Unboxed values as first class citizens in a non-strict functional language

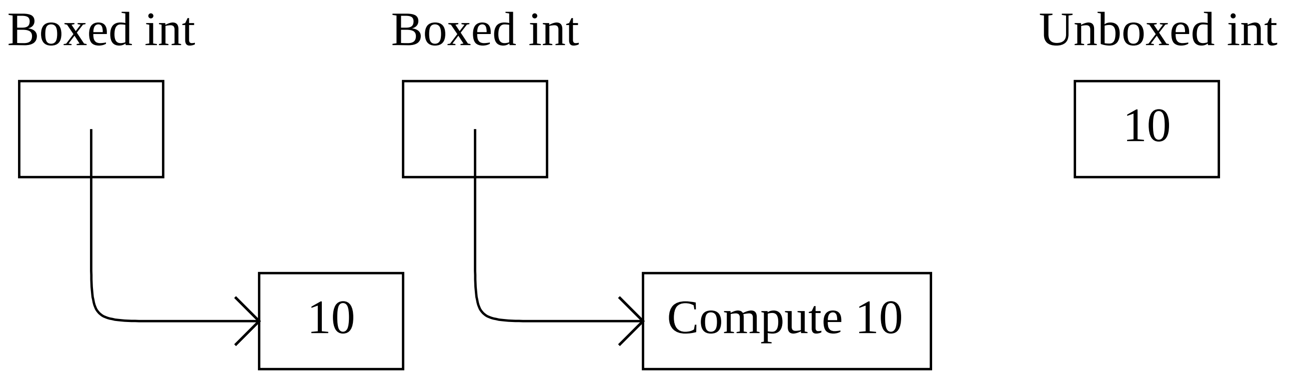
Simon L Peyton Jones and John Launchbury

Department of Computing Science, University of Glasgow

Presented by JungTaek Kim
October 21, 2000

Preliminaries

- Boxed vs. Unboxed



- Non-strict functional langauge (lazy evaluation)

  Evaluate the arguments when they are needed.

- First class citizens

  True values. We can pass the first class citizens as an argument or as a result.

- Strictness analysis

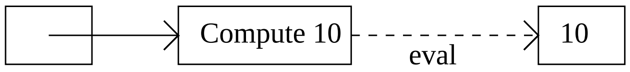
  A function $\texttt{f}$ *is strict for an argument* $\texttt{a}_\texttt{i}$:

  if $\texttt{e}_\texttt{i}$ is non-terminating, $\texttt{f} \cdots \texttt{e}_\texttt{i} \cdots$ is also non-terminating.

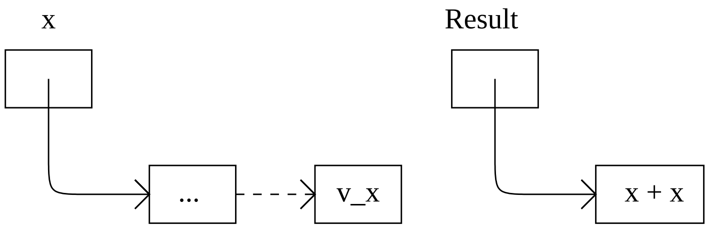
  For a strict argument, we can pass the argument after complete evaluation.

What's the problem?

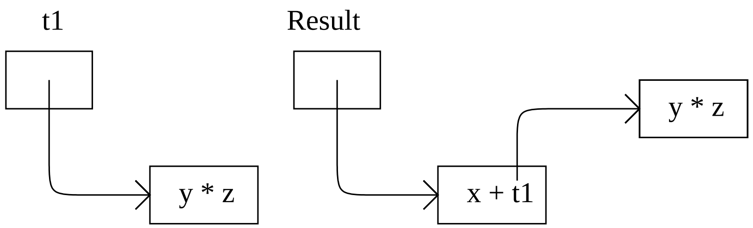
- In non-strict language, only boxed values are first class citizens.



- `double x = x + x`



- `f x y z = x + (y * z)`



- Optimization for this problem is done at code generation phase (using *ad hoc* manners). Usual program transformations can not be used .

Main Idea

- Make explicit the unboxed values in the language (make them first class citizens).
- Represent a boxed type using an unboxed type.

  For example,

  `data Int = Int Int#`.

  Boxed value `10` can be represented as `Int (10#)` using the unboxed value `10#`. `Int` means a *box* for the integer type terms.
- `double x = x + x` becomes

```
double x = case x of
           Int(x1#) -> case x of
                       Int(x2#) -> case (x1# +# x2#) of
                                   t# -> Int (t#)
```

Program transformation(I)

- We can use program transformations for explicit unboxed values.
- Example 1 : `double x = x + x`

```
double x = case x of
           Int(x1#) -> case x of
                       Int(x2#) -> case (x1# +# x2#) of
                                   t# -> Int (t#)
```

becomes

```
double x = case x of
           Int(x#) -> case (x# +# x#) of
                      t# -> Int (t#)
```

Program transformation(II)

- Example 2 : `f x y z = x + (y * z)`

```
f x y z = case x of Int(x#) ->
               case y of Int(y#) ->
                    case z of Int(z#) ->
                         case (y# *# z#) of
                         t1# -> case (Int t1#) of
                                Int(w#) -> case (x# +# w#) of
                                           t2# -> Int t2#
```

becomes

```
f x y z = case x of Int(x#) ->
               case y of Int(y#) ->
                    case z of Int(z#) ->
                         case (y# *# z#) of
                         t1# -> case (x# +# t1#) of
                                t2# -> Int t2#
```

Using strictness analysis(I)

- Previous optimizations are intra-procedural.
- All the functions received boxed arguments and return a boxed result.
- For a strict argument, we can pass an unboxed value (completely evaluated term).
- Divide a function into two function - worker and wrapper.
  - Worker : Is unboxed and do real thing.
  - Wrapper : Make the worker to seem boxed.
- Example `double x = x + x`

```
double x = case x of Int x# -> double# x#
double# x# = case (x# +# x#) of
             t# -> Int (t#)
```

And `double Int(10#)` becomes `double# 10#`.

Using strictness analysis(II)

- Another example : a factorial function `fac`.

```
fac n = case n of
        Int(n#) -> case n# of
                   0# -> 1
                   n'# -> n * (fac (n-Int(1#)))
```

becomes

```
fac n = case n of Int(n#) -> fac# n#

fac# n# = case n# -> of
          0# -> Int(1#)
          n'# -> case (n# -# 1#) of
                 n1# -> case (fac# n1#) of
                        Int(m#) -> case (n# *# m#) of
                                   r# -> Int(r#)
```

Using strictness analysis(III)

- In the previous example, `fac#` has type `Int# -> Int`, not `Int# -> Int#`.
- At the recursive call, we make new boxed `Int` and unbox it immediately.
- Result of the function should be used. (otherwise, why it is called?)
- Make `fac#` return an unboxed result.

```
fac n = case n of Int(n#) -> case (fac# n#) of t# -> Int(t#)

fac# n# = case n# -> of
          0# -> 1#
          n'# -> case (n# -# 1#) of
                 n1# -> case (fac# n1#) of
                        m# -> (n# *# m#)
```

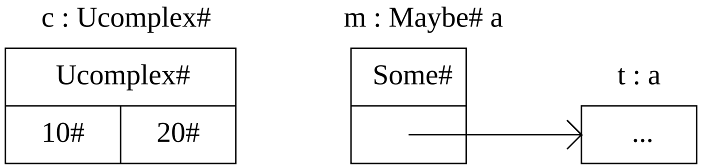
Generalising to data types(I)

- Generalise the unboxed idea to the arbitrary data types.
- Constructors can have unboxed arguments.
- Any data type can be unboxed.
- Example 1 : no argument.

```
data unboxed Color# = Red# | Green# | Blue#
```

- Example 2 : with arguments.

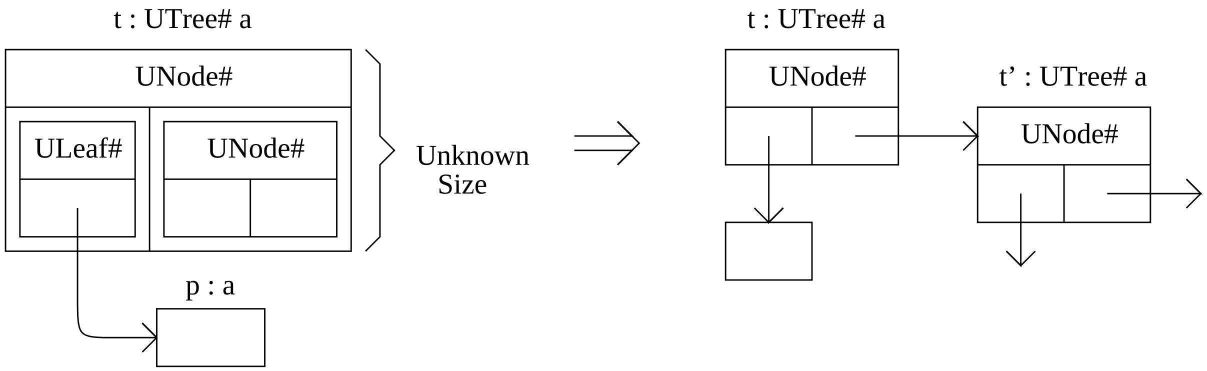
```
data unboxed UComplex# = UComplex# Int# Int#
data unboxed Maybe# a = Some# a | Nothing#
```



Generalising to data types(II)

- Recursive data types can not be fully unboxed.
- Example

  `data unboxed UTree# a = ULeaf# a | UNode# (UTree# a) (UTree# a)`



- In the recursive loop of data type, there should exist a boxed type.

Restrictions

- Loss of polymorphism.

  Polymorphic functions cannot manipulate unboxed values.

- Explicit evaluation.

  We can bind the unboxed variable only with completely evaluated unboxed expressions.

  ```
  let x# = (#3 +# #y)

  f# (x# +# y#)
  ```

  Completely evaluated unboxed expressions are :

  – an unboxed variable.

  – an unboxed constant.

  – an application of an unboxed constructor.

- No recursive unboxed data types

  In any recursive loop of types, there should exist a boxed type.

Semantics Change : Static semantics

- Type system should contains the restrictions.
- Loss of polymorphism.

  This can be represented by restricting the specialization rule.

$$\frac{\Gamma \vdash e : \forall \alpha . \sigma}{\Gamma \vdash e : \sigma[\tau/\alpha]} \implies \frac{\Gamma \vdash e : \forall \alpha . \sigma}{\Gamma \vdash e : \sigma[\pi/\alpha]}$$

when π is a boxed type

- Explicit evaluation.

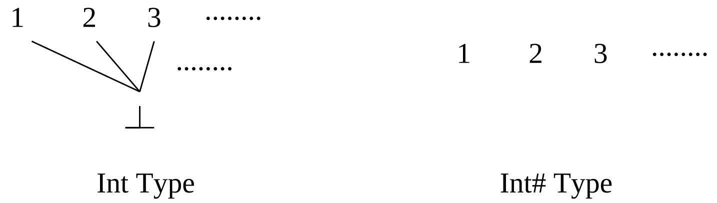
  This can be represented by restricting the application rule.

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \implies \frac{\Gamma \vdash e_1 : \pi \rightarrow \tau \quad \Gamma \vdash e_2 : \pi}{\Gamma \vdash e_1 e_2 : \tau} \;,\; \frac{\Gamma \vdash e_1 : \nu \rightarrow \tau \quad \Gamma \vdash h : \nu}{\Gamma \vdash e_1 h : \tau}$$

when π is a boxed type, ν is an unboxed type,
and h is an unboxed value.

Semantics Change : Dynamic semantics

- What's the interpretation of unboxed type?
- `Int` vs. `Int#`



- unpointed domains.

$$D[\![\texttt{Int}]\!] = \{\text{The fixed-precision integers}\}_\perp$$

$$D[\![\texttt{Int\#}]\!] = \{\text{The fixed-precision integers}\}$$

Conclusion

- Making explicit unboxed values enable us to manipulate the unboxed objects in the core language level.
- Unboxed optimization becomes a program transformation.
- Unboxed optimization can be used with other program transformations.