

Compiling Embedded Programs to Byte Code

Morten Rhiger

BRICS
University of Aarhus

PADL'02

Plan

- Run-time code generation: Motivation and issues
- Our work: Run-time code generation for OCaml
 1. System description
 2. Examples and applications
 3. Foundation
- Conclusions

Plan

- Run-time code generation: Motivation and issues
- Our work: Run-time code generation for OCaml
 1. System description
 2. Examples and applications
 3. Foundation
- Conclusions

Run-time code generation: Motivation

Exploit *run-time* invariants to specialize program parts

Run-time code generation: Application

Embedded languages

- Provide domain-specific operations:

if $x \in \mathcal{L}(\{a, \dots, z\}^*)$ then ...

↓

if `match("[a-z]*", x)` then ...

- Use existing compiler or interpreter for domain-independent infrastructure:

`if` $x \in \mathcal{L}(\{a, \dots, z\}^*)$ `then ...`

Run-time code generation: Issues of style

- Implicit
Fabius (Leone & Lee)
- In-between (partial evaluation)
Tempo (Consel et al), PGG (Sperber & Thiemann), TDPE (Balat & Danvy), DyC (Chambers et al)
- Explicit (generating extensions)
Lisp (quasiquote + eval), MetaML (Taha & Sheard),
MetaOCaml (Taha et al), 'C (Engler et al)

Run-time code generation: Issues of design and implementation

- Syntax
- Type policy
(λ° , λ^\square , MetaML, ...)
- Semantics
(Ideally, run-time code generation is a “no-op”)
- Implementation

Plan

- Run-time code generation: Motivation and issues
- Our work: Run-time code generation for OCaml
 1. System description
 2. Examples and applications
 3. Foundation
- Conclusions

Our goal

- Provide RTCG for an existing (higher-order) language and an existing compiler
(OCaml)
- Apply RTCG to existing non-trivial applications
(MetaPRL)

Our design

- Syntax: Higher-order abstract syntax
- Type policy: Phantom types
- Implementation: Church encoding

Result:

1. A library of byte-code combinators
(to construct specialized programs)
2. An `eval` function
(to run specialized programs)

Starting point

- First-order datatype of source programs

```
datatype term = VAR of string
              | LAM of string * term
              | APP of term * term
```

Starting point

- First-order datatype of source programs

```
datatype term = VAR of string
              | LAM of string * term
              | APP of term * term
```

- Recursive-descent compiler

```
fun C(VAR x)      ρ = ...ρ(x)...
    | C(LAM(x, e)) ρ = ...C e ρ[x↦ℓ]...
    | C(APP(f, a)) ρ = ...C f ρ ...C a ρ...
```

Starting point

- First-order datatype of source programs

```
datatype term = VAR of string
              | LAM of string * term
              | APP of term * term
```

- Recursive-descent compiler

```
fun C(VAR x)      ρ = ...ρ(x)...
    | C(LAM(x, e)) ρ = ...C e ρ[x ↦ ℓ]...
    | C(APP(f, a)) ρ = ...C f ρ ...C a ρ...
```

- Inferred type:

```
C : term → env → bytecode
```

Higher-order abstract syntax

- Object-language bindings are represented by meta-language bindings:

```
fun app f a = APP(f, a)
```

```
fun lam f =  
  let val x = gensym()  
  in LAM(x, f (VAR x))  
  end
```

- Inferred types:

```
app : term → (term → term)
```

```
lam : (term → term) → term
```

Higher-order abstract syntax: Example

- Object term:

`fn f ⇒ fn x ⇒ f(x)`

- First-order encoding:

`LAM ("f", LAM ("x", APP (VAR "f", VAR "x")))`

- Higher-order encoding:

`lam (fn f ⇒ lam (fn x ⇒ app f x))`

Pfenning & Elliott (PLDI'88)

Phantom types

- Object-language types are represented by meta-language types

- New type synonym:

`type α exp = term`

- Enforced types:

`app : ($\alpha \rightarrow \beta$) exp \rightarrow (α exp \rightarrow β exp)`

`lam : (α exp \rightarrow β exp) \rightarrow ($\alpha \rightarrow \beta$) exp`

Peyton Jones et al (Software Reuse'98), Leijen & Meijer (USENIX'99),
Finne et al (ICFP'99), Elliott et al (SAIG'00)

Church encoding

- Higher-order representation of object-language terms:

```
fun C(VAR x)      ρ = ...ρ(x)...  
  | C(LAM(x, e)) ρ = ...C e ρ[x↦ℓ]...  
  | C(APP(f, a)) ρ = ...C f ρ ...C a ρ...
```

↓

```
fun VAR x      ρ = ...ρ(x)...  
fun LAM (x, e) ρ = ...e ρ[x↦ℓ]...  
fun APP (f, a) ρ = ...f ρ...a ρ...
```

Church encoding + higher-order abstract syntax

- Higher-order abstract syntax:

```
fun mkapp f a = APP(f, a)
```

```
fun mklam f =  
  let val x = gensym()  
  in LAM(x, f (VAR x))  
  end
```

Church encoding + phantom types

- New type synonym:

```
type 'a bcc = env → bytecode
```

- Enforced types:

```
mkapp : (α → β) bcc → (α bcc → β bcc)
```

```
mklam : (α bcc → β bcc) → (α → β) bcc
```

An eval function

- Implementation:

```
fun eval e = execute (e  $\emptyset$ )
```

- Enforced type:

```
eval :  $\alpha$  bcc  $\rightarrow$   $\alpha$ 
```

- Employs OCaml's virtual machine:

```
execute : bytecode  $\rightarrow$   $\alpha$ 
```

Plan

- Run-time code generation: Motivation and issues
- Our work: Run-time code generation for OCaml
 1. System description
 2. Examples and applications
 3. Foundation
- Conclusions

Example

- Original program (compares the lengths of two strings):

```
fun equal [] ys = (ys = [])  
  | equal (_::xs) ys =  
    if (ys = []) then  
      false  
    else  
      equal xs (tail ys)
```

Example

- Original program (compares the lengths of two strings):

```
fun equal [] ys = (ys = [])  
  | equal (_::xs) ys =  
    if (ys = []) then  
      false  
    else  
      equal xs (tail ys)
```

Example (continued)

- Generating extension:

```
fun mkequal' []      ys = mkisnil ys
  | mkequal' (_::xs) ys =
    mkif (mkisnil ys)
        mkfalse
        (mkequal' xs (mktail ys))
```

```
fun mkequal xs = mklam (fn ys => mkequal' xs ys)
```

Example (continued)

- Inferred types:

```
val mkequal' :  $\alpha$  list  $\rightarrow$  ( $\beta$  list bcc  $\rightarrow$  bool bcc)
```

```
val mkequal :  $\alpha$  list  $\rightarrow$  ( $\beta$  list  $\rightarrow$  bool) bcc
```

- Hence, for example

```
eval (mkequal [1,2,3]) :  $\beta$  list  $\rightarrow$  bool
```

Application: MetaPRL's term rewriter

MetaPRL:

- A higher-order logical framework
- Supports reasoning about logics
- Implements NuPrl, Coq, Isabelle, ...
- Ensemble
- More than 200,000 lines of OCaml code

Run-time code generation for MetaPRL

The term rewriter

- Interprets inference rules (byte-code language)
- Roughly 500 lines of OCaml code
- Applied extensively (e.g., 2 million times for proving the pigeon-hole principle for 4 holes)

Our goal: To specialize the term rewriter with respect to the inference rules

Benchmarks

Unfortunately, speedups are lower than hoped for:

- Pigeon-hole (2 holes): 10% speedup
- Pigeon-hole (4 holes): 36% speedup

Plan

- Run-time code generation: Motivation and issues
- Our work: Run-time code generation for OCaml
 1. System description
 2. Examples and applications
 3. Foundation
- Conclusions

Foundation for phantom types

The meta-language terms represent object-language terms

For example,

```
mklam (fn f => mkapp f (mkint 1))  
      : ((int → α) → α) exp
```

represents

```
fn f => f 1   : (int → α) → α
```

Foundation for phantom types

- Soundness:

Are all **ill-typed** object terms **rejected**?

- Completeness:

Are all **well-typed** object terms **accepted**?

Foundation for phantom types

- Enabling theory for embedded languages
- Not presented in the proceedings (see my dissertation)

Soundness for an idealized ML

For any meta-language program E ,

*if E has type t **exp** in the meta language
and E evaluates to e*

then

$e = \llbracket P \rrbracket$ for some object-language program P with type t

Proof: Structural induction on E (using a Kripke logical relation)

Soundness for Standard ML

- Higher-order abstract syntax is inadequate
(No case analysis)
- Side-effects may break soundness

(Higher-order abstract syntax is also inadequate for Haskell)

Completeness for Standard ML

For any object language program P ,

if P has type t in the object language

then

$\lceil P \rceil$ has type t exp in the meta language

Proof: Structural induction on P

Plan

- Run-time code generation: Motivation and issues
- Our work: Run-time code generation for OCaml
 1. System description
 2. Examples and applications
 3. Foundation
- Conclusions

Conclusions

Our contributions

- Run-time code generation in a library (OCaml)
- Large application: MetaPRL's term rewriter (Ensemble)
- Formal justification of type safety

Phantom types: Erroneous expression

- In ML:

```
let val err = ref (mkint 1)
in  mklam (fn x => (err := x; x));
    err
end
```

Phantom types: Erroneous expression

- In ML and Haskell:

```
mklam (fn x => ... eval x ...)
```