ML-specific optimizations

Łukasz Czapliński

October 28, 2015

What do we have today?

Objectives

- Named records
- Data types
- Pattern matching
- Equality
- Exceptions
- Module system

Tools

- Type system
- Records (tuples)
- Functions
- our invention

Named records

Examples

```
type t = { name: string, number: int }
val t_ins = { name="Sam", number=5 }
type 'a r = { name: string, number: 'a }
```

Named records

Examples

```
type t = { name: string, number: int }
val t_ins = { name="Sam", number=5 }
type 'a r = { name: string, number: 'a }
```

Implementation

Named records

Examples

```
type t = { name: string, number: int }
val t_ins = { name="Sam", number=5 }
type 'a r = { name: string, number: 'a }
```

Implementation

Simple records

Modules and Functors

Example

```
structure Stack1 =
    struct type 'a stack = 'a list
    fun push(a,s) = a::s
    fun top(a::rest) = a | top(nil) = raise Empty
    fun pop(a::rest) = rest | pop(nil) = raise Empty
    val empty = nil
end
```

Signatures

Example

```
signature STACK =
    sig   type 'a stack
    val empty : 'a stack
    val push: 'a * 'a stack -> 'a stack
    val top : 'a stack -> 'a
    val pop : 'a stack -> 'a stack
end
```

Signatures

Example

```
signature STACK =
    sig   type 'a stack
    val empty : 'a stack
    val push: 'a * 'a stack -> 'a stack
    val top : 'a stack -> 'a
    val pop : 'a stack -> 'a stack
end
```

Example

```
structure Stack2 : STACK =
    struct datatype 'a stack = empty
                             | push of 'a * 'a stack
    fun top(push(a,rest)) = a
    fun pop(push(a,rest)) = rest
end
```

Functors

Example

```
functor F(S : STACK) = struct
    val em = S.empty
end
structure T = F(Stack2)
```

Implementation

Implementation

- Modules $\rightarrow$ Records
- Functors $\rightarrow$ Functions

Linker and runtime system have no idea of module system

Data types

Examples

```
type posint = int (* positive integers *)
datatype money = COIN of posint | BILL of posint
     | CHECK of {amount:real, from: string}
datatype color = RED | BLUE | GREEN | YELLOW
datatype 'a list = nil | :: of 'a * 'a list
datatype register = REG of int
datatype tree = LEAF of int | TREE of tree * tree
datatype xxx = M | N | P of int list
datatype xxxp = M | N | P of tree
datatype foo = F of int | O of tree
datatype yyy = W of int * int | X of real * real * real
datatype gen = A | B | C | D of int | E of real
     | F of gen * gen | G of int * int * gen
```

Assumption 0

Pointers indistinguishable from integers

Tagged

```
datatype money = COIN of posint | BILL of posint
    | CHECK of {amount:real, from: string}
```

Assumption 0

Pointers indistinguishable from integers

Tagged

```
datatype money = COIN of posint | BILL of posint
    | CHECK of {amount:real, from: string}
```

Transparent

```
datatype register = REG of int
```

Assumption 1

Pointers only distinguishable from small integers

Assumption 1

Pointers only distinguishable from small integers

Constant

```
datatype color = RED | BLUE | GREEN | YELLOW
```

Assumption 1

Pointers only distinguishable from small integers

Constant

```
datatype color = RED | BLUE | GREEN | YELLOW
```

Better transparent constructors

TransB

```
datatype xxx = M | N | P of int list
```

TransU

```
datatype color = RED | BLUE | GREEN | YELLOW
datatype rgb_color = RGB of int * int * int
                   | BASIC of color
```

Assumption 2

All pointers distinguishable from all integers

Assumption 2

All pointers distinguishable from all integers

Optimization

```
datatype tree = LEAF of int | TREE of tree * tree
```

Assumption 3

Pointers to records of different length distinguishable from each other

Assumption 3

Pointers to records of different length distinguishable from each other

Optimization

```
datatype yyy = W of int * int | X of real * real * real
```

Problems

Polymorphic datatypes

```
datatype 'a t = A of 'a | B of (real*real)
type u = int t
```

Problems

Polymorphic datatypes

```
datatype 'a t = A of 'a | B of (real*real)
type u = int t
```

Functors

```
functor F(S: sig  type 'a t
datatype 'a list = nil | :: of 'a t
end
) = struct ... end
structure A = struct
datatype 'a list = nil | :: of 'a * 'a list
type 'a t = 'a * 'a list
end

structure FA = F(A) (* Where is your runtime system now? *)
```

Conclusion

Choosing best solution

- Use only assumption 1
- Constructors
  - Tagged
  - Constant
  - Transparent
  - TransB
- Exceptions handled separately
  - Variable
  - VariableC
- Functor mismatch errors at functor-application time

Exceptions

Open type (unbounded number of constructors)

Example

Module A

```
exception C
exception D = J
```

Module B

```
exception E of int
```

Exceptions

Open type (unbounded number of constructors)

Example

Module A

```
exception C
exception D = J
```

Module B

```
exception E of int
```

Implementation

Tagged type

Pattern matching

Match between value of expression and rule of pattern-expression list

Example

```
case a
    of (false, nil)    =>  nil
    | (true, w)        =>   w
    | (false, x::nil) =>  x::x::nil
    | (false, y::z)    =>  z
```

Pattern matching

Match between value of expression and rule of pattern-expression list

Example

```
case a
    of (false, nil)    =>  nil
    | (true, w)        =>   w
    | (false, x::nil) =>  x::x::nil
    | (false, y::z)    =>  z
```

Implementation

Use special instruction: decon. Apply one level of deconstruction per switch.

Equality

Structural equality

```
[1,2,3] = [1,2,3] (* true *)
ref 5 = ref 5 (* false *)
```

Equality

Structural equality

```
[1,2,3] = [1,2,3] (* true *)
ref 5 = ref 5 (* false *)
```

Enter polymorphism

```
fun member(x, a::rest) = x=a orelse member(x,rest)
  | member(x, nil) = false
```

Equality

Structural equality

```
[1,2,3] = [1,2,3] (* true *)
ref 5 = ref 5 (* false *)
```

Enter polymorphism

```
fun member(x, a::rest) = x=a orelse member(x,rest)
    | member(x, nil) = false
```

Implementation

Use runtime tags to compare records field-for-field. Recall that's assumption 3!

Unboxed updates

Example

```
datatype color = Red | Green | Blue (* unboxed *)
val x = ref Red
val _ = x := Green
```

Unboxed updates

Example

```
datatype color = Red | Green | Blue (* unboxed *)
val x = ref Red
val _ = x := Green
```

Implementation

Use types to conservatively mark safe operations to use unboxed instructions.

Our approach and problems

. . . Look at the code . . .