Code-Carrying Proofs

Aytekin Vargun

October 2, 2004

Abstract

Code Carrying Proofs are an alternative to the Proof Carrying Code (PCC) approach to secure delivery of code. With PCC, code is accompanied by a proof of its correctness or of other required properties. The code consumer does not accept delivery unless it first succeeds in generating theorems from the code called verification conditions and checking that the supplied proof proves these theorems. With Code Carrying Proofs (CCP), instead of transmitting both code and proof explicitly, only proofs are transmitted to the consumer. If proof checking succeeds, the code is then obtained by applying a Code Extractor, either to the proofs themselves or to the theorems they prove. This document explains the design and implementation of a preliminary version of a Code Extractor that extracts code from theorems. It is implemented in Athena, which is both a functional programming language and a deduction language. Several small examples of Code Carrying Proofs and corresponding code extraction are given.

Contents

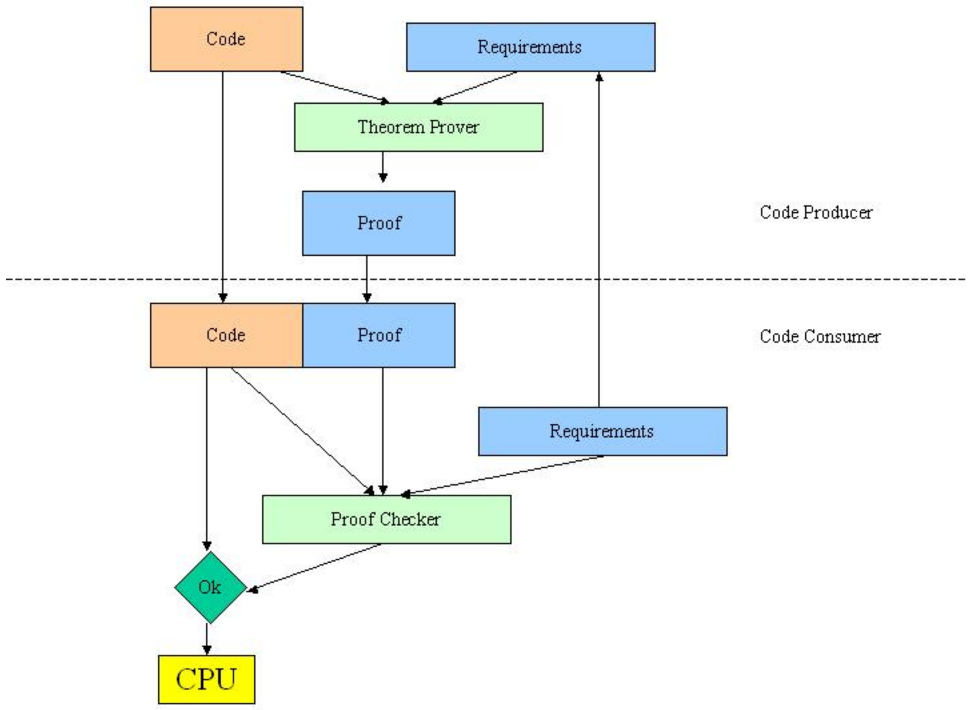


Figure 1: Proof Carrying Code, Successfully Checked and Delivered

1 Introduction

Today's ever increasing trends toward pervasive computing include the phenomenon of *mobile code*, which must be installed and run on a computer host even though it is delivered through an insecure network or from an untrusted source. During the delivery code might be corrupted or a malicious user may change the code. Potential problems can be categorized as security problems (i.e., unauthorized access to data or system resources), safety problems (i.e., illegal operations or illegal access to memory) or functional correctness (i.e., whether the delivered code does correctly what it is formally required to do). *Proof-Carrying Code* (PCC) [27] and *Code-Carrying Proofs* (CCP) are two alternatives among other solutions to these problems. These two alternatives can generally provide stronger assurance of security properties or other required properties.

A quick picture of main idea of the PCC approach can be seen in Figure 1 and Figure 2. The basic idea of PCC is that *a code consumer does not* *accept delivery of new code unless it is accompanied by a formal proof of its correctness that can be checked by the code consumer.* One way of doing this is to attach to the code an easily-checkable proof at the code producer's site. This proof must be checked by the code consumer and should prove that the code does not violate the predefined safety policy. Here "safety" means *memory safety* (no memory accesses outside of prescribed ranges) or *type safety* (no attempts to apply an operation to inappropriate data) or both.

The transmitted proof can be regarded as a form of certification that it is safe to execute the code. In Necula and Lee's PCC approach [27] code is annotated with correctness assertions and a program called a verification condition generator is applied to produce a set of verification conditions (VCs), which are propositions whose validity implies that the code satisfies the stated assertions. Both the annotated code and the proof of its VCs must be sent to the consumer, who regenerates the VCs, checks that the proof proves them, and accepts delivery of the code only if this proof-checking step succeeds. In Appel's PCC approach [3] the definitions of safe machine instructions are encoded with a suitable logic. The safety conditions are included in these definitions implicitly. Therefore, a VC generator is not necessary, resulting in a smaller "trusted code base," i.e., a smaller amount of code subjected only to manual inspection, not formal checking. A program is simply defined as a sequence of machine instructions which are represented by integers. Then a safety theorem can be constructed based on the idea that the program counter points to an illegal instruction if any unsafe instruction is encountered during the execution of the code. As in Necula and Lee's approach the proof of the safety theorem is transmitted with the code to the consumer.

CCP, Code-Carrying Proofs, is a new alternative to PCC with similar goals and technology, but it is based on the idea of *proof-based program synthesis* [23, 9, 12, 2, 7] rather than program verification. That is, instead of starting with program code in a conventional programming language, annotating it with assertions, generating VCs and proving them, one instead starts with axioms that define functions (generally with equations and induction) and proves that the defined functions obey certain requirements. The form of the function-defining axioms is such that it is easy to extract executable code from them. Thus all that has to be transmitted from producer to consumer is the axioms, the correctness theorems, and their proofs; there is no need to transmit code explicitly or to apply a VC generator. The consumer checks that the proofs actually *do* prove the correctness theorems, and only if that step succeeds does the consumer apply a "code extractor" to

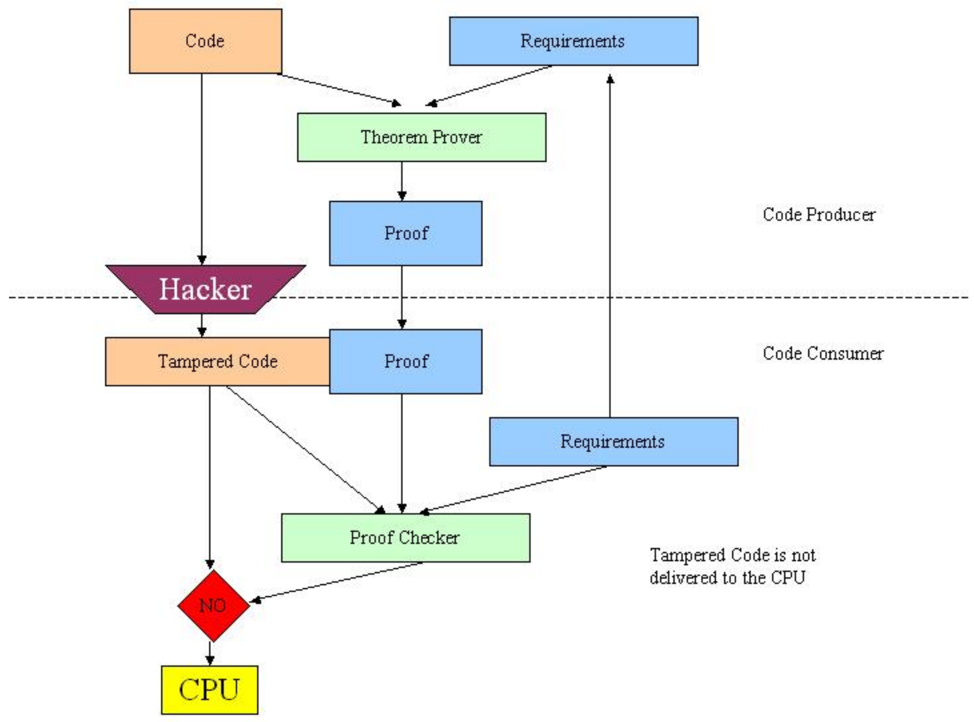


Figure 2: Proof Carrying Code, Tampered-With But Checked and Prevented from Acceptance

the function-defining axioms to obtain the executable code. A variant on this approach would be to extract code directly from proofs based on their structure in terms of how inference rules of different types are used [23]. In this paper, we do not explore this second possibility of code extraction further. Figure 3 illustrates the basic CCP approach.

We have implemented a preliminary version of a Code Extractor that is capable of producing code that implements "nondestructive" functions, i.e., ones that examine data structures but don't make any changes, such as searching or element counting functions. We plan to implement an improved version that will also be able to extract "destructive" functions like in-place reverse or sort functions. Although we have implemented more generic and complicated examples we are presenting simple code samples which could be extracted from function-defining axioms. These examples are similar in some respects to functions in the C++ Standard Template Library (STL), such as accumulate, max_element, copy, find, and many others. One can use such

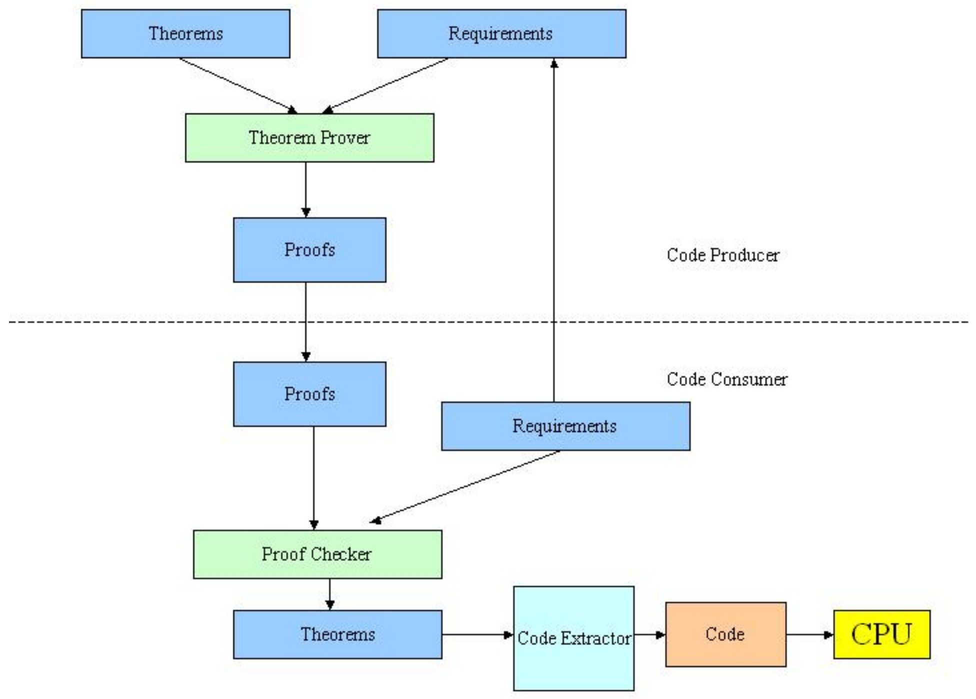


Figure 3: Code Carrying Proofs, Successfully Checked and Delivered

functions to compose more complicated functions. In particular one of the functions we present is sum, which is very similar to the STL's accumulate function. We show how to extract code from the axioms defining sum and tail-recursive sum in particular. In analogy to STL it would be possible for the code extractor to extract complex functions composed of such simple functions which would be defined by the small number of simple axioms. The proofs of complex functions may use other theorems or lemmas about other functions directly.

The implementation tool we use is Athena [5], which provides a language both for ordinary computation and for logical deduction. As a functional programming language, it provides higher-order functions and is lexically scoped in the tradition of Scheme and ML. As a deductive programming language, Athena makes it possible to formulate and write proofs as function-like constructs 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. This high-level abstraction of proofs allows the user to call the functions and apply methods defining them in many different ways. Therefore the same proof doesn't have to be redeveloped for different parameters. A generic library of proofs can be constructed with this ability.

By correctness we mean that the code does what its formal requirements state it should do. Here requirements can take the form of input-output correctness requirements (e.g., an algorithm sorts its input); safety requirements (e.g., no out-of-bounds array-indexing); or security requirements (e.g., no unauthorized access to classified data). In the case of safety, the requirements are stated as a set of "safety rules" and are also called a "safety policy." One can define different safety policies for accessing memory or other system resources. PCC has been developed and applied solely as a method for achieving safety, but in principle it could be used with other forms of requirements (though, under the current state of the art of program verification and theorem proving technology, with a lesser degree of automation than is achievable with safety properties). CCP could also be used with any of the three forms of requirements, but in this paper we focus on applying CCP to input-output correctness (also called "functional correctness"). In conjunction with either PCC or CCP, one could employ additional certification techniques, such as encrypted signatures or check-sums; [13] discusses such combinations as well as the use of "trusted hardware." Figure 4 shows how CCP guarantees requirements are satisfied by checking the transmitted proofs; if the proofs have been tampered with by a malicious hacker (as shown in the diagram) or errors were inadvertently introduced at any point by the producer or in transmission, proof-checking fails and no new code is installed.

The next section reviews the existing PCC approaches. In Section 3, we illustrate and discuss the CCP approach in detail. Section 4 explains Athena briefly and shows the term rewriting methods we have implemented. In Section 5, we explain some sample code and proofs showing how one can use the code extractor to extract code. We illustrate the CCP approach with two examples: "sum defined on lists" and "tail-recursive sum defined with iterator ranges." The next section contains information about our future goals and potential problems to be solved in this project. Section 7 shows the timetable for the remaining work.

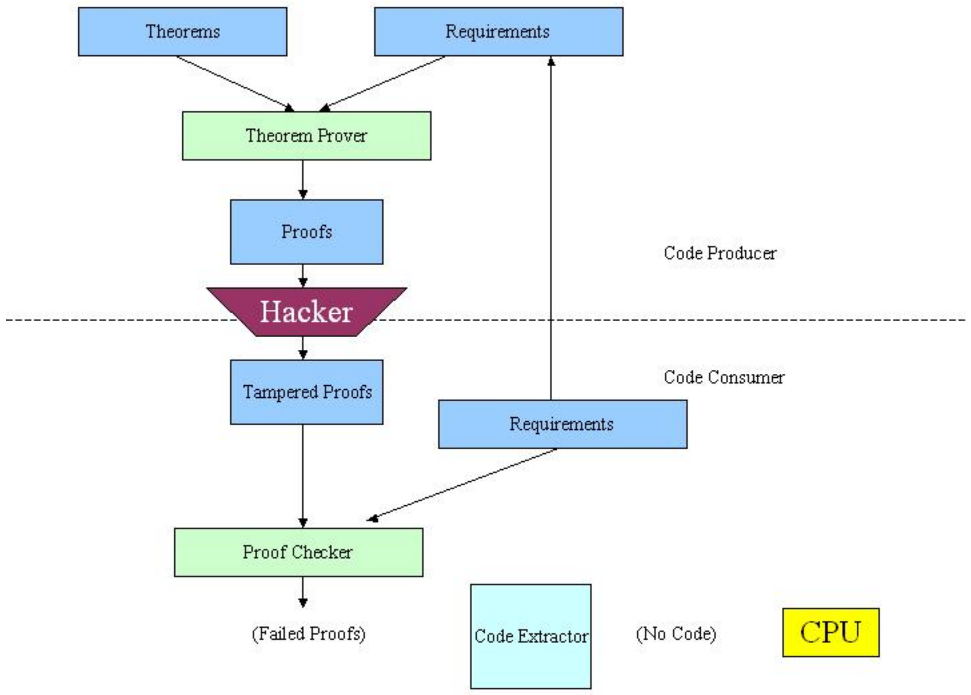


Figure 4: Code Carrying Proof, Tampered-With But Checked and Prevented from Acceptance

2 Proof-Carrying Code

2.1 Type-Specialized Proof-Carrying Code

Figure 5 shows the original proof-carrying code developed by Necula and Lee. Appel refers to this type of PCC as *type-specialized PCC* [3]. The architecture is composed of a compiler, verification condition generator (VCGen), theorem prover and a proof checker. This technique is fundamentally based on the idea of defining the safety policy in terms of *types* and achieving safety by *type checking*.

In general the predefined safety policy consists of memory safety rules, typing rules, predicates defining the semantics of the machine instructions, and rules of first-order logic. Some examples of the rules which could be defined to achieve safety would be: specific memory regions are readable, specific memory regions are writable, if a variable's type is natural numbers

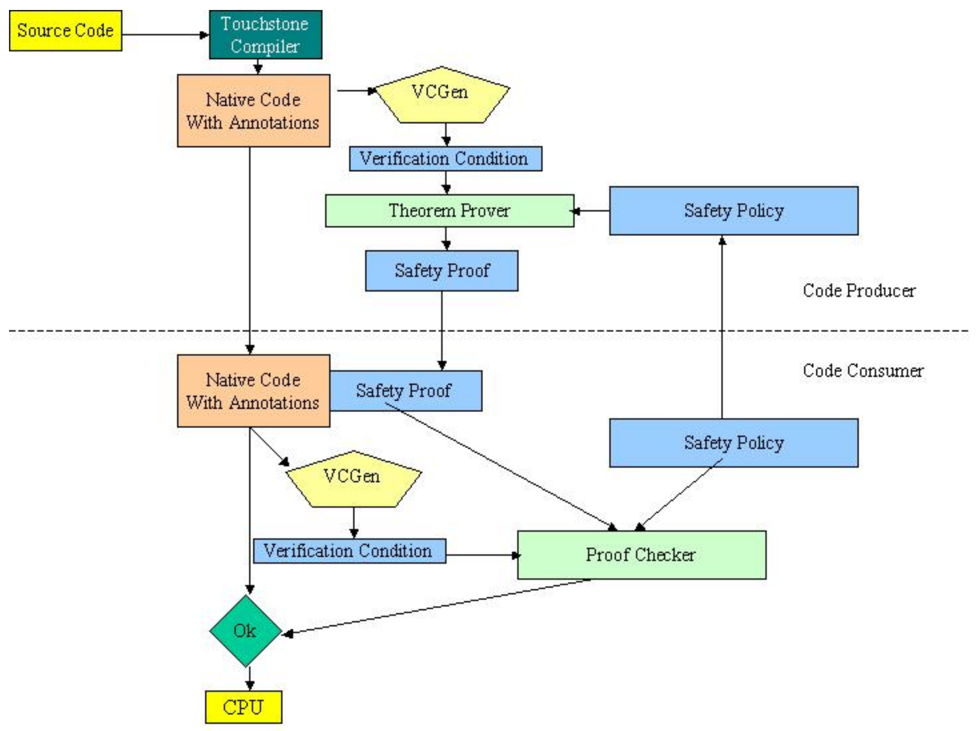


Figure 5: Necula and Lees Type-Specialized Proof Carrying Code

then it is readable and so on.

All the rules in the safety policy are expressed by using the higher-order LF syntax [16]. Representing the rules of safety policy in a formal logic is crucial to generate and check the safety proof during the later steps. Since type checking is used to decide the safety policy all the typing rules are required to be introduced as predicates by using the LF logic. An example of such a predicate is "expression e has type t."

An important note is that both the code producer and the consumer will obtain the safety policy before the code transfer. In the remaining part of this section we explain the details of type-specialized PCC.

2.1.1 Creating the Native Code with Annotations

In the first step, the source code written in a type-safe subset of C is translated by a compiler called Touchstone [28] into annotated native code and type specifications. Touchstone is an optimizing compiler whose target language is assembly. The native code will be transferred to the consumer along with a proof of safety. The code annotations provide more information about the code to be sent by the producer. This extra information can be used to make the VCGen's job easier, to enable optimizations, or to resolve ambiguities.

2.1.2 Generating the Verification Condition

The next step is to construct a verification condition as the safety theorem. The code producer applies the VCGen to the code and the annotations to construct a predicate in the LF-style first-order logic defining the meaning of the source code. While doing this it uses the loop invariants, function preconditions and postconditions as well as the typing rules. Each of the created verification conditions is a *Floyd-style verification condition* [15] which is a predicate expressed in first-order logic.

The VCGen is a simple and fast program written in C. The code consumer will use the VCGen and create the same verification conditions later for the purpose of checking the proof of safety.

2.1.3 Finding a Proof of the Verification Condition

After the VCGen generates the verification condition, the code producer uses a theorem prover to get a proof of this condition. As it is seen from Figure 5, the theorem prover uses the axioms defined as part of the safety policy. The proof is the certification that the code is both type safe and memory safe which means it can be executed safely.

The proof must be encoded formally so that it can be checked by the consumer automatically. In type-specialized PCC, the proofs are done by using Elf[29] theorem prover which produces LF style representation of proofs. It is important that the proofs are represented in LF because when the consumer uses the proof checker to check the proof it assumes this pre-negotiated framework.

In general proving the verification condition is a time-consuming and a complex process to be done by the producer. But it is the beauty of this technique that this difficult job is not being done by the code consumer during any steps of PCC.

Finally this proof is submitted with the native code to the consumer.

2.1.4 Verifying the Proof

The code consumer is required to check the proof *before* loading the executable submitted by the producer. He runs the VCGen to find the verification condition as the producer did previously. The next step is to see if the submitted proof is really the proof of the verification condition. After the proof check succeeds the consumer can load the executable safely.

One important reason for using LF-style encoding of proofs is to use the type checking in LF to verify the proofs. To be able to use this facility one should define all the predicates as LF types and write the proof as an LF expression. More formally: let `P` is a predicate and `pf P` is a proof of this predicate. Assume also that we have the following declarations:

```
pred : Type

pf   : pred-> Type
```

Having `pf P` means that we actually have the proof of `P` and this proof should have a type. Based on this information we use the verification condition as the predicate `P` and want to see if the LF type checks the proof of the verification condition `pf P` provided by the code producer. If it succeeds then the proof validation is completed. After this step the code can be executed safely.

2.2 Foundational Proof-Carrying Code

Even though a VCGen is simple and fast, Appel points out that *its size is too large* [3] and it needs to be verified itself. If there were a bug in either the VCGen or the typing rules then the trusted base would be vulnerable. In fact League [21] found an error in the typing rules of *Special J* [11], 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. Since Appel avoids using a VCGen *the trusted base is smaller.*

Foundational PCC is based on the idea of defining the semantics of the machine instructions and the safety rules using only a foundational mathematical logic (i.e., a logic that itself has no programming-language–specific axioms or safety rules). This approach does not rely on a particular type system since it does not introduce any type constructors as primitives or prove lemmas about these constructors as in type-specialized PCC. As in type-specialized PCC the code producer must send both the executable code and its safety proof.

2.2.1 Defining and Proving Safety

A specific von Neumann machine such as a Sparc processor is required to be modeled for a program P implemented in a machine-language, with the model consisting mainly of a register bank and a memory. Each of these can be thought of a function from integers (addresses) to integers (contents). The foundational logic used to define instructions and instruction executions is a higher-order logic. Each execution step is specified by a *step relation* $(r, m) \mapsto (r', m')$ where r and r' are registers and m and m' are memory units. Then by executing one instruction the machine state is changed from (r, m) to (r', m'). As an example (from [3]), one could define the add(r_1, r_2, r_3) instruction $(r_1 \leftarrow r_2 + r_3)$ as

$$(r, m) \mapsto (r', m') \equiv r'(1) = r(2) + r(3) \wedge (\forall x \neq 1. r'(x) = r(x)) \wedge m' = m$$

An important property is that some steps do not have any successor steps. In these cases the program counter points to an illegal instruction which should never exist in a safe program. In addition, if there is an undefined instruction it will be considered to be an illegal instruction that violates the safety policy. Having this in mind the safety policy is defined as: A given state is safe if, for any state reachable in the Kleene closure of the step relation, there is a successor state. Here is the formal definition:

$$\textit{safe-state}(r, m) \equiv \forall r', m'.(r, m) \mapsto^* (r', m') \Rightarrow \exists r'', m''.(r', m') \mapsto (r'', m'')$$

where $\mapsto^*$ denotes zero or more steps.

Note that this definition is used for formulating the verification condition for each program. The code producer should decide on an invariant which should hold during the execution of the code satisfying $\textit{safe-state}(r, m)$ for each step. If any step reaches an illegal instruction then the safety policy is violated and the program will not be accepted by the consumer.

Defining different safety policies is possible by using a different form of step relation. Let us define a policy saying “only the contents of memory addresses between 0 and 1000 are readable.” We first define the predicate *readable* as follows:

$$readable(x) = 0 \leq x < 1000$$

We next define a general `load` instruction $r_i \leftarrow m[r_j + c]$

$$(load(i, j, c) \equiv \lambda r, m, r', m'.(r'(i) = m(r(j){+}c)){\wedge}(\forall x \neq i.r'(x) = r(x)){\wedge}m' = m$$

Now we just need to change the semantics of the `load` with an extra conjunct:

$$\begin{aligned} \texttt{load}(i, j, c) \equiv \lambda r, m, r', m'.& \\ & r'(i) = m(r(j) + c) \ \wedge \ (\forall x \neq i.r'(x) = r(x)) \ \wedge \ m' = m \\ & \wedge \ \text{readable}(r(j) + c) \end{aligned}$$

When executing the program if the program counter points to a `load` instruction which wants to read the content of memory address 1001, the state relation will not relate this state to any other state.

In order to prove that a program P is safe what we need to do is to prove safe-state(r, m) for steps $= S_0, S_1, \ldots, S_n$ for n steps. This can be done by using proof by induction on n. Note that if we succeed in finding a proof this means that after executing the program P we get a safe state.

In foundational PCC the proofs are currently done by *hand* and represented using the LF syntax. Therefore if one has the proof of a program P the proof validation would be achieved by just type-checking the given proof in LF. The theorem prover used in this process is Twelf [30].

3 Code-Carrying Proofs

Figures 3 and 4 show scenarios corresponding to Figures 1 and 2 using Code Carrying Proofs instead of Proof Carrying Code. With Code Carrying Proofs, instead of transmitting both code and proof explicitly, only a proof is transmitted. As has been well-known for at least two decades, it is possible to structure theorems so that executable code can be *extracted* from them. Such extraction of code from theorems is called the *proof-based program synthesis* approach. Although it has been studied for many years and is still a focus of a few current research projects such as NuPrl [9, 12] and

Coq [2, 7], it has not achieved much of a foothold in the realm of software engineering, probably because when compared to ordinary methods of software development it 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 for 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.

3.1 Other PCC/CCP differences

Aside from relative simplicity, there are other differences in the CCP approach from existing PCC work that are potential advantages. These include lower requirements for both memory and processor speed, both of which are especially important in the application to code that is being moved onto embedded computers (which typically have much more limited hardware than mainframes, workstations, or PCs).

In CCP one runs program extraction, which appears to be generally less CPU intensive than its counterpart in type-specialized PCC, verification condition generation. Appel's foundational approach [1, 4] avoids the use of a verification condition generator, but evidently only by conducting proofs at a much more detailed level. The computing time cost of working in such detail seems likely to be even greater than when using a VC generator.

3.2 Other significant differences

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 CCP the "source language" is actually a logic as defined and implemented in the theorem proving system, so when we talk about the programming language involved we mean the language in which the extracted code is expressed. Previous research such as Necula and Lee's and Appel's has been carried out with very low level languages (machine code or Java byte code). The theorem proving system we are using, Athena [5, 6], provides both a Scheme-like programming language 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 (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.

Moreover, we are experimenting with the use of both these programming and deductive language facilities at a substantially higher level of abstraction than in most programming activity. We have already found elegant ways to use higher-order functions and methods in Athena to express generic requirements specifications much like the *theory* specifications of research languages such as Isabelle [34], Imps [14], or Maude [10], or the *concept descriptions* of Tecton [17, 24, 26]. There are two major benefits that stem from expressing proofs at this high abstraction level. First, once a proof of a theorem is written at this level, the functions and methods defining it can be called in many different ways, which means that the proof doesn't have to be redeveloped when its conclusion is needed in a different setting. Second, from such high-level theory or concept specifications one can extract *generic* software components—ones that have a single source code expression but which can be instantiated in many different ways to produce different useful specific versions by plugging in other suitable components. Thus the substantial effort required in constructing such proofs can be amortized over the many repeated uses of both proofs and the generic software components that can be extracted from them.

3.2.1 Generic Proofs and Extracting Generic Code

One of our goals is to write *generic proofs* and *extract generic code* from these proofs. Generic libraries such as the C++ Standard Template Library and the Boost Graph Library have proven their value. Having generic *proof libraries* would make it possible for developers to write complicated proofs easily and rapidly by just selecting the correct combinations of existing proofs in the libraries. It would take less time to validate and maintain the proofs which would perhaps have the best and shortest possible implementations.

Although in future work we plan to concentrate on generic proofs, for simplicity we restrict the examples in this paper to non-generic proofs. Even with generic proofs, however, there will be little difference in terms of what the code extractor does, since we plan to apply it only to non-generic proofs

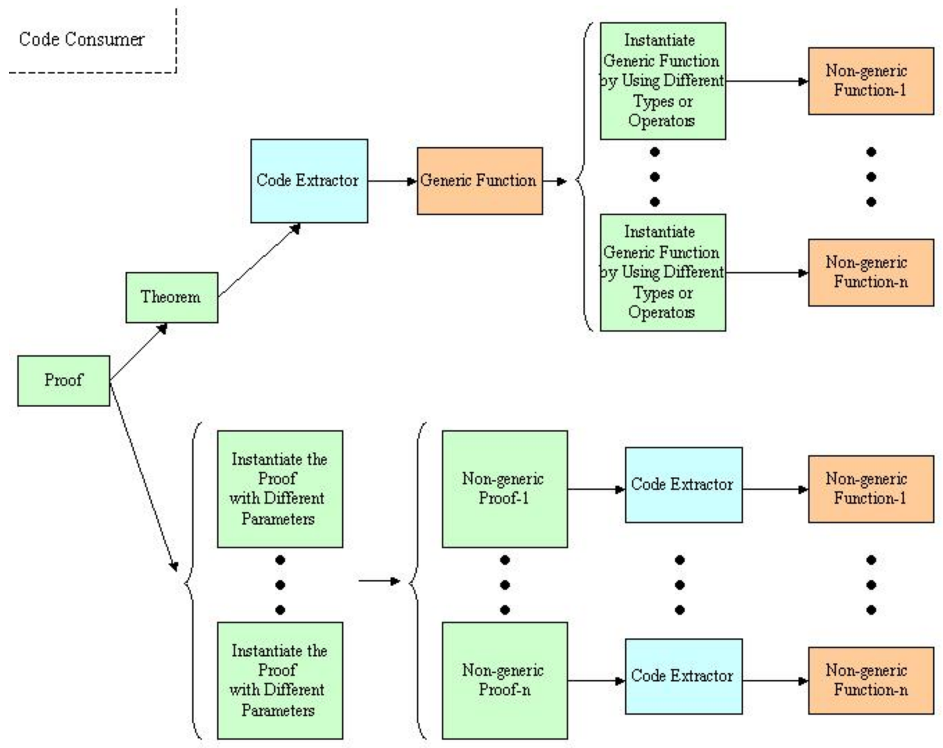

Code Consumer
Proof
Theorem
Code Extractor
Generic Function
Instantiate Generic Function by Using Different Types or Operators
Non-generic Function-1
Instantiate Generic Function by Using Different Types or Operators
Non-generic Function-n
Instantiate the Proof with Different Parameters
Instantiate the Proof with Different Parameters
Non-generic Proof-1
Non-generic Proof-n
Code Extractor
Code Extractor
Non-generic Function-1
Non-generic Function-n

Figure 6: Generic Code Extraction

that result from instantiation of the generic proofs. This is not the only possibility; Figure 6 shows two different approaches to extracting generic code in CCP. In the first case the code extractor works on the theorems and extracts a generic function first. This function has the nature that one can instantiate it using different types and operators producing many different non-generic functions. In the second case the proofs are first instantiated by using different parameters like types and operators. After this step there will be many different non-generic proofs. The code extractor works on each of these different proofs and extracts non-generic functions.

There are two reasons for why we prefer the bottom approach to the upper one:

- The extractor is not required to be capable of synthesizing templated functions, and the subsequent language processing doesn't have to deal with them (allowing the generated code to be C, for example).

- The control over instantiation is done explicitly in the bottom case in the boxes labeled "Instantiate the Proof with Different Parameters" rather than being left up to the language processor as in the top case. Leaving it up the language processor (C++ or Generic Java or Ada, presumably, since it would have to be some language that supports generics) would introduce some uncertainty about what exactly which instantiation would be done, since the language rules governing the way that instantiations are selected are complex, and that would result in uncertainty about the validity of the whole approach.

4 Athena

The Athena language and proof system developed at MIT by K. Arkoudas provides a way to present and work with mathematical proofs. It provides a language both for ordinary computation and for logical deduction. As a functional programming language, it provides higher-order functions and is lexically scoped in the tradition of Scheme and ML. 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. It is an example of a *denotational proof language* (DPL) [5] and has been proved sound and complete. A proof can be expressed at a high-level of abstraction, allowing the user to call the functions and methods defining it in many different ways. Therefore the same proof doesn't have to be redeveloped for different parameters. A generic library of proofs can be constructed with this ability.

Athena has an abstraction called *assumption base* which is an associative memory maintained by the language as it applies deductive methods; it is basically a record of what propositions have been asserted or proved since the beginning of the current session (or since the most recent `clear-assumption-base` directive which removes all of the propositions in the assumption base). All of Athena's primitive methods refer to the assumption base and most extend it with new propositions (none can remove a proposition from it).

Athena has primitive functions for unification, matching, substitution and mapping. ML-like pattern matching is provided for defining both functions and methods. Athena also offers support for both forward and backward inference steps, proofs by induction, proof by contradiction, and equational reasoning.

A *cell* is a value that contains another value. It acts as memory location into which other values can be placed. When a cell is created it is given an initial value. Cells (and only cells) can be updated (by `set!`). Note that this is the same as in ML but different from Scheme, which allows for unrestricted updating of the value associated with any symbol. A cell can contain another cell, since cells are values.

4.1 Primitive Deduction Methods

Athena provides fifteen primitive deduction methods, which will be described in this section.

claim P

succeeds if proposition P is in the assumption base, and fails (results in an error) otherwise. If succeeds, it returns P as a theorem.

mp (if P Q) P

adds the proposition Q to the assumption base if both (if P Q) and P are in the assumption base.

both P Q

adds the proposition (and P Q) to the assumption base when P and Q are both in the assumption base. Note that the order in which the arguments are given to the both method is important because (and P Q) and (and Q P) are considered to be distinct propositions — commutativity of `and` is not automatically taken into account.

left-and (and P Q)

adds the proposition P to the assumption base when (and P Q) is in the assumption base.

right-and (and P Q)

adds the proposition Q to the assumption base when (and P Q) is in the assumption base.

One can use a combination of `left-and`, `right-and`, and `both` to reorder the arguments of `and` (which, as previously noted, may be occasionally necessary because commutativity of `and` is not automatically taken into account)

equiv (if P Q) (if Q P)

adds the proposition (iff P Q) to the assumption base when both (if P Q) and (if Q P) are in the assumption base.

left-iff (iff P Q)

adds the proposition (if P Q) to the assumption base when (iff P Q) is in the assumption base.

right-iff (iff P Q)

adds the proposition (if Q P) to the assumption base when (iff P Q) is in the assumption base.

Just as with `left-and`, `right-and`, and `both`, one can use a combination of `left-iff`, `right-iff`, and `equiv` to reverse the order the arguments of an iff expression.

either P Q

adds the proposition (or P Q) to the assumption base when either P or Q is in the assumption base.

cd (or P Q) (if P R) (if Q R)

This method, called "constructive dilemma," adds the proposition R to the assumption base when all three of the argument propositions are in the assumption base;

cases (if P R) (if (not P) R)

adds the proposition R to the assumption base when both of the argument propositions are in the assumption base.

absurd P (not P)

adds the proposition `false` to the assumption base when both P and (not P) are in the assumption base. It is meant to be used within a by-contradiction deduction, as the last step of a *proof by contradiction.*

dn (not (not P))

This method, called "double negation," adds the proposition P to the assumption base when (not (not P)) is in the assumption base. This can be useful, for example, when a proof by contradiction is done using by-contradiction with a proposition of form (not P), producing, if it succeeds, one of the form (not (not P)).

The following methods, unlike all the previously described primitive methods, perform deductions that depend on their argument proposition having a (top-level) `forall` or `exists` quantifier. For a proposition P, variable $?x$, and term t, let $P\{t/?x\}$ denote the proposition that results from replacing all free occurrences of $?x$ in P with t. The type of t must be compatible with the type of $?x$.

uspec (forall $?x$ P) t

universal specialization adds the proposition $Pt/?x$ to the assumption base when (forall $?x$ P) is in the assumption base and t is a term whose type is compatible with the type of $?x$.

egen (exists $?x$ P) t

existential generalization adds the proposition (exists $?x$ P) to the assumption base when the proposition $Pt/?x$ is in the assumption base.

There is also a useful generalization of the `uspec` method:

uspec* P $[t_1 \ldots t_n]$

replaces the first n quantified variables in P with the terms $t_1, \ldots, t_n$ and adds the resulting proposition to the assumption base, provided P is in the assumption base.

4.2 Some Example Proofs

In this section we illustrate how to write simple proofs in Athena with formalization of some properties of an order relation.

We first declare the arity of the `<` operator in terms of an arbitrary domain we call `D`:

```
(domain D)

(declare < (-> (D D) Boolean))
```

which means that `<` is a function that takes two arguments from the domain `D` and produces an element of the domain `Boolean`, which is a predefined domain in Athena.

We now assert as axioms the irreflexive and transitive laws for `<`:

```
(define Order-Irreflexive
  (forall ?x
    (not (< ?x ?x))))

(define Order-Transitive
  (forall ?x ?y ?z
          (if (and (< ?x ?y) (< ?y ?z))
              (< ?x ?z))))

(assert Order-Irreflexive Order-Transitive)
```

The `assert` command is used for inserting propositions into the Athena's assumption base (without proof; it is up to the user to avoid introducing contradictory assertions).

These axioms characterize `<` as a *strict partial order* relation. From these axioms we can prove another law, asymmetry:

```
(define Order-Asymmetry
  (forall ?x ?y
          (if (< ?x ?y)
              (not (< ?y ?x)))))
```

Before getting into the details, we first describe some of the Athena special deductive forms to be used in the proof.

assume P D

first adds the proposition P to the assumption base temporarily. Then deduction D is evaluated. If this evaluation fails Athena the whole deduction fails. Otherwise the proposition `(if` P R`)` is added to the assumption base, where R is the proposition proved by D. In either case, the proposition P is removed from the assumption base after this evaluation.

P BY D

adds the proposition P to the assumption base if it can be deduced from the steps given in D. It is an error if deduction D fails or proves a proposition different from P. (Deductions of this form could always be shortened just to D, but including "P BY" serves to clarify for the human reader what is the goal of the deduction.)

pick-any I D

provides variables via the identifier I. D is then evaluated by using these variables.

In addition to these special deductive forms we define the following method:

by-contradiction (assume P D)

temporarily adds to the assumption base proposition P by using an `assume` and performs deduction D to show that `false` follows from P. If this evaluation fails Athena the whole deduction fails. Otherwise the proposition `(not` P`)` is added to the assumption base. In either case, the proposition P is removed from the assumption base after this evaluation.

Now let's return to the proof of Order-Asymmetry. The proof is *by contradiction*. We use the `assume` special deductive form to put `(< x y)` is in the assumption base temporarily, because the goal we are trying to prove is a conditional proposition. To be able to prove `(not (< y x))` we use `by-contradiction` and add `(< y x)` to the assumption base temporarily with an `assume`. Then we proceed to derive a contradiction.

```
(Order-Asymmetry
 BY
 (pick-any x y
   (assume (< x y)
     ((not (< y x))
      BY (!by-contradiction
          (assume (< y x)
            (!absurd
             ((< x x)
              BY (!mp (!uspec* Order-Transitive [x y x])
                      (!both (< x y)
                             (< y x))))
             ((not (< x x))
              BY (!uspec Order-Irreflexive x)))))))))
```

The contradiction we get, expressed in the arguments to `absurd`, is `(< x x)` and `(not (< x x))`.

Thus, we've shown that the asymmetry law is a theorem when one has a partial order relation; *one does not have to assert it as a separate axiom.*

Now suppose we define a binary relation `E` as follows:

```
(declare E (-> (D D) Boolean))

(define E-Definition
  (forall ?x ?y
          (iff (E ?x ?y)
               (and (not (< ?x ?y))
                    (not (< ?y ?x))))))
```

The name `E` for this relation is motivated by fact that we can show that if `E` is assumed to be transitive then in combination with the partial order axioms for `<` we can prove that `E` is in fact an equivalence relation; i.e., that it also obeys the other two axioms of an equivalence relation besides the transitive law, namely the reflexive and symmetric laws.

```
(define E-Transitive
  (forall ?x ?y ?z
          (if (and (E ?x ?y) (E ?y ?z))
              (E ?x ?z))))
```

```
(assert E-Definition E-Transitive)

(define E-Reflexive
  (forall ?x
    (E ?x ?x)))

(E-Reflexive
 BY (pick-any x
      (dbegin
       ((not (< x x))
        BY (!uspec Order-Irreflexive x))
       ((E x x)
        BY (!mp (!right-iff (!uspec* E-Definition [x x]))
                (!both (not (< x x))
                       (not (< x x))))))))

(define E-Symmetric
  (forall ?x ?y
    (if (E ?x ?y)
        (E ?y ?x))))

(E-Symmetric
 BY (pick-any x y
      (assume (E x y)
        (dlet ((both-not
                ((and (not (< x y))
                      (not (< y x)))
                 BY (!mp (!left-iff (!uspec* E-Definition [x y]))
                         (E x y)))))
          ((E y x)
           BY (!mp (!right-iff (!uspec* E-Definition [y x]))
                   (!both (!right-and both-not)
                          (!left-and both-not))))))))

(define E-Equivalent
  (and E-Reflexive
       (and E-Symmetric
            E-Transitive)))
```

```
(E-Equivalent
  BY (!both E-Reflexive
            (!both E-Symmetric
                   E-Transitive)))
```

4.3 Inductive Definitions of New Types

In Athena one way of introducing a new type is to use a `structure` declaration.

```
(structure Nat zero (succ Nat))
```

This declaration introduces two constructors that produce values of type `Nat`. The first constructor, `zero`, is a *nullary, or constant, constructor.* The second constructor, `succ`, is a *unary constructor* which takes a natural number as an input and returns another natural number. In addition to saying that these constructors produce values of type `Nat`, it also implicitly states that the *only values* of type `Nat` are those that can be constructed by some finite number of applications of these constructors; there are no other *Nat* values. That is, the `Nat` type is exactly the (infinite) set

$$\{\texttt{zero}, \texttt{succ}(\texttt{zero}), \texttt{succ}(\texttt{succ}(\texttt{zero})), \ldots\}$$

This inductive definition principle is the basis of the mathematical induction proof method, implemented for each structure-defined type in Athena with its `by-induction-on` construct, as discussed below. First, however, we discuss another important kind of proof method, term rewriting methods for proving equations. The kind of equations we will be most concerned with are ones defining a new function symbol on a type such as `Nat`.

Suppose, for example, we define a `Plus` function. We do so by specifying its behavior axiomatically in terms of equations. This function takes two `Nat` values as parameters and return their sum as a `Nat` value.

```
(declare Plus (-> (Nat Nat) Nat))
```

Here are the propositions we intend to use to define the meaning of `Plus` axiomatically.

```
(define Plus-zero-axiom
  (forall ?n (= (Plus ?n zero) ?n)))
```

```
(define Plus-succ-axiom
  (forall ?n ?m
          (= (Plus ?n (succ ?m))
             (succ (Plus ?n ?m)))))
```

After defining these propositions, we add them as axioms to Athena's assumption base.

```
(assert Plus-zero-axiom Plus-succ-axiom)
```

In the next two sections we will use these axioms first for a purely equational proof and then in proofs involving both the equations and the principle of mathematical induction.

4.4 Term Rewriting Methods

In many proofs, equations such as those defining `Plus` play an important role. When working with equations, one often structures a proof as sequence of *rewriting* steps, in which a term is shown to be equal to another term by means of a substitution that is justified by an equational axiom (or other equational proposition already proved as a theorem). Athena does not currently have such rewriting methods built in, so we have created the following rewriting methods:

with c t

initializes cell c to hold term t. Actually it internally holds the equation $(= t\ t)$, and the reduce and expand methods transform the right hand side of this equation. There are two predefined cells named left and right.

reduce c u E

attempts to transform the term t in cell c to be identical with the given term u by using theorem E (or an appropriate specialization of it, if it contains quantified variables) as a left-to-right rewriting rule. E can be either an unconditional or a conditional equation, in which case, the condition of E has to be in the assumption base.

expand c u E

attempts to transform the term u to the term t in cell c by using theorem E (or an appropriate specialization of it, if it contains quantified variables) as a left-to-right rewriting rule. E can be either an unconditional or a conditional equation, in which case, the condition of E has to be in the assumption base.

combine left right

attempts to combine the equation internally stored in cell left, say $(= t\ t')$, with the equation internally stored in cell right, say $(= u\ u')$, to deduce $(= t\ u)$, succeeding if and only if t' and u' are identical terms).

All these rewriting methods are provided in a file called `utilities.ath`.

As a simple example consider the following deduction in which we prove an equality by reducing both its left and its right hand side term to the same term, `(succ (succ (succ zero)))`.

```
((= (Plus zero (succ (succ (succ zero))))
    (succ (succ (succ (Plus zero zero)))))
 BY (dbegin
     (!with left   (Plus zero (succ (succ (succ zero)))))
     (!with right  (succ (succ (succ (Plus zero zero)))))
     (!reduce left (succ (Plus zero (succ (succ zero))))
              Plus-succ-axiom)
     (!reduce left (succ (succ (Plus zero (succ zero))))
              Plus-succ-axiom)
     (!reduce left (succ (succ (succ (Plus zero zero))))
              Plus-succ-axiom)
     (!reduce left (succ (succ (succ zero ))) Plus-zero-axiom)
     (!reduce right (succ (succ (succ zero))) Plus-zero-axiom)
     (!combine left right)))
```

It should be noted that these rewriting methods, although user-defined, are guaranteed to be logically sound because they are programmed in terms of Athena's primitive methods (assuming Athena's methods are sound).

4.5 Proof by Induction

Consider the following proposition about the `Plus` function:

```
(define Plus-zero-property
  (forall ?n (= (Plus zero ?n) ?n)))
```

Whereas the `Plus-zero-axiom` states that `zero` serves as a right-identity element for the `Plus` operator, this proposition states that it is also a left-identity element. The validity of this property is, however, *not* a consequence simply of the equational axioms we've stated about `Plus` and `zero`; it depends on the fact that `Nat` is inductively defined by the structure declaration `(structure Nat zero (succ Nat))`.

There are two main steps in a proof by induction of a property of natural numbers, as represented by type `Nat`:

- The basis case. This is a special case in which the proposition to be proved is instantiated with `zero`.
- The induction step. We instantiate the proposition with (succ n) for a `Nat` type. After the instantiation we attempt to prove it, assuming the proposition is true for n.

If these steps are successful, we can combine their results using Athena's `by-induction-on` construct to conclude the proposition is true for all n.

An overview of the structure of the proof is as follows:

⟨Proof of the Plus-zero property 29⟩ ≡

```
(Plus-property
 BY (by-induction-on
     ?n
     (= (Plus zero ?n) ?n)
     (zero
      (⟨Basis-case for Plus-zero property 30a⟩
       BY
       ⟨Proof of basis-case for Plus-zero property 30b⟩))
     ((succ n)
      (⟨Induction-step for Plus-zero property 31a⟩
       BY
       ⟨Proof of induction-step for Plus-zero property 31b⟩))))
```

Not used.

Here we are adopting Knuth's *literate programming* style [20], in which code or other formal text (here, proofs) are presented incrementally via cross-referenced "parts" with interspersed commentary, such that the parts can be automatically extracted from the document and assembled into a complete unit for compilation or other processing. In this case the parts referenced are each defined on a subsequent page (as indicated by the number at the end of the part name) but one or more of them could have been defined on a prior page; in general, parts can be presented in whatever order seems best in terms of documentation, as they can be assembled by the extraction tool in a different order as may be required by compilers or other tools. For more information about this style of presentation, see Appendix A.

In general,

by-induction-on V P $(V_1\ P_1)\ldots(V_n\ P_n)$

adds the proposition P to the assumption base if for the induction variable V whose sort is some structure S and for each individual pattern V_i, P_i is the proof of $P(V_i)$ for all possible cases that could be defined with the constructors of S. Each V_i is built from one of the constructors of the structure S. It is an error if any of the cases does not have a successful proof or not all of the possible cases are considered.

4.5.1 Basis case proof

We first need to prove the basis case, which is:

⟨Basis-case for Plus-zero property 30a⟩ ≡

```
    (= (Plus zero zero) zero)
```

Used in part 29.

The proof for this case is simply

⟨Proof of basis-case for Plus-zero property 30b⟩ ≡

```
    (dbegin
      (!with left (Plus zero zero))
      (!reduce left zero Plus-zero-axiom))
```

Used in part 29.

In the proof itself, the `with` applications set up `left` and `right` to hold the left and right hand sides of the equation to be proved. In this simple case only one application of `reduce`, applied to `left`, is sufficient to produce the same term, `zero`, as we have placed in `right`. Note that `reduce` has to specialize the quantified variable `?n` in `Plus-zero-axiom` to `zero` in order to use the result `(= (Plus zero zero) zero)` to substitute `zero` for `(Plus zero zero)`.

4.5.2 Induction step proof

The induction step proof requires a bit more machinery to set up, but again the heart of the proof is reduction using equations. This time, however, the equations we have available to use include not only the axioms but also the induction hypothesis.

We have:

⟨Induction-step for Plus-zero property 31a⟩ ≡

```
(= (Plus zero (succ n)) (succ n))
```

Used in part 29.

⟨Proof of induction-step for Plus-zero property 31b⟩ ≡

```
(dlet ((induction-hypothesis
        (= (Plus zero n) n)))
  (dbegin
   (!with left  (Plus zero (succ n)))
   (!with right (succ n))
   (!reduce left (succ (Plus zero n)) Plus-succ-axiom)
   (!reduce left (succ n) induction-hypothesis)
   (!combine left right)))
```

Used in part 29.

Note that `by-induction-on` automatically adds the induction hypothesis to the assumption base according to the inductive structure of the type.

5 CCP Approach with Examples

In this section we illustrate the CCP approach with two examples. The target language of the code extractor is currently Athena's computation language. But the extractor could easily be retargetted to another more widely used functional language like Scheme or even a functional subset of C.

5.1 Sum Defined on Lists

The `sum` function computes the sum of all natural numbers in a list. It takes two parameters. The first parameter is used for passing the list of numbers and the second one is used to provide an initial value of the computation. To find the sum of all elements in a list `L`, one would pass `L` as the first parameter and `zero` as the second. Appendix B) contains definitions of natural numbers and lists in Athena.

Here is a recursive implementation expressed in Athena's computation language.

```
(define (sum L x)
  (match L
    (Nil x)
    ((Cons n M ) (Plus n (sum M x)))
  )
)
```

We describe below the special form, `match`, which is used in the code shown above:

match E $(\pi_1\ E_1)$ ... $(\pi_n\ E_n)$

matches the patterns π_1, π_2 ,... with the value of E until it finds a match. If any one of the π_i matches then E_i is evaluated and the value of it is returned as the result. An error message will be printed if not match is found.

The following block summarizes the CCP steps to be followed for this example. Each step will be explained in the following subsections.

`"sum.ath"` 32 ≡
  ⟨Requirements for sum on lists 33a⟩
  ⟨Efficient implementation of sum on lists 36a⟩
  ⟨Code Extraction 44⟩

5.1.1 Defining Requirements (Code Consumer)

The rules and the declarations in the requirements are chosen by the code consumer. The requirements for this example include the definitions of natural numbers and lists as types, declarations and propositions that give a specification of the sum function.

The following theorems are located in the library we built for natural numbers (called as "natural-numbers-library" in Figure 7 and Figure 9): `succ-not-zero-axiom`, `Plus-zero-axiom`, `Plus-succ-axiom`, `Plus-zero-property`, `Plus-succ-property`, `Plus-Commutativity`, `Plus-Associativity`.

Similarly there is a list library but it is not used in any of the examples in this paper.

The reader can find more explanations about these libraries in Appendix B). Since we frequently use the definitions of natural numbers and lists in our proofs, we are assuming that these two libraries are part of the requirements and should be loaded by both the code producer and the consumer as a default.

Defining the requirements can be broken down into the following parts:

⟨Requirements for sum on lists 33a⟩ ≡

> ⟨Load lists and natural numbers 69b⟩
> ⟨Declare sum symbol 33b⟩
> ⟨Define axioms for sum 34a⟩
> ⟨Add axioms to assumption base 34b⟩

Used in part 32.

The consumer declares a `sum` symbol first.

⟨Declare sum symbol 33b⟩ ≡

```
    (declare sum ((T) -> ((List-Of T) T) T))
```

Used in part 33a.

The `List-Of` type used above is a built-in structure which has the following inductive definition (that is explained in Appendix B):

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

The next step is to define the `sum` function axiomatically. The following two axioms are used for this purpose. In general the axioms and theorems are in the form of *universally quantified, and possibly conditional, equations.* The left hand side of each equation is an expression in which the function is applied to arguments of a specific form: `Nil` and `?x` in the first equation and `(Cons ?n ?M)` and `?x` in the second.

⟨Define axioms for sum 34a⟩ ≡

```
(define sum-Nil-axiom
    (forall ?x
      (= (sum Nil ?x) ?x)))

(define sum-Cons-axiom
  (forall ?n  ?x ?M
          (= (sum (Cons ?n ?M) ?x)
             (Plus ?n (sum ?M ?x)))))
```

Used in part 33a.

Note that from these equations the consumer could easily extract code implementing the sum function, but we will instead be developing equations from which a more efficient implementation can be extracted.

The consumer add these propositions to Athena's assumption base, in effect treating them as axioms.

⟨Add axioms to assumption base 34b⟩ ≡

```
(assert sum-Nil-axiom sum-Cons-axiom)
```

Used in part 33a.

5.1.2 Defining Correctness Theorem (Code Consumer)

In the CCP system the code consumer may need to declare new function symbols and state theorems which indicate that the functions represented by these symbols are equal to some specified functions. When the code producer receives these symbols and theorems he first defines the symbols with new axioms and then proves the given theorems. Since the consumer is not aware of these new definitions the producer sends them to the consumer along with the proofs of the given theorems. After the consumer gets them he should make the decision of whether to put them into the assumption base or not. In general the consumer *cannot* accept and assert arbitrary definitions and propositions. If this were allowed the producer or the hacker would be able to use them to prove anything.

Now the question is, how can the consumer recognize certain kinds of propositions as constituting a definition of a symbol, and accept them and add them to its assumption base? For this to be possible the new definitions should satisfy some syntactic and semantic properties that guarantee that they cannot introduce any inconsistencies if they are entered into the assumption base. Unless these checks succeed, the consumer cannot accept the definitions.

In our example the `sum-imp` symbol is required to be declared by the consumer as shown below.

⟨Declare sum-imp symbol 35a⟩ ≡

```
(declare sum-imp ((T) -> ((List-Of T) T) T))
```

Used in part 36a.

Finally the consumer states the following "correctness theorem" which says that the specified `sum` function is equal to the `sum-imp` function and enters it into the proof checker's assumption base.

⟨Define correctness-theorem 35b⟩ ≡

```
(define correctness-theorem
    (forall ?M ?x
            (= (sum ?M ?x) (sum-imp ?M ?x))))
```

Used in part 36a.

Now the consumer is ready to send the requirements to the producer.

5.1.3 Efficient Implementation (Code Producer)

After the code producer receives the declarations and definitions mentioned in Section 5.1.1 and Section 5.1.2 it enters the ones explained in Section 5.1.1 into the proof checker's assumption base. The code producer wants to send an efficient implementation of the sum function and a proof that it satisfies the required specification to the code consumer. Rather than sending the efficient sum function as actual code, the producer will send only definitions and proofs.

The following part shows other steps to be done by *both* the producer and the consumer, except that the producer is constructing the proofs and checking them, whereas the consumer is only checking them.

⟨Efficient implementation of sum on lists 36a⟩ ≡

> ⟨Declare sum-imp symbol 35a⟩
> ⟨Define axioms for sum-imp 36b⟩
> ⟨Assert axioms for sum-imp 39⟩
> ⟨Define correctness-theorem 35b⟩
> ⟨Define sum-Plus property 41b⟩
> ⟨Proof of the sum-Plus property 41c⟩
> ⟨Proof of the correctness-theorem 40a⟩

Used in part 32.

The producer starts with defining the `sum-imp` symbol with the following axioms which will be transmitted to the consumer later.

⟨Define axioms for sum-imp 36b⟩ ≡

```
(define sum-imp-Nil-axiom
    (forall ?x
      (= (sum-imp Nil ?x) ?x)))

(define sum-imp-Cons-axiom
  (forall ?n ?x ?M
          (= (sum-imp (Cons ?n ?M) ?x)
             (sum-imp ?M (Plus ?x ?n)))))

(assert sum-imp-Nil-axiom sum-imp-Cons-axiom)
```

Used in part 36a.

One can construct a "tail-recursive" sum function from the axioms given above. While recursion is the essential way of code development in functional programming languages, extremely deep recursions may cause performance degradation. In general with each recursive call, the compiler allocates some stack space in memory. If the number of calls are too many or unbounded, memory consumption will be so high that the program will run out of memory. An alternative methodology which can be used to solve this inefficiency problem is tail-recursion. If one writes tail-recursive code it can be easily optimized to produce an iterative version making it possible to use the same memory space repeatedly. As a result, a tail-recursive version of the function will be more efficient and run faster than an embedded recursive one.

Note that the declaration for `sum-imp` has already been received from the consumer. The producer first checks if the declaration and definition of `sum-imp` are acceptable and don't introduce any inconsistencies if they are entered into the assumption base. If these checks succeed, then the consumer puts them into the assumption base using an assert statement. Specifically the following tests have to be applied for a successful check:

- Syntactic Property. Syntactically, the new definitions of symbols and axioms should be well-formed Athena constructs.

- Internal Consistency. The new definitions introduced should not cause any inconsistencies in the proof checker's assumption base. The Knuth-Bendix procedure [18] can be used to show that the new definitions don't introduce any inconsistencies. When the definitions are received, a new rewrite rule for each of the new axioms is introduced. In the next step the Knuth-Bendix procedure is applied to the resulting set of rewrite rules. Although this procedure may not terminate in some cases, we need to apply it only in simple cases for which termination is guaranteed.

- Termination. The recursive function to be extracted from the axioms has to terminate properly. Although termination is in general undecidable, it is easily proved in special cases by showing there is a well-founded ordering, such as "$<$" on the natural numbers, such that the size, length or any other similar measure of the arguments decreases in the ordering with each recursive call of the function. For example, if the type of argument is `Nat` and the function is called with `(succ (succ zero))` initially, the value of the same argument in any recursive call inside the function should be less than `(succ (succ zero))`. In addition to this, all the statements other then the recursive calls have to terminate.

It is the code producer's responsibility to define the axioms which can be used to extract a function that terminates. One way of checking this is to compare the size of the terms located in both left and right hand side of the rules to be used in code extraction. If, for example, the size is getting reduced on the right hand side it will be safe to use these axioms.

Athena provides a "definitional principle" that permits the sound extension of the system via the introduction of new symbols and their definitions. It has the following command to provide this ability in a limited way:

define-symbol < *symbol* > < *predicate* >

checks that < *symbol* > is not already defined, and that the < *predicate* > satisfies some syntactic and semantic properties that guarantee that it cannot introduce any inconsistencies if it is entered into the assumption base. If these checks succeed, then the < *predicate* > is entered, just as if it had been asserted or proved.

In the current Athena system, `define-symbol` is not fully implemented, but even if it were, it's unlikely we'd be able to use it since it would require a function definition to already be in the form of a recursive function, not a set of equational axioms. For this reason we will need to implement the necessary checks so that we know it is safe to assert the axioms; plans for adding this implementation are discussed in a later section.

For now let's perform the checks on our specific example.

- Syntactic Property: Athena returns an error message if a declaration or definition is not well-formed syntactically. Entering the new definitions will be sufficient to apply this test. Since we don't get any error messages, the new declaration and definition of `sum-imp` are acceptable.

- Internal Consistency: `sum-imp` is a new symbol which is not used in any axioms in the assumption base. Therefore, it is easy to check whether the new axioms are consistent or not. Consider first the `sum-imp-Nil-axiom`. To see that it is consistent we try to superpose the left hand side of the `sum-imp-Nil-axiom` with every non-variable subterm of the left hand side of every rule in the assumption base, as it is done in Knuth-Bendix procedure. But superposition fails since `sum-imp` is a new symbol and therefore cannot appear in other rules. The other symbol, `Nil`, is a pure constructor: it is not a top level symbol in any existing rules. Therefore, there can be no superpositions of left hand sides of other rules with a nonvariable subterm of the left hand side of the new axiom. We repeat the same process for `sum-imp-Cons-axiom`. Again `Cons` is a pure constructor and thus is not a top level symbol in any other rules. Since we cannot get any superpositions for this case either, it is not possible that the new rules cause any inconsistencies.

- Termination: In the first axiom, the left hand side contains the constructor `Nil` representing an empty list but the right hand side does not have any list. Clearly the right hand side is simpler and `Nil` will be a termination condition in code extracted from this axiom.

  In the second axiom, `(Cons ?n ?M)` is a list which is larger than the list `?M` located on the right. Therefore, if we extracted code from this axiom the recursive call will contain a list whose size is reduced in each call.

Since the checks succeed for `sum-imp` the producer enters its declaration and asserts the axioms defining it as follows:

⟨Assert axioms for sum-imp 39⟩ ≡

```
(assert sum-imp-Nil-axiom sum-imp-Cons-axiom)
```

Used in part 36a.

The code producer is now ready to construct the proofs which will be submitted to the consumer. The producer has to prove the correctness theorem so that the consumer can use the definitions of `sum-imp` to get an efficient implementation of the sum function. The code producer constructs its proof from the existing axioms. Since `sum-imp` is also defined in terms of the `List-Of` type, which is inductively-defined, the proof by induction method can be used to prove the `correctness-theorem`.

An overview of the structure of the proof is as follows:

⟨Proof of the correctness-theorem 40a⟩ ≡

```
(correctness-theorem
   BY (by-induction-on
       ?M
       (forall ?x
         (= (sum ?M ?x) (sum-imp ?M ?x)))
       (Nil
        (⟨Basis-case for correctness-theorem 40b⟩
        BY
        ⟨Proof of basis-case for correctness-theorem 40c⟩))
       ((Cons n M)
        (⟨Induction-step for correctness-theorem 41a⟩
        BY
        ⟨Proof of induction-step for correctness-theorem 43⟩))))
```

Used in part 36a.

The variable `?M` is the induction variable. We start with proving the `basis-case`.

⟨Basis-case for correctness-theorem 40b⟩ ≡

```
(forall ?x
    (= (sum Nil ?x) (sum-imp Nil ?x)))
```

Used in part 40a.

Since `sum` works on a `List-Of` structure, the special case in which the property to be proved is the `correctness-theorem` instantiated with `Nil`.

⟨Proof of basis-case for correctness-theorem 40c⟩ ≡

```
(pick-any
      x
    (dbegin
     (!with left  (sum Nil x))
     (!with right  (sum-imp Nil x))
     (!reduce left x sum-Nil-axiom)
     (!reduce right x sum-imp-Nil-axiom)
     (!combine left right)))
```

Used in part 40a.

The next step is to prove the induction-step:

⟨Induction-step for correctness-theorem 41a⟩ ≡

```
(forall ?x
    (= (sum (Cons n M) ?x) (sum-imp (Cons n M) ?x)))
```

Used in part 40a.

The induction-step for `correctness-theorem` uses the following theorem:

⟨Define sum-Plus property 41b⟩ ≡

```
(define sum-Plus-property
    (forall ?M ?y ?x
            (= (Plus ?y (sum ?M ?x))
               (sum ?M (Plus ?y ?x)))))
```

Used in part 36a.

The producer has to provide the proof of this property as well.

We use the same strategy to prove `sum-Plus-property`. An overview of the structure of the proof is as follows:

⟨Proof of the sum-Plus property 41c⟩ ≡

```
(sum-Plus-property
   BY (by-induction-on
        ?M
        (forall ?y ?x
                (= (Plus ?y (sum ?M ?x))
                   (sum ?M (Plus ?y ?x))))
        (Nil
         (⟨Basis-case for sum-Plus property 42a⟩
          BY
          ⟨Proof of basis-case for sum-Plus property 42b⟩))
        ((Cons n M)
         (⟨Induction-step for sum-Plus property 42c⟩
          BY
          ⟨Proof of induction-step for sum-Plus property 42d⟩))))
```

Used in part 36a.

We again prove the basis-case first:

⟨Basis-case for sum-Plus property 42a⟩ ≡

```
(forall ?y ?x
          (= (Plus ?y (sum Nil ?x))
             (sum Nil (Plus ?y ?x))))
```

Used in part 41c.

⟨Proof of basis-case for sum-Plus property 42b⟩ ≡

```
(pick-any
      y x
      (dbegin
       (!with left  (Plus y (sum Nil x)))
       (!with right (sum Nil (Plus y x)))
       (!reduce left (Plus y x) sum-Nil-axiom)
       (!reduce right (Plus y x) sum-Nil-axiom)
       (!combine left right)))
```

Used in part 41c.

The same idea is used to prove the induction-step.

⟨Induction-step for sum-Plus property 42c⟩ ≡

```
(forall ?y ?x
          (= (Plus ?y (sum (Cons n M) ?x))
             (sum (Cons n M) (Plus ?y ?x))))
```

Used in part 41c.

⟨Proof of induction-step for sum-Plus property 42d⟩ ≡

```
(dlet ((induction-hypothesis
          (forall ?y ?x
                  (= (Plus ?y (sum M ?x))
                     (sum M (Plus ?y ?x))))))
    (pick-any
        y x
        (dbegin
         (!with left (Plus y (sum (Cons n M) x)))
         (!with right (sum (Cons n M) (Plus y x)))
         (!reduce left (Plus y (Plus n (sum M x))) sum-Cons-axiom)
         (!expand left (Plus (Plus y n) (sum M x)) Plus-Associativity)
         (!reduce left (Plus (Plus n y) (sum M x)) Plus-Commutativity)
         (!reduce left (Plus n (Plus y (sum M x))) Plus-Associativity)
         (!reduce right (Plus n (sum M (Plus y x))) sum-Cons-axiom)
         (!expand right (Plus n (Plus y (sum M x))) induction-hypothesis)
         (!combine left right))))
```

Used in part 41c.

These steps complete the proof of `sum-Plus-property` shown in the `by-induction-on` deduction. Since the proof successfully deduces `sum-Plus-property`, it is added into the assumption base and can be used in the proof of other propositions.

In particular, the producer can use `sum-Plus-property` as a lemma to prove the induction-step of the `correctness-theorem`.

⟨Proof of induction-step for correctness-theorem 43⟩ ≡

```
     (dlet ((induction-hypothesis
               (forall ?x
                 (= (sum M ?x) (sum-imp M ?x)))))
         (pick-any
             x
           (dbegin
            (!with left (sum (Cons n M) x))
            (!with right (sum-imp (Cons n M) x))
            (!reduce left (Plus n (sum M x)) sum-Cons-axiom)
            (!reduce left (sum M (Plus n x)) sum-Plus-property)
            (!reduce left (sum M (Plus x n)) Plus-Commutativity)
            (!reduce right (sum-imp M (Plus x n)) sum-imp-Cons-axiom)
            (!expand right (sum M (Plus x n)) induction-hypothesis)
            (!combine left right))))
```

Used in part 40a.

Definitions to be added to the initial requirements
sum-imp-Nil-axiom sum-imp-Cons-axiom

Table 1: Definitions to be sent by the code Producer

Since we proved the `correctness-theorem` we can conclude that both the `sum` and `sum-imp` do the same job. The only difference is that the latter can be used to provide an efficient implementation.

Table 1 shows the additional definitions to be done by the code producer. After the producer constructs the definitions it should send them to the consumer.

5.1.4 Checking the Proofs (Code Consumer)

After the consumer gets the proofs and new definitions from the producer it first checks if the new definitions are acceptable. For this to be possible it applies the same tests explained in Section 5.1.3. We already showed that the new definitions don't introduce any inconsistencies and can be used for extraction of a terminating function. Since the checks succeed the consumer puts them into the assumption base with an assert directive.

The next step is to check the proofs. The consumer enters them one by one into Athena and checks if they are verified. If Athena returns an error message for any one of them then they won't be acceptable. In that case the code extraction cannot be done. In our specific example, the consumer gets two proofs from the producer; one for the `sum-Plus-property` and the other one for the `correctness-theorem`. Once the proof of `sum-Plus-property` is checked in Athena, it will be added to the consumer's assumption base. The next step is to check the proof of `correctness-theorem`. Figure 7 shows the CCP steps for this example in detail.

5.1.5 Code Extraction (Code Consumer)

The following part lists the steps to be done during the code extraction phase.

$\langle$Code Extraction 44$\rangle \equiv$

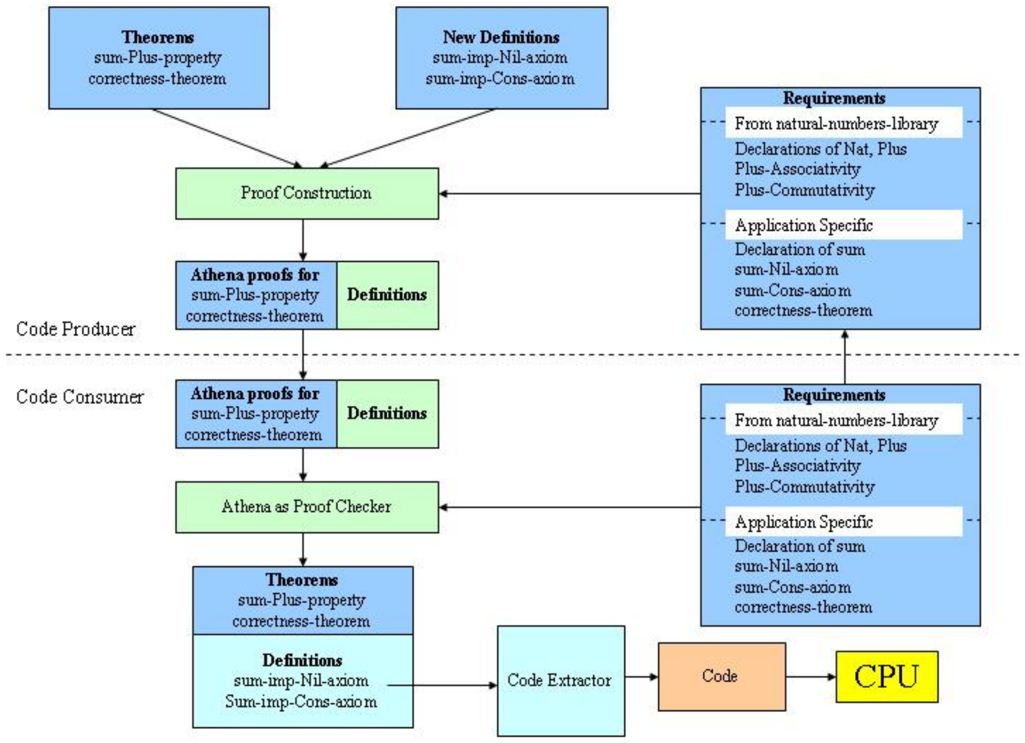


Figure 7: CCP steps for sum defined on lists

⟨Define the rules for sum-imp 46a⟩
⟨Extract sum-imp 46b⟩

Used in part 32.

Because proof checking succeeded during the previous step the consumer can use any axioms or theorems in the assumption base for code extraction. Figure 8 shows the extraction of the tail-recursive sum function from `sum-imp-Nil-axiom` and `sum-imp-Cons-axiom`. In general the code extractor needs to get the following arguments:

- the name of the function to be extracted.
- a list of propositions which define the function we want to construct.
- a list of variables as the parameters of this function.

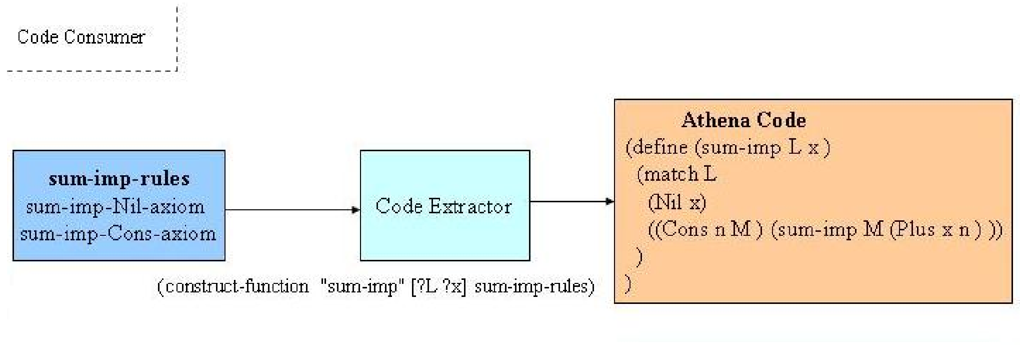


Figure 8: Code Extraction step for sum defined on lists

Note that it is the code producer's responsibility to send a designation of which axioms should be used for each code extraction.

⟨Define the rules for sum-imp 46a⟩ ≡

```
(define sum-imp-rules
    [(urep sum-imp-Nil-axiom [?x])
     (urep sum-imp-Cons-axiom [?n ?x ?M])])
```

Used in part 44.

The method `urep` is a built-in universal specialization method:

urep P $[t_1 \ldots t_n]$

replaces the first n quantified variables in P with the terms $t_1, \ldots, t_n$.

Our code extractor is called `construct-function` which is located in the file called `code-extractor.ath`. We first load this file and then extract `sum-imp` by using the given parameters in the following part.

⟨Extract sum-imp 46b⟩ ≡

```
(load-file "code-extractor.ath")

(construct-function  "sum-imp" [?L ?x] sum-imp-rules)
```

Used in part 44.

Here is the output we get for this case:

```
(define (sum-imp L x )
  (match L
    (Nil x)
    ((Cons n M ) (sum-imp M (Plus x n ) ))
  )
)
```

5.2 Sum Defined with Iterator Ranges

Our second example is a different version of `sum` that computes the sum of the values in a sequence determined by iterators, as in the STL (`sum` models the STL accumulate function template, except that in this version the type of values in the sequence is `Nat`, rather than being a template parameter.).

As in the previous example the code producer has to prove some theorems that show that the results computed by the non-tail recursive and tail-recursive `sum` with iterators are the same.

The following part contains the required steps in CCP approach to get the proof of correctness and extract the tail-recursive sum with iterators.

`"sum-range.ath"` 47 ≡

```
(load-file "range-induction.ath")
⟨Declare sum0 symbol with iterator ranges 48⟩
⟨Define axioms for sum0 49a⟩
⟨Declare sum symbol with iterator ranges 49b⟩
⟨Define sum-correct with iterator ranges 49c⟩
⟨Define sum symbol with iterator ranges 50⟩
⟨Assert axioms defining sum0 and sum 51a⟩
⟨Define sum0-property with iterator ranges 51b⟩
⟨Construct P with sum0-property 51c⟩
⟨Prove basis case for sum0-property 52a⟩
⟨Prove induction step for sum0-property 52b⟩
⟨Setting up the range-induction definitions for sum0-property 54a⟩
⟨Construct P with sum-correct 54b⟩
⟨Prove basis case for sum-correct 54c⟩
⟨Prove induction step for sum-correct 55⟩
⟨Setting up the range-induction definitions for sum-correct 56b⟩
```

⟨Define the rules for sum with iterator ranges 57⟩
⟨Extract sum with iterator ranges 58⟩

5.2.1 Formalization of Valid Range Property in Athena

The Standard Template Library (STL) [33, 25] is a C++ library of generic algorithms, containers, and iterators which make it possible to write efficient generic code. Some of the STL containers are `vector`, `list`, `set`, and `map`. The data in the containers are accessed by C++ pointer-like objects called iterators. STL provides standard algorithms to solve most common problems like sorting, searching, merging, copying, etc. The algorithms are adaptable and easy to use.

Accessing sequence elements through a pair of iterators is another important operation used in STL-style generic code. A pair of iterators that points to the beginning and end of a computation is called *range*. For example, a range $[i, j)$ refers to the elements in the container starting with the one pointed to by i and up to but not including the one pointed to by j. When accessing the element, not just any pair of iterators can be used, however; the pair must comprise a valid range. A pair of iterators constitute valid range if and only if the first iterator will become equal to the second after a (finite, possibly empty) sequence of applications of `operator++`.

The `range.ath` file which is included in Appendix C contains some definitions, theorems and axioms about iterators and iterator ranges. This is our preliminary attempt to formalizing STL iterator concepts. This file is loaded automatically when the file `range-induction.ath` is loaded. We refer to this library of definitions as the "iterator-ranges-library" in later discussion and figures.

5.2.2 Defining sum with iterator ranges (Code Consumer)

The code consumer starts with declaring the `sum0` symbol which will be used to specify the `sum` function.

⟨Declare sum0 symbol with iterator ranges 48⟩ ≡

```
(declare sum0 ((T) -> ((Range (Iterator T)) T) T))
```

Used in part 47.

The following two axioms are used for defining the `sum0`. In the case that the two iterators are equal we have an empty range and the summation should be equal to the second parameter which is `?x` here. The second axiom shows the operation to be performed if the two iterators are not equal to each other. In this case the content of the element pointed by the first iterator is added to the summation of the remaining elements in the new range we get after applying the operator++ to the first iterator.

⟨Define axioms for sum0 49a⟩ ≡

```
(define sum0-empty-range-axiom
  (forall ?i ?j ?x
          (if (= ?i ?j)
              (= (sum0 (range ?i ?j) ?x)
                 ?x))))

(define sum0-nonempty-range-axiom
  (forall ?i ?j ?x
          (if (not (= ?i ?j))
              (= (sum0 (range ?i ?j) ?x)
                 (Plus (eval (* ?i))
                       (sum0 (range (++ ?i) ?j) ?x))))))
```

Used in part 47.

From this specification one could easily get a recursive function that computes the sum of numbers given in a range. But the recursion would be embedded and therefore not as efficient as a tail-recursive version (which we regard as the equivalent of a loop).

The next step is to introduce a correctness theorem. The consumer first declares the following `sum` symbol.

⟨Declare sum symbol with iterator ranges 49b⟩ ≡

```
(declare sum ((T) -> ((Range (Iterator T)) T) T))
```

Used in part 47.

The correctness theorem is then expressed by the consumer as follows:

⟨Define sum-correct with iterator ranges 49c⟩ ≡

```
(define sum-correct
  (forall ?i ?j ?x
          (if (valid (range ?i ?j))
              (= (sum (range ?i ?j) ?x)
                 (sum0 (range ?i ?j) ?x)))))
```

Used in part 47.

Finally, the consumer sends the declarations of both the `sum0` and `sum` symbols, the definition of `sum0`, and the correctness theorem `sum-correct` to the code producer.

5.2.3 Efficient Implementation (Code Producer)

After receiving the declarations the producer first defines the `sum` symbol.

⟨Define sum symbol with iterator ranges 50⟩ ≡

```
(define sum-empty-range-axiom
  (forall ?i ?j ?x
          (if (= ?i ?j)
              (= (sum (range ?i ?j) ?x)
                 ?x))))

(define sum-nonempty-range-axiom
  (forall ?i ?j ?x
          (if (not (= ?i ?j))
              (= (sum (range ?i ?j) ?x)
                 (sum (range (++ ?i) ?j)
                      (Plus (eval (* ?i)) ?x))))))
```

Used in part 47.

The next step is to apply the tests explained in Section 5.1.3 to see if these new definitions are acceptable. Syntactically, they are well-formed. Athena will not return any errors if they are entered. For the consistency check, since `sum` is a new symbol which is not being used in any other rules that exist in the assumption base and since `range` is a pure constructor, superpositions will not be possible with any of the axioms. Therefore, they cannot introduce any inconsistencies. To prove termination, we check the left and right hand side of each axiom for reduction in some well-formed ordering. For the first axiom, the right hand side `?x` is a proper subterm of the left hand side and is therefore smaller in the subterm ordering. The second axiom reduces the size of the range; this is a consequence of the assumption that we begin with a valid range.

Since the checks succeed the producer enters the definitions of both `sum` and `sum0` into Athena with an assert directive.

⟨Assert axioms defining sum0 and sum 51a⟩ ≡

```
(assert sum0-empty-range-axiom
  sum0-nonempty-range-axiom
  sum-empty-range-axiom
  sum-nonempty-range-axiom)
```

Used in part 47.

While proving the correctness theorem the code producer will need the following property which will be proved by using proof by range-induction.

⟨Define sum0-property with iterator ranges 51b⟩ ≡

```
(define sum0-property
  (forall ?i ?j
          (if (valid (range ?i ?j))
              (forall ?x ?k
                      (= (sum0 (range ?i ?j)
                               (Plus (eval (* ?k)) ?x))
                         (Plus (eval (* ?k))
                               (sum0 (range ?i ?j) ?x)))))))
```

Used in part 47.

To be able to use the proof by range-induction, the `sum0-property` is stated in a function called `P` in the following format:

⟨Construct P with sum0-property 51c⟩ ≡

```
(define (P r)
  (match r
    ((range i j)
     (forall ?x ?k
             (= (sum0 (range i j) (Plus (eval (* ?k)) ?x))
                (Plus (eval (* ?k)) (sum0 (range i j) ?x)))))))
```

Used in part 47.

The producer can now prove the basis case of `sum0-property`. The `(range i i)` represents an empty range and is the basis case for proof by range-induction. To get an empty range the quantified variables `?i` and `?j` in the proposition `sum0-property` should be instantiated with `i`.

⟨Prove basis case for sum0-property 52a⟩ ≡

```
((range-basis-case P)
 BY
 (pick-any i
   (pick-any
       x k
       (dbegin
        (!with left (sum0 (range i i) (Plus (eval (* k)) x)))
        (!with right  (Plus (eval (* k)) (sum0 (range i i) x)))
        (!eq-reflex i)
        (!reduce left (Plus (eval (* k)) x) sum0-empty-range-axiom)
        (!reduce right (Plus (eval (* k)) x) sum0-empty-range-axiom)
        (!combine left right)))))
```

Used in part 47.

Here is the induction step:

⟨Prove induction step for sum0-property 52b⟩ ≡

```
((range-induction-step P)
 BY
 (pick-any
     i j
     (assume-let
         ⟨Assumptions for the induction step of sum0-property 53⟩
         (pick-any
             x k
             (dbegin
              (!left-and assumptions)
              (!left-and (!right-and assumptions))
              (!uspec* (!right-and (!right-and assumptions)) [x k])
              (!with left (sum0 (range i j) (Plus (eval (* k)) x)))
              (!with right (Plus (eval (* k)) (sum0 (range i j) x)))
```

```
                  (!reduce left (Plus (eval (* i)) (sum0 (range (++ i) j)
                                                         (Plus (eval (* k)) x)))
                           sum0-nonempty-range-axiom)
                  (!reduce left (Plus (eval (* i))
                                      (Plus (eval (* k))
                                            (sum0 (range (++ i) j) x)))
                           (= (sum0 (range (++ i) j) (Plus (eval (* k)) x))
                              (Plus (eval (* k)) (sum0 (range (++ i) j) x))))
                  (!expand left (Plus (Plus (eval (* i)) (eval (* k)))
                                      (sum0 (range (++ i) j) x))
                           Plus-Associativity)
                  (!reduce left (Plus (Plus (eval (* k)) (eval (* i)))
                                      (sum0 (range (++ i) j) x))
                           Plus-Commutativity)
                  (!reduce right (Plus (eval (* k))
                                       (Plus (eval (* i))
                                             (sum0 (range (++ i) j) x)))
                           sum0-nonempty-range-axiom)
                  (!expand right (Plus (Plus (eval (* k)) (eval (* i)))
                                       (sum0 (range (++ i) j) x))
                           Plus-Associativity)
                  (!combine left right))))))
```

Used in part 47.

For the induction step we first assume that the iterators `i` and `j` are not equal to each other and constitute a valid range. The third assumption is that the proposition `sum0-property` is true for `(range (++ i) j)`.

⟨Assumptions for the induction step of sum0-property 53⟩ ≡

```
((assumptions
  (and (not (= i j))
       (and (valid (range i j))
            (forall ?x ?k
                    (= (sum0 (range (++ i) j)
                             (Plus (eval (* ?k)) ?x))
                       (Plus (eval (* ?k))
                             (sum0 (range (++ i) j) ?x))))))))
```

Used in part 52b.

After setting up the assumptions we first instantiate `sum0-property` with `(range i j)` whose length is larger than the length of `(range (++ i) j)` by one. Finally, we apply reductions to both `left` and `right` to get the same term at the end.

⟨Setting up the range-induction definitions for sum0-property 54a⟩ ≡
```
(declare Q ((T) -> ((Range T)) Boolean))

(assert
    (forall ?i ?j
            (iff (Q (range ?i ?j))
                 (P (range ?i ?j)))))

(!range-induction Q P)
```
Used in part 47.

The code producer can now prove the correctness theorem with the same strategy. Here is the proof for `sum-correct`:

⟨Construct P with sum-correct 54b⟩ ≡
```
(define (P r)
  (match r
    ((range i j)
     (forall ?x
       (= (sum (range i j) ?x)
          (sum0 (range i j) ?x))))))
```
Used in part 47.

Basis case proof:

⟨Prove basis case for sum-correct 54c⟩ ≡
```
((range-basis-case P)
 BY
 (pick-any
     i x
     (dbegin
      (!with left (sum (range i i) x))
      (!with right (sum0 (range i i) x))
      (!eq-reflex i)
      (!reduce left x sum-empty-range-axiom)
      (!reduce right x sum0-empty-range-axiom)
      (!combine left right)))))
```

Used in part 47.

Induction step:

⟨Prove induction step for sum-correct 55⟩ ≡

```
((range-induction-step P)
 BY
 (pick-any
     i j
     (assume-let
         ⟨Assumptions for the induction step of sum-correct 56a⟩
         (pick-any
             x
           (dbegin
            (!left-and assumptions)
            (!left-and (!right-and assumptions))
            (!uspec (!right-and (!right-and assumptions))
                    (Plus (eval (* i)) x))
            (!with left (sum (range i j) x))
            (!with right (sum0 (range i j) x))
            (!reduce left (sum (range (++ i) j) (Plus (eval (* i)) x))
                     sum-nonempty-range-axiom)
            (!reduce left (sum0 (range (++ i) j) (Plus (eval (* i)) x))
                     (= (sum (range (++ i) j) (Plus (eval (* i)) x))
                        (sum0 (range (++ i) j) (Plus (eval (* i)) x))))
            ((valid (range (++ i) j))
             BY
             (dbegin
              (!both (valid (range i j)) (not (= i j)))
              (!specialize (!theorem
                             ['Valid-Range 'Step-Property]
                             VNR1 Range-theorems) [i j])))
            (!reduce left (Plus (eval (* i))
                                (sum0 (range (++ i) j) x))
                     (!specialize sum0-property [(++ i) j]))
            (!reduce right (Plus (eval (* i)) (sum0 (range (++ i) j) x))
                     sum0-nonempty-range-axiom)
            (!combine left right))))))
```

Used in part 47.

Assumptions for induction step:

⟨Assumptions for the induction step of sum-correct 56a⟩ ≡

```
((assumptions
  (and (not (= i j))
       (and (valid (range i j))
            (forall ?x
              (= (sum (range (++ i) j) ?x)
                 (sum0 (range (++ i) j) ?x)))))))
```

Used in part 55.

⟨Setting up the range-induction definitions for sum-correct 56b⟩ ≡

```
(assert
    (forall ?i ?j
            (iff (Q (range ?i ?j))
                 (P (range ?i ?j)))))

(!range-induction Q P)
```

Used in part 47.

5.2.4 Checking the Proofs (Code Consumer)

After proving the correctness theorem the producer sends the proofs and new definitions to the consumer. For this example `sum` is the only symbol defined by the producer. The consumer has already asserted the definitions of `sum0`. When he gets the following new definitions of `sum` he applies the same tests applied by the producer. We already showed that the new definitions don't introduce any inconsistencies and can be used for an extraction of a terminating function.

Since the checks succeed the consumer puts them into the assumption base with an assert directive.

The next step is to verify the proof of the correctness theorem. The consumer first checks the proof of `sum-property` since it is used in the proof of the correctness theorem. Then he checks the proof of `sum-correct`. They both will be validated. Figure 9 shows the CCP steps for this example in detail.

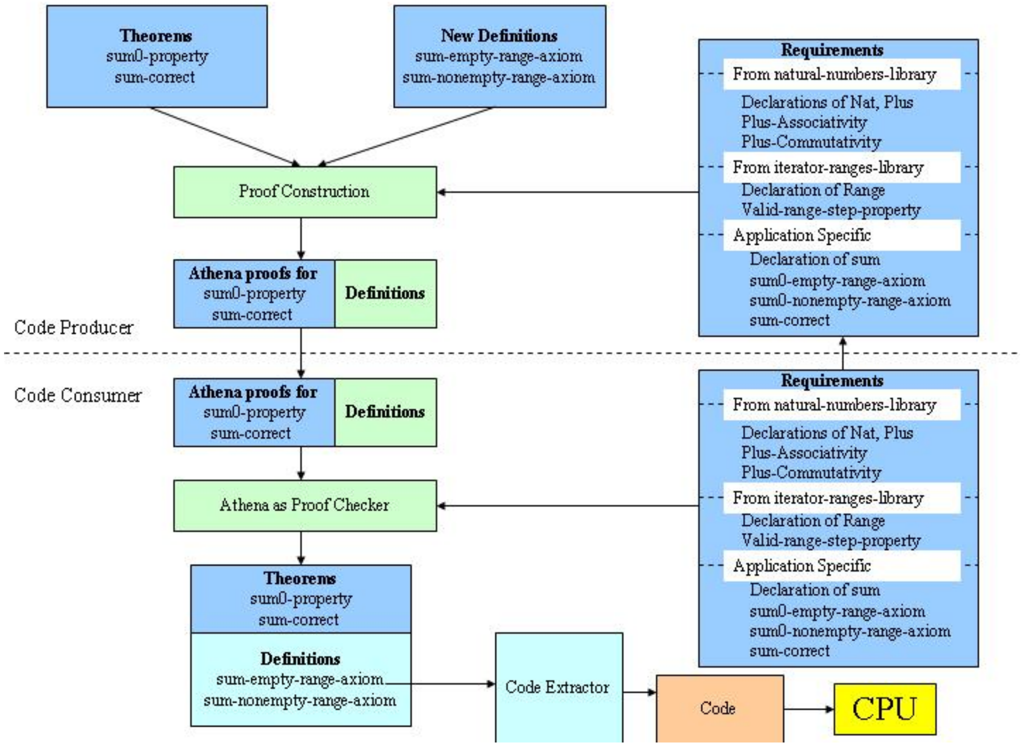


Figure 9: CCP steps for sum defined with iterator ranges

5.2.5 Code Extraction (Code Consumer)

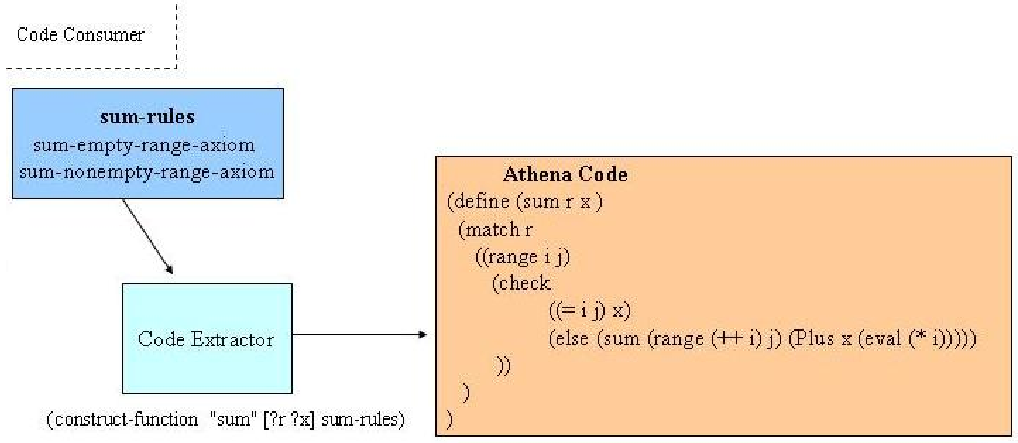
The code consumer repeats the same process in the order which has been done for the previous example to extract the code. Figure 10 shows the extraction process for this example.

We first specialize the theorems which are defining the function to be extracted and put the specializations in a list:

⟨Define the rules for sum with iterator ranges 57⟩ ≡

```
(define sum-rules
  [(urep sum-empty-range-axiom [?i ?j ?x])
   (urep sum-nonempty-range-axiom [?i ?j ?x])])
```

Used in part 47.



```
(define (sum r x )
 (match r
   ((range i j)
     (check
              ((= i j) x)
              (else (sum (range (++ i) j) (Plus x (eval (* i)))))
     ))
   )
)
```

Figure 10: Code Extraction step for sum defined with iterator ranges

This time the sum function takes two parameters. We construct another list which consists of these parameters and pass this list with the function-defining axioms to the code extractor as shown below:

⟨Extract sum with iterator ranges 58⟩ ≡

```
(load-file "code-extractor.ath")

(construct-function  "sum" [?r ?x] sum-rules)
```

Used in part 47.

Here is the output we get:

```
(define (sum r x )
  (match r
     ((range i j)
        (check
             ((= i j) x)
             (else (sum (range (++ i) j) (Plus x (eval (* i))))))
         ))
    )
)
```

6 Future Work

6.1 New Definitions and Tests to be Applied

One way of doing consistency and termination checks is to use Boyer-Moore's approach in Nqthm [32]. In this approach, the definitional principle allows the user to define *new functions* in the logic. To admit a new function under the definitional principle, the function is used as an input and a decision is made whether this function is acceptable. In CCP, since the code extractor generates a function from the axioms, Nqthm's test can be applied to this extracted code to decide whether it is acceptable or not. If it is acceptable we can make the decision that the new definitions don't introduce any inconsistencies or termination problem. We need to explore if this alternative works in CCP.

6.2 Defining Memory

We are in the process of defining a model of memory to use in proofs about state-updating procedures. We axiomatically defined operations like `Access`, `Assign`, `Swap` and their counterparts which can work on iterator ranges. We have started proving some simple theorems that could be used in more complicated proofs. However, the specification of memory and these operations needs to be improved, because we have ignored some efficiency problems. For example, assignment to a single memory location has to be done efficiently. The `Assign` operator we defined does not actually update the memory being accessed. Instead it returns a whole new copy of the memory. We believe this problem can be solved by the code extractor during an optimization phase. The code extractor can check either the axioms which define specific memory accesses and assignments or optimize the code extracted during the first phase of extraction to construct the code that does state changing memory operations. For this to be possible, we should also be able to define memory with a vector-like data structure in Athena. We believe that by using the `cell` structure and `set` command we can create a similar data structure. If we succeed then the code extractor will be capable of producing code that defines destructive functions too.

6.3 More Proof Examples

Since we defined the memory and can do operations on it we can now define and use more update functions. Having many different and more complicated update functions will require our code extractor to be more complicated too. We plan to have additional proof examples. We may define functions like in-place `reverse`, `rotate`, `copy`, `insertion sort` and carry out more difficult proofs. If time permits, a sorting algorithm, or set operations like `intersection`, `union`, and `difference` might be other good choices.

6.4 Memory Safety and Security

We are currently aiming to solve the "functional correctness" problem. In the future, it will be possible to provide solutions to the "memory safety" related problems. By having the specification of memory we are now able to define some memory safety policies.

Even though we believe that CCP can be improved to provide solutions for security problems too, we are not planning to go in that direction.

6.5 Complexity Issue - Proof Size vs Code Size

In general, our Athena proofs are easy to read, maintain and are machine checkable. However, the proof sizes are very large compared to the amount of code to be extracted. Our main approach for addressing this problem is to do generic proofs and instantiate them by using different parameters when needed. We believe this can reduce the ratio of proof size to code size dramatically.

It would be good to be able to decide the complexity of how the proof size grows with the code size. Nevertheless, we do not plan to compare proof sizes to the code to be extracted in this project.

6.6 Generic Proofs and Proof Packaging

We will continue to write generic proofs and construct generic proof libraries. We investigate the ways of packaging the proofs which will make it easier to reuse them by just instantiating with different parameters.

6.7 Creating Proofs Automatically

In this project we have not aimed to create proofs automatically. Our proofs are hand written. In the future versions of CCP, automatic generation of proofs might be a possibility. This would make it easier to deal with more complex problems since hand-construction of a proof by a human can be very time-consuming.

6.8 Other Problems

The code producer or a hacker may repeatedly send files to the consumer ("Denial of Service Attack") or send very large files causing the consumer's memory to be full. In the first case the consumer can refuse to receive files from outside if too many files are being sent. In the second case the consumer may limit the size of the files to be accepted from the producer.

Another problem is how to exchange the requirements. There are two possible scenarios:

- The consumer sends everything to the producer. For example, one producer may need to use definitions of natural numbers. In this case the producer may not have these definitions and expect to get them from the consumer.
- They both may have some common libraries like definitions of natural numbers and lists. In this case the consumer will just need to send application specific definitions to the producer. We are currently using this second approach.

7 Schedule/Timetable for the remaining work

Some major steps to finish this thesis are listed below with estimated finishing dates (Some of these tasks will be done in parallel based on a full-time work).

- Completing the development of the memory theory and more proof examples (until December 1)
- Improving and Testing the Code Extractor (until January 1)
- Providing solutions for Consistency Check, Termination and other checks (until January 15)

- Testing the CCP approach and Code Extractor (until March 1)
- Dealing with proof packaging, generic proofs and other problems (until April 1)
- Writing a paper (by working in parallel until May)
- Completing the thesis (Summer 2005)

References

[1] A. Ahmed, A. Appel, and R. Virga. A stratified semantics of general references embeddable in higher-order logic, 2002. 3.1

[2] Roberto M. Amadio and Denis Lugiez. On the reachability problem in cryptographic protocols. *Lecture Notes in Computer Science*, 1877:380+, 2000. 1, 3

[3] Andrew W. Appel. Foundational proof-carrying code. In *Symposium on Logic in Computer Science*, pages 247–258, boston, 2001. IEEE Computer Society Press. 1, 2.1, 2.2, 2.2.1

[4] Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *Programming Languages and Systems*, 23(5):657–683, 2001. 3.1

[5] Konstantine Arkoudas. *Denotational Proof Languages*. PhD thesis, MIT, 2000. 1, 3.2, 4

[6] Konstantine Arkoudas. Certified computation, 2001. `citeseer.nj.nec.com/arkoudas01certified.html`. 3.2

[7] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The Coq proof assistant reference manual: Version 6.1. Technical Report RT-0203, 1997. `citeseer.nj.nec.com/barras97coq.html`. 1, 3

[8] Preston Briggs. Nuweb: A simple literate programming tool. Technical report, 1993. A

[9] James L. Caldwell and John Cowles. Representing Nuprl proof objects in ACL2: toward a proof checker for nuprl. `citeseer.nj.nec.com/caldwell02representing.html`. 1, 3

[10] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001. 3.2

[11] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth 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. 2.2

[12] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986. 1, 3

[13] Premkumar T. Devanbu, Philip W. L. Fong, and Stuart G. Stubblebine. Techniques for trusted software engineering. In *Proceedings of the 1998 International Conference on Software Engineering: ICSE 98, Kyoto, Japan*, pages 126–135, Los Alamitos, California, 1998. 1

[14] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. In *Conference on Automated Deduction*, pages 653–654, 1990. `citeseer.nj.nec.com/farmer93imps.html`. 3.2

[15] R. W. Floyd. Assigning meanings to programs. In J.T.Schwartz, editor, *In Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967. 2.1.2

[16] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993. 2.1

[17] D. Kapur and D. R. Musser. Tecton: A language for specifying generic system components. Technical Report 92-20, Rensselaer Polytechnic Institute Computer Science Department, July 1992. 3.2

[18] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, Elmsford, N.Y., 1970. 5.1.3

[19] Donald E. Knuth. The web system of structured documentation. Technical report, 1983. A

[20] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984. 4.5

[21] Christopher League, Zong Shao, and Valery 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. 2.2

[22] Silvio Levy. Web adapted to c, another approach. *TUGBoat*, 8(1):12–13, Apr 1987. A

[23] David R. Musser. Automated theorem proving for analysis and synthesis of computations. In G. Birtwistle and P. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer-Verlag, New York, 1989. 1

[24] David R. Musser. Tecton description of STL container and iterator concepts. `http://www.cs.rpi.edu/~musser/gp/tecton/container.ps.gz`, August 1998. 3.2

[25] David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library Second Edition.* Addison-Wesley, 2001. 5.2.1

[26] David R. Musser, Sibylle Schupp, Christoph Schwarzweller, and Rüdiger Loos. Tecton Concept Library. Technical Report WSI-99-2, Fakultät für Informatik, Universität Tübingen, January 1999. 3.2

[27] George 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. 1

[28] George Ciprian Necula. *Compiling with Proofs*. PhD thesis, CMU, 1998. 2.1.1

[29] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, CA, 1989. 2.1.3

[30] Frank Pfenning and Carsten Schurmann. System description: Twelf — a meta-logical framework for deductive systems. In *In the 16th International Conference on Automated Deduction*. Springer-Verlag, July 1999. 2.2.1

[31] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, Sep 1994. A

[32] R.S.Boyer and J S. Moore. *A Computational Logic*. Academic Press, 1979. 6.1

[33] Alexander Stepanov and Meng Lee. The standard template library. Technical Report HPL-TR-95.11, 1995. 5.2.1

[34] Markus Wenzel. The Isabelle/Isar Reference Manual, 2001. `citeseer.nj.nec.com/article/wenzel01isabelleisar.html`. 3.2

Figure 11: Code and document generation using nuweb

A Literate Programming

Knuth introduced *Literate programming* for a style of program development in which code and its documentation are developed simultaneously, in a single source file. When the programmer implements the code he should preserve a logical order depending on the implementation environment. However, when it comes to explaining the interacting parts and particular commands in the code the programmer may prefer to use any order that would help him to describe it better. With Literate programming style he would be free to use any order to document by placing the code blocks anywhere in the source file.

Knuth used a tool called `WEB` [19] for literate programming. Since Knuth's original paper, several variations on literate programming have been developed. The original `Web` tool required coding in Pascal. Many variations on `Web` have been developed, including `Cweb` [22], for coding in C or C++ (Knuth describes `Cweb` as his favorite programming language); `Noweb` [31], which is programming-language independent: one can code in any language; and `Nuweb` [8], which is also programming-language independent, and has a very simple command set. We are currently using `nuweb`.

Running `nuweb` on the `nuweb` source file would produce two different types of files:

- an Athena Program or programs,
- a latex file which could be processed by latex. The next step would probably be to generate a pdf file from the latex source. We use `pdflatex` for this process.

Figure 5 shows how we use `nuweb` to generate both the Athena code and the latex file. In fact there might be many small programs located in the `nuweb` file. For example, one could describe both an Athena program and test files named differently and implemented in Athena in the same source file. `nuweb` would be successful to output each of these programs as separate files, ready to be loaded in Athena.

Since `nuweb` is programming-language independent it would be possible to write the programs in any other language too. For example, one can implement the code in C/C++ or in his any other favorite programming language and can explain what the code does using `nuweb`. The extracted C/C++ code would be ready for compilation.

The following components are essential to write a `nuweb` file.

- parts:

  In general when using `nuweb` the programmer starts with naming the main program to be extracted with "parts". Each part in `nuweb` contains either code, or references to definitions, or combinations of both. Parts can be defined with the following command:

  ```
  @O file-name flags @{

  ...

  @}
  ```

  The file-name is the name of the output file to be extracted after running `nuweb`. The ... represents the part where we can put either some code and/or references into the definitions. The flags are optional. Sometimes it is helpful to use them. For example, `-d` flag can be used to write line numbers when producing the output file. These line numbers refer to the actual locations of the code lines located in the `nuweb` file. This would be helpful if debugging is required and the programmer prefers to work on the source code located in the `nuweb` file.

- definitions:

  Each definition will have either some code fragments or references to the other definitions, or combinations of both. "Definitions" can be defined with the following command:

  ```
  @D file-name @{

  ...

  @}
  ```

- references to the definitions:

  When describing the code the programmer may want to divide the code into small blocks and refer each block whenever possible. “References to the definitions” can be used for this purpose. Here is the command to define a reference.

  ```
  @<definition-name@>
  ```

When `nuweb` encounters a definition-name it expands the code referred by that definition-name into the body of its parent part or definition as the resultant code is written out to a file.

B Defining Natural Numbers and Lists in Athena

According to the built-in semantics of `structure`, the only values of type `Nat` are those that can be expressed with *syntactically-correct* and *type-correct* combinations of `zero` and `succ` terms:

```
zero
(succ zero)
(succ (succ zero))
(succ (succ (succ zero)))
...
```

For such *inductively-defined* types, we can prove propositions using the method of proof by induction.

The next type we frequently use is `List-Of` which is a built-in structure in Athena. The following usage of `structure` directive introduces an additional property. Athena's type system allows *polymorphism*: we can declare polymorphic lists as a structure:

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

Here `T` is a type variable that can be instantiated with any type, so that, for example,

- `(Cons zero (Cons (succ zero) Nil))` is a term of type `(List-Of Nat)`.

- `(Cons true Nil)` is a term of type `(List-Of Boolean)`;
- however, `(Cons zero (Cons true Nil))` is an error ("Ill-sorted term") since all elements of a `List-Of` must have the same type.

We introduce another unary constructor here. `Cons` is used for inserting a new element in front of a given list. The `List-Of` type has an inductive structure which will be used for doing proofs by induction on *lists*.

After defining the structure `Nat` and `List-Of` they are ready to be used in function declarations. For example, the following declaration of `insert` function tells us that it takes a natural number and a list of natural numbers as an input and returns another list of natural numbers as an output.

⟨set-of-Nat 69a⟩ ≡
```
    (declare insert (-> (Nat (List-Of Nat)) (List-Of Nat)))
```

Not used.

Since we frequently use the natural numbers and lists in our implementations and proofs we have already defined them. The file `induction-naturals1.ath` contains the declarations of `Nat` and functions using natural numbers as well as some related Athena theorems and their proofs. We will refer to this file as "natural-numbers-library". Similarly the file `induction-lists1.ath` provides functions using lists with related theorems and their proofs. This file is called "lists-library" in this document. Athena's `load` directive can be used to load both of these files as shown below:

⟨Load lists and natural numbers 69b⟩ ≡
```
    (load-file "induction-lists1.ath")
    (load-file "induction-naturals1.ath")
```

Used in part 33a.

Here is a list of theorems about natural numbers:

```
    (define succ-not-zero-axiom
      (forall ?n
        (not (= (succ ?n) zero))))
```

```
(define Plus-zero-axiom
  (forall ?n
    (= (Plus ?n zero) ?n)))

(define Plus-succ-axiom
  (forall ?n ?m
          (= (Plus ?n (succ ?m))
             (succ (Plus ?n ?m)))))

(define Plus-zero-property
  (forall ?n
    (= (Plus zero ?n) ?n)))

(define Plus-succ-property
  (forall ?n ?m
          (= (Plus (succ ?m) ?n)
             (succ (Plus ?m ?n)))))

(define Plus-Associativity
  (forall ?n ?m ?p
          (= (Plus (Plus ?m ?p) ?n)
             (Plus ?m (Plus ?p ?n)))))
```

C Iterator ranges and valid range

The `range.ath` defines the STL iterator ranges. It contains definition of valid ranges and proofs of some theorems which are used frequently in other proofs. Note that some of these proofs use “proof packaging” methods that are not discussed further in this paper.

"range.ath" 70 ≡

```
(load-file "iterator.ath")

(structure (Range T) (range T T))

(declare valid ((T) ->
                ((Range (Iterator T)))
                 Boolean))
```

```
(define (Valid-Range-Definition name ops)
  (try (Forward-Iterator name ops)
       (let ((- (ops '-))
             (valid (ops 'valid)))
         (match (resolve name 'Valid-Range)
           ('Definition
             (forall ?i ?j
                     (iff (valid (range ?i ?j))
                          (exists ?n (= ?i (- ?j ?n))))))))))

(define (Valid-Range-Lemma1 name ops)
  (let ((- (ops '-))
        (valid (ops 'valid)))
     (match name
       (['Valid-Range 'Lemma1]
        (forall ?n ?j
                (valid (range (- ?j ?n) ?j)))))))


(define (Valid-Range-Lemma1-proof the-theorem instance theorems)
  (the-theorem
   BY (dlet ((axiom (instance 'axioms))
             (op (instance 'ops))
             (- (op '-))
             (valid (op 'valid)))
        (pick-any m j
                  (dbegin
                   (!eq-reflex (- j m))
                   (!egen (exists ?n (= (- j m) (- j ?n)))
                          m)
                   (!specialize-right (axiom ['Valid-Range 'Definition])
                                      [(- j m) j]))))))

(define Range-theorems (cell [Forward-Iterator-theorems]))

(add-theorem
 [Valid-Range-Lemma1
  Valid-Range-Lemma1-proof]
 Range-theorems)
```

```
(define RI
  (Instance Valid-Range-Definition
            (function (op)
              (match op
                ('valid valid)
                ('- -)
                ('++ ++)))))

(assert ((RI 'axioms) ['Valid-Range 'Definition]))

(!theorem-test ['Valid-Range 'Lemma1] RI Range-theorems)


(define nat-property
  (forall ?n
    (if (not (= ?n zero))
        (exists ?m (= ?n (succ ?m))))))

(nat-property
 BY (by-induction-on
     ?n
     (if (not (= ?n zero))
         (exists ?m (= ?n (succ ?m))))
     (zero
      (assume (not (= zero zero))
        ((exists ?m (= zero (succ ?m)))
         BY (!by-contradiction
              (assume (not (exists ?m (= zero (succ ?m))))
                (!absurd
                 ((= zero zero)
                  BY (!eq-reflex zero))
                 (not (= zero zero))))))))
     ((succ m)
      (assume (not (= (succ m) zero))
        (dbegin
         ((= (succ m) (succ m))
          BY (!eq-reflex (succ m)))
         (!egen (exists ?m (= (succ m) (succ ?m)))
                m))))))
```

```
(define (Valid-Range-Lemma2 name ops)
  (let ((- (ops '-))
        (valid (ops 'valid)))
    (match name
      (['Valid-Range 'Lemma2]
       (forall ?i ?j
               (if (and (valid (range ?i ?j))
                        (not (= ?i ?j)))
                   (exists ?n (= ?i (- ?j (succ ?n))))))))))

(define (Valid-Range-Lemma2-proof the-theorem instance theorems)
  (the-theorem
   BY (dlet ((op (instance 'ops))
             (axiom (instance 'axioms))
             (- (op '-))
             (valid (op 'valid)))
        (pick-any
            i j
            (assume-let ((assumptions (and (valid (range i j))
                                           (not (= i j)))))
              (dbegin
               ((valid (range i j))
                BY (!left-and assumptions))
               ((not (= i j))
                BY (!right-and assumptions))
               ((exists ?n (= i (- j ?n)))
                BY (!specialize (axiom ['Valid-Range 'Definition])
                                [i j]))
               (pick-witness n
                 (exists ?n (= i (- j ?n)))
                 (dbegin
                  ((not (= n zero))
                   BY (!by-contradiction
                       (assume (= n zero)
                         (!absurd
                          ((= i j)
                           BY (dbegin
                                (!with left i)
                                (!with right j)
```

```
                               (!reduce left (- j n) (= i (- j n)))
                               (!reduce left (- j zero) (= n zero))
                               (!reduce left j (axiom 'diff-zero-axiom))
                               (!combine left right)))
                          (not (= i j))))))
                  ((exists ?m (= n (succ ?m)))
                   BY (!specialize nat-property [n]))
                  (pick-witness m
                    (exists ?m (= n (succ ?m)))
                    (dbegin
                     ((= i (- j (succ m)))
                      BY (dbegin
                          (!with left i)
                          (!with right (- j (succ m)))
                          (!reduce left (- j n) (= i (- j n)))
                          (!reduce left (- j (succ m)) (= n (succ m)))
                          (!combine left right)))
                     (!egen (exists ?m (= i (- j (succ ?m))))
                            m)))))))))))

(add-theorem
 [Valid-Range-Lemma2
  Valid-Range-Lemma2-proof]
 Range-theorems)

(define VR1
  (Instance Valid-Range-Definition
            (function (op)
              (match op
                ('valid valid)
                ('- -)
                ('++ ++)))))

(!theorem-test ['Valid-Range 'Lemma2] VR1 Range-theorems)

(define (Valid-Range-Reflexive name ops)
  (match name
    (['Valid-Range 'Reflexive]
     (forall ?i
       (valid (range ?i ?i))))))
```

```
(define (Valid-Range-Reflexive-proof the-theorem instance theorems)
  (the-theorem
   BY (dlet ((op (instance 'ops))
             (axiom (instance 'axioms))
             (- (op '-)))
        (pick-any i
          (dbegin
           ((exists ?n (= i (- i ?n)))
            BY (dbegin
                ((= i (- i zero))
                 BY (dbegin
                     (!with left i)
                     (!with right (- i zero))
                     (!reduce right i (axiom 'diff-zero-axiom))
                     (!combine left right)))
                (!egen (exists ?n (= i (- i ?n)))
                       zero)))
           (!specialize-right (axiom ['Valid-Range 'Definition])
                              [i i]))))))

(add-theorem
 [Valid-Range-Reflexive
  Valid-Range-Reflexive-proof]
 Range-theorems)

(!theorem-test ['Valid-Range 'Reflexive]
               VR1
               Range-theorems)

(define (Valid-Range-Step-Property name ops)
  (let ((valid (ops 'valid))
        (++ (ops '++)))
    (try (Valid-Range-Definition name ops)
         (match name
           (['Valid-Range 'Step-Property]
            (forall ?i ?j
                    (if (and (valid (range ?i ?j))
                             (not (= ?i ?j)))
                        (valid (range (++ ?i) ?j)))))))))
```

```
(define (Valid-Range-Step-Property-proof the-theorem instance theorems)
 (the-theorem
  BY
  (dlet ((axiom (instance 'axioms))
         (op (instance 'ops))
         (theorem (method (name) (!theorem name instance theorems)))
         (++ (op '++))
         (- (op '-))
         (valid (op 'valid)))
    (pick-any
        i j
        (assume (and (valid (range i j))
                     (not (= i j)))
          ((valid (range (++ i) j))
           BY (dbegin
                ((exists ?n (= i (- j (succ ?n))))
                 BY (!specialize (!theorem ['Valid-Range 'Lemma2])
                                 [i j]))
                (pick-witness n (exists ?n (= i (- j (succ ?n))))
                              (dlet ((n-prop (= i (- j (succ n)))))
                                (dbegin
                                 ((= (++ i) (- j n))
                                  BY (dbegin
                                      (!with left (++ i))
                                      (!with right (- j n))
                                      (!reduce left (++ (- j (succ n)))
                                               n-prop)
                                      (!reduce left (- j n)
                                               (axiom '++-axiom))
                                      (!combine left right)))
                                 (!egen (exists ?n (= (++ i) (- j ?n)))
                                        n))))
                (!specialize-right (axiom ['Valid-Range 'Definition])
                                   [(++ i) j]))))))))

(add-theorem
 [Valid-Range-Step-Property
  Valid-Range-Step-Property-proof]
 Range-theorems)
```

```
(define (Valid-Nonempty-Range-Definition name ops)
    (try (Valid-Range-Definition name ops)
         (let ((name (resolve name 'Valid-Nonempty-Range))
               (valid (ops 'valid))
               (valid-nonempty-range (ops 'valid-nonempty-range)))
           (match name
             ('Definition
               (forall ?i ?j
                       (iff (valid-nonempty-range ?i ?j)
                            (and (valid (range ?i ?j))
                                 (not (= ?i ?j))))))))))

(declare valid-nonempty-range ((T) -> ((Iterator T) (Iterator T)) Boolean))

(define VNR1
  (Instance Valid-Nonempty-Range-Definition
            (function (op)
              (match op
                ('valid valid)
                ('valid-nonempty-range valid-nonempty-range)
                ('- -)
                ('++ ++)))))

(!theorem-test ['Valid-Range 'Step-Property]
               VNR1
               Range-theorems)
```

The `range-induction.ath` file defines the `range-induction` method, which transforms induction over iterator ranges into induction over the natural numbers (and back-translates the resulting natural number theorem into one directly about iterator ranges).

`"range-induction.ath"` 77a ≡

> ⟨Defining the cases and setting up the range induction 77b⟩
> ⟨General principle of range induction 81⟩
> ⟨User level method for performing range induction 81⟩

⟨Defining the cases and setting up the range induction 77b⟩ ≡

```
(load-file "range.ath")

(define (range-basis-case Q)
  (forall ?i (Q (range ?i ?i))))

(define (range-induction-step Q)
  (forall ?i ?j
          (if (and (not (= ?i ?j))
                   (and (valid (range ?i ?j))
                        (Q (range (++ ?i) ?j))))
              (Q (range ?i ?j)))))

Following is for internal use only

(define (setup-range-induction Q)
  (dlet ((P (function (n) (forall ?j (Q (range (- ?j n) ?j)))))
         (ops (function (op)
                (match op
                  ('valid valid)
                  ('++ ++)
                  ('- -))))
         (Basis-case (P zero))
         (Induction-step (forall ?n (if (P ?n) (P (succ ?n))))))
    (dbegin
     (Basis-case
      BY (pick-any j
           (dbegin
            ((Q (range j j))
             BY (!uspec (range-basis-case Q) j))
            (!sym ((= (range (- j zero) j)
                      (range j j))
                   BY (dbegin
                       (!with left (range (- j zero) j))
                       (!reduce left (range j j)
                                (Forward-Iterator
                                 'diff-zero-axiom
                                 ops)))))
            ((Q (range (- j zero) j))
             BY (!rel-cong-2 (Q (range j j)) [(range (- j zero) j)])))))
     (Induction-step
BY (pick-any n
     (assume (P n)
       (pick-any j
         ((Q (range (- j (succ n)) j))
          BY
          (dbegin
           (!equality (- j (succ n)) (- j (succ n)))
           (!egen (exists ?k
                    (= (- j (succ n))
                       (- j ?k)))
                  (succ n))
           (!specialize-right (Valid-Range-Definition
                               ['Valid-Range 'Definition]
                               ops)
                              [(- j (succ n)) j])
           (!both ((not (= (- j (succ n)) j))
                   BY (!by-contradiction
                       (assume (= (- j (succ n)) j)
                         (dbegin
                          ((= j (- j zero))
                           BY (!sym
                               (!uspec (Forward-Iterator
                                        'diff-zero-axiom
                                        ops) j)))
                          ((= (- j (succ n)) (- j zero))
                           BY (!tran (= (- j (succ n)) j)
                                     (= j (- j zero))))
                          (!absurd ((= (succ n) zero)
                                    BY (!specialize (Forward-Iterator
                                                    'diff-equal-axiom
                                                    ops)
                                                   [j (succ n) zero]))
                                   ((not (= (succ n) zero))
                                    BY (!uspec succ-not-zero-axiom
                                               n)))))))
                  (!both (valid (range (- j (succ n)) j))
                         ((Q (range (++ (- j (succ n))) j))
                          BY (dbegin
                              ((Q (range (- j n) j))
                               BY (!uspec (P n) j))
                               (!sym
                                ((= (range (++ (- j (succ n))) j)
                                    (range (- j n) j))
                                 BY (dbegin
                                     (!with left (range (++ (- j (succ n))) j
                                     (!with right (range (- j n) j))
                                     (!reduce left (range (- j n) j)
                                              (Forward-Iterator
                                               '++-axiom
                                               ops)))))
                               (!rel-cong-2 (Q (range (- j n) j))
                                            [(range (++ (- j (succ n))) j)]))
            (!specialize (range-induction-step Q) [(- j (succ n)) j]))))))))
(by-induction-on
 ?n
 (P ?n)
 (zero
  (!claim Basis-case))
 ((succ n)
  (!mp (!uspec Induction-step n)
       (P n))))
((forall ?i ?j (if (valid (range ?i ?j))
                   (Q (range ?i ?j))))
 BY (pick-any
        i j
        (assume (valid (range i j))
          (dbegin
           ((exists ?n
                  (= i (- j ?n)))
            BY (!specialize
                (Valid-Range-Definition ['Valid-Range 'Definition] ops)
                [i j]))
           (pick-witness n (exists ?n
                             (= i (- j ?n)))
                         (dbegin
                          ((Q (range (- j n) j))
                           BY (!uspec* (forall ?n ?j
                                               (Q (range (- ?j ?n) ?j)))
                                       [n j]))
                          ((= (range (- j n) j)
                                                  (range i j))
                                                BY (dbegin
                                                    (!with left (range (- j n) j))
                                                    (!with right (range i j))
                                                    (!reduce right (range (- j n) j)
                                                             (= i (- j n)))
                                                    (!combine left right)))
                                              (!rel-cong-2
                                               (Q (range (- j n) j))
                                               [(range i j)]))))))))))
```

Used in part 77a.

This is the general principle of range induction, expressed in terms of the arbitrary predicate `Q`.

⟨General principle of range induction 81⟩ ≡

```
(define (abstract-range-induction-theorem Q)
  (assume
      (range-basis-case Q)
    (assume
        (range-induction-step Q)
      (!setup-range-induction Q))))
```

Used in part 77a.

This is the user-level method for performing range induction for a specific predicate `P`. `Q` is a predicate symbol and `P` is a proposition such that `Q-definition`, defined as at the beginning of the method, is in the assumption base, and so are `(range-basis-case P)` and `(range-induction-step P)`. The result of the proof carried out in the method is the theorem:

```
(forall ?i ?j
   (if (valid (range ?i ?j))
       (P (range ?i ?j))))
```

⟨User level method for performing range induction 81⟩ ≡

```
(define (range-induction Q P)
  (dlet ((Q-definition
          (forall ?i ?j
                  (iff (Q (range ?i ?j))
                       (P (range ?i ?j))))))
    (dbegin
     (!mp (!mp (!abstract-range-induction-theorem Q)
               (pick-any i
                 (!mp (!right-iff (!uspec* Q-definition [i i]))
                      (!uspec (range-basis-case P) i))))
          (pick-any
              i j
              (assume-let ((assumptions
                            (and (not (= i j))
                                 (and (valid (range i j))
                                      (Q (range (++ i) j))))))
                (dbegin
                 (!left-and (!right-and assumptions))
                 (!left-and assumptions)
                 (!right-and (!right-and assumptions))
                 (!both (not (= i j))
                        (!both (valid (range i j))
                               (!specialize Q-definition [(++ i) j])))
                 (!specialize (range-induction-step P) [i j])
                 (!specialize-right Q-definition [i j])))))
     ((forall ?i ?j
              (if (valid (range ?i ?j))
                  (P (range ?i ?j))))
      BY
      (pick-any
          i j
          (assume (valid (range i j))
            (dbegin
             (!specialize (forall ?i ?j
                                  (if (valid (range ?i ?j))
                                      (Q (range ?i ?j))))
                          [i j])
             (!specialize Q-definition [i j]))))))))
```

Used in part 77a.