Generating Propagation Complete Encodings in Haskell

Arunothia Marappan

Under Guidance of Graeme Gange

Submitted to Prof. Harald Sondergaard as a part of Research Internship (May'16 - July'16)

Abstract

In this project, we implemented the algorithm given in [1] for finding the propagation complete encoding for any given input encoding. Propagation complete encodings are those encodings who when fed to a SAT Solver can efficiently prune away all the unyielding paths that could otherwise occur while performing unit propagation. The implementation has been done in Haskell.

Computing and Information Systems
University of Melbourne

1 Implementation Details

1.1 Representing CNF

In our implementation, we represent a CNF encoding as a list of list of integers.

- The elements within the outer list are to be combined with an AND.
- The elements within the innner list are to be combined with an OR.
- Positive Integer i represents the literal X_i and negative integer i represents the literal $\neg X_i$.
- Integer 0 is not included in the representation and hence should be avoided.

1.2 Input Format

We assume the input to be a CNF encoding. The input arguments are

- An integer 'n', implying the vocabulary or variable set is X_1 to X_n
- A list of integers denoting the variables of interest.
- A list of list of integers that will represent the CNF of E that is being computed (It will be E_0 at the start)
- A list of list of integers that will represent the CNF of E_{ref} (which is the CNF for which equi-satisfiable formula is to be found)
- A minHeap (priority queue) that is required by the algorithm for looping. It is not a user input argument. It should by default be a singleton queue with the fully undefined partial assignment.

1.3 Output Format

The output is

- List of list of Integers representing the encoding that is equi-satisfiable to E_{ref} and is propagation complete.

1.4 Methodology in Haskell

PAValue is the defined Data Type that can either be PATrue, PAFalse or PAQuest representing the values True, False or Question that a partial assignment can assign to a variable. We define Partial Assignments as Maybe[PAValue], where Nothing represents the contradicting partial assignment or the bottom. The ordering amongst Partial Assignments are established as

- Nothing - the lowest bottom.
- The more undefined the partial assignment is the higher it is in the ordering.
- Partial assignments with same number of undefined variables are hard coded to follow a certain order of sorting .

We implemented it in two ways

- For the implementation of Priority Queues, we use Haskell Package Data.Heap. From the ordering of Partial Assignments mentioned, it is clear that we used maxHeap to mimic the priority queue desired. Sat Solver for the implementation has been taken from [2]. This method turned out to be very slow as it was unable to give output for large test cases.
- For the implementation of Priority Queues, we use Haskell Package Data.Set. Sat Solver for the implementation was the Haskell Package Picosat. This method was reasonable in its speed and all the results presented in the result section are using this method.

2 Examples

2.1 If-then-Else Gadget

$$x = if\ b\ then\ y\ else\ z$$

Which can be encoded as -

$$(b \longrightarrow (x \longleftrightarrow y)) \wedge (\neg b \longrightarrow (x \longleftrightarrow z))$$

Which in the CNF form corresponds to

$$(\neg b \vee \neg x \vee y) \wedge (\neg b \vee \neg y \vee x) \wedge (b \vee \neg x \vee z) \wedge (b \vee \neg z \vee x)$$

Hence, the input to our algorithm is

$$n = 4$$
$$E_0 = [[]]$$
$$E_{ref} = [[-2, -1, 3], [-2, -3, 1], [2, -1, 4], [2, 1, -4]]$$

The output as given by the algorithm is

$$E = [[-1, 2, 4], [-1, 3, 4], [-1, -2, 3], [1, -2, -3], [1, -3, -4], [1, 2, -4]]$$

The additional 2 clauses being

$$((y \wedge z) \longrightarrow x) \wedge ((\neg y \wedge \neg z) \longrightarrow \neg x)$$

2.2 Unsigned-less-than Gadget

In the following b_{in} stands for the input which marks whether the digits till here have been found to be smaller or not.

$$b_{out} = (x < y)\ or\ ((x == y) \wedge b_{in})$$

Which can be encoded as -

$$(b_{out} \longleftrightarrow (\neg x \wedge y) \vee ((x \longrightarrow y) \wedge b_{in})$$

Which in the CNF form corresponds to

$$(x \vee \neg y \vee b_{out}) \wedge (x \vee b_{in} \vee b_{out}) \wedge (\neg y \vee b_{in} \vee b_{out}) \wedge (\neg b_{out} \vee \neg x \vee y) \wedge (\neg b_{out} \vee \neg x \vee b_{in}) \wedge (\neg b_{out} \vee y \vee b_{in})$$

Hence, the input to our algorithm is

$$n = 4$$
$$E_0 = [[]]$$
$$E_{ref} = [[1, -2, 4], [1, 3, 4], [-2, 3, 4], [-4, -1, 2], [-4, -1, 3], [-4, 2, 3]]$$

The output as given by the algorithm is

$$E = [[1, -2, 4], [1, 3, 4], [1, 2, 3], [-1, 2, -4], [2, 3, -4], [-2, 3, 4], [-1, 3, -4], [-1, -2, 3]]$$

The additional 2 clauses being

$$((x \wedge y) \longrightarrow b_{in}) \wedge ((\neg x \wedge \neg y) \longrightarrow b_{in})$$

3 Results

Propagation Complete Encodings in Haskell		
Gadget Name	No. of Clauses in the PCE (as reported in [1])	No. of Clauses in the PCE (our implementation)
ult-gadget	6	6
slt-gadget	6	6
full-add	14	14
bc3to2	76	78
bc7to3	254	254
mult2	19	33
mult-const3	20	22
mult-const5	24	43
mult-const7	32	58
ult-6bit	158	158
add-3bit	96	96
add-4bit	336	336
bc3to2-3bit	1536	1584
mult-4bit	670	761

4 Conclusion

- The additional clauses in the complete encodings produced from our implementation is because we haven't removed the following redundancy -

$$x \longrightarrow y \text{ and } y \longrightarrow z$$
$$\text{then } x \longrightarrow z \text{ is redundant.}$$

- Because of this redundancy, the results of our implementation depends largely on the order in which the Partial assignments are being processed. 1.4

- The largest of the test cases takes over 10 minutes to give its output which is not as efficient as the C++ implementation reported in [1], but is good enough for a functional programming language like Haskell.

- A lot of functions could be improved to become much more efficient in our implementation.

References

[1] Martin Brain, Liana Hadarean, Daniel Kroening, and Ruben Martins. Automatic generation of propagation complete sat encodings. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *Lecture Notes in Computer Science*, pages 536–556. Springer, 2016.

[2] gatlin. sat.hs. https://gist.github.com/gatlin/1755736, 2012. [Online; accessed June-2016].