Semantics of Functions in Alloy

Manu Sridharan
6.898 Final Project

May 16, 2002

Abstract

While functions have been a very useful feature in the new Alloy language, their semantics have some undesirable properties and seem to be counterintuitive for most Alloy users. For my final project, I studied the semantics of Alloy functions more carefully with the hope of identifying its problems more concretely and finding a new semantics which addresses those problems. With the help of Daniel Jackson, I identified an issue related to partial functions which may have been the source of a great deal of confusion and found a way to restrict the language to avoid confusing invocation semantics as much as possible.

1 Background

Alloy functions are a flexible construct which can make an Alloy model much easier to write and understand. Function paragraphs (declared with keyword `fun`) are a generalization of the `cond` paragraph from Alloy's predecessor, Alloy Alpha. Unlike functions in typical programming languages, Alloy functions are just typed macros, so an invocation of a function always desugars to some sort of inlining of the function body. Because of the variety of ways in which a function can be written and invoked, interesting issues arise in defining the exact semantics of this inlining step.

Consider the following partial model of processes and mutexes, in which the global state of the system is represented by the `holds` and `waits` relation, respectively representing which mutexes are held and waited on in each state:

```
sig Process {}
sig Mutex {}
sig State { holds, waits: Process -> Mutex }
fun freeInState(s : State, m : Mutex) {
  no s.holds.m
}
```

As with `cond` paragraphs in Alloy Alpha, functions can be invoked where a formula is expected, and the semantics of such an invocation are a straightforward inlining; so, the invocation `freeInState(s',m')` desugars to `no s'.holds.m'`. However, unlike `cond` paragraphs, functions can also be invoked where an expression is expected. For example, the following function defines the condition where a process grabs a mutex that was free in another state:

```
fun GrabMutex(s, s': State, p : Process) {
  p -> freeInState(s) in s'.holds
}
```

This invocation of `freeInState` has no obvious meaning that handles all cases. If it is known that exactly one mutex m is free in state s, a sensible behavior for this invocation would be for it to return m. However, several mutexes could be free in state s. In this case, current Alloy semantics treats the function as being *non-deterministic* and tries to check the formula containing the invocation for all possible return values. Unfortunately, this treatment of non-determinism has some counterintuitive results. For example, in the above case, the `GrabMutex` function would desugar into a formula stating that p grabs *all* free mutexes from state s, as opposed to non-deterministically choosing one of them. Also, what if there are no mutexes free in s? Current Alloy semantics allows for essentially arbitrary behavior in this case, which can be extremely puzzling. The new semantics proposed in this paper disallows invocations like these and slightly changes the handling of other constructs, in the hope of limiting function invocations to cases where their behavior is reasonable and easy to understand.

2 Deficiencies in Current Semantics

The handling of invocations as a formula and as a expression when the function uses the "`result = ...`" syntax is straightforward and will not be discussed here. In the remaining case, when a function is invoked as an expression and its body constrains its result implicitly, the current inlining method depends on whether the function was declared to be deterministic (with the keyword `det`). Consider an invocation `f`$(a_0, a_1, ..., a_n)$ whose smallest enclosing formula is `F`, where `f` has $n+1$ arguments and its second argument (the result) has `A` as the right-hand side of its declaration. If `f` is non-deterministic, the invocation desugars to

    all result : A | f(a_0,result,a_1,...,a_n) => F [result / f($a_0, a_1, ..., a_n$)]

which says that for all possible values for the result of the invocation of `f`, `F` should be true when the invocation is replaced by the result value. `f` can be declared as deterministic, in which case the invocation desugars to

    some result : A | f(a_0,result,a_1,...,a_n) && F [result / f($a_0, a_1, ..., a_n$)]

which says that `F` is true when the invocation is replaced by the one valid result it can return. Note that determinism is not defined precisely, an issue that I will return to later.

One problem with these semantics is that they do not seem to match the intuitions of many Alloy users. From experience, most people writing Alloy models seem to expect the inlining behavior associated with deterministic functions as the default, and when a user is having problems with functions, adding the `det` keyword to the relevant function declarations makes things work as they expect most of the time. Also, it seems to be quite difficult to identify a bug related to a misunderstanding of function semantics, with users often reporting that they spent many hours stumped over the problem with their model (however, this may reflect the general difficulty of debugging Alloy specifications).

Apart from comprehensibility issues, the existing handling of non-deterministic functions is problematic. It is currently impossible to use the result of a non-deterministic function twice, decreasing its usefulness. For someone used to the behavior of let declarations in programming languages like Scheme and ML, it may seem like binding the result of the function invocation in a let would have the desired behavior. However, Alloy's let construct is desugared by simple syntactic replacement, so placing an invocation in a let declaration would lead to multiple invocations after desugaring rather than multiple uses of one result.

Another peculiarity of non-deterministic functions stems from negation of the set equality operator. In the current semantics, the formulas `b != f(a)` and `!(b = f(a))` have different meanings when `f` is a non-deterministic function. Assuming `f` has result declaration `A`, `b != f(a)` desugars to

```
all result : A | f(a,result) => b != result
```

which means that `b` should not equal any possible result of `f(a)`. In contrast, `!(b = f(a))` translates to

```
some result : A | f(a,result) && !(b = result)
```

which says that there exists some possible result value of `f(a)` other than `b`. Our use of the smallest enclosing formula when inlining results in this bizarre difference.

The most serious problem with non-deterministic function inlining is its poor semantics when the function is partial. Previously, we assumed that the following formula was true for any non-deterministic function `f` which has argument and result with declaration `A`:

```
all a, b: A | b = f(a) => f(a,b)
```

However, Sarfraz Khurshid and Darko Marinov recently provided a function which disproved this hypothesis:

```
fun f(a : A): A { a != a }
```

With this function, `b = f(a)` desugars to true, since the invocation `f(a,result)` on the left-hand side of the implication in the resulting quantifier will always be false. So, for any `a` and `b`, the above formula evaluates to false, since the body becomes true implies false. In general, if a non-deterministic function is undefined for the arguments of some invocation, then the formula enclosing that invocation becomes true. This behavior is both counterintuitive and inconsistent with how Alloy makes other partial functions false outside their domain (by having the join operator evaluate to the empty set for such cases).

3 New Semantics

In considering how inlining semantics could be improved, it seemed that the cases in which invocation of a non-deterministic function leads to the creation of a universal quantifier were not compelling enough to offset its problem. Therefore, the proposed new semantics for Alloy functions disallows this behavior by adding the rule that functions which constrain their result implicitly must be marked as deterministic to be invoked as an expression. This new semantics eliminates the bad behavior of partial functions, since with the inlining method used for deterministic functions, the invocation of a function with arguments outside its domain will make the enclosing formula false instead of true (since the generated existentially quantified formula becomes false if no result exists). Furthermore, prohibiting the old non-deterministic function inlining will hopefully save time for Alloy users since they will not have to puzzle out its often unexpected effects on their models.

This proposed restriction has the unfortunate side effect of not allowing unnamed intermediate states for several invocations of non-deterministic functions. For example, the following formula from the lists example given in the Alloy book would be illegal under the new semantics:

```
assert GetBack { all p: List, e: Elt | car (cons (p,e)) = e }
```

Instead, if the car and cons functions must remain non-deterministic, the more verbose version of the formula must be written:

```
assert GetBack { all p,q: List, e,f: Elt | cons (p,q,e) && car(q,f)
                                            =>  f = e }
```

Although this restriction sometimes forces a bit more typing, it seems that users have lost much more time figuring out the bad cases of non-deterministic function inlining than they would ever gain by avoiding this typing, so the tradeoff seems reasonable.

Some weird behavior remains with respect to the != operator, but I believe it is more tractable. The meaning of writing a formula with != and moving the negation out now only differs for partial functions, where the formula is false for the != case and is true for the outer negation when the function is undefined on its arguments. To be consistent with the policy of making formulas false when partial functions are evaluated outside their domain, it is proposed that negations are moved in so that != is always used.

Another interesting issue is what exactly it means for a function to be deterministic. The above discussion of != assumes that if a function is deterministic, then it will always have the same set of atoms as its result given the same values for the other arguments. This definition of determinism would often require the user to *canonicalize* atoms of types used as results, so that each atom has some field value distinct from those of all other atoms. Others have proposed a more semantic notion of determinism based on always returning atoms with the same field values, but I think basing determinism on equality of atoms is simpler and leads to more understandable behavior (such as the case of !=). Also, if deterministic functions are allowed to return different sets of atoms on different invocations with the same arguments, the problem of not being able to use an invocation result twice remains. It may be useful to add macros or some reflective features to the language to make writing the canonicalization condition easier.

4 Conclusions

After careful study of the drawbacks of the existing semantics of Alloy functions, we have developed a new semantics which is more sensible and intuitive. One lesson I learned from this project was to respect the intuitions of users: if some language feature seems to confuse many users of the language, it is a strong sign that there are deeper problems with the feature. Also, I think the decision to disallow the confusing non-deterministic function invocations in the new semantics reflects a good language design principle espoused by Daniel Jackson: if a language construct has no obvious and satisfying meaning, it is probably better not to have the construct rather than giving it a possibly confusing meaning. Finally, if a feature with potentially confusing semantics such as non-deterministic functions are included in a language, there should be strong, concrete evidence that excluding the feature would make using the language much more tedious or complex (this evidence seemed to be a bit lacking for non-deterministic functions).