Tag-free Garbage Collection Using Explicit Type Parameters

Andrew Tolmach (94)

Presented by

Hahm Jaegyoon

Dec. 2, 2000

□ Introduction

- Tagged GC(Garbage Collection).
- Tag-Free GC.
  - Compiler has the type information.
  - "Two-finger" collection.
- GC using abstracted polymorphic type.
- GC using explicit type parameters.
  - explicitly-typed 2nd-order λ-calculus.

□ Language

$$
\begin{array}{llcl}
\mathit{Expr} & e & ::= & c \mid x \mid e_1 e_2 \mid \lambda x.e_1 \mid \\
 & & & \texttt{let}\ x \texttt{=} e_1\ \texttt{in}\ e_2 \mid \\
 & & & \textbf{letrec}\ f = \lambda x.e_1\ \textbf{in}\ e_2 \mid \\
 & & & \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \mid \\
 & & & e_1; e_2 \\
 & & & \\
\mathit{Type} & \tau & ::= & \alpha \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \mid \tau\ \mathit{list} \mid \\
 & & & \mathit{unit} \mid \mathit{int} \mid \mathit{float} \mid \mathit{bool} \mid .... \\
 & & & \\
\mathit{TypeScheme} & \sigma & ::= & \forall \alpha_1 ... \alpha_n . \tau
\end{array}
$$

□ Type rules

(CONST) $$\frac{TypeOf(c) \succ \tau}{\Gamma \vdash c \colon \tau}$$

(VAR) $$\frac{\Gamma(x) \succ \tau}{\Gamma \vdash x \colon \tau}$$

(ABS) $$\frac{\Gamma + x \colon \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

(APP) $$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

(LET) $$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x \colon \mathit{Clos}_\Gamma(\tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x \texttt{=} e_1\ \texttt{in}\ e_2 : \tau_2}$$

(LETREC) $$\frac{\Gamma + f \colon \tau_2 \to \tau_1, x \colon \tau_2 \vdash e_1 \colon \tau_1 \quad \Gamma + f \colon \mathit{Clos}_\Gamma(\tau_2 \to \tau_1) \vdash e_2 \colon \tau}{\Gamma \vdash \textbf{letrec}\ f = \lambda x.e_1\ \textbf{in}\ e_2 : \tau}$$

□ Type rules

(IF) $$\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}$$

(SEQ) $$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \; ; \; e_2 : \tau}$$

□ Explicit parameterization

$$
\begin{array}{lll}
\mathit{Expr} \quad e & ::= & c\{\tau_1, ..., \tau_n\} \mid x\{\tau_1, ..., \tau_n\} \mid \\
 & & e_1 e_2 \mid \lambda x : \tau . e_1 \mid \\
 & & \texttt{let } x : \forall \alpha_1 ... \alpha_n . \tau \texttt{ = } \Lambda(\alpha_1, ..., \alpha_n). e_1 \texttt{ in } e_2 \mid \\
 & & \textbf{letrec } f : \forall \alpha_1 ... \alpha_n . \tau_1 \rightarrow \tau_2 = \Lambda(\alpha_1, ..., \alpha_n). \lambda x : \tau_1 . e_1 \textbf{ in } e_2 \mid \\
 & & \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \mid \\
 & & e_1 ; e_2
\end{array}
$$

□ Example

<a simple polymorphic program>

```
let f = λ x.(x,x)
in let g = λ (y,z).(f y, f z)
   in g (3, 2.0);
      g (1.0, 0)
```

<2nd-order translation>

let f:$\forall\alpha.\alpha \to (\alpha \times \alpha) = \Lambda\alpha.\lambda\ x : \alpha.(x, x)$
in let g:$\forall\beta,\gamma.(\beta \times \gamma) \to ((\beta \times \beta) \times (\gamma \times \gamma)) = \Lambda(\beta,\gamma).\lambda(y : \beta, z : \gamma).(f\{\beta\}y,\ \ f\{\gamma\}z)$
   in g{int,float} (3, 2.0);
      g{float,int} (1.0, 0)

□ Runtime type descriptions

```
datatype rttd =
  Integer                     (* simple integer *)
| Float                       (* simple float; only legal within list *)
| Static_pointer              (* pointer to non-gc'ed memory (e.g., code) *)
| Closure_pointer             (* pointer to self-describing closure *)
| Record of rtrd              (* pointer to heap record *)
| Type_var of int             (* index into type environment *)


and rtrd =
| Sum of rtrd vector          (* list of variant types; indexed by header tag *)
| Product of rttd vector      (* list of recored fields *)


datatype type_env =
  Tenv of {instance:rttd vector, mentioner:type_env}
| Empty_tenv
```

□ Instantiation

- How to instantiate such expression at runtime?

- simple method
  - To have the compiler generate code to construct, at runtime, a completely fresh description of the instantiated type.
  - There is no need to polymorphic types at runtime.
  - Time/space inefficient.

- lazy method
  - To represent polymorphic types in parametric form and to interpret them relative to an instantiation vector of types.
  - Now simply to form association between an rttp Type_var(i) and an appropriate instantiation vector is okay.

□ Example

```
let f = λftenv.λx:Type_var(0).(x,x)
in let g = λgtenv.λ(y:Type_var(0),z:Type_var(1)).
           (f Tenv{instance=[|Type_var(0)|], mentioner=gtenv} y,
            f Tenv{instance=[|Type_var(1)|], mentioner=gtenv} z)
   in g Tenv{instance=[|Integer,Float|], mentioner=Empty_tenv} (3, 2.0);
      g Tenv{instance=[|Float,Integer|], mentioner=Empty_tenv} (1.0, 0)
```

□ Benchmark

Benchmark	Ratios (New Collector/Ordinary Gallium)				
	Record Allocated	Bytes Allocated	Execution Time		
			Small Heap	Medium Heap	Large Heap
sieve	0.8	0.8	1.0	1.0	1.0
sumlist	0.5	0.6	0.4	0.6	0.6
sumlistf	1.0	0.8	2.6	1.0	0.9
pair	1.0	1.0	1.1	1.0	1.0
pairf	1.2	1.1	0.7	0.8	1.2
pclist	1.0	0.9	1.9	1.1	1.0

□ check point

- What's key idea?
- Do we really need it?
- Can we improve it?
- Do we understand it?
- Is anyone skeptical about it?
- What are alternatives?