Reasoning About DrScheme Programs in ACL2

Melissa Wiederrecht, Christopher MacLellan, and Ruben Gamboa

University of Wyoming
Department of Computer Science
Laramie, WY

Abstract. Beginning programmers need to learn more than the syntax of programming languages. They also need to learn how to reason about the programs they write. Thus we believe that beginners will benefit from tools that help them understand their programs, just as they already benefit from IDEs that help them to build and debug their programs. This paper describes a project aimed at automating some of the techniques required to reason about programs in Beginning Student Language (BSL), the first language in DrScheme's *How to Design Programs* curriculum [4]. The automation is based on the theorem prover ACL2.

1 Introduction

Beginning programming students have a much larger job in front of them than mastering the syntax of their first programming language. These students need to learn how to think like programmers. But reasoning about programs involves sophisticated techniques from logic, which are usually at the levels of graduate students or advanced undergraduates, certainly not freshmen. So what can help them to fill the gap between a student's first intuition as to how a program should be written and what is in fact a correct solution to a given problem? We propose that a mechanical theorem prover, written fresh, automated, extended, and embellished with pedagogical apparatus could be used to provide students with a tool at their fingertips that would grant them instant feedback about the correctness of their programs and what they might possibly do to improve them.

Writing theorem provers is hard! So the most convenient solution to this problem is to take an existing mechanical theorem prover, such as ACL2 [7], and modify it to our liking. However, existing theorem provers were designed for researchers, not for students. For example, a beginning student could probably manage to write factorial in a perfectly reasonable manner in Lisp like this:

```
(defun fact (n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))))
```

A more forward-thinking student may write it in the following form instead:

```
(defun fact (n)
```

```
(if (≤ n 0)
  1
  (* n (fact (- n 1)))))
```

However, neither of these definitions is appropriate in ACL2. The reason is that ACL2 views definitions as axioms that introduce new function symbols via equality. The first definition above introduces the function *fact* as the unique solution to the equation

```
(fact n) = (if (= n 0)
               1
               (* n (fact (- n 1))))
```

But this equation does not have a unique solution! There are many functions that agree with *fact* on the naturals, but differ in the values they assign to other numbers. Worse yet, ACL2 is a total logic, so the expression (*fact* 'hello) must have *some* value. This is a lot for a beginning programmer to take in!

Other languages, more suitable for beginners (and some would argue for professionals as well) eschew the notion that definitions introduce equations. Instead, the languages define rewrite rules that transform terms in the language into other terms. Eventually, arbitrary expressions are transformed into terminal expressions, i.e., results. From this viewpoint, both of the definitions above for *fact* are appropriate, and they can be viewed as introducing new rewrite rules into the language. This is the framework followed by Beginning Student Language (BSL) and all the other languages in the *How to Design Programs* sequence of scheme-like languages [4, 3].

Ultimately, we believe that a new theorem prover tailored specifically to reasoning about programs in these languages is required. It may even be necessary to define several different theorem provers, corresponding to the different language levels in the *How to Design Programs* sequence. But before that process can be undertaken, it is necessary to determine which inference rules and strategies work well for typical beginning programs written in BSL. That was the goal of this project. In order to test our ideas, we implemented them in ACL2 by writing an interpreter for BSL in ACL2. This embedding of BSL in ACL2 permitted us to reason about BSL programs mechanically. More precisely, ACL2 reasoned about the behavior of the BSL interpreter on certain ACL2 expressions that corresponded to BSL programs and terms.

The *How to Design Programs* philosophy is intertwined with the DrScheme programming environment, which offers a flexible and powerful programming environment for users in a wide range of programming ability [5]. A testament to the flexibility of DrScheme, DrACuLa is an environment that allows users to interact with the theorem prover ACL2 inside DrScheme [2]. The goals of the DrACuLa project overlap with ours, in that both projects aim to introduce beginning programmers to formal ways of reasoning about programs. In fact, DrACuLa has been used to teach logic to freshmen and to more advanced undergraduates. These experiences have helped to inform and motivate our project.

In the remainder of this paper, we will describe various versions of our interpreter and the theorems that we were able to prove in ACL2 about programs in BSL.

2 First Iteration

Our first attempt at an interpreter for BSL in ACL2 consisted of several mutually recursive functions, the key functions of which are the following:

- **bsl:** evaluate a scheme expression
- **bsl-list:** evaluate a list of scheme expressions (e.g. a list of function arguments)
- **bsl-builtin:** evaluate an expression that matches a form built into BSL
- **bsl-userdefined:** evaluate an expression that is defined in the current BSL environment

The interpreter was defined using the standard method for Lisp interpreters, dating back to McCarthy [8]. The mutually recursive functions accept three arguments: a BSL expression, an environment mapping variable names to values, and a counter. The last argument is necessary because ACL2 functions must be proven to terminate before the system will accept their definition. Since interpreters cannot generally guarantee termination, the counter is used to arbitrarily halt execution after a given number of steps.

Similarly, the functions return three values. The first is a status marker that can be either *success*, *error*, or *timeout*. The marker *success* is returned when the interpreter manages to evaluate the expression. On the other hand, if the interpreter encounters an invalid input while executing the program, it returns *error*. This may happen, for example, if an undefined (in BSL) function is called, or if an arithmetic operator is applied to a string. Finally, *timeout* is returned if the value provided for the counter was not sufficient to evaluate the given expression. The second returned value of these functions is the actual value of the expression, if the evaluation was successful. And the third returned value is the new version of the environment. This can be modified, for example, when the expression that was evaluated was a BSL **define** form.

For example, the environment '((x . 1) (y . 2)) gives the variable x a value of 1, and y, 2. So the expression

(*bsl* '(+ x y) '((x . 1) (y . 2)) 1000)

returns the value

(*list* 'success 3 '((x . 1) (y . 2)))

Notice that the environment is unchanged by the addition. Conversely, the expression

(*bsl* '(+ x z) '((x . 1) (y . 2)) 1000)

returns the value

(*list* 'error nil nil)

The error is encountered when *bsl* attempts to evaluate the variable z, which is not bound in the environment. Finally, the expression

(*bsl* '(+ (∗ 2 x) y) '((x . 1) (y . 2)) 1)

returns the value

(*list* 'timeout nil nil)

In this case, more than one step of *bsl* is required to evaluate the given, nested expression.

The first class of theorems that we proved had to do with safety. A representative, albeit simple, example is the following:

```
(defthm bsl-of-any-number
  (implies (acl2-numberp expr)
           (not (equal (car (bsl expr env c))
                       'error)))
  :hints (("Goal"
           :expand (bsl expr env c))))
```

This (modest) theorem establishes that *bsl* does not encounter an error when evaluating numeric literals. We proved several such theorems, with the goal of establishing that no errors would be encountered while evaluating more substantial programs, such as *fahrenheit-to-celsius* or *factorial*.

However, as we proceeded, we noticed that we were continuously instructing ACL2 to expand recursive definitions, such as *bsl*. ACL2 generally does not open up recursive definitions, because it would never know when to stop expanding. In our case, however, expanding all the mutually recursive functions turned out to be exactly what was needed for ACL2 to prove our theorems automatically.

Explicitly opening up recursive definitions was tedious, but what eventually led us to abandon this approach happened when we started proving theorems that included built-in functions (e.g., ∗). At this point, we discovered that our theorems were not being used as rewrite rules because they were all written in this negative form. ACL2 does not know how to substitute what an expression is not equal to into the proof of a new theorem.

3 Second Iteration

After our first experience with reasoning about the recursive functions in the *bsl* family, we decided to start fresh with a new approach. Instead of ruling out *error* as a possible result, our new theorems explicitly stated what values *bsl* would return for certain expressions. A representative example is the following:

```
(defthm bsl-of-any-number-2
   (implies (acl2-numberp expr)
```

```
(equal (bsl expr env c)
       (if (and (integerp c)
                (< 1 c))
           `(success ,expr ,env)
           '(timeout nil nil))))
:hints (("Goal" ...)))
```

When written in this positive form, the theorems became useful rewrite rules in ACL2. This meant that we did not have to instruct ACL2 to expand every single occurrence of the recursive functions, but we still did have to expand many of them. We included into each of these theorems both the fact that the evaluation succeeded and that it returned the correct answer.

Because ACL2 could now use our theorems as rewrite rules, the proofs of theorems that included built-in functions became almost trivial. We moved on to tackle theorems about functions that included recursion, specifically *sum-up-to-n*, which finds the nth triangle number by adding 1 through n.

It was at this point that we discovered the fatal flaw in our interpreter: sadly, ACL2 does not do well with mutually recursive functions.

4 Third Iteration

We started over again, this time rewriting the interpreter from scratch. We combined all of our mutually recursive functions into one gigantic, singly recursive function *bsl*. The translation follows the approach to writing mutually recursive functions in NQTHM, which does not support mutual recursion directly [1]. This uses a flag to distinguish between the functionality of *bsl* and *bsl-list*. In other words, we added a parameter to the function that would be set to false when we wanted to evaluate a single expression, and true when we wanted to evaluate a list of expressions.

With this new interpreter, we were able to prove partial correctness results for many functions, including recursive functions, such as *sum-up-to-n*. In fact, we were able to prove that when the input argument is a natural number, the BSL version of *sum-up-to-n* is equivalent to a version written in ACL2, even though the ACL2 version must be written to work for any possible input value, as described in the introduction.

```
(defthm sum-n-partial-correctness
    (implies (and (sum-n-defined env)
                  (true-listp expr)
                  (equal (len expr) 2)
                  (equal (car expr) 'sum-n)
                  (equal (car (bsl expr nil env c)) *success*)
                  (equal (cadr (bsl (second expr) nil env c)) n)
                  (integerp n)
                  (<= 0 n))
             (equal (bsl expr nil env c)
```

```
(list *success*
      (sum-n n)
      env))))
```

Notice the hypothesis that *bsl* succeeds when evaluating the expression. This corresponds to the notion of partial correctness.

We were also able to prove that the BSL expression (∗ n (+ n 1) 1/2) is equivalent to the same expression in ACL2, again when n is a natural number. It is, of course, possible to prove in ACL2 that the ACL2 version of *sum-up-to-n* is equal to (∗ n (+ n 1) 1/2), and taken together with the two previous theorems, this also shows that the BSL expressions are equal to each other. Such results demonstrate how our approach can be used to reason mechanically about beginning programs in BSL.

In the process of carrying out these proofs, we learned some valuable lessons. First of all, we discovered the importance of monotonicity results. Specifically, reasoning about the counter argument to *bsl* turned out to be cumbersome in the best cases, and completely impractical when reasoning about recursive functions. But given that when *bsl* succeeds, it always returns the same value for the expression, it is possible to ignore specific values of the counter, and simply use one that is "large enough" to evaluate the current expression, and hence all its subexpressions as well. This latter style of reasoning—that the successful evaluation of an expression guarantees the successful evaluation of all its subexpressions—took up a substantial amount of the effort.

Finally, it is well known that induction is the most appropriate way of reasoning about recursive functions. Moreover, theorem provers like ACL2 can automatically generate induction schemes by considering the definition of the recursive functions appearing in a particular theorem. However, when reasoning about BSL programs, the only recursive function that ACL2 sees is the interpreter *bsl* itself—and its recursive definition is unlikely to be useful in reasoning about most functions in BSL.

We discovered that this problem can be solved by defining a corresponding function in ACL2, and then instructing ACL2 to induct according to this definition. This also required some care when stating the correctness theorem. For example, the statement of *sum-n-partial-correctness* includes the following hypothesis:

```
(equal (cadr (bsl (second expr) nil env c)) n)
```

This hypothesis introduces the variable n, and such hypotheses are usually avoided in ACL2, since they introduce free variables in the corresponding rewrite rule, which renders them almost useless from an automation viewpoint. But in this case, the free variable plays the important role of connecting the induction hypothesis with the final theorem. Consider how this variable is bound in these two cases:

– **Theorem.** Expr: (*sum-n* n)

```
(equal (cadr (bsl 'n
```

```
                nil (cons (cons 'n n) env)
                c))
               n)
```

– **Induction Hypothesis.** Expr: (*sum-n* (− *n* 1))

```
(equal (cadr (bsl '(− n 1)
                  nil (cons (cons 'n n) env)
                  c))
               (− n 1))
```

In both cases, the free variable n is bound to the argument of the *sum-n* expression, so the hypotheses of the theorem can be easily discharged.

5 Conclusions and Lessons Learned

As it stands, ACL2 can be used to reason about simple functions written in BSL, but not without significant effort. However, we have become convinced that the inference rules and strategies required to perform such reasoning can be fully automated, at least for simple programs, such as the ones beginners are likely to write in BSL.

In the case of non-recursive functions, a pass that pushes input value types down through the expressions is sufficient to guarantee that no errors are encountered while evaluating the expression, in a manner very similar to (but theoretically more powerful than) static type checking. Our experience suggests that this can be accomplished by a higher-level tool that uses ACL2 as a simple rewriter. The higher-level tool can generate commands to ACL2 using the proof checker interface.

In the case of recursive functions, we can assume that the evaluator terminates while evaluating the functions. This assumption can be pushed down into any subexpressions, and it allows us to infer the values returned by the function, given that it really does terminate. Naturally, this requires induction, using a scheme suggested by the definition of the recursive function written in BSL. A careful analysis of the lemmas needed to reason about recursive programs suggests that this, too, can be automated — at least for simple programs in BSL. The most complicated lemma applies the inductive hypothesis to the evaluation of a subterm. In the case of the funcion *sum-n* the induction hypothesis talks about the behavior of the interpreter on the term (*sum-n* X) when X has the value $n - 1$. This must be used to reason about the subterm (*sum-n* (− X 1)) when X has the value n. The necessary "bridging" lemmas can be generated automatically. In fact, the *How to Design Programs* philosophy is to develop programs through the use of templates that can be instantiated. These templates correspond to general recursion techniques, so an automated system need only recognize these few templates to suggest the appropriate induction scheme and necessary bridging lemmas. This approach is much simpler than the one we used in earlier work [6].

Although we have focused our efforts to reason about partial correctness of recursive functions, a similar approach can also be used to guarantee that the evaluation of the recursive function does not generate an error. This is very similar to the guard-checking algorithm in ACL2, which is a generalization of static type checking.

References

1. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, Orlando, 1979.
2. C. Eastlund, D. Vaillancourt, and M. Felleisen. ACL2 for freshmen: First experiences. In *Proceedings of the Seventh International Workshop of the ACL2 Theorem Prover and its Applications (ACL2-2007)*, 2007.
3. M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex.* MIT Press, Boston, 2009.
4. M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing.* MIT Press, Boston, 2001.
5. R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. Drscheme: A pedagogic programming environment for scheme. In *International Symposium on Programming Languages: Implementations, Logics, and Programs*, 1997.
6. R. Gamboa and P. Weissbrod. A cost-aware evaluator for ACL2 functions. In *Proceedings of the Fifth International Workshop of the ACL2 Theorem Prover and its Applications (ACL2-2004)*, 2004.
7. M. Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.
8. J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 1960.