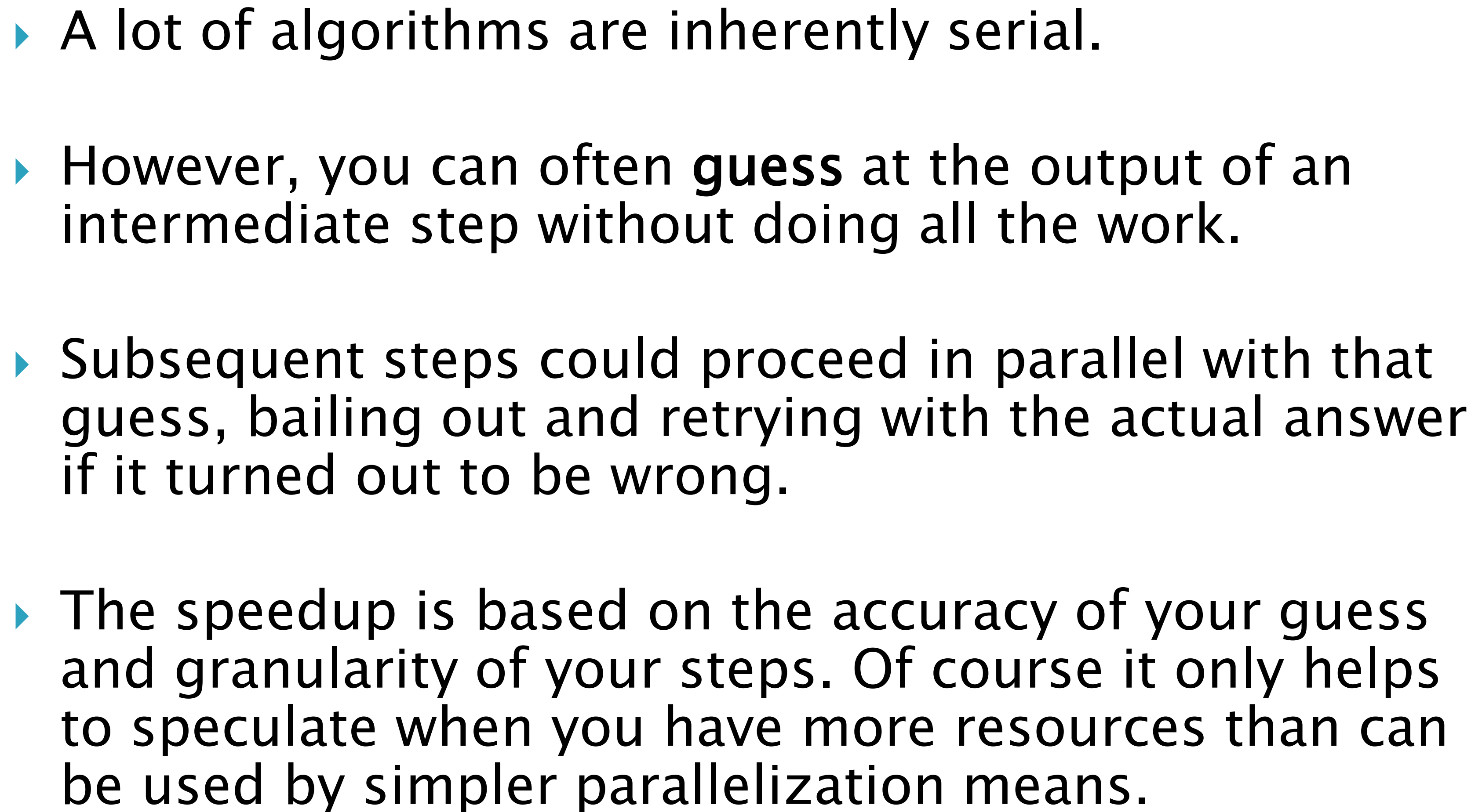
Introducing Speculation

Edward Kmett

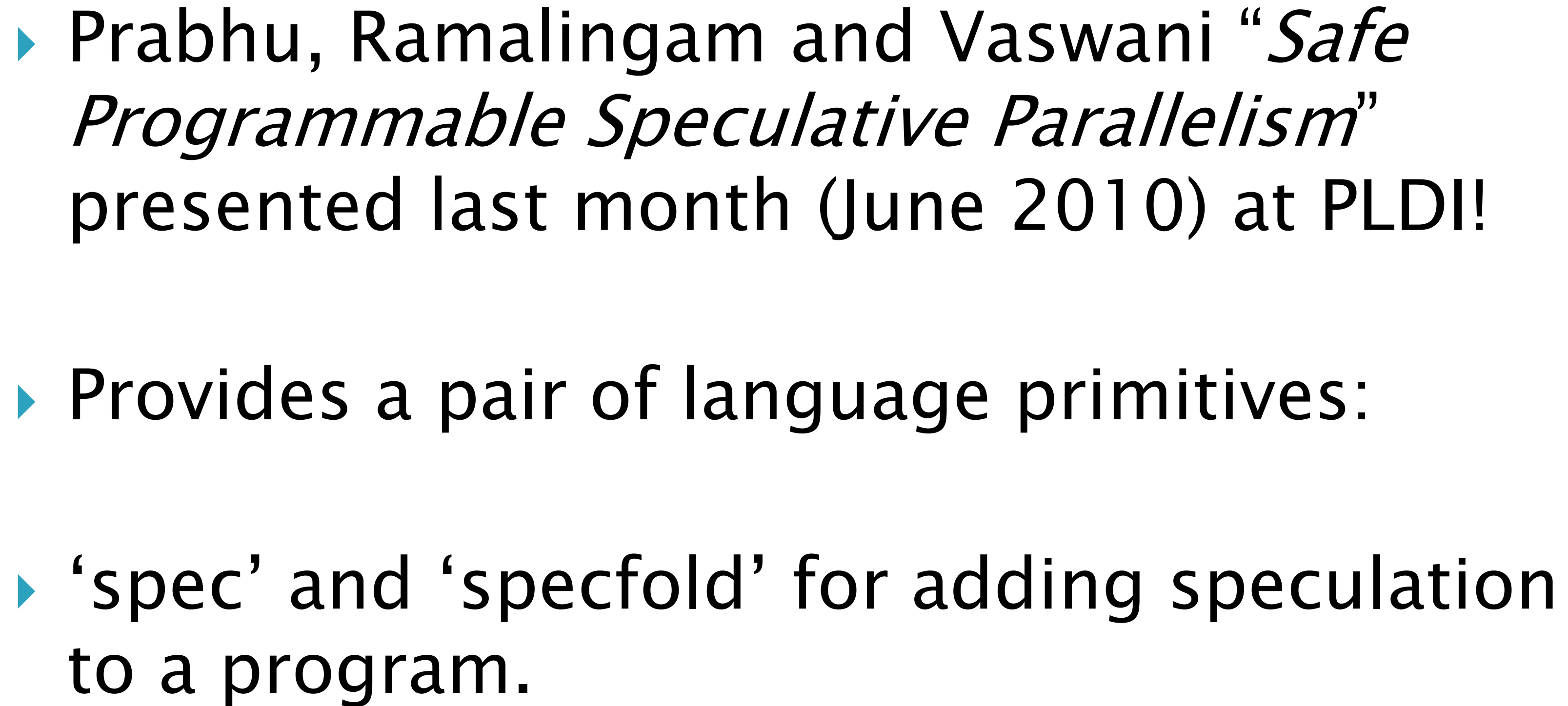
Speculation

- Speculation in C#
  - What is it?
  - Benchmarks!
- Speculation in Haskell
  - Naïve Speculation
  - Abusing GHC
    - Heap Layout
    - Dynamic Pointer Tagging
    - Observing Evaluation
    - Speculation With Observed Evaluation
  - STM Issues
  - Downloading

Adding Parallelism With Guesswork

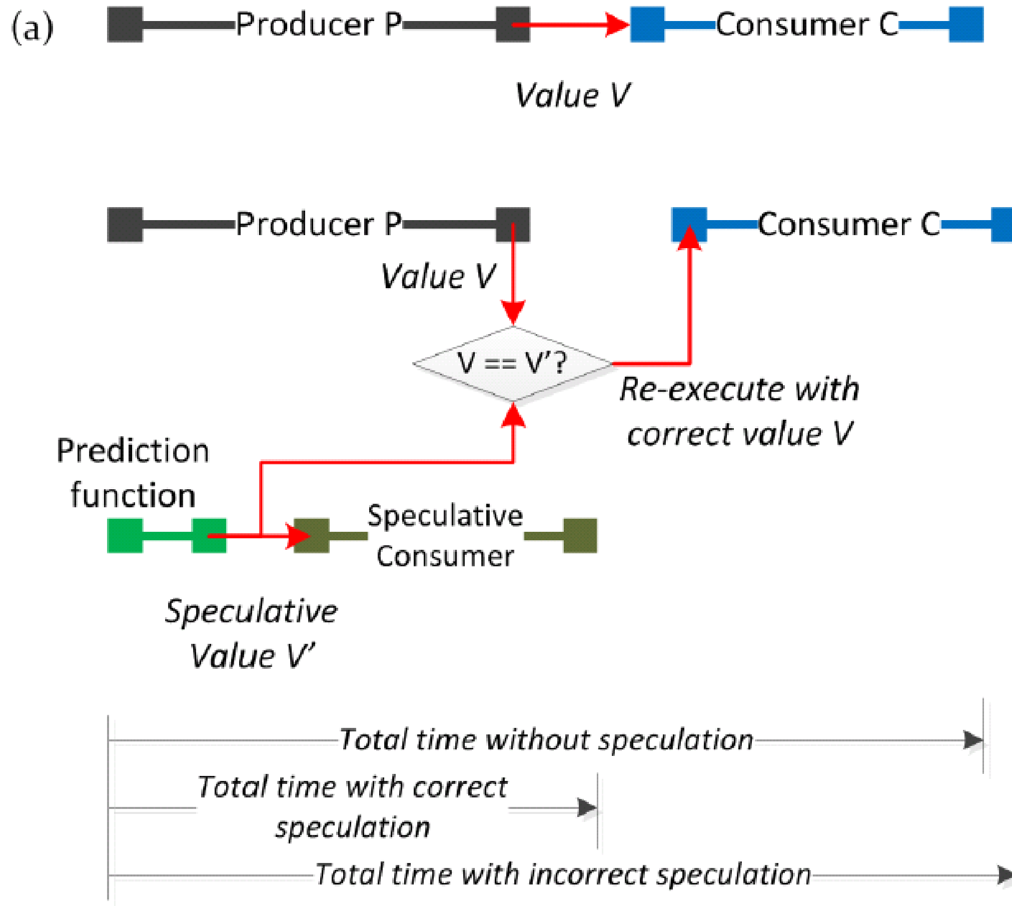


Speculation in C#

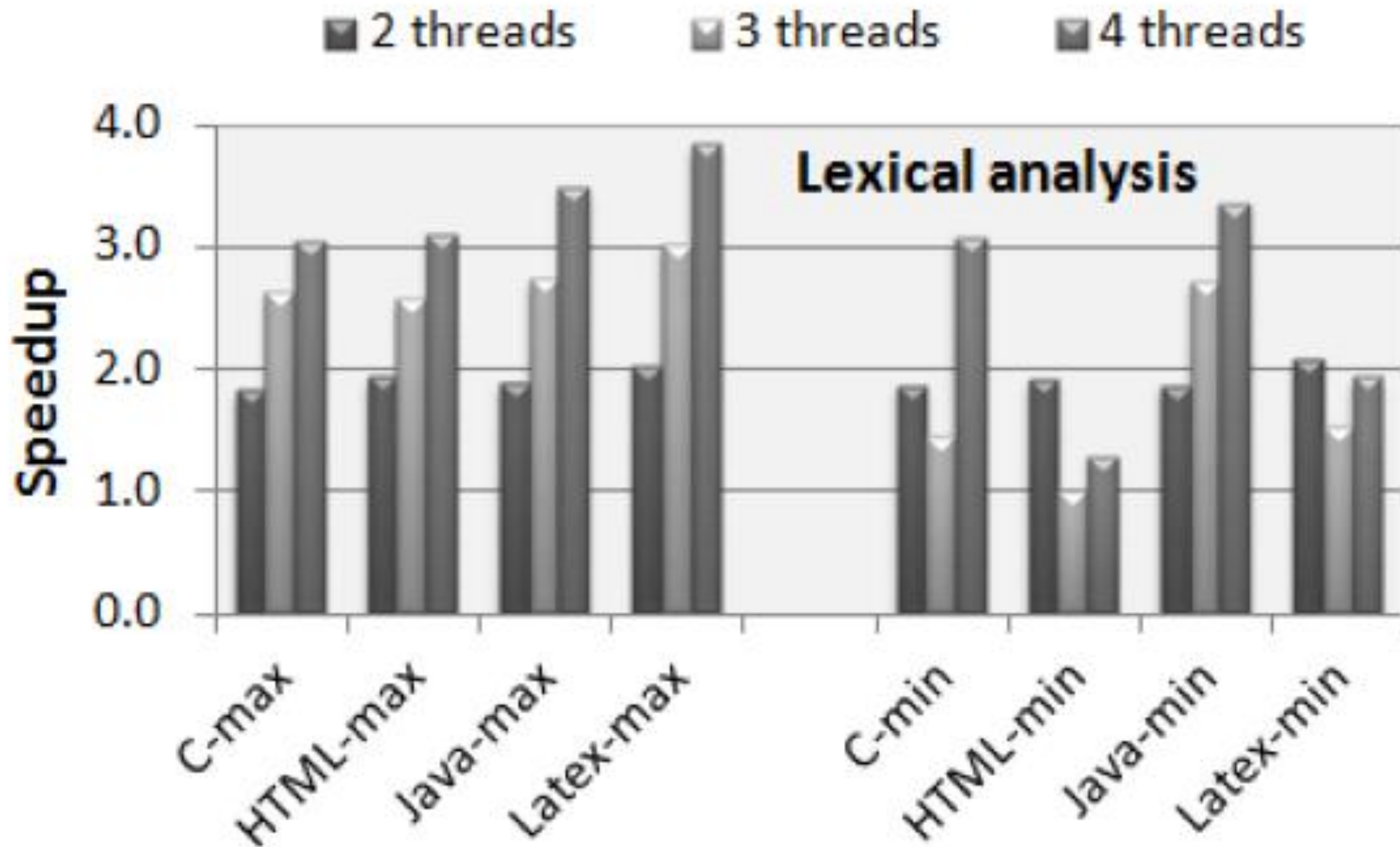


- Prabhu, Ramalingam and Vaswani "*Safe Programmable Speculative Parallelism*" presented last month (June 2010) at PLDI!
- Provides a pair of language primitives:
- 'spec' and 'specfold' for adding speculation to a program.

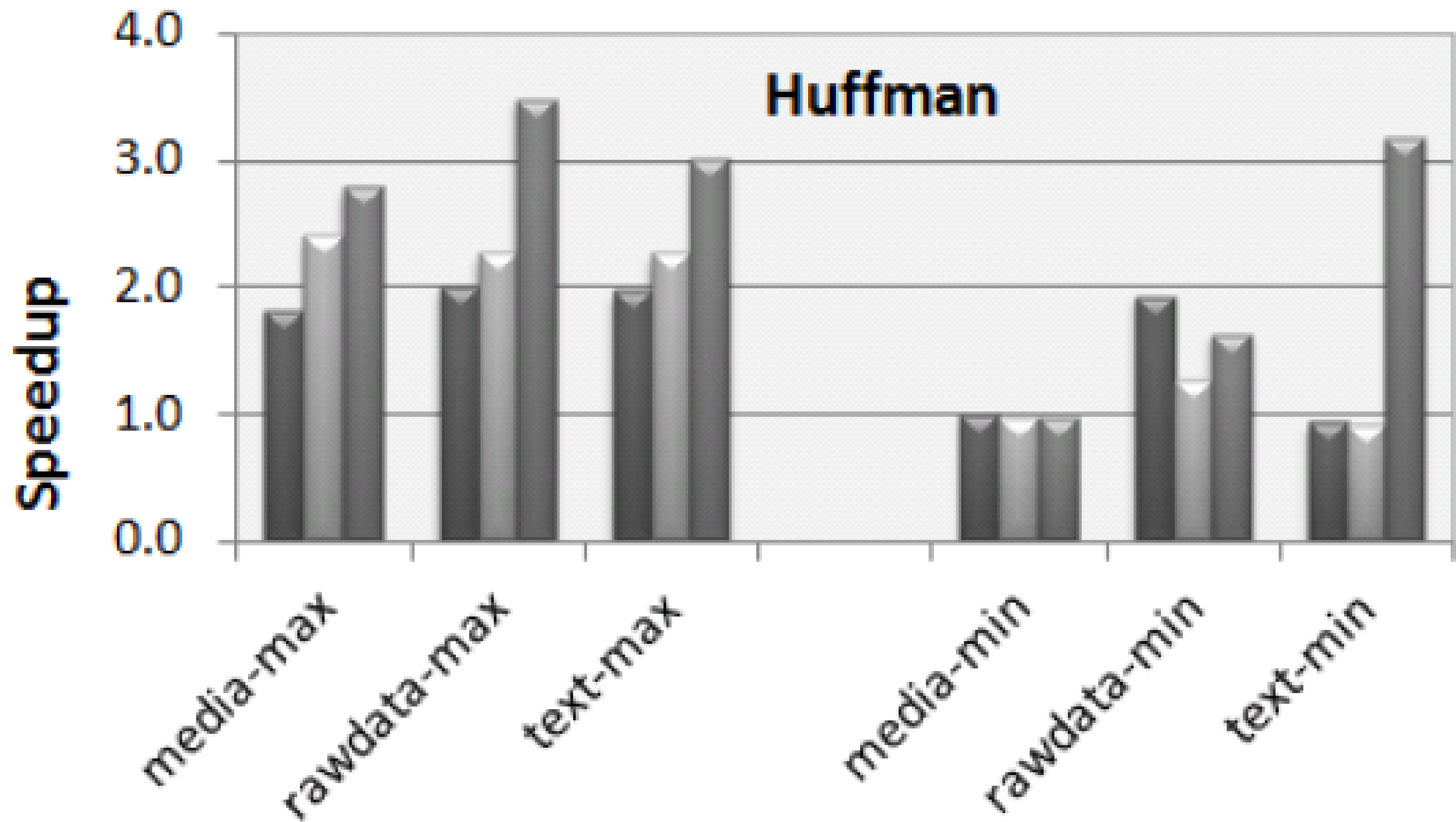
Speculation Timeline



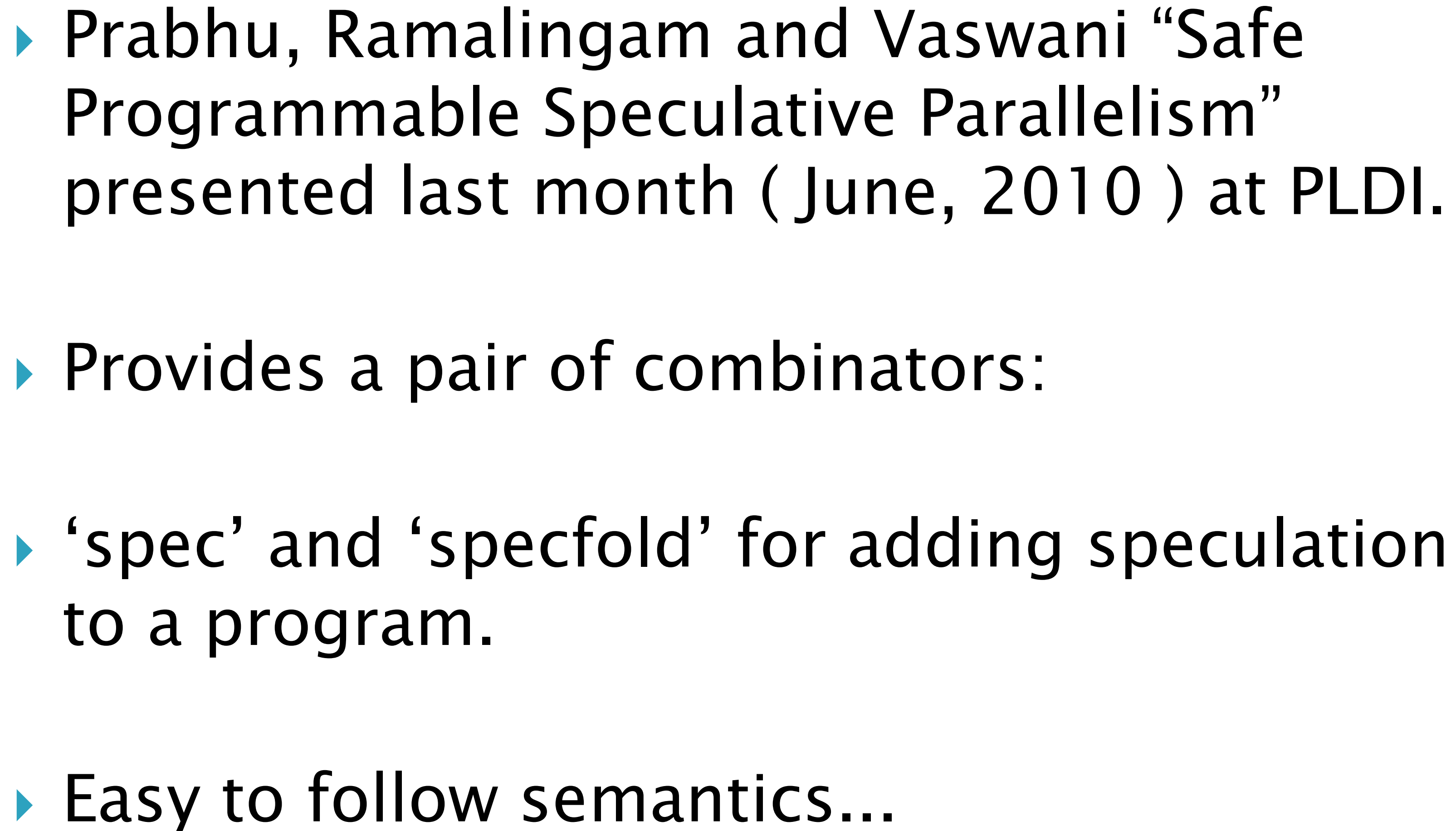
Speculative Lexing



Speculative Huffman Decoding



Speculation in C#



Semantics of Speculation in C# (1of 2)

(a) Syntax and Semantic Domains

$x \in \text{Var}$ $\quad c \in \text{Const}$ $\quad t \in \text{Tid}$ $\quad l \in \text{Loc}$ $\quad v \in \text{Val}$ $\quad e \in \text{Exp}$ $\quad \mathsf{H} \in \text{Heap} = \text{Loc} \hookrightarrow \text{Val}$

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda x.e \mid e_1\ e_2 \mid e_1; e_2 \mid \mathsf{if}\ e_1\ e_2\ e_3 \mid \mathsf{new}\ e \mid e_1 := e_2 \mid\ !e \mid \mathsf{fold}\ e_f\ e_i\ e_l\ e_u \mid \mathsf{spec}\ e_p\ e_g\ e_c \mid \mathsf{specfold}\ e_f\ e_g\ e_l\ e_u \mid r \\
r & ::= & \mathsf{wait}\ t \mid \mathsf{cancel}\ t \mid \mathsf{check}\ t_p\ t_g\ t_c\ e_c \mid \mathsf{auxfold}\ e_f\ e_g\ e_l\ e_u t_p \\
v & ::= & c \mid x \mid \lambda x.e \mid l \mid t \mid \mathsf{unit}
\end{array}
$$

(b) Evaluation Context

$$
E ::= [\cdot] \mid E\ e \mid v\ E \mid E;\ e \mid \mathsf{if}\ E\ e_2\ e_3 \mid \mathsf{new}\ E \mid\ !E \mid E\ :=\ e \mid l\ :=\ E \mid \mathsf{spec}\ e_p\ e_g\ E \mid op_k\ v_1 \cdots v_{i-1}\ E\ e_{i+1} \cdots e_k
$$

$$
(\mathsf{fold}, \mathsf{specfold} \in op_4, \mathsf{check}, \mathsf{auxfold} \in op_5)
$$

(c) Common Evaluation Rules (C)

[THREAD]
$$\frac{\mathsf{H}, e \to \mathsf{H}', e'}{\mathsf{H}, t[e] \parallel \mathsf{T} \Rightarrow \mathsf{H}', t[e'] \parallel \mathsf{T}}$$

[CONTEXT-1]
$$\frac{\mathsf{H}, e \to \mathsf{H}', e'}{\mathsf{H}, E[e] \to \mathsf{H}', E[e']}$$

[CONTEXT-2]
$$\frac{\mathsf{H}, t[e] \parallel \mathsf{T} \Rightarrow \mathsf{H}', t[e'] \parallel \mathsf{T}'}{\mathsf{H}, t[E[e]] \parallel \mathsf{T} \Rightarrow \mathsf{H}', t[E[e']] \parallel \mathsf{T}'}$$

[APPLY]
$$\frac{}{\mathsf{H}, (\lambda x.e)\ v \to \mathsf{H}, e[v/x]}$$

[SEQ]
$$\frac{}{\mathsf{H}, v; e \to \mathsf{H}, e}$$

[IF-ZERO]
$$\frac{}{\mathsf{H}, \mathsf{if}\ 0\ e_2\ e_3 \to \mathsf{H}, e_3}$$

[IF-NON-ZERO]
$$\frac{c \neq 0}{\mathsf{H}, \mathsf{if}\ c\ e_2\ e_3 \to \mathsf{H}, e_2}$$

[ALLOC]
$$\frac{l \notin Dom(\mathsf{H})}{\mathsf{H}, \mathsf{new}\ v \to \mathsf{H}[l \mapsto v], l}$$

[SET]
$$\frac{}{\mathsf{H}, l := v \to \mathsf{H}[l \mapsto v], v}$$

[GET]
$$\frac{}{\mathsf{H}, !l \to \mathsf{H}, \mathsf{H}(l)}$$

[FOLD-1]
$$\frac{v_l > v_u}{\mathsf{H}, \mathsf{fold}\ v_f\ v_{init}\ v_l\ v_u \to \mathsf{H}, v_{init}}$$

[FOLD-2]
$$\frac{v_l \leq v_u}{\mathsf{H}, \mathsf{fold}\ v_f\ v_{init}\ v_l\ v_u \to \mathsf{H}, \mathsf{fold}\ v_f\ (v_f\ v_l\ v_{init})\ (v_l + 1)\ v_u}$$

Semantics of Speculation in C# (2 of 2)

(d) Speculative Evaluation Rules (S)

[WAIT]

$$\frac{}{\mathsf{H}, t'[\mathsf{wait}\ t] \parallel t[v] \parallel \mathsf{T} \Rightarrow \mathsf{H}, t'[v] \parallel t[v] \parallel \mathsf{T}}$$

[CANCEL]

$$\frac{}{\mathsf{H}, t'[\mathsf{cancel}\ t] \parallel t[e] \parallel \mathsf{T} \Rightarrow \mathsf{H}, t'[()] \parallel \mathsf{T}}$$

[SPEC-APPLY]

$$\frac{t_p, t_g, t_c \text{ fresh in } \mathsf{T}}{\mathsf{H}, t[\mathsf{spec}\ e_p\ e_g\ v_c] \parallel \mathsf{T} \Rightarrow \mathsf{H}, t_p[e_p] \parallel t_g[e_g] \parallel t_c[v_c\ (\mathsf{wait}\ t_g)] \parallel t[\mathsf{check}\ t_p\ t_g\ t_c\ v_c] \parallel \mathsf{T}}$$

[CHECK]

$$\frac{x_p, x_g \text{ not free in } v_c}{\mathsf{H}, \mathsf{check}\ t_p\ t_g\ t_c\ v_c \Rightarrow \mathsf{H}, (\lambda x_p, x_g.\mathsf{if}(x_p =_{int} x_g)\ (\mathsf{wait}\ t_c)\ (\mathsf{cancel}\ t_c; v_c\ x_p))(\mathsf{wait}\ t_p)(\mathsf{wait}\ t_g)}$$

[SPEC-ITERATE-1]

$$\frac{v_l \le v_u \text{ and } t_g, t_b \text{ fresh in } \mathsf{T}}{\mathsf{H}, t[\mathsf{specfold}\ v_f\ v_g\ v_l\ v_u] \parallel \mathsf{T} \Rightarrow \mathsf{H}, t[\mathsf{auxfold}\ v_f\ v_g\ (v_l + 1)\ v_u t_b] \parallel t_g[v_g\ v_l] \parallel t_b[v_f\ (\mathsf{wait}\ t_g) v_l] \parallel \mathsf{T}}$$

[SPEC-ITERATE-2]

$$\frac{v_l \le v_u \text{ and } t_g, t_b, t_c \text{ fresh in } \mathsf{T}}{\mathsf{H}, t[\mathsf{auxfold}\ v_f\ v_g\ v_l\ v_u t_p] \parallel \mathsf{T} \Rightarrow \mathsf{H}, t[\mathsf{auxfold}\ v_f\ v_g\ (v_l + 1)\ v_u t_c] \parallel t_g[v_g\ v_l] \parallel t_b[v_f\ v_l\ (\mathsf{wait}\ t_g)] \parallel t_c[\mathsf{check}\ t_p\ t_g\ t_b\ (v_f v_l)] \parallel \mathsf{T}}$$

[SPEC-ITERATE-3]

$$\frac{v_l > v_u}{\mathsf{H}, t[\mathsf{auxfold}\ v_f\ v_g\ v_l\ v_u t_p] \parallel \mathsf{T} \Rightarrow \mathsf{H}, t[\mathsf{wait}\ t_p] \parallel \mathsf{T}}$$

(e) Non-Speculative Evaluation Rules For Speculative Constructs (N)

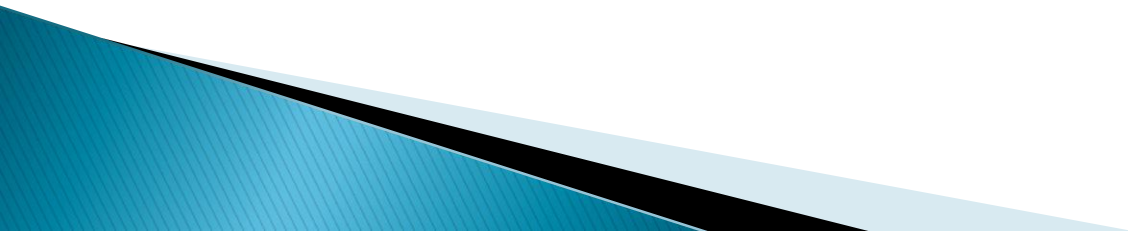
[NONSPEC-APPLY]

$$\frac{}{\mathsf{H}, \mathsf{spec}\ e_p\ e_g\ e_c \rightarrow \mathsf{H}, e_c\ e_p}$$

[NONSPEC-ITERATE]

$$\frac{v_l \le v_u}{\mathsf{H}, \mathsf{specfold}\ v_f\ v_g\ v_l\ v_u \rightarrow \mathsf{fold}\ v_f\ (v_g\ v_l)\ v_l\ v_u}$$

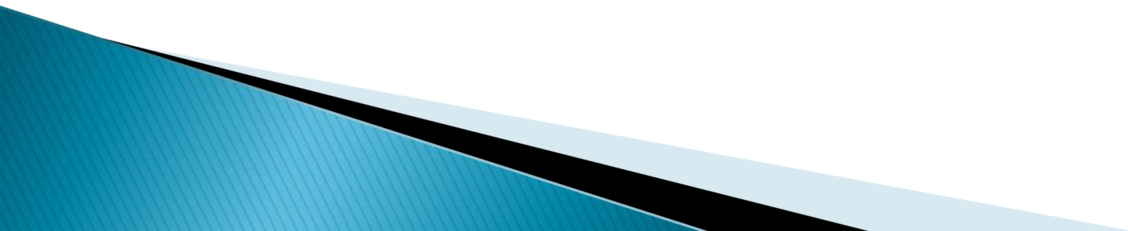
Any Questions?



Speculation in Haskell

Within 5 minutes of the paper reaching reddit, I replied with an implementation in Haskell.

Sadly, it has yet to accumulate any upvotes.



Speculation in Haskell

```
spec :: Eq a => a -> (a -> b) -> a -> b
spec guess f a =
   let speculation = f guess in
   speculation `par`
        if guess == a
        then speculation
        else f a
```

Speculation in Haskell

```
spec :: Eq a => a -> (a -> b) -> a -> b
spec guess f a =
  let speculation = f guess in
  speculation `par`
    if guess == a
    then speculation
    else f a
```

Naïve Speculation

Under load the spark doesn't even happen. Therefore we don't kill ourselves trying to speculate with resources we don't have! This is an improvement over the C# implementation, which can start to diverge under speculation.

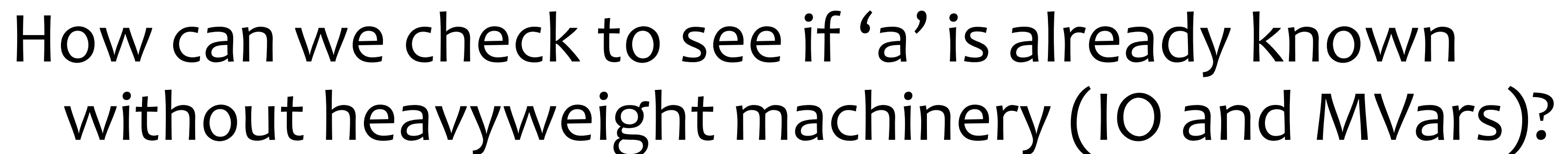
If we speculated wrongly, the garbage collector (in HEAD) is smart enough to collect the entire spark!

I want more!

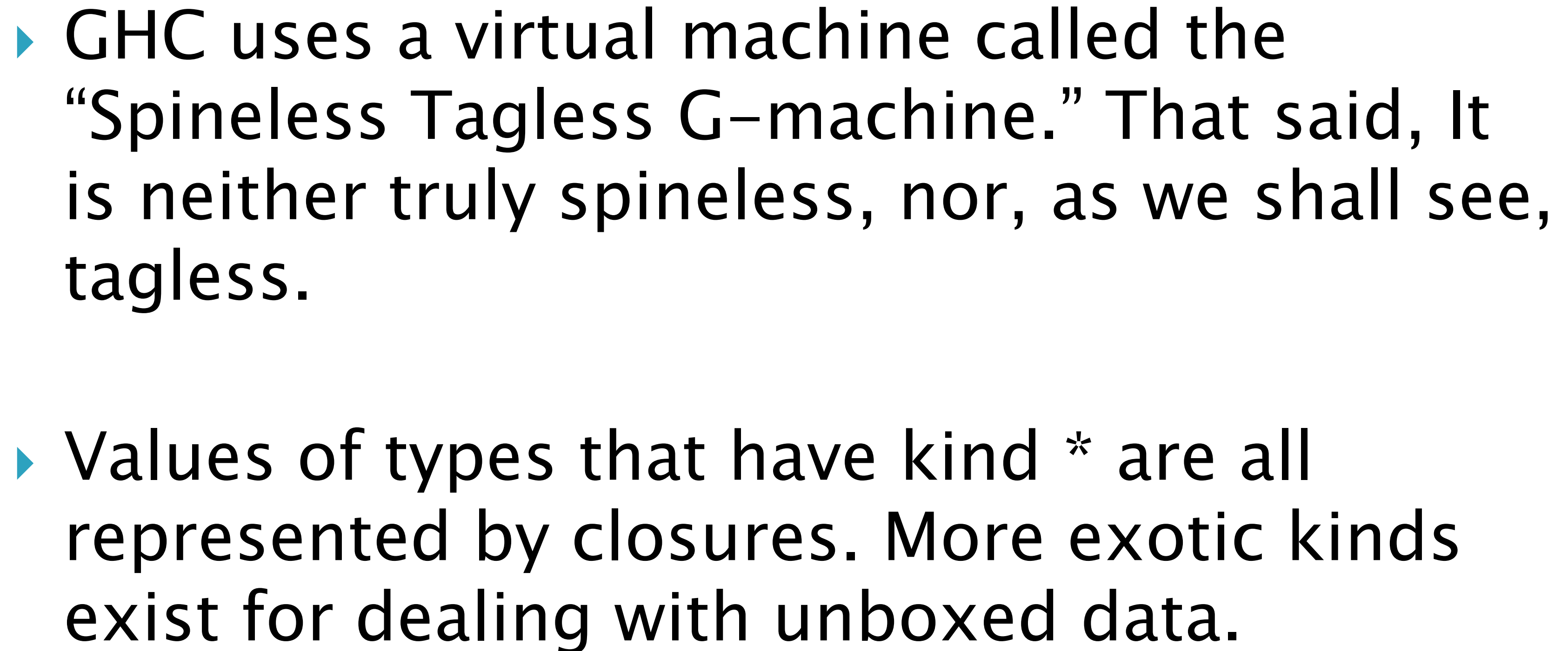
What if we already know ‘a’ by the time we go to evaluate the spec? (it may have been sparked and completed by now)

Then by construction any time spent computing a guess is wasted.

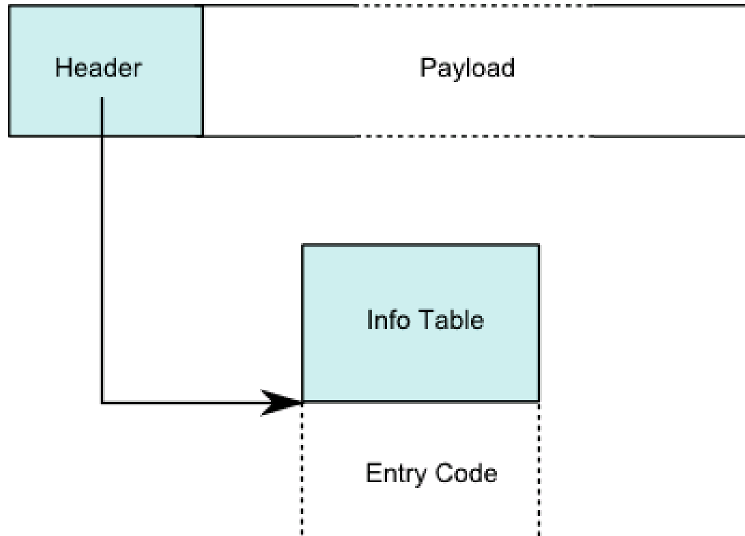
How can we check to see if ‘a’ is already known without heavyweight machinery (IO and MVars)?



Heap Layout



Heap Layout



- The entry code for a (saturated) data constructor just returns itself.

- Indirections entry code just returns the value of the target of the indirection.

- Thunk entry code evaluates the thunk, and then rewrites its header into an indirection!

- Garbage collection removes indirections!

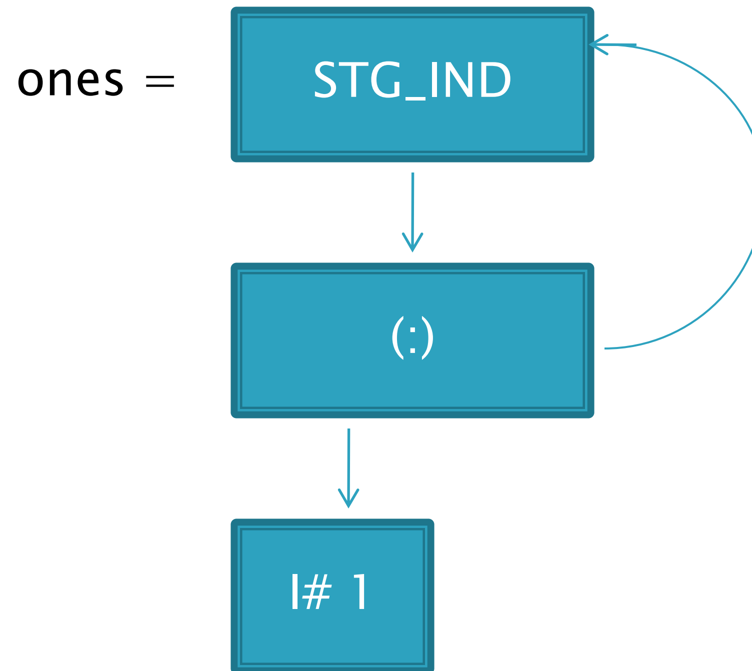
Evaluation

- ones :: [Int]
- ones = 1 : ones

ones = thunk_1234

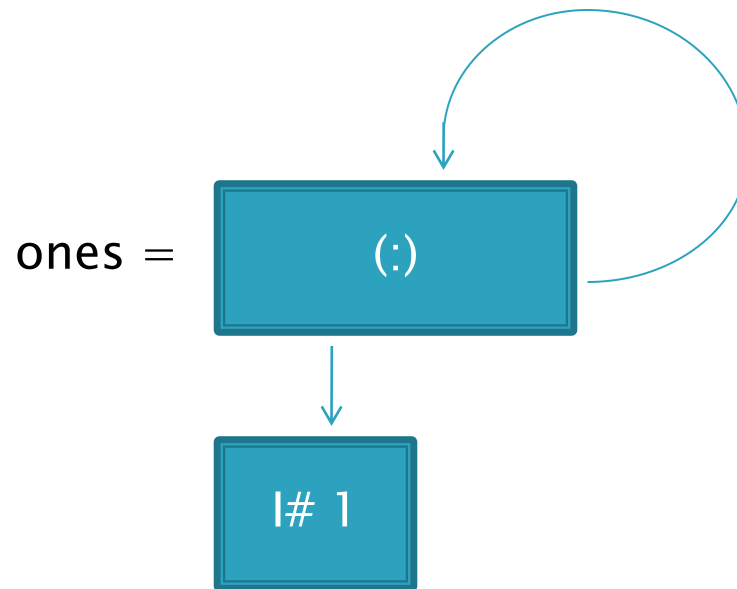
Evaluation

- ones :: [Int]
- ones = 1 : ones



Evaluation

- ones :: [Int]
- ones = 1 : ones



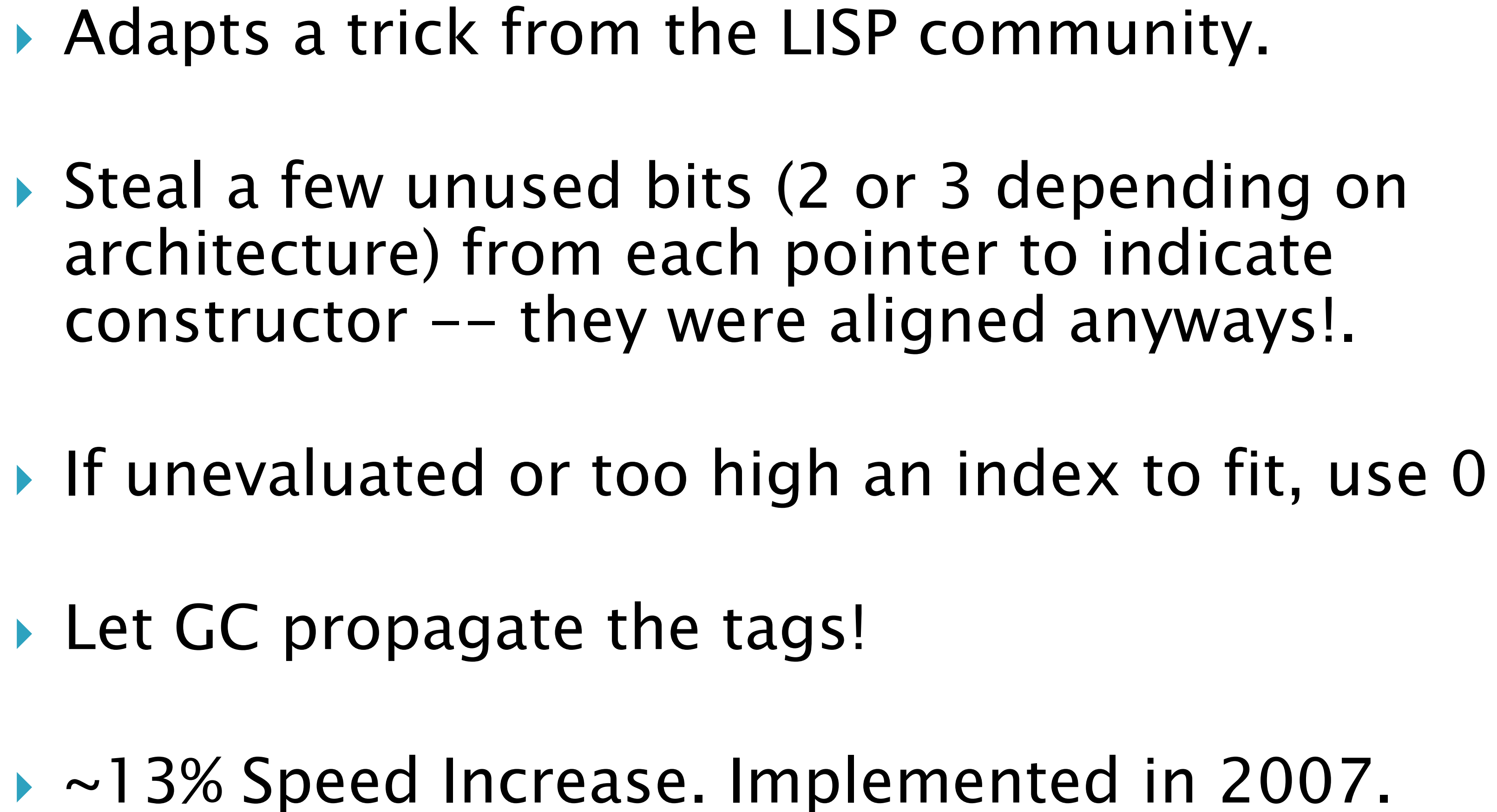
Dynamic Pointer Tagging

- Jumping into unknown code to “evaluate” already evaluated data is inefficient.

- More than half the time, the target is already evaluated.

Program	Evaluated scrutinee (%)
anna	65.1
cacheprof	72.8
constraints	54.5
fulsom	41.5
integrate	67.6
mandel	73.9
simple	78.6
sphere	72.8
typecheck	56.5
wang	41.9
(81 more)	...
Min	0.20
Max	99.00
Average	61.86

Dynamic Pointer Tagging



Dynamic Pointer Tagging

Family Size	Distribution (%)	Cumulative (%)
1	42.5	42.5
2	52.4	94.9
3	1.2	96.1
4	0.5	96.6
5	1.7	98.3
6	0.9	99.2
7	0.0	99.2
> 7	0.8	100.0

- Handles 99.2% (96.1%) of constructors in practice

Abusing Dynamic Pointer Tags

- Can we get at the tag from Haskell?

```
data Box a = Box a

unsafeGetTagBits :: a -> Int
unsafeGetTagBits a =
  unsafeCoerce (Box a) .&.
  (sizeOf (undefined :: Int) – 1)
```

Relies on the fact that we can treat a Box as an Int due to tagging! In practice we can use the unsafeCoerce# primop to directly coerce to an unboxed Word#, and avoid the extra box.

Abusing Dynamic Pointer Tags

- This function is unsafe! It may return either 0 or the final answer depending on if the thunk it is looking at has been evaluated and if GC has run since then, but it'll never lie about the tag if not 0.
- You have an extra obligation: Your code should give the same answer regardless of whether or not unsafeGetTagBits returns 0!
- But that is exactly what 'spec' does!

Smarter Speculation

```
spec :: Eq a => a -> (a -> b) -> a -> b
spec guess f a
  | unsafeGetTagBits a /= 0 = f a
  | otherwise =
  let speculation = f guess in
  speculation `par`
  if g == a
  then speculation
  else f a
```

Is that it?

- The complicated semantics for the C# implementation come from checking that the speculated producer (guess) and consumer could read and write to references, without seeing side-effects from badly speculated code.
- We don't have any side-effects in pure code, so we can skip all of those headaches in the common case, but how can we model something where these transactional mutations occur?

Speculating STM

```
specSTM :: Eq a => STM a –> (a -> STM b) -> a -> STM b
specSTM mguess f a = a `par` do
    guess <- mguess
    result <- f guess
    unless (guess == a) retry
    return result
  `orElse`
    f a
```

Problems with Speculating STM

```
specSTM :: Eq a => STM a –> (a -> STM b) -> a -> STM b
specSTM mguess f a = a `par` do ...
```

Before we could spark the evaluation of *f guess*, so that if it was forgotten under load, we reverted more or less to the original serial behavior.

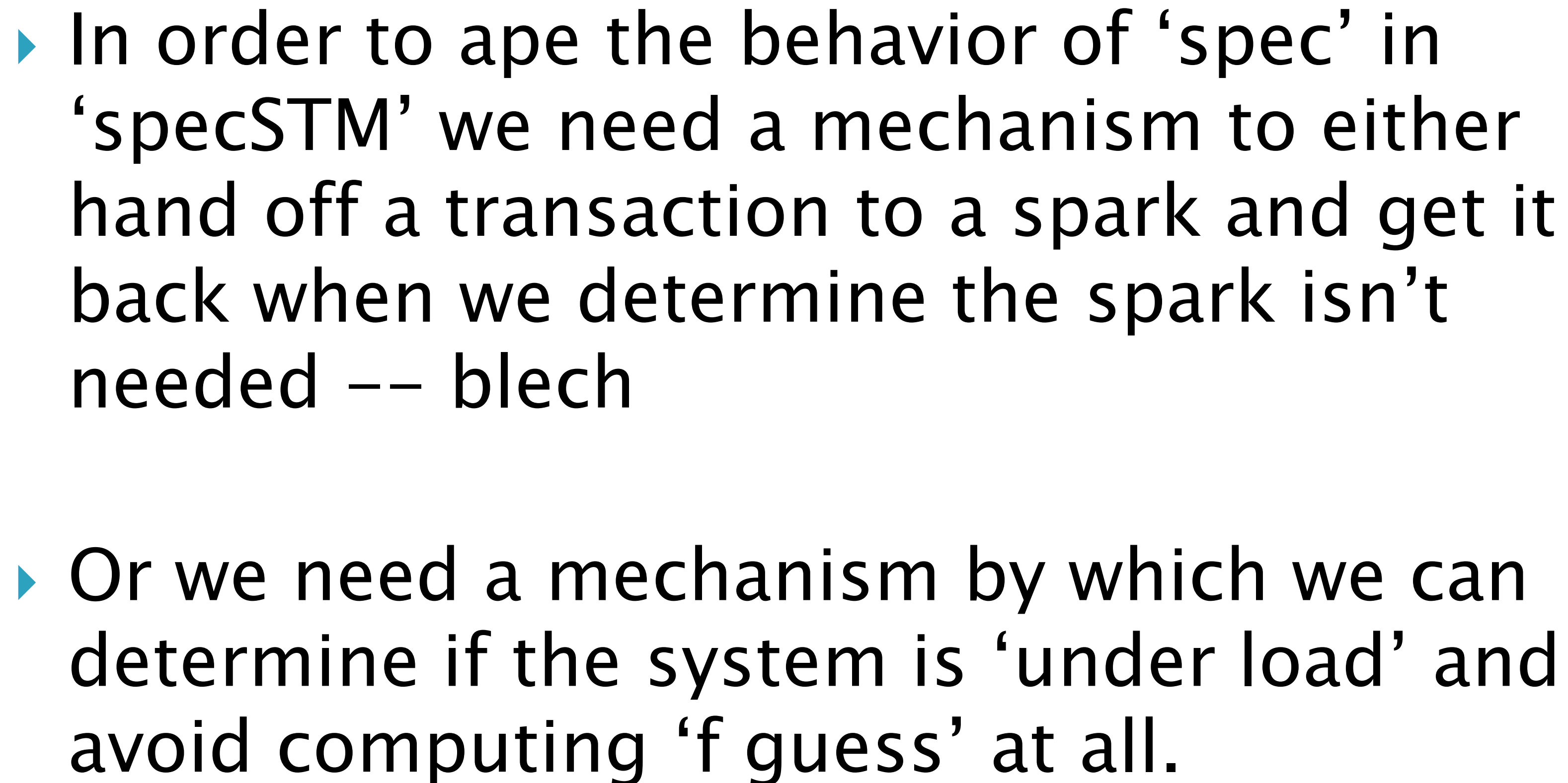
Here we are forced to evaluate the *argument* in the background! The problem with this shows up under load.

Problems with Speculating STM

Under load, the spark queue will fill up and 'spec' will skip the evaluation of the spark, in its case, 'f guess', before returning either 'f a' or 'f guess' based on comparing 'guess' with 'a'. So the only wasted computation is checking 'guess == a'

However, specSTM can merely skip the evaluation of 'a', because evaluating 'f guess' needs the current transaction, which is bound deep in the bowels of GHC to the current thread and capability, etc. Therefore, it can only skip the only thing we know it will actually need, since it ultimately must check if 'guess == a', which will need the value of 'a' that we sparked.

Paths to Resolution



How Loaded is Loaded?

- Ultimately the definition of under load is somewhat tricky. You can't just look at the load of the machine. It is the depth of the spark queue that determines if you're loaded!

- All we need to do is count the number of entries in the spark queue for the current capability. In "C--":

```
dequeElements(cap->spark)
```

Adding numSparks#

- What we need is a new "primop":

numSparks# :: State# s -> (# State# s, Int# #)

- GHC has even added the ability to let third-party libraries define their own primops so that they could factor out the use of GMP from base and into its own library!

- Sadly, the details of 'cap' and 'spark' are buried in GHC's "private" headers and so we can't exploit this mechanism. The extension has to be done in GHC itself. (feature request #4167)

Speculative Folds

foldr :: (Foldable f, Eq b) => (Int -> b) -> (a -> b -> b) -> b -> f a -> b

Takes an extra argument that computes the guess at the answer after *n* items, the **last** *n* items.

This way the estimator is counting the number of items being estimated. Otherwise foldr over the tail of a list would be receiving entirely different numbers.

Speculative Folds

```
foldr :: (Foldable f, Eq b) => (Int -> b) -> (a -> b -> b) -> b -> f a -> b
foldr guess f z = snd . Foldable.foldr f’ (0, z)
  where
    f’ a (!n, b) = (n + 1, spec (guess n) (f a) b)
```

Speculation on Hackage

'speculation' on hackage is currently at version 0.9.0.0

It provides:

- Control.Concurrent.Speculation
  - spec, specSTM, unsafeGetTagBits and generalizations

And a number of modules full of speculative folds:

- Data.List.Speculation (scanl, etc.)
- Data.Foldable.Speculation (foldl, foldr, etc.)
- Data.Traversable.Speculation (traverse, etc.)
- Control.Morphism.Speculation (hylo!)

Lots of Combinators! Data.Foldable.Speculation

```
fold :: (Foldable f, Monoid m, Eq m) => (Int -> m) -> f m -> m
foldBy :: (Foldable f, Monoid m) => (m -> m -> Bool) -> (Int -> m) -> f m -> m
foldMap :: (Foldable f, Monoid m, Eq m) => (Int -> m) -> (a -> m) -> f a -> m
foldMapBy :: (Foldable f, Monoid m) => (m -> m -> Bool) -> (Int -> m) -> (a -> m) -> f a -> m
foldr :: (Foldable f, Eq b) => (Int -> b) -> (a -> b -> b) -> b -> f a -> b
foldrBy :: Foldable f => (b -> b -> Bool) -> (Int -> b) -> (a -> b -> b) -> b -> f a -> b
foldl :: (Foldable f, Eq b) => (Int -> b) -> (b -> a -> b) -> b -> f a -> b
foldlBy :: Foldable f => (b -> b -> Bool) -> (Int -> b) -> (b -> a -> b) -> b -> f a -> b
foldr1 :: (Foldable f, Eq a) => (Int -> a) -> (a -> a -> a) -> f a -> a
foldr1By :: Foldable f => (a -> a -> Bool) -> (Int -> a) -> (a -> a -> a) -> f a -> a
foldl1 :: (Foldable f, Eq a) => (Int -> a) -> (a -> a -> a) -> f a -> a
foldl1By :: Foldable f => (a -> a -> Bool) -> (Int -> a) -> (a -> a -> a) -> f a -> a
foldrM :: (Foldable f, Monad m, Eq (m b)) => (Int -> m b) -> (a -> b -> m b) -> m b -> f a -> m b
foldrByM :: (Foldable f, Monad m) => (m b -> m b -> Bool) -> (Int -> m b) -> (a -> b -> m b) -> m b -> f a -> m b
foldlM :: (Foldable f, Monad m, Eq (m b)) => (Int -> m b) -> (b -> a -> m b) -> m b -> f a -> m b
foldlByM :: (Foldable f, Monad m) => (m b -> m b -> Bool) -> (Int -> m b) -> (b -> a -> m b) -> m b -> f a -> m b
foldrSTM :: (Foldable f, Eq b) => (Int -> STM b) -> (a -> b -> STM b) -> STM b -> f a -> STM b
foldrBySTM :: Foldable f => (b -> b -> STM Bool) -> (Int -> STM b) -> (a -> b -> STM b) -> STM b -> f a -> STM b
foldlSTM :: (Foldable f, Eq a) => (Int -> STM a) -> (a -> b -> STM a) -> STM a -> f b -> STM a
foldlBySTM :: Foldable f => (a -> a -> STM Bool) -> (Int -> STM a) -> (a -> b -> STM a) -> STM a -> f b -> STM a
```

Lots of Combinators! Data.Foldable.Speculation

```
traverse_ :: (Foldable t, Applicative f, Eq (f ())) => (Int -> f c) -> (a -> f b) -> t a -> f ()
traverseBy_ :: (Foldable t, Applicative f) => (f () -> f () -> Bool) -> (Int -> f c) -> (a -> f b) -> t a -> f ()
for_ :: (Foldable t, Applicative f, Eq (f ())) => (Int -> f c) -> t a -> (a -> f b) -> f ()
forBy_ :: (Foldable t, Applicative f) => (f () -> f () -> Bool) -> (Int -> f c) -> t a -> (a -> f b) -> f ()
sequenceA_ :: (Foldable t, Applicative f, Eq (f ())) => (Int -> f b) -> t (f a) -> f ()
sequenceByA_ :: (Foldable t, Applicative f, Eq (f ())) => (f () -> f () -> Bool) -> (Int -> f b) -> t (f a) -> f ()
asum :: (Foldable t, Alternative f, Eq (f a)) => (Int -> f a) -> t (f a) -> f a
asumBy :: (Foldable t, Alternative f) => (f a -> f a -> Bool) -> (Int -> f a) -> t (f a) -> f a
mapM_ :: (Foldable t, Monad m, Eq (m ())) => (Int -> m c) -> (a -> m b) -> t a -> m ()
mapByM_ :: (Foldable t, Monad m) => (m () -> m () -> Bool) -> (Int -> m c) -> (a -> m b) -> t a -> m ()
forM_ :: (Foldable t, Monad m, Eq (m ())) => (Int -> m c) -> t a -> (a -> m b) -> m ()
forByM_ :: (Foldable t, Monad m) => (m () -> m () -> Bool) -> (Int -> m c) -> t a -> (a -> m b) -> m ()
sequence_ :: (Foldable t, Monad m, Eq (m ())) => (Int -> m b) -> t (m a) -> m ()
sequenceBy_ :: (Foldable t, Monad m) => (m () -> m () -> Bool) -> (Int -> m b) -> t (m a) -> m ()
msum :: (Foldable t, MonadPlus m, Eq (m a)) => (Int -> m a) -> t (m a) -> m a
msumBy :: (Foldable t, MonadPlus m) => (m a -> m a -> Bool) -> (Int -> m a) -> t (m a) -> m a
mapSTM_ :: Foldable t => STM Bool -> (Int -> STM c) -> (a -> STM b) -> t a -> STM ()
forSTM_ :: Foldable t => STM Bool -> (Int -> STM c) -> t a -> (a -> STM b) -> STM ()
sequenceSTM_ :: Foldable t => STM Bool -> (Int -> STM a) -> t (STM b) -> STM ()
```

...

Lots of Combinators! Data.Traversable.Speculation

```
traverse :: (Traversable t, Applicative f, Eq a) => (Int -> a) -> (a -> f b) -> t a -> f (t b)
traverseBy :: (Traversable t, Applicative f) => (a -> a -> Bool) -> (Int -> a) -> (a -> f b) -> t a -> f (t b)
for :: (Traversable t, Applicative f, Eq a) => (Int -> a) -> t a -> (a -> f b) -> f (t b)
forBy :: (Traversable t, Applicative f) => (a -> a -> Bool) -> (Int -> a) -> t a -> (a -> f b) -> f (t b)
sequenceA :: (Traversable t, Applicative f, Eq (f a)) => (Int -> f a) -> t (f a) -> f (t a)
sequenceByA :: (Traversable t, Applicative f) => (f a -> f a -> Bool) -> (Int -> f a) -> t (f a) -> f (t a)
mapM :: (Traversable t, Monad m, Eq a) => (Int -> a) -> (a -> m b) -> t a -> m (t b)
mapByM :: (Traversable t, Monad m) => (a -> a -> Bool) -> (Int -> a) -> (a -> m b) -> t a -> m (t b)
sequence :: (Traversable t, Monad m, Eq (m a)) => (Int -> m a) -> t (m a) -> m (t a)
sequenceBy :: (Traversable t, Monad m) => (m a -> m a -> Bool) -> (Int -> m a) -> t (m a) -> m (t a)
forM :: (Traversable t, Monad m, Eq a) => (Int -> a) -> t a -> (a -> m b) -> m (t b)
forByM :: (Traversable t, Monad m) => (a -> a -> Bool) -> (Int -> a) -> t a -> (a -> m b) -> m (t b)
mapSTM :: (Traversable t, Eq a) => (Int -> STM a) -> (a -> STM b) -> t a -> STM (t b)
mapBySTM :: Traversable t => (a -> a -> STM Bool) -> (Int -> STM a) -> (a -> STM b) -> t a -> STM (t b)
forSTM :: (Traversable t, Eq a) => (Int -> STM a) -> t a -> (a -> STM b) -> STM (t b)
forBySTM :: Traversable t => (a -> a -> STM Bool) -> (Int -> STM a) -> t a -> (a -> STM b) -> STM (t b)
mapAccumL :: (Traversable t, Eq a) => (Int -> a) -> (a -> b -> (a, c)) -> a -> t b -> (a, t c)
mapAccumLBy :: Traversable t => (a -> a -> Bool) -> (Int -> a) -> (a -> b -> (a, c)) -> a -> t b -> (a, t c)
mapAccumR :: (Traversable t, Eq a) => (Int -> a) -> (a -> b -> (a, c)) -> a -> t b -> (a, t c)
mapAccumRBy :: Traversable t => (a -> a -> Bool) -> (Int -> a) -> (a -> b -> (a, c)) -> a -> t b -> (a, t c)
```

Lots of Combinators! Data.List.Speculation

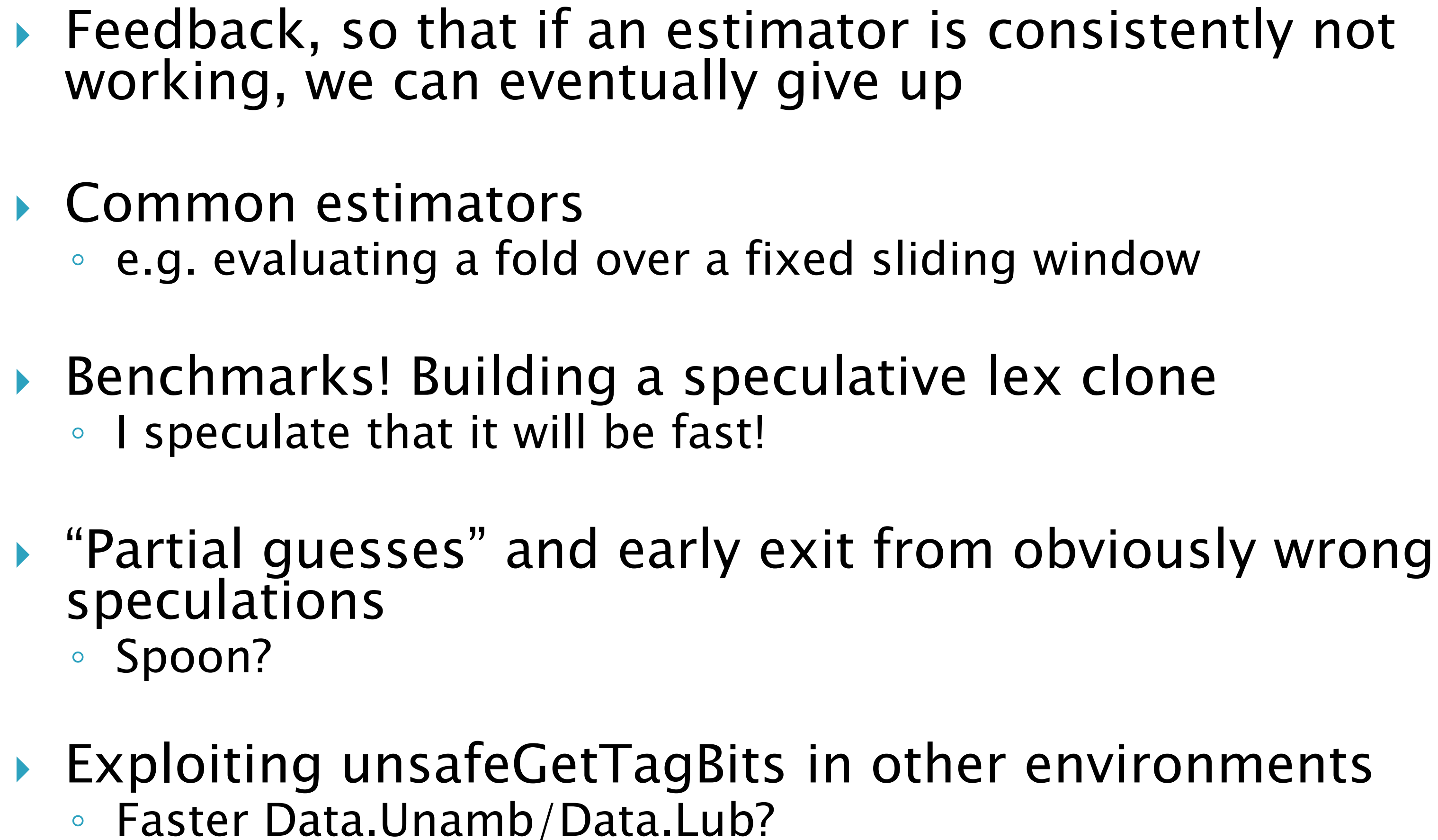
```
scan :: (Monoid m, Eq m) => (Int -> m) -> [m] -> [m]
scanBy :: Monoid m => (m -> m -> Bool) -> (Int -> m) -> [m] -> [m]
scanMap :: (Monoid m, Eq m) => (Int -> m) -> (a -> m) -> [a] -> [m]
scanMapBy :: Monoid m => (m -> m -> Bool) -> (Int -> m) -> (a -> m) -> [a] -> [m]
scanr :: Eq b => (Int -> b) -> (a -> b -> b) -> b -> [a] -> [b]
scanrBy :: (b -> b -> Bool) -> (Int -> b) -> (a -> b -> b) -> b -> [a] -> [b]
scanl :: Eq b => (Int -> b) -> (b -> a -> b) -> b -> [a] -> [b]
scanlBy :: (b -> b -> Bool) -> (Int -> b) -> (b -> a -> b) -> b -> [a] -> [b]
scanr1 :: Eq a => (Int -> a) -> (a -> a -> a) -> [a] -> [a]
scanr1By :: (a -> a -> Bool) -> (Int -> a) -> (a -> a -> a) -> [a] -> [a]
scanl1 :: Eq a => (Int -> a) -> (a -> a -> a) -> [a] -> [a]
scanl1By :: (a -> a -> Bool) -> (Int -> a) -> (a -> a -> a) -> [a] -> [a]
```

Lots of Combinators! Control.Morphism.Speculation

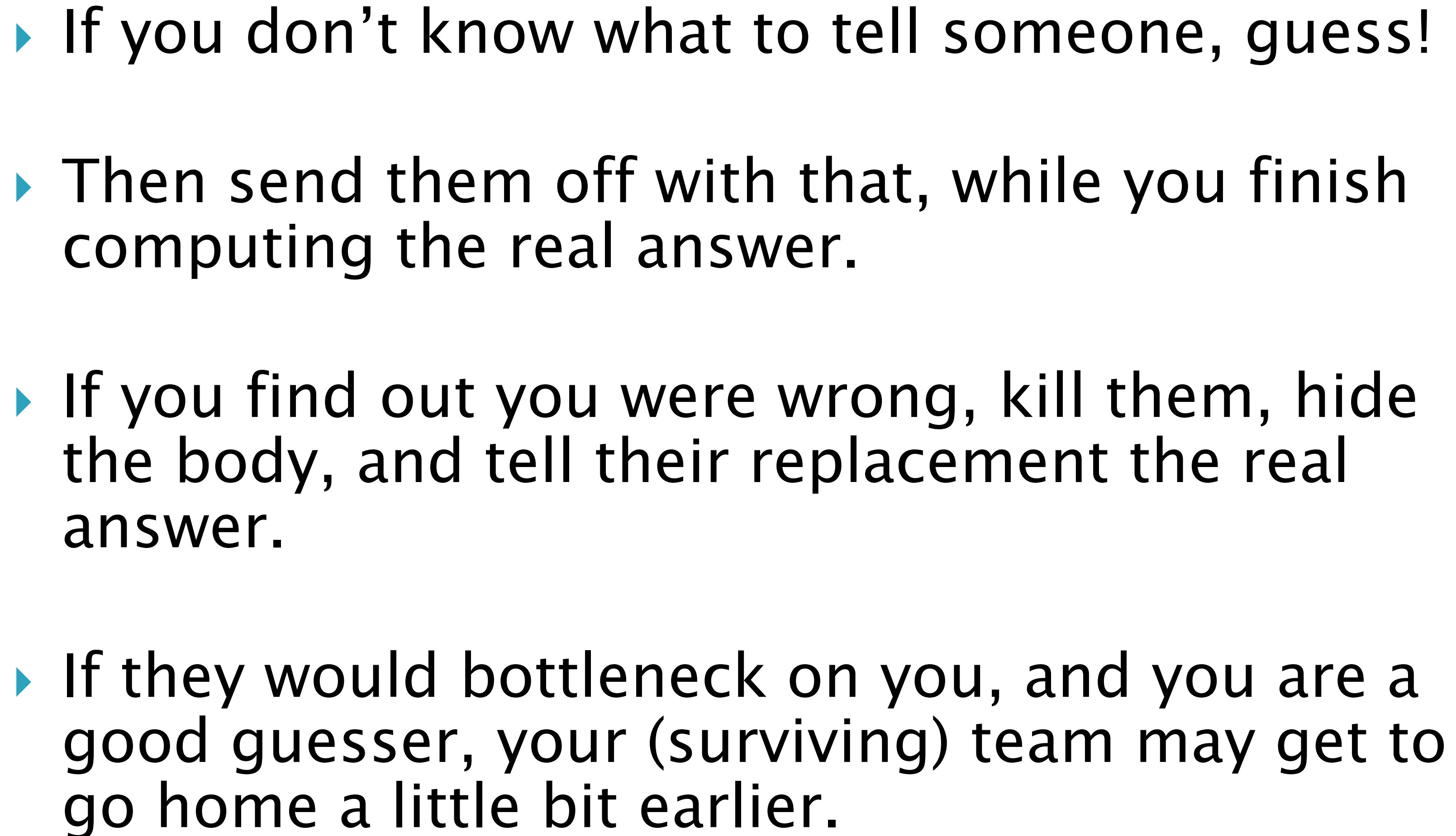
```
hylo :: (Functor f, Eq a) => (Int -> a) -> (f b -> b) -> (a -> f a) -> a -> b
```

`hylo g phi psi` is a hylomorphism using a speculative anamorphism, where `g n` estimates the seed after n iterations of psi.

Future Directions

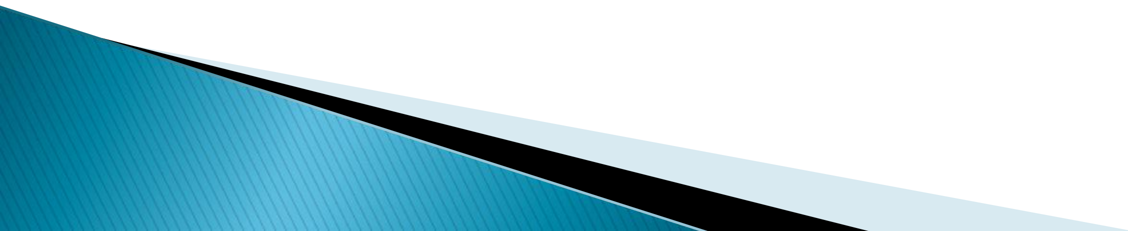


Lessons for the Real World



- If you don’t know what to tell someone, guess!
- Then send them off with that, while you finish computing the real answer.
- If you find out you were wrong, kill them, hide the body, and tell their replacement the real answer.
- If they would bottleneck on you, and you are a good guesser, your (surviving) team may get to go home a little bit earlier.

EXTRA SLIDES



Dynamic Pointer Tagging

- Simon Marlow, Alexey Rodriguez Yakushev, Simon Peyton Jones, *Faster Laziness using Dynamic Pointer Tagging.*