The Benchmark Models

July 14, 2015

The programs in our benchmark come from two sources: synthetic examples based on our experience with Redex over the years and from models that we and others have developed and bugs that were encountered during the development process.

The benchmark has six different Redex models, each of which provides a grammar of terms for the model and a soundness property that is universally quantified over those terms. Most of the models are of programming languages and most of the soundness properties are type-soundness, but we also include red-black trees with the property that insertion preserves the red-black invariant, as well as one richer property for one of the programming language models (discussed in section 3).

For each model, we have manually introduced bugs into a number of copies of the model, such that each copy is identical to the correct one, except for a single bug. The bugs always manifest as a term that falsifies the soundness property.

The table in figure 1 gives an overview of the benchmark suite, showing some numbers for each model and bug. Each model has its name and the number of lines of code for the bug-free model (the buggy versions are always within a few lines of the originals). The line number counts include the model and the specification of the property.

Each bug has a number and, with the exception of the rvm model, the numbers count from 1 up to the number of bugs. The rvm model bugs are all from ??? (???)'s work and we follow their numbering scheme (see section 8 for more information about how we chose the bugs from that paper).

The **S/M/D/U** column shows a classification of each bug as:

- **S** (Shallow) Errors in the encoding of the system into Redex, due to typos or a misunderstanding of subtleties of Redex.
- **M** (Medium) Errors in the algorithm behind the system, such as using too simple of a data-structure that doesn't allow some important distinction, or misunderstanding that some rule should have a side-condition that limits its applicability.
- **D** (Deep) Errors in the developer's understanding of the system, such as when a type system really isn't sound and the author doesn't realize it.
- **U** (Unnatural) Errors that are unlikely to have come up in real Redex programs but are included for our own curiosity. There are only two bugs in this category.

The size column shows the size of the term representing the smallest counterexample we know for each bug, where we measure size as the number of pairs of parentheses and atoms in the s-expression representation of the term.

Each subsection of this section introduces one of the models in the benchmark, along with the errors we introduced into each model.

1 stlc

A simply-typed λ-calculus with base types of numbers and lists of numbers, including the constants `+`, which operates on numbers, and `cons`, `head`, `tail`, and `nil` (the empty list), all of which operate only on lists of numbers. The property checked is type soundness: the combination of preservation (if a term has a type and takes a step, then the resulting term has the same type) and progress (that well-typed non-values always take a reduction step).

We introduced nine different bugs into this system. The first confuses the range and domain types of the function in the application rule, and has the small counterexample: `(hd 0)`. We consider this to be a shallow bug, since it is essentially a typo and it is hard to imagine anyone with any knowledge of type systems making this conceptual mistake. Bug 2 neglects to specify that a fully applied `cons` is a value, thus the list `((cons 0) nil)` violates the progress property. We consider this be be a medium bug, as it is not a typo, but an oversight in the design of a system that is otherwise correct in its approach.

We consider the next three bugs to be shallow. Bug 3 reverses the range and the domain of function types in the type judgment for applications. This was one of the easiest bug for all of our approaches to find. Bug 4 assigns `cons` a result type of `int`. The fifth bug returns the head of a list when `tl` is applied. Bug 6 only applies the `hd` constant to a partially constructed list (i.e., the term `(cons 0)` instead of `((cons 0) nil)`). Only the grammar based random generation exposed bugs 5 and 6 and none of our approaches exposed bug 4.

The seventh bug, also classified as medium, omits a production from the definition of evaluation contexts and thus doesn't reduce the right-hand-side of function applications.

Bug 8 always returns the type `int` when looking up a variable's type in the context. This bug (and the identical one in the next system) are the only bugs we classify as unnatural. We included it because it requires a program to have a variable with a type that is more complex that just `int` and to actually use that variable somehow.

Bug 9 is simple; the variable lookup function has an error where it doesn't actually compare its input to variable in the environment, so it effectively means that each variable has the type of the nearest enclosing lambda expression.

2 poly-stlc

This is a polymorphic version of section 1, with a single numeric base type, polymorphic lists, and polymorphic versions of the list constants. No changes were made to the

Model	**LoC**	**Bug#**	**S/M/D/U**	**Size**	**Description of Bug**
stlc	211	1	S	3	app rule the range of the function is matched to the argument
		2	M	5	the ((cons v) v) value has been omitted
		3	S	8	the order of the types in the function position of application has been swappe
		4	S	9	the type of cons is incorrect
		5	S	7	the tail reduction returns the wrong value
		6	M	7	hd reduction acts on partially applied cons
		7	M	9	evaluation isn't allowed on the rhs of applications
		8	U	12	lookup always returns int
		9	S	15	variables aren't required to match in lookup
poly-stlc	277	1	S	6	app rule the range of the function is matched to the argument
		2	M	11	the (([cons @ τ] v) v) value has been omitted
		3	S	14	the order of the types in the function position of application has been swappe
		4	S	15	the type of cons is incorrect
		5	S	16	the tail reduction returns the wrong value
		6	M	16	hd reduction acts on partially applied cons
		7	M	9	evaluation isn't allowed on the rhs of applications
		8	U	15	lookup always returns int
		9	S	18	variables aren't required to match in lookup
stlc-sub	241	1	S	8	forgot the variable case
		2	S	13	wrong order of arguments to replace call
		3	S	10	swaps function and argument position in application
		4	D	22	variable not fresh enough
		5	SM	17	replace all variables
		6	S	8	forgot the variable case
		7	S	13	wrong order of arguments to replace call
		8	S	10	swaps function and argument position in application
		9	SM	17	replace all variables
let-poly	640	1	S	8	use a lambda-bound variable where a type variable should have been
		2	D	28	the classic polymorphic let + references bug
		3	M	3	mix up types in the function case
		4	S	8	misspelled the name of a metafunction in a side-condition, causing the occurs
		5	M	3	eliminate-G was written as if it always gets a Gx as input
		6	M	6	$\vee$ has an incorrect duplicated variable, leading to an uncovered case
		7	D	12	used let --> left-left-λ rewrite rule for let, but the right-hand side is less poly
list-machine	256	1	S	22	confuses the lhs value for the rhs value in cons type rule
		2	M	22	var-set may skip a var with matching id (in reduction)
		3	S	29	cons doesn't actually update the store
rbtrees	187	1	M	13	ins does no rebalancing
		2	M	15	the first case is removed from balance
		3	S	51	doesn't increment black depth in non-empty case
delim-cont	287	1	M	46	guarded mark reduction doesn't wrap results with a list/c
		2	M	25	list/c contracts aren't applied properly in the cons case
		3	S	52	the function argument to call/comp has the wrong type
rvm	712	2	M	24	stack offset / pointer confusion
		3	D	33	application slots not initialized properly
		4	M	17	mishandling branches when then branch needs more stack than else branch; b
		5	M	23	mishandling branches when then branch needs more stack than else branch; b
		6	M	15	forgot to implement the case-lam branch in verifier
		14	M	27	certain updates to initialized slots could break optimizer assumptions
		15	S	21	neglected to restrict case-lam to accept only 'val' arguments

Figure 1: Benchmark Overview

model except those necessary to make the list operations polymorphic. There is no type inference in the model, so all polymorphic terms are required to be instantiated with the correct types in order for the function to type check. Of course, this makes it much more difficult to automatically generate well-typed terms, and thus counterexamples. As with **stlc**, the property checked is type soundness.

All of the bugs in this system are identical to those in **stlc**, aside from any changes that had to be made to translate them to this model.

This model is also a subset of the language specified in ??? (???), who used a specialized and optimized QuickCheck generator for a similar type system to find bugs in GHC. We adapted this system (and its restriction in **stlc**) because it has already been used successfully with random testing, which makes it a reasonable target for an automated testing benchmark.

3 stlc-sub

The same language and type system as section 1, except that in this case all of the errors are in the substitution function.

Our own experience has been that it is easy to make subtle errors when writing substitution functions, so we added this set of tests specifically to target them with the benchmark. There are two soundness checks for this system. Bugs 1-5 are checked in the following way: given a candidate counterexample, if it type checks, then all βv-redexes in the term are reduced (but not any new ones that might appear) using the buggy substitution function to get a second term. Then, these two terms are checked to see if they both still type check and have the same type and that the result of passing both to the evaluator is the same.

Bugs 4-9 are checked using type soundness for this system as specified in the discussion of the section 1 model. We included two predicates for this system because we believe the first to be a good test for a substitution function but not something that a typical Redex user would write, while the second is something one would see in most Redex models but is less effective at catching bugs in the substitution function.

The first substitution bug we introduced simply omits the case that replaces the correct variable with the term to be substituted. We considered this to be a shallow error, and indeed all approaches were able to uncover it, although the time it took to do so varied.

Bug 2 permutes the order of arguments when making a recursive call. This is also categorized as a shallow bug, although it is a common one, at least based on our experience writing substitutions in Redex.

Bug 3 swaps the function and argument positions of an application while recurring, again essentially a typo and a shallow error, although one of the more difficult to find in this model.

The fourth substitution bug neglects to make the renamed bound variable fresh enough when recurring past a lambda. Specifically, it ensures that the new variable is not one that appears in the body of the function, but it fails to make sure that the variable is different from the bound variable or the substituted variable. We categorized this error as deep because it corresponds to a misunderstanding of how to generate fresh variables, a central concern of the substitution function.

Bug 5 carries out the substitution for all variables in the term, not just the given variable. We categorized it as SM, since it is essentially a missing side condition, although a fairly egregious one.

Bugs 6-9 are duplicates of bugs 1-3 and bug 5, except that they are tested with type soundness instead. (It is impossible to detect bug 4 with this property.)

4 let-poly

A language with ML-style `let` polymorphism, included in the benchmark to explore the difficulty of finding the classic let+references unsoundness. With the exception of the classic bug, all of the bugs were errors made during the development of this model (and that were caught during development).

The first bug is simple; it corresponds to a typo, swapping an `x` for a `y` in a rule such that a type variable is used as a program variable.

Bug number 2 is the classic let+references bug. It changes the rule for `let`-bound variables in such a way that generalization is allowed even when the initial value expression is not a value.

Bug number 3 is an error in the function application case where the wrong types are used for the function position (swapping two types in the rule).

Bugs 4, 5, and 6 were errors in the definition of the unification function that led to various bad behaviors.

Finally, bug 7 is a bug that was introduced early on, but was only caught late in the development process of the model. It used a rewriting rule for `let` expressions that simply reduced them to the corresponding `((`λ expressions. This has the correct semantics for evaluation, but the statement of type-soundness does not work with this rewriting rule because the let expression has more polymorphism that the corresponding application expression.

5 list-machine

An implementation of ??? (???)'s list-machine benchmark. This is a reduction semantics (as a pointer machine operating over an instruction pointer and a store) and a type system for a seven-instruction first-order assembly language that manipulates `cons` and `nil` values. The property checked is type soundness as specified in ??? (???), namely that well-typed programs always step or halt. Three mutations are included.

The first list-machine bug incorrectly uses the head position of a cons pair where it should use the tail position in the cons typing rule. This bug amounts to a typo and is classified as simple.

The second bug is a missing side-condition in the rule that updates the store that has the effect of updating the first position in the store instead of the proper position in the store for all of the store update operations. We classify this as a medium bug.

The final list-machine bug is a missing subscript in one rule that has the effect that the list cons operator does not store its result. We classify this as a simple bug.

6 rbtrees

A model that implements the red-black tree insertion function and checks that insertion preserves the red-black tree invariant (and that the red-black tree is a binary search tree).

The first bug simply removes the re-balancing operation from insert. We classified this bug as medium since it seems like the kind of mistake that a developer might make in staging the implementation. That is, the re-balancing operation is separate and so might be put off initially, but then forgotten.

The second bug misses one situation in the re-balancing operation, namely when a black node has two red nodes under it, with the second red node to the right of the first. This is a medium bug.

The third bug is in the function that counts the black depth in the red-black tree predicate. It forgets to increment the count in one situation. This is a simple bug.

7 delim-cont

??? (???)'s model of a contract and type system for delimited control. The language is Plotkin's PCF extended with operators for delimited continuations, continuation marks, and contracts for those operations. The property checked is type soundness. We added three bugs to this model.

The first was a bug we found by mining the model's git repository's history. This bug fails to put a list contract around the result of extracting the marks from a continuation, which has the effect of checking the contract that is supposed to be on the elements of a list against the list itself instead. We classify this as a medium bug.

The second bug was in the rule for handling list contracts. When checking a contract against a cons pair, the rule didn't specify that it should apply only when the contract is actually a list contract, meaning that the cons rule would be used even on non-list contacts, leading to strange contract checking. We consider this a medium bug because the bug manifests itself as a missing `list/c` in the rule.

The last bug in this model makes a mistake in the typing rule for the continuation operator. The mistake is to leave off one-level of arrows, something that is easy to do with so many nested arrow types, as continuations tend to have. We classify this as a simple error.

8 rvm

A existing model and test framework for the Racket virtual machine and bytecode verifier (??? ???). The bugs were discovered during the development of the model and reported in section 7 of that paper. Unlike the rest of the models, we do not number the bugs for this model sequentially but instead use the numbers from ??? (???)'s work.

We included only some of the bugs, excluding bugs for two reasons:

- The paper tests two properties: an internal soundness property that relates the verifier to the virtual machine model, and an external property that relates the verifier model to the verifier implementation. We did not include any that require the latter properties because it requires building a complete, buggy version of the Racket runtime system to include in the benchmark.

- We included all of the internal properties except those numbered 1 and 7 for practical reasons. The first is the only bug in the machine model, as opposed to just the verifier, which would have required us to include the entire VM model in the benchmark. The second would have required modifying the abstract representation of the stack in the verifier model in contorted way to mimic a more C-like implementation of a global, imperative stack. This bug was originally in the C implementation of the verifier (not the Redex model) and to replicate it in the Redex-based verifier model would require us to program in a low-level imperative way in the Redex model, something not easily done.

These bugs are described in detail in ??? (???)'s paper.

This model is unique in our benchmark suite because it includes a function that makes terms more likely to be useful test cases. In more detail, the machine model does not have variables, but instead is stack-based; bytecode expressions also contain internal pointers that must be valid. Generating a random (or in-order) term is relatively unlikely to produce one that satisfies these constraints. For example, of the first 10,000 terms produced by the in-order enumeration only 1625 satisfy the constraints. The ad hoc random generator generators produces about 900 good terms in 10,000 attempts and the uniform random generator produces about 600 in 10,000 attempts.

To make terms more likely to be good test cases, this model includes a function that looks for out-of-bounds stack offsets and bogus internal pointers and replaces them with random good values. This function is applied to each of the generated terms before using them to test the model.