Code-Carrying Theory

Aytekin Vargun
Rensselaer Polytechnic Institute
110 8th St., Troy, NY, 12180, USA
varguna@gmail.com

David R. Musser
Rensselaer Polytechnic Institute
110 8th St., Troy, NY, 12180, USA
musser@cs.rpi.edu

ABSTRACT

Code-Carrying Theory (CCT) is an alternative to the Proof-Carrying Code (PCC) approach to secure delivery of code. With PCC, code is accompanied by assertions and a proof of correctness or of other required properties. The code consumer does not accept delivery unless it first succeeds in generating theorems, called verification conditions, from the code and assertions and checking that the supplied proof proves these theorems. With CCT, instead of transmitting code explicitly, only assertions and proofs are transmitted to the consumer. If proof checking succeeds, code is then obtained by applying a simple tool called CODEGEN to the resulting theory. This paper explains the design and implementation of CCT and shows how it can be used to achieve secure delivery of code with required correctness or safety properties. All the tools used in the verification steps are implemented in ATHENA, which is both a traditional programming language and a deduction language.

Categories and Subject Descriptors

F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical verification, Specification techiques; D.2.0 [**General**]: Protection mechanisms; K.6.5 [**Security and Protection**]

General Terms

Theory, Verification

Keywords

secure code delivery, correctness, memory safety, proof-carrying code

1. INTRODUCTION

Proof-Carrying Code (PCC) [1, 3, 11, 15, 16] is a means of protecting software consumers from malicious or inadvertent corruption of code that arrives from an untrusted source or through an insecure network. Important examples of such mobile code include Web applets, actor-based distributed system software, and updates to embedded computers. This proof-based protection approach is based on a policy of not accepting delivery of new code unless it is accompanied by a formal proof of correctness that can be machine-checked by the code consumer.

Most PCC research has focused on safety requirements (e.g., no out-of-bounds memory references are allowed) and, to a lesser-extent, security requirements (e.g., no unauthorized access to classified data), but in principle the method could be used with functional correctness requirements (e.g., the output of a sorting algorithm must be an ordered permutation of its input). However, proofs of functional correctness are inherently more difficult to automate than those of safety or security properties, and from the research literature it is apparent that many researchers today consider proof-based approaches to be infeasible unless proofs can be fully automated. That is, if the required proofs must be constructed under human direction, the level of expertise and effort is considered to be so high as to be impractical. The work reported in this paper is intended to challenge this view, in addition to presenting a somewhat different take on the proof-based approach to assurance that we call Code-Carrying Theory (CCT).

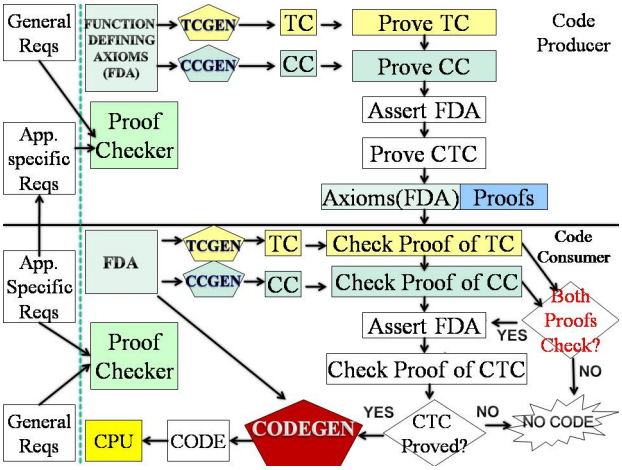

Figure 1: Code-Carrying Theory (CC stands for Additional Consistency Condition, TC is Termination Condition, CTC is Correctness Condition, and FDA is Function Defining Axioms)

2. CODE-CARRYING THEORY

Figure 1 illustrates the basic Code-Carrying Theory approach in detail. With CCT, instead of transmitting code, assertions, and proofs explicitly, as in PCC, only assertions and proofs are trans-

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil

mitted. The assertions transmitted are limited to axioms that define new function symbols and can be readily checked by the software consumer to do so properly as discussed in the next section. The proofs, expressed in a Denotational Proof Language [4, 5, 6], take the form of executable methods, which, when executed in the presence of the new axioms and ones already held by the consumer, prove additional propositions as theorems.

From the resulting theory—axioms and theorems—code is extracted by a simple process that perfectly preserves the semantics expressed in the theory.[1] While PCC can be said to be a variation on the traditional program verification approach, CCT can be viewed as a variation on the traditional proof-based program synthesis approach. When compared to ordinary methods of software development, the proof-based synthesis approach has much higher demands on human time and expertise than seem justified for most software. However, in light of the new demands of mobile code with high-stakes requirements, such as those of safety-critical embedded systems or code to be distributed to millions of consumers, investing in the time and resources required to write theorems and generate the proofs is much more justifiable.

2.1 Relation to PCC Research

We are working in a higher-level programming language than one finds being used in most of the PCC work. Of course, in the case of CCT the "source language" is actually a logic as defined and implemented in the proof checking system, so when we talk about the programming language involved we mean the language in which the extracted code is expressed, which in our case is the computational language of ATHENA [4, 5, 6], a high-level language. Previous research such as Necula and Lee's [15] and Appel's [2] has been carried out with very low-level languages (machine code or Java byte code). Working with programs at this low level makes complete, or nearly complete, automation of proofs necessary, and this is turn limits the applicability of the approach to restricted classes of programs and properties for which automated proof is feasible. By contrast, by working with programs at a high level of abstraction and requiring automation only of proof checking, as opposed to proof-finding, we can apply our approach to a larger class of program properties, including deep functional correctness.

2.2 Athena

For human proof construction and machine checking, we are working in a Denotational Proof Language (DPL) [4], which provides separate but intertwined languages for ordinary programming and logical deduction. The DPL system we are using, ATHENA [7], provides an ML-like programming language (but with Scheme-like syntax) in which programs are typically expressed as collections of recursive function definitions, and a structurally similar language for expressing deductive methods whose executions carry out proofs using both primitive inference methods of first order logic with equality (of which there are about a dozen, such as modus ponens, existential generalization, universal specialization, etc.) and "calls" of other deductive methods previously defined by the user or imported from a deductive-method library. Both of these languages are high level by most programming language standards, offering, for example, higher-order functions (and methods)—the ability to pass functions/methods to a function/method or return them as results.

As a deductive programming language, ATHENA makes it possible to formulate and write proofs as function-like constructions called *methods* whose executions either result in theorems if they properly apply rules of inference, or stop with an error condition if they don't. Those that succeed update the *assumption base*, an associative memory of propositions that have been asserted or proved in a proof session. The assumption base is fundamental to ATHENA's approach to deduction; all proof activity centers around it in such a way that ATHENA method bodies are both human-readable and efficiently machine-checkable.

2.3 Code-Carrying Theory Steps

We first outline the steps of the CCT "protocol" in general terms, then illustrate each step in the case study presented in the following section.

Step 1: Defining Requirements (Code Consumer)

The code consumer is responsible for the construction of requirements, which are of two types: *general* and *application-specific*. General requirements contain declarations and axiomatic definitions of some common symbols which are being used frequently in CCT. We have also proved new theorems from this initial set and built *theory libraries* which are ready to be loaded as part of the general requirements. Both consumers and producers have the same set of general requirements initially. This reduces redundant transfers of a large set of definitions and proofs between them.

Code consumers use application-specific requirements to describe new functions. These requirements include a new symbol for each new function and propositions specifying its correctness property or properties. Initially code producers do not have these types of requirements and receive them from the consumers later in the process.

Step 2: Defining a New Function (Code Producer)

After the code producer receives the specifications of a function requested by the consumer, it enters the symbol declarations into the theorem prover (ATHENA in our case) and sets up the specified properties as goals to be proved. The code producer wants to send to the code consumer an efficient implementation of the specified function and a proof that it satisfies the required specification. Rather than sending the efficient function as actual code, the producer will send only definitions and proofs.

Step 3: Proving Termination (Code Producer)

Looking ahead, the consumer, after receiving the function definition in the form of a set of axioms, must decide whether to accept the axioms or not. In general the consumer *cannot accept and assert* arbitrary axioms and propositions. It is the producer's responsibility to show that the new definitions do not introduce any *inconsistency* into the assumption base. This can be done by proving that the new function-defining axioms satisfy a *definitional principle*. In logics in which a function is defined by a single recursive equation, such as the Boyer-Moore computational logic [17], the definitional principle takes the form of a requirement that the equation defines the function totally, i.e., the function terminates for all inputs. We enforce this requirement also for function definitions in the form of a set of axioms, but in this case there is an additional consistency requirement: that the axioms together do not impose such strong requirements that there could not exist a function satisfying them.

The main disadvantage of Boyer-Moore approach is having first to define a separate well-founded ordering to prove termination. Our approach [20, 21] is similar to the ordering-relation approach, but instead of defining an ordering relation we construct the proof of termination as a proof by induction that reflects the recursion structure in the axioms. We have implemented a program called TCGEN which generates a *termination axiom* corresponding to each

[1] An exception to the exact semantic correspondence is the code for memory-updating operations, as explained in Section 5.

axiom that defines the function to be extracted. In addition to these axioms, TCGEN produces a *termination condition* (TC) for each function. We then prove these TC's using the termination axioms.

Step 4: Proving the Additional Consistency Condition (Code Producer)

We also wrote a program called CCGEN which takes a list of function-defining axioms and produces a predicate that we call the (additional) *consistency condition* (CC). As with the termination condition, the producer must construct a proof of this condition.

The additional consistency condition expresses that it is possible to define a function that satisfies the conditions in the axioms: corresponding to every sequence of values $x_1 \ldots x_k$ in the function domain there exists a range value y that we can associate with $x_1 \ldots x_k$ to form a function (considered as a set of ordered tuples $(x_1 \ldots x_k, y)$). That is, the axioms are not so strong that they rule out the existence of a function satisfying them.

Both the termination and additional consistency proofs have to be done before proceeding with the next step. If they cannot be done then the producer has to return to Step 2 and revise the function-defining axioms or write new ones. But if both of these conditions are proved then the new axioms can be asserted and used safely.

Step 5: Proving Correctness (Code Producer)

Even if the consistency and termination requirements are satisfied, the new definitions will not be accepted by the consumer without a *correctness proof*. The code producer has to convince the consumer that the new function satisfies the correctness conditions that the consumer requires. In this step, after asserting the function-defining axioms, the producer attempts to construct a proof of each of the required correctness conditions using those axioms, along with any axioms and theorems in the theory library that are known by the producer to be available also in the consumer's theory library.

Once all the proofs are ready, the producer can send them to the consumer along with the function-defining axioms and proofs of the termination and additional consistency condition. Thus *the producer does not send actual code but the theory which carries it*.

Step 6: Termination Checking (Code Consumer)

On receiving the theory and proofs for a new function the consumer first runs TCGEN to obtain the termination condition (TC). Since both the consumer and producer use TCGEN to get this condition, the termination proof sent by the producer must prove it, unless it has been corrupted in transmission or hacked by an intruder, in which case the consumer will reject the theory immediately, not allowing code to be generated from it. If the termination proof checks, the consumer proceeds to the next step.

Step 7: Additional Consistency Checking (Code Consumer)

The consumer next runs CCGEN and recreates the same additional consistency condition as the producer did; again, unless it has been changed in transmission, the consistency proof provided by the producer will prove this condition. (In our current CCT experiments, the transmitted "proof" just takes the form of a single command to the external SPASS prover to automatically search for the proof.) Otherwise, the function-defining axioms will be rejected.

Step 8: Checking Correctness (Code Consumer)

If we reach this point it is safe to assert the new function-defining axioms into ATHENA's assumption base. The consumer then checks the proofs supplied by the producer of the required correctness properties. If each such proof is accepted, the next and final step of generating code from the axioms can be carried out.

Note that *proving theorems (manually or by automated search) is generally much harder and more time-consuming than checking a given proof.* In CCT, generally the producer is responsible for providing the proofs whereas the code consumer is only responsible for checking them. If the producer sends all the proofs as manual proofs (ATHENA-only, with no use of an external automated prover), this reduces the consumer's workload. Also the trusted computing base is smaller since the consumer relies on less software (no need for an automated theorem prover). On the other hand, if the consumer has enough resources, some simple properties like CC and TC can be proved automatically with an external automated prover like SPASS [22]. Our experience shows that proving correctness conditions is usually harder than proving the termination or additional consistency conditions. Therefore, we believe the producer should always provide manual proofs for the correctness conditions; for TC's and CC's, we leave it as optional to use automation since their proofs can be found fairly quickly anyway.

Step 9: Code Extraction (Code Consumer)

The consumer now knows that the new definitions are acceptable. The form of the function-defining axioms is such that it is easy to extract executable code from them, for which purpose we implemented a function called CODEGEN. (Some theorem provers, Coq for example, provide code generation, but ATHENA does not.) In general the axioms defining new functions are in the form of universally quantified, and possibly conditional, equations. Each axiom is used in extraction of one clause of a function definition. The left hand side of each equation is an expression in which the function is applied to arguments of a specific form. The code extractor generates cases from the patterns of constructors in function arguments. The right hand side of the equation, with appropriate substitutions, becomes the expression to be evaluated by the clause. These expressions may contain calls to the function itself or to other recursive functions. If there are any conditional axioms, CODEGEN first checks different argument patterns and creates cases, which are then translated into *if-then* expressions accordingly. When there are two *if-then* expressions with opposite conditions, they are combined into one *if-then-else* expression. The target language of CODEGEN is Oz [10, 19] or ATHENA's computational language.

3. CASE STUDY: A PURE FUNCTION EXAMPLE

In this section we illustrate the CCT approach with a simple example, in which a small theory is developed from which a pure function for summing values in a sequence is extracted. CODEGEN can also produce memory-updating functions, examples of which are briefly discussed in Section 5.

In most of our experiments with CCT we have developed generic functions that operate on iterator ranges, modeling generic functions from the C++ Standard Template Library (STL), such as a summation function much like the STL generic `accumulate` function. Iterator ranges are pairs of abstract pointer-like objects delimiting a range of locations, which could be contained in an array, vector, list, deque, etc.; functions like `accumulate` work with any of these representations. In this section, for brevity and to concentrate on illustrating the CCT steps, we avoid introducing our theories of iterators and ranges and simply use lists. In ATHENA, we can inductively define lists with the following declaration, where `T` is a

type variable that can be instantiated with any type, and `Nil` and `Cons` are introduced as list constructors, as in Lisp or Scheme (but constructing only homogeneous lists):

```
(datatype (List-Of T) Nil
                      (Cons T (List-Of T)))
```

In addition to declaring the type specifications of constructors, a `datatype` declaration also implicitly defines a corresponding induction principle for use in a `by-induction` proof step, as will be illustrated in Step 5.

Our definitions and proofs are in most cases also generic in their assumptions about value types and operations, such as only assuming that summation is applied to values from a monoid, with an associative binary operation `Plus` and identity element `Zero`. To avoid the complications of this genericity in this case study, we just assume the values to be summed are natural numbers, which we introduce with

```
(datatype Nat zero (succ Nat))
```

and for which we define functions such as `Plus` axiomatically. The theory and proofs for the generic case are given in [20].

Step 1: Defining Requirements

The consumer declares an operator `sum-list@` and defines it by axioms, as the specification of the function, and sends them to the producer.

```
(declare sum-list@ (-> ((List-Of Nat)) Nat))

(define sum-list-empty-axiom
  (= (sum-list@ Nil) zero))

(define sum-list-nonempty-axiom
  (forall ?L ?x
     (= (sum-list@ (Cons ?x ?L))
        (Plus ?x (sum-list@ ?L)))))
```

The consumer also declares and specifies the function to be defined by the producer.

```
(declare sum-list (-> ((List-Of Nat)) Nat))
```

The following correctness condition (CTC) is part of the requirements and must be satisfied:

```
(define sum-list-correctness
  (forall ?L
    (= (sum-list ?L) (sum-list@ ?L))))
```

At the end of this step, the consumer sends the CTC and all the declarations and definitions to the producer.

Step 2: Defining the New Function Axiomatically

The producer defines `sum-list` as a more efficient way of computing the same function as `sum-list@`.

```
(define sum-list-empty
 (= (sum-list Nil) zero))

(define sum-list-nonempty
  (forall ?L ?x
     (= (sum-list (Cons ?x ?L))
        (sum-list1 ?L ?x))))

(define sum-list-axioms
        [sum-list-empty sum-list-nonempty])
```

Note that `sum-list` calls a helper function, `sum-list1`, which can be expressed with tail-recursion. The producer next declares and defines `sum-list1`:

```
(declare sum-list1 (-> ((List-Of Nat) Nat)
                       Nat))

(define sum-list1-empty
  (forall ?x
    (= (sum-list1 Nil ?x) ?x)))

(define sum-list1-nonempty
  (forall ?L ?x ?y
   (= (sum-list1 (Cons ?y ?L) ?x)
      (sum-list1  ?L (Plus ?x ?y)))))

(define sum-list1-axioms
  [sum-list1-empty
   sum-list1-nonempty])
```

Step 3: Proving Termination

Termination must be proved for both `sum-list` and `sum-list1`. The code producer first declares a new predicate function symbol, `sum-list1_t`, in order to express whether or not `sum-list1` terminates:

```
(declare sum-list1_t (-> ((List-of Nat) Nat)
                         Boolean))
```

TCGEN produces the following termination condition (TC) from `sum-list1-axioms`:

```
(forall ?L ?x
  (sum-list1_t ?L ?x))
```

and the termination axioms produced by TCGEN are

```
(forall ?x
   (sum-list1_t Nil ?x))

(forall ?L ?x ?y
  (if (sum-list1_t ?L (Plus ?x ?y))
     (sum-list1_t (Cons ?y ?L) ?x)))
```

Some symbols like `Plus` and `Cons` represent functions that are known to be terminating. These symbols are stored in a table that is read by TCGEN to simplify the termination axioms. The code producer asserts these termination axioms into the assumption base and constructs a proof of the termination condition. The proof is a simple induction proof based on cases defined by ATHENA from the form of the `List-Of` datatype. This and the other proofs of the theorems in this paper are presented in detail in [20].

In the next step, the following termination axioms are produced to prove termination of `sum-list`.

```
(sum-list_t Nil)

(forall ?L ?x
   (if (sum-list1_t ?L ?x)
       (sum-list_t (Cons ?x ?L))))
```

and the termination condition to be proved is

```
(define sum-list-termination
  (forall ?L
    (sum-list_t ?L)))
```

Step 4: Proving Consistency

For the proof of the (additional) consistency condition for `sum-list1`, application of CCGEN to the `sum-list1` axioms produces

```
(forall ?y ?L ?x ?L2
  (exists ?result
    (and (if (= ?L2 (Cons ?y ?L))
             (= ?result (sum-list1
                         ?L (Plus ?x ?y))))
         (if (= ?L2 Nil)
             (= ?result ?x)))))
```

This condition is easily proved with a call to SPASS:

```
(!prove-from (consistency-condition
              sum-list1-rules)
             (datatype-axioms "List-Of"))
```

SPASS carries out this proof using only the predefined axioms for the `List-Of` data type. Similarly, the producer proves the CC for `sum-list` that is produced by CCGEN.

Step 5: Proving Correctness

Since the termination and additional consistency conditions for the given axioms are proved, these axioms can be asserted into the assumption base in order to prove the correctness condition (CTC).

As in common mathematical practice, we frequently break down and organize proofs with the aid of lemmas. Once a lemma's proof has been constructed, it is available for use whenever the lemma is needed in any other proof. Factoring of proofs via lemmas is important in reducing the size of proofs that must be transmitted between producer and consumer. For this example, we first prove the following lemma:

```
(define sum-list1-relation
  (forall ?L ?x
     (= (sum-list@ (Cons ?x ?L))
        (sum-list1 ?L ?x))))

(by-induction sum-list1-relation
 (Nil
  (!(conclude
      (forall ?x (= (sum-list@ (Cons ?x Nil))
                    (sum-list1 Nil ?x))))
    (pick-any x
      (dseq
       (!setup left (sum-list@ (Cons x Nil)))
       (!setup right (sum-list1 Nil x))
       (!reduce left (Plus x (sum-list@ Nil))
                sum-list@-nonempty)
       (!reduce left (Plus x zero)
                sum-list@-empty)
       (!reduce left x Plus-zero-axiom)
       (!reduce right x
                sum-list1-empty)
       (!combine left right)))))
 ((Cons y L)
  (!(conclude
      (forall ?x
         (= (sum-list@ (Cons ?x (Cons y L)))
            (sum-list1 (Cons y L) ?x))))
    (dlet ((induction-hypothesis
              (forall ?x
                 (= (sum-list@ (Cons ?x L))
                    (sum-list1 L ?x)))))
      (pick-any x
        (dseq
          (!setup left
             (sum-list@ (Cons x (Cons y L))))
          (!setup right
             (sum-list1 (Cons y L) x))
          (!reduce left
             (Plus x (sum-list@ (Cons y L)))
                sum-list@-nonempty)
          (!reduce left
             (Plus x (Plus y (sum-list@ L)))
                sum-list@-nonempty)
          (!reduce right
             (sum-list1 L (Plus x y))
                sum-list1-nonempty)
          (!expand right
             (sum-list@ (Cons (Plus x y) L))
                induction-hypothesis)
          (!reduce right
             (Plus (Plus x y) (sum-list@ L))
                sum-list@-nonempty)
          (!reduce right
             (Plus x (Plus y (sum-list@ L)))
                Plus-Associativity)
          (!combine left right)))))))
```

This proof is done by a combination of induction and rewriting steps. ATHENA provides `by-induction`, an inference special form that can be used if the type of the first quantified variable of the predicate to be proved is defined with `datatype`. In each of the two cases of the induction argument, the proposition to be proved is a quantified equation. The main technique for proof of equational propositions is rewriting, in which one performs a series of rewrites — applications of the substitution property of equality — to the term on the left hand side of the equation and another series of rewrites to the right hand side until obtaining two terms that are syntactically identical. We implemented a simple form of rewriting [14] by defining in ATHENA's method language the methods `setup` for isolating the left and right hand side terms of an equation, `reduce` and `expand` for performing a single rewriting step, and `combine` for deducing the original equation once the left and right terms have been rewritten to the same term.

The proof of the correctness theorem itself is also done with `by-induction`, but this time only as a convenient way to express proof by cases (it being unnecesary to use the induction hypothesis).

```
(by-induction sum-list-correctness
 (Nil
  (dseq
   (!setup left (sum-list Nil))
   (!setup right (sum-list@ Nil))
   (!reduce left zero sum-list-empty)
   (!reduce right zero sum-list@-empty)
   (!combine left right)))
 ((Cons x L)
  (dseq
   (!setup left (sum-list (Cons x L)))
   (!setup right (sum-list@ (Cons x L)))
   (!reduce left (sum-list1 L x)
       sum-list-nonempty)
   (!reduce right (sum-list1 L x)
```

```
        sum-list1-relation)
    (!combine left right))))
```

The producer transmits all the axioms defining `sum-list` and `sum-list1`, and the proofs of the CC, TC, and CTC to the consumer.

Steps 6, 7, and 8

To check for termination, the consumer executes TCGEN which produces the same TCs and termination rules generated at Step 3. After the termination rules are asserted, their proofs are entered into ATHENA. Since the proofs check, the consumer will execute CCGEN and produce the same CC generated at Step 4 to check for consistency. The consumer enters the proof of the CC and ATHENA accepts it. Since both checks are successful the consumer asserts the function-defining axioms safely, and checks the proof of correctness condition given in Step 5.

Step 9: Code Extraction

The consumer passes each list of axioms defining `sum-list1` and `sum-list` separately to CODEGEN, which generates the following two Oz functions:[2]

```
fun {SumList1 L X}
   case L
   of nil then X
   [] Y|L then {SumList1 L (X + Y)}
   end
end

fun {SumList L}
   case L
   of nil then 0
   [] X|L then {SumList1 L X}
   end
end
```

Since the first function is tail recursive, and Oz executes tail-recursion in constant stack space, `SumList` is memory efficient — roughly equivalent to being coded with a loop.

4. CHOICE OF DIFFERENT TARGET LANGUAGES

In addition to Oz as a target language for CODEGEN, we have also experimented with generating code in ATHENA's computational language. This choice has the advantage of reducing the size of the trusted computed base (TCB), but also has the drawback that the current implementation of ATHENA has no special treatment of tail-recursion. To remedy this, one could extend CODEGEN to generate loops with assignment statements (ATHENA has `while` loops and updatable cells). If the target language were C, there would be a similar problem since the standard for the language does not require compilers to remove tail-recursion, but again, CODEGEN could be extended to target one of C's loop constructs (or labels and goto's). Targeting to Scheme, however, could be done with minimal change to CODEGEN since Scheme implementations are required to do tail-recursion removal.[3]

Targeting any other language besides ATHENA's increases the size of the TCB, but Oz offers the potential of a fairly small addition. Although we are currently testing CCT using a complete implementation (the Mozart-Oz Windows version [10]) of the language, only a small part of the implementation is actually necessary for compiling and executing code produced by CODEGEN. Due to the way Oz is defined in terms of translations into a very small kernel language, even the full language implementation is available on at least one handheld computer (the IPAC), and an implementation of the small language subset targeted by CODEGEN should fit on many modern embedded computers. A similar argument can be made for a subset of a more widely known language like Scheme, but the kernel language approach and the existing formal specification of Oz [19] appear to provide a better basis for tackling compiler verification (thus moving it out of the TCB).

Eventually we may see other scenarios where CCT would be useful if there are enough resources to implement a larger target language and an automated proof search capability such that proof construction requires less human effort. (The necessarily large size of the TCB might be deemed acceptable if sufficient trust in the tools is built up through years of use.) For example, the CCT protocol could help different departments of a company to develop code and deliver it to customers. In this scenario, axiom experts and analysts in one department analyze the problem and define it axiomatically by working with customers. In the next step, they deliver these definitions with correctness and/or safety conditions to another department whose role is equivalent to code producers in CCT. Proof experts in the second department write required axioms and proofs that can be used in construction of efficient code, and deliver them back to the first department. In this scenario, there might be unintentional mistakes or hackers might attack the company's network and change the definitions at any point, but CCT will provide high assurance of correctness or safety, and the code will not be constructed and delivered to the customer unless the axiom experts in the first department are satisfied. Finally, CCT could also be used between the company and the customer, perhaps in conjunction with cryptography, to deliver verified code to the customer.

5. EXAMPLES AND ISSUES OF MEMORY-UPDATING FUNCTIONS

An important issue is how we deal with memory-updating functions such as `reverse-range`, a function that reverses, in place, the order of elements of a range of locations delimited by iterators (like STL's generic `reverse` function). Other examples of generic memory-updating functions we have axiomatized and proved correctness properties for include functions for copying a range (like STL's `copy`), swapping the contents of subranges of a range (like STL's `rotate`), and for partially partitioning a range in place into subranges of elements based on a unary predicate.

In order to generate efficient code from axioms that define memory-updating functions, our original design for the code extractor called for it to carry out the conversion of tail-recursion into while loops with assignments, so that passing of a parameter like `(assign` $M\ i\ x$`)` for memory M would become explicit as an assignment statement $M \leftarrow$ `(assign` $M\ i\ x$`)`, which could then be converted to a more efficient assignment statement $M[i] \leftarrow x$. In fact, however, the extractor can simply leave functions in tail-recursive form, even when memory-updating functions are involved. The key idea that makes things work out is that memory arguments can be passed by reference, thus not requiring expensive copying operations: when `(assign` $M\ i\ x$`)` is passed for M, the assignment $M \leftarrow$ `(assign` $M\ i\ x$`)` is actually done, but it is just assigning to M a reference returned by the `assign` function, not a copy of the memory. The only real work that is done

[2]The `case` statement is used for pattern matching in Oz.

[3]With either C or Scheme, it would be necessary to extend CODEGEN to implement a pattern-matching capability similar to what Oz and ATHENA directly provide.

to update a memory location, $M[i] \leftarrow x$, is carried out within the `assign` operation itself. This idea works out readily in Oz where one can represent memory by an array and take advantage of the fact that arrays are passed (to functions like `Array.put` and `Array.get`) by reference.[4] The same technique can be used to implement efficient memory updating, without the need for explicit array or vector assignment statements, in recursive functions in C or C++. (See, for example, the way that quicksort is implemented with tail-recursion by Bailey and Weston [8].)

Of course, this pass-by-reference semantics in the target programming language differs fundamentally from the pass-by-value semantics assumed in first order logic with equality (as reflected in its equality substitution proof rules), so we have to be careful to limit the contexts in which pass-by-reference is used to those where the difference cannot be detected by subsequent operations. Principally this means that a memory variable should not appear more than once in a term if it occurs as argument to an `assign` or `swap` call within the term.

For example, `(f (Assign M i x) (Access M))` is ruled out since in the copy-semantics of the logic, the change in memory effected by the first argument (assuming it is evaluated first) is not seen when the second argument is evaluated, but it is seen assuming reference semantics.

5.1 Note on Safety Properties

In dealing with memory-updating functions, the issue of memory safety arises (e.g., that the function does not update memory outside a certain range of locations). We have concentrated mostly on proving correctness rather than safety properties, but one cannot always avoid proving safety properties even when the original goal is to prove a correctness property. For example, in our correctness proof for `reverse-range`, a function that reverses in place the order of elements in a memory range, it was necessary to prove a lemma stating that the function does not update any memory locations outside the range to which it is applied — otherwise the induction hypothesis, which concerns the result of the function's recursive call, could not be applied.

6. DISCUSSION: ENSURING CORRECTNESS OR SAFETY

In CCT, both the producer and the consumer have to run CCGEN and TCGEN to get the same conditions. Therefore, there is no way of changing the verification conditions (CC or TC). Nevertheless, an intruder might change their proofs during transmission. Correctness is still not compromised since either the proofs will not go through, or are valid but not the proofs of TC and CC. Similarly, if an intruder attempts to change the proof of a correctness condition, the axioms will be rejected if the proof is not the proof of the required correctness conditions (which are decided on beforehand by the consumer).

In another scenario, an intruder may change the function-defining axioms. In this case, the original TC, CC, and correctness conditions all have to be proved by the intruder. If any of them has not been proved, the axioms will be rejected. If the proofs are provided, then there is no danger in using the axioms introduced by the intruder, at least in terms of correctness. In another scenario, an intruder might change the application-specific requirements during their transmission from the consumer to the producer. Again, the producer might be able to prove all verification conditions. But since the consumer has the original requirements, not the modified ones, the proofs of the modified requirements will generally not serve to prove the requirements held by the consumer.

In our last scenario, an intruder changes both the axioms and the proofs. Since the axioms are different, CCGEN and TCGEN will generate a TC and a CC that differ from the ones generated by the producer. If the proofs are corrupted and do not prove the new CC and TC, the new axioms will be rejected and will not be asserted by the consumer. If the changed proofs are valid, however, and prove both the new TC and CC, the consumer can assert the changed axioms safely (i.e., without fear of introducing an inconsistency in the assumption base). If CTC is proved too, correctness is assured and the consumer can use the axioms in code extraction even if they are not the ones provided by the producer. Otherwise, the changed axioms will be rejected.

We have not formally verified our implementation of CCT (i.e., of TCGEN, CCGEN and CODEGEN) but we note that the total amount of ATHENA code is less than 1000 lines, a size much smaller than has been reported for the VCGen program (23,000 lines) in type-specialized PCC [2], making it much more amenable to manual inspection. (ATHENA is a higher level language than C.) If there were a bug in either the VCGen or the typing rules then the trusted base would be vulnerable. In fact League [12] found an error in the typing rules of *Special J* [9], which is another certifying compiler that generates machine code from Java source code. Needless to say this bug affects the overall safety in type-specialized PCC.

Some theorem proving systems strive for complete automation—as in resolution provers or in Boyer-Moore provers that use highly developed heuristics to construct induction proofs—while others strive mainly for human readability of proofs and require the human user to provide almost all of the proof structure and only check for following rules of the logic—the MIZAR system [18] being perhaps the extreme case. In the spectrum between these extremes, there is always a tension between automation and human-readability goals. In our opinion, ATHENA deals with this tension better than any other system we have encountered. Its format of methods operating on an assumption base provides an elegant framework for expressing proofs in great detail where that is desired—and we believe it is desirable for proofs in generic theories, since then one can learn much about how such proofs work and often use them as guides to construction of other proofs. It is also desirable for proofs that must be efficiently checkable, without resorting to extensive search, as is likely to be the case for the embedded system type of code consumer. On the other hand, the hooks that ATHENA provides to highly-tuned resolution provers such as SPASS [22] and OTTER [13] support more productive human activity and could serve well even in the CCT setting in cases where the consumer has sufficient computational resources.

7. CONCLUSION

In this paper, we have presented a general framework called Code-Carrying Theory (CCT) to provide strong assurance to code consumers who receive, install, and execute new or updated code. CCT has a small computing base. It satisfies requirements without requiring run-time checks, if possible, in order to avoid performance degradation. It is flexible and general. Code consumers can decide on and define correctness requirements without reference to the specifics of machine instructions or architectures or any language-specific type systems. CODEGEN requires only a small amount of modification to generate code in a different target language. In addition, we proposed termination and consistency

[4] Similarly, in Scheme one can represent memory by a `vector` and pass vectors (to functions like `vector-assign!` and `vector-ref`) by reference. Though `vector-assign!` does not return the reference, one can encapsulate it in an `assign!` function that does.

checking mechanisms which are easy to apply in CCT.

We have tested CCT with thousands of lines of generic definitions and generic proofs, which also helps reduce proof sizes. In many cases, producers do not need to send similar definitions repeatedly since instantiation of existing generic theorems and proofs is sufficient for consumers. We simulate type-parameterization simply by parameterizing functions and methods by functions that carry operator mappings. This approach is illustrated in the way we have already formalized—and used in proofs—numerous properties of ordering theories (such as partial ordering, strict weak ordering, total ordering); algebraic theories (such as monoid, group, ring, integral domain, field), and sequential computation theories (such as iterator and range). Though space limitations have not permitted a full presentation in this paper, Vargun's thesis [20] gives more information about generic specification and proof writing, including examples in which proof sizes are dramatically reduced.

8. REFERENCES

[1] A. Ahmed, A. Appel, and R. Virga. A stratified semantics of general references embeddable in higher-order logic, 2002.

[2] A. W. Appel. Foundational proof-carrying code. In *Symposium on Logic in Computer Science*, pages 247–258, boston, 2001. IEEE Computer Society Press.

[3] A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *Programming Languages and Systems*, 23(5):657–683, 2001.

[4] K. Arkoudas. *Denotational Proof Languages*. PhD thesis, MIT, 2000.

[5] K. Arkoudas. Certified computation, 2001. `citeseer.nj.nec.com/arkoudas01certified.html`.

[6] K. Arkoudas. Specification, abduction, and proof. In *2nd International Symposium on Automated Technology for Verification and Analysis (ATVA 2004)*, National Taiwan University, October 2004.

[7] K. Arkoudas. An Athena tutorial, 2005. `http://www.cag.csail.mit.edu/~kostas/dpls/athena/athenaTutorial.pdf`.

[8] M. W. Bailey and N. C. Weston. Performance benefits of tail recursion removal in procedural languages. Technical Report TR-2001-2, Computer Science Department, Hamilton College, June 2001.

[9] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 95–107, Vancouver, Canada, June 2000. ACM Press.

[10] M. Consortium. The Mozart Programming System. `http://www.mozart-oz.org`.

[11] P. W. L. Fong. *Proof Linking: A Modular Verification Architecture for Mobile Code Systems*. PhD thesis, Simon Fraser University, 2004.

[12] C. League, Z. Shao, and V. Trifonov. Precision in practice: A type-preserving java compiler. Technical report, YALE/DCS/TR-1223, Dept. of Computer Science, Yale University, New Haven, CT, January 2002.

[13] W. W. McCune. Otter 3.0 reference manual and guide. Technical Report ANL 94/6, Argonne National Laboratory, January 1994.

[14] D. Musser and A. Vargun. Proving theorems with Athena. `http://www.cs.rpi.edu/~musser/gsd/athena.html`.

[15] G. C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, Jan 1997.

[16] G. C. Necula. *Compiling with Proofs*. PhD thesis, CMU, 1998.

[17] R.S.Boyer and J. S. Moore. *A Computational Logic*. Academic Press, 1979.

[18] A. Trybulec. Some features of the Mizar language. In *ESPRIT Workshop*, Torino, 1993.

[19] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Mar. 2004.

[20] A. Vargun. *Code-Carrying Theory*. PhD thesis, RPI, 2006. `http://www.cs.rpi.edu/~vargua/thesis/cct-thesis.pdf`.

[21] A. Vargun. Termination checking without using an ordering relation. In *11th IASTED International Conference on Software Engineering and Applications*, Cambridge, Massachusetts, USA, 2007. ACTA Press.

[22] C. Weidenbach. SPASS: An automated theorem prover for first-order logic with equality. `http://spass.mpi-sb.mpg.de`.