

Maximal Laziness

An Efficient Interpretation Technique for Purely Functional DSLs

Eelco Dolstra

Delft University of Technology, EWI,
Department of Software Technology

LDTA 2008
April 5, 2008



Technische Universiteit Delft

- ▶ Laziness: any variable is evaluated at most once.
- ▶ *Maximal laziness*: syntactically equal terms are evaluated at most once.
- ▶ So if any two terms e_1 and e_2 have the same AST, they are only evaluated once.
- ▶ Expensive in general, but trivial to implement in *term-rewriting interpreters* based on *maximal sharing*.
- ▶ Makes it easier to write efficient interpreters.
 - ▶ No closure updating needed.
 - ▶ Translation of call-by-name semantic rules gives a call-by-need interpreter.
 - ▶ Reduces gap between language specification and implementation.
- ▶ Used in the *Nix expression DSL*.

Motivating example: Nix

- ▶ Nix: A purely functional package manager
- ▶ *Purely functional* package management: package builds only depend on declared inputs; never change after they have been built.
- ▶ Main features:
 - ▶ Enforce correct dependency specifications.
 - ▶ Support concurrent variants/versions.
 - ▶ Safe and automatic garbage collection of unused packages.
 - ▶ Transparent source/binary deployment model.
 - ▶ Atomic upgrades/rollbacks.
 - ▶ Purely functional language (*Nix expressions* for describing packages).
 - ▶ ...
- ▶ Forms the basis of NixOS, a purely functional Linux distribution (<http://nixos.org/>).

Motivating example: Nix

- ▶ Nix: A purely functional package manager
- ▶ *Purely functional* package management: package builds only depend on declared inputs; never change after they have been built.
- ▶ Main features:
 - ▶ Enforce correct dependency specifications.
 - ▶ Support concurrent variants/versions.
 - ▶ Safe and automatic garbage collection of unused packages.
 - ▶ Transparent source/binary deployment model.
 - ▶ Atomic upgrades/rollbacks.
 - ▶ Purely functional language (*Nix expressions* for describing packages).
 - ▶ ...
- ▶ Forms the basis of NixOS, a purely functional Linux distribution (<http://nixos.org/>).

Motivating example: Nix

- ▶ Nix: A purely functional package manager
- ▶ *Purely functional* package management: package builds only depend on declared inputs; never change after they have been built.
- ▶ Main features:
 - ▶ Enforce correct dependency specifications.
 - ▶ Support concurrent variants/versions.
 - ▶ Safe and automatic garbage collection of unused packages.
 - ▶ Transparent source/binary deployment model.
 - ▶ Atomic upgrades/rollbacks.
 - ▶ Purely functional language (*Nix expressions* for describing packages).
 - ▶ ...
- ▶ Forms the basis of NixOS, a purely functional Linux distribution (<http://nixos.org/>).

The Nix expression language

Packages are built from *Nix expressions*, a dynamically typed, lazy, purely functional language.

```
helloFun =
{stdenv, fetchurl, perl}:

stdenv.mkDerivation {
  name = "hello-2.1.1";
  src = fetchurl {
    url = mirror://gnu/hello/hello-2.1.1.tar.gz;
    md5 = "70c9ccf9fac07f762c24f2df2290784d";
  };
  buildInputs = [perl];
};

hello = helloFun {
  inherit fetchurl stdenv;
  perl = perl58;
};

stdenv = ...; perl58 = ...; perl510 = ...; fetchurl = ...;
```

The Nix expression language

Packages are built from *Nix expressions*, a dynamically typed, lazy, purely functional language.

```
helloFun =  
{stdenv, fetchurl, perl}:
```

Function arguments

```
  stdenv.mkDerivation {  
    name = "hello-2.1.1";  
    src = fetchurl {  
      url = mirror://gnu/hello/hello-2.1.1.tar.gz;  
      md5 = "70c9ccf9fac07f762c24f2df2290784d";  
    };  
    buildInputs = [perl];  
  };
```

```
hello = helloFun {  
  inherit fetchurl stdenv;  
  perl = perl58;  
};
```

```
stdenv = ...; perl58 = ...; perl510 = ...; fetchurl = ...;
```

The Nix expression language

Packages are built from *Nix expressions*, a dynamically typed, lazy, purely functional language.

```
helloFun =  
  {stdenv, fetchurl, perl}:  
  
  stdenv.mkDerivation {  
    name = "hello-2.1.1";  
    src = fetchurl {  
      url = mirror://gnu/hello/hello-2.1.1.tar.gz;  
      md5 = "70c9ccf9fac07f762c24f2df2290784d";  
    };  
    buildInputs = [perl];  
  };  
  
hello = helloFun {  
  inherit fetchurl stdenv;  
  perl = perl58;  
};  
  
stdenv = ...; perl58 = ...; perl510 = ...; fetchurl = ...;
```

Build attributes

The Nix expression language

Packages are built from *Nix expressions*, a dynamically typed, lazy, purely functional language.

```
helloFun =  
  {stdenv, fetchurl, perl}:  
  
  stdenv.mkDerivation {  
    name = "hello-2.1.1";  
    src = fetchurl {  
      url = mirror://gnu/hello/hello-2.1.1.tar.gz;  
      md5 = "70c9ccf9fac07f762c24f2df2290784d";  
    };  
    buildInputs = [perl];  
  };  
  
hello = helloFun {  
  inherit fetchurl stdenv;  
  perl = perl58;  
};  
  
stdenv = ...; perl58 = ...; perl510 = ...; fetchurl = ...;
```

Function call

Syntax

- ▶ Plain lambdas: arg: body
- ▶ The most important type, *attribute sets*:
`{ x = "foo"; y = 123; }`
- ▶ Attribute selection: `{ x = "foo"; y = 123; }.y`
evaluates to 123
- ▶ Recursive attribute sets: `rec { x = y; y = 123; }.x`
 - ▶ Evaluates to 123.
- ▶ Inheriting from the lexical scope: `x: { inherit x; y = 123; }`
 - ▶ So inherit x is basically sugar for `x = x;`
 - ▶ But not in recs: `x: rec { inherit x; y = 123; }` doesn't get in an infinite loop.
- ▶ Pattern matching on attribute sets: `{x, y}: x + y`
 - ▶ Argument order doesn't matter:
`({x, y}: x + y) {y = "bar"; x = "foo";}`
yields "foobar"

Syntax

- ▶ Plain lambdas: arg: body
- ▶ The most important type, *attribute sets*:
`{ x = "foo"; y = 123; }`
- ▶ Attribute selection: `{ x = "foo"; y = 123; }.y`
evaluates to 123
- ▶ Recursive attribute sets: `rec { x = y; y = 123; }.x`
 - ▶ Evaluates to 123.
- ▶ Inheriting from the lexical scope: `x: { inherit x; y = 123; }`
 - ▶ So inherit x is basically sugar for `x = x;`
 - ▶ But not in recs: `x: rec { inherit x; y = 123; }` doesn't get in an infinite loop.
- ▶ Pattern matching on attribute sets: `{x, y}: x + y`
 - ▶ Argument order doesn't matter:
`({x, y}: x + y) {y = "bar"; x = "foo";}`
yields "foobar"

Syntax

- ▶ Plain lambdas: arg: body
- ▶ The most important type, *attribute sets*:
 $\{ x = "foo"; y = 123; \}$
- ▶ Attribute selection: $\{ x = "foo"; y = 123; \}.y$
evaluates to 123
- ▶ Recursive attribute sets: $rec \{ x = y; y = 123; \}.x$
 - ▶ Evaluates to 123.
- ▶ Inheriting from the lexical scope: $x: \{ inherit x; y = 123; \}$
 - ▶ So inherit x is basically sugar for $x = x;$
 - ▶ But not in recs: $x: rec \{ inherit x; y = 123; \}$ doesn't get in an infinite loop.
- ▶ Pattern matching on attribute sets: $\{x, y\}: x + y$
 - ▶ Argument order doesn't matter:
 $(\{x, y\}: x + y) \{y = "bar"; x = "foo";\}$
yields "foobar"

Syntax

- ▶ Plain lambdas: arg: body
- ▶ The most important type, *attribute sets*:
 $\{ x = "foo"; y = 123; \}$
- ▶ Attribute selection: $\{ x = "foo"; y = 123; \}.y$
evaluates to 123
- ▶ Recursive attribute sets: $rec \{ x = y; y = 123; \}.x$
 - ▶ Evaluates to 123.
- ▶ Inheriting from the lexical scope: $x: \{ inherit x; y = 123; \}$
 - ▶ So inherit x is basically sugar for $x = x;$
 - ▶ But not in recs: $x: rec \{ inherit x; y = 123; \}$ doesn't get in an infinite loop.
- ▶ Pattern matching on attribute sets: $\{x, y\}: x + y$
 - ▶ Argument order doesn't matter:
 $(\{x, y\}: x + y) \{y = "bar"; x = "foo";\}$
yields "foobar"

Syntax

- ▶ Plain lambdas: arg: body
- ▶ The most important type, *attribute sets*:
 $\{ x = "foo"; y = 123; \}$
- ▶ Attribute selection: $\{ x = "foo"; y = 123; \}.y$
evaluates to 123
- ▶ Recursive attribute sets: $rec \{ x = y; y = 123; \}.x$
 - ▶ Evaluates to 123.
- ▶ Inheriting from the lexical scope: $x: \{ inherit x; y = 123; \}$
 - ▶ So inherit x is basically sugar for $x = x;$
 - ▶ But not in recs: $x: rec \{ inherit x; y = 123; \}$ doesn't get in an infinite loop.
- ▶ Pattern matching on attribute sets: $\{x, y\}: x + y$
 - ▶ Argument order doesn't matter:
 $(\{x, y\}: x + y) \{y = "bar"; x = "foo";\}$
yields "foobar"

Rewrite rules: small step semantics

Rules for if-then-else:

$$\text{IFTHEN} : \frac{e_1 \xrightarrow{*} \text{true}}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mapsto e_2}$$

$$\text{IFELSE} : \frac{e_1 \xrightarrow{*} \text{false}}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mapsto e_3}$$

Semantics (cont'd)

Function calls: β -reduction

$$\beta\text{-REDUCE} : \frac{e_1 \stackrel{*}{\mapsto} x: e_3}{e_1 \ e_2 \mapsto \text{subst}(\{x \rightsquigarrow e_2\}, e_3)}$$

Substitution

$$\text{subst}(subs, x) = \begin{cases} e & \text{if } (x \rightsquigarrow e) \in subs \\ x & \text{otherwise} \end{cases}$$

$$\begin{aligned} \text{subst}(subs, \{as\}) &= \\ &\{\text{map}(\lambda \langle n = e \rangle. \langle n = \text{subst}(subs, e) \rangle, as)\} \end{aligned}$$

$$\begin{aligned} \text{subst}(subs, x: e) &= x: \text{subst}(subs', e) \\ \text{where } subs' &= \{x_2 \rightsquigarrow e | x_2 \rightsquigarrow e \in subs \wedge x \neq x_2\} \end{aligned}$$

...

Function calls with attribute sets

$$\beta\text{-REDUCE}': \frac{e_1 \xrightarrow{*} \{fs\}: e_3 \wedge e_2 \xrightarrow{*} \{as\} \wedge \text{names}(as) = fs}{e_1 \ e_2 \mapsto \text{subst}(\{n \rightsquigarrow e \mid \langle n = e \rangle \in as\}, e_3)}$$

Implementing an interpreter

- ▶ Semantic rules are easily to implement.
- ▶ For instance the IFTHEN rule

$$\text{IFTHEN} : \frac{e_1 \xrightarrow{*} \text{true}}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mapsto e_2}$$

in Stratego:

```
eval: If(e1, e2, e3) -> e2
    where <eval> e1 => Bool(True)
```

Implementing an interpreter

- ▶ Nix is implemented in C++, but it's still straight-forward, e.g. the IFTHEN and β -REDUCE rules:

```
Expr eval(Expr e)
{
    Expr e1, e2, e3;
    if (matchIf(e, e1, e2, e3) && evalBool(e1))
        return eval(e2);

    ATerm x;
    if (matchCall(e, e1, e2) &&
        matchFunction1(eval(e1), x, e3)) {
        ATermMap subs; subs.set(x, e2);
        return eval(subst(subs), e3);
    }

    ...
}
```

- ▶ C++ implementation uses *ATerms*.

Implementing an interpreter

- ▶ Nix is implemented in C++, but it's still straight-forward, e.g. the IFTHEN and β -REDUCE rules:

```
Expr eval(Expr e)
{
    Expr e1, e2, e3;
    if (matchIf(e, e1, e2, e3) && evalBool(e1))
        return eval(e2);
    else
        ATerm
$$\text{IFTHEN} : \frac{e_1 \xrightarrow{*} \text{true}}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \xrightarrow{} e_2}$$

    if (matchCall(e, e1, e2) &&
        matchFunction1(eval(e1), x, e3)) {
        ATermMap subs; subs.set(x, e2);
        return eval(subst(subs), e3);
    }
    ...
}
```

- ▶ C++ implementation uses *ATerms*.

Implementing an interpreter

- ▶ Nix is implemented in C++, but it's still straight-forward, e.g. the IFTHEN and β -REDUCE rules:

```
Expr eval(Expr e)
{
    Expr e1, e2, e3;
    if (matchIf(e, e1, e2, e3) && evalBool(e1))
        return eval(e2);

    ATerm x;
    if (matchCall(e, e1, e2) &&
        matchFunction1(eval(e1), x, e3)) {
        ATermMap subs; subs.set(x, e2);
        return eval(subst(subs), e3);
    }
    ...
}
```

$$\beta\text{-REDUCE} : \frac{e_1 \xrightarrow{*} x: e_3}{e_1 \ e_2 \mapsto \text{subst}(\{x \rightsquigarrow e_2\}, e_3)}$$

- ▶ C++ implementation uses *ATerms*.

The problem

- ▶ The Stratego/C++ implementations are incredibly slow.
- ▶ Call-by-name semantics: arguments directly substituted in function bodies \Rightarrow work duplication.
- ▶ Solution: call-by-need / laziness: evaluate only once.
- ▶ Requires updating semantics.
- ▶ Significantly complicates the interpreter mechanics: need to keep environments of variables in scope, etc.

Maximal sharing

- ▶ Maximal sharing (aka hash-consing): equal terms are stored only once in memory.
- ▶ So term equality testing == simple pointer equality test.
- ▶ Term creation becomes a bit more expensive (maybe).
- ▶ Significantly less memory use.

Maximal laziness

- ▶ Just memoise the eval() function, i.e. every evaluation result.

```
Expr eval(Expr e) :  
    if cache[e] ≠ ε :  
        return cache[e]  
    else :  
        e' ← realEval(e)  
        cache[e] ← e'  
        return e'
```

Maximal laziness (cont'd)

- ▶ Now the simple BetaReduce rule is suddenly efficient!
- ▶ E.g. $(x : x + x)e \xrightarrow{*} e + e$;
expression e will be cached, so evaluated only once.
- ▶ No direct “updating” semantics needed, plain “call by name” rule is sufficient.

Optimising substitution

- ▶ Just one problem: subst now becomes the bottleneck.
- ▶ It will substitute under previously substituted terms.
- ▶ E.g. $(x : y : e_1) e_2 e_3$ where e_2 is large.
- ▶ Second substitution will traverse into e_2 .
- ▶ Unnecessary since e_2 is a closed term (invariant).
- ▶ ATerms are a graph, but if you naively recurse over them they become a tree.

Optimising substitution (cont'd)

- ▶ Solution: wrap closed terms in $\text{closed}(e)$ nodes.

$$\begin{aligned}\text{subst}(\text{subs}, x) &= \begin{cases} \text{closed}(e) & \text{if } (x \rightsquigarrow \text{closed}(e)) \in \text{subs} \\ \text{closed}(e) & \text{if } (x \rightsquigarrow e) \in \text{subs} \\ x & \text{otherwise} \end{cases} \\ \text{subst}(\text{subs}, \text{closed}(e)) &= \text{closed}(e)\end{aligned}$$

- ▶ $\text{closed}(e)$ is a semantic no-ops:

$$\text{CLOSED} : \text{closed}(e) \mapsto e$$

Simple extension to detect some kinds of infinite recursion.

```
Expr eval(Expr e) :  
    if cache[e] ≠ ε :  
        if cache[e] = blackhole :  
            Abort; infinite recursion detected.  
        return cache[e]  
    else :  
        cache[e] ← blackhole  
        e' ← realEval(e)  
        cache[e] ← e'  
    return e'
```

Blackholing (cont'd)

Detects more kinds of infinite recursion than blackholing in GHC:

```
(rec {f = x: f x;}).f 10
```

evaluates as

$$\begin{array}{lcl} & (\text{rec } \{f = x: f x; \}).f 10 \\ (\text{REC}) & \mapsto & \{f = x: (\text{rec } \{f = x: f x; \}).f x; \}.f 10 \\ (\text{SELECT}) & \mapsto & (x: (\text{rec } \{f = x: f x; \}).f x) 10 \\ (\beta\text{-REDUCE}) & \mapsto & (\text{rec } \{f = x: f x; \}).f 10 \end{array}$$

Optimisations: full laziness

- ▶ Move subexpressions outward as far as possible:
let { $f x = \text{let } \{y = \text{fac } 100\} \text{ in } x + y\}$ } in $f 1 + f 2$
becomes
let { $y = \text{fac } 100$; $f x = x + y$ } in $f 1 + f 2$
- ▶ Full laziness transform is unnecessary here: inner terms are automatically shared between calls.

Function memoisation

- ▶ Slow function:

```
fib = n: if n == 0 then 0 else  
           if n == 1 then 1 else fib (n-1) + fib (n-2);
```

but becomes fast if memoised.

- ▶ Functions are memoised automatically now...
- ▶ ... but it won't do any good (usually) in a non-strict language.
- ▶ This is because the function argument is an unevaluated AST.
- ▶ I.e. instead of `Int(1)`, `Int(2)`, ... we get `OpMin(OpMin(Int(9), Int(1)), Int(1))` etc.
- ▶ Solution: function short-circuiting.
 - ▶ When you're evaluating a function $f x$...
 - ▶ ... and you at some point find the normal form x' of x
 - ▶ ... and $f x'$ is in the cache
 - ▶ then unwind the stack, return the normal form of $f x'$.

Evaluation

- ▶ Scales to large Nix expressions
- ▶ `nix-env -qa`: evaluates all packages in the Nix Packages collection, shows information about them
 - ▶ 743 source files
 - ▶ 22191 lines of code
 - ▶ 2.75 seconds on Athlon X2 3800+
 - ▶ Consumes 21 MiB of memory
- ▶ Evaluating the derivation graph for NixOS
 - ▶ 162 source files
 - ▶ 13503 lines of code
 - ▶ 0.99 seconds

Conclusion

- ▶ Simple techniques that allows purely functional DSLs to be implemented/prototyped quickly
- ▶ Straight-forward implementation of semantic rules gives a reasonably efficient implementation: good enough for Nix over the last few years
- ▶ Future work: reducing memory use by clearing the cache periodically