Kripke: a Countermodel Checker for Intuitionistic Propositional Logic

Christian Haack
Department of Computing and Information Sciences
Kansas State University
234 Nichols Hall
Manhattan, Kansas 66506, U.S.A.
haack@cis.ksu.edu

October 22, 1996

The file "kripke.sml" contains an ML-program that generates a Unix executable. The generated executable takes in a Kripke-tree and a sequent of propositional formulas and decides whether the Kripke-tree is a countermodel to the sequent, i.e. whether its root forces all assumptions of the sequent but not the conclusion. As another option it annotates the nodes of the tree with all forced subformulas of formulas that appear in the sequent. A third option finds a maximal node that forces all assumptions of the sequent but not the conclusion, if such a node exists. Finally, there is an option that lists all maximal nodes of that kind.

How to execute the executable

Let's assume you have called the generated executable `kripke`. In order to execute it type a command of the following format:

`% kripke [`*option*`]...` *filename sequent*

Possible options are:

`-m`	`--minimal`	use minimal logic
`-c`	`--check`	check whether the tree's root forces the sequent's assumptions but not its conclusion (default)
`-a`	`--annotate`	annotate tree with forced subformulas
`-f`	`--find`	find a maximal node that forces assumptions but not conclusion
`-l`	`--list`	list all maximal nodes that force assumptions but not conclusion
`-h`	`--help`	output help information and exit

The *sequent* is of the form

formula`, ...,` *formula* `=>` *formula*

Each *formula* is a propositional formula built up using parentheses, whitespace, propositional variables and the following symbols:

`f`	(falsity)		
`~`	(negation)	highest precedence	
`&`	(conjunction)	.	right associative
`|`	(disjunction)	.	right associative
`->`	(implication)	.	right associative
`<->`	(biimplication)	lowest precedence	right associative

A propositional variable is a nonempty string of lower case letters, upper case letters and digits whose first character is either an upper case letter or a digit.

filename is the name of a file that contains a *labeled tree*. A labeled tree is represented by a sequence of blank and non-blank lines. Each non-blank line represents a node of the labeled tree and is of the following form:

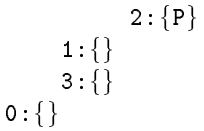
name `:` `{` *varList* `}`

The *name* of the node is a nonempty string of lower case letters, upper case letters and digits. *varList* is a possibly empty list of propositional variables separated by commas. In connection with the option `--minimal` the falsity symbol `f` is also a legal element of a *varList*. Each line can be indented. The indentation level is used to indicate how two nodes that are represented by adjacent lines are related.

For each natural number n we define sets $Tree(n)$ and $Forest(n)$ of sequences of lines of the above format. $Tree(n)$ will contain labeled trees where the line representing the root is indented to column n. $Forest(n)$ will contain forests of labeled trees where each line representing a root is indented to column n. These sets are defined inductively by the following rules:

1. $() \in Forest(n)$, for each natural number n.

Figure 1: A labeled tree



2. If $(k_1, .., k_s) \in Forest(n)$ and $(l_1, .., l_r) \in Tree(n)$ then $(k_1, .., k_s, l_1, .., l_r) \in Forest(n)$, for each natural number n.

3. If l is a line indented to column n, $m > n$ and $(k_1, .., k_r) \in Forest(m)$ then $(k_1, .., k_r, l) \in Tree(n)$, for all natural numbers n and m.

Now, a *labeled tree* is a sequence of lines such that when neglecting all blank lines the resulting sequence of lines is an element of $Tree(n)$ for some natural number n. From the inductive definition it is clear how such a sequence of lines represents a tree whose nodes are labeled with sets of propositional variables. The representation mapping that maps every such sequence of lines to a labeled tree is given in the obvious recursive way. The labeled tree that is obtained is not a Kripke-tree yet, because there might be nodes whose labeling sets do not include the labeling sets of all their ancestors. However, the labeling sets can be extended to obtain a Kripke-tree by adding to the labeling set of each node the labeling sets of all of its ancestors.

Examples

Figures 1 and 2 show labeled trees. The sequence of lines in figure 3 is not a labeled tree because of improper indentation. Figures 4 and 5 show the Kripke-trees represented by the labeled trees from figures 1 and 2. Figures 6, 7, 8, 9, 11 and 12 show example responses to different invocations of `kripke`. In those examples `fig1`, `fig2` and `fig10` are names of files that contain the trees from figures 1, 2 and 10, respectively.

Figure 2: A labeled tree

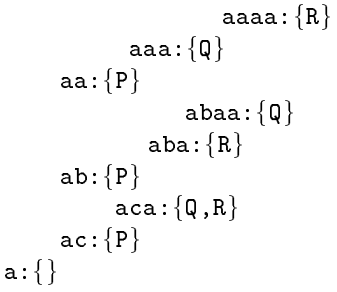


Figure 3: Not a labeled tree

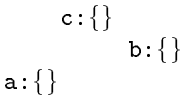


Figure 4: The Kripke-tree represented by figure 1

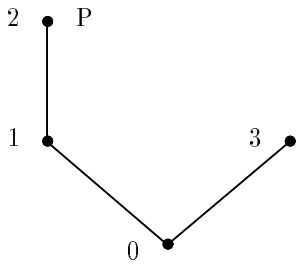


Figure 5: The Kripke-tree represented by figure 2

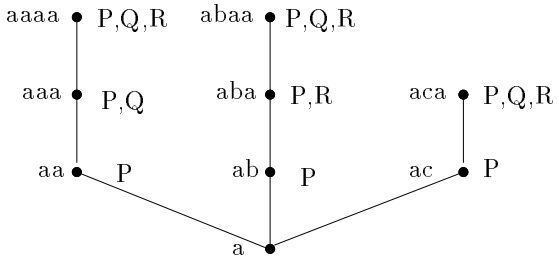


Figure 6: The check option (default)

```
% kripke fig1 " => (~~P -> P) | ~P | ~~P "

The assumptions of the sequent hold in the model but the conclusion does not.

[0.010 sec user cpu time (0.010 sec non-gc, 0.000 sec gc)]
```

Figure 7: The annotate option

```
% kripke -a fig1 " => (~~P -> P) | ~P | ~~P "

     3: { ~P,
          ~P | ~~P,
          (~~P -> P) | ~P | ~~P,
          ~~P -> P }
        2: { P,
             ~~P,
             ~P | ~~P,
             (~~P -> P) | ~P | ~~P,
             ~~P -> P }
     1: { ~~P,
          ~P | ~~P,
          (~~P -> P) | ~P | ~~P }
 0: {  }

The assumptions of the sequent hold in the model but the conclusion does not.

[0.010 sec user cpu time (0.010 sec non-gc, 0.000 sec gc)]
```

Figure 8: The find option

```
% kripke -f fig2 " (P -> Q) -> R => (P -> ~~Q) -> R "

Node "ab" forces all assumptions but not the conclusion.

[0.000 sec user cpu time (0.000 sec non-gc, 0.000 sec gc)]
```

Figure 9: The list option

```
% kripke -l fig2 " (P -> Q) -> R => (P -> ~~Q) -> R "

Node(s) "ab", "ac" force(s) all assumptions but not the conclusion.

[0.000 sec user cpu time (0.000 sec non-gc, 0.000 sec gc)]
```

Figure 10: A representation of a Kripke-tree for minimal logic

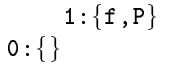


Figure 11: The minimal option

```
% kripke -m fig10 " ~~P -> ~~Q => ~~(P -> Q) "

The assumptions of the sequent hold in the model but the conclusion does not.

[0.000 sec user cpu time (0.000 sec non-gc, 0.000 sec gc)]
```

Figure 12: The minimal option in connection with the annotate option

```
% kripke -m -a fig10 " ~~P -> ~~Q => ~~(P -> Q) "

     1: { f,
          P,
          ~P,
          ~Q,
          ~~P,
          ~~Q,
          ~~(P -> Q),
          ~(P -> Q),
          ~~P -> ~~Q }
 0: { ~P,
      ~Q,
      ~(P -> Q),
      ~~P -> ~~Q }

The assumptions of the sequent hold in the model but the conclusion does not.

[0.020 sec user cpu time (0.020 sec non-gc, 0.000 sec gc)]
```