1 Outline

1. Motivation
2. \( \mathbb{Z} \) -language
3. Futamura projections
4. Implementation
5. Challenges
6. Conclusion
2 Motivation
2.1 Idea

- partial evaluation = program specialization w.r.t. static input
- analogous to function restriction in analysis:

\[ f(x, y) = 2x + y \]
\[ f_2(y) = 4 + y \]

- overall goal: generate efficient specialized programs from general ones by completely automatic methods
2.2 Partial evaluator

![Diagram of a partial evaluator]

Figure 1.1: A partial evaluator.
2.3 How do we make code fast?

1. Take static knowledge and manually specialize code
2. Optimizing compilers do this automatically
3. Factor compiler into optimizer and translator
4. Partial evaluator = optimizer in its own language

*Will this lead to significant speedup?*
2.4 Notation

- \( L = \) implementation language
- \( S = \) source language
- \( T = \) target language
- \( p = \) subject program written in \( S \)
- \( mix = \) partial evaluator written in \( L \)
- \( in1 = \) static input
- \( in2 = \) dynamic input

\[
[p]_S [in1, in2] = [[mix]_L [p, in1]]_T [in2]
\]

\( \text{specialized program} \)
2.5 Futamura projections

1. Partial evaluation as compilation:

\[
[[\text{int}]] [\text{source, input}] = [[[[\text{mix}]] [\text{int, source}]]] [\text{input}]
\]

\text{compiled program}

2. Compiler generation by self-application (S = L required):

\[
[[\text{mix}]] [\text{int, source}] = [[[[\text{mix}]] [\text{mix, int}]]] [\text{source}]
\]

\text{compiler}

3. Compiler generator generation:

\[
[[\text{mix}]] [\text{mix, int}] = [[[[\text{mix}]] [\text{mix, mix}]]] [\text{int}]
\]

\text{compiler generator}
2.6 Baron von Münchhausen
2.7 Practical relevance

- GraalVM (universal virtual machine for running applications written in JavaScript, Python, Ruby, R, JVM-based languages like Java etc.)
- uses partial evaluation to compile code using an interpreter and augmenting the interpreted program with available information
- i.e. implements the first Futamura projection
- benchmarks:

Figure 10: Speedup of our system relative to best production VM (higher is better).
3 ℤ language
4 Futamura Projections (in 15 minutes or less)
4.1 Basic Specialization

- **Specializer \textit{mix}**: 
  - (maybe written in S0);
  - taking programs in S0
  - and some of their input;
  - returning programs in S0;

- **Program p\_S0**: 
  - written in S0;
  - taking \textit{i\_1} and \textit{i\_2} as input;
  - returning \textit{output}

\textbf{The Very Important Equation:}

\[
\text{output} = p\_S0(\textit{i\_1, i\_2}) = \text{mix}(p\_S0, \textit{i\_1}) (\textit{i\_2})
\]
4.2 Futamura 1

- Specializer mix:
  - written in S0;
  - taking programs in S0
  - and some of their input;
  - returning programs in S0;

- Interpreter z_int for Z:
  - written in S0;
  - taking programs in Z
  - and some their input;
  - returning the program's output;

- Program p_Z:
  - written in Z;
  - taking input i;
  - returning result

output = p_S0 (i_1, i_2)
result = z_int (p_Z, i)
4.2.1 Futamura 1 in Action

\[
\text{output} = p_{S0} \ (i_1, i_2) = \text{mix}(p_{S0}, \ i_1) \ (i_2)
\]

\[
\text{result} = z_{\text{int}} \ (p_z, i) = \text{mix}(z_{\text{int}}, p_z) \ (i) \quad ;; \text{Futamura 1!}
\]

\[
p_{z_C} = \text{mix}(z_{\text{int}}, p_z)
\]

\[
\text{result} = p_{z_C} \ (i)
\]

- \( p_{z_C} \) is a program \( p_z \) compiled to \( S0! \)
4.2.2 Scheme0

- Jones et. al. in *Partial Evaluation and Automatic Program Generation*
- A VERY restricted subset of Scheme
- There is a Partial Evaluator from Scheme0 to Scheme0
- Just implement a Z-Interpreter in Scheme0
4.2.3 Z-Interpreter

- Input:
  - \( p_z \): a program in Z;
  - \( x_n \): a list of variable names;
  - \( x_v \): a list of variable values (numerical);
- Result: \( \text{result} = z_{\text{int}}(p_z, x_n, x_v) \).
- Execution:
  - Check if \( p_z \) is a valid Z Term;
  - Substitute variables in \( p_z \);
  - If there are variables left in \( p_z \), die horribly;
  - Step until \( p_z \) is reduced to a number (recall homework 3 :).
4.2.4 Z-interpreter

(define (z-int p_z xn xv)
  (if
    (call is-z-term? p_z)
    (call z-stepper (call z-subst p_z xn xv))
    (op error 'z-int "Not a valid $\mathbb{Z}$ term ~s" p_z))))
4.2.5 Applying Futamura 1 to Z-Interpreter

\[ \text{output} = p_{S0} \ (i_1, i_2) = \text{mix}(p_{S0}, i_1) \ (i_2) \]
\[ \text{result} = z_{\text{int}} \ (p_z, i) = \text{mix}(z_{\text{int}}, p_z) \ (i) ;; \text{Futamura 1!} \]
\[ \text{result} = z_{\text{int}} \ (p_z, x_n, x_v) = \text{mix}(z_{\text{int}}, [p_z, x_n]) \ (x_v) ;; \text{Ours} \]

\[ p_{z_C} = \text{mix}(z_{\text{int}}, [p_z, x_n]) \]
\[ \text{result} = p_{z_C}(x_v) \]
4.2.6 Inspecting a Compiled Z Program

Interpreter

```
(define (z-int p_z xn xv)
  (if
    (call is-z-term? p_z)
    (call z-stepper (call z-subst p_z xn xv))
    (op error 'z-int "Not a valid Z term ~s" p_z)))
```

Source

```
(++ x (i-t-e y (++ 4 x) (++ 5 6)))
```

Compiled to Scheme0

```
(define ( ((z-int 1) S S D) (x y)) ;; what a name!
xv ;; no xn as input anymore
(xv)
(call (((z-stepper 2) D)) ;; is-z-term? is gone!
  (op cons ;; z-subst got inlined
    '++
    ...)))
```

Does it run faster?
4.2.7 Futamura 1 Speedup

It does!

output = $z_{\text{int}}(p_z, [x_n, x_v])$

vs

output = $p_z_C(x_v)$
4.2.8 Compiling Z Programs

```c
p_z_C = mix (z_int, [p_z, xn])
output = p_S0 (i_1, i_2)    // wink-wink ;)
```

*Do we really need mix here?*
4.2.9 Creating a Compiler with Futamura 2

\[
\text{output} = p_{S0}(i_1, i_2) = \text{mix}(p_{S0}, i_1)(i_2)
\]
\[
p_{z_C} = \text{mix}(z_{int}, [p_z, xn]) = \text{mix}([mix, z_{int}])[p_z, xn]
\]
\[
z_{compiler} = \text{mix}([mix, z_{int}])
\]
\[
p_{z_C} = z_{compiler}([p_z, xn])
\]
4.2.10 Futamura 2 Speedup

\[ p_{z_C} = \text{mix} \left( z_{\text{int}}, [p_z, x_n] \right) \]

vs

\[ p_{z_C} = \text{z_compiler} \left( [p_z, x_n] \right) \]
4.2.11 Creating a Compiler

```
z_compiller = mix (mix, z_int)
output     = p_S0(i_1, i_2  )    ;; wink
```

*Do we really need mix here?*
4.2.12 Creating a Compiler Generator with Futamura 3

\[
\text{output} = p_{S0}(i_1, i_2) = \text{mix}(p_{S0}, i_1)(i_2)
\]
\[
z_{\text{compiler}} = \text{mix} \ (\text{mix}, z_{\text{int}}) = \text{mix}(\text{mix}, \text{mix})(z_{\text{int}})
\]

\[
\text{compiler}_\text{generator} = \text{mix}(\text{mix}, \text{mix})
\]

\[
z_{\text{compiler}} = \text{compiler}_\text{generator}(z_{\text{int}})
IMP_{\text{compiler}} = \text{compiler}_\text{generator}(IMP_{\text{int}})
\]

... 

\textit{Wait, WHAT}!?
4.2.13 Futamura 3 speedup:

It's real, and it's fast too.

\[
z\text{\_compiler} = \text{mix}\ (\text{mix}, z\_\text{int})
vs
z\_\text{compiler} = \text{compiler\_generator}(z\_\text{int})
\]
4.2.14 What Do We Need to Make It Work?

Let's look at the three Futamura projections and see what was required. The only requirement is that `mix` and the `z_int` are written in S0.

- Futamura 1: Z-Interpreter written in S0
- Futamura 2: Z-Interpreter written in S0 and `mix` written in S0
- Futamura 3: `mix` written in S0 (it's not about Z anymore)
4.2.15 Futamura projections

1. Partial evaluation as compilation:

\[
\begin{align*}
[[\text{int}]] [\text{source, input}] &= [[[[\text{mix}]] [\text{int, source}]] [\text{input}]] \\
&= \text{compiled program}
\end{align*}
\]

2. Compiler generation by self-application (S = L required):

\[
\begin{align*}
[[\text{mix}]] [\text{int, source}] &= [[[[\text{mix}]] [\text{mix, int}]] [\text{source}]] \\
&= \text{compiler}
\end{align*}
\]

3. Compiler generator generation:

\[
\begin{align*}
[[\text{mix}]] [\text{mix, int}] &= [[[[\text{mix}]] [\text{mix, mix}]] [\text{int}]] \\
&= \text{compiler generator}
\end{align*}
\]
5 Implementation
5.1 Scheme0

- lexically scoped subset of Scheme
- recursion, but *no anonymous* functions
- specialization points: at *function definition*
- possible also to specialize on *conditionals*
- Chapter 5 of Jones et al.
5.2 Binding-time analysis

find static program parts to evaluate

1. type inference or
2. abstract interpretation (here)
   • add annotations to divide expressions
   • reduces set of function definitions found
5.2.1 Division between static and dynamic

- find *lowest upper bound* \( \sqcup \) on partial order of divisions
- use annotations in code, i.e. extended Scheme0
- *polyvariant* binding times \( \rightarrow \) many *monovariant*
5.2.2 Analysis algorithm

\[
\begin{align*}
B_c[e] &: \text{BTEnv} \rightarrow \text{BindingTime} \\
B_c[c][\tau] &= S \\
B_c[x_j][\tau] &= t_j \text{ where } \tau = (t_1, \ldots, t_a) \\
B_c[(\text{if } e_1 \ e_2 \ e_3)][\tau] &= B_c[e_1][\tau] \cup B_c[e_2][\tau] \cup B_c[e_3][\tau] \\
B_c[(\text{call } f \ e_1 \ \ldots \ e_a)][\tau] &= \bigsqcup_{j=1}^a B_c[e_j][\tau] \\
B_c[(\text{op } e_1 \ \ldots \ e_a)][\tau] &= \bigsqcup_{j=1}^a B_c[e_j][\tau]
\end{align*}
\]

Figure 5.3: The Scheme0 binding-time analysis function \( B_c \).

Jones et al.
5.2.3 Propagation algorithm

\[
\begin{align*}
\mathcal{B}_v[e] : & \text{BTEnv} \to \text{FuncName} \to \text{BTEnv} \\
\mathcal{B}_v[c] & = (S, \ldots, S) \\
\mathcal{B}_v[x_j] & = (S, \ldots, S) \\
\mathcal{B}_v[(\text{if } e_1 e_2 e_3)] & = \mathcal{B}_v[e_1] \sqcup \mathcal{B}_v[e_2] \sqcup \mathcal{B}_v[e_3] \\
\mathcal{B}_v[(\text{call } f e_1 \ldots e_a)] & = \begin{cases} 
\mathcal{B}_v[e_1], \ldots, \mathcal{B}_v[e_a] & \text{if } f = g \\
\mathcal{B}_v[e_1], \ldots, \mathcal{B}_v[e_a] & \text{if } f \neq g 
\end{cases} \\
\mathcal{B}_v[(\text{op } e_1 \ldots e_a)] & = \bigcup_{j=1}^a \mathcal{B}_v[e_j]
\end{align*}
\]

Figure 5.4: The Scheme0 binding-time propagation function $\mathcal{B}_v$.

Jones et al.
6 Challenges
6.1 Infinite unfolding

Solution: Do not unfold calls in dynamic conditionals \texttt{ifd}

\begin{verbatim}
(define (app (ys) (xs))
  (ifd (null?d xs)
    (lift ys)
    (consd (card xs) (calld app (ys) ((cdrd xs))))))
\end{verbatim}
6.2 Code duplication

\[
\begin{align*}
\text{(define (f n) (if (= n 0) 1 (g (f (- n 1))))))} \\
\text{(define (g m) (+ m m))}
\end{align*}
\]

\(f\) linear in \(n\) is after unfolding \(g\)

\[
\begin{align*}
\text{(define (f n) (if (= n 0) 1 (+ (f (- n 1)) (f (- n 1))))})
\end{align*}
\]

\textbf{exponential} in \(n\)
6.3 Side-effects
6.3 Side-effects
6.4 Resource analysis & al.

- *how much* speed up is possible?
- *predict* speed up before specialization
- *generate machine abstractions* tailor-made to a source language defined by an interpreter
7 Partial Evaluation in Graal
7.1 Truffle framework

Grimmer et al.
import polyglot
array = polyglot.eval(language="js", string="[1,2,42,4]")
print(array[2])

Specialization happens across language boundaries!
7.3 Partial evaluation approach

- *JIT* compiler integration: *deoptimization*
- only *optimize happy path* when *stable* runtime state
- else *use interpreter*
- *reoptimize* when state changes
7.4 Tweaks

- *method-external invalidation* of compiled code
- define *runtime boundaries* for PE
- only code that was *explicitly designed for PE* is reachable
- still *longer warmup* times of a minute until peak performance
8 Related work

- PyPy also only requires interpreter, but uses meta-tracing
- JVM hotspot compiler, v8 compiler
9 Conclusion

- partial evaluation is very powerful and practical
- partial evaluation applied to deep learning and bayesian inference
  https://gitlab.com/whilo/foppl-compiler
10 References

11 Backlog
(define (specialize program vs₀)
   (let ((define (f₁ _ _ _) _ _) = program
       in (complete (list (f₁ :: vs₀))) () program))
)

(define (complete pending marked program)
   (if pending is empty then
       ()
     else
       (let (f . vs) ∈ pending
           (let (define (f (x₁...xₘ) (xₘ₊₁...xₐ)) e) = (lookup f program)
               (let (vs₁ ... vsₘ) = vs
                   (let eₜₐₗ = (reduce e (x₁...xₘ xₘ₊₁...xₐ) (vs₁...vsₘ xₘ₊₁...xₐ))
                       (let newmarked = marked ∪ {(f . vs)}
                           (let newpending = (pending ∪ (successors eₜₐₗ)) \ newmarked
                               let newdef = (list 'define (list (f . vs) xₘ₊₁...xₐ) eₜₐₗ)
                                   in (newdef :: (complete newpending newmarked program))))
               ))
       )
   )
)

Jones et al.
11.2 Stepper in Scheme0

```
(define (z-step t)
  (if (op atom? t)
      (if (op number? t) t
          (op error))
      (slet (toplevel (tag t))
        (if (op equal? toplevel '++)
            (call handle++ t)
            (if (op equal? toplevel 'i-t-e)
                (call handle-ite t)
                (op error))))))
```
11.3 Scheme0 syntax

\[
\begin{align*}
\langle \text{Program} \rangle & ::= (\langle \text{Equation} \rangle \ldots \langle \text{Equation} \rangle) & \text{Function definitions} \\
\langle \text{Equation} \rangle & ::= (\text{define} (\langle \text{FuncName} \rangle \langle \text{Varlist} \rangle) \langle \text{Expr} \rangle) \\
\langle \text{Varlist} \rangle & ::= \langle \text{Var} \rangle \ldots \langle \text{Var} \rangle & \text{Formal parameters} \\
\langle \text{Expr} \rangle & ::= \langle \text{Constant} \rangle & \text{Constant} \\
& \mid \langle \text{Var} \rangle & \text{Variable} \\
& \mid (\text{if} \langle \text{Expr} \rangle \langle \text{Expr} \rangle \langle \text{Expr} \rangle) & \text{Conditional} \\
& \mid (\text{call} \langle \text{FuncName} \rangle \langle \text{Arglist} \rangle) & \text{Function application} \\
& \mid (\langle \text{Op} \rangle \langle \text{Expr} \rangle \ldots \langle \text{Expr} \rangle) & \text{Base application} \\
\langle \text{Arglist} \rangle & ::= \langle \text{Expr} \rangle \ldots \langle \text{Expr} \rangle & \text{Argument expressions} \\
\langle \text{Constant} \rangle & ::= \langle \text{Numeral} \rangle \\
& \mid (\text{quote} \langle \text{Value} \rangle) \\
\langle \text{Op} \rangle & ::= \text{car} | \text{cdr} | \text{cons} | = | + | \ldots
\end{align*}
\]
11.4 Type inference excerpt

\[ \tau \vdash c : S \]
\[ \tau[x \mapsto t] \vdash x : t \]
\[ \tau \vdash e_1 : S \ldots \tau \vdash e_a : S \]
\[ \tau \vdash (\text{ops } e_1 \ldots e_a) : S \]
\[ \tau \vdash e_1 : D \ldots \tau \vdash e_a : D \]
\[ \tau \vdash (\text{opd } e_1 \ldots e_a) : D \]
\[ \tau \vdash e_1 : S \quad \tau \vdash e_2 : t \quad \tau \vdash e_3 : t \]
\[ \tau \vdash (\text{ifs } e_1 \; e_2 \; e_3) : t \]
\[ \tau \vdash e_1 : D \quad \tau \vdash e_2 : D \quad \tau \vdash e_3 : D \]
\[ \tau \vdash (\text{ifd } e_1 \; e_2 \; e_3) : D \]

Jones et al.
11.5 Syntax

Program Static Syntax

\[ t \in \text{TERM}, \quad n \in \mathbb{Z}, \quad x \in \text{VAR}, \]
\[ t ::= n \mid x \mid \text{if } t \text{ then } t \text{ else } t \mid t + t \]

Program Runtime Syntax

\[ nr \in \text{RESIDUAL} \subseteq \text{TERM}, \]
\[ F \in \text{FRAME}, \quad R \in \text{RESIDUAL FRAME}, \quad o \in \text{OBS} \]
\[ i \in \text{IVARS} = \text{VAR} \rightarrow \mathbb{Z}, \]

\[ R ::= nr + \Box \mid \text{if } nr \text{ then } nr \text{ else } \Box \]
\[ F ::= \Box \mid R \mid \text{if } \Box \text{ then } t \text{ else } t \mid \text{if } nr \text{ then } \Box \text{ else } t \mid \Box + t \]
\[ nr ::= n \mid x \mid R[nr] \]
\[ o ::= n \mid \text{unknown-identifier} \]
11.6 Semantics

$\rightarrow^* \subseteq \text{TERM} \times \text{TERM}$  
**Single-step Reduction**

- **(splus)**
  
  \[
  n_1 + n_2 \rightarrow n_1 + n_2
  \]

- **(sif-t)**
  
  \[
  \frac{n \neq 0}{\text{if } n \text{ then } t_2 \text{ else } t_3 \rightarrow t_2}
  \]

- **(sif-f)**
  
  \[
  \frac{0}{\text{if } 0 \text{ then } t_2 \text{ else } t_3 \rightarrow t_3}
  \]

- **(sframe)**
  
  \[
  \frac{p \rightarrow p'}{F[p] \rightarrow F[p']}
  \]

$\rightarrow^* \subseteq \text{TERM} \times \text{TERM}$  
**Multi-step Reduction**

- **(incl)**
  
  \[
  \frac{t_1 \rightarrow t_2}{t_1 \rightarrow^* t_2}
  \]

- **(refl)**
  
  \[
  \frac{t \rightarrow t}{t \rightarrow^* t}
  \]

- **(trans)**
  
  \[
  \frac{t_1 \rightarrow^* t_2 \quad t_2 \rightarrow^* t_3}{t_1 \rightarrow^* t_3}
  \]
11.7 Intepreter

\[
\text{input} : \text{TERM} \times \text{IVARS} \to \text{TERM}
\]

\[
\begin{align*}
\text{input}(n, i) &= n \\
\text{input}(x, i) &= n, \exists n \in \mathbb{N}. i(x) = n \\
\text{input}(x, i) &= x, \nexists n \in \mathbb{N}. i(x) = n \\
\text{input}(\text{if } t_1 \text{ then } t_2 \text{ else } t_3, i) &= \text{if } \text{input}(t_1, i) \text{ then } \text{input}(t_2, i) \text{ else } \text{input}(t_3, i) \\
\text{input}(t_1 + t_2, i) &= \text{input}(t_1, i) + \text{input}(t_2, i)
\end{align*}
\]

\[
\text{peval}_{HPE} : \text{TERM} \times \text{IVARS} \to \text{RESIDUAL}
\]

\[
\text{peval}_{HPE}(t, i) = \text{nr if } \text{input}(t, i) \longrightarrow^* \text{nr}
\]

\[
\text{eval}_{HPE} : \text{TERM} \times \text{IVARS} \to \text{OBS}
\]

\[
\begin{align*}
\text{eval}_{HPE}(t, i) &= \text{unknown-identifier if } \text{Vars}(\text{input}(t, i)) \neq \emptyset \\
\text{eval}_{HPE}(t, i) &= n \text{ if } \text{input}(t, i) \longrightarrow^* n
\end{align*}
\]