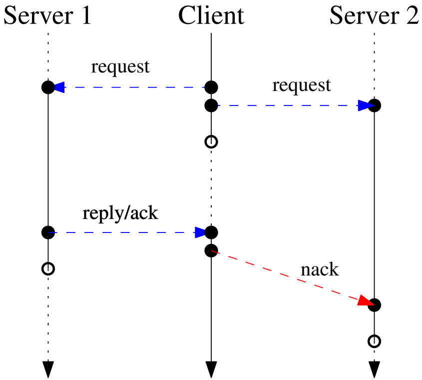
- in most concurrent languages with message passing, must explicitly list the blocking communications:

```
select inCh1?x => x + (recv inCh2)
     | inCh2?y => (recv inCh1) + y
     | outCh!42 => 0
```

- makes it difficult to construct abstract synchronous operations, because constituent `recv`s/`?` and `send`s/`!` must be revealed, breaking abstraction

Selective Communication vs. Abstraction

Consider a possible interaction between a client and two servers



Selective Communication vs. Abstraction

Consider a possible interaction between a client and two servers

Without abstraction, the code is a mess:

```
let val replCh1 = channel ()
    val nackCh1 = channel ()
    val replCh2 = channel ()
    val nackCh2 = channel ()
in
    send (reqCh1, (req1, replyCh1, nackCh1)) ;
    send (reqCh2, (req2, replyCh2, nackCh2)) ;
    select replCh1?repl1 => (setNack nackCh2 ; act1 repl1)
         | replCh2?repl2 => (setNack nackCh1 ; act2 repl2)
end
```

Want an abstraction mechanism that supports choice

First-class Synchronous Operations

First-class (abstract) synchronous operations (*Events*)

- decouple the description of a synchronous operation
  from the act of synchronizing

Events and synchronization

- an event value represents a potential synchronous operations
  (analogy: a function value represents a potential computation)

```
type 'a event
```

- force synchronization on an event value
  (analogy: application forces evaluation of a function value)

```
val sync : 'a event -> 'a
```

First-class Synchronous Operations

First-class (abstract) synchronous operations (*Events*)

- decouple the description of a synchronous operation from the act of synchronizing

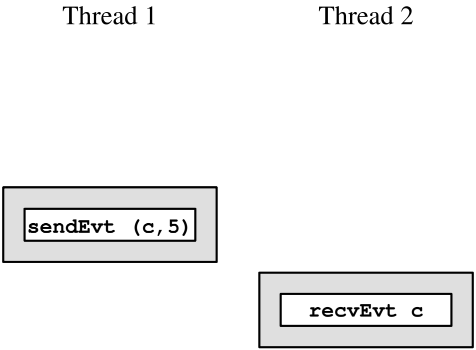
Base-event constructors

- event values that describe a primitive synchronous operation
- channel communication

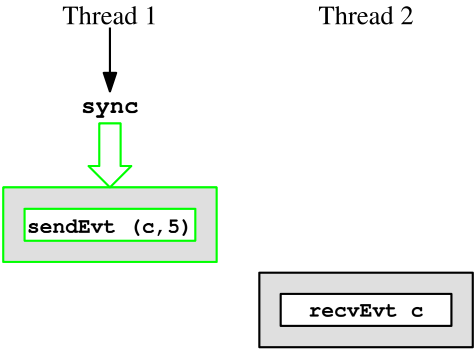
```
val recvEvt : 'a chan -> 'a event
val sendEvt : 'a chan * 'a -> unit event
```

```
val recv = fn ch => sync (recvEvt ch)
val send = fn (ch, x) => sync (sendEvt (ch, x))
```

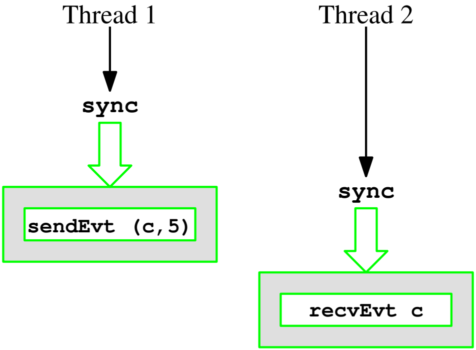
Base-event Constructors for Channel Communication



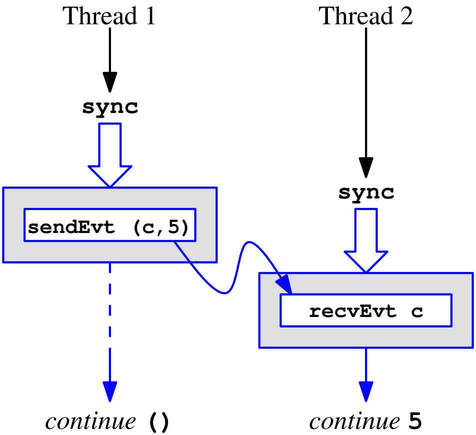
Base-event Constructors for Channel Communication



Base-event Constructors for Channel Communication



Base-event Constructors for Channel Communication



First-class Synchronous Operations

First-class (abstract) synchronous operations (*Events*)

- decouple the description of a synchronous operation from the act of synchronizing

Event combinators

- build more complicated event values from the base-event values
- generalized selective communication mechanism

```
val choose : 'a event * 'a event -> 'a event
```

- event wrapper for post-synchronization actions

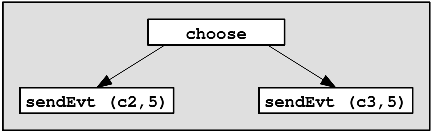
```
val wrap : 'a event * ('a -> 'b) -> 'b event
```

- event generator for pre-synchronization actions

```
val guard : (unit -> 'a event) -> 'a event
```

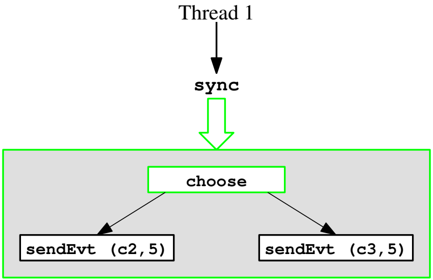
Event Combinator for Generalized Choice

```
val choose : 'a event * 'a event -> 'a event
```



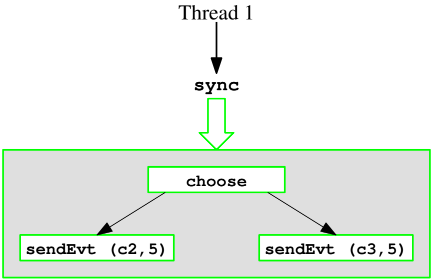
Event Combinator for Generalized Choice

```
val choose : 'a event * 'a event -> 'a event
```



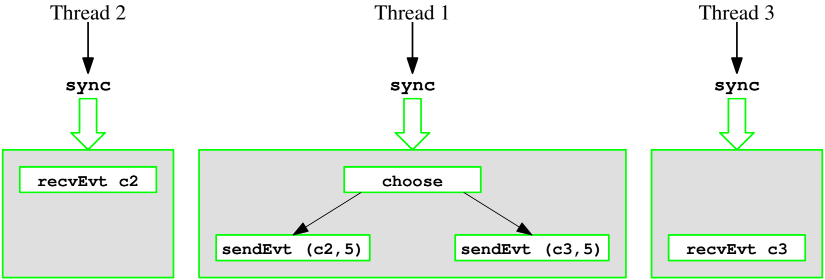
Event Combinator for Generalized Choice

```
val choose : 'a event * 'a event -> 'a event
```



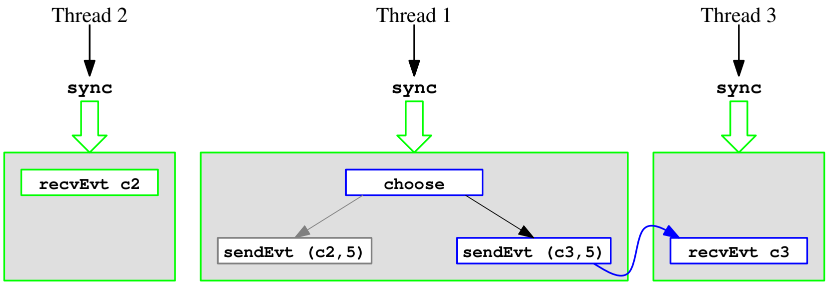
Event Combinator for Generalized Choice

```
val choose : 'a event * 'a event -> 'a event
```



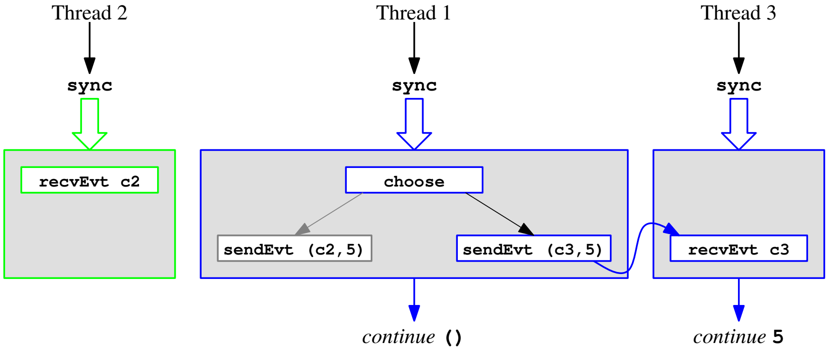
Event Combinator for Generalized Choice

```
val choose : 'a event * 'a event -> 'a event
```



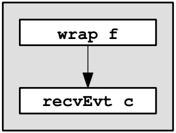
Event Combinator for Generalized Choice

```
val choose : 'a event * 'a event -> 'a event
```



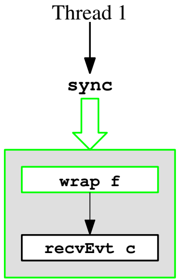
Event Combinator for Post-synchronization Actions

```
val wrap : 'a event * ('a -> 'b) -> 'b event
```



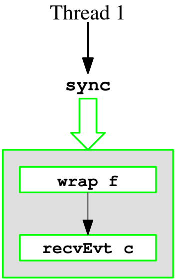
Event Combinator for Post-synchronization Actions

```
val wrap : 'a event * ('a -> 'b) -> 'b event
```



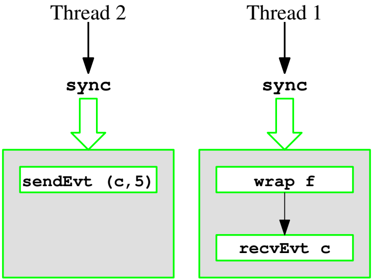
Event Combinator for Post-synchronization Actions

```
val wrap : 'a event * ('a -> 'b) -> 'b event
```



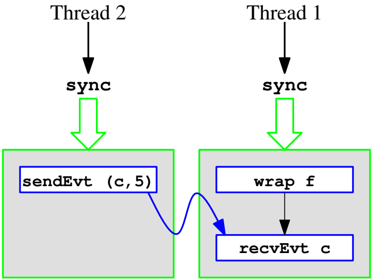
Event Combinator for Post-synchronization Actions

```
val wrap : 'a event * ('a -> 'b) -> 'b event
```



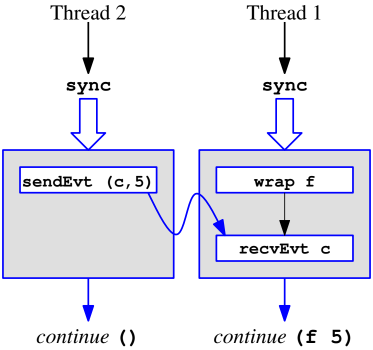
Event Combinator for Post-synchronization Actions

```
val wrap : 'a event * ('a -> 'b) -> 'b event
```



Event Combinator for Post-synchronization Actions

```
val wrap : 'a event * ('a -> 'b) -> 'b event
```



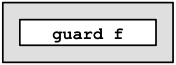
Using Event Combinators

```
fun addStrms (inCh1, inCh2, outCh) =
  forever () (fn () =>
    let val (a, b) =
          sync (choose (
            wrap (recvEvt inCh1, fn a => (a, recv inCh2)),
            wrap (recvEvt inCh2, fn b => (recv inCh1, b))
          ))
    in  send (a + b)
    end)

fun copyStrm (inCh, outCh1, outCh2) =
   forever () (fn () =>
     let val x = recv inCh
     in
          sync (choose (
            wrap (sendEvt (outCh1, x), fn () => send (outCh2, x)),
            wrap (sendEvt (outCh2, x), fn () => send (outCh1, x))
          ))
     end)
```

Event Combinator for Pre-synchronization Actions

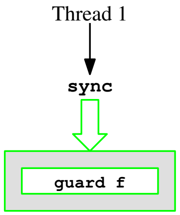
```
val guard : (unit -> 'a event) -> 'a event
```




```
fun f () =
  let val c = channel ()
      val _ = spawn (sync (sendEvt (c, 5)))
  in  recvEvt c
  end
```

Event Combinator for Pre-synchronization Actions

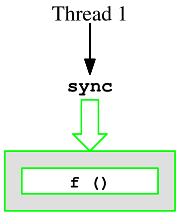
```
val guard : (unit -> 'a event) -> 'a event
```




```
fun f () =
  let val c = channel ()
      val _ = spawn (sync (sendEvt (c, 5)))
  in  recvEvt c
  end
```

Event Combinator for Pre-synchronization Actions

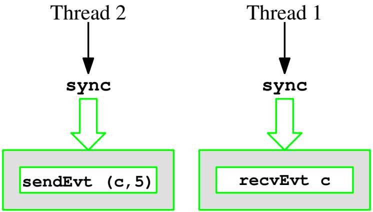
```
val guard : (unit -> 'a event) -> 'a event
```




```
fun f () =
  let val c = channel ()
      val _ = spawn (sync (sendEvt (c, 5)))
  in  recvEvt c
  end
```

Event Combinator for Pre-synchronization Actions

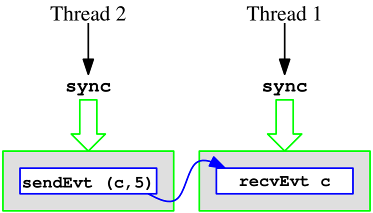
```
val guard : (unit -> 'a event) -> 'a event
```




```
fun f () =
  let val c = channel ()
      val _ = spawn (sync (sendEvt (c, 5)))
  in  recvEvt c
  end
```

Event Combinator for Pre-synchronization Actions

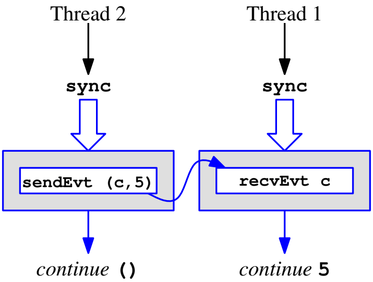
```
val guard : (unit -> 'a event) -> 'a event
```




```
fun f () =
  let val c = channel ()
      val _ = spawn (sync (sendEvt (c, 5)))
  in  recvEvt c
  end
```

Event Combinator for Pre-synchronization Actions

```
val guard : (unit -> 'a event) -> 'a event
```




```
fun f () =
  let val c = channel ()
      val _ = spawn (sync (sendEvt (c, 5)))
  in  recvEvt c
  end
```

Example: Swap Channels

Swap Channels

- a synchronous abstraction
- allows (exactly) two threads to swap values

```
signature SWAP_CHAN =
  sig
    type 'a swap_chan
    val swapChannel : unit -> 'a swap_chan
    val swapEvt     : 'a swap_chan * 'a -> 'a event
  end
```

Example: Swap Channels

```
structure BadSwapChan : SWAP_CHAN =
  struct
    datatype 'a swap_chan = SC of 'a chan

    fun swapChannel () = SC (channel ())

    fun swapEvt (SC ch, msgOut) =
      choose (
        wrap (recvEvt ch, fn msgIn =>
          (send (ch, msgOut) ; msgIn)),
        wrap (sendEvt (ch, msgOut), fn () =>
          recv ch)
      )
  end
```

Example: Swap Channels

```
structure SwapChan : SWAP_CHAN =
  struct
    datatype 'a swap_chan = SC of ('a * 'a chan) chan

    fun swapChannel () = SC (channel ())

    fun swapEvt (SC ch, msgOut) =
      guard (fn () =>
        let val inCh = channel ()
        in
            choose (
              wrap (recvEvt ch, fn (msgIn, outCh) =>
               (send (outCh, msgOut) ; msgIn)),
              wrap (sendEvt (ch, (msgOut, inCh)), fn () =>
               recv inCh)
            )
        end)
  end
```

Additional First-class Synchronous Operations

Base-event constructors

- event values that describe a primitive synchronous operation

- base-event constructors for trivial synchronizations

```
val alwaysEvt  : 'a -> 'a event
val neverEvt   : 'a event

val chooseList : 'a event list -> 'a event =
  fn l => foldl choose neverEvt l
```

Additional First-class Synchronous Operations

Event combinators

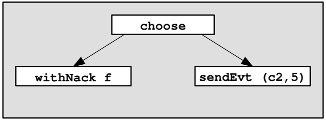
- build more complicated event values from the base-event values

- event generator for pre-synchronization actions with cancellation

```
val withNack : (unit event -> 'a event) -> 'a event
```

Event Combinator for Pre-sync Actions w/ Cancellation

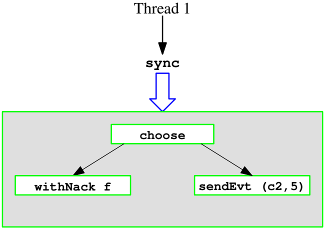
```
val withNack : (unit event -> 'a event) -> 'a event
```




```
fun f nackEvt =
  let val _ = spawn (sync (choose (
                          recvEvt c3, wrap (nackEvt, g))))
  in  sendEvt (c3, 5)
  end
```

Event Combinator for Pre-sync Actions w/ Cancellation

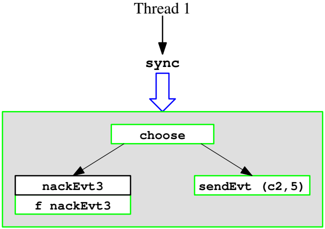
```
val withNack : (unit event -> 'a event) -> 'a event
```




```
fun f nackEvt =
  let val _ = spawn (sync (choose (
                    recvEvt c3, wrap (nackEvt, g))))
  in  sendEvt (c3, 5)
  end
```

Event Combinator for Pre-sync Actions w/ Cancellation

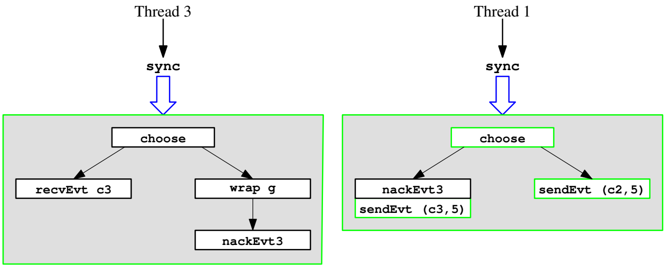
```
val withNack : (unit event -> 'a event) -> 'a event
```




```
fun f nackEvt =
  let val _ = spawn (sync (choose (
                      recvEvt c3, wrap (nackEvt, g))))
  in  sendEvt (c3, 5)
  end
```

Event Combinator for Pre-sync Actions w/ Cancellation

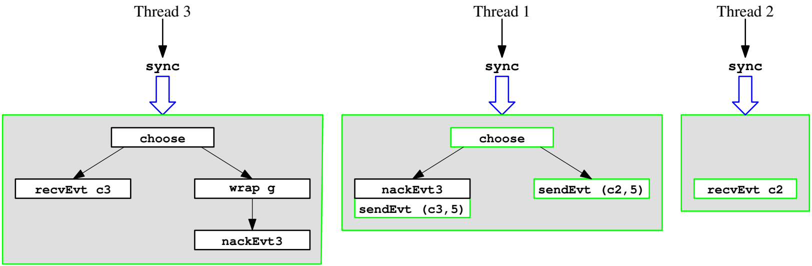
```
val withNack : (unit event -> 'a event) -> 'a event
```




```
fun f nackEvt =
  let val _ = spawn (sync (choose (
                        recvEvt c3, wrap (nackEvt, g))))
  in  sendEvt (c3, 5)
  end
```

Event Combinator for Pre-sync Actions w/ Cancellation

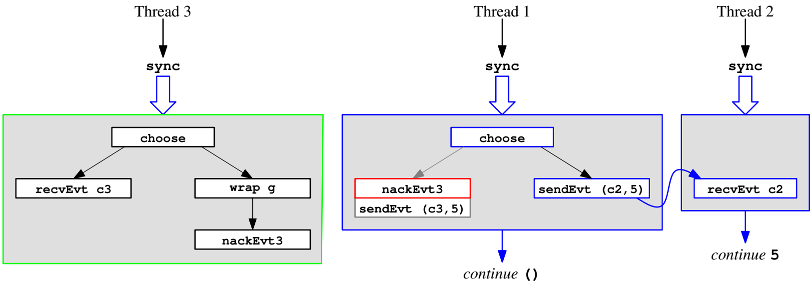
```
val withNack : (unit event -> 'a event) -> 'a event
```




```
fun f nackEvt =
  let val _ = spawn (sync (choose (
                       recvEvt c3, wrap (nackEvt, g))))
  in  sendEvt (c3, 5)
  end
```

Event Combinator for Pre-sync Actions w/ Cancellation

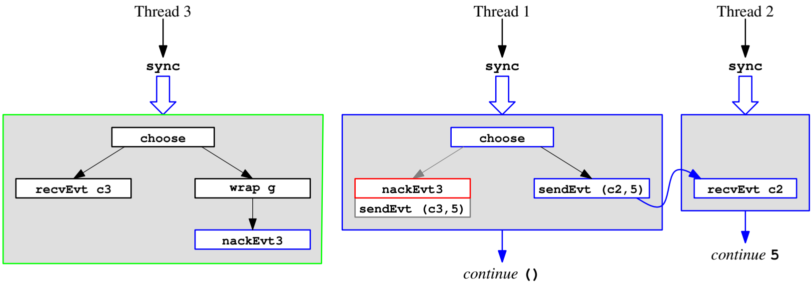
```
val withNack : (unit event -> 'a event) -> 'a event
```




```
fun f nackEvt =
  let val _ = spawn (sync (choose (
                         recvEvt c3, wrap (nackEvt, g))))
  in  sendEvt (c3, 5)
  end
```

Event Combinator for Pre-sync Actions w/ Cancellation

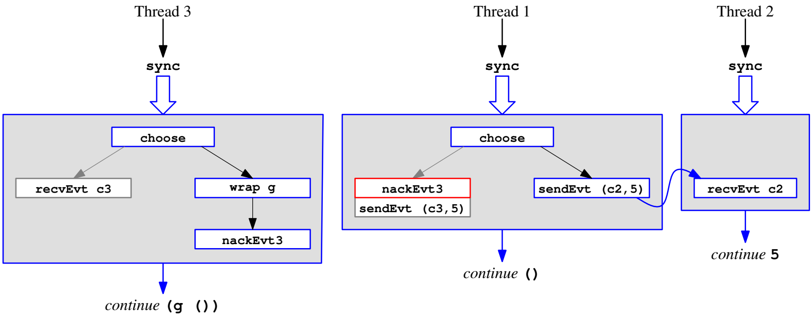
```
val withNack : (unit event -> 'a event) -> 'a event
```




```
fun f nackEvt =
  let val _ = spawn (sync (choose (
                      recvEvt c3, wrap (nackEvt, g))))
  in  sendEvt (c3, 5)
  end
```

Event Combinator for Pre-sync Actions w/ Cancellation

```
val withNack : (unit event -> 'a event) -> 'a event
```




```
fun f nackEvt =
  let val _ = spawn (sync (choose (
                    recvEvt c3, wrap (nackEvt, g))))
  in  sendEvt (c3, 5)
  end
```

Selective Communication vs. Abstraction

Consider a possible interaction between a client and two servers

Without abstraction, the code is a mess:

```
let val replCh1 = channel ()
    val nackCh1 = channel ()
    val replCh2 = channel ()
    val nackCh2 = channel ()
in
    send (reqCh1, (req1, replyCh1, nackCh1)) ;
    send (reqCh2, (req2, replyCh2, nackCh2)) ;
    select replCh1?repl1 => (setNack nackCh2 ;
                             act1 repl1)
         | replCh2?repl2 => (setNack nackCh1 ;
                             act2 repl2)
end
```

Selective Communication vs. Abstraction

Consider a possible interaction between a client and two servers
With abstraction, the code is clean:

```
structure Server : sig
  val rpcEvt : server * req -> repl event
end = struct
  fun rpcEvt (srv, req) =
    withNack (fn nack =>
      let val replyCh = channel
      in
          ... send (reqCh, (req, replyCh, nack)) ... ;
          recvEvt replyCh
      end)
end

sync (choose (
  wrap (Server.rpcEvt server1, fn repl1 => act1 repl1),
  wrap (Server.rpcEvt server2, fn repl2 => act2 repl2)
))
```

External Synchronous Events

Motivations for concurrent programming:

- application domains with naturally concurrent structure:
  - interactive systems (e.g., graphical-user interfaces)

Interactive systems

- multiple (asynchronous) input streams
  - keyboard, mouse, network
- multiple (asynchronous) output streams
  - display, audio, network

In sequential languages, dealt with through complex event loops and callback functions

First-class synchronous events can treat these external events using the same framework as internal synchronization

External Synchronous Events: Input/Output

For a console application, take standard input, output, and error streams to be character channels

```
val stdInCh  : char chan
val stdOutCh : char chan
val stdErrCh : char chan
```

Better interface is to expose the streams as events

- should only `recv` from standard input stream
- should only `send` to standard output and error streams

```
val stdInEvt  : char event
val stdOutEvt : char -> unit event
val stdErrEvt : char -> unit event
```

In practice, build higher-level I/O library on top

External Synchronous Events: Timeouts

Mechanisms for “timing out” on a blocking operation

```
val timeOutEvt : time -> unit event
val atTimeEvt  : time -> unit event
```

Pause for one second

```
sync (timeOutEvt (timeFromSeconds 1))
```

Prompt for `Y/N` with default

```
choose (
  wrap (timeOutEvt (timeFromSeconds 10), fn () => #"N"),
  stdInEvt
)
```

Examples

Two final examples

- Buffered channels
- Futures

Example: Buffered Channels

Sometimes useful to support asynchronous communication

- sender does not block, message is buffered in the channel
- receiver blocks until there is an available message

```
signature BUFFERED_CHAN =
  sig
    type 'a buff_chan
    val buffChannel : unit -> 'a buff_chan
    val buffSend    : 'a buff_chan * 'a -> unit
    val buffRecvEvt : 'a buff_chan -> 'a event
  end
```

Example: Buffered Channels

```
structure BufferedChan : BUFFERED_CHAN =
  struct
    datatype 'a buff_chan =
      BC of {inCh: 'a chan, outCh: 'a chan}

    fun buffSend (BC {outCh, ...}, x) =
      send (outCh, x)

    fun buffRecvEvt (BC {inCh, ...}) =
      recvEvt inCh
```

Example: Buffered Channels

```
  fun buffChannel () : 'a buff_chan =
    let val (inCh, outCh) = (channel (), channel ())
        fun loop ([], []) = loop ([recv inCh], [])
          | loop ([], rear) = loop (rev rear, [])
          | loop (front as frHd::frTl, rear) =
              (loop o sync o choose) (
                wrap (recvEvt inCh, fn y =>
                  (front, y::rear)),
                wrap (sendEvt (outCh, frHd), fn () =>
                  (frTl, rear))
              )
        val _ = spawn (loop ([], []))
    in  BC {inCh = inCh, outCh = outCh}
    end
end
```

Example: Futures

Futures: a common mechanism for specifying parallel computation

- future creation: takes a computation, creates a separate thread and returns a placeholder (*future cell*)
- future touching: read a value from a future cell, blocking until value is computed

```
signature FUTURE =
  sig
    datatype 'a result = VAL of 'a | EXN of exn
    val future : ('a -> 'b) -> 'a -> 'b result event
  end
```

Example: Futures

```
structure Future : FUTURE =
  struct
    datatype 'a result = VAL of 'a | EXN of exn
    fun future f x =
      let val ch = channel ()
          let val _ = spawn (
            let val r = (VAL (f x)) handle exn => EXN exn
            in  forever () (fn () => (send (ch, r)))
            end)
      in  recvEvt ch
      end
```

Conclusion

Explicit-concurrency mechanisms in Manticore

- support concurrent programming (systems programming)
- unified interface to synchronization
  via first-class synchronous operations

More sophisticated applications

- graphical-user interface toolkit (eXene)
- distributed tuple-space implementation
- software build system

Next: Implicit Parallelism in Manticore

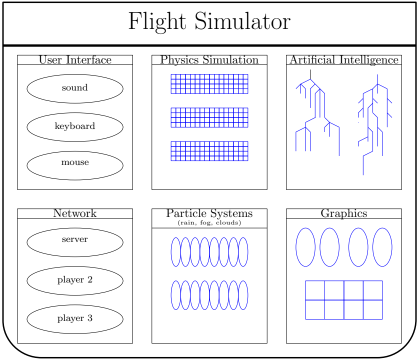
Part III

Implicit Parallelism in Manticore

Introduction

Language mechanisms for *implicitly-threaded* parallelism

- programmer annotates fine-grained parallel computations
- compiler and runtime map onto parallel threads



Introduction

Motivations for implicitly-threaded parallelism:

- improve performance by exploiting multiprocessors
- ease the burden for both programmer and compiler
  - programmer able to utilize simple parallel constructs:
    efficiently (in terms of program text) express the desired parallelism
  - compiler able to analyze and optimize simple parallel constructs:
    efficiently (in terms of time and computational resources) execute

Implicitly-threaded parallelism is more specific (and less expressive) than explicitly-threaded concurrency, but

- express common idioms of parallel computation
- limited expressiveness allows the compiler and runtime
  to better manage the parallel computation

Introduction

Manticore provides several light-weight syntactic forms for introducing implicitly-parallel computations.

These forms are *hints* to the compiler and runtime that a computation is a good candidate for parallel execution.

- *Parallel arrays*: fine-grain data-parallel computations over seqs
- *Parallel tuples*: basic fork-join parallel computation
- *Parallel bindings*: data-flow and work-stealing parallelism
- *Parallel case*: non-deterministic speculative parallelism
- *Cancellation*: unused/abandoned subcomputations

Introduction

Manticore provides several light-weight syntactic forms for introducing implicitly-parallel computations.

The semantics of (most of) these constructs is sequential:

- provides programmer with a deterministic programming model
- formalizes the expected behavior of the compiler/runtime
- if subcomputation raises an exception, then delay delivery until a sequentially prior subcomputations have terminated
- if subcomputation performs synchronization (message-passing), then execute sequentially
- compiler/runtime may choose to execute in a single thread

Parallel Arrays

Support for parallel computations on arrays and matrices
is common in parallel languages.

Operations on arrays and matrices naturally express *data parallelism*

- a single computation is performed in parallel across a large number of data elements

Manticore adopts the nested parallel array mechanism (NESL)

```
type 'a parray
```

- immutable sequences that can be computed in parallel
- *nested data parallelism*
  - arbitrary element types: arrays of floating-point numbers, arrays of user-defined datatypes, arrays of arrays

Parallel-Array Introduction

Basic expression forms for creating parallel arrays

- Explicit enumeration of expressions

```
[| e1, ..., en |]
```

- Integer enumeration

```
[| el to eh by es |]
```

  - `el`: start integer (low)
  - `eh`: end integer (high)
  - `es`: step integer (optional)
  - Example `[| 1 to 31 by 10 |]` evaluates to `[| 1, 11, 21, 31 |]`

Parallel-Array Introduction

Basic expression forms for creating parallel arrays

- Parallel-array comprehension

```
[| e | x1 in ea1, ..., xn in ean where ep |]
```

  - `e`: computes elements of the array
  - `ea`$_i$: parallel-array expressions that provide inputs
  - `ep`: boolean expression that filters input (optional)

  - zip semantics (not Cartesian-product semantics)
    - if the input arrays `ea`$_i$ have different lengths, then all are truncated to length of the shortest input and processed in lock-step

Parallel-Array Comprehension

Examples

- Double each positive integer in a parallal array of integers `num`

```
[| 2 * n | n in nums where n > 0 |]
```

- Parallel map and parallel filter combinators

```
fun mapP f xs = [| f x | x in xs |]
fun filterP p xs = [| x | x in xs where p x |]
```

- Inner loop of ray tracer (nested data parallelism)

```
[| [| trace (x, y) | x in [| 0 to w-1 |] |]
                   | y in [| 0 to h-1 |] |]
```

Parallel-Array Elimination

Basic expression forms for consuming parallel arrays

- Parallel-array comprehension
- Subscript operation

```
ea ! ei
```

  - `ea`: parallel-array expression
  - `ei`: integer expression

  - parallel arrays are indexed by zero
  - if the index is outside the range of the array then the `Subscript` exception is raised
  - random-access may not be constant time

Parallel-Array Elimination

Basic expression forms for consuming parallel arrays

- Parallel-array reduction

```
reduceP f b ea
```

  - `f`: binary function, should be associative
  - `b`: base value, should be zero of `ef`
  - `ea`: parallel-array expression
  - similar to folding `f` over the elements of `ea`, using the base value `b`
  - function is applied in parallel to elements, using a tree-like decomposition of array
  - Example

```
fun sumP xs = reduceP (fn (x, y) => x + y) 0 a
```

Additional Parallel-Array Operations

Additional combinators for manipulating parallel arrays

- Size of parallel arrays

```
val lengthP : 'a parray -> int
```

- Concatenate and flatten parallel arrays

```
val concatP : 'a parray * 'a parray -> 'a parray
val flattenP : 'a parray parray -> 'a parray
```

These combinators have direct implementations for efficiency, but consider implementing them in terms of the previous forms.

Additional Parallel-Array Operations

- Size of parallel arrays

```
fun lengthP a = sumP (mapP (fn _ => 1) a)
```

- Concatenate and flatten parallel arrays

```
fun concatP (a1, a2) =
  let val l1 = lengthP a1
      val l2 = lengthP a2
  in
      [| if i < l1 then a1 ! i else a2 ! (i - l1)
         | i in [| 0 to (l1 + l2 - 1) |] |]
  end
fun flattenP a = reduceP concatP [| |] a
```

Examples

Three examples

- Image manipulation
- Sparse-matrix vector multiplication
- Quicksort

Example: Image Manipulation

Parallel arrays are a natural representation for images:

```
type pixel = int * int * int
type img = pixel parray parray
```

Image transformations expressed as a computation that is applied to each pixel of an image

```
fun xformImg xformPix img =
  [| [| xformPix pix | pix in row |] | row in img |]

fun rgbPixToGrayPix ((r, g, b) : pixel) : pixel =
  let val m = (r + g + b) / 3
  in  (m, m, m)
  end
fun rgbImgToGrayImg (img : img) : img =
  xformImg rgbPixToGrayPix img
```

Example: Sparse-matrix Vector Multiplication

Parallel arrays can represent both dense and sparse vectors and matrices:

```
type vector = real parray
type sparse_vector = (int * real) parray

type sparse_matrix = sparse_vector parray
```

To multiply a sparse matrix by a dense vector, compute the dot product for each row:

```
fun dotp (sv: sparse_vector) (v: vector) : real =
  sumP [| x * (v!i) | (i,x) in sv |]
fun smvm (sm: sparse_matrix) (v: vector) : vector =
  [| dotp (row, v) | row in sm |]
```

Example: Quicksort

Quicksort an array of integers:

```
fun quicksort (a: int parray) : int parray =
  if lengthP a < 2
    then a
    else let val pivot = ns ! 0
             val ss = [| filterP cmp a
                         | cmp in [| fn x => x < pivot,
                                     fn x => x = pivot,
                                     fn x => x > pivot |] |]
             val rs = [| quicksort a | a in [| ss!0, ss!2 |] |]
             val sorted_lt = rs!0
             val sorted_eq = ss!1
             val sorted_gt = rs!1
         in  flattenP [| sorted_lt, sorted_eq, sorted_gt |]
         end
```

Some awkwardness in using parallel arrays exclusively

Parallel Tuples

Parallel arrays provide a very *regular* form of parallelism.

Sometimes more convenient to express *irregular* forms of parallelism.

Parallel-tuple expression form provides a simple *fork/join* parallelism:

```
(| e1, ..., en |)
```

- each of the tuple components is evaluated in parallel
- computation of the tuple result blocks
  until all of the tuple components are fully evaluated

Example: Quicksort

Quicksort an array of integers:

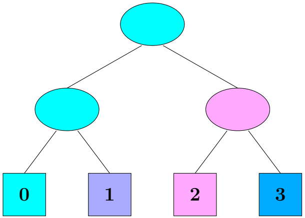
```
fun quicksort (a: int parray) : int parray =
  if lengthP a < 2
    then a
    else let val pivot = ns ! 0
             val (sorted_lt, sorted_eq, sorted_gt) =
               (| quicksort (filterP (fn x => x < pivot) a),
                  filterP (fn x => x = pivot) a,
                  quicksort (filterP (fn x => x > pivot) a) |)
         in  flattenP [| sorted_lt, sorted_eq, sorted_gt |]
         end
```

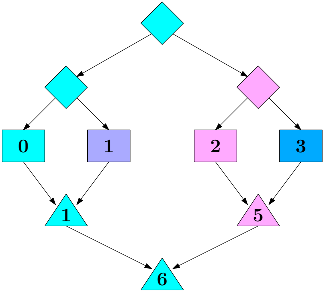
More natural using parallel tuples

Parallel Tuples

Consider adding the leaves of a binary tree.

```
datatype tree = LF of int | ND of tree * tree
fun treeAdd t =
  case t of
     LF n => n
   | ND(t1, t2) => add (| treeAdd t1, treeAdd t2 |)
```





Parallel Tuples

Very easy to express parallel computations

Express more parallelism than can be effectively utilized

- compiler and runtime must determine when the overhead of starting a parallel execution doe not outweigh the benefits of parallel execution

Adding *all* branches of a binary tree in parallel

- balanced binary tree of depth N yields $2^N - 2$ parallel computations
- Realizing each as a separate thread yields more threads than physical processors
- Realizing each as a unit of work for work-stealing threads incurs overhead

Parallel Tuples: Future Work

Use compiler to transform to a semantically equivalent program.

```
datatype tree = Tr of int * tree'
     and tree' = LF of int | ND of tree * tree
fun Lf n = Tr (1, Lf' n)
fun Br (t1 as Tr (d1, _), t2 as Tr (d2, _)) =
  Tr (max (d1, d2) + 1, Br' (t1, t2))

fun trAdd (Tr (d,t')) =
  if d < 16 orelse numIdleProcs () < 2
    then tr'Add_seq t'
    else tr'Add_par t'
and trAdd_seq (Tr (_,t')) = tr'Add_seq t'
and tr'Add_seq' =
  case t' of
     Lf' n => n
   | Br' (t1, t2) => add ( trAdd_seq t1, trAdd_seq t2 )
and tr'Add_par' =
  case t' of
     Lf' n => n
   | Br' (t1, t2) => add (| trAdd t1, trAdd t2 |)
```

Parallel Bindings

Parallel arrays and parallel tuples provide fork/join parallelism.

Sometimes want more flexible scheduling of computations.

Parallel-binding declaration form provides *speculative* parallelism:

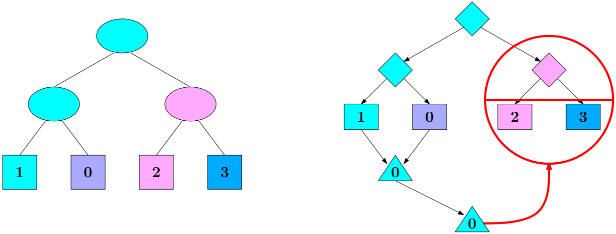
```
pval p = e
```

- spawns the evaluation of the expression as a parallel thread
- evaluation forced when a variable in the pattern is demanded
  - exception raised by evaluation is raised at the point of use
- evaluation cancelled when no variable will be demanded

Parallel Bindings

Consider multiplying the leaves of a binary tree.

```
fun trMul t =
  case t of
      LF n => n
    | ND(t1, t2) =>
        let pval p2 = trMul t2
            val p1 = trMul t1
        in  if p1 = 0 then (* cancel p2 *) 0 else p1 * p2
        end
```



Parallel Bindings

Cancellation performed by a simple, syntactic analysis.

A parallel binding expression may inherit other parallel bindings

```
let
  pval x = f 0
  pval y = (| g 1, g 2 + x |)
in
  if b
    then (* cancel y *) x
    else (* cannot cancel x *) h y
end
```

Parallel Bindings

Behavior of parallel tuples may be encoded using parallel bindings.

Encode

```
(| e1, ..., en |)
```

as

```
let
  pval x1 = e1
  ...
  pval xn = en
in
  (x1, ..., xn)
end
```

Parallel Cases

Pattern matching is a fundamental functional-programming idiom.
Parallel-case expression form provides speculative
and *nondeterministic* pattern matching

```
pcase e1 & ... & en of
   pp11 & ... & pp1n => e'1
 | ...
 | ppm1 & ... & ppmn => e'm
 | otherwise => eo
```

- expressions $\mathtt{e}_i$ are evaluated in parallel and cancelled in matches
- $\mathtt{pp}_{i,j}$ are *parallel patterns*
  - a nondeterministic wildcard pattern **`?`**
  - a handle pattern **`handle`** `p`
  - a (normal, SML) pattern `p`
- **`otherwise`** branch (optional) has lowest precedence

Parallel Cases:

Parallel patterns

- A nondeterministic wildcard pattern **?** always matches, even if the corresponding scruitinee is still evaluating.
- A handle pattern **handle** `p` matches a computation that raises an exception; the pattern `p` is bound to the raised exception.

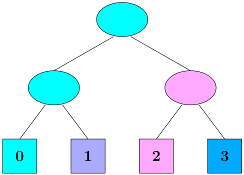
otherwise `=> eo` branch

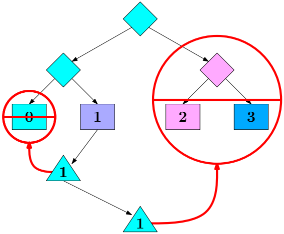
- If present, equivalent to
  `(_ | handle _) &... &(_ | handle _) => eo`,
  but with lowest precedence
- If absent, defaults to **otherwise** `=>` **raise** `Match`

Parallel Cases

Consider picking an arbitrary leaf that satisfies a predicate.

```
fun trFind (p, t) =
  case t of
     LF n => if p n then SOME n else NONE
   | Br (t1, t2) =>
       (pcase trFind (p, t1) & trFind (p, t2) of
           SOME n & ? => SOME n
         | ? & SOME n => SOME n
         | NONE & NONE => NONE)
```





Parallel Cases

Consider multiplying the leaves of a binary tree.

```
fun trMul t =
  case t of
      LF n => n
    | Br (t1, t2) =>
        (pcase trMul t1 & trMul t2 of
            0 & ? => 0
          | ? & 0 => 0
          | x & y => x * y)
```

Parallel Cases

A number of derived forms are desugared to use **`pcase`**

```
       e1 |?| e2  ≡  pcase e1 & e2 of
                        n & ? => n
                      | ? & n => n

  e1 |andalso| e2  ≡  pcase e1 & e2 of
                        false & ? => false
                      | ? & false => false
                      | true & true => true

   e1 |orelse| e2  ≡  pcase e1 & e2 of
                        true & ? => true
                      | ? & true => true
                      | false & false => false
```

Exceptions

Exceptions follow the sequential semantics.

- the exception raised by an expression is precise and well-defined

```
fun f n = if n = 100 then raise Foo else ...
fun g n = if n = 100 then raise Goo else ...

(| f 100, g 100 |)
handle Foo => ... (* handled Foo *)
     | Goo => ... (* unreachable *)
```

Requires a slightly more restrictive implementation
of the implicitly-threaded parallel constructs,
but the precise semantics is crucial for systems programming.

Exceptions

Exceptions follow the sequential semantics.

Implementation uses compensation code
to propagate the correct exception.

- can simplify compensation code with program analyses

```
[| ... raise Foo ..., ... raise Foo ... |]
```

Implementation cancels abandoned computations.

- free computational resources devoted to computations

```
(| ... raise Foo ..., fact(100), fib(100) |)
```

Exceptions

Exceptions follow the sequential semantics.

Implementation cancels abandoned computations.

```
fun f n = ... raise Foo ...
fun g n = ... raise Goo ...
fun h n = ... raise Hoo ...

let pval x = add (| f 200, g 200 |)
    pval y = mul (| f 300, g 300 |)
in  [| if h z then x + z else y * z | z in zs |]
end
```

Multiple forms of parallelism with cross-cutting concerns motivates the need for a common, but flexible, runtime scheduling framework.

Examples

Two final examples

- Parallel Type-checking and Evaluation
- Parallel Game Search

Example: Parallel Type-checking and Evaluation

Simple programming language

- type-check and evaluate in parallel
- parallel type-checking
- parallel evaluation

Example: Parallel Type-checking and Evaluation

Types and expressions

```
datatype ty = NatTy | BoolTy | ArrowTy of ty * ty

datatype exp = Exp of loc * term
     and term = NatTerm of int
              | AddTerm of exp * exp
              | BoolTerm of bool
              | IfTerm of exp * exp * exp
              | VarTerm of var
              | LetTerm of var * exp * exp
              | LamTerm of var * ty * exp
              | AppTerm of exp * exp
              | ...
```

Example: Parallel Type-checking and Evaluation

Comparing types for equality

```
fun tyEq (ty1, ty2) =
  case (ty1, ty2) of
     (BoolTy, BoolTy) => true
  | (NatTy, NatTy) => true
  | (ArrowTy (ty1a, ty1r), ArrowTy (ty2a, ty2r)) =>
      (pcase tyEq (ty1a, ty2a) & tyEq (ty1r, ty2r) of
           false & ? => false
         | ? & false => false
         | true & true => true)
  | _ => false
```

Example: Parallel Type-checking and Evaluation

Comparing types for equality

```
fun tyEq (ty1, ty2) =
  case (ty1, ty2) of
     (BoolTy, BoolTy) => true
   | (NatTy, NatTy) => true
   | (ArrowTy (ty1a, ty1r), ArrowTy (ty2a, ty2r)) =>
       tyEq (ty1a, ty2a) |andalso| tyEq (ty1r, ty2r)
   | _ => false
```

Example: Parallel Type-checking and Evaluation

Parallel type-checker

- if well-typed, then report type
- if ill-typed, then report one error

```
datatype 'a res = Ans of 'a | Err of loc
val typeOfExp : env * exp -> ty res
```

Example: Parallel Type-checking and Evaluation

Parallel type-checker

```
fun typeOfExp (G, e as Exp (loc, term)) =
  case term of
     NatTerm _ => Ans NatTy
   | AddTerm (e1, e2) =
       let pval rty2 = typeOfExp (G, e2)
       in
           case typeOfExp (G, e1) of
              Ans NatTy =>
                (case rty2 of
                    Ans NatTy => Ans NatTy
                  | Ans _ => Err (locOf e2)
                  | Err loc => Err loc)
            | Ans _ => Err (locOf e1)
            | Err loc => Err loc
       end
```

Example: Parallel Type-checking and Evaluation

Parallel type-checker

```
| BoolTerm _ => Ans BoolTy
| IfTerm (e1, e2, e3) =
    let pval rty2 = typeOfExp (G, e2)
        pval rty3 = typeOfExp (G, e3)
    in
        case typeOfExp (G, e1) of
           Ans BoolTy =>
             (case (rty2, rty3) of
                 (Ans ty2, Ans ty3) =>
                   if tyEq (ty2, ty3)
                     then Ans ty2
                     else Err (locOf e)
               | (Err loc, _) => Err loc
               | (_, Err loc) => Err loc)
         | Ans _ => Err (locOf e1)
         | Err loc => Err loc
    end
```

Example: Parallel Type-checking and Evaluation

Parallel type-checker

```
| ApplyTerm (e1, e2) =
    let pval rty2 = typeOfExp (G, e2)
    in
        case typeOfExp (G, e1) of
           Ans (ArrowTy (ty11, ty12)) =>
             (case rty2 of
                  Ans ty2 =>
                    if tyEq (ty2, ty11)
                      then Ans ty12
                      else Err (locOf e2)
                | Err loc => Err loc)
         | Ans _ => Err (locOf e1)
         | Err loc => Err loc
    end
```

Example: Parallel Type-checking and Evaluation

Parallel type-checker

```
| VarTerm var =>
    (case envLookup (G, var) of
        NONE => Err (locOf e)
      | SOME ty => Ans ty)
| LamTerm (var, ty, e) =>
    (case typeOfExp (envExtend (G, (var, ty)), e) of
        Ans ty' => Ans (ArrowTy (ty, ty'))
      | Err loc => Err loc)
```

No obvious parallelism, but representation of the environment (e.g., as a balanced binary tree) may enable parallelism in the `envLookup` and `envExtend` functions.

Example: Parallel Type-checking and Evaluation

Parallel substitution

```
fun substExp (t, x, e as Exp (p, t')) =
  Exp (p, substTerm (t, x, t'))
and substTerm (t, x, t') =
  case t' of
     NumTerm n => NumTerm n
    | AddTerm (e1, e2) =>
       AddTerm (| substExp (t, x, e1),
                  substExp (t, x, e2) |)
    | BoolTerm b => BoolTerm b
    | IfTerm (e1, e2, e3)
       IfTerm (| substExp (t, x, e1),
                 substExp (t, x, e2),
                 substExp (t, x, e3) |)
    (* ... *)
```

Example: Parallel Type-checking and Evaluation

Parallel evaluation

```
exception EvalError

fun evalExp (p, t) =
  case t of
      NumTerm n => NumTerm n
    | AddTerm (e1, e2) =>
        (pcase evalExp e1 & evalExp e2 of
            NumTerm n1 & NumTerm n2 => NumTerm (n1 + n2)
          | otherwise => raise EvalError)
```

Example: Parallel Type-checking and Evaluation

Parallel evaluation

```
| IfTerm (e1, e2, e3) =>
    let pval v2 = evalExp e2
        pval v3 = evalExp e3
    in
        case evalExp e1 of
           BoolTerm true => v2
         | BoolTerm false => v3
         | _ => raise EvalError
    end
```

Abandoned branch is implicitly cancelled, even if it raises `EvalError`.

Example: Parallel Type-checking and Evaluation

Parallel type-checking and evaluation

```
fun typedEval e : term res =
  pcase typeOfExp (emptyEnv, e) & evalExp e of
      Err loc & ? => Err loc
    | Ans _ & v => Ans v
```

Evaluation is cancelled if type-checking returns `Err`.

Example: Parallel Game Search

Construct a tic-tac-toe game tree using minimax

Represent players and boards

```
datatype player = X | O
type board = player option parray   (* 9 elements *)
```

Represent a game tree as a rose tree

```
datatype 'a rose_tree = RoseTree of 'a * 'a rose_tree parray

(*  1 iff X has winning position *)
(*  0 iff tie                    *)
(* ~1 iff O has winning position *)
type ttt_game_tree = (board * int) rose_tree
```

Example: Parallel Game Search

Construct a tic-tac-toe game tree using minimax

Generate the next boards

```
fun availPositions (b: board) : int parray =
  [| i | s in b, i in [| 0 to 8 |] where isNone s |]

fun succBoards (b: board, p: player) : board parray =
  [| mapP (fn j => if i = j then SOME p else b!j) [| 0 to 8 |]
     | i in availPositions b |]
```

Generate the next boards

```
(* SOME  1 iff X wins      *)
(* SOME  0 iff tie         *)
(* SOME ~1 iff O wins      *)
(* NONE    iff incomplete  *)
fun boardScore (b: board) : int option = ...
```

Example: Parallel Game Search

Construct a tic-tac-toe game tree using minimax

```
fun maxP a = reduceP (fn (x, y) => max (x, y)) ~1 a
fun minP a = reduceP (fn (x, y) => min (x, y)) 1 a

fun treeScore (RoseTree (_, s)) = s

fun minimax (b: board, p: player) : ttt_game_tree =
  case boardScore b of
      SOME s => RoseTree ((b, s), [| |])
    | NONE =>
        let val ss = succBoards (b, p)
            val ch = [| minimax (b, flipPlayer p) | b in ss |]
            val chScores = [| treeScore t | t in ch |]
        in
            case p of
               X => RoseTree ((b, maxP chScores), ch)
             | O => RoseTree ((b, minP chScores), ch)
        end
```

Conclusion

- Implicit-parallelism mechanisms in Manticore
  - simple mechanisms — by design!
  - light-weight syntactic hints of available parallelism
    - relieves programmer of orchestrating the computation
  - parallel-tuples, parallel-bindings, and parallel-cases allow parallelism to be expressed in a familiar style
- Next: Implementation of Manticore

Part IV

Implementation of Manticore

Overview

Initial implementation of the Manticore system

- consisting of a compiler and a runtime system
- targetting x86-64 architecture under Linux and MacOS X

- most of the parallel features implemented
- current implementation efforts focused on testing and bug fixing

Significant aspect of the system is a runtime model designed to support multiple scheduling policies in a common framework.

Process Abstractions

Runtime model is based on *heap-allocated first-class* continuations

- creating a continuation is fast and small
- continuations are values; avoids race conditions in the scheduler

Runtime model has three distinct notions of process abstraction

Fibers correspond to unadorned flows of sequential control; a suspended fiber is represented as a unit continuation.

Threads created by `spawn` and assigned unique thread id; a thread may consist of multiple fibers

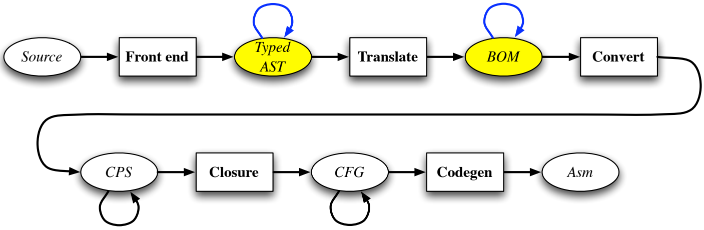
Virtual Processors (VProcs) correspond to a computation resource; each VProc is hosted by its own `pthread` and assigned to a physical core

Runtime maintains a dynamic binding between fibers and fiber-local storage (FLS).

Manticore Compiler

Compiler organized as a series of transformations between IRs:

Typed AST	explicitly-typed, polymorphic, abstract-syntax tree
BOM	direct-style, normalized, λ-calculus
CPS	continuation-passing-style λ-calculus
CFG	first-order control-flow graph



AST Optimizations

AST — explicitly-typed, polymorphic, abstract-syntax tree

- Compilation of pattern matching
- Introduce compensation code for exceptions
- Introduction of futures for implicitly-threaded parallelism
- Some flattening of nested-data parallelism (AOS to SOA)

BOM Intermediate Representation

BOM — direct-style, normalized, λ-calculus

- First-class continuations with a binding form that reifies the current continuation
- Simplified datatypes with simple pattern matching
  - allow BOM code to be independent of datatype represenations
- High-level operators, used to abstract over the implementation of various higher-level operations (thread creation, message passing, etc.)
  - rewriting rules for high-level operators to implement various optimizations
- Atomic operations (such as *compare-and-swap* (**cas**))

BOM Continuations

The **`cont`** binding

```
let cont k x = e in body end
```

binds `k` to the first-class continuation

```
fn x => (throw k' e)
```

where `k'` is the continuation of the whole expression

Scope of `k` includes both the expression `body` and the expression `e`

- `k` may be recursive

BOM Continuations

Traditional `callcc` function:

```
fun callcc f = let cont k x = x in f k end
```

Create a fiber (unit continuation) from a function:

```
fun fiber f =
  let
    cont k () = ( f () ; @stop () )
  in
    k
  end
```

where `@stop` returns control to the scheduler.

BOM Optimizations

BOM — direct-style, normalized, λ-calculus

- Standard functional-PL compiler optimizations
  - uncurrying, inlining, contraction
- High-level operator expansion
  - BOM types and code
    - embedded in Manticore modules
    - loaded at compile time
    - introduced by translations
  - used to implement concurrency and parallel features
  - used to import and implement scheduling code

HLOp Expansion

Manticore source (AST):

```
spawn e
```

Translated to (BOM):

```
let fun f_thnk (z: unit) : unit = e'
    val tid : tid = @spawn (f_thnk)
in  tid
end
```

Expanded with (BOM):

```
fun @spawn (f : unit -> unit) : tid =
  let cont fiber () = ( f () ; @stop () )
      val tid : tid = @new_tid ()
      val _ : unit = @enq_with_tid (tid, fiber)
  in  tid
  end
```

Garbage Collection Overview

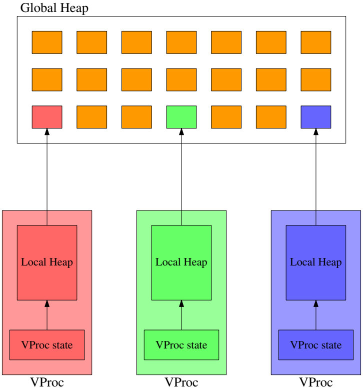
Goal: minimize synchronization and communication b/w VProcs

GC is a combination of the Appel semi-generational collector and the Doligez-Leroy-Gonthier parallel collector

- Minor GCs are completely asynchronous
- Major GCs are mostly asynchronous
- Global GCs are parallel stop-the-world

Heap Architecture

Goal: minimize synchronization and communication b/w VProcs

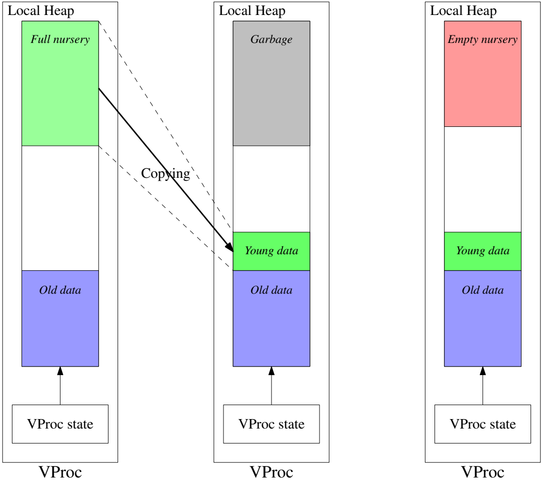


- Invariant: no pointers from global heap to local heaps
- Invariant: no pointers from one local heap to another

Minor Garbage Collections

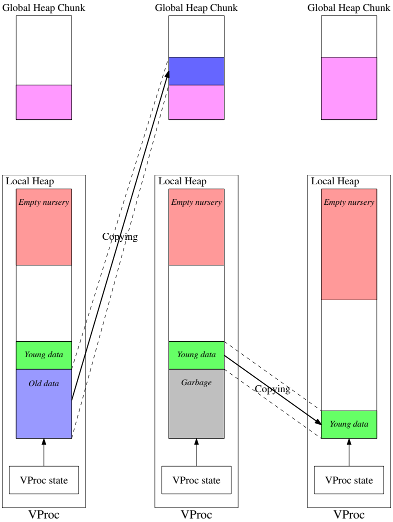
Minor collections use the Appel semi-generational collector.

Allows data to age in the local heap



Major Garbage Collections

Major collections promote older data to the global heap



Global Garbage Collections

Global collector is a simple parallel collector

All VProcs start by doing a major colletion

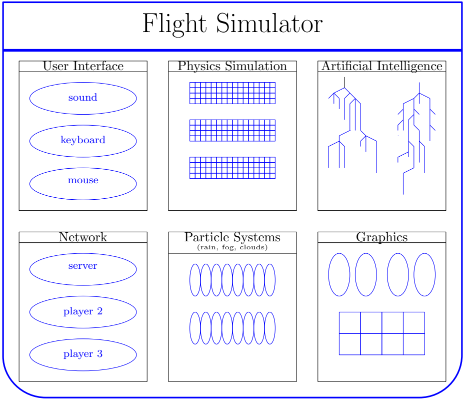
Each VProc does a copy collection in the global heap, using its local heap and registers as roots

Forward pointers are set using atomic CAS instructions

No load balancing

Nested Schedulers

Coordinating heterogeneous parallelism with nested schedulers.



Schedulers

Decide what work to do and when and where to do it.

Many scheduling techniques:

- round-robin thread scheduler
- interactive-threads scheduler, engines, nested engines, workcrews/gangs, work-stealing, lazy-task creation
- cancellation

Infrastructure for Nested Schedulers

Provide an infrastructure

- core mechanisms for building schedulers
- express all of the previous policies

Support for nested schedulers

- multiple scheduling policies in one application
- hierarchies of parallel compuations

Scheduler Actions

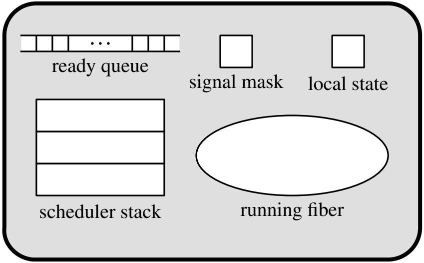
A *scheduler* is represented as a function (called an *action*) that implements scheduling logic.

An action is executed in response to signals:

- `STOP` — the executing fiber has terminated
- `PREEMPT` — the VProc has preempted the executing fiber

Virtual Processors (VProcs)

A VProc has a ready queue, a stack of scheduler actions, a signal mask, local state, and a currently executing fiber.



Infrastructure provides primitive operations to manipulate the state of a VProc.

Scheduler Operations: Fiber-local storage

Dynamically-bound per-fiber storage

- part of vproc local state
- access to scheduler data structures
- thread IDs
- other per-fiber information

```
type fls
val newFls : unit -> fls
val setFls : fls -> unit
val getFls : unit -> fls

type 'a tag
val getFromFls : fls * 'a tag -> 'a option ref
```

Scheduler Operations: Scheduling Queues

One logical scheduling queue per vproc

Two physical scheduling queues per vproc

- local queue
  - no synchronization overhead
  - only accessed by local vproc
- global queue
  - synchronization by mutex lock
  - accessed by local and other vprocs

Fibers in global queue moved to local queue at preemption

```
val enq : fiber -> unit
val deq : unit -> fiber
val enqOnVP : vproc * fiber -> unit
```

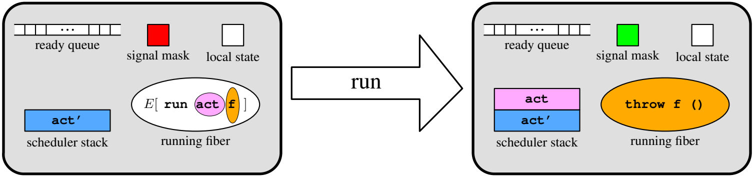
Scheduler Operations: Signal Mask

Operations for explicitly masking and unmasking preemption

```
val mask   : unit -> unit
val unmask : unit -> unit
```

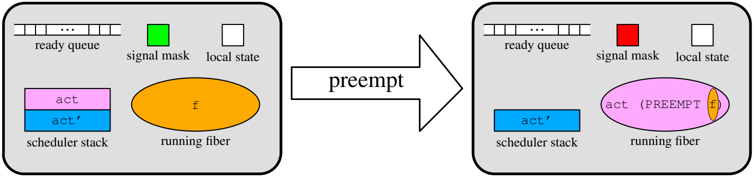
Scheduler Operations: `run`

- `run act f` — pushes the action onto the action stack and starts executing the fiber



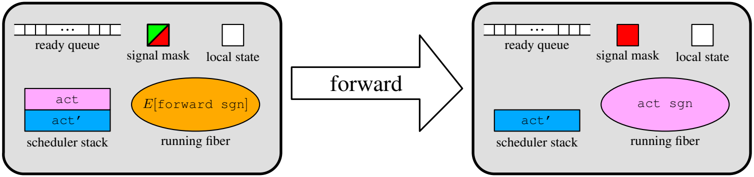
Scheduler Operations: PREEMPT

- *preempt* — captures the executing computation, pops an action from the stack, and delivers a preemption signal.



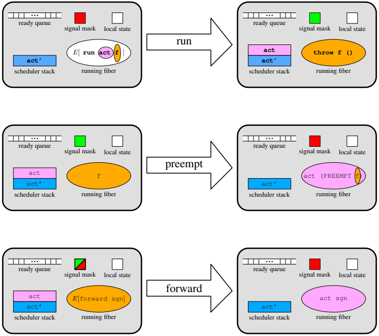
Scheduler Operations: `forward`

- `forward` **`sig`** — pops an action from the stack and delivers the signal.



VProc Operations

A scheduler action performs scheduler specific duties,
and concludes either by forwarding a signal up the stack
or by pushing a new scheduler onto the stack and running a fiber.



Derived VProc Operations

Fiber exit function:

```
fun stop () = forward STOP
```

Fiber yield function:

```
fun preempt k = forward (PREEMPT k)
fun yield () =
  let cont k x = x
  in  preempt k
  end
```

Fiber atomic yield function

```
fun atomicYield () = ( yield () ; mask () )
```

- used to pass preemptions up the action stack

Derived VProc Operations

Fiber migration:

```
fun migrateTo vp =
  let val fls = getFls ()
      cont k x = ( set Fls fls ; x )
  in
      enqOnVP (vp, k) ;
      stop ()
  end
```

- migrated computation takes its fiber-local storage

Default Scheduler

Simple round-robin scheduling policy for fibers in the scheduling queue

```
cont dispatch () = run (roundRobin, deq ())
and roundRobin sgn =
  case sgn of
     STOP => dispatch ()
    | PREEMPT k =>
        let val fls = getFls ()
            cont k' () = ( setFls fls ; throw k () )
        in
            enq k' ;
            dispatch ()
        end
```

Each vproc executes an instance of this scheduler

Future Directions

Address costs with a combination of static analyses and dynamic policies

- Neither static nor dynamic information alone will maximize performance on parallel hardware
- Exclusively static information is necessarily conservative and misses opportunities for parallelism that are apparent dynamically
- Exclusively dynamic information imposes unacceptable overhead to maintain information that may have been available statically

Future Directions

Scheduling (migration) costs

- “hot” cache lines are not migrated along with a thread;
  the migrated thread resumes execution with a “cold” cache.
- migrating a thread to a remote VProc requires promoting the thread (and any object reachable from the thread).

Future Directions

Addressing scheduling (migration) costs

- static promotion analysis — when should data be allocated in the global heap
- static reachability analysis — estimate the amount of (local) data reachable from each program point, encode result into representation of continuations
- dynamic migration policies — use result of static reachability analysis as a cheap, dynamic estimation of the cost of migrating a thread

Future Directions

Scheduling costs

- trade off between locality and load balancing
- sending a message to a remote VProc requires promoting the message (and any object reachable from the message).
- setup and teardown of schedulers for implicitly threaded parallelism

Future Directions

Addressing scheduling costs

- dynamic and static thread characterization — classify threads (or portions of threads) as interactive or computational
- static communication topology analysis — specialize the communication and scheduling of threads
- static scheduler analysis — identify regions of implicitly threaded parallelism that can share setup and teardown

Future Directions

Implicitly threaded parallelism costs

- flattening everywhere (i.e., all types and all expressions) suitable for wide vector hardware, but introduces overheads on non-vector hardware
- preserving the sequential semantics of exceptions and communications introduces compensation code
- granularity of parallel work must exceed the overhead of coordinating the parallel execution

Future Directions

Addressing implicitly threaded parallelism costs

- selective flattening transformation — bias flattening towards
- static effect analysis — when *may* a function raise an exception or perform a communication
- dynamic effect inspection — encode effect in closure representation, dispatch to more efficient parallel code in pure case
- top-down or bottom-up cutoff — switch over to a sequential code-path when exceeding a threshhold (e.g., `trAdd`)

Conclusion

- Implementation of Manticore
  - ???
  - ???

Part V

Conclusion

People and Acknowledgements

The Manticore Project is a joint project between the University of Chicago and the Toyota Technological Institute at Chicago:

- Lars Bergstrom — University of Chicago
- Matthew Fluet — Toyota Technological Institute at Chicago
- Mike Rainey — University of Chicago
- John Reppy — University of Chicago
- Adam Shaw — University of Chicago
- Yingqi Xiao — University of Chicago

and supported (in part) by the

- National Science Foundation