Run-time Code Generation and ML

Survey

January 7, 2000

1. RTCG and Staging
2. The Fabius compiler
3. The Mini-ML$_e^{\Box}$ formalism
4. MetaML
5. ML$^{\Box}$

RTCG / Staging Transformations

RTCG extends compile-time optimizations to run-time by means of fresh code emissions.

Compile-time analyses are conservative,
—
(1) they collect informations about what *may* happen at run-time and
(2) even further compromises in precision are to be done to reduce their complexity
—
RTCG is a solution to many uncomputabilities since it allows code optimizations that take advantage of values computed at run-time.

Deferring parts of the compilation process to run-time leads to greater opportunities for optimizations, yet the cost of performing such a task must be repaid by improved performances.

Staging transformations are optimizations we may divide into two classes.
—
(1) *Frequency reduction* which puts computations into contexts in which they will be performed less frequently.
(2) *Precomputation* which puts computations into contexts in which efficiency is less critical.
—
Staging transformations intend to enable work to be done as early as possible and as infrequently as possible. They give rise to a *staging* of the computation process.

Staging Analyses / Partial Evaluation

Staging is the temporal separation of the computations of a program.

It arises naturally in:
□ Software organiztion:
- Compile-time + Run-time for the compiled language.
- Generation + Compilation + Execution for generation tools.
- Binding-time analysis + Specialization in partial evaluation.

□ Code:
- Currying in functional programming languages.
- Nested loops in imperative languages.
- Macro expanders.

Staging can be done:
□ Semi-automatically:
- Partial evaluation.

□ Manually through notations:
- Lisp's antiquote.

But:
1. It takes ingenuity to write programs that will carry out all the computations possible at an early stage when given early inputs.
2. It is desirable to declare the expected binding-time separation and obtain constructive feedback afterwards.

$\rightarrow$ This suggets that the binding-time separation should be expressed inside a type system and that staging analysis should be a form of type checking.

The relationship between partial evaluation and staging analyses is the intermediate data structure: a typed language that include binding-time annotations (Nielson & Nielson's two-stage notation and B-level languages).

RTCG and Staging Analysis

Staging transformations at compile-time can be exploited at run-time by means of RTCG.

Example

```
fun raise exp bases =
    map (power exp) bases
and power exp base =
    if exp = 0 then 1
    else base * power (exp - 1) base
```

Staging by hoisting yields the following code:

```
fun power exp =
    if exp = 0 then (fn base => 1)
    else let pow = power (exp - 1)
         in
            fn base => base * pow base
         end
```

An overhead has been introduced by the creation of many closures.

We want RTCG to reduce the frequency of loop-invariant without this overhead.

RTCG and Staging Analysis

Specialization transformations at compile-time can be exploited at run-time by means of RTCG.

Example

```
fun powgen exp =
    nth exp [fn base => 1,
             fn base => base,
             fn base => base * base,
             fn base => base * base * base,
             .
             .
             .
             ]
    handle Nth => power exp
```

We simply choose the power function corresponding to the exponent. But how do we decide the number of functions to be specialized?

We want RTCG to do specialization on demand.

Fabius: Optimizations

- register allocation — by graph coloring,
- constant propagation — of run-time values,
- instruction selection — `emit(add)` that can check `add`'s operands before code emission,
- hoisting — loop-invariant hoisted at the loop preheader,
- folding — evaluation of expressions,
- loop unrolling — when the control parameter is known at run-time,
- function inlining — copy of the body with substitution of the formal parameters for the actual ones,
- dead code elimination — if a function is inlined whenever it is called,
- strength reduction — for instance, we may replace the multiplication by four with an addition in the following piece of code.

```
        s <- 0
        i <- 0
L1:     if i >= n goto L2
        j <- i*4
        k <- j + a
        x <- M[k]
        s <- s + x
        i <- i + 1
        goto L1
L2:     ...
```

Fabius: Deferred Compilation

We can invoke a partial evaluator at run-time to specialize f to x: $mix([\![f]\!], x) = f_x$.

Going further, we can reduce the cost of run-time specialization by specializing at compile-time mix to f: $mix([\![mix]\!], [\![f]\!]) = mix_f$, so that mix_f now specializes f_x without the overhead of processing $[\![f]\!]$.

Deferred compilation employs compile-time specialization to reduce the cost of RTCG.

RTCG is said to be lighweight when no intermediate representation at run-time is processed.

The primary goal of FABIUS is to reduce the RTCG cost to a minimum.

The source language is a strict first-order functional language.

FABIUS generates MIPS R2000 native code.

The compilation has three major phases:
□ *Staging analysis* identifies computation stages.
□ *Resgister allocation* assigns registers to variables and intermediate values.
□ *Code generation* compiles:
• Early computations in the usual way, and
• Late computations into machine code that generates (optimized) instruction at run-time.

Fabius: Example

Matrix multiplication

We face a triply nested loop. The two outer loops select a vector from each matrix. The inner loop computes their dot product.

```
fun dotprod (v1, v2, sum) =
        if v1 = nil then sum
        else dotprod (tl v1, tl v2, sum + hd v1 * hd v2)
```

Usual compilation may generate the following code.

```
dotprod:
        beq     r1, r0, L1      ; if v1 = nil goto L1
        ld      r4, (r1)        ; x1 = hd v1
        ld      r5, (r2)        ; x2 = hd v2
        mul     r4, r4, r5      ; prod = x1 * x2
        add     r3, r3, r5      ; sum = sum + prod
        ld      r1, 4(r1)       ; v1 = tl v1
        ld      r2, 4(r2)       ; v2 = tl v2
        jmp     dotprod         ; goto dotprod
L1:
        move    r1, r3          ; result = sum
        ret                     ; return
```

Fabius: Example

The availability of the arguments differs:
- `v1` is available at an early stage,
- `v2` and `sum` are available at a late stage.

All subexpressions are subsequently annotated by the compiler.

```
fun dotprod (v1, v2, sum) =
  if v1 = nil then sum
  else dotprod (tl v1, tl v2, sum + hd v1 * hd v2)
```

□ Detecting program stages is a difficult problem. FABIUS relies on a syntactic notation similar to currying.

□ Staging analysis must also determine which stages will benefit from run-time compilation.

□ Register allocation is complicated by deferred compilation:

- The notion of *variable lifetime range* is changed: edges should be added between overlapping life-time ranges of variables from the same program stage.
- An exact interference graph cannot be built at compile-time since inlining and unrolling may occur at run-time.

□ Early computations are compiled the usual way and late computations give rise to `emit` instructions.

Fabius: Example

`emit` expands into a sequence of instructions that:

- allocates space in a dynamic code segment,
- builds the representation of an instruction, and
- writes the instruction in the allocated space.

```
dotgen:
        beq     r1, r0, L1        ; if v1 = nil goto L1
        ld      r2, (r1)          ; x1 = hd v1
        emit    ld  r3, (r1)      ; emit x2 = hd v2
        emit    mul r3, [r2], r3  ; emit prod = x1 * x2
        emit    add r2, r2, r3    ; emit sum = sum + prod
        ld      r1, 4(r1)         ; v1 = tl v1
        emit    ld r1, 4(r1)      ; emit v2 = tl v2
        jmp     dotgen            ; goto dotgen
L1:
        emit    move r1, r2       ; emit result = sum
        ret                       ; return
```

`dotgen` effectively performs:

- 'constant' propagation,
- conditional folding, and
- inlining at run-time.

```
ld      r3, (r1)        ; x2 = hd v2
mul     r3, X1, r3      ; prod = X1 * x2
add     r2, r2, r3      ; sum = sum + prod
ld      r1, 4(r1)       ; v2 = tl v2
.
.
.
ld      r3, (r1)        ; x2 = hd v2
mul     r3, Xn, r3      ; prod = Xn * x2
add     r2, r2, r3      ; sum = sum + prod
ld      r1, 4(r1)       ; v2 = tl v2
move    r1, r2          ; result = sum
```

Fabius: Results

Instructions counts

For 200×200 matrices, deferred compilation reduces the total number of instructions executed by 18% for dense data and 72% for sparse data.

The cost of run-time optimization and code generation accounts for 3% of the total number of instructions executed for 200×200 matrices.

Execution times

Deferred compilation does not yield any speedup for dense matrices and yields a speed-up of only 50% for sparse matrices.

This is due to the integer-multiply instruction of MIPS which requires several cycles to complete. The instruction pipeline is actually stalling whenever an attempt is made to access the result before it is ready.

Mini-ML$_e^\Box$: Design Principle

A stage of the computation is **a world in the Kripke semantics of modal logic.**

• That is why it seems sensible that a type system based on modal logic provides a nice framework for staging analysis.

• Moreover this gives rise to a generalization of (two-level) binding-time analyses by allowing multiple computation stages.

A feature of staged computation is the manipulation of object code or the manipulation of any intermediate representation of object code. Mini-ML$_e^\Box$ does not commit oneself to any particulial representation.

With a new type constructor, we can abstract away from the implementation and simultaneously explicitly represent operations on code and the staging of the computation.

$\rightarrow$ Mini-ML$_e^\Box$ is therefore based on a typed language — Mini-ML — to which a new type constructor $\Box$ is added, where $\Box A$ represents code of type A to be executed in a late stage.

Mini-ML$_e^{\Box}$: Syntax / Typing Rules

$$
\begin{array}{llll}
\text{Types} & A & ::= & \textsf{nat} \mid A_1 \to A_2 \mid A_1 \times A_2 \mid \Box A \\
\text{Terms} & E & ::= & x \mid \lambda x : A.E \mid E_1\ E_2 \\
 & & & \mid \textbf{fix}\ x : A.E \\
 & & & \mid \langle E_1, E_2 \rangle \mid \textbf{fst}\ E \mid \textbf{snd}\ E \\
 & & & \mid \textbf{z} \mid \textbf{s}\ E \\
 & & & \mid (\textbf{case}\ E_1\ \textbf{of}\ \textbf{z} \Rightarrow E_2 \mid \textbf{s}\ x \Rightarrow E_3) \\
 & & & \mid \textbf{box}\ E \mid \textbf{let box}\ x = E_1\ \textbf{in}\ E_2 \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

The type checker should reject programs for which the desired binding-time separation is violated — even though they may execute correctly.

A constructor **box** is introduced that allows us to manipulate any arbitrary closed expression as code.

A restriction on code substitution is enforced by introducing two contexts in the typing judgement $\Delta, \Gamma \vdash^e E : A$.
- Δ contains variables that are bound only to code during evaluation.
- Γ contains variables that are bound to arbitrary values.

The elimination construct for **box** allows us to bind a variable x of type A in Δ to code of type $\Box A$.

Mini-ML$_e^{\Box}$: Modal Fragment

Typing rules

$$\frac{x : A \text{ in } \Delta}{\Delta; \Gamma \vdash^e x : A}\text{tpe\_gvar}$$

$$\frac{\Delta; \cdot \vdash^e E : A}{\Delta; \Gamma \vdash^e \mathbf{box}\ E : \Box A}\text{tpe\_let\_box}$$

$$\frac{\Delta; \Gamma \vdash^e E_1 : \Box A \quad (\Delta, x : A); \Gamma \vdash^e E_2 : B}{\Delta; \Gamma \vdash^e \mathbf{let\ box}\ x = E_1\ \mathbf{in}\ E_2 : B}\text{tpe\_let\_box}$$

Values $V ::= \lambda x : A.E \mid \langle V_1, V_2 \rangle \mid z \mid s\ V \mid \mathbf{box}\ E$

Operational Semantics

$$\frac{}{\mathbf{box}\ E \hookrightarrow \mathbf{box}\ E}\text{ev\_box}$$

$$\frac{E_1 \hookrightarrow \mathbf{box}\ E_1' \quad [E_1'/x]E_1 \hookrightarrow V_2}{\mathbf{let\ box}\ x = E_1\ \mathbf{in}\ E_2 \hookrightarrow V_2}\text{ev\_let\_box}$$

Does Mini-ML$_e^\Box$ Capture Staging?

LEMMA 1 (SUBSTITUTION)

1. If $\Delta; \Gamma \vdash^e E_1 : A$ and $\Delta; (\Gamma, x : A) \vdash^e E_2 : B$ then $\Delta; \Gamma \vdash^e [E_1/x]E_2 : B$.
2. If $\Delta; \cdot \vdash^e E_1 : A$ and $(\Delta, x : A); \Gamma \vdash^e E_2 : B$ then $\Delta; \Gamma \vdash^e [E_1/x]E_2 : B$.

THEOREM 1 (DETERMINACY AND TYPE PRESERVATION)

1. If $E \hookrightarrow V$ then V is a value.
2. If $E \hookrightarrow V$ and $E \hookrightarrow V'$ then $V = V'$ (modulo renaming of bound variables).
3. If $E \hookrightarrow V$ and $\cdot; \cdot \vdash^e E : A$ then $\cdot; \cdot \vdash^e V : A$.

Let us uppose that $\cdot; \cdot \vdash^e E : \Box A$ and $E \hookrightarrow V$. Then V is a value of type $\Box A$ (by 1. and 3.), that is $V =$ **box** E' for a certain E'. The result of evaluating a code expression therefore consists of residual code to be executed in the next stage.

The parts of the original program not enclosed by any **box** constructor are therefore static and should be computer in the current stage.

There is no way to evaluate under a **box** constructor. The only way to evaluate the body of a **box** constructor is to use the variable bound by a **let box** elimination construct to indicate where the delayed computation should be performed.

Mini-ML$_e^{\Box}$: Example

Evaluation of code

$$eval \equiv (\lambda x : \Box A.\textbf{let box } y = x \textbf{ in } y) : (\Box A) \rightarrow A$$

The Function Power

The type nat $\rightarrow \Box$(nat $\rightarrow$ nat) expresses that $power$ evaluates everything that depends on the first argument of type nat and returns residual code of type $\Box$(nat $\rightarrow$ nat).

$$\begin{array}{l} power \equiv \textbf{fix } p : \mathsf{nat} \rightarrow \Box(\mathsf{nat} \rightarrow \mathsf{nat}). \\ \lambda n : \mathsf{nat}.\textbf{case } n \textbf{ of} \\ \quad \textbf{z} \Rightarrow \textbf{box } (\lambda x : \mathsf{nat}.s\ \textbf{z}) \\ \quad \mid \textbf{s } m \Rightarrow \textbf{let box } q = p\ m \textbf{ in} \\ \qquad \textbf{box } (\lambda x : \mathsf{nat}.times\ (x\ (q\ x)) \end{array}$$

Indeed, we calculate with our operational semantics.

$$\begin{array}{rcl} power\ \textbf{z} & \hookrightarrow & \textbf{box } (\lambda x : \mathsf{nat}.s\ \textbf{z}) \\ power(\textbf{s z}) & \hookrightarrow & \textbf{box } (\lambda x : \mathsf{nat}.times\ x\ ((\lambda x : \mathsf{nat}.\textbf{s z})x)) \\ power\ (\textbf{s}\ (\ \textbf{s z})) & \hookrightarrow & \textbf{box } (\lambda x : \mathsf{nat}.times\ x \\ & & ((\lambda x : \mathsf{nat}.times\ x\ ((\lambda x : \mathsf{nat}.\textbf{s z})x))x) \end{array}$$

MetaML: Introduction

MetaML is a statically-typed multi-stage programming language, extension of the polymorphically-typed call-by-value λ-calculus.

Multi-stage languages should be real programming languages and should support staging without changing the programming style.

MetaML provides us with:

- Four staging annotations
— the formal proof of safety is ongoing work.
- A type system
— ensuring the well-formedness of multi-stage programs.
- *Cross-stage persitence*
— Variables of any stage are available in all future stages.

MetaML introduces four distinct staging annotations:

- Meta-Brackets $\langle \_ \rangle$ delay a computation.
$40 + 2 \rightarrow \langle 40 + 2 \rangle$.
NB: A type constructor $\langle \_, \_ \rangle$ is used to distinguish type.
$\text{int} \rightarrow \langle \text{int}, {}'a \rangle$
- Escape ‘_ occurs inside meta-brackets.
$\text{let val } a = \langle 1 + 4 \rangle \text{ in } \langle 72 + {}`a \rangle \text{ end} \rightarrow \langle 72 + (1 + 4) \rangle$
- Run *run* _ runs a code argument.
$\text{let val } a = \langle 50 - 10 \rangle \text{ in } 2 + (\text{run } a) \text{ end} \rightarrow 42$
- Lift (*lift* _) converts a ground value into code.
$\langle 1 + {}`(\text{lift } 2 + 3) \rangle \rightarrow \langle 1 + 5 \rangle$
NB: Meta-brackets turn a syntactic expression into code, lift also computes it.

MetaML: Example

```
(* member : int -> int list -> bool *)
fun member v l =
if (null l)
        then false
        else v = (hd l)
                then true
                else member v (tl l);
```

Staging is performed manually: we first determine the target type and annotate the program afterwards by going inwards until all subexpressions are treated.

```
(* member : <int, 'a> -> int list -> <bool, 'a> *)
fun member v l =
if (null l)
        then <false>
        else <if `v = `(lift hd l)
                then true
                else `(member v (tl l))>;
```

Those annotations give rise to the following transformation.

```
(* <fn x => `(member <x> [1,2,3])> *)
<fn d1 =>
if d1 %=1
        then true
        else if d1 %=2
                then true
                else if d1 %=3
                        then true
                        else false>
```

MetaML: Design

Definition 1 (Cross-stage Persistence)
A variable i bound in stage n will be available in stages n, $n+1$ and all future stages.

We can see a code value as containing an environment which binds the free variables in it. In

$$\text{let val } a = 1 + 4 \text{ in } \langle 72 + a \rangle \text{ end} \rightarrow \langle 72 + \%a \rangle$$

5 is bound to a during the first stage and $\%a$ is a new constant. If evaluated later, $\%a$ will return 5 no matter what the new binding of $\%a$ is.

It turns out that a hygienic treatement of free variables is difficult to implement in this way.

Definition 2 (Cross-stage Safety)
An input first available at stage m cannot be used at a stage n is $m > n$.

This problem is introduced by the escape notation. E.g.,

$$\text{fn } a \Rightarrow \langle \text{fn } b \Rightarrow \text{`}(a + b) \rangle$$

The type system ensures that the program will not go wrong, where going wrong now includes the violation of cross-stage safety conditions.

MetaML: Static Semantics

A set of inference rules determines before the first stage well-formedness of expressions together with their types.

The type inference algorithm assigns the same context name to expressions that share the same context.

$\Sigma\Delta \vdash^n x : \tau_1, \tau_2$ **means that under context the stack Σ, the type environment Δ, the term x has type τ_1 at level n and may execute in the context with name τ_2.**

In the above judgement, Δ maps variables to types, levels and contexts, and a level n is the level of the abstraction in which the variable is bound.

$$\textbf{Abs n} : \frac{\Sigma\Delta, x \mapsto (\tau_1, \alpha)^n \vdash^n e : \tau_2, \alpha}{\Sigma\Delta \vdash^n \lambda x^{\tau_1}.e : \tau_1 \to \tau_2, \alpha}$$

n is incremented for an expression inside meta-brackets.

$$\textbf{Br n} : \frac{(\alpha_2, \Sigma)\Delta \vdash^{n+1} e : \tau, \alpha_1}{\Sigma\Delta \vdash^n \langle e \rangle : \langle \tau \rangle^{\alpha_1}, \alpha_2}$$

n is decremented for an expression inside an escape.

$$\textbf{Esc n+1} : \frac{\Sigma\Delta \vdash^n e : \langle \tau \rangle^{\alpha_1}, \alpha_2}{(\alpha_2; \Sigma)\Delta \vdash^{n+1} \text{`}e : \tau, \alpha_1}$$

MetaML: Cross-stage Safety

- A variable cannot be used in a stage before it is available. The following rules enforce cross-persistence safety check.

$$\textbf{Var=0}: \frac{\Delta x = (\tau, \alpha_1)^0}{\Sigma\Delta \vdash^n x : \tau, \alpha_2}$$

$$\textbf{Var} \leq \textbf{n}: \frac{\Delta x = (\tau, \alpha)^i \quad i \neq 0 \wedge i \leq n}{\Sigma\Delta \vdash^n x : \tau, \alpha}$$

- Run and escape cannot be given expressions that are not code. This is enforce by the following rules.

$$\textbf{Run n}: \frac{\Sigma\Delta \vdash^n e : \langle\tau\rangle^{\alpha_2}, \alpha_1}{\Sigma\Delta \vdash^n \textbf{run } e : \tau, \alpha_1} \quad \alpha_2 \notin FV(\Delta, \tau)$$

$$\textbf{Esc n+1}: \frac{\Sigma\Delta \vdash^n e : \langle\tau\rangle^{\alpha_1}, \alpha_2}{(\alpha_2; \Sigma)\Delta \vdash^{n+1} \text{`}e : \tau, \alpha_1}$$

- Run cannot be given an expression which is code with free variables. Only code whose context is completely unconstrainted may be run. Hence the above free variable check.

MetaML: Cross-stage Persistence

In all previously studied statistically typed multi-staged languages, only a monolithic type rule was used for variables.

$$\textbf{Var (Monolithic):}\frac{(\Delta x) = \tau^m}{\Delta \vdash^n x : \tau} \text{ when } m = n$$

Cross-stage persistence allows a more general condition $m \leq n$.

□ The expression

```
val shift_like = fn x => <x>
```

is accepted (inside the meta-brackets, $n = 1$ and $\Delta x = \alpha^0$).

□ The expression

```
fn a => <fn b => ~(a+b)>
```

is rejected because (inside the escape, $n = 0$ and $\Gamma b = \alpha^1$).

$\mathsf{ML}^{\Box}$: Introduction

$\mathsf{ML}^{\Box}$ extends ML with the modal operators from Mini-$\mathsf{ML}_e^{\Box}$.

We target an abstract machine extended with facilities for RTCG.

Whereas partial evaluators use a semi-automatic binding-time analysis to determine staging, $\mathsf{ML}^{\Box}$ is a programming language that supports explicit description of stages and that supports checking of their consistency (cf. Mini-$\mathsf{ML}_e^{\Box}$). The automatic analysis is therefore replaced by an ML-like type checker, where a staging error becomes a type error.

$\mathsf{ML}^{\Box}$ allows a user to specify through types the staging that he wants, and this information is exploited by the compiler to perform RTCG.

$\mathsf{ML}^{\Box}$ also supports in a natural way multiple stages of run-time specialization (cf. Mini-$\mathsf{ML}_e^{\Box}$).

ML$^\Box$: Syntax

ML$^\Box$ arises from the simply-typed λ-calculus by adding a type constructor $\Box$.

`code` M, where M is any ML$^\Box$ expression, is a generator and $\Box\tau$ is the type of code generators of code of type τ.

Two kinds of variables are available. Value variables are bound by λ's and code variables bound by `let cogen`'s.

Type variables	α, β, γ		
Types	τ, σ	$::=$	$\alpha \mid \tau \to \sigma \mid \Box\tau$
Terms	M, N	$::=$	$x \mid \lambda x.M \mid M N \mid u$
			$\mid$ `code` $M \mid$ `lift` M
			$\mid$ `let cogen` $u = M$ `in` N `end`
Contexts	Γ	$::=$	$\cdot \mid \Gamma, x : \tau \mid \Gamma, u : \tau$

The generation of code is postponed as long as possible. So, evaluation of `let cogen` $u = M$ `in N end` will not immediately generate code: the generation of specialized code for M is deferred until the code variable u is encountered during evaluation.

`lift` $M : \Box\tau$ is $M : \tau$ returns a generator which just quotes the resulting value.

$\mathsf{ML}^{\Box}$: Typing Rules

The typing judgement $\Delta; \Gamma \vdash M : \tau$ uses one context Δ for code variables and one context Γ for value variables.

$$\frac{x{:}\tau \in \Gamma}{\Delta;\Gamma \vdash x{:}\tau} \qquad \frac{u{:}\tau \in \Delta}{\Delta;\Gamma \vdash u{:}\tau} \qquad \frac{\Delta;\Gamma \vdash M{:}\tau\to\sigma \quad \Delta;\Gamma \vdash N{:}\tau}{\Delta;\Gamma \vdash MN{:}\sigma}$$

$$\frac{\Delta;\cdot \vdash M{:}\tau}{\Delta;\Gamma \vdash M{:}\Box\tau} \qquad \frac{\Delta;\Gamma \vdash M{:}\tau}{\Delta;\Gamma \vdash \texttt{lift}\ M{:}\Box\tau}$$

$$\frac{\Delta;(\Gamma,x{:}\tau) \vdash M{:}\sigma}{\Delta;\Gamma \vdash \lambda x.M{:}\tau\to\sigma} \qquad \frac{\Delta;\Gamma \vdash M{:}\Box\tau \quad (\Delta,u{:}\tau);\Gamma \vdash N{:}\sigma}{\Delta;\Gamma \vdash \texttt{let cogen}\ u{=}M\ \texttt{in}\ N\ \texttt{end}{:}\sigma}$$

Only code variables are permitted to occur free in generators.

The `let cogen` rule expresses that if we have a value which is a code generator (and therefore of type $\Box\tau$), we can bind a code variable u of type τ which may be included in other code generators.

Examples

$\vdash \texttt{code}(\lambda x.x) : \Box(\alpha \to \alpha)$

$\vdash \lambda x.\texttt{let cogen}\ u = x\ \texttt{in}\ u\ \texttt{end} : (\Box\alpha) \to \alpha$

$\vdash \lambda f.\lambda g.\texttt{let cogen}\ f' = f\ \texttt{in}$
$\texttt{let cogen}\ g' = g\ \texttt{in}$
$\texttt{code}\ \lambda x.f'(g'(x))\texttt{end end}$
$: \Box(\beta \to \gamma) \to \Box(\alpha \to \beta) \to \Box(\alpha \to \gamma)$

ML$^\Box$: Programming

```
type poly = int list;
val poly1 = [2, 4, 0, 2333];
(* val evalPoly : int * poly -> int *)
fun     evalPoly (x, nil) = 0
|       evalPoly (x, a::p) = a + (x * evalPoly (x, p));
```

Let us assume we want to specialize the evaluation function to a particular polynomial.

```
fun     specPoly (nil) = (fn x => 0)
|       specPoly (a::p) =
        let val polyp = specPoly p
        in fn x => a + (x * polyp x) end
val poly1target = specPoly poly1;
```

Without support of the compiler optimizations such as unfolding are not performed. Staging transformations need RTCG to be effective.

```
(* val compPoly : poly -> [int -> int] *)
fun     compPoly (nil) = code (fn x => 0)
|       compPoly (a::p) =
        let cogen f = compPoly p
            cogen a' = lift a
        in code (fun x => a' + (x * f x)) end
val codeGenerator = compPoly poly1;
val mlPolyFun = eval codeGenerator;
```

One can also use library routines to take benefit from RTCG, without commiting oneself to write staged code.

```
(* val client : t1 -> [t2 -> t3] *)
fun client x =
   ... code (fun y => ... compPoly (makePoly y) ...)
   ...
```

ML$^\Box$: The CAM

The CAM is an stack-based abstract machine which uses categorical combinators as its instructions.

$$
\begin{array}{llcl}
\text{State} & T & ::= & (S, P) \\
\text{Instructions} & i & ::= & \texttt{id} \mid \texttt{fst} \mid \texttt{snd} \mid \texttt{push} \mid \texttt{swap} \\
 & & & \mid \texttt{cons} \mid \texttt{app} \mid \text{`}e \mid \texttt{Cur}(P) \\
\text{Values} & e, f & ::= & () \mid [e : P] \mid (e, f) \\
\text{Programs} & P & ::= & i; \cdots ; i \mid \cdot \\
\text{Stacks} & S & ::= & \cdot \mid e :: S
\end{array}
$$

The top element e of the stack $(e :: S)$ is the current environment in which the program P is to be executed.

The environment consisting of values $f_1, \ldots , f_n$ is represented as the value $(((), f_1), \ldots , f_n)$.

The fundamental invariant is that the instructions for P consume the current environment and replace it by the value of P.

ML$^\Box$: The CCAM

The CCAM is an extension of the CAM that provides facilities for RTCG

The CCAM encodes generating extensions as a series of `emit` instructions. For technical reasons, `emit` cannot be nested, hence the division of the set of instructions.

Instructions are emitted directly into a dynamically created code block that can afterwards be inserted in the instruction stream.

Simple Inst	i	$::=$	$\texttt{id} \mid \texttt{fst} \mid \texttt{snd} \mid \texttt{push} \mid \texttt{swap}$
			$\mid \texttt{cons} \mid \texttt{app} \mid {}`e \mid \texttt{lift}$
			$\mid \texttt{arena} \mid \texttt{merge} \mid \texttt{call}$
Composite Inst	I	$::=$	$i \mid \texttt{emit}(i) \mid \texttt{Cur}(P)$
Values	e, f	$::=$	$(e, f) \mid [e : P] \mid B \mid ()$
Code Blocks	B	$::=$	$\{P\}$
Programs	P	$::=$	$I; \cdots ; I$
Stacks	S	$::=$	$\cdot \mid e :: S$

`arena` creates an empty code block on the top of the stack.

`lift` inserts a quoted value into the current code block.

`merge` inserts a function into the current code block.

`call` invokes the code block by inserting it in the stream.

$\mathbf{ML}^{\Box}$: Transitions of the CCAM

Stack	*Program*	*Stack*	*Program*
S	$\mathtt{id}; P$	S	P
$(e, f) :: S$	$\mathtt{fst}; P$	$e :: S$	P
$(e, f) :: S$	$\mathtt{snd}; P$	$f :: S$	P
$e :: S$	$`f; P$	$f :: S$	P
$e :: S$	$\mathtt{push}; P$	$e :: e :: S$	P
$e :: f :: S$	$\mathtt{swap}; P$	$f :: e :: S$	P
$e :: f :: S$	$\mathtt{cons}; P$	$(f, e) :: S$	P
$e :: S$	$\mathtt{Cur}(P'); P$	$[e : P'] :: S$	P
$([e : P'], f) :: S$	$\mathtt{app}; P$	$(e, f) :: S$	$P'; P$
$e :: S$	$\mathtt{arena}; P$	$\{\cdot\} :: S$	P
$(e, \{P'\}) :: S$	$\mathtt{emit}(i); P$	$(e, \{P'; i\}) :: S$	P
$(e, \{P'\}) :: S$	$\mathtt{lift}; P$	$(e, \{P'; `e\}) :: S$	P
$(\{P'\}, (e, \{P''\})) :: S$	$\mathtt{merge}; P$	$(e, \{P''; \mathtt{Cur}(P')\}) :: S$	P
$(e, \{P'\}) :: S$	$\mathtt{call}; P$	$e :: S$	$P'; P$

ML$^\square$: Compilation

RTCG need to make use of values that are computed at run-time. As in Mini-ML$^\square_e$, that kind of information is supplied through the free code variables in a code expression.

$$\texttt{let cogen u = code 0 in code } \lambda x.u * x \texttt{ end}$$

Underlining pictures expressions turned into `emit` sequences.

$$\texttt{let cogen u = code } \underline{0} \texttt{ in code } \underline{\lambda x.}u\underline{*x} \texttt{ end}$$

When the expression is executed, the code generator bound to u is activated, and the emitted expression will turn to be $\lambda x.0 * x$. As an optimization, one may specialize the emission in case u turns out to be 0, and if so emit $\lambda x.0$.

We use variable contexts Ω and Λ to keep track of the position of bound variables in the run-time environment.

$$\text{Variable Contexts} \quad \Omega, \Delta ::= ()|(\Omega, a)$$

We define a pair of functions from ML$^\square$ terms to CCAM programs:

- $[\![M]\!]_\Omega$ compiles an ML$^\square$ term M with free variables in Ω into normal CCAM code.
- $[\![M]\!]_\Omega^\Lambda$ translates M with free variables either in Ω or Λ into a generating extension for M, specializing it to variables contained in Λ.

ML$^{\Box}$: Compiling Core ML Terms

$[\![M]\!]_\Omega$ is used to compile in the context Ω terms which are not enclosed in a code construct.

$$
\begin{array}{ll}
[\![x]\!]_\Omega & = \Omega(x) \\
[\![\lambda x.M]\!]_\Omega & = \mathtt{Cur}([\![M]\!]_{(\Omega,x)}) \\
[\![M\ N]\!]_\Omega & = \langle [\![M]\!]_\Omega, [\![N]\!]_\Omega \rangle ; \mathtt{app}
\end{array}
$$

Example

The value of M is $[e' : P']$ and the value of N if f. We show the compiled version of the application evaluates as expected.

$$
\begin{array}{lll}
(e :: S, & \langle [\![M]\!]_\Omega, [\![N]\!]_\Omega \rangle ; \mathsf{app}; P) & \Rightarrow \\
(e :: e :: S, & [\![M]\!]_\Omega, [\![N]\!]_\Omega \rangle ; \mathsf{app}; P) & \Rightarrow^* \\
([e' : P'] :: e :: S, & , [\![N]\!]_\Omega \rangle ; \mathsf{app}; P) & \Rightarrow \\
(e :: [e' : P'] :: S, & [\![N]\!]_\Omega \rangle ; \mathsf{app}; P) & \Rightarrow^* \\
(f :: [e' : P'] :: S, & \rangle ; \mathsf{app}; P) & \Rightarrow^* \\
(([e' : P'], f) :: S, & \mathsf{app}; P) & \Rightarrow \\
((e', f) :: S, & P'; P) &
\end{array}
$$

ML$^{\Box}$: Compiling Code Generators

The compilation function for generating extensions employs two contexts: Λ to hold the free variables to which the term should be specialized (all are free variables), and Ω to hold the local bindings set we are into.

$$
\begin{array}{ll}
[\![\texttt{let cogen } u = M \texttt{ in } N \texttt{ end}]\!]_{\Omega} & = \langle [\![M]\!]_{\Omega} \rangle ; [\![N]\!]_{(\Omega,u)} \\
[\![\texttt{code } M]\!]_{\Omega} & = \texttt{Cur}([\![M]\!]^{\Omega}_{()}) \\
[\![u]\!]_{\Omega} & = \langle \Omega(u), \texttt{arena} \rangle ; \texttt{app}; \texttt{call} \\
[\![\texttt{lift } M]\!]_{\Omega} & = [\![M]\!]_{\Omega} ; \texttt{Cur}(\texttt{lift})
\end{array}
$$

$$
\begin{array}{ll}
[\![x]\!]^{\Lambda}_{\Omega} & = \underline{\Omega(x)} \\
[\![M \ N]\!]^{\Lambda}_{\Omega} & = \underline{\langle} [\![M]\!]^{\Lambda}_{\Omega} \underline{,} [\![N]\!]^{\Lambda}_{\Omega} \underline{\rangle} ; \underline{\mathsf{app}} \\
[\![\texttt{let cogen } u = M \texttt{ in } N \texttt{ end}]\!]^{\Lambda}_{\Omega} & = \underline{\langle} [\![M]\!]^{\Lambda}_{\Omega} \underline{\rangle} ; [\![N]\!]^{\Lambda}_{(\Omega,u)}
\end{array}
$$

Example

P_M is the program emitted by $[\![M]\!]^{\Lambda}_{\Omega}$ and P_N is the program emitted by $[\![N]\!]^{\Lambda}_{\Omega}$.

$$
\begin{array}{l}
((e, \{P'\}) :: S, \underline{\langle} [\![M]\!]^{\Lambda}_{\Omega} \underline{,} [\![N]\!]^{\Lambda}_{\Omega} \underline{\rangle} ; \underline{\mathsf{app}}; P) \Rightarrow \\
((e, \{P'; \langle\}) :: S, [\![M]\!]^{\Lambda}_{\Omega} \underline{,} [\![N]\!]^{\Lambda}_{\Omega} \underline{\rangle} ; \underline{\mathsf{app}}; P) \Rightarrow^{*} \\
((e, \{P'; \langle P_M\}) :: S, \underline{,} [\![N]\!]^{\Lambda}_{\Omega} \underline{\rangle} ; \underline{\mathsf{app}}; P) \Rightarrow \\
((e, \{P'; \langle P_M,\}) :: S, [\![N]\!]^{\Lambda}_{\Omega} \underline{\rangle} ; \underline{\mathsf{app}}; P) \Rightarrow^{*} \\
((e, \{P'; \langle P_M, P_N\}) :: S, \underline{\rangle} ; \underline{\mathsf{app}}; P) \Rightarrow \\
((e, \{P'; \langle P_M, P_N\rangle\}) :: S, \underline{\mathsf{app}}; P) \Rightarrow \\
((e, \{P'; \langle P_M, P_N\rangle ; \mathsf{app}\}) :: S, P)
\end{array}
$$

ML$^{\Box}$: Technical Problems

□ A technical problem arises when we compile functions in code generators that gives rise to the `merge` instruction.

$$[\![\lambda x.M]\!]^{\Lambda}_{\Omega} = \langle\langle \texttt{fst}, \texttt{arena}\rangle; [\![M]\!]^{\Lambda,x}_{\Omega}; \texttt{snd}, \texttt{id}\rangle; \texttt{merge}$$

It comes from the fact we cannot emit `Cur` instructions.

□ Code variables are bound when executing. Thus, when executing the innermost let-expression in the following example, `u'` has not been entered in the environment yet.

```
let cogen u = code (4+5)
in code (let cogen u' = code 2 in 3 + u' + u end) end
```

$$\begin{aligned} & [\![u]\!]^{\Lambda}_{\Omega} \\ &= \underline{\langle \Omega(u),}\underline{\texttt{arena}\rangle}; \underline{\texttt{app}}; \underline{\texttt{call}} \quad \text{if } u \text{ is in } \Omega \\ &= \langle \texttt{fst}, \langle \texttt{fst}; \Lambda(u), \texttt{snd}\rangle; \texttt{app}; \texttt{snd}\langle \quad \text{if } u \text{ is in } \Lambda \end{aligned}$$

□ More complicated.

$$[\![\texttt{code } M]\!]^{\Lambda}_{\Omega} = \underline{\langle}\langle \texttt{fst}, \langle `(); \texttt{Cur}(\texttt{fst}; \texttt{Cur}([\![M]\!]^{\Lambda|\Omega}_{()})), \texttt{snd}\rangle; \texttt{lift}; \texttt{snd}\rangle\underline{,\texttt{id}}\underline{\rangle}; \underline{\texttt{app}}$$

$$[\![\texttt{lift } M]\!]^{\Lambda}_{\Omega} = [\![M]\!]^{\Lambda}_{\Omega}; \langle\langle \texttt{fst}, \texttt{arena}\rangle; \underline{\texttt{lift}}; \texttt{snd}, \texttt{id}\rangle; \texttt{merge}$$

Conclusion

- Automatic staging analysis will not elude soon, the programmer's insight and notations will still be desirable.
- No intermediate representation of code sould be processed at run-time. Instead, portion of programs should be compiled that will generate code on demand.
- The instruction stream encoding enables a great deal of flexibility in the kind of low-level optimizations performed at run-time.
- Multi-stage languages are not only intermediate representations for binding-time analysis, but also programming languages in their own right.

References

[LL94] Mark Leone, Peter Lee, *Lightweight Run-time Code Generation*, Partial Evaluation and Semantics-based Program Manipulation, 1994.

[LL96] Peter Lee, Mark Leone, *Optimizing ML with Run-Time Code Generation*, POPL '96.

[TS97] Walid Taha, Tim Sheard, *Multi-Stage Programming with Explicit Annotations*, PEPM '97.

[D96] Rowan Davies, *A Temporal-Logic Approach to Binding-Time Analysis*, IEEE Symposium on Logic in Computer Science, 1996.

[DP96] Rowan Davies, Frank Pfenning, *A Modal Analysis of Staged Computation*, POPL '96.

[WLP98] Philip Wickline, Peter Lee, Frank Pfenning, *Run-time Code Generation and Modal-ML*, PLDI '98.