1 Próximos passos

Para finalizar, vamos explorar aqui alguns tópicos que não puderam ser explorados no resto do texto, na tentativa de mostrar algumas direções em que é possível caminhar, além das já apresentadas. Começamos com *CLP(SMT)*.

Antes, convém começarmos com problemas *SAT* [*]. SAT o problema de determinar se existe uma interpretação (atribuição de valores *booleanos* a variáveis) que satisfaça uma fórmula *booleana*.Foi o primeiro problema a ser provado NP-completo mas, apesar disso, tem se mostrado de grande utilidade para diversos problemas práticos. Como é de se esperar da formulação, SAT é um problema central na ciência da computação. Assim, tem recebido muita atenção e conta com implementações eficientes para grandes classes de problemas específicos.

Uma instância de SAT é o de obter atribuições para as variáveis A, B e C tal que a fórmula $A \wedge B \vee (\neg C \vee (A \wedge \neg B))$.

SMT[†] é uma generalização do problema SAT para lidar com outras teorias, de forma mais geral, como sobre inteiros, listas, vetores, dentre outros. Semelhante ao caso já visto com problemas em CLP, um *solver* SMT terá, no caso geral, algorítmos especializados para lidar com teorias diferentes (lembre-se que uma das vantagens de CLP é facilitar essa unificação de diferentes métodos). Devido em grande parte a sua generalidade e eficiência, o uso de *solvers* SMT tem ganho grande popularidade para se lidar com problemas de diversas áreas, como de análise de programas[11], síntese de programas[1], verificação de hardware[4], automação de design eletrônico[4], segurança de computadores[9], IA, pesquisas operacionais (MAXSAT)[3] e biologia[10].

Existem atualmente vários *solvers* de SMT, dos quais os mais populares são Z3, criado pela Microsoft e CVC4[‡], mantido por um grupo de pesquisadores (em grande parte, da Universidade de Stanford). Neste texto, focaremos no Z3[§], um *solver* considerado estado-da-arte, que se tornou um projeto de código aberto em 2012 (veja [12] para detalhes).

Para que se tenha alguma ideia de como se usa um *solver* SMT, segue um exemplo (retirado de [8]) de código para Z3[¶]:

```
; declare a mutually recursive parametric datatype
(declare-datatypes (T) ((Tree leaf (node (value T)
                                         (children TreeList)))
                        (TreeList nil (cons (car Tree)
                                            (cdr TreeList)))))
(declare-const t1 (Tree Int))
```

[*]SAT vem de *satisfiability*.

[†]SMT vem de *satisfiability modulo theories*.

[‡]De *cooperating validity checking*.

[§]Explicaremos apenas o básico para que seja compreensível. Para mais informações, cheque, por exemplo, [13].

[¶]Vale notar, ele está expresso em *Sexps*, mas não em Scheme.

```
(declare-const t2 (Tree Bool))
; we must use the 'as' construct to distinguish the leaf
; (Tree Int) from leaf (Tree Bool)
(assert (not (= t1 (as leaf (Tree Int)))))
(assert (> (value t1) 20))
(assert (not (is-leaf t2)))
(assert (not (value t2)))
(check-sat)
(get-model)
```

que avalia para:

```
sat
(model
  (define-fun t2 () (Tree Bool)
    (node false (as nil (TreeList Bool))))
  (define-fun t1 () (Tree Int)
    (node 21 (as nil (TreeList Int)))))
```

O *sat* é de *satisfyable*, a resposta ao `(check-sat)` (seria *unsat*, se o modelo posto não fosse satisfazível).

1.1 Breve Introdução a SMT e sobre o problema da estratégia

Solvers SMT se baseiam fortemente em *solvers* SAT, baseados em DPLL *. Para SMT, temos o DPLL(T), , que é um formalismo para descrever como *solvers* da teoria T devem ser integrados com os *solvers* SAT.

Uma dificuldade é que, para um *solver* SMT de alta performance, parte importante da implementação não está no esquema formal do DPLL(T), mas sim na forma de heurísticas. Essas heurísticas são frequentemente projetadas para funcionar muito bem para algumas classe de problemas, tendendo a funcionar mal para outra classes. À medida que *solvers* SMT ganham a atenção de cientistas e engenheiros passou a se tornar claro que isso é um problema, porque há uma necessidade de grande controle sobre o *solver* para que ele se comporte de forma eficiente, o que significa expor até centenas de parâmetros para que usuários decidam quais heurísticas usar e como. Esse é um problema posto por Leonardo de Moura, um criador do Z3, no artigo [2]. Nesse artigo, ele define o problema da estratégia como o de prover ao usuário meios adequados de dirigir a busca. Mais em geral, ele define estratégia (neste contexto) como "*adaptations of general search mechanisms which reduce the search space by tailoring its exploration to a particular class of problems*"[2] e, assim, o problema da estratégia

*O algorítmo de Davis–Putnam–Logemann–Loveland é um algorítmo de busca completo, baseado em *backtracking*, para decidir a se dadas fórmulas de lógica proposicional em forma normal conjuntiva podem ser satisfeitas, introduzido em 1962, que ainda forma a base de *solvers* eficientes para SAT.

se traduziria como o de prover uma linguagem adequada para o usuário realizar sua busca.

Existe mais de uma forma de buscar resolver esse problema. A abordagem que tomaremos aqui é a adotada por Nada Amin e William Byrd, de juntar SMT com CLP, em um CLP(SMT), usando miniKanren como base. Para checar esse trabalho, cheque [5] ou [6] (esse último, na linguagem Clojure). Vale notar que, no momento da escrita deste texto (em Outubro de 2018), isso é trabalho em progresso.

A ideia é pensar no *solver* SMT como um implementador de restrições de baixo nível, com o qual o usuário interaje com o miniKanren, de forma mais abstrata.

Se você prestou atenção no caminho até aqui, deve ter adquirido algum conhecimento básico sobre programação por restrições e, assim esperamos, quando e se precisar de fazer uso de algumas das

Referências

[1] Beyene, Tewodros A., Swarat Chaudhuri, Corneliu Popeea, e Andrey Rybalchenko "Recursive games for compositional program synthesis." Working Conference on Verified Software: Theories, Tools, and Experiments, pp. 19-39. Springer, Cham, 2015.

[2] De Moura, Leonardo, and Grant Olney Passmore. "The strategy challenge in SMT solving." Automated Reasoning and Mathematics. Springer, Berlin, Heidelberg, 2013. 15-44.

[3] Li, Yi, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, Marsha Chechik. "Symbolic optimization with SMT solvers." In ACM SIGPLAN Notices, vol. 49, no. 1, pp. 607-618. ACM, 2014.

[4] Mukherjee, Rajdeep, Daniel Kroening, and Tom Melham "Hardware verification using software analyzers." In VLSI (ISVLSI), 2015 IEEE Computer Society Annual Symposium on, pp. 7-12. IEEE, 2015.

[5] CLP(SMT) miniKanren: https://github.com/namin/clpsmt-miniKanren

[6] Explorations in logic programming: https://github.com/namin/logically

[7] Trindade, Alessandro B., Lucas C. Cordeiro. "Applying SMT-based verification to hardware/software partitioning in embedded systems." Design Automation for Embedded Systems 20, no. 1 (2016): 1-19.

[8] Z3 Tutorial: https://rise4fun.com/z3/tutorial

[9] Vanegue, Julien, Sean Heelan, Rolf Rolles. "SMT Solvers in Software Security." WOOT 12 (2012): 9-22.

[10] Yordanov, Boyan, Christoph M. Wintersteiger, Youssef Hamadi, Hillel Kugler. "Z34Bio: An SMT-based framework for analyzing biological computation.". SMT'13 (2013).

[11] Zheng, Yunhui, Xiangyu Zhang, e Vijay Ganesh. "Z3-str: a z3-based string solver for web application analysis.", Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 114-124. ACM, 2013.

[12] Z3 se torna *open source*: http://leodemoura.github.io/blog/2012/10/02/open-z3.html

[13] Z3 wiki: https://github.com/Z3Prover/z3/wiki