

Proving Type Soundness of Simple Languages in ACL2*

Daniel J. Singer
University of Michigan
Department of Philosophy
435 South State Street
Ann Arbor, MI 48109
singerdj@umich.edu

ABSTRACT

This paper discusses the use of mechanized proof systems to prove type soundness properties of programming languages. It provides an account of using ACL2, an automated theorem prover with a relatively weak logic, to produce a proof of the type soundness of the toy language given by Pierce. It describes the toy language and type system, shows how it is implemented in ACL2's programming language, and shows how type soundness of the language is proven in ACL2 by proving the progress and preservation theorems. It also discusses a failed attempt at the same proof and reviews some avoidable pitfalls in proving soundness results in ACL2. Finally, it briefly discusses some related work on mechanized type soundness proofs, particularly in ACL2.

1. INTRODUCTION

Many programming languages use *type systems* to guarantee that programs in the language have some desired property. Type systems typically work by assigning to each term in a program a *type*, where the type typically picks out the kind of data the term manipulates or the kind of operation done by the term. For instance, it might assign to each number or numeric operation the type `Int`. Using its designation of some basic terms with a type, the type system then recursively evaluates the combined type of the whole program. For instance, it might assign to a string concatenation operator whose inputs are typed as string the type `Str`. Because the type system only needs to look at the terms in the program to determine its type, this can be performed in nearly linear time on the length of the program.

Useful type systems are designed in such a way such that if a whole program (or a major part of the program) is typed, then that program (or major part thereof) has the desired

*This paper and relevant code is available at www.umich.edu/~singerdj/cs.html. The work detailed here was submitted in partial completion of the requirements for EECS 590 at the University of Michigan in Winter 2009.

property. When a program has a type (or all of its major sections have a type) the program is said to be *well-typed*. For example, type systems are used to guarantee the safety of a program. That is, if a program is well-typed, it will not fail in some designated respect, such as entering a stuck state, i.e. not being able to be evaluated and also not being a value. When a type system guarantees this safety property for a language, the language is called *type safe* and that guarantee is called *type soundness*. As the popular slogan used to describe type soundness says, “well-typed programs never go wrong.”¹

So, it's desirable to have type safe languages since programs in these languages can be guaranteed not to fail. But the guarantee is only practically useful if it is also guaranteed that the language we take to be type safe really is type safe. Therefore, it's desirable to have a *proof* of type soundness to guarantee the desired property in the purportedly type safe language. Furthermore, humans often forget about weird corner cases and overlook important details in constructing proofs, thereby rendering the proof unsound. Automated proof techniques have been developed to help cope with this problem and to reduce the overall work load in producing proofs.

Automated theorem proving is the process of mechanically deciding if a program, traditionally formulated in a logical language, has some property. This works by proving that the result, formulated as a theorem in the logical language, follows logically from the program as premises. During software development, automated theorem proving is used statically to prove results about runtime properties of programs. Some automated theorem provers work automatically without programmer input, but often automated provers require direction from the user either in the form of helping it get over hurdles or by providing intermediate lemmas to be proved. Importantly, if a theorem is proved or checked by an automated theorem prover and the prover is sound, then the result is guaranteed to follow.

A well-known automated theorem prover is ACL2 [3] (A Computational Logic for Applicative Common Lisp), which is “industrial strength” GPL software for proving results about programs in an untyped variant of LISP. ACL2 takes as axioms the semantics of its programming language and user provided extensions of that language that maintain its

¹This slogan is often attributed to Robin Milner but is also used without citation by Pierce [10] on p. 168.

Figure 1: Operational Semantics of \mathfrak{P}

$pred (succ\ n) \rightarrow n$	$iszero\ 0 \rightarrow true$	$iszero (succ\ n) \rightarrow false$	$\frac{t_1 \rightarrow t'_1}{succ\ t_1 \rightarrow succ\ t'_1}$
$if\ false\ then\ t_2\ else\ t_3 \rightarrow t_3$	$pred\ 0 \rightarrow 0$	$\frac{t_1 \rightarrow t'_1}{pred\ t_1 \rightarrow pred\ t'_1}$	$\frac{t_1 \rightarrow t'_1}{iszero\ t_1 \rightarrow iszero\ t'_1}$
$\frac{t_1 \rightarrow t'_1}{if\ t_1\ then\ t_2\ else\ t_3 \rightarrow if\ t'_1\ then\ t_2\ else\ t_3}$		$if\ true\ then\ t_2\ else\ t_3 \rightarrow t_2$	

logical consistency. The main mechanism of proof is term rewriting: it works by systematically replacing terms with equivalent or implied terms until it has a trivial entailment. ACL2 is also extensible in that users can define theorems that are verified by ACL2 and used to extend ACL2's rewriting capacity. The prover is an extension of the previous Boyer-Moore theorem prover, Nqthm [1]. ACL2 has been used in industrial applications including proving the correctness of the floating point division operations of the AMD K5 microprocessor after the discovery of the Pentium FDIV bug [7].

Here, I detail some of my experience using ACL2 to prove the type soundness of a toy language. First, I'll describe the toy language and type system, which is given by Pierce in *Types and Programming Languages* [10]. Then, I'll describe how it is implemented in ACL2's programming language. I'll then show how type soundness of the language is proven in ACL2 by proving the progress and preservation theorems. Finally, I'll detail a failed attempt at the same proof, point out some avoidable pitfalls in proving soundness results in ACL2, and describe related work.

2. THE TOY LANGUAGE

The toy language used in this project is the small language defined by Pierce in chapter 3 of *Types and Programming Languages* [10]. I will call this language \mathfrak{P} . The grammar, semantics, and type system of the language are given in this section.

2.1 Toy Language Grammar and Semantics

The grammar of that language is defined below as the terms (t) and the values (v):

$$\begin{aligned}
 t &::= true \mid false \mid if\ t\ then\ t\ else\ t \mid \\
 &\quad 0 \mid succ\ t \mid pred\ t \mid iszero\ t \\
 v &::= true \mid false \mid n \\
 n &::= 0 \mid succ\ n
 \end{aligned}$$

The formal operational semantics of the language are defined in figure 1 and are easily inferable from the natural semantics of these terms.² `true`, `false`, and `succ*0` are values. `if t then t else t` acts like a normal conditional branch, `pred t` returns the predecessor of `t`, and `iszero t` evaluates to `true` or `false` depending on whether `t` is 0.

²The only peculiarity in the semantics is that `pred 0` evaluates to 0. This entails that the values (as you'd naturally think) are `true`, `false`, and any number of `succ` applied to 0.

Figure 2: Typing Relation of \mathfrak{P}

$true : Bool$	$false : Bool$
$0 : Nat$	$\frac{t_1 : Nat}{succ\ t_1 : Nat}$
$\frac{t_1 : Nat}{pred\ t_1 : Nat}$	$\frac{t_1 : Nat}{iszero\ t_1 : Bool}$
$\frac{t_1 : Bool\ t_2 : T\ t_3 : T}{if\ t_1\ then\ t_2\ else\ t_3 : Bool}$	

2.2 Toy Language Type System

The type system of \mathfrak{P} assigns to each well-typed term a type from $\{Bool, Nat\}$, with the intention that terms that are assigned `Bool` evaluate to booleans (i.e. `true` or `false`) and terms that are assigned `Nat` evaluate to natural numbers (i.e. some number of `succ` followed by 0).

The typing relation of the language is given in figure 2.

3. TYPE SOUNDNESS

As mentioned above, the intension of the type system is to only give a term a type if that term would evaluate to a value of the particular type with which the term is associated. But, this was a bit of a gloss. Really, all the type system aims to do is give a term a particular type if that term will evaluate *in one step* to a term of the same type. If we also know that any term will become a value after a finite number of small step evaluations (that is, that evaluation terminates on well-typed terms), then we can conclude that the type system guarantees that a term of a certain type will evaluate to a value of that type. But, this latter property is in general undecidable.³ Instead, all that the type system intends is that if a term is well-typed then it will evaluate in a one-step evaluation to a term of the same type. That the type system does this job is the essential message of the *type soundness theorem*.

THEOREM 1. (Type Soundness for \mathfrak{P}) *If $t : T$ and t is not a value, then there is a t' such that $t \rightarrow t'$ and $t' : T$.*⁴

³This is the halting problem. In the particular case of this toy language, evaluation will terminate, though. Pierce proves that on page 39.

⁴Notice that this formulation requires t' to be the same type

In other words, the type soundness theorem says that if t is not a value and is of type T , then t evaluates in one step to another term t' , which is also of type T .⁵ If this theorem holds, then a well typed term will never end up in a stuck state.

3.1 Proving Type Soundness

There's a popular slogan that "Safety = Progress + Preservation".⁶ The idea behind it is that there are fundamentally two different claims being made by the soundness theorem: first, if a term is well-typed, then it will not be a stuck state (i.e. it will either be a value or be able to be evaluated); second, if a well-typed term can be evaluated, then the result of a one-step evaluation of that term will also be well-typed. These ideas are more formally summarized as follows:

THEOREM 2. (Progress for \mathfrak{P}) *If $t : T$ and t is not a value, then there is a t' such that $t \rightarrow t'$.*

THEOREM 3. (Preservation for \mathfrak{P}) *If $t : T$ and there is a t' such that $t \rightarrow t'$, $t' : T$.*⁷

It is easy to see that the conjunction of Progress and Preservation entail type soundness. Further, this toy language satisfies both Progress and Preservation as shown by Pierce (p. 96-98). Therefore, the type system of \mathfrak{P} is type sound.

3.2 Why Prove It Again?

The type soundness result for \mathfrak{P} falls out of Pierce's proof of Progress and Preservation in his chapter 8, and for simple languages that will never be used for high stakes projects (like this language), this is probably a sufficient guarantee of safety. So, why prove it again?

There are several reasons why a mechanically proven or checked proof of type soundness is preferred. First, by checking it mechanically, we can rest assured that as long as the prover is sound, the proof is sound. We need not wonder if we've missed a corner case or made a mistake in the proof. Second, by using an automated theorem prover, we're able to prove the result about an actual implementation of the language, rather than merely an abstract specification of it. In this sense, we're more sure that the result holds for the actual code in question.⁸ Third, once the technology and our proficiency improves, automated proof techniques will be more easily and cheaply applicable to difficult and complex cases. Exploring these simple cases in detail will help

as t . This means that if the evaluation terminates, the normal form that the evaluation terminates with will be of that type. This appears to be strictly stronger than what some others take to be typical type soundness, where instead of requiring that t' have the *same* type as t , it merely requires that t' be well-typed. This formulation is fine for this toy language, though.

⁵Pierce doesn't give this formulation of type soundness explicitly. Rather, I reconstructed it from the discussion on page 95.

⁶See Pierce p. 95.

⁷See the caveat for the type soundness theorem in note 4.

⁸There are some reasons to wonder about the strength and accuracy of this proposed upside. I discuss this more below.

advance that progress. Therefore, I take it that it's preferable to have a mechanically proven or checked proof of type soundness.

4. PROVING IT AGAIN (IN ACL2)

ACL2 is an automated theorem prover and programming language, which is an extension of a sublanguage of Common Lisp. Whereas most common languages, including Common Lisp, have higher order functionals and iteration via loops and `for` clauses, ACL2's weak programming language is missing these elements. Because of this, it is difficult to test and prove results about most programs in ACL2 even when they're written in Common Lisp (but more on this later). It also proved difficult to implement this language in a natural way in ACL2's language.

The original attempt at an implementation of an evaluator of this language involved evaluating on strings of the language, reading and producing new strings that represented the one-step evaluation of the input string. This attempt made proofs in ACL2 of the safety of this language overly difficult (but more on this later, too). Instead, I implemented the language using Lisp lists. I'll call the following language of Lisp lists \mathfrak{P}_S :

$$\begin{aligned} t &::= '(T) \mid '(F) \mid '(i\ t\ t\ t) \mid '(0) \mid \\ &\quad '(s\ t) \mid '(p\ t) \mid '(z\ t) \\ v &::= '(T) \mid '(F) \mid n \\ n &::= '(0) \mid '(s\ n) \end{aligned}$$

This language is clearly isomorphic to \mathfrak{P} in a natural way, where `'(T)` maps to `True`, `'(i t t t)` maps to `if t then t else t`, etc. `PinLang` is an implemented a function in ACL2 that checks whether a list is in \mathfrak{P}_S and returns a boolean (either `T` or `nil` in ACL2). The source of that function is included in the appendix. Using that isomorphism, it's natural to define the intended operational semantics and typing relation on \mathfrak{P}_S . A reproduction of those is omitted here.

4.1 Evaluating and Typing \mathfrak{P}_S in ACL2

To prove the type soundness theorem for this language in ACL2, it was necessary to implement both a one-step evaluator and a type checker for the language in ACL2. One of the most challenging aspects of proving type soundness in ACL2 is producing admissible functions in ACL2's language. Two constraints are the most difficult on the programmer: First, every function in ACL2 must be a *total* function; that is, every function must be a function with a domain of all ACL2 objects. This means that no function can assume that its input will have any particular property or structure. The second, and more programming-time intensive, constraint is that upon defining the function, ACL2 must be able to prove that the function will terminate on any input. As the documentation states

The final conditions on admissibility concern the termination of the recursion. Roughly put, all applications of [the function] must terminate. In particular, there must exist a binary relation *rel* and some unary predicate *mp* such that *rel* is

well-founded on objects satisfying mp , the measure term m must always produce something satisfying mp , and the measure term must decrease according to rel in each recursive call, under the hypothesis that all the tests governing the call are satisfied.⁹

The first hurdle is overcome by doing checks on the input before the function application. For example, the typing function, as I implemented it, requires that the input be in the language using the `PinLang` test. The second hurdle is not as easily overcome. Overcoming it by trial-and-error seems to be the most effective method, but this can be extremely frustrating to someone learning to use ACL2. By using the language consisting of Lisp lists instead of strings, much of these issues were overcome. Using the following final version of the evaluator and type checker, ACL2 is able to admit these functions without additional lemmas of guidance in terms of the measure for proving termination.

The single step evaluator `PStep` is implemented here:

```
(defun PStep (x)
  (if (PinLang x)
      (case (car x)
        ('t '(t))
        ('f '(f))
        ('0 '(0))
        ('p
         (cond
          ((equal (cadr x) '(0)) '(0))
          ((and (equal 's (caadr x))
                (PisNV (cadadr x)))
           (cadadr x))
          (T (list 'p (PStep (cadr x))))))
        ('z
         (cond
          ((equal (cadr x) '(0)) '(T))
          ((PisNV (cadr x)) '(F))
          (T (list 'z (PStep (cadr x))))))
        ('s (list 's (PStep (cadr x))))
        ('i
         (cond
          ((equal '(t) (cadr x)) (caddr x))
          ((equal '(f) (cadr x)) (caddr x))
          (T (list 'i (PStep (cadr x))
                  (caddr x) (caddr x))))
          (T nil)))
        nil))
```

`PStep` evaluates the input list one step if it's in the language and can be evaluated. The Lisp functions `cadr` and `cadadr` return the second thing in the list and the second thing in the second list in the list, respectively. The other Lisp functions act similarly. An important difference between my implementation and the semantics given by Pierce is that in my implementation, all values evaluate to themselves. In

⁹This is taken from the documentation on `defun` available in the online documentation on events in ACL2 at www.cs.utexas.edu/users/moore/acl2/v3-4/DEFUN.html. The other conditions on function admissibility are given there as well.

Pierce's semantics, values do not evaluate. It is easily provable that values are the only inputs that evaluated to themselves, so there is an easy translation between my function and his. Using this formulation decreases the complexity of the proofs ahead. See the appendix for a commented version of this code detailing its functioning.

The type checker for this language is implemented as follows:

```
(defun Ptype (x)
  (if (PinLang x)
      (cond
       ((equal '(t) x) 1)
       ((equal '(f) x) 1)
       ((equal '(0) x) 2)
       ((and (or (equal (car x) 's)
                 (equal (car x) 'p))
              (equal (Ptype (cadr x)) 2))
          2)
       ((and (equal (car x) 'z)
              (equal (Ptype (cadr x)) 2))
          1)
       ((and (equal (car x) 'i)
              (equal (Ptype (cadr x)) 1)
              (equal (Ptype (caddr x))
                    (Ptype (caddr x))))
          (T nil))
       (T nil))
```

`Ptype` returns 1 iff the type of the input is `Bool` and returns 2 iff the type of the input is `Nat`. Again, see the appendix for a detailed explanation of the code.

Both of these functions are admissible in ACL2 and clear the two most challenging constraints on function admissibility mentioned above. Both functions are total and return `nil` for any input not in the language (that is the role of the `if (PinLang x)` check in both functions). Also, ACL2 is able to prove that both functions will terminate on any input by induction on the `ACL2-COUNT` of the input. In a loose sense, the `ACL2-COUNT` of a Lisp list is the length of the list plus the `ACL2-COUNT` of its sublists. ACL2 is then able to prove that on any input, either that input is not in the language (in which case the functions return `nil`) or the function is at most recursively called on a list of a smaller `ACL2-COUNT` than the input. Since `ACL2-COUNT` is well-founded, ACL2 concludes that the functions must terminate.

4.2 Type Soundness of \mathfrak{P}_S in ACL2

Recall that for \mathfrak{P}_S to be type safe is for whenever a term is well-typed (in this case by `Ptype`), then that term evaluates to a term of the same type by `PStep`. In the language of ACL2, that is,

```
(implies
 (or
  (equal (ptype x) 1)
  (equal (ptype x) 2))
 (equal (ptype x) (ptype (pstep x))))
```

Type Soundness for \mathfrak{P}_S

In other words, if the type of a term x is 1 or 2 (recall that 1 and 2 stand in for `Bool` and `Nat`, respectively). Then the type of the one-step evaluation of x is the same as the type of x .

In order to prove the type soundness of \mathfrak{P}_S in ACL2, first the Progress and Preservation theorems were proven. In ACL2, Progress for \mathfrak{P}_S is formulated as

```
(implies
  (or
    (equal (ptype x) 1)
    (equal (ptype x) 2))
  (not (equal (pstep x) nil)))
```

Progress for \mathfrak{P}_S

Recall that `PStep` only returns `nil` when the input is not a value and cannot be evaluated in a one-step evaluation. Therefore, this claim is equivalent to the claim that if x is well-typed, then x is a value or x evaluates in one step successfully, so this claim is equivalent to Progress.

Preservation for \mathfrak{P}_S is the claim that if a term is well-typed and evaluates to something else, then that too is has the same type. In ACL2, that's

```
(implies
  (and
    (or
      (equal (ptype x) 1)
      (equal (ptype x) 2))
    (not (equal (pstep x) nil)))
  (equal (ptype (pstep x)) (ptype x)))
```

Preservation for \mathfrak{P}_S

The antecedent requires that x is well-typed and x is either a value or can be evaluated in one step. The consequent then says that the type of the result of `PStep x` is the same as the type of x . Since this claim holds for values as well as nonvalues, it's strictly stronger than Preservation, so proving it suffices to prove Preservation. Together, Progress and Preservation for \mathfrak{P}_S imply Type Soundness for \mathfrak{P}_S .

After defining all of the functions included in the full source of the final implementation (included below), ACL2's `defthm` function proves both Progress and Preservation for \mathfrak{P}_S without further assistance. On a Pentium 2.6 GHz processor using ACL2 v3.4 in Windows XP, these proofs take about 90 seconds. ACL2 also succeeds in proving Type Soundness for \mathfrak{P}_S without first proving either of the other two theorems in about 90 seconds.

In other methods of proof, such as manual proof, breaking type soundness result down into two lemmas, Progress and Preservation, helps to guide the prover, but in proving in ACL2, the existence of the lemmas does not provide any necessary support to the reasoning.

This concludes the exposition of the successful attempt to prove the type soundness result of the toy language. I take it that this proof, as long as the implementation of the eval-

uator and type checker work as intended, is successful as a proof of the type soundness of the original language due to the trivial isomorphism between the languages.

5. A FAILED ATTEMPT

In this section, I'll briefly point out some pitfalls to be avoided in proving similar results mechanically in pure ACL2. First, I'll discuss a failed attempt at the result discussed above, then I'll discuss some concerns from a general perspective.

5.1 A Stringy Attempt

Before implementing the toy language as Lisp lists, an attempt was made to implement the language as strings. For instance, the string `"i(z0)(ps0)(0)"` would be equivalent to `if iszero 0 then pred succ 0 else 0`. The `PStep` and `Ptype` functions were implemented for this language as well. Testing shows that these functions appear to be equivalent to their Lisp list counterparts, but proving the type soundness result for the string implementation was significantly more difficult than proving the result for the Lisp list language. In fact, most of the proofs in the string language seemed to loop forever (some were left to run for days), where the counterpart proofs ran in a matter of minutes. In the end, it looked like ACL2 was hung up on trying to deal with the helper functions that `PStep` and `Ptype` used to locate the guard and branches of the conditional statements. As far as the author can tell, these functions are successful and manual proofs of their correctness are obvious. The reader can find that code on the author's website (given above).

The main problem that seemed to trip up ACL2 in proving soundness on the string language is the use of recursion in the helper functions. Whereas the Lisp list language pretty much only uses recursion to call the main functions (`PStep` and `Ptype`) on sublists of the input, in the string implementation, the helper functions had to recursively call themselves on the original input. These functions were used for separating different parts of the input. The result was that the existence of the recursion in the helper functions meant that, for that language, there was much more recursion in every call of the main functions, which resulted in difficult proofs for ACL2. ACL2 was able to admit these functions, so it found a proof of their termination, but it hung on proving that they succeeded in the same task as their list counterparts. This brings out one of the limitations of the project of using ACL2 to prove these results: what appears to be trivial increases in complexity in manual proofs may prove to be unsurmountable complexity increases for ACL2. Future work could be done in this area to prove the practical equivalence of the string and list implementations.

6. PROVING TYPE SOUNDNESS IN ACL2 AND RELATED WORK

As brought out above, the original motivations for producing a mechanized proof of the type soundness of languages were threefold: (1) By having a mechanized proof, we have a guarantee that as long as the prover is sound, the proof is sound. (2) By using an automated theorem prover, we're able to prove the result about an actual implementation of the language, rather than merely an abstract specification of it. (3) Once the technology and our proficiency improves,

automated proof techniques will be more easily and cheaply applicable to difficult and complex cases. In some sense, each of these three motivations is satisfied by this type of proof, but there are important sense in which the last two fail.

As to the last motivation, that in the future, using these types of proof methods will out strip manual proof capabilities: This is probably accurate, but it's important to notice that using an automated theorem prover to prove type soundness will, for the foreseeable future, require significant user guidance. The proof I've given uses only ACL2's built-in proving techniques and functions (except for functions defined explicitly for the language), but attempts to automate the process even more are under way.

For example, Swords and Cook [13] work to further the automation of these types of proofs by providing a new macro for pattern (regular expression) matching in ACL2. They also make significant contributions to automating this type of proof in ACL2 by examining issues that did not arise here such as dealing with variable bindings and recursive data types, which must be "flattened" for ACL2 to deal with them. Their project includes increasing ACL2's capabilities by adding new macros and reasoning abilities, which could allow for unsoundness to unintentionally creep in. The proof discussed here does not suffer from that (admittedly only slight) risk. Swords and Cook also point out that using their method requires more user interaction in some cases and fails to fully utilize all of ACL2's powerful heuristics in useful ways. It is clear, both from the work here and their work, that user interaction will be a large cost of this method of type soundness proof for the near future.

The second original motivation for the use of automated provers in proving these results was that the proofs can prove that actual implementations work, rather than abstract specifications thereof. This motivation can be satisfied by this method as long as the relevant specification that needs to be checked is implemented in ACL2's programming language, which is an extension of a weak subset of Common Lisp. ACL2 does not contain some very commonly used Common Lisp functions such as `loop`, `while`, and `map`. This means that in practice, really only a very small amount of actual implementations can be checked in pure ACL2, like this toy language was.

Again, the work of Swords and Cook [13] can help with this problem some, by helping to more easily implement languages in (the extended) ACL2. Also, the work in [4] helps to make partial functions, a typical feature of language implementations, available to ACL2's reasoning.

A new strategy has been suggested to help this problem in [2]. The authors of that project call the strategy "first proving and then ... testing." The idea behind that approach could be applied to this type of situation as follows: take a language that is already implemented, implement the language again in ACL2, prove the soundness results about the ACL2 implementation, then use model checking to confirm that the ACL2 implementation and the original act the same in all cases. If they do, then the proof carries over to the implementation in the other language. Of course, this is not,

strictly speaking, a *proof* of the soundness of the original language, but it is a relatively easily implementable alternative to one.

Of course, work has been done to help expand the power of ACL2 to reason about other languages, in general. For instance, see [5] and [6]. Also, a lot of work has gone into doing similar types of proofs in other provers, such as HOL [8]. For example, see [14], [9], [11] and [12], which works to link ACL2 and HOL.

7. CONCLUSION

The paper discussed some details of constructing a mechanically produced and verified proof of the type soundness of the simple language given by Pierce. By having a mechanized proof, we have a strong guarantee of the accuracy of the conclusion. But, there are many hurdles to be overcome before these techniques can be applied to *automatically* proving the type soundness of more complex or useful languages. Some progress towards this end has been achieved, but current techniques only practically allow for proofs for simple languages without significant programmer cost. It was pointed out that apparently small details, such as using strings instead of lists, which would make little difference in implementing the language for practical applications, can make the difference between being able to construct a proof easily versus failing to construct one at all. In the end, the methods and proof techniques were honed to allow ACL2 to produce a proof of the type soundness of this toy language in a matter of minutes.

8. REFERENCES

- [1] Nqthm, the Boyer-Moore prover. www.cs.utexas.edu/users/boyer/ftp/nqthm/, 1992. Information available at the website.
- [2] M. Andr es, L. Lamb an, and J. Rubio. Executing in Common Lisp, Proving in ACL2. In *Calcuemus '07 / MKM '07: Proceedings of the 14th symposium on Towards Mechanized Mathematical Assistants*, pages 1–12, Berlin, Heidelberg, 2007. Springer-Verlag.
- [3] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000. Also see the website at www.cs.utexas.edu/users/moore/ac12/.
- [4] P. Manolios and J. S. Moore. Partial functions in ACL2. *Journal of Automated Reasoning*, 31:2003, 2003.
- [5] J. S. Moore. Proving theorems about Java-like byte code. In *Correct System Design, Recent Insights and Advances*, pages 139–162. Springer-Verlag, 1999.
- [6] J. S. Moore. Mechanized Operational Semantics. www.cs.utexas.edu/users/moore/publications/talks/marktoberdorf-08/, 2008. lectures given at Marktoberdorf Summer School.
- [7] J. S. Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5K86 Floating-Point Division Algorithm. *IEEE Transactions on Computers*, 47, 1996.
- [8] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer-Verlag Heidelberg, 2002.

- [9] S. Owens and G. Peskine. Verifying Type Soundness for OCaml: the Core Language. One page abstract, Sept. 2007.
- [10] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
- [11] C. Pusch. Proving the soundness of a Java bytecode verifier in Isabelle/HOL. In *IN OOPSLA98 WORKSHOP FORMAL UNDERPINNINGS OF JAVA*, 1998.
- [12] M. Staples. Linking ACL2 and Hol. Technical report, Computer Laboratory, University of Cambridge, 1999.
- [13] S. Swords and W. R. Cook. Soundness of the Simply Typed Lambda Calculus in ACL2. In *Intl. Workshop on the ACL2 Theorem Prover and Its Applications (ACL2)*, 2006.
- [14] D. Syme and A. D. Gordon. Automating type soundness proofs via decision procedures and guided reductions. In *In 9th International Conference on Logic for Programming Artificial Intelligence and Reasoning, volume 2514 of LNCS*, page 418, 2002.

APPENDIX

A. SOURCE CODE OF \mathfrak{P}_S

This section includes the implementation of the relevant functions and theorems for the proof of the type soundness of \mathfrak{P}_S in ACL2.

These three functions are helper functions used by the main functions. `PinLang` decides if a list is in the language. `PisNV` decides if a list is a numeric value. `PisValue` decides if a list is a value.

```
(defun PinLang (x)
; returns T iff x is in language
; check that length of x = 1, 2, or 4
  (if
    (or (not (consp x))
        (not (or (equal (length x) 1)
                 (equal (length x) 2)
                 (equal (length x) 4))))
      nil
      ;check that it's wellformed if so
      (cond
        ((equal '(t) x) T)
        ((equal '(f) x) T)
        ((equal '(0) x) T)
        ; if length x = 2,
        ; then x is a succ, pred or iszero
        ((and (equal (length x) 2)
              (or (equal (car x) 's)
                  (equal (car x) 'p)
                  (equal (car x) 'z))))
         (PinLang (cadr x)))
        ; if it's length 4, then is an if
        ((and (equal (car x) 'i)
              (equal (length x) 4))
         (and (PinLang (cadr x))
              (PinLang (caddr x))
              (PinLang (caddrr x))))
         (T nil))))))
```

```
(defun PisNV (x)
; returns T iff x is a num val
  (cond
    ((equal (length x) 1)
     (equal (car x) '0))
    ((and (equal (length x) 2)
          (equal (car x) 's))
```

```
      (equal (length (cadr x)) 2))
      (PisNV (cadr x)))
    (T nil)))
```

```
(defun PisValue (x)
; returns T iff x is a value
  (if (PinLang x)
      (or
        (equal '(t) x)
        (equal '(f) x)
        (PisNV x))
      nil))
```

This is the main one-step evaluator for \mathfrak{P}_S :

```
(defun PStep (x)
; returns onestep eval of x
  (if (PinLang x) ; check x in language
      (case (car x) ; case on the first call in x
        ('t '(t)) ; t > t
        ('f '(f)) ; f > f
        ('0 '(0)) ; 0 > 0
        ; case that (car x) is p
        ('p
         (cond
           ; pred 0 > 0
           ((equal (cadr x) '(0)) '(0))
           ; p (s nv) > nv
           ((and (equal 's (caddr x))
                 (PisNV (caddrr x)))
            (caddrr x))
           ; t > t' then p t > p t'
           (T (list 'p (PStep (cadr x))))))
        ; case that (car x) is z
        ('z
         (cond
           ; z 0 > T
           ((equal (cadr x) '(0)) '(T))
           ; z s 0 > F
           ((Pisnv (cadr x)) '(F))
           ; t > t' then z t > z t'
           (T (list 'z (PStep (cadr x))))))
        ; case that (car x) is s
        ; t > t' then s t > s t'
        ('s (list 's (PStep (cadr x))))
        ; case that (car x) is i
        ('i
         (cond
           ; i (T) (a) (b) > a
           ((equal '(t) (cadr x)) (caddr x))
           ; i (F) (a) (b) > b
           ((equal '(f) (cadr x)) (caddrr x))
           ; t > t' then i(t)(a)(b) > i(t')(a)(b)
           (T (list 'i (PStep (cadr x))
                  (caddrr x) (caddrrr x))))
         (T nil)) ; else, cannot step
        nil)) ; not in language
```

Here is the type checker:

```
(defun Ptype (x)
; returns 1 iff type of x is Bool
; returns 2 iff type of x is Nat
  (if (PinLang x)
      (cond
        ;type of t, f, 0 = 1, 1, 2
        ((equal '(t) x) 1)
        ((equal '(f) x) 1)
        ((equal '(0) x) 2)
        ; if x is a pred or succ and
        ; if the cadr is a nat, then type x = 2
        ((and
          (or
```

```

      (equal (car x) 's)
      (equal (car x) 'p))
    (equal (Ptype (cadr x)) 2))
  2)
; if x is a iszero and
; cadr is type 2, then type x is 1
((and
  (equal (car x) 'z)
  (equal (Ptype (cadr x)) 2))
  1)
; if x is an if
; if type guard = 1
; if type of branches is equal
; if type of branches are nonnil
; type x = type of branch
((and
  (equal (car x) 'i)
  (equal (ptype (cadr x)) 1)
  (equal (ptype (caddr x))
    (ptype (caddr x))))
  (ptype (caddr x)))
(T nil)) ; nothing else is welltyped
nil)) ;not in lang

```

These are the three important theorems, progress, preservation, and soundness, as proven by ACL2:

```

(defthm progress (implies (or
  (equal (ptype x) 1)
  (equal (ptype x) 2))
  (not (equal (pstep x) nil))))

(defthm preservation (implies (and (or
  (equal (ptype x) 1)
  (equal (ptype x) 2))
  (not (equal (pstep x) nil))))
  (equal (ptype (pstep x)) (ptype x)))

(defthm soundness (implies (or
  (equal (ptype x) 1)
  (equal (ptype x) 2))
  (equal (ptype x) (ptype (pstep x)))))

```

All of this code, along with the code for the failed attempt at implementing this as strings, can be found at www.umich.edu/~singerdj/cs.html.