Referências Bibliográficas

[1] The Java hall of shame. http://www.cs.arizona.edu/sumatra/hallofshame/, 1996.

[2] Critical Mass Modula-3 (CM3) 5.1 Documentation. http://www.elego-soft.com/cm3/doc, 2001.

[3] Tuning garbage collection with the 1.4.2 Java Virtual Machine. http://java.sun.com/docs/hotspot/gc1.4.2/, 2003. Sun Developer Network.

[4] Perl 5.8.0 Documentation. http://www.perldoc.com, 2004.

[5] The Standard ML of New Jersey Structure Documentation. http://www.smlnj.org/doc/SMLofNJ, 2004.

[6] The Glasgow Haskell Compiler User's Guide, version 6.2. http://www.haskell.org/ghc, 2004.

[7] C# Language Specification. http://msdn.microsoft.com/library/default.asp?-url=/library/en-us/csspec/html/CSharpSpecStart.asp, 2005.

[8] Dolphin Smalltalk Patterns. http://www.object-arts.com/EducationCentre/Patterns/Patterns.htm, mar 2005.

[9] NET Framework Developer's Guide - Implementing a Dispose method. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/-html/cpconimplementingdisposemethod.asp, Mar. 2005.

[10] TIOBE Programming Community Index. http://www.tiobe.com/, jun 2005.

[11] Achour, M., Betz, F., Dovgal, A., Lopes, N., Olson, P., Richter, G., Seguy, D., and Vrana, J. PHP Manual. http://www.php.net/manual/en/, mar 2005.

[12] Atkins, M. C., and Nackman, L. R. The active deallocation of objects in object-oriented systems. *Software – Practice and Experience 18*, 11 (Nov. 1988), 1073–1089.

[13] Barendsen, E., and Smetsers, S. Conventional and uniqueness typing in graph rewrite systems. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science* (London, UK, 1993), Springer-Verlag, pp. 41–51.

[14] Bartlett, J. F. A generational compacting garbage collector for C++. In *ECOOP/OOPSLA 1990 Workshop on Garbage Collection in Object-Oriented Systems* (Ottawa, Canada, 1990).

[15] Blackburn, S. M., Cheng, P., and McKinley, M. Myths and reality: The performance impact of garbage collection. In *Sigmetrics - Performance 2004, Joint International Conference on Measurement and Modeling of Computer Systems* (June 2004), ACM Press, pp. 25–36.

[16] Bloch, J. *Effective Java Programming Language Guide*. The Java Series. Addison-Wesley, 2001.

[17] Boehm, H.-J. Space efficient conservative garbage collection. In *Proceedings of the 1993 ACM Conference on Programming Languages Design and Implementation* (Albuquerque, NM, June 1993), vol. 28(6) of *ACM SIGPLAN Notices*, ACM Press, pp. 197–206.

[18] Boehm, H.-J. Destructors, finalizers, and synchronization. In *Proceedings of the 2003 ACM Symposium on Principles of Programming Languages* (June 2003), ACM Press, pp. 262–272.

[19] Boehm, H.-J., and Weiser, M. Garbage collection in an uncooperative environment. *Software—Practice and Experience 18*, 9 (1988), 807–820.

[20] Brownbridge, D. R. Cyclic reference counting for combinator machines. In *Proceedings of Functional Programming Languages and Computer Architecture* (1985), vol. 201 of *LNCS*, Springer-Verlag.

[21] Byrne, S. B., Bonzini, P., and Valencia, A. GNU Smalltalk User's Guide. http://www.gnu.org/software/smalltalk/gst-manual/gst.html, August 2002.

[22] Cheney, C. J. A nonrecursive list compacting algorithm. *Communications of the ACM 13*, 11 (Nov. 1970), 677–678.

[23] Chirimar, J., Gunter, C. A., and Riecke, J. G. Proving memory management invariants for a language based on linear logic. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming* (1992), ACM Press, pp. 139–150.

[24] Christiansen, T., and Torkington, N. *Perl Cookbook*, second ed. O'Reilly, Aug. 2003.

[25] Collins, G. E. A method for overlapping and erasure of lists. *Communications of the ACM 3*, 12 (Dec. 1960), 655–657.

[26] Demers, A., Weiser, M., Hayes, B., Boehm, B., Bobrow, D., and Shenker, S. Combining generational and conservative garbage collection: Framework and implementations. In *Proceedings of the 1990 ACM Symposium on Principles of Programming Languages* (1990), ACM Press, pp. 261–269.

[27] Dybvig, R. K., Bruggeman, C., and Eby, D. Guardians in a generation-based garbage collector. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation* (1993), ACM Press, pp. 207–216.

[28] Elsman, M. Garbage collection safety for region-based memory management. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (2003), ACM Press, pp. 123–134.

[29] Fenichel, R. R., and Yochelson, J. C. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM 12*, 11 (Nov. 1969), 611–612.

[30] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented software*. Addison Wesley, 1995.

[31] Haible, B., and Steingold, S. Implementation notes for GNU CLISP. http://clisp.cons.org/impnotes/index.html, 2004.

[32] Hanson, C. MIT Scheme Reference. http://www.swiss.ai.mit.edu/projects/-scheme/documentation/scheme.html, June 2002.

[33] Harry, B. Resource management. http://discuss.develop.com/archives/-wa.exe?A2=ind0010A&L=DOTNET&P=R28572, Oct. 2000. DOTNET Discussion List.

[34] Hayes, B. Finalization in the collector interface. In *Proceedings of the 1992 International Workshop on Memory Management* (Sept. 1992), vol. 637 of *LNCS*, Springer-Verlag, pp. 277–298.

[35] Hayes, B. Ephemerons: A new finalization mechanism. In *Proceedings of the 1997 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (1997), ACM Press, pp. 176–183.

[36] Horning, J., Kalsow, B., McJones, P., and Nelson, G. Some useful Modula-3 interfaces. Technical Report 113, Systems Research Center, Digital Equipment Corp, Dec. 1993.

[37] Huang, X., Blackburn, S. M., McKinley, K. S., Moss, J. E. B., Wang, Z., and Cheng, P. The garbage collection advantage: improving program locality. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2004), ACM Press, pp. 69–80.

[38] Hudak, P. A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (1986), ACM Press, pp. 351–363.

[39] Hunter, R., and Krishnamurthi, S. A model of garbage collection for OO languages. In *10th International Workshop on Foundations of Object-Oriented Languages (FOOL10)* (2003).

[40] Ierusalimschy, R. *Programming in Lua*. Lua.org, 2003.

[41] Jones, R., and Lins, R. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, New York, NY, USA, 1996.

[42] Jones, S. P., Marlow, S., and Elliott, C. Stretching the storage manager: Weak pointers and stable names in Haskell. In *Implementation of Functional Languages, 11th International Workshop* (2000), vol. 1868 of *LNCS*, Springer-Verlag, pp. 37–58.

[43] Joy, B., Steele, G., Gosling, J., and Bracha, G. *The Java Language Specification*, second ed. Addison-Wesley, June 2000.

[44] Leal, M. A., and Ierusalimschy, R. A formal semantics for finalizers. *Journal of Universal Computer Science - JUCS 11*, 7 (Sept. 2005), 1198–1214.

[45] Leroy, X. The Objective Caml System Documentation and User's Manual. http://caml.inria.fr/ocaml/htmlman/, July 2004.

[46] Lieberman, H., and Hewitt, C. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM 26*, 6 (June 1983), 419–429.

[47] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM 3*, 4 (Nov. 1960), 184–195.

[48] Merizen, F., Zendra, O., and Colnet, D. Designing efficient and safe non-strong references in Eiffel with parametric types. Research Report A04-R-149, INRIA Lorraine / LORIA UMR 7503, Sept. 2004.

[49] Milner, R. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.

[50] Morrisett, G., Felleisen, M., and Harper, R. Abstract models of memory management. In *Proceedings of the 1995 ACM Conference on Functional Programming Languages and Computer Architecture* (1995), ACM Press, pp. 66–77.

[51] Morrisett, G., and Harper, R. Semantics of memory management for polymorphic languages. In *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1998, pp. 175–226.

[52] Pawlam, M. Reference objects and garbage collection. Sun Developer Connection, August 1998.

[53] Plotkin, G. D. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, Sept. 1981.

[54] Rees, J. A., Adams, N. I., and Meehan, J. R. *The T Manual*, fourth ed. Yale University Computer Science Department, Jan. 1984.

[55] Reinhold, M. B. Cache performance of garbage-collected programs. In *Proceedings of SIGPLAN'94 Conference on Programming Languages Design and Implementation* (Orlando, FL, June 1994), vol. 29 of *ACM SIGPLAN Notices*, ACM Press, pp. 206–217.

[56] Richter, J. *Applied Microsoft .NET Framework Programming*. Microsoft Press, 2002.

[57] Rovner, P. On adding garbage collection and runtime types to a strongly-typed, statically-checked, concurrent language. Technical Report CSL–84–7, Xerox PARC, Palo Alto, CA, July 1985.

[58] Schwartz, R. L., and Melliar-Smith, P. M. The finalization operation for abstract types. In *ICSE '81: Proceedings of the 5th International Conference on Software Engineering* (1981), IEEE Press, pp. 273–282.

[59] Scott, M. L. *Programming Languages Pragmatics*. Morgan-Kaufmann, 2000.

[60] Stroustrup, B. *The Design and Evolution of C++*, first ed. Addison-Wesley, Mar. 1994.

[61] Stroustrup, B. *The C++ Programming Language*, third ed. Addison-Wesley, June 1997.

[62] Sun Microsystems. *Java 2 Platform, Standard Edition, v 1.4.0: API Specification*, 2002.

[63] Thomas, D., Fowler, C., and Hunt, A. *Programming Ruby: The Pragmatic Programmers' Guide*, second ed. Pragmatic Bookshelf, Oct. 2004.

[64] Ungar, D. Generation Scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices 19*, 5 (May 1984), 157–167.

[65] van Rossum, G. *Python Library Reference*, 2.3.3 ed., Dec. 2003.

[66] Venners, B. Object finalization and cleanup. *JavaWorld* (June 1998).

[67] Wadler, P. Monads for functional programming. In *Advanced Functional Programming*, vol. 925 of *LNCS*. Springer-Verlag, 1995.

[68] Walker, D., Crary, K., and Morrisett, G. Typed memory management via static capabilities. *ACM Trans. Program. Lang. Syst. 22*, 4 (2000), 701–771.

[69] Wilson, P. R. Uniprocessor garbage collection techniques. In *Proceedings of the 1992 International Workshop on Memory Management* (Sept. 1992), vol. 637 of *LNCS*, Springer-Verlag, pp. 1–42.

[70] Xerox Palo Alto Research Center (PARC). *InterLISP Reference Manual*. Palo Alto, CA, Oct. 1985.

[71] Zorn, B. The measured cost of garbage collection. Tech. Rep. CU–CS–573–92, University of Colorado at Boulder, Apr. 1992.

A
Tabelas Fracas

Neste apêndice apresentamos a descrição completa de duas implementações distintas de tabelas fracas com valores fortes. Na Seção A.1 implementamos tabelas fracas usando referências fracas ordinárias, conforme discutido na Seção 5.4.2. Na Seção A.2 modificamos esta implementação para usar ephemerons.

A.1
Tabelas Fracas com Referências Fracas

Iremos implementar arrays associativos (tabelas) como listas de pares chave/valor. A forma mais simples de representar um par é usando uma expressão condicional, onde cada elemento do par ocupa uma das cláusulas. Para criar um par definimos o operador $pair$. As operações de projeção (π_i) são definidas como aplicações com valores pré-definidos (*true* e *false*).

$$\begin{aligned} pair\ v_1\ v_2 &= \lambda x_i.(x_i?\ v_1 : v_2) \\ \pi_1\ v &= v(\lambda x_i.nil) \\ \pi_2\ v &= v\ nil \end{aligned}$$

Uma lista é representada por um par cujo primeiro elemento contém o primeiro elemento da lista, e o segundo elemento representa o resto da lista (o tail). Uma lista vazia é representada pelo valor nil.

$$\begin{aligned} cons\ v_1\ v_2 &= pair\ v_1\ v_2 \\ head\ v &= v?\ \pi_1\ v\ : nil \\ tail\ v &= v?\ \pi_2\ v\ : nil \end{aligned}$$

A operação $cons$ constrói uma nova lista concatenando um valor a uma lista. As operações $head$ e $tail$ obtém respectivamente o primeiro elemento de uma lista, e uma lista contendo todos os elementos da lista original menos o primeiro elemento.

Caso a lista seja vazia, estas operações retornam nil.

Tabelas são representadas por listas de pares chave/valor, e tabelas vazias são representadas por listas vazias (nil). Para inserir e recuperar elementos de uma tabela usamos as operações get e set, que são definidas como[1]:

$$\begin{aligned} get\ t_i\ k_i \quad = \quad & (\lambda x_t.\lambda x_k.(x_t?\ (\textbf{let}\ x_i = head\ x_t\ \textbf{in} \\ & \quad (\pi_1\ x_i) = x_k?\ \pi_2\ x_i : get\ (tail\ x_t)\ x_k \\ & \quad) : nil) \\ &)t_i k_i \end{aligned}$$

$$\begin{aligned} set\ t_i\ k_i\ v_i \quad = \quad & (\lambda x_t.\lambda x_k.\lambda x_v.(x_t?\ (\textbf{let}\ x_i = head\ x_t\ \textbf{in} \\ & \quad (\pi_1\ x_i) = x_k?\ cons\ (pair\ x_k\ x_v)\ (tail\ x_t) \\ & \quad : cons\ x_i\ (set\ (tail\ x_t)\ x_k\ x_v) \\ & \quad) : cons\ (pair\ x_k\ x_v)\ nil) \\ &)t_i k_i v_i \end{aligned}$$

Para implementar tabelas fracas com chaves fracas e valores fortes basta encapsular as chaves em referências fracas antes de compor o par chave/valor. As operações get_w

[1]Tanto *get* quanto *set* são operações recursivas, e portanto precisam ser definidas formalmente usando um operador de ponto-fixo. No entanto, para facilitar a compreensão do leitor iremos apresentar as definições desta seção empregando uma notação tradicional, isto é, usando o nome da própria função na expressão recursiva que a define.

O ponto fixo de um função f é qualquer ponto x tal que $fx = x$. No λ-cálculo qualquer função possui um ponto fixo, o qual pode ser calculado através de funções conhecidas como combinadores de ponto-fixo. Por definição o combinador de ponto-fixo é qualquer função fix que satisfaz a relação $f\ fix\ f = fix\ f$.

Apenas como ilustração, apresentamos abaixo a definição mais rigorosa de *get* empregando um combinador de ponto-fixo.

$$\begin{aligned} get^* \quad = \quad & \lambda f_i.\lambda x_t.\lambda x_k.(x_t?\ \textbf{let}\ x_i = head\ x_t\ \textbf{in} \\ & \quad (\pi_1\ x_i) = x_k?\ \pi_2\ x_i : f_i(tail\ x_t)x_k \\ & \quad : nil) \\ get\ t_i\ k_i \quad = \quad & (fix\ get^*)t_i k_i \end{aligned}$$

e set_w, que recuperam e inserem elementos em tabelas fracas, são definidas como:

$$\begin{aligned} get_w\ t_i\ k_i \;=\; & (\lambda x_t.\lambda x_k.(x_t?\ (\mathbf{let}\ x_i = head\ x_t\ \mathbf{in} \\ & \quad (!\pi_1\ x_i) = x_k?\ (\pi_2\ x_i) : get_w\ (tail\ x_t)\ x_k \\ & \quad) : nil) \\ &)t_i k_i \end{aligned}$$

$$\begin{aligned} set_w\ t_i\ k_i\ v_i \;=\; & (\lambda x_t.\lambda x_k.\lambda x_v.(x_t?\ (\mathbf{let}\ x_i = head\ x_t\ \mathbf{in} \\ & \quad (!\pi_1\ x_i) = x_k?\ cons\ (pair\ \pi_1\ x_i\ x_v)\ tail\ x_t \\ & \quad : cons\ x_i\ (set_w\ (tail\ x_t)\ x_k\ x_v) \\ & \quad) : \mathbf{let}\ x_j = \mathbf{new}\ \mathbf{in}\ (\\ & \quad x_j :\overset{w}{=} x_k; \\ & \quad cons\ (pair\ x_j\ x_v)\ nil)) \\ &)t_i k_i v_i \end{aligned}$$

Para limpar as tabelas implicitamente modificamos a operação set_w para checar e descartar os elementos com chaves iguais a nil. Esta operação (set^*_w) é definida como

$$set^*_w\ t_i\ k_i\ v \;=\; set_w\ clear\ t_i\ k_i\ v$$

onde $clear$ é definido por

$$\begin{aligned} clear\ t_i \;=\; & (\lambda x_t.(x_t?\ (\mathbf{let}\ x_i = head\ x_t\ \mathbf{in} \\ & \quad !\pi_1\ x_i?\ cons\ x_i\ (clear\ (tail\ x_t)) \\ & \quad : clear\ (tail\ x_t) \\ & \quad) : nil) \\ &)t_i \end{aligned}$$

A.2
Tabelas Fracas com Ephemerons

Além de permitir a coleta de objetos associados a ciclos internos, ephemerons simplificam a implementação de tabelas fracas. Nesta seção descrevemos uma nova implementação dessa estrutura de dados usando ephemerons. Listas continuam sendo

implementadas com pares, mas as tabelas fracas passam a ser representadas como listas de ephemerons.

$$
\begin{aligned}
get_w\ t_i\ k_i \;=\; & (\lambda x_t.\lambda x_k.(x_t?\ (\mathbf{let}\ x_i = head\ x_t\ \mathbf{in} \\
& \quad (\pi_1\ x_i) = x_k?\ \pi_2\ x_i : get_w\ (tail\ x_t)\ x_k \\
& \quad) : nil) \\
&)t_i k_i
\end{aligned}
$$

$$
\begin{aligned}
set_w\ t_i\ k_i\ v_i \;=\; & (\lambda x_t.\lambda x_k.\lambda x_v.(x_t?\ (\mathbf{let}\ x_i = head\ x_t\ \mathbf{in} \\
& \quad (\pi_1\ x_i) = x_k?\ cons\ (\phi\ x_k\ x_v)\ tail\ x_t \\
& \quad\ : cons\ x_i\ (set_w\ (tail\ x_t)\ x_k\ x_v) \\
& \quad) : cons\ (\phi\ x_k\ x_v)\ nil) \\
&)t_i k_i v_i
\end{aligned}
$$

Para limpar as tabelas implicitamente usamos a operação (set^*_w) definida na última seção.

$$
set^*_w\ t_i\ k_i\ v \;=\; set_w\ clear\ t_i\ k_i\ v
$$

A operação $clear$ é modificada da seguinte forma.

$$
\begin{aligned}
clear\ t_i \;=\; & (\lambda x_t.(x_t?\ (\mathbf{let}\ x_i = head\ x_t\ \mathbf{in} \\
& \quad \pi_1\ x_i?\ cons\ x_i\ (clear\ (tail\ x_t)) \\
& \quad\ : clear\ (tail\ x_t) \\
& \quad) : nil) \\
&)t_i
\end{aligned}
$$