at the time of writing, αCheck is unique as a model checker for binding signatures and specifications.

All test have been performed under Ubuntu 15.4 on a Intel Core i7 CPU 870, 2.93GHz with 8GB RAM. We time-out the computation when it exceeds 200 seconds. We report 0 when the time is <0.01. These tests must be taken with a lot of salt: not only is our tool under active development but the comparison with the other systems is only roughly indicative, having to factor differences between logic and functional programming (PLT-Redex), as well as the sheer scale and scope of counter-examples search in a system such as Isabelle/HOL.

5.1 Head-to-Head with PLT-Redex

We first measure the amount of *time to exhaust the search space* (TESS) using the three versions of negations supported in αCheck, over a bug-free version of the *Stlc* benchmark for $n = 1, 2, \ldots$ up to the point where we time-out. This gives some indication of how much of the search space the three techniques explore, keeping in mind that what is traversed is very different in shape; hence the more reliable comparison is between *NE* and *NEs*. As the results depicted in Figure 4 suggests, *NEs* shows a clear improvement over *NE*, while *NF* holds its ground, however hindered by the explosive exhaustive generation of terms.

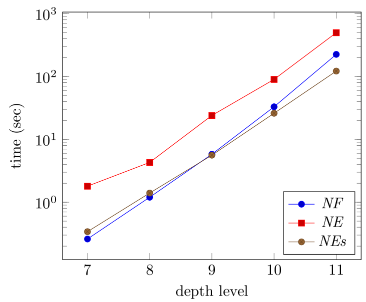

Fig. 4. Loglinear-plot of TESS on prog theorem

However, our mission is finding counterexamples and so we compare the *time to find counterexamples* (TFCE) using *NF*, *NE*, *NEs* on the said benchmarks. We list in Table 1 the 9 mutations from the cited site. Every row describes the mutation inserted with an informal classification inherited from ibidem — (S)imple, (M)edium or (U)nusual, better read as artificial. We also list the counterexamples found by αCheck under *NF* (NE(s) being analogous but less instantiated) and the depths at which those are found or a time-out occurred.

The results in Table 1 show a remarkable improvement of *NEs* over *NE*, in terms of counter-examples that were timed-out (bug 2 and 5), as well as major speedups of more than an order of magnitude (bugs 3 (ii) and 7). Further, *NEs* never under-performs *NE*, probably because it locates counterexample at a lower depth. In rare occasions (bug 5 again) *NEs* even outperforms *NF* and in several cases it is comparable (bug 1, 3, 7, 8 and 9). Of course there are occasions (2 and 6), where *NF* is still dominant, as *NEs* counter-examples live at steeper depths (12 and 16, respectively) that cannot yet be achieved within the time-out.

bug	check	NF	NE	NEs	cex	Description/Class
1	pres	0.3 (7)	1 (7)	0.37 (7)	$(\lambda x.\ x\ err)\ n$	range of function in app rule
	prog	0 (5)	3.31 (9)	0.27 (5)	*hd n*	matched to the arg. (S)
2	prog	0.27 (8)	t.o. (11)	85.3 (12)	*(cons n) nil*	value *(cons v) v* omitted (M)
3	pres	0.04 (6)	0.04 (6)	0.3 (6)	$(\lambda x.\ n)\ m$	order of types swapped
	prog	0 (5)	3.71 (9)	0.27 (8)	*hd n*	in function pos of app (S)
4	prog	t.o.	t.o.	t.o.	?	the type of cons is incorrect (S)
5	pres	t.o. (9)	t.o. (10)	41.5 (10)	*tl ((cons n) err)*	tail red. returns the head (S)
6	prog	29.8 (11)	t.o. (11)	t.o. (12)	*hd ((cons n) nil)*	hd red. on part. appl. cons (M)
7	prog	1.04 (9)	18.5 (10)	1.1 (9)	$hd\ ((\lambda x.\ err)\ n)$	no eval for argument of app (M)
8	pres	0.02 (5)	0.03 (5)	0.1 (5)	$(\lambda x.\ x)\ nil$	lookup always returns int (U)
9	pres	0 (5)	0.02 (5)	0.1 (5)	$(\lambda x.\ y)\ n$	vars do not match in lookup (S)

Table 1. TFCE on the *Stlc* benchmark, Redex-style encoding

We do not report TFCE of PLT-Redex, because, being based on randomized testing, what we really should measure is time spent *on average* to find a bug. The two encodings are quite different: Redex has very good support for evaluation contexts, while we use congruence rules. Being untyped, the Redex encoding treats *err* as a string, which is then procedurally handled in the statement of preservation and progress, whereas for us it is part of the language. Since [18], Redex allows the user to write certain judgments in a declarative style, provided they can be given a functional mode, but more complex systems, such as typing for a polymorphic version of a similar calculus, require very indirect encoding, e.g. CPS-style. We simulate addition on integers with numerals (omitted from the code snippets presented in Section 2 for the sake of space), as we currently require our code to be pure in the logical sense, as opposed to Redex that maps integers to Racket's ones. *W.r.t.* lines of code, the size of our encoding is roughly 1/4 of the Redex version, not counting Redex's built-in generators and substitution function. The adopted checking philosophy is also somewhat different: they choose to test preservation and progress together, using a cascade of three built-in generators and collect all the counterexamples found within a timeout.

The performance of the negation elimination variants in this benchmark is not too impressive. However, if we adopt a different style of encoding (let's call it PCF, akin to what we used in [9]), where constructors such as `hd` are *not* treated as constants, but are first class, e.g.:

```
tc(G,hd(E),intTy)         :- tc(G,E,listTy).
step(hd(cons(H,Tl)), H) :- value(H),value(Tl).
```

then all counter-examples are found very quickly, as reported in Table 2. In bug 4, *NEs* struggles to get at depth 13: on the other hand PLT-Redex fails to find that very bug. Bug 6 as well as several counterexamples disappear as not well-typed. This improved efficiency may be due to the reduced amount of nesting of