Temas para trabalhos de Implementação de Linguagens Funcionais

Pedro Vasconcelos

6 de Maio de 2020

1. O trabalho consiste no aprofundamento de um tema abordado nas aulas e implementação num protótipo.
2. O trabalho é **individual**; situações de manifesto plágio (cópia de trabalhos de outros alunos ou da Internet) será penalizado com a perda da classificação em *todos* os trabalhos da disciplina.
3. Os alunos devem entregar o código de um protótipo que extende uma implementação do compilador SECD apresentado nas aulas.
4. O objectivo do trabalho é demonstrar a compreensão dos conceitos de implementação de linguagens estudados. A utilização de uma boa estrutura de código, clareza de código e inclusão de resultados experimentais são também valorizados.
5. Os aluno devem comunicar ao docente o tema de trabalho escolhido até 16 de maio. A entrega do trabalho deverá ser feita até ao final das aulas (29 de maio).

1 Otimização de código da SECD

Estude algumas optimizações de código virtual da máquina SECD e implemente-as no compilador e interpretador apresentados nas aulas. Algumas das técnicas a considerar (e.g. Seção 7.8 de [1]):

1. Evitar construção desnecessária de *closures* (exemplo: em expressões *let*);
2. Simplificar instruções condicionais no final duma função, por exemplo, sequências de código "SEL [...JOIN] [...JOIN], RTN";
3. Optimização de última chamada, por exemplo sequências de código "AP RTN";
4. Valorização: combinar o *stack* e *dump* numa única pilha;
5. Valorização: tratamento de funções com múltiplos argumentos.

6. Valorização: optimização de recursão de cauda (*tail recursion optimization*).

Sugere-se que implemente estas otimizações como funções independentes de forma a permitir a sua combinação modular. Tenha ainda o cuidado de incluir exemplos simples que ilustrem os efeitos das optimizações.

2 Implementar tuplos, variantes e *records*

Pretende-se que implemente as extensões ao compilador e interpretador da SECD para tuplos, variantes e *records* [3].

1. Extender a sintaxe abstracta com pares, construtores e expressões-*case* para variantes;
2. Extender o compilador e interpretador da máquina SECD para essas novas formas;
3. Valorização: implementar constructores e selectores para *records*;
4. Valorização: implementar o esquema de tradução de *pattern-matching* genérico de Wadler [4].

Sugere-se ainda que ilustre estas extensões com funções recursivas simples sobre listas, por exemplo: *length*, *append*, *map*, *filter*

3 Implementar inferência de tipos

Pretende-se que estude e implemente em Haskell o algoritmo Damas-Milner de de inferência de tipos polimorficos para a linguagem FUN.

Considere como *tipos simples* um único tipo base int e funções $t_1 \to t_2$. Os *tipos polimórficos* são tipos simples quantificados universalmente em uma ou mais variáveis: $\forall\alpha\beta\ldots.t$. Para mais informação sobre este sistema de tipos e o algoritmo de inferência consulte [6, 7, 8].

Como valorização, pode implementar um programa completo que lê o programa na linguagem FUN, efetua o *parsing* para sintaxe abstrata [9], corre o algoritmo de inferência e que indique eventuais erros de tipos no programa.

4 Implementar um recolector de lixo

Pretende-se que implemente um alocador de memória e algoritmo de recolha de lixo no interpretador SECD apresentado nas aulas. A recolha de lixo deve ser executada quando se esgota a memoria *heap* (i.e. usada para *closures*).

Pode implementar um algoritmo "*naive mark-sweep*" ou de cópia de Cheney "*two-space*". Tenha o cuidado de manter a essência não-recursiva destes algoritmos.

Para mais informações consulte [10, 11, 12].

Referências

[1] *The Architecture of Symbolic Computers*, Peter M. Kogge. 1991, McGraw-Hill International.

[2] *Functional Programming: Application and Implementation*, Peter Henderson. 1980, Prentice-Hall International.

[3] *Programming languages*, notas de um curso de Mike Grant e Scott Smith. `http://www.cs.jhu.edu/~scott/pl/book`.

[4] *Efficient Compilation of Pattern Matching*, Philip Wadler. Capítulo 5 de *The Implementation of Functional Programming Languages*, Simon Peyton Jones, 1987.
`https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/`

[5] *SK Combinators*, *The Implementation of Functional Programming Languages*, Simon Peyton Jones, 1987, Capítulo 16.
`http://research.microsoft.com/~simonpj/Papers/papers.html`.

[6] *Types and Programming Languages*, Benjamin Pierce, The MIT Press, 2002. Capítulo 22 (*Type reconstruction*).

[7] *Polymorphic type checking*, Peter Hancock. Capítulo 8 de *The Implementation of Functional Programming Languages*, Simon Peyton Jones, 1987.

[8] *A type checker*, Peter Hancock. Capítulo 9 de *The Implementation of Functional Programming Languages*, Simon Peyton Jones, 1987.

[9] Biblioteca *Parsec*, `http://hackage.haskell.org/package/parsec`

[10] *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Richard Jones and Rafael Lins, Wiley and Sons (1996),

[11] *A nonrecursive list compacting algorithm*, CACM, vol. 13, n. 11, p. 677–678. New-York, USA, 1970. `http://doi.acm.org/10.1145/362790.362798`

[12] *Tracing garbage collection*, artigo da Wikipedia, `https://en.wikipedia.org/wiki/Tracing_garbage_collection`