Unboxed objects and polymorphic typing

Xavier Leroy
Presented by Woongsik Choi

10/30/2000

□ Introduction

- Exact type is not known when compiling polymorphic functions.
  - doesn't know size of function parameter
  - can't use specialized features directly.
    e.g.) floating point registers for type float.

- Two conventional methods.
  - defer compiling until it is used.
    code duplication and loss of seperate compilation.
  - boxing
    allocate all objects at heap and handle through pointers.
    every object is represented and handled same way.

□ Introduction

- In this paper
  - improve boxing method by using mixed representation.
  - when monomorphic specialized(unboxed) representation.
  - when polymorphic uniform(boxed) representation.
  - explicit coercion between specialized and uniform representation.
  - give translation rule that inserts necessary coercions.

□ Coercion

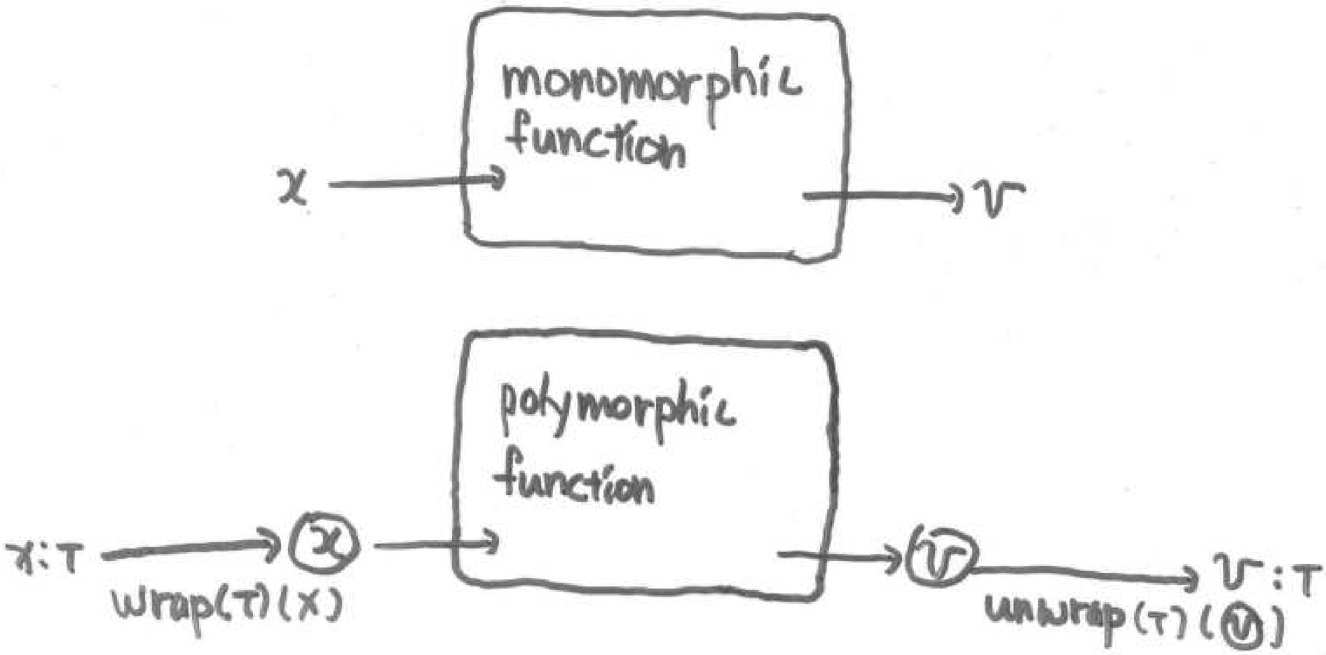
- $\texttt{wrap}(\tau)(\texttt{x})$:
  converts x in specilized repsresentation for τ to uniform representation

- $\texttt{unwrap}(\tau)(\texttt{x})$:
  converts x in uniform repsresentation to specialized representation for τ

- polymorphic functions are compiled assuming uniform represention

- when calling polymorphic function, coercions take place before and after calling

□ Coercion


monomorphic function
x
v
polymorphic function
x : T
x
wrap(T)(x)
v
v : T
unwrap(T)(v)

□ Example

- $\forall\alpha.\alpha \to \alpha \times \alpha$

```
let make_pair = λx.(x,x)
in make_pair(3.14)
```

$$\Downarrow$$

```
let x = make_pair(wrap(float)(3.14))
in (unwrap(float)(fst(x)),unwrap(float)(snd(x)))
```

- $\forall\alpha,\beta.(\alpha \to \beta) \to \alpha \times \alpha \to \beta \times \beta$

```
let map_pair = λf.λx.(f(fst(x)),f(snd(x)))
in map_pair( int_of_float )(3.14,2.718)
```

$$\Downarrow$$

```
let y = map_pair
  (λx.wrap(int)(int_of_float(unwrap(float)(x))))
  (wrap(float)(3.14), wrap(float)(2.718))
in (unwrap(float)(fst(y)), unwrap(float)(snd(y))))
```

□ The language

$$
\begin{array}{rcl}
a & ::= & i \mid f \mid x \mid \lambda x.a \\
 & \mid & \texttt{let}\ x = a\ \texttt{in}\ a \mid a\ a \\
 & \mid & (a, a) \mid \texttt{fst}(a) \mid \texttt{snd}(a) \\
a' & ::= & i \mid f \mid x \mid \lambda x.a' \\
 & \mid & \texttt{let}\ x = a'\ \texttt{in}\ a' \mid a'\ a' \\
 & \mid & (a', a') \mid \texttt{fst}(a') \mid \texttt{snd}(a') \\
 & \mid & \texttt{wrap}(\tau)(a') \mid \texttt{unwrap}(\tau)(a') \\
\tau & ::= & \alpha \mid \texttt{int} \mid \texttt{float} \mid \tau \to \tau \mid \tau \times \tau \\
\sigma & ::= & \forall a_1 \ldots a_n.\tau
\end{array}
$$

□ Translation

$$\frac{E(x) = \forall \alpha_1, \ldots, \alpha_n.\tau \quad Dom(\rho) \subseteq \{\alpha_1, \ldots, \alpha_n\}}{E \vdash x : \rho(\tau) \Rightarrow S_\rho(x : \tau)}$$

$$E \vdash i : \texttt{int} \Rightarrow i \qquad\qquad E \vdash f : \texttt{float} \Rightarrow f$$

$$\frac{E[x : \tau_1] \vdash a : \tau_2 \Rightarrow a'}{E \vdash \lambda x.a : \tau_1 \to \tau_2 \Rightarrow \lambda x.a'}$$

$$\frac{E \vdash a_2 : \tau_1 \to \tau_2 \Rightarrow a'_2 \quad E \vdash a_1 : \tau_1 \Rightarrow a'_1}{E \vdash a_2\ a_1 : \tau_2 \Rightarrow a'_2\ a'_1}$$

$$\frac{E \vdash a_1 : \tau_1 \Rightarrow a'_1 \quad E[x : Gen(\tau_1, E)] \vdash a_2 : \tau_2 \Rightarrow a'_2}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2 \Rightarrow \texttt{let}\ x = a'_1\ \texttt{in}\ a'_2}$$

$$\frac{E \vdash a_1 : \tau_1 \Rightarrow a'_1 \quad E \vdash a_2 : \tau_2 \Rightarrow a'_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2 \Rightarrow (a'_1, a'_2)}$$

$$\frac{E \vdash a : \tau_1 \times \tau_2 \Rightarrow a'}{E \vdash \texttt{fst}(a) : \tau_1 \Rightarrow \texttt{fst}(a')} \qquad\qquad \frac{E \vdash a : \tau_1 \times \tau_2 \Rightarrow a'}{E \vdash \texttt{snd}(a) : \tau_2 \Rightarrow \texttt{snd}(a')}$$

□ Putting coercions

- $S_\rho(a' : \tau)$:
  a' is represented according to τ,
  converts it to be used with type $\rho(\tau)$.

$$
\begin{array}{rcl}
S_\rho(a' : \alpha) & = & \mathtt{unwrap}(\rho(\alpha))(a') \\
S_\rho(a' : \mathtt{int}) & = & a' \\
S_\rho(a' : \mathtt{float}) & = & a' \\
S_\rho(a' : \tau_1 \times \tau_2) & = & \mathtt{let}\ x = a'\ \mathtt{in} \\
 & & (S_\rho(\mathtt{fst}(x) : \tau_1), S_\rho(\mathtt{snd}(x) : \tau_2)) \\
S_\rho(a' : \tau_1 \to \tau_2) & = & \lambda x. S_\rho(a'\ G_\rho(x : \tau_1) : \tau_2) \\
 & & \text{where } x \text{ is not free in } a'
\end{array}
$$

- $G_\rho(a' : \tau)$:
  a' is represented according to $\rho(\tau)$,
  converts it to be used with type τ.

$$
\begin{array}{rcl}
G_\rho(a' : \alpha) & = & \mathtt{wrap}(\rho(\alpha))(a') \\
G_\rho(a' : \mathtt{int}) & = & a' \\
G_\rho(a' : \mathtt{float}) & = & a' \\
G_\rho(a' : \tau_1 \times \tau_2) & = & \mathtt{let}\ x = a'\ \mathtt{in} \\
 & & (G_\rho(\mathtt{fst}(x) : \tau_1), G_\rho(\mathtt{snd}(x) : \tau_2)) \\
G_\rho(a' : \tau_1 \to \tau_2) & = & \lambda x. G_\rho(a'\ S_\rho(x : \tau_1) : \tau_2) \\
 & & \text{where } x \text{ is not free in } a'
\end{array}
$$

□ Experimental Result

Test	Gallium	Gallium0	SML-NJ	Caml	cc -02
1. Takeushi	3.00	5.09	4.47	34.0	1.96
2. Integral	0.80	2.83	8.46	15.2	0.40
3. List summation	3.60	3.45	5.12	7.90	
4. Sieve	1.00	0.94	2.31	5.74	
5. Boyer	1.80	2.76	3.60	14.6	
6. Knuth-Bendix completion	0.90	0.98	1.11	12.4	
7. Church integrals	6.58	2.40	2.90	16.1	
8. Solitaire	5.84	10.8	12.6	17.1	0.70

Test	What is tested
1. Takeushi	function calls(3 args), integer arithmetic
2. Integral	floating-point arithmetic, loops
3. List summation	list processing, integer arithmetic
4. Sieve	list processing, functionals, polymorphism
5. Boyer	term processing, function calls
6. Knuth-Bendix completion	term processing, functionals, polymorphism
7. Church integrals	functionals polymorphism
8. Solitaire	function calls, arrays, loops

□ Conclusion

- Simple and intuitive method to use unboxed objects for monomorphic functions.
- Optimal representation for monomorphic objects
- Translation rules that depends only on static type information.
- In cost, polymorphic functions introduce coercion overhead compared to method of boxing every object.