Compiling without continuations

Luke Maurer, Zena Ariola, Paul Downen, Simon Peyton Jones

December 2017

Peter Landin 1930-2009

“I recall knocking on Peter’s door one evening to show him some observations about the *zip* function. As I was writing down the observation on Peter’s blackboard, I asked him if he was familiar with this function; he replied “I think I may have invented it”.

Paul Boca

"Around Easter 1961, a course on ALGOL 60 was offered, with Peter Naur, Edsger W. Dijkstra, and Peter Landin as tutors. ... It was there that I wrote the procedure, immodestly named QUICKSORT, on which my career as a computer scientist is founded. Due credit must be paid to the genius of the designers of ALGOL 60 who included recursion in their language and enabled me to describe my invention so elegantly to the world. I have regarded it as the highest goal of programming language design to enable good ideas to be elegantly expressed.“

Tony Hoare

Peter Landin 1930-2009

"His approach is best described by John Reynolds: '*Peter Landin remarked long ago that the goal of his research was to tell beautiful stories about computation*' Reynolds (1999). Furthermore, Peter aimed for precision, to be contrasted with formality, in story telling.

…

"My recollection of Peter is that he was funny, generous, haughty, egalitarian, shy, uncertain, curious and ferociously intellectual. But, *boy* what beautiful stories!"

Tony Clark

Peter Landin 1930-2009

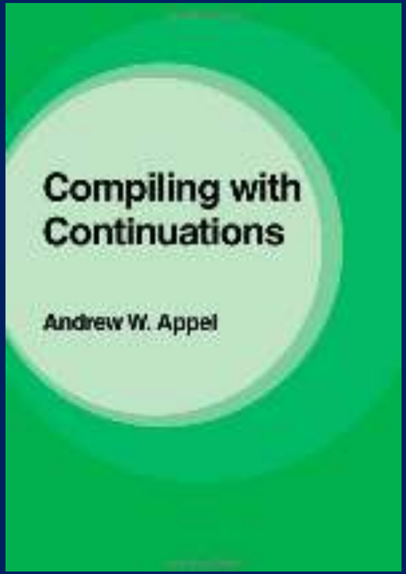
"To be able to translate Algol 60 into applicative expressions, Landin later extended these expressions and their interpreter with an assignment operation, and also a ***control operator J used to express the translation of goto's and labels*** [15, 16]. In the extended SECD machine, the result of applying J was a value containing a dump.

Thus, in modern terminology, the J operator provided a means of embedding continuations in values and was an ancestor of operations such as Reynolds's escape [36], and catch [44] and call/cc [8] in Scheme."

John Reynolds

Functional programming is all about data flow
But Landin recognised the importance of control flow too

Contuation passing style (CPS):
clever, but complicated



Zap! Flanagan et al, PLDI 1993

Blam! Kennedy et al, ICFP 2007

Pow!

Direct style with join points:

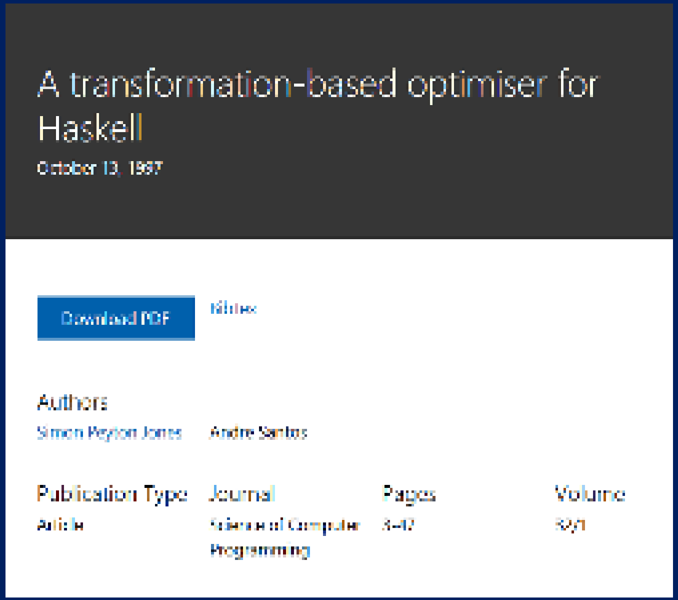
Simple, simple, simple

All the goodness of CPS with none of the pain

Unexpected new wins (fusion)

Works at scale (GHC)

Direct style:
simple,
but misses some tricks


A transformation-based optimiser for Haskell

October 13, 1997

Download PDF

Authors

Simon Peyton Jones

Andre Santos

Publication Type: Article

Journal: Science of Computer Programming

[PLDI 2017]

Natural deduction → Lambda calculus → Core GHC IR → Core + join points GHC's new IR

Curry-Howard isomorphism

Sequent logic → Sequent calculus → Sequent Core New GHC IR? → Core + join points GHC's new IR

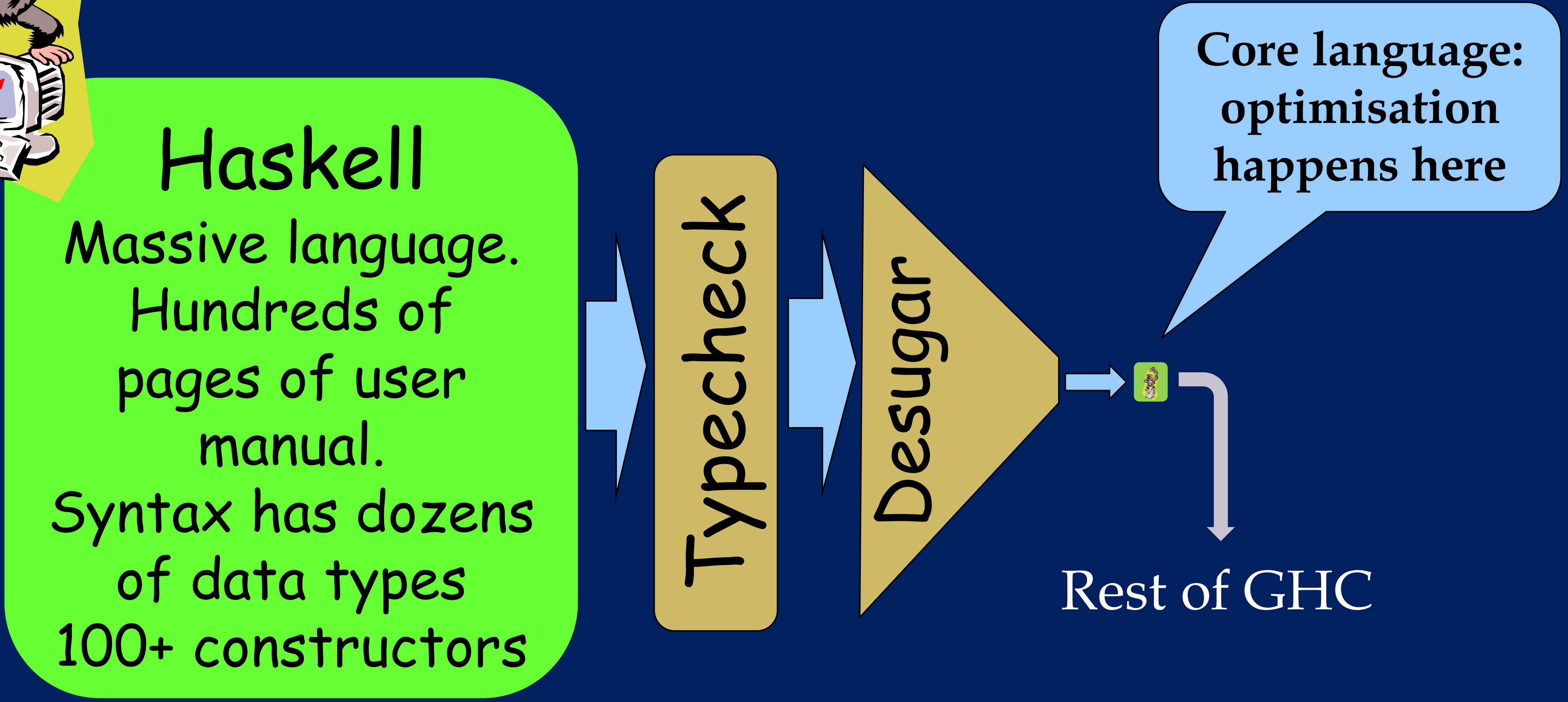
[ICFP 2016]

Zena Ariola

Paul Downen

Luke Maurer

How GHC works



Example

```haskell
module Prelude where

not :: Bool -> Bool
not True = False
not False = True

null :: [a] -> Bool
null []     = True
null (x:xs) = False

data Bool = False | True
data [a]  = []    | a:[a]
```

Haskell

```
module Prelude where

not :: Bool -> Bool
 = \(b::Bool). case b of
                 True  -> False
                 False -> True

null :: [a] -> Bool
  = \(xs::[a]). case xs of
                 []     -> True
                 (x:xs) -> False
```

Core

Example

```
data [a] = []  |  a : [a]

map :: (a->b) -> [a] -> [b]
map f []      = []
map f (x:xs) = f x : map f xs
```

```
map = \(f::a->b). \(xs:[a]).
        case xs of
          []    -> []
          y:ys -> f y : map f ys
```

Haskell

Core

Example

```
data [a] = []  |  a : [a]

filter :: (a->Bool) -> [a] -> [a]
filter p []     = []
filter p (x:xs) = if p x
                  then x : filter p xs
                  else     filter p xs
```

```
filter
  = \(p::a->Bool). \(xs:[a]).
    case xs of
      []    -> []
      y:ys -> case p y of
                True  -> y : filter p ys
                False ->     filter p ys
```

Haskell

Core

Commuting conversions

Case of case

```
notNull xs = not (null xs)
```

```
 = case (null xs) of
     True -> False
     False -> True
```

```
 = case (case xs of
              []     -> True
              (p:ps) -> False) of
     True -> False
     False -> True
```

```
module Prelude where

null xs = case xs of
             []      -> True
             (x:xs)  -> False

not b = case b of
           True  -> False
           False -> True
```

```
case e of
  True  -> r1
  False -> r2
```

means

```
if e
then r1
else r2
```

CASE: case of case

```
notNull xs = not (null xs)
```

```
 = case (case xs of
           []     -> True
           (p:ps) -> False) of
     True  -> False
     False -> True
```

```
 = case xs of
     []     -> case True of
                 True -> False
                 False -> True

     (p:ps) -> case False of
                 True  -> False
                 False -> True
```

```
 = case xs of
     []     -> False
     (p:ps) -> True
```

Commuting conversions. All good compilers do them.

A worry

```
= case (case xs of
          []     -> True
          (p:ps) -> False) of
    True  -> BIG1
    False -> BIG2
```

```
= case xs of
    []     -> case True of
                True  -> BIG1
                False -> BIG2

    (p:ps) -> case False of
                True  -> BIG1
                False -> BIG2
```

Duplicates arbitrary amounts of code :-(

Join points

```
= case (case xs of
          []     -> True
          (p:ps) -> False) of
    True  -> BIG1
    False -> BIG2
```

```
= let j1 () = BIG1
      j2 () = BIG2
  in case xs of
    []     -> case True of
                True  -> j1 ()
                False -> j2 ()

    (p:ps) -> case False of
                True  -> j1 ()
                False -> j2 ()
```

No duplication :-)

So far, a "join point" is just an ordinary function

Join points

```
= case (case xs of
          []     -> True
          (p:ps) -> False) of
    True  -> BIG1
    False -> BIG2
```

```
= let j1 () = BIG1
      j2 () = BIG2
  in case xs of
    []     -> case True of
                True  -> j1 ()
                False -> j2 ()

    (p:ps) -> case False of
                True  -> j1 ()
                False -> j2 ()
```

```
= let j1 () = BIG1
      j2 () = BIG2
  in case xs of
    []     -> j1 ()
    (p:ps) -> j2 ()
```

Ordinary inlining applies

```
= case xs of
    []     -> BIG1
    (p:ps) -> BIG2
```

Join points. All good compilers do them.

Arbitrary patterns

```
 = case (case xs of
           []     -> Nothing
           (p:ps) -> Just p) of
     Just x   -> BIG1[x]
     Nothing -> BIG2
```

Pattern binds variable x

```
 = let j1 x  = BIG1[x]
       j2 () = BIG2
   in case xs of
     []     -> case Nothing of
                  Just x  -> j1 x
                  Nothing -> j2 ()

     (p:ps) -> case Just p of
                  Just x  -> j1 x
                  Nothing -> j2 ()
```

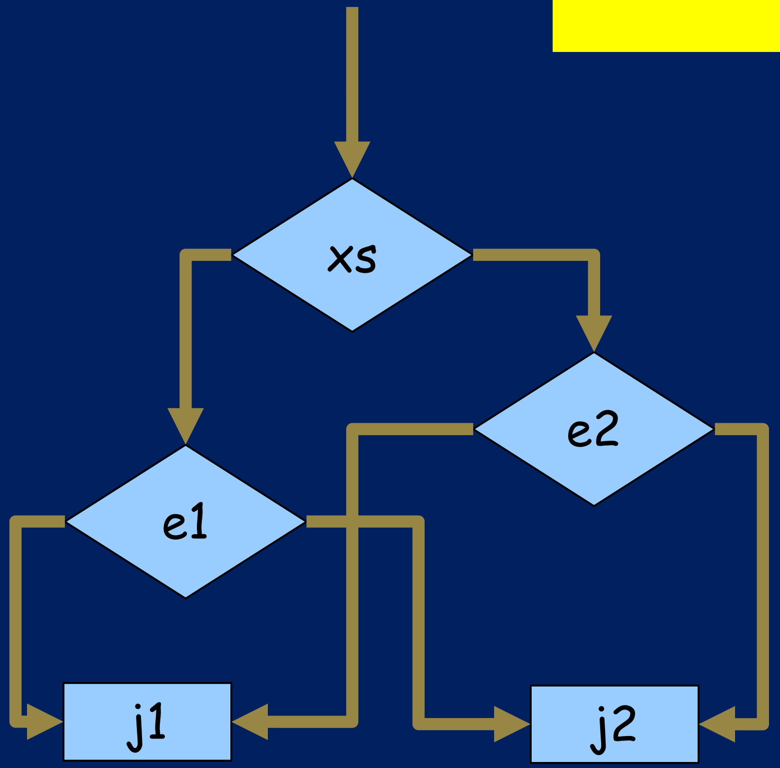
Simply abstract over the pattern bound variables

- Works fine for existentials, GADTs

Join points are like control-flow labels

```
= let j1 x  = let v = x+1 in Just v
      j2 () = BIG2
  in case xs of
     []     -> case e1 of
                 Just x -> j1 x
                 False  -> j2 ()

     (p:ps) -> case e2 of
                 True  -> j1 p
                 False -> j2 ()
```



- Let bindings allocate a thunk or data constructor
- Join points allocate nothing!
- "Calling" a join point = adjust stack pointer and goto

- Typically the back end
  - Spots functions that happen to be join points
  - Implements them as jumps

```
= let j1 x  = let v = x+1 in Just v
      j2 () = BIG2
  in case xs of
    []     -> case e1 of
                Just x -> j1 x
                False  -> j2 ()

    (p:ps) -> case e2 of
                True  -> j1 p
                False -> j2 ()
```

What characterises a join point?

- All calls are tail calls, relative to binding site
- No call is captured in a thunk or closure
- All calls saturated

The Main Idea of this talk

Problem: losing join points

```
case (let j x = E1
       in case xs of
            Just x   -> j x
            Nothing -> E2) of
     True  -> R1
     False -> R2
```

```
= let j x = E1
  in case xs of
    Just x  -> case (j x) of
                  True  -> R1
                  False -> R2
    Nothing -> case E2 of
                  True  -> R1
                  False -> R2
```

Bad bad bad!

Two bad things

1. 'j' is no longer a join point
2. The outer black case does not scrutinise E1

Keeping join points

```
case (let j x = E1
      in case xs of
           Just x   -> j x
           Nothing -> E2) of
     True  -> R1
     False -> R2
```


```
= let j' x = case E1 of
                True  -> R1
                False -> R2
  in case xs of
    Just x   -> j' x
    Nothing -> case E2 of
                  True  -> R1
                  False -> R2
```

Move outer case into join point

As well as the case RHS

1. 'j' remains a join point
2. The outer black case now scrutinises E1

Keeping join points

```
case (join j x = E1
      in case xs of
           Just x   -> jump j x
           Nothing -> E2) of
     True  -> R1
     False -> R2
```

⇓

```
= join j' x = case E1 of
                   True  -> R1
                   False -> R2
  in case xs of
     Just x   -> jump j' x
     Nothing -> case E2 of
                     True  -> R1
                     False -> R2
```

Move outer case into join point

Outer case evaporates when it hits a jump

Outer case wraps this RHS

Make join points part of the syntax

Very like let!

PS: if R1, R2 are big, then you can bind them as join points before doing this.

The solution: formalise join points as language construct

Terms

x	$\in$	Term variables	
j	$\in$	Label variables	
e, u, v	$::=$	$x \mid l \mid \lambda x{:}\sigma.e \mid e\,u$	
	$\mid$	$\Lambda a.e \mid e\,\varphi$	Type polymorphism
	$\mid$	$K\,\vec{\varphi}\;\vec{e}$	Data construction
	$\mid$	$\mathbf{case}\;e\;\mathbf{of}\;\overrightarrow{alt}$	Case analysis
	$\mid$	$\mathbf{let}\;vb\;\mathbf{in}\;v$	Let binding
	$\mid$	$\mathbf{join}\;jb\;\mathbf{in}\;u$	Join-point binding
	$\mid$	$\mathbf{jump}\;j\;\vec{\varphi}\;\vec{e}\;\tau$	Jump
alt	$::=$	$K\,\overrightarrow{x{:}\sigma} \to u$	Case alternative

Value bindings and join-point bindings

vb	$::=$	$x{:}\tau = e$	Non-recursive value
	$\mid$	$\mathbf{rec}\;\overrightarrow{x{:}\tau = e}$	Recursive values
jb	$::=$	$j\;\vec{a}\;\overrightarrow{x{:}\sigma} = e$	Non-recursive join point
	$\mid$	$\mathbf{rec}\;\overrightarrow{j\;\vec{a}\;\overrightarrow{x{:}\sigma} = e}$	Recursive join points

$$\boxed{\langle e;\ s;\ \Sigma\rangle \mapsto \langle e';\ s';\ \Sigma'\rangle}$$

$$\begin{array}{rcll}
\langle F[e];\ s;\ \Sigma\rangle & \mapsto & \langle e;\ F : s;\ \Sigma\rangle & (\mathit{push}) \\
\langle \lambda x.e;\ \Box\, v : s;\ \Sigma\rangle & \mapsto & \langle e;\ s;\ \Sigma, x = v\rangle & (\beta) \\
\langle \Lambda a.e;\ \Box\, \varphi : s;\ \Sigma\rangle & \mapsto & \langle e\{\varphi/a\};\ s;\ \Sigma\rangle & (\beta_\tau) \\
\langle \mathbf{let}\ vb\ \mathbf{in}\ e;\ s;\ \Sigma\rangle & \mapsto & \langle e;\ s;\ \Sigma, vb\rangle & (\mathit{bind}) \\
\langle x;\ s;\ \Sigma[x = v]\rangle & \mapsto & \langle v;\ s;\ \Sigma[x = v]\rangle & (\mathit{look}) \\
\left\langle \begin{array}{c} K\ \vec{\varphi}\ \vec{v}; \\ \mathbf{case}\ \Box\ \mathbf{of}\ \overrightarrow{alt} : s; \\ \Sigma \end{array} \right\rangle & \mapsto & \begin{array}{l} \langle u;\ s;\ \Sigma, \overrightarrow{x = v}\rangle \\ \text{if } (K\ \vec{x} \to u) \in \overrightarrow{alt} \end{array} & (\mathit{case}) \\
\left\langle \begin{array}{c} \mathbf{jump}\ j\ \vec{\varphi}\ \vec{v}\ \tau; \\ s' \mathbin{+\!\!+} (\mathbf{join}\ jb\ \mathbf{in}\ \Box : s); \\ \Sigma \end{array} \right\rangle & \mapsto & \begin{array}{l} \left\langle \begin{array}{c} u\{\overrightarrow{\varphi/a}\}; \\ \mathbf{join}\ jb\ \mathbf{in}\ \Box : s; \\ \Sigma, \overrightarrow{x = v} \end{array} \right\rangle \\ \text{if } (j\ \vec{a}\ \vec{x} = u) \in jb \end{array} & (\mathit{jump}) \\
\left\langle \begin{array}{c} A; \\ \mathbf{join}\ jb\ \mathbf{in}\ \Box : s; \\ \Sigma \end{array} \right\rangle & \mapsto & \langle A;\ s;\ \Sigma\rangle & (\mathit{ans})
\end{array}$$

Figure 3: Call-by-name operational semantics for System F_J.

$$\begin{array}{rcl}
F & ::= & \Box\, v \\
& | & \Box\, \tau \\
& | & \mathbf{case}\ \Box\ \mathbf{of}\ \overrightarrow{p \to u} \\
& | & \mathbf{join}\ jb\ \mathbf{in}\ \Box
\end{array}$$

- Claim: join points are control flow labels
- Operational semantics validates these claims

Core with join points

- **Identify** join points as a proper syntactic construct, with typing rules, operational semantics etc
- **Exploit** join points in commuting conversions
- **Infer** which let-bindings are in fact join points (contification)

Optimising transformations for join points

$$
\begin{aligned}
E[\mathbf{let}\ vb\ \mathbf{in}\ e] &= \mathbf{let}\ vb\ \mathbf{in}\ E[e] \\
E[\mathbf{join}\ j\ \vec{a}\ \vec{x} = u\ \mathbf{in}\ e] &= \mathbf{join}\ j\ \vec{a}\ \vec{x} = E[u]\ \mathbf{in}\ E[e]
\end{aligned}
$$

- Move evaluation context E into
  - RHS of join point bindings
  - as well as the body

Optimising transformations for join points

$$
\begin{array}{rcl}
E[\textbf{let}\ vb\ \textbf{in}\ e] & = & \textbf{let}\ vb\ \textbf{in}\ E[e] \\
E[\textbf{join}\ j\ \vec{a}\ \vec{x} = u\ \textbf{in}\ e] & = & \textbf{join}\ j\ \vec{a}\ \vec{x} = E[u]\ \textbf{in}\ E[e]
\end{array}
$$

```
case (join j x = B1
        in case xs of
           Just x   -> j x
           Nothing -> B2) of
     True   -> R1
     False -> R2
```

```
join j x = case B1 of
              True   -> R1
              False -> R2
  in case xs of
     Just x   -> j x
     Nothing -> case B2 of
                   True   -> R1
                   False -> R2
```

Here!

And here

Optimising transformations for join points

$$E[\mathbf{jump}\; j\; \vec{\varphi}\; \vec{e}\; \tau] : \tau' \quad = \quad \mathbf{jump}\; j\; \vec{\varphi}\; \vec{e}\; \tau'$$

- Discard E altogether at jumps

```
case (join j x = B1
          in case xs of
             Just x   -> j x
             Nothing -> B2) of
     True   -> R1
     False -> R2
```

```
join j x = case B1 of
              True   -> R1
              False -> R2
  in case xs of
    Just x   -> j x
    Nothing -> case B2 of
                 True   -> R1
                 False -> R2
```

Here!

Implementing join points

Implementing join points

- GHC: a big, 25-yr-old optimising compiler for Haskell
- But it was easy to add join points to GHC's tiny intermediate language, Core

A join-point binding is almost exactly like an ordinary letrec

```
join j x = <rhs>
in ...(jump j <arg>) ...
```

```
let j x = <rhs>
in ...(j <arg>) ...
```

Implementing join points

A join-point binding is almost exactly like an ordinary letrec

- Join points are just a variant of letrec, and share much in common (hurrah):
  - strictness analysis
  - inlining decisions
- But not all things! Every Core-to-Core pass needed review, often beneficial.
- Inferring join points is extremely easy (paper)
- Checked guarantee: join points are never lost

Performance

Program	Allocs
fibheaps	-1.1%
ida	-1.4%
nucleic2	+0.2%
para	-4.3%
primetest	-3.6%
simple	-0.9%
solid	-8.4%
sphere	-3.3%
transform	+1.1%
(45 others)	
Min	-8.4%
Max	+1.1%
Geo. Mean	-0.4%

Rather modest performance gains, but

- GHC already does a lot; no low hanging fruit
- Replaces some ad-hoc hacks with simpler, more reliable transformations
- Makes optimisation much more robust; less fragile to inlining decisions
- Absolutely nails some inner loops to zero
- And, intriguingly: may affect programming style

Recursive join points

Recursive join points

```
letrec last xs = case xs of
                   [x]    -> x
                   (x:xs) -> last xs
in last blah
```

```
joinrec last xs = case xs of
                    [x]    -> x
                    (x:xs) -> jump last xs
in jump last blah
```

- Join points can be recursive => loop in control flow graph
- Easy, easy. Everything just works.

Loopification

```
module Utils( last ) where
  last xs = case xs of
              [x]  -> x
              (x:xs) -> last xs
```

- Uh oh!   Now last is not a join point ☹
- Idea: introduce a local letrec

Loopification

```
module Utils( last ) where
  last xs = case xs of
                 [x] -> x
                 (x:xs) -> last xs
```

```
module Utils( last ) where
  last xs = joinrec lj xs = case xs of
                                  [x] -> x
                                  (x:xs) -> jump lj xs
            in jump lj xs
```

- The local loop is a join point ☺
- Introducing a join point directly expresses "turning tail recursion into a loop". Better code.
- Replaces a rather ad-hoc back-end optimisation

Fusion: an unexpected bonus

Streams

```
data Stream a where
  Mk :: forall a s. s -> (s->Step s a) -> Stream a

data Step s a = Done | Yield s a

filterS :: (a->Bool) -> Stream a -> Stream a
filterS p (Mk state step)
  = Mk state (\s. letrec step' s = case step s of
                    Done -> Done
                    Yield s' x | p x       -> Yield s' x
                               | otherwise -> step' s'
                  in step' s)
```

```
filter2 p q xs
  = filterS p (filterS q xs)
```

- step' is recursive so filter2 will not fuse
- So (Leshchinskiy et al) add a Skip constructor to Step.
- But that leads to other Bad Things.
- With join points: step' is now inferred as a (recursive) join point, so fusion just works without Skip.

Stream fusion

```
case ( joinrec step' s = case step s of              )
     (      Done                     -> Done         )
     (      Yield s' x | p x         -> Yield s' x   ) of
     (                 | otherwise -> jump step' s'  )
     (  in jump step' s                              )
   BLAH
```

```
joinrec step' s = case step s of
      Done                     -> case Done       of BLAH
      Yield s' x | p x         -> case Yield s' x of BLAH
                 | otherwise -> jump step' s'
in jump step' s
```

Commuting conversions automatically work over loops!

Inlining

GHC’s single most important optimisation decision

Inlining of join points: just like ordinary functions

```
join j x = <small>
in case y of
  True  -> jump j e1
  False -> jump j e2
```

Inline if <small> is small

```
join j x = <BIG>
in case y of
  True  -> e1
  False -> jump j e2
```

Inline if j is used only once

Inlining into join-point RHSs

```
let v = f 22
in join j x = …v…
in …
```

Build a thunk

Can we inline v?

Inlining into join point RHSs

```
let v = f 22
in join j x = …v…
in …
```

Yes! Inline v! Non-recursive join points can only be called once

```
join j x = …(f 22)…
in …
```

Exit floating

Exit floating

```
f x = let v = g x
      in joinrec go 10 = h (v + 8)
                 go i  = jump go (i+1)
         in jump go 0
```

- The local binding for v allocates a thunk for (g x)
- Call to h allocates a thunk for (v+8)
- h may or may not evaluate its argument

But we can do better!

Exit floating

```
f x = let v = g x
      in joinrec go 10 = h (v + 8)
                 go i  = jump go (i+1)
         in jump go 0
```

Thunk

Thunk

- The local binding for v allocates a thunk for (g x)
- Call to h allocates a thunk for (v+8)
- h may or may not evaluate its argument

Good! No thunk for v

We want this:

```
f x = join go 10 = h (g x + 8)
           go i  = jump go (i+1)
      in jump go 0
```

OK, so why not just inline v into the join-point RHS?

Exit floating

```
f x = let v = g x
      in joinrec go 10 = h 8
                 go i  = jump go (I + v)
         in jump go 0
```

Uses v every iteration

- But inlining v could be very very bad!

Inline v

```
f x = join go 10 = h 8
           go i  = jump go (i + g x)
      in jump go 0
```

BAD! Evaluates (g x) on every iteration

Exit floating

```
f x = let v = g x
      in joinrec go 10 = h 8
                 go i  = jump go (I + v)
         in jump go 0
```

Uses v every iteration

- But inlining v could be very very bad!

Inline v

```
f x = joi
                                          + g x)
         in jump go 0
```

No no no

- So we can't unconditionally inline a thunk into a joinrec
- But sometimes we want to!
- What we want: treat exit paths specially

Exit floating

```
f x = let v = g x
      in joinrec go 10 = h (v + 8)
                 go i = jump go (i+1)
         in jump go 0
```

Pull out exit path

Exit path as a join point

```
f x = let v = g x
      in join jx = h (v+8)
      in joinrec go 10 = jump jx
                 go i  = jump go (i+1)
      in jump go 0
```

- Key idea: float out the exit path as a join point
- Now we can inline v, because j is a non-recursive join point

Exit floating

```
f x = let v = g x
      in join jx = h (v+8)
      in joinrec go 10 = jump jx
                 go i  = jump go (i+1)
      in jump go 0
```

- Now we can inline v, because j is a non-recursive join point

Inline v

Exit path as a join point

```
f x = join jx = h (g x + 8)
      in joinrec go 10 = jump jx
                 go i  = jump go (i+1)
      in jump go 0
```

Wrap up

Bottom line

- An extremely (almost embarrassingly) simple idea
- Excellent power-to-weight ratio

It's a no-brainer
Every direct-style compiler
should use join points

- Paper: on my home page
  http://research.Microsoft.com/~simonpj

To CPS or not to CPS

CPS is extremely cool, but

- CPS fixes order of evaluation
- Some transformations much harder e.g
  - Common subexpression elimination
  - Rewrite rules   f (g x) --> h x

CPS: lambda-focused
Join points: let-focused

Join points appear to give you all the advantages of CPS with none of the pain