An instantiation algorithm for TLA+ expressions

February 16, 2017

1 Overview

TLA+ has two kinds of substitution: instantiation of modules, which preserves validity, and beta-reduction of lambda-expressions, which does not necessarily preserve validity. Moreover, the two substitutions do not commute. For example, let us consider modules Foo and Bar:

```
---- MODULE Foo ----

VARIABLE x

E(u) == x' # u'
D(u) == ENABLED ( E(u) )

THEOREM T1 == ~D(x)

====
```

```
---- MODULE Bar ----

VARIABLE y

I == INSTANCE Foo with x <- y

THEOREM T2 == D(y)

====
```

It looks like I!T1 and T2 talk about the same formula D(y), but this is not the case: the first can be read as I!(D(y))[1] and the second as (I!D)(y). In other words, it makes a difference if we beta-reduce first or if we instantiate first. Reducing D(x) first leads to ENABLED (x' # x'), which – following the renaming instuctions in "Specifying Systems" – becomes ENABLED ($x' # $x') by the instantiation of x with y, where primed occurrences of $x are bound by their enclosing ENABLED and therefore untouched.

Instantiating first keeps the occurrence of the variable u intact, leading to I!D(u) == ENABLED (u # $x') . Unfolding the definitions and reducing the application I!D(y) subsequently leads to ENABLED (y' # $x'). Now it is clear that I!(D(y)) is unsatisfiable while (I!D)(y) is satisfiable. Since TLA+ contains set theory, finite domains – and in particular, single element domains – are excluded. Therefore (I!D)(y) is a theorem in TLA+.

In the following, we will develop algorithms for both kinds of substitutions. Since inner substitutions have to be carried out before applying outer ones, special consideration will be taken with regard to partially unfolded definitions.

2 New Attempt: Explicit subtitutions

The original idea here is to represent both beta-reduction and instantiation explicitly in the term graph. Then the two formulas in the introduction could be written as $D(u)\{u \mapsto x\}[x \mapsto y]$ and $D(u)[x \mapsto y]\{u \mapsto x\}$, where reduction is denoted by curly braces and instantiation is denoted by square braces. Actually, the SANY data-structures allow to write reduction as application to an abstraction: $D(u)\{u \mapsto x\}$ is then just (LAMBDA u: D(u))(x). Again, this is not legal TLA+ but allowed by SANY.

[1] This is not valid TLA+ syntax.

2.1 Datastructures

node	content	comment
Module	list of constants, list of variables,	
	list of instances, list of definitions	
Constant	name, arity	Set of Constants CS
Variable	name	arity == 0, Set of Variables VS
Parameter	name, arity	Set of Paramters FP
Definition	name, arity, list of parameters, expression body	Set of Definitions DS
Expression	constant	
	or variable	
	or parameter	
	or definition	
	or abstraction	
	or application	
	or substin	
Application	head expression, argument expression list	head.arity == list length
Abstraction	parameter, expression body	
SubstIn	instantiation, expression body	the explicit instantiation node
Instance	module, parameters, instantiation	
Instantiation	list of assigments of variables/constants to expressions	

The definition and instantiation elements do not contain arguments since they can always be rewritten in terms of abstractions: D(x) == F is equivalent to D == LAMBDA x : F and I(x)!D is equivalent to LAMBDA x : I!D. We write abstraction as $\lambda x : F$ and instantiation as $(\rho M\ with x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n)!e$ where the variables / constants x_i are exactly those declared in the module M.

We assume we have a set $Unfolded$ of definitions which are unfolded and a context tracing if we are inside ENABLED.

Rewrite rules:

$(\lambda x : c)(e)$	$\rightarrow$	c	$c \in CS \cup VS$	
$(\lambda x : x)(e)$	$\rightarrow$	e	$x \in FP$	
$(\lambda x : y)(e)$	$\rightarrow$	y	$y \in FP$	
$(\lambda x : D)(e)$	$\rightarrow$	$(\lambda x : b)(e)$	$D \in DS, b = D.body, D \in Unfolded$	
$(\lambda x : f(g))(e)$	$\rightarrow$	$((\lambda x : f)(e))(\lambda x : g)(e))$		
$(\lambda x : \lambda y : s)(t)$	$\rightarrow$	$(\lambda y : (\lambda x : s)(t))$	$y \notin FV(s) \cup FV(t)$	$***$
$(\lambda x : \lambda y : s)(t)$	$\rightarrow$	$(\lambda x : (\lambda z : (\lambda y : s)(z))(t))$	$y \in FV(s) \cup FV(t)$,	
			$z \notin FV(s) \cup FV(t)$	
$(\rho M\ with\ c \leftarrow s)!x$	$\rightarrow$	s	$c \in CS \cup VS$	$*$
$(\rho M\ with\ c \leftarrow s)!x$	$\rightarrow$	x	$x \in FP$	
$(\rho M\ with\ c \leftarrow s)!D$	$\rightarrow$	$(\rho M\ with\ c \leftarrow s)!(b)$	$D \in DS, D \in Unfolded, b = D.body$	
$(\rho M\ with\ c \leftarrow s)!(f(g))$	$\rightarrow$	$(\rho M\ with\ c \leftarrow s)!(f)((\rho M\ with\ c \leftarrow s)!(g))$	$f \neq'$ or f outside of EN	
$(\rho M\ with\ c \leftarrow s)!(g')$	$\rightarrow$	$((\rho M\ with\ c \leftarrow \$c)!(g))'$	$'$ inside of $EN, \$c \notin CS \cup VS$	$**$

Remarks:

*	all modules of M are instantiated
**	this is unclean since it doesn't capture the difference between
	$EN(x \neq x' \wedge x = x')$ and $EN(x \neq x') \wedge EN(x = x')$ well
	FP-substitution stops at folded definitions and CS/VS subtitutions
	CS/VS-substitution stops at folded definitions and FP-substitutions

2.2 Termination

We define a lexicographic order on the following measures:

1. Variable name clashes:

- $clash(c) = clash(v) = clash(fp) = 0$
- $clash(s(t)) = clash(s) + clash(t)$ where s is not abstraction
- $clash(\lambda x.s)t = clash(s)$ if $x \notin FV(t)$
- $clash(\lambda x.s)t = clash(s) + 1$ if $x \in FV(t)$
- $clash(\lambda x.s) = clash(s)$ for the cases not covered above

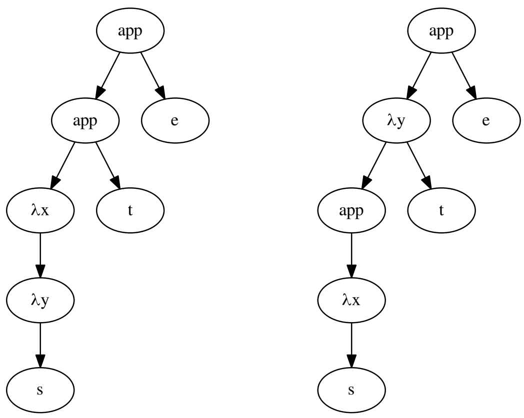
2. Deepest term under an abstraction which can be applied:

   - $d(v) = d(c) = d(fp) = 0$
   - $d(\lambda x.s) = 1 + d(s)$
   - $d(s(t)) = 1 + max(d(s), d(t))$

   - $da(v) = da(c) = da(fp) = 0$
   - $da(\lambda x.s) = 1 + da(s)$
   - $da(s(t)) = \begin{cases} 1 + d(s) + max(da(s), da(t)) & \text{if } s = \lambda x.r \\ max(da(s), da(t)) & \text{otherwise} \end{cases}$

3 Open Problems

- Confluence: I only checked overlaps of root position vs root position, non-root overlaps are possible
- The ordering da does not decrease in some cases (e.g.: wrap the redex of the *** rule into an application redex(e)). The reason is that outer pattern matches weigh heavier than inner ones:



The weight of the abstraction on x decreases as expected, but at the same time the weight of the abstraction on y increases. Since y is higher, its weight contributes more.

4 Excursion: parametrized instantiations in SANY

In SANY, parametrized instantuitions have a representation where the parameter is shifted over the instantiation. Let us consider the modules Foo and Bar:

−−−− **MODULE** Foo −−−−

VARIABLE a

D(u) == u' # a'

−−−− **MODULE** Bar −−−−
EXTENDS Naturals

VARIABLE x

E(u) == D(u) \/ **ENABLED** D(u)

====

I(v) == **INSTANCE** Foo WITH a <− x+v

====

The term I(x)!E(x) is represented as I!E(x,x) but they are not equivalent. In the meta-notation, they would be E{u $\mapsto$ x}[a $\mapsto$ x+v]{v $\mapsto$ x} vs. E{u $\mapsto$ x, v $\mapsto$ x}[a $\mapsto$ x+v] with the same consequences as in the introduction.

TODOs:

- compute the set of unfolded defs
- $(ENABLED\ A) \Leftrightarrow C$ used instantiated