The MetaOCaml files
Status report and research proposal

Oleg Kiselyov
Chung-chieh Shan

ICFP Eve, 2010  Somewhere in Maryland
Q&A

How to reconcile generality with performance?

▶ Write custom code generators! Common practice.

How to assure generated code well-formed? (Why?)

▶ Use MetaOCaml! Extends full OCaml. Widely used.
How to reconcile generality with performance?

- **Write custom code generators!** Common practice.

How to assure generated code well-formed? (Why?)

- **Use MetaOCaml!** Extends full OCaml. Widely used.

---

**The Design and Implementation of FFTW3**

MATTEO FRIGO AND STEVEN G. JOHNSON

Invited Paper

---

**BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture**

Andrew Begel, Steven McCann, Susan L. Graham
University of California, Berkeley
{abegel, mccann, graham}@cs.berkeley.edu

---

**Abstract**

A packet filter is a programmable selection criterion for classifying or selecting packets from a packet stream in a generic, reusable fashion. Previous work on packet filters falls roughly into two categories, namely those efforts that investigate flexible and extensible filter abstractions but sacrifice performance, and those that focus on low-level, optimized filtering representations but sacrifice flexibility. In routers (e.g., for real-time services or layer-four switching) [14, 20], firewall filtering, and intrusion detection [10].

The earliest representations for packet filters were based on an imperative execution model. In this form, a packet filter is represented as a sequence of instructions that conform to some abstract virtual machine, much as modern Java byte codes represent programs that can be executed on a Java virtual machine. Mogul et al.'s original packet filter (known as the CMU/Stanford...
How to reconcile generality with performance?

- **Write custom code generators!** Common practice.

How to assure generated code well-formed? (Why?)

- **Use MetaOCaml!** Extends full OCaml. Widely used.

Accomplishments and Research Challenges in Meta-programming

Invited Paper

Tim Sheard

Pacific Software Research Center

Standard ML as a Meta-Programming Language

Samuel Kamin

Computer Science Dept.

University of Illinois

Urbana, Illinois

s-kamin@uiuc.edu

September 30, 1996
Q&A

How to reconcile generality with performance?

▶ **Write custom code generators!** Common practice.

How to assure generated code well-formed? *(Why?)*

▶ **Use MetaOCaml!** Extends full OCaml. Widely used.
Q&A

How to reconcile generality with performance?

- **Write custom code generators!** Common practice.

How to assure generated code well-formed? *(Why?)*

- **Use MetaOCaml!** Extends full OCaml. Widely used.

<table>
<thead>
<tr>
<th>MetaOCaml</th>
<th>BER MetaOCaml</th>
</tr>
</thead>
<tbody>
<tr>
<td>–January 2006</td>
<td>March 2010—?</td>
</tr>
<tr>
<td>OCaml 3.09.1</td>
<td>OCaml 3.11.2</td>
</tr>
<tr>
<td>bytecode + native</td>
<td>bytecode</td>
</tr>
</tbody>
</table>
Q&A

How to reconcile generality with performance?
  ▶ Write custom code generators! Common practice.

How to assure generated code well-formed?
  ▶ Use MetaOCaml! Extends full OCaml. Widely used.

How to type-check generated code?
  ▶ Preserve type environments
  ▶ Rename shadowed identifiers?
  ▶ Follow explicit substitutions?

How to maintain type soundness with side effects?
  ▶ Later binders delimit earlier effects
  ▶ Regions of generated names?
  ▶ Earlier effects prevent later generalization?

How to implement code generation as syntactic sugar?
  ▶ camlp4/5 quotations
  ▶ Represent let-polymorphism by higher polymorphism?
Crash course

<table>
<thead>
<tr>
<th></th>
<th>MetaOCaml</th>
<th>is not quite like</th>
<th>Lisp</th>
</tr>
</thead>
<tbody>
<tr>
<td>bracket</td>
<td>(&lt;x + y&gt;).</td>
<td>quasiquote</td>
<td>‘(+ x y)</td>
</tr>
<tr>
<td>escape</td>
<td>.~body</td>
<td>unquote</td>
<td>,body</td>
</tr>
<tr>
<td>run</td>
<td>.!code</td>
<td>eval</td>
<td>(eval code)</td>
</tr>
<tr>
<td>persist</td>
<td>r</td>
<td></td>
<td>’,r</td>
</tr>
</tbody>
</table>
### Crash course

<table>
<thead>
<tr>
<th>MetaOCaml</th>
<th>is not quite like</th>
<th>Lisp</th>
</tr>
</thead>
<tbody>
<tr>
<td>bracket</td>
<td><code>&lt;x + y&gt;</code></td>
<td>quasiquote</td>
</tr>
<tr>
<td>escape</td>
<td><code>~body</code></td>
<td>unquote</td>
</tr>
<tr>
<td>run</td>
<td><code>!code</code></td>
<td>eval</td>
</tr>
<tr>
<td>persist</td>
<td><code>r</code></td>
<td></td>
</tr>
</tbody>
</table>

```
<fun x -> ~(let body = `<x>`.
  in `<fun x -> ~(body)`.).

'(lambda (x) ,(let ((body 'x))
  '(lambda (x) ,body)))

'(lambda (x) (lambda (x) x))
```

Implicit binding context …
## Crash course

<table>
<thead>
<tr>
<th></th>
<th>MetaOCaml</th>
<th>is not quite like</th>
<th>Lisp</th>
</tr>
</thead>
<tbody>
<tr>
<td>bracket</td>
<td><code>&lt;x + y&gt;</code></td>
<td>quasiquote</td>
<td><code>( + x y)</code></td>
</tr>
<tr>
<td>escape</td>
<td><code>.~body</code></td>
<td>unquote</td>
<td><code>,body</code></td>
</tr>
<tr>
<td>run</td>
<td><code>.!code</code></td>
<td>eval</td>
<td><code>(eval code)</code></td>
</tr>
<tr>
<td>persist</td>
<td><code>r</code></td>
<td></td>
<td><code>',r</code></td>
</tr>
</tbody>
</table>

```
<fun x -> `.~(let body = `<x>`.  
            in `.<fun x -> `.~body>.`)>.
<fun x_1 -> `.~(let body = `<x_1>`.  
             in `.<fun x -> `.~body>.`)>.
<fun x_1 -> `.~<fun x_2 -> `.~.<x_1>.>`.
<fun x_1 -> `.~<fun x_2 -> x_1>`.
<fun x_1 -> fun x_2 -> x_1>.
```
Crash course

<table>
<thead>
<tr>
<th>MetaOCaml</th>
<th>is not quite like</th>
<th>Lisp</th>
</tr>
</thead>
<tbody>
<tr>
<td>bracket</td>
<td>.&lt;x + y&gt;.</td>
<td>quasiquote</td>
</tr>
<tr>
<td>escape</td>
<td>.~body</td>
<td>unquote</td>
</tr>
<tr>
<td>run</td>
<td>.!code</td>
<td>eval</td>
</tr>
<tr>
<td>persist</td>
<td>r</td>
<td></td>
</tr>
</tbody>
</table>

```ocaml
<fun x -> .~(let body = .<x>.
    in .<fun x -> .~body>.)>.
<fun x_1 -> .~(let body = .<x_1>.
    in .<fun x -> .~body>.)>.
<fun x_1 -> .~.<fun x -> .~.<x_1>..>..>.
<fun x_1 -> .~.<fun x_2 -> .~.<x_1>..>..>.
<fun x_1 -> .~.<fun x_2 -> x_1>..>.
<fun x_1 -> fun x_2 -> x_1>.
```
Q&A

How to reconcile generality with performance?
- Write custom code generators! Common practice.

How to assure generated code well-formed?
- Use MetaOCaml! Extends full OCaml. Widely used.

How to type-check generated code?
- Preserve type environments
- Rename shadowed identifiers?
- Follow explicit substitutions?

How to maintain type soundness with side effects?
- Later binders delimit earlier effects
- Regions of generated names?
- Earlier effects prevent later generalization?

How to implement code generation as syntactic sugar?
- camlp4/5 quotations
- Represent let-polymorphism by higher polymorphism?
Passes

- Source text
  - parse
  - AST
    - type-check
    - Typed IR
      - compile
      - Executable
        - run

Generated code never goes wrong either each node annotated with type environment
Passes

Source text → parse → AST → type-check → Typed IR → compile → Executable → run

generated code never goes wrong either
Passes

Source text → parse → AST → type-check → Typed IR → compile → Executable → run

generated code never goes wrong either
Passes

- Source text
  - parse
  - AST
    - type-check
      - Typed IR
        - compile
          - Executable
            - run

- Generated code never goes wrong either
- Each node annotated with type environment

- Typed IR
  - compile
    - Executable
      - run
Passes

Source text

parse

AST

type-check

Typed IR

compile

Executable

run

Generated code never goes wrong either

Each node annotated with type environment
Preserving type environments

```ocaml
# type foo = Foo
let x = .<Foo>.

type bar = Foo | Bar
let y = .<Foo>.

let z = .<(.~x, .~y)>.

val z : ('a, foo * bar) code = .<((Foo), (Foo))>.
```
Preserving type environments

```ocaml
# type foo = Foo
let x = .<Foo>.
type bar = Foo | Bar
let y = .<Foo>.
let z = .<(~x, ~y)>.

val z : ('a, foo * bar) code = .<((Foo), (Foo))>.
```

Currently, `.<Foo>` means to make an AST node `Foo` and stash the type environment here in it.
Preserving type environments

```ocaml
# type foo = Foo
let x = <Foo>.

type bar = Foo | Bar
let y = <Foo>.

let z = <(~x, ~y)>.

val z : ('a, foo * bar) code = <((Foo), (Foo))>.
```

Perhaps simpler:

```ocaml
type foo = Foo1

type bar = Foo2 | Bar
```

Need guidance from a calculus with explicit substitutions!
How to reconcile generality with performance?
- Write custom code generators! Common practice.

How to assure generated code well-formed?
- Use MetaOCaml! Extends full OCaml. Widely used.

How to type-check generated code?
- Preserve type environments
- Rename shadowed identifiers?
- Follow explicit substitutions?

How to maintain type soundness with side effects?
- Later binders delimit earlier effects
- Regions of generated names?
- Earlier effects prevent later generalization?

How to implement code generation as syntactic sugar?
- `camlp4/5 quotations`
- Represent `let`-polymorphism by higher polymorphism?
Scope extrusion

Pure staging works great, especially with polymorphism. But effects are oh so useful.
Scope extrusion

Pure staging works great, especially with polymorphism. But effects are oh so useful.

```ocaml
# let code =
    let r = ref <1>. in
  let _ = <fun x -> ~(r := <x>.; <()>.)> in
  !r ;;

val code : ('a, int) code = <x_1>.
```

Unbound value x_1

Exception: Trx.TypeCheckingError.

To restore soundness:

later binders delimit earlier effects

To express even more:

regions of generated names?
Scope extrusion

Pure staging works great, especially with polymorphism. But effects are oh so useful.

```ocaml
# let code =
  let r = ref .<1>. in
  let _ = .<fun x -> ~(r := .<x>.; .<()>.)> in
  !r ;;

val code : ('a, int) code = .<x_1>.

# .!code ;;

Unbound value x_1
Exception: Trx.TypeCheckingError.
```

To restore soundness: later binders delimit earlier effects
To express even more: regions of generated names?
Imperative polymorphism redux

# let f () = ref []
Imperative polymorphism redux

```ocaml
# let f () = ref []
in f () := [1];
   "hello" :: !(f ()) ;;

- : string list = ["hello"]
```
Imperative polymorphism redux

```ocaml
# let c = let f () = ref []

val c : ('a, string list) code = let f_2 () = ref []
in f_2 () := [1];
"hello" :: !(f_2 ()) .

# .!c ;;
- : string list = ["hello"]
```
Imperative polymorphism redux

```
# let c = <let f () = ref []
    in f () := [1];
    "hello" :: !(f ())>. ;;

val c : ('a, string list) code =
  <let f_2 () = ref []
    in f_2 () := [1];
    "hello" :: !(f_2 ())>.

# !c ;;
- : string list = ["hello"]
```
Imperative polymorphism redux

```ocaml
# let c = .<let f () = .~(.<ref []>).

val c : ('a, string list) code = .<let f_2 () = ref []
in f_2 () := [1];
"hello" :: !(f_2 ()).>

# .!c ;;
- : string list = ["hello"]
```
Imperative polymorphism redux

```ocaml
# let c = .<let f () = .~(.<ref []>.)
  in f () := [1];
  "hello" :: !(f ())>. ;;

val c : ('a, string list) code =
  .<let f_2 () = ref []
  in f_2 () := [1];
  "hello" :: !(f_2 ())>.

# .!c ;;

- : string list = ["hello"]
```
Imperative polymorphism redux

# let c = .<let f () = .~(let r = ref [] in .<r>).}
Imperative polymorphism redux

```ocaml
# let c = .<let f() = .~(let r = ref [] in .<r>).
    in f() := [1];
    "hello" :: !(f())>. ;;

val c : ('a, string list) code =
  .<let f_2() = (* cross-stage persistent value
     (as id: r) *)
     in f_2() := [1];
     "hello" :: !(f_2())>. 
```

To restore soundness:
earlier effects prevent later generalization?
Imperative polymorphism redux

```ocaml
# let c = 
  (let r = ref [] in .<r>.)
  in f () := [1];
  "hello" :: !(f ())>. ;;

val c : ('a, string list) code = 
  .<let f_2 () = (* cross-stage persistent value 
    (as id: r) *)
    in f_2 () := [1];
    "hello" :: !(f_2 ())>. 

# .!c ;;

Segmentation fault

To restore soundness:
earlier effects prevent later generalization?
```
Q&A

How to reconcile generality with performance?
  ▶ Write custom code generators! Common practice.

How to assure generated code well-formed?
  ▶ Use MetaOCaml! Extends full OCaml. Widely used.

How to type-check generated code?
  ▶ Preserve type environments
  ▶ Rename shadowed identifiers?
  ▶ Follow explicit substitutions?

How to maintain type soundness with side effects?
  ▶ Later binders delimit earlier effects
  ▶ Regions of generated names?
  ▶ Earlier effects prevent later generalization?

How to implement code generation as syntactic sugar?
  ▶ camlp4/5 quotations
  ▶ Represent let-polymorphism by higher polymorphism?
Code generation as syntactic sugar

camlp4/5 quotations? CUFP BoF, tutorial.

\.let id = fun x -> x in id 1\..
Let_ (Lam (fun x -> x)) (fun id -> App id (Lit 1))

Seems straightforward, but how to represent polymorphic let?

\[\frac{e : \tau}{e : \forall \alpha. \tau}\]
\[\text{Gen} : \forall \tau: \ast \rightarrow \ast. (\forall \alpha. \alpha \tau \text{ code}) \rightarrow (\forall \alpha. \alpha \tau \text{ code})\]

\[\frac{e : \forall \alpha. \tau}{e : \tau[\sigma/\alpha]}\]
\[\text{Spec} : \forall \tau: \ast \rightarrow \ast. (\forall \alpha. \alpha \tau \text{ code}) \rightarrow (\forall \alpha. \alpha \tau \text{ code})\]

Need higher-rank, higher-kind polymorphism?
Don’t generate code that uses polymorphism? ‘Metacircular let’

let id = Lam (fun x -> x) in App id id
Code generation as syntactic sugar

camlp4/5 quotations? CUFP BoF, tutorial.

$.\langle \text{let } id = \text{fun } x \to x \text{ in id id} \rangle$. 
Let_ (Lam (fun x -> x)) (fun id -> App id id)

Seems straightforward, but how to represent polymorphic let?

\[
\begin{align*}
\frac{e : \tau}{e : \forall \alpha. \tau} & \quad \text{Gen} : \forall \tau : \star \to \star. (\forall \alpha. \alpha \tau \text{ code}) \to (\forall \alpha. \alpha \tau \text{ code}) \\
\frac{e : \forall \alpha. \tau}{e : \tau[\sigma/\alpha]} & \quad \text{Spec} : \forall \tau : \star \to \star. (\forall \alpha. \alpha \tau \text{ code}) \to (\forall \alpha. \alpha \tau \text{ code})
\end{align*}
\]

Need higher-rank, higher-kind polymorphism?
Don’t generate code that uses polymorphism? ‘Metacircular let’

let id = Lam (fun x -> x) in App id id
Code generation as syntactic sugar

camlp4/5 quotations? CUFP BoF, tutorial.

`.let id = fun x -> x in id id.`
Let_ (Lam (fun x -> x)) (fun id -> App id id)

Seems straightforward, but how to represent polymorphic let?

\[
\begin{align*}
  e : \tau & \quad \text{Gen : } \forall \tau : \star \rightarrow \star. (\forall \alpha. \alpha \tau \text{ code}) \rightarrow (\forall \alpha. \alpha \tau) \text{ code} \\
  e : \forall \alpha. \tau & \quad \text{Spec : } \forall \tau : \star \rightarrow \star. (\forall \alpha. \alpha \tau \text{ code}) \rightarrow (\forall \alpha. \alpha \tau \text{ code})
\end{align*}
\]

Need higher-rank, higher-kind polymorphism?
Don’t generate code that uses polymorphism? ‘Metacircular let’

let id = Lam (fun x -> x) in App id id
Code generation as syntactic sugar

camlp4/5 quotations? CUFP BoF, tutorial.

.\(\text{let id = fun x -> x in id id}.\)
Let_ (Lam (fun x -> x)) (fun id -> App id id)

Seems straightforward, but how to represent polymorphic let?

\[
\begin{array}{c}
\text{e} : \tau \\
\hline
\text{Gen} : \forall \tau : \star \rightarrow \star. (\forall \alpha . \alpha \tau \text{ code}) \rightarrow (\forall \alpha . \alpha \tau \text{ code}) \\
\end{array}
\]

\[
\begin{array}{c}
\text{e} : \forall \alpha . \tau \\
\hline
\text{Spec} : \forall \tau : \star \rightarrow \star. (\forall \alpha . \alpha \tau \text{ code}) \rightarrow (\forall \alpha . \alpha \tau \text{ code}) \\
\end{array}
\]

Need higher-rank, higher-kind polymorphism?
Don’t generate code that uses polymorphism? ‘Metacircular let’

\[
\text{let id = Lam (fun x -> x) in App id id}
\]
How to type-check generated code?
How to maintain type soundness with side effects?
How to implement code generation as syntactic sugar?