Abstract. We prove the equivalence of a function that efficiently recognizes XML name characters with its specification in ACL2. We conduct the proof by reducing the search space to a finite set of cases, then efficiently and exhaustively testing the remaining cases by executing a hand-written function. Our proof is possible because of the way ACL2 integrates executability and reasoning.

1 Introduction

ACL2 is a system that integrates a functional programming language, a first-order mathematical logic, and a general purpose theorem prover. The ACL2 user creates programs by defining functions in the programming language. Each of these definitions corresponds to a new axiom in the logic so that theorems in the logic are theorems about function executions. Using the integrated theorem prover, a user can prove theorems in the logic and hence can prove properties about the executions of their functions.

In this paper we take a typical “character recognition” production from the XML specification and formalize it in ACL2. Our formalization is executable, straightforward, and cleanly corresponds to the informal XML specification, but is too slow to be practically useful in XML parsing. To address this, we reimplement the function in a far more efficient manner, then prove that our efficient implementation is correct with respect to our straightforward formalization. This theorem is “shallow” in the sense that only the simplest of mathematics are required in order to prove it. Yet, it is also “wide” in that there are a vast number of cases that can overwhelm the theorem prover. To address this, we employ a technique that we call “proof by reduction to exhaustive testing”:

1. To begin, we show that no counterexamples exist anywhere except perhaps in some finite space.
2. We then write a custom testing function to check if this space is free of counterexamples. We prove that if the testing function is logically equivalent to true, then no counterexamples exist within the space.
3. We combine (1) and (2) to reduce our proof to showing that our testing function is logically equal to true. This can be done by appealing to the
relationship between ACL2’s logic and its programming language, that is, by simply running our function and seeing that it returns true.

This technique requires little human effort to set up the proof and allows the proof to be completed in a matter of seconds. Furthermore, this technique can be applied to any problem that can be reduced to a “practical” size, where “practical” is ultimately based on the speed of our computers and the ability of our underlying language to make efficient use of that speed.

The ACL2 language is ultimately compatible with Common Lisp [6], and ACL2 functions can be compiled using one of several Lisp implementations such as GCL and Allegro. In this paper we will use GCL, which translates our functions into C, then calls upon GCC to compile them into native machine code. As a result, our functions can be executed quite efficiently, and we have been able to apply this technique to prove a theorem involving the exhaustive testing of $2^{32}$ finite cases in under 2 minutes on a 2.8 GHz Pentium 4.

Various benefits from executability in theorem proving have been noted before. For example, executability allows the ACL2 user to quickly test his or her functions and discover bugs before becoming involved in a potentially time consuming proof effort [5].

ACL2 is also frequently used to model microprocessors [1] or virtual machines [4] and prove properties about these models. Efficient executability allows these models to serve “double duty” as as simulators with speeds that are close to traditional C models [2]. Also, in these efforts there is typically some physical artifact which the formal model is supposed to correspond to, and testing can be used to help validate this correspondence [5].

Still, we are not aware of any other work which has taken such advantage of the close relationship between the ACL2 logic and the execution of ACL2 functions as a method for proving theorems. Our work uses efficient execution as the driving force behind proofs, and we have been pleasantly surprised at the size of problems which can be approached with this methodology.

In the remainder of this paper, we will introduce a fast character recognition function and a formalization of its specification (Section 2). We then provide a detailed walk through of how our technique can be applied to prove that the function meets its specification (Section 3). We conclude with some final thoughts about our technique (Section 4).

2 Background and Motivation

XML is a popular and generic system for describing structured, textual data by nesting tags in a style similar to HTML. XML is ultimately based on Unicode [7], a character representation system which encodes roughly 1.1 million separate characters. We are developing an XML parser in ACL2, and in this work we represent Unicode characters simply as integers.

The XML Specification [3] describes the syntax of valid XML documents with an extended BNF grammar of over 80 productions. At the lowest levels
of this grammar are “character type” productions, similar to what one would find in a lexer for any programming language to define digits, letters, whitespace characters, and so forth.

Because Unicode has so many characters, these simple productions can actually be quite elaborate. We show a few of these productions in Figure 1, and in this paper we will look primarily at the NameChar production as our example. As you can see from the grammar fragment, NameChar ultimately includes the many ranges specified by BaseChar, as well as the many ranges specified for CombiningChar (which we have omitted for brevity).

To build our parser, we need to be able to recognize NameChars, which in turn requires us to be able to recognize these subsidiary productions. We begin by writing ACL2 formalizations of these grammar productions. This is all quite straightforward, for example we begin by writing the function BaseChar? to decide whether some input is a valid BaseChar.

\[ \text{(defun BaseChar? (x)} \]  
\[ \quad \text{(and (integerp x)} \]  
\[ \quad \quad \text{(or (in-range? #x0041 #x005A)} \]  
\[ \quad \quad \quad \text{(in-range? #x0061 #x007A)} \]  
\[ \quad \quad \quad \text{(in-range? #x00C0 #x00D6)} \]  
\[ \quad \quad \quad \ldots \]  
\[ \quad \quad \quad \text{(in-range? #xAC00 #xD7A3))}) \]

Function BaseChar? simply checks to see that its input is an integer (our representation of Unicode characters), and that this input is in one of the ranges as specified by the grammar production for BaseChar. This function comes directly from the XML Specification — in fact, we simply “copied and pasted” the grammar production into our ACL2 file and then applied some search/replace operations to create appropriate calls to our in-range? function. As you can see, the ranges come directly from Figure 1.

We write similar functions named Letter?, Digit?, CombiningChar?, and Extender?, then combine them all to create a recognizer for NameChars:

\[ \text{(defun NameChar? (x)} \]  
\[ \quad \text{(or (Letter? x)} \]  
\[ \quad \quad \text{(Digit? x)} \]  
\[ \quad \quad \quad \text{(equal x *period-character*)} \]  
\[ \quad \quad \quad \text{(equal x *dash-character*)} \]  
\[ \quad \quad \quad \text{(equal x *underscore-character*)} \]  
\[ \quad \quad \quad \text{(equal x *colon-character*)} \]  
\[ \quad \quad \quad \text{(CombiningChar? x)} \]  
\[ \quad \quad \quad \text{(Extender? x))} \]

These functions are satisfying formalizations of the XML Specification because of the obvious way they relate to the grammar productions. However, NameChar? is inefficient — it is just a litany of range tests. In fact, a rough timing loop indicates that this function is over 2,000 times slower than the equivalent XMLChar.isName function implemented in Java by the Apache Xerces XML
Fig. 1. Some Low Level Productions in the XML Grammar
parser. If our XML parser is to be efficient, our low level character recognition functions will need to do better than this.

After visually examining the ranges given in BaseChar, Ideographic, Digit, and so forth, we suspect that all NameChars can be found in the range [0, 65535]. Since this is not a daunting number, we imagine that building a lookup table mapping x to (NameChar? x) would not be prohibitively expensive in terms of memory consumption. We construct this table, name it nctable, and write a new, “fast” version of our NameChar? function which is essentially the following:

\[
\begin{align*}
\text{(defun fast-namechar? (x)} & \text{ (and (integerp x)} \\
& \text{ (in-range? x 0 65535)} \\
& \text{ (aref1 'nctable *nctable* x))} // look up nctable[x]
\end{align*}
\]

This function is about 1,025 times faster than our original NameChar function, but still only about half as fast as the Xerces version. The Xerces implementation uses essentially the same approach that we do, and this speed difference seems to be intrinsic to differences between array access in ACL2 and Java.

3 The Theorem and Proof

We would like to know that fast-namechar? is a correct implementation of the NameChar production. If we are willing to accept that NameChar? is a correct formalization of this production (and we are), then we can do this by simply showing that fast-namechar? is equivalent to NameChar?. In other words, we would like to prove that:

\[
\forall i, (\text{fast-namechar? } i) = (\text{NameChar? } i)
\]

Before we begin, we should say a few words about the correspondence between arrays in the ACL2 language and the ACL2 logic.

The logic of ACL2 is a first-order logic which only “knows” about numbers, symbols, strings, characters, and pairs. We will denote the pair of \( a \) and \( b \) with \( \langle a, b \rangle \). Lists are typically built recursively from pairs, and we will denote them with square brackets, e.g., \([1, 2, 3]\).

The ACL2 language supports these same kinds of objects, but also supports some things which are not part of the Logic. For example, efficiently accessible arrays are available (and we use them in fast-namechar?), but in the Logic, arrays are instead treated as lists whose elements are pairs of \( \langle \text{index}, \text{value} \rangle \), and arrays are accessed by using simple recursive functions that traverse the list looking for the correct index.

So, what are we asking ACL2 to prove when we pose the above conjecture? Essentially we are trying to show a correspondence between the 216-element list nctable = \([\langle 0, \text{false} \rangle, \langle 1, \text{false} \rangle, \ldots, \langle 61, \text{true} \rangle, \ldots, \langle 65535, \text{false} \rangle]\) with the enormous if-nest that NameChar? ultimately expands to.
This theorem is “shallow” — that is, it seems that there is little in the way of mathematics that would be useful to us here — but it is also “wide” in the sense that the list is very large and our if-nest is quite elaborate. If we ask ACL2 to try to prove this theorem without help, the system is not able to cope with all of these cases.

However, we can help to guide the system to find the proof. To do so, we will employ a technique that we call “reduction to exhaustive testing.” This is a four step process, and we describe each step in detail below.

**Step 1** Our first step is to show that if any counterexample exists, it lies within some finite space. To show this we will need to prove the two functions always return the same value for all but some finite set of inputs. We make the following observations:

- Both functions return false unless $i$ is an integer.
- Both functions return false if $i$ is negative.
- Both functions return false if $i$ is greater than 65535.

Each of these observations required some small level of human involvement, but truly these are easy properties to identify by examining the grammar productions making up NameChar, and ACL2 can prove each of these statements without our guidance. Taken together, these observations limit the search space to just $2^{16}$ cases, one for integer each $i \in [0, 65535]$. In the concrete syntax of ACL2, we have the following defthm:

```
(defthm lemma-1
  (implies (or (not (integerp i))
              (< i 0))
              (<= 65536 i))
  (equal (fast-namechar? i) (NameChar? i)))
```

**Step 2** The next step is to write an efficient function to ensure the remaining space has no counterexamples. We write a trivial recursive function that, given some input $n$, first checks to ensure that the two functions agree for $n$, then recursively calls itself until it reaches 0. (The function zp returns true whenever its argument is not a positive integer.)

```
(defun namechar-equiv tester (n)
  (and (equal (fast-namechar? n) (NameChar? n)) // test n
       (or (zp n) // stop at n = 0
           (namechar-equiv tester (- n 1))))))) // recur on n - 1
```

The function searches the space from $[0, n]$ for any $n > 0$, and returns true if no counterexamples are found. This function can be ultimately be compiled into machine code and efficiently executed for particular values of $n$. Ultimately, we
will want to have the theorem prover use this function in order to test the space [0,65535] for counterexamples.

But before we continue, we probably want to run this function to see if it can find any counterexamples. We test the function on the input \( n = 65535 \), and find that it returns true. In other words, the test is successful and there are no counterexamples in this range. By running this test, we have not proven anything, but we are now convinced that our proof method will work. Note that if the function had returned false, we could easily augment it to output a message explaining where it found a counterexample, and we could then use that information to help track down the error in our code.

**Step 3** We now prove that our testing function works correctly. That is, we prove that if our test function returns true, then any point in the space it has searched is not a counterexample. We submit the following as a theorem for ACL2 to prove for all \( i \) and \( n \):

\[
i \in \mathbb{Z} \land n \in \mathbb{Z} \land 0 \leq i \leq n \land (\text{namechar-equiv-tester } n) \Rightarrow (\text{fast-namechar? } i) = (\text{NameChar? } i)
\]

This is an easy proof by induction. As a basis case, suppose that \((zp n)\) is true, i.e., that \( n \) is not a positive integer. Since \( n \in \mathbb{Z} \) and \( 0 \leq i \leq n \), we conclude that \( i = n = 0 \). Now since \((\text{namechar-equiv-tester } n)\) and \( n = 0 \), we have \((\text{namechar-equiv-tester } 0)\). From the logical definition of \(\text{namechar-equiv-tester}\), we conclude \((\text{fast-namechar? } 0) = (\text{NameChar? } 0)\) and since \( i = 0 \), we have \((\text{fast-namechar? } i) = (\text{NameChar? } i)\).

For the inductive case, suppose that \((zp n)\) is false and inductively assume our conjecture holds for \( n - 1 \). In other words, our inductive hypothesis is: if \( i \in \mathbb{Z} \land n - 1 \in \mathbb{Z} \land 0 \leq i \leq n - 1 \land (\text{namechar-equiv-tester } n - 1) \), then \((\text{fast-namechar? } i) = (\text{NameChar? } i)\).

Since \((\text{namechar-equiv-tester } n)\), we conclude (1) \((\text{fast-namechar? } n) = (\text{NameChar? } n)\), and (2) \((\text{namechar-equiv-tester } n - 1)\). Now consider two cases: if \( i = n \), by (1) we know that \((\text{fast-namechar? } i) = (\text{NameChar? } i)\) and we are done. Otherwise, \( i \neq n \), so since \( 0 \leq i \leq n \) we see that \( 0 \leq i \leq n - 1 \). But this means that we can use our inductive hypothesis and conclude that \((\text{fast-namechar? } i) = (\text{NameChar? } i)\), and we are done. \(\square\)

In the concrete syntax of ACL2, this `defthm` looks like the following:

```lisp
(defthm lemma-2
  (implies (and (namechar-equiv-tester n)
                (integerp n)
                (integerp i)
                (<= 0 i)
                (<= i n))
            (equal (fast-namechar? i) (NameChar? i)))
```
Step 4 We now combine our previous steps in order to prove our goal theorem. Loosely speaking, by lemma-1, we know that any \( i \) which is not an integer in the range of \([0, 65535]\) satisfies our conclusion. By lemma-2, we know that if 
\((\text{namechar-equiv-tester } 65535)\) is true, then every integer in \([0, 65535]\) satisfies our conclusion, covering the remainder of the proof. We have already convinced ourselves in Step 2 that 
\((\text{namechar-equiv-tester } 65535)\) will return true, so we think that this should complete the proof.

To have ACL2 accept our goal as a theorem, we submit the following \texttt{defthm} to ACL2 along with a hint that suggests that ACL2 should instantiate \texttt{lemma-2} with \( n = 65535 \). In the concrete syntax of ACL2, our final theorem along with this hint looks like the following:

\[
\text{(defthm fast-namechar-is-namechar)}
\begin{align*}
&\text{(equal (fast-namechar? i) (namechar? i))} \\
&\text{:hints(('"Goal"}} \\
&\text{:use ("instance lemma-2 (n 65535)")))}
\end{align*}
\]

This hint instructs ACL2 to create an instance of \texttt{lemma-2} where \( n = 65535 \). In other words, ACL2 considers the following fact:

\[
(\text{implies (and (namechar-equiv-tester 65535)} \text{)} \text{(integerp 65535)} \text{)} \text{(integerp i)} \text{)} \text{)} \text{(<= 0 i)} \text{)} \text{)} \text{(<= i 65535)} \text{)}) \text{(equal (fast-namechar? i) (NameChar? i))})
\]

Since 65535 is the only argument to \texttt{namechar-equiv-tester} and is a constant, ACL2 can simplify \texttt{(namechar-equiv-tester 65535)} by executing it. This is valid because the ACL2 logic corresponds with ACL2 function execution. This execution takes a couple of seconds. Similarly, ACL2 evaluates \texttt{(integerp 65535)} to true. Altogether, we are left with the following fact:

\[
(\text{implies (and (integerp i)} \text{)} \text{)} \text{(<= 0 i)} \text{)} \text{)} \text{(<= i 65535)} \text{)}) \text{(equal (fast-namechar? i) (NameChar? i))})
\]

Essentially, the above fact shows us that our conclusion is true for the entire space that we were unable to cover in \texttt{lemma-1}, and ACL2 discovers that it can appeal to \texttt{lemma-1} to finish the proof. In the end, ACL2 accepts \texttt{fast-namechar-is-namechar} as a theorem.

4 Conclusions

We have now covered the entire proof in detail. As you can see, this proof process is quite straightforward. The entire sequence takes only around 3.5 seconds to
replay in ACL2 2.9 on a 2.8 GHz Pentium 4. In the end, we are able to convince ACL2 that our implementation of \texttt{fast-namechar?}, which is roughly 1,025 times faster than \texttt{NameChar?}, computes the same function.

Our method is to first reduce the search space to a finite size, then to exhaustively test the remaining space for counterexamples. This technique makes extensive use of ACL2’s ability to use the results of efficient execution in order to prove logical theorems. The proof required little understanding of the functions \texttt{fast-namechar?} and \texttt{NameChar?}, and allows ACL2 to quickly prove that the theorem is indeed true for all inputs.

There are two major restrictions on the applicability of this method. First, the proof must be reducible to a finite number of cases through traditional proof techniques. (In our example, we accomplished this in Step 1.) Secondly, the number of remaining cases must be small enough that we can exhaustively test them in a reasonable amount of time. (In our example, it takes only seconds.) When the method is applicable, it removes the need to really understand in any deep sense the way that our functions operate, and allows counterexamples to be reported easily.

Since discovering this method, we have used it in several proofs related to our XML parser and the processing of Unicode files. We have been happily surprised with the range of problems it can solve. Our most impressive result comes from the verification of our UTF-8 decoder, where we have extended this technique to a proof involving four integer inputs which each range from \(0 \ldots 255\) for a total of \(2^{32}\) cases. The theorem is complicated, but essentially states that the result of decoding is correct with respect to Tables 3-5 and 3-6 in the Unicode Specification [7]. The proof takes just under 2 minutes on the same 2.8 GHz Pentium 4 machine, and involves the use of four testing functions (one for each input), and four lemmas in the spirit of \texttt{lemma-2}. Larger input spaces may be possible if the user is willing to work to tune the efficiency of their functions.

We owe our success to the ability of ACL2 to use the results of function execution in proofs, and the ability of ACL2 functions to be executed efficiently. When using GCL, ACL2 can ultimately be translated into machine code and can consequently run quite efficiently. We expect that the size of problems which can be reasonably solved in this way will grow as our computers get faster and as our compilers are optimized for greater efficiency.

These proofs may make interesting challenging problems for provers which do not support efficient execution to this degree.

References


