VIETNAM NATIONAL UNIVERSITY OF HO CHI MINH CITY
THE INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

FORMAL VERIFICATION OF ETHEREUM SMART CONTRACTS WITH A PROOF ASSISTANT

By
Huỳnh Trần Khanh

A thesis submitted to the School of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
Bachelor of Computer Science

Ho Chi Minh City, Vietnam
May 2025

FORMAL VERIFICATION OF ETHEREUM SMART CONTRACTS WITH A PROOF ASSISTANT

APPROVED BY:

Committee name here

Committee name here

Committee name here

Committee name here

Committee name here

THESIS COMMITTEE

Acknowledgments

I wish to express my sincere gratitude to the individuals and communities who have supported me throughout the journey of writing this thesis.

First and foremost, I would like to extend my heartfelt thanks to my maternal grandparents. Their daily reminders were a constant, gentle encouragement that kept me focused on the task of completing this manuscript.

My exploration into the field of formal verification would not have been possible without the vibrant and knowledgeable Coq and Lean communities. I am deeply grateful for the resources, discussions, and guidance they provided, which were instrumental in my learning process.

I am also indebted to my employer in the People's Republic of China for affording me the invaluable opportunity to engage with challenging, real-world problems in formal verification. This practical experience significantly shaped the perspective and content of this work.

I would like to thank my thesis advisor, Dr. Trần Thanh Tùng, for his input and support. Specifically, I would like to thank him for teaching me how to explain my work in the most accessible manner.

The completion of this thesis is a reflection of the support and opportunities provided by many, and I am truly thankful.

Table of Contents

Abstract

Smart contracts deployed on blockchains like Ethereum automate critical agreements but demand high levels of security and correctness assurance. Current practices predominantly rely on code audits, which, while valuable, are fundamentally limited as they focus on finding existing flaws rather than proving their absence. Audits may miss complex bugs and necessitate trusting the auditors, conflicting with the trustless ethos of blockchain technology. This thesis addresses these limitations by proposing and developing a framework for the formal verification of Ethereum smart contracts using the Coq proof assistant. We introduce a novel, JavaScript-inspired programming language specifically designed to facilitate verification. A core contribution is a compiler that translates contracts written in this language into both formal specifications in Coq, suitable for rigorous proof development, and executable Solidity code for deployment. We establish the necessary foundational semantic definitions within Coq, leveraging concepts like algebraic effects and monads to model contract behavior accurately. The practical applicability of the framework is demonstrated through the implementation and verification of two non-trivial smart contracts. For these examples, we manually construct detailed, machine-checked proofs in Coq, formally guaranteeing critical properties such as the absence of vulnerabilities, integer overflow/underflow errors, out-of-bounds access, and the preservation of essential data structure invariants. While proof construction remains manual, this work demonstrates the feasibility of using a dedicated language and compiler infrastructure coupled with a proof assistant to achieve a significantly higher degree of verifiable assurance for smart contracts than is possible through auditing alone, thereby strengthening their trustworthiness.

Chapter 1
INTRODUCTION

1.1 Motivation

Smart contracts are revolutionizing various domains by enabling self-executing agreements on blockchains like Ethereum. The integrity and security of these contracts are paramount, as flaws can lead to significant financial losses and undermine trust in the underlying systems. Currently, the primary method for ensuring smart contract quality is through code audits. While valuable, audits fundamentally focus on finding existing flaws rather than proving their absence. This approach has inherent limitations: auditors might miss subtle, deeply hidden bugs due to the complexity of exploring all possible execution paths, or they might be incompetent or even malicious. Furthermore, relying on auditors introduces an element of trust into a system designed to be trustless, as users typically lack the resources or expertise to perform independent audits and must therefore trust the auditors' reports.

Formal verification offers a mathematically rigorous alternative. Instead of searching for bugs, it aims to construct a mathematical proof that the software adheres to a precise specification of its desired properties. This process involves describing the software and its required behaviors mathematically and then proving that the software implementation satisfies these requirements. While creating these proofs is labor-intensive and requires specialized expertise, the resulting proofs can be automatically checked for logical correctness by computer programs (proof assistants like Coq). This shifts the trust requirement from fallible human auditors to well-vetted proof-checking software and the clarity of the mathematical specifications. By enabling verifiable proofs of correctness, formal verification holds the potential to significantly enhance the reliability and trustworthiness of smart contracts, aligning more closely with the foundational "trustless" principles of blockchain technology.

1.2 Problem Statement

The core problem addressed by this thesis is the inadequacy of current methodologies, primarily code auditing, in providing sufficient guarantees about the security and correctness of Ethereum smart contracts. Auditing, while beneficial for identifying common vulnerabilities, cannot definitively prove the absence of all bugs, especially complex or novel ones. This leaves a residual risk of exploitation. Moreover, the reliance on human auditors introduces points of failure related to thoroughness, competence, and integrity, fundamentally conflicting with the blockchain ethos of minimizing reliance on trusted third parties. End-users interacting with smart contracts lack practical means to independently verify the contract's safety beyond relying on these potentially flawed audit reports. There is a need for a more robust approach that allows for stronger, verifiable assurances of smart contract behavior.

1.3 Scope

This thesis focuses on exploring the application of formal verification to Ethereum smart contracts using the Coq proof assistant. The scope encompasses the following:

1. **Development of a Custom Programming Language:** Designing and specifying a novel programming language with syntax inspired by JavaScript, tailored for writing smart contracts amenable to formal verification.

2. **Compiler Implementation:** Building a compiler that takes smart contracts written in the custom language and generates two outputs:

   - Coq code representing the contract's semantics for formal verification.
   - Solidity code suitable for deployment on the Ethereum blockchain.

3. **Foundational Coq Definitions:** Establishing the necessary theoretical framework within Coq, including definitions for algebraic effects (handling blockchain state, array manipulation, local variables, loop control), an action monad, and effect handlers to model smart contract execution.

4. **Example Smart Contracts:** Implementing two smart contracts of non-trivial complexity in the custom language to serve as case studies.

5. **Manual Proof Construction:** Manually developing formal proofs of correctness for the example smart contracts within Coq. These proofs will target critical properties such as resistance to reentrancy attacks, prevention of integer overflow/underflow, bounds checking, preservation of data structure invariants, and correctness of state transitions.

The scope *excludes* the automatic generation of these formal proofs; proofs must be constructed manually by an expert, although their correctness is checked automatically by Coq. The verification process is limited to contracts written in the newly developed custom language.

1.4 Objectives

The primary objectives of this thesis are:

1. **Design and Implement a Verifiable Smart Contract Language:** To create a high-level programming language specifically designed for developing Ethereum smart contracts while facilitating formal verification.

2. **Develop a Dual-Target Compiler:** To implement a compiler that translates source code from the custom language into both verifiable Coq definitions and deployable Solidity code.

3. **Establish Formal Semantics in Coq:** To define the necessary foundational libraries and theories (algebraic effects, monads, handlers) in Coq to accurately model the execution semantics of the smart contracts.

4. **Demonstrate Practical Verification:** To implement non-trivial example smart contracts in the custom language and construct rigorous, machine-checked proofs of their correctness and security properties using Coq.

5. **Showcase Feasibility:** To demonstrate that this approach provides a viable path towards achieving higher assurance in smart contract correctness compared to traditional auditing, by enabling the verification of deep functional and security properties.

The source code of this project and additional documentation are available at `https://github.com/huynhtrankhanh/CoqCP`.

Chapter 2
RELATED WORK

The challenge of ensuring smart contract correctness and security has spurred the development of various analysis and verification techniques. These approaches range from dynamic testing and lightweight static analysis to more rigorous formal verification methods. While each offers distinct advantages, they also possess limitations that motivate the interactive theorem proving approach explored in this thesis.

2.0.1 Dynamic Analysis and Fuzzing

Dynamic analysis techniques execute smart contracts with concrete or symbolic inputs to uncover vulnerabilities. **Fuzzing**, a popular dynamic testing method, involves bombarding a contract with a large volume of random or semi-random inputs to identify unexpected behaviors or crashes. **Echidna** [1] is a prominent *property-based fuzzer* for Ethereum smart contracts. It allows developers to define properties that should hold true and then uses instrumentation and coverage guidance to generate inputs that attempt to violate these properties.

Another category of dynamic analysis includes tools based on **symbolic** or **concolic execution**. **MythX** [2], and its open-source predecessor Mythril [3], analyze smart contracts by executing them with symbolic values. This allows them to explore multiple execution paths simultaneously and use an SMT (Satisfiability Modulo Theories) solver to generate concrete inputs that trigger specific paths, aiming to identify vulnerabilities like reentrancy or arithmetic overflows.

While these dynamic techniques are effective at finding concrete bugs and can simulate *true runtime behavior*, they fundamentally struggle with the vastness of the *state space* inherent in smart contracts. Consequently, they cannot guarantee the exploration of all possible execution scenarios and thus cannot prove the absence of bugs, only their presence when found.

2.0.2 Static Analysis and Linters

Static analysis tools examine smart contract code without executing it, looking for patterns indicative of potential vulnerabilities. **Linters** such as **Solhint** [4] and the now-archived **Ethlint** (formerly Solium) [5] perform surface-level analysis of the Abstract Syntax Tree (AST) of Solidity code. They are adept at identifying common anti-patterns, style guide violations, and known simple security pitfalls. Tools like **EtherSolve** also perform static analysis, often focusing on control-flow graph properties.

These tools are generally fast and easy to integrate into development workflows. However, their reliance on *syntactic patterns* and *surface analysis* means they often lack deep semantic understanding. They are typically unable to detect complex vulnerabilities that depend on runtime states or intricate inter-contract interactions, and they may produce a significant number of false positives or negatives. They can indicate where a bug is *likely* but cannot confirm its presence or absence under all runtime conditions.

2.0.3 Automated Formal Verification with Constraint Solvers

To achieve stronger guarantees, various **automated formal verification** techniques have been applied to smart contracts. These methods often translate contract semantics and desired properties into logical formulas and employ automated reasoning engines, such as **SMT solvers** or **HornSAT solvers**, to check for violations or prove adherence.

The **Solidity SMTChecker** [6] is an SMT-based bounded model checker built into the Solidity compiler. It attempts to formally prove that certain assertions hold or find counterexamples for properties like unreachable code or trivial contradictions. **Certora Prover** [7] is a commercial tool that allows users to specify properties in its own language (CVL - Certora Verification Language) and uses advanced static analysis and SMT techniques to verify them against EVM bytecode. Other research efforts and tools like **KEVM** [8, 9], which provides a complete formal semantics of the EVM in the K framework, also enable formal verification. KEVM allows for rigorous, executable semantics that can be used for model checking and theorem proving about contract behavior at the bytecode level. Tools like `solc-verify` often provide a harness or interface to these underlying SMT-based verification engines.

While these automated approaches offer more rigor than static or dynamic analysis alone, they face inherent limitations. The underlying constraint-solving problems are often undecidable or computationally expensive for complex contracts or properties. As such, solvers may *timeout* or run out of memory. More critically, these systems are often *limited by decidable theories* and may lack the *expressivity* needed to state and prove more general or domain-specific correctness properties. The reasoning process is largely a black box, offering limited avenues for *user-guided input* when a proof fails or when a particularly nuanced property needs to be formalized. This dependency on the capabilities of the automated solver can make these systems feel *finicky or stiff* when dealing with properties outside their optimized scope.

2.0.4 Interactive Theorem Proving

The limitations in completeness of dynamic testing, the superficiality of static linters, and the expressiveness or decidability constraints of automated formal verification tools highlight the need for methods that can offer higher assurance for critical smart contract properties. **Interactive Theorem Proving (ITP)** provides such an alternative. Systems like **Coq** [10], Isabelle/HOL, or Lean allow for the expression of highly complex properties and software semantics within rich logical frameworks.

In ITP, a human expert guides the construction of a formal proof, which is then meticulously checked for logical soundness by the proof assistant. This approach offers unparalleled *expressiveness*, allowing for the formalization and verification of deep functional correctness properties, intricate invariants, and complex security policies that are often beyond the reach of fully automated tools. The process, while *labor-intensive and requiring specialized expertise*, results in a machine-checked mathematical argument for the software's correctness with respect to its specification.

The work presented in this thesis builds upon the strengths of ITP. By developing a custom programming language designed for verifiability, a compiler that translates this language into Coq, and by manually constructing proofs for non-trivial smart contracts, we aim to demonstrate that this expert-driven approach can provide the highest level of assurance for critical smart contract behaviors. This aligns with the goal of shifting trust from potentially fallible auditors or opaque automated tools to the transparent,

mathematically rigorous foundations of a proof assistant like Coq and the well-defined semantics of the verification framework.

Chapter 3
METHODOLOGY

This work introduces a formal verification framework centered around a custom imperative language. The primary goal is to enable rigorous, machine-checked proofs of correctness for programs intended for blockchain execution, specifically by compilation to Solidity. This approach aims to enhance the reliability and security of smart contracts. The methodology encompasses the design of the custom language, its compilation to the Coq proof assistant for formal verification, and its subsequent compilation to executable Solidity code for deployment.

3.0.1 Language Design for Verifiable Smart Contracts

The foundation of the framework is a custom imperative language that utilizes JavaScript-like syntax. This language has been intentionally designed with a restricted feature set, specifically selecting constructs that are amenable to formal representation within a proof assistant and are well-suited for the deterministic and resource-constrained environment of blockchains like Ethereum. This design philosophy aims to reduce the inherent complexities often encountered when attempting to formally verify programs written in more feature-rich, general-purpose programming languages.

Why the custom language?

The custom language is created to simplify implementation effort. Using Solidity itself as the input language means the framework has to understand every Solidity feature in existence, and also means that the framework needs to include a modified fork of the Solidity compiler.

Not all Solidity features are simple to model in Coq. Solidity has a very prominent feature that is especially problematic to formalize: its Application Binary Interface. In the Application Binary Interface, functions are identified by hashing the function signature and taking the first 32 bits. As 32 bits is too short, collisions between functions often occur and can't be easily ignored. To model the possibility of collisions, an implementation of the hash function has to be modeled in Coq, and this increases the verification complexity of programs.

In order for a language to be expressive, it should support most of the Ethereum Virtual Machine features, as all smart contract languages have to eventually get compiled to Ethereum Virtual Machine bytecode. According to this yardstick, the custom language is slightly less expressive than Solidity but still covers most features needed to develop a smart contract. The three main features that the language doesn't support are transient storage, computing SHA-3 and retrieving low-level block information. These features can always be added later as the framework develops. Contracts written in the custom language can interact with all existing smart contracts in the Ethereum ecosystem, as all Ethereum contracts take in binary data and return binary data, regardless of the language they are written in.

Key features

Key features of this internal imperative language include:

- **Variables**: Support for local variables of primitive types such as integers (`int8` through `int64`, `int256`), booleans (`bool`), and blockchain-specific addresses (`address`).
- **Global Arrays**: Declaration of global arrays with elements being tuples of primitive types (e.g., `array([int32], 100)` or `array([int8, int64], 3)`). These map to persistent storage in Solidity.
- **Arithmetic and Boolean Operations**: Standard arithmetic (`+`, `-`, `*`, `divide`, `sDivide`) and bitwise operators (`|`,`^`, `&`, `~`), along with boolean logic (`||`, `&&`, `!`) and comparison operations (`==`, `!=`, `less`, `sLess`). Arithmetic is generally unsigned with wraparound, except for specific signed division commands.
- **Control Flow**: Conditional execution using `if/else` statements (with mandatory curly braces) and loops implemented via a `range` command, which iterates a counter from 0 to a specified end value. Loop control statements `"break"` and `"continue"` are supported as string literals.
- **Blockchain-Specific Commands**: Dedicated instructions for interacting with the blockchain environment, such as retrieving the sender's address (`getSender()`), the transaction value (`getMoney()`), transferring funds (`donate()`), invoking other smart contracts (`invoke()`), and managing calldata/return data via a communication array (`communicationSize()`, `retrieve(index)`, `store(index, value)`).

We outline the completeness and sufficiency of the custom programming language's features, as outlined in Section 3.0.1. We will clarify the scope of its components and justify its design choices, particularly by comparing it to Solidity to demonstrate its capability in expressing common smart contract logic amenable to formal verification.

3.0.2 Feature Completeness and Sufficiency

The design philosophy of the custom language emphasizes a deliberate restriction of features to those that are both essential for a broad class of smart contracts and conducive to formal verification. As stated in Section 3.0.1, the language is "intentionally designed with a restricted feature set...to reduce the inherent complexities often encountered when attempting to formally verify programs written in more feature-rich, general-purpose programming languages."

While the custom language is, by design, slightly less expressive than Solidity in its entirety, it covers the majority of features required for developing typical smart contracts. The primary features not supported, as noted, are transient storage (EIP-1153), direct computation of SHA-3 hashes within the language, and retrieval of certain low-level block information (e.g., `block.timestamp`, `block.difficulty`).

The sufficiency of the current feature set is justified as follows:

1. **Core Logic Representation:** The existing features (variables, arrays, operations, control flow) are fundamental programming constructs, allowing for the expression of complex algorithmic logic.
2. **State Management:** Global arrays provide a robust mechanism for persistent state, which is central to smart contract functionality.

3. **Blockchain Interaction:** The blockchain-specific commands provide the necessary primitives to interact with the Ethereum environment (users, other contracts, Ether transfers) in a way that mirrors standard smart contract capabilities.

4. **Focus on Verifiability:** The omission of features like a complex ABI or unbounded loops and recursion simplifies the formal semantics and the verification process significantly. For instance, avoiding a direct ABI implementation sidesteps the complexities of function signature hashing and potential collisions, as detailed in Section 3.0.1.

5. **Extensibility:** The framework is designed such that these currently unsupported features could potentially be added in the future if deemed critical and if methods for their formalization are developed.

The current feature set is therefore deemed sufficient for the primary goal of this work: to enable formal verification of a significant range of smart contract patterns and logic, demonstrating the viability of this approach.

3.0.3 Comparison with Solidity

To further illustrate the capabilities and design choices of the custom language, a comparison with Solidity is instructive.

Similarities / Functional Equivalence in Core Areas

- **Data Types:** Both languages support fundamental data types like integers of various sizes (e.g., `uint256` in Solidity, `int256` in the custom language), booleans, and addresses.
- **Persistent Storage:** The custom language's global arrays serve the same purpose as Solidity's state variables. While Solidity offers more complex data structures like mappings and dynamic arrays directly, the custom language's tuple-based arrays can model many common storage patterns, and the fixed-size nature aids verification.
- **Operations:** Standard arithmetic, logical, and comparison operations are present in both, forming the basis of contract logic.
- **Control Flow:** `if/else` statements and looping constructs (Solidity's `for`/`while` vs. custom language's `range`) allow for similar conditional execution and iteration patterns.
- **Blockchain Primitives:**
  - Accessing transaction sender: `msg.sender` in Solidity is equivalent to `getSender()`.
  - Accessing transaction value: `msg.value` in Solidity is equivalent to `getMoney()`.
  - External Calls: Solidity's `address.call()`, `address.delegatecall()`, `address.staticcall()` are conceptually mirrored by the `invoke()` command, which handles external interactions. The custom language focuses on a simplified external call mechanism suitable for verification.

  - Ether Transfer: Solidity's `payable.transfer()` or `payable.send()` (and now `address.call{value: ...}("")`) are functionally covered by `donate()` for simple transfers and `invoke()` when sending value with a call.

- **Interoperability:** As stated in Section 3.0.1, "Contracts written in the custom language can interact with all existing smart contracts in the Ethereum ecosystem," because, at the EVM level, interactions are based on binary data (calldata and return data), which the custom language's communication array mechanism handles.

Key Differences and Design Justifications

- **Application Binary Interface (ABI):** Solidity has a complex ABI specification involving function signature hashing to create function selectors. The custom language deliberately avoids this complexity to simplify formalization. Instead, it likely relies on a more direct or simplified dispatch mechanism when compiling to Coq and Solidity, making the control flow easier to reason about formally.

- **Unsupported High-Level Solidity Features:**

  - **Mappings:** Solidity's `mapping` type is a powerful key-value store. While not directly available, similar functionality can often be simulated using arrays and careful indexing, or by designing contracts differently. The omission simplifies state modeling for verification.
  - **Dynamic Arrays (in memory/storage):** Solidity supports dynamically-sized arrays. The custom language uses fixed-size global arrays for persistent storage, which simplifies proofs about bounds and state size.
  - **Modifiers, Events, Inheritance:** These are higher-level Solidity constructs that aid in code organization and off-chain communication (events). The custom language focuses on core executable logic for verification, and such features would add significant complexity to the formal model.
  - **Error Handling (`require`, `assert`, `revert`):** Solidity has explicit error-handling functions. The custom language models errors (like division by zero or out-of-bounds access) through the `Trap` effect (Section 3.0.4), which is a formal way to represent exceptional control flow. The Solidity compiler for the custom language then translates these traps into appropriate revert mechanisms (Section 4.7).
  - **Inline Assembly:** Solidity allows inline Yul assembly, offering low-level control. This is antithetical to the goals of a high-level, verifiable language and is thus not supported.
  - **Transient Storage, SHA-3, Low-Level Block Info:** These features are currently not available because of implementation effort, but they can be added later as needed. Transient storage and low-level block info are not commonly used features in smart contracts. The user can reimplement the SHA-3 algorithm themselves in the absence of a way to invoke the SHA-3 opcode.

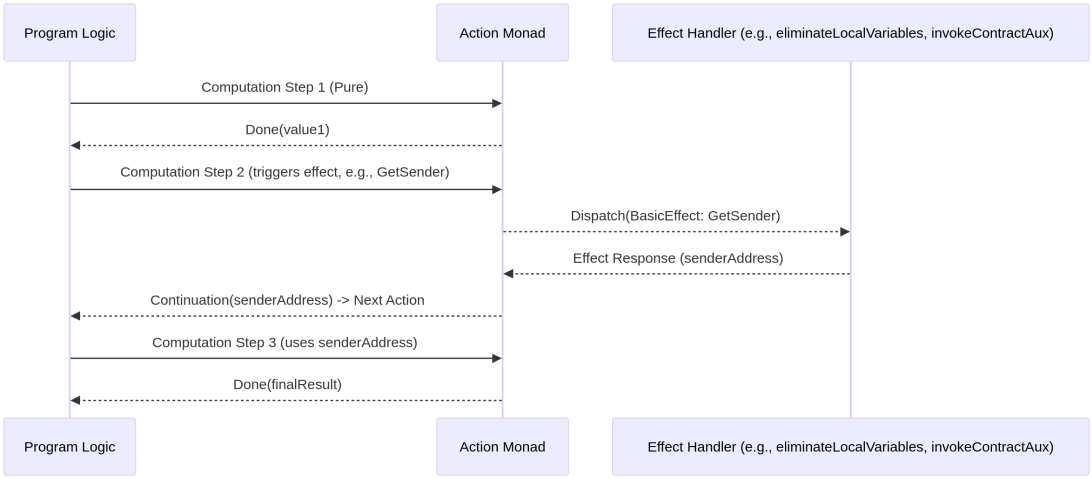


Figure 3.1: Action monad

Achieving Functional Equivalence for Verifiable Logic

The custom language is not intended to be a line-for-line replacement for Solidity across all its features. Instead, it aims to provide a computationally complete that is sufficiently expressive for a wide range of common smart contract use cases, while being significantly more amenable to formal proof. The "equivalence" sought is in the ability to express the fundamental computational logic and stateful behavior of smart contracts, rather than in replicating every syntactic sugar or advanced feature of Solidity.

The fact that contracts written in this custom language can be compiled into executable Solidity code (Section 3.0.5) further underscores its capability to express logic that is deployable and functional on the Ethereum blockchain. This compilation step acts as a bridge, demonstrating that the abstractions chosen for verifiability can be mapped back to a widely-used execution environment.

3.0.4 Formal Verification via Compilation to Coq

A critical aspect of the methodology is the compilation of the custom language into the Coq proof assistant. This translation facilitates the formal specification and machine-checked verification of program properties.

Monadic Framework for Effects

To formally manage side effects—such as state modifications, external calls, or error handling—the compiled Coq code employs a monadic framework. An `Inductive` type, described as `Action (effectType : Type) (effectResponse : effectType`

`-> Type) (returnType : Type)`, is defined. This monad, based on the principles of algebraic effects, models a program that can perform a sequence of operations with potential side effects and ultimately yield a result of `returnType`. The core of the monad consists of two constructors:

- `Done (returnValue : returnType)`: Represents a computation that has finished and produced a `returnValue`.
- `Dispatch (effect : effectType) (continuation : effectResponse effect -> Action ...)`: Represents a computation that performs an `effect` of `effectType`. The `continuation` function then takes the `effectResponse` (the result of handling the effect) and produces the next `Action`.

A monadic `bind` operator (denoted as `x >>= f`) is defined to chain these actions, allowing the output of one action to be passed as input to the next, while correctly sequencing the effects.

Hierarchy of Effects for Structured Reasoning

The framework organizes effects into a hierarchy, allowing for a layered and modular approach to their definition and handling. This is particularly relevant for distinguishing between general computational effects and blockchain-specific interactions.

- `BasicEffect`: This is the most fundamental layer of effects. For the Solidity compilation target, the relevant `BasicEffect`s include:
  - `Trap`: Triggered by runtime errors (e.g., division by zero, signed integer overflow). As execution cannot safely continue, its Coq return type is `False` (the bottom type).
  - `Donate (money : Z) (address : list Z)`: Transfers `money` (an integer representing value) to a target `address` without invoking any code at that address. (Here, Z represents Coq's integers).
  - `Invoke (money : Z) (address : list Z) (array : list Z)`: Calls a smart contract at the given `address`, sending `money` and an `array` (calldata). It returns the `list Z` representing the return data from the invoked contract.
  - `GetSender`: Returns the `list Z` representing the Ethereum address of the entity that initiated the current contract execution.
  - `GetMoney`: Returns a Z (integer) representing the amount of cryptocurrency transferred with the contract invocation (similar to `msg.value` in Solidity).
  - `GetCommunicationSize`: Returns a Z indicating the size of the communication array (calldata) passed to the contract.
  - `ReadByte (index : Z)`: Reads a byte (represented as Z) from the communication array at the specified `index`.
  - `SetByte (index value : Z)`: Writes a `value` (byte) to the communication array at the given `index`.

  The execution model when compiled to Solidity stipulates that upon invocation, the transaction's calldata is copied into a "communication array." The contract can then use `ReadByte` to access this data and `SetByte` to modify it; the final state of this array is returned as the contract's return data.

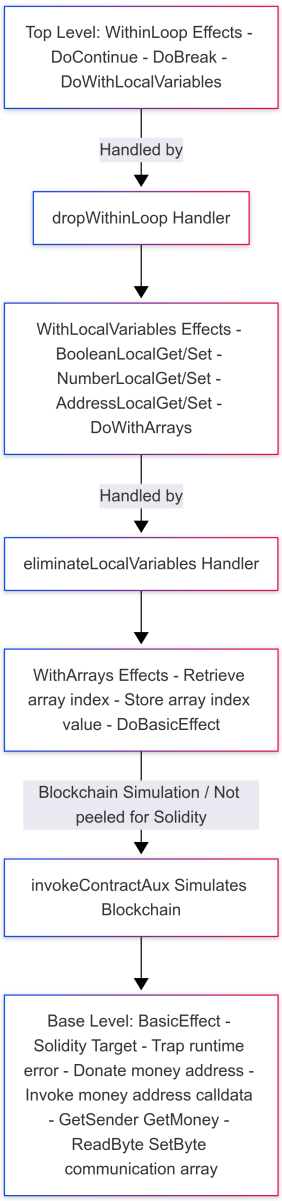


Figure 3.2: Hierarchy of effects

- **`WithArrays`**: This effect type extends `BasicEffect` to incorporate operations on global arrays.
  - `DoBasicEffect (effect : BasicEffect)`: Embeds a basic effect.
  - `Retrieve (arrayName : arrayIndex) (index : Z)`: Retrieves a value from the array `arrayName` at `index`.
  - `Store (arrayName : arrayIndex) (index : Z) (value : arrayType arrayName)`: Stores a `value` into `arrayName` at `index`.

  When compiling to Solidity, arrays declared in the program's environment are mapped to persistent storage variables within the smart contract. Thus, the `WithArrays` layer is an integral part of the contract's stateful behavior and is not "peeled away" by a handler in the same way it might be for a stateless C++ target. The compiler generates an `arrayIndex` inductive type (with constructors for each declared array) and an `arrayType` function defining the type of elements in each array. The `translateArrays` Coq function is used to map array accesses when calling procedures in external modules.

- **`WithLocalVariables`**: This layer builds upon `WithArrays` to manage procedure-local variables.
  - `DoWithArrays (effect : WithArrays arrayIndex arrayType)`: Embeds a `WithArrays` effect.
  - `BooleanLocalGet/Set`, `NumberLocalGet/Set`, `AddressLocalGet/Set`: Effects for accessing and modifying local variables of boolean, number (integer), or address types.

  The compiler generates a `variableIndex` type for the local variables of each procedure. A Coq `Fixpoint` function `eliminateLocalVariables` serves as an effect handler. It takes the initial state of local variables (as maps from `variableIndex` to their respective types) and an `Action` using `WithLocalVariables` effects, and it returns an `Action` with only `WithArrays` effects, effectively managing local variable state internally.

- **`WithinLoop`**: This effect type, layered on `WithLocalVariables`, introduces loop-specific control flow effects:
  - `DoWithLocalVariables (effect : WithLocalVariables ...)`: Embeds a `WithLocalVariables` effect.
  - `DoContinue`: Terminates the current iteration of a loop and proceeds to the next.
  - `DoBreak`: Exits the loop entirely.

  A `LoopOutcome` type (with constructors `KeepGoing` and `Stop`) is used by the `dropWithinLoop` handler. This handler takes an `Action (WithinLoop ...)` (representing the loop body) and returns an `Action (WithLocalVariables ...)` that yields a `LoopOutcome`. The `loop Fixpoint` function in Coq uses this outcome to decide whether to continue iterating or terminate. The compiler automatically inserts `liftToWithinLoop` calls to adapt actions for the loop context and wraps loop bodies with `dropWithinLoop`.

Effect Handling and Program Structure in Coq

When a procedure from the custom language is compiled to Coq, it becomes a Coq function. This function typically takes the initial states of its local variables and returns an `Action (WithArrays arrayIndex arrayType) withArraysReturnValue ()`. This is because the `eliminateLocalVariables` handler is immediately applied to the procedure body, which initially produces an `Action (WithLocalVariables ...)` type. Effects from lower layers (like `BasicEffect`) must be appropriately lifted (e.g., a `BasicEffect` is lifted to `WithArrays`, which is then lifted to `WithLocalVariables`) to be used within this structure. To simplify this, helper Coq functions are defined for each effect at every layer, constructing the necessary `Dispatch` and `Done` calls, and returning an `Action (WithLocalVariables ...)` which can then be easily chained using the monadic `bind`.

Automated Reasoning and Blockchain Simulation in Coq

The framework is equipped with several rewrite rules in Coq to assist users in reasoning about their programs and to automate parts of the proof process. Many of these rules can be applied using Coq's `autorewrite` tactic. Examples include:

- `pushDispatch` rules: These rules are designed to move `Dispatch` effects (specifically `DoWithArrays` effects) from inside an `eliminateLocalVariables` call to the outside, so they can be handled by effect handlers lower in the hierarchy.
- Local variable rules: For instance, `eliminateLocalVariables ... (Dispatch ... (NumberLocalGet ...name) (fun x => Done ... x)) >>= continuation` rewrites to `eliminateLocalVariables ... (continuation (numbers name))`, effectively substituting the variable's value. Similar rules exist for setting variables using the `update` helper function.
- Loop manipulation rules: These include rules for unrolling loop iterations (`loop_S`) and for simplifying `dropWithinLoop` when encountering `DoContinue` (rewrites to `Done ... KeepGoing`), `DoBreak` (rewrites to `Done ... Stop`), or an empty loop body.

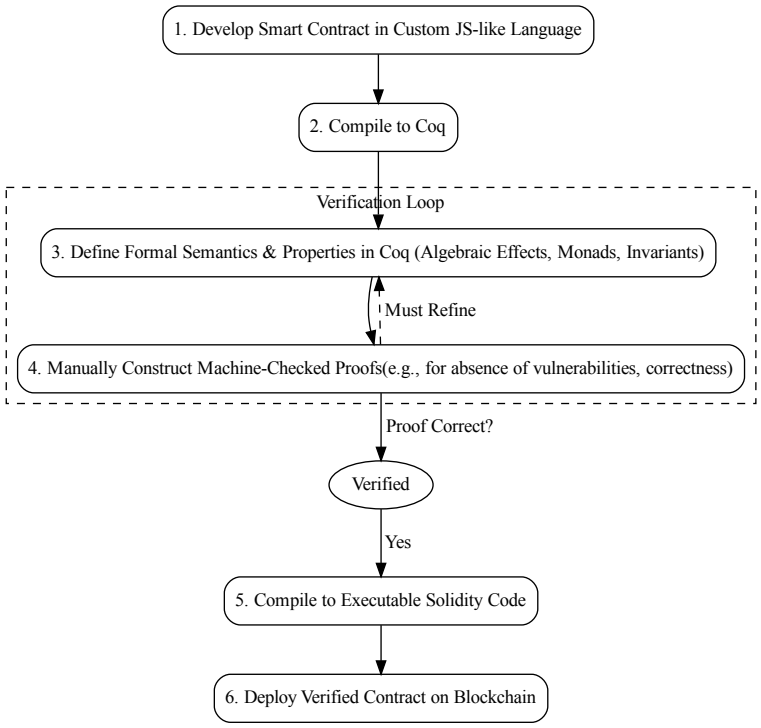
Crucially for Solidity, the lowest-level effect handler in Coq is a recursive function named `invokeContractAux`. This function simulates the behavior of a blockchain environment. It takes parameters such as sender, target address, money, current blockchain state, the communication data (calldata), and importantly, a `fuel` parameter (a natural number). This `fuel` parameter constrains the number of recursive calls (e.g., contract-to-contract invocations), which is essential for ensuring termination of proofs involving potentially recursive smart contract calls and mimics the gas mechanism in Ethereum. The definitions for each case of this recursive `invokeContractAux` function (handling `Done` or dispatching various `BasicEffects` under `WithArrays`) are added to an `autorewrite` hint database (e.g., `advance_program`), allowing Coq to automatically simplify execution steps during proofs.

3.0.5 Compilation to Executable Solidity Code

Following formal verification in Coq (or as a direct compilation step), the code written in the custom imperative language is compiled into executable Solidity code for deployment on Ethereum-compatible blockchains.

- **State Management**: Global arrays declared in the `environment` block of the custom language are translated into state variables in the generated Solidity contract. If array elements are tuples, they are typically represented as Solidity `struct`s, and the array itself becomes a Solidity array of these structs.
- **Blockchain Interactions**:
  - `BasicEffect`s like `Donate`, `Invoke`, `GetSender`, and `GetMoney` are mapped to their corresponding Solidity constructs. For example, `GetSender` becomes `msg.sender`, `GetMoney` becomes `msg.value`, `Donate` uses `selfdestruct` with a throwaway smart contract, and `Invoke` translates to an external contract call (e.g., `address.call{value: money}(calldata)`).
  - The communication array mechanism (`communicationSize()`, `retrieve(index)` for reading calldata bytes, `store(index, value)` for writing to the return data array) is implemented using `msg.data` for incoming data and by preparing a `bytes memory` array that is returned at the end of the execution. The sample Solidity output shows helper functions like `communicationGet` and `communicationSet` for these purposes.
- **Control Flow and Operations**: The imperative language's control flow structures (loops, conditionals) and operations are translated into their Solidity equivalents. Division operations, for example, are translated to Solidity division, with checks for division by zero (and signed overflow for `sDivide`) that revert the transaction, mirroring the `Trap` effect.

3.0.6 Workflow Summary



The intended development and verification workflow using this framework is as follows:

1. **Development**: Smart contract logic is written in the custom, JavaScript-like imperative language.

2. **Compilation to Coq**: The source code is compiled into its formal Coq representation, structured around the `Action` monad and the hierarchy of effects.

3. **Formal Verification**: Within the Coq environment, the developer specifies desired correctness properties (e.g., invariants, pre/post-conditions) for the smart contract. Proofs are then constructed, often interactively but aided by the `autorewrite` tactic and the specialized rewrite rules, including those for the `invokeContractAux` blockchain simulation.

4. **Compilation to Solidity**: When the contract is deemed ready for deployment, or when running the contract in a real execution environment is desired, the contract can be compiled to Solidity.

Chapter 4
IMPLEMENTATION

The framework described in the previous section has been implemented in Coq and TypeScript.

4.1 Outline and Justification of Implementation Strategies

The implementation choices detailed in this chapter—spanning parsing, program analysis, and code generation for both Coq and Solidity—were guided by principles of pragmatism, correctness, and the specific needs of formal verification. This section elaborates on the rationale behind these decisions, highlighting how they contribute to a robust and effective framework.

4.1.1 Parsing and Abstract Syntax Tree (AST) Design

The parsing strategy was designed to efficiently convert the custom, JavaScript-like source code into a structured representation suitable for subsequent processing.

- The choice of **TypeScript** for the parser's implementation offers the benefits of static typing, which helps in catching errors early during development, and robust tooling, common in modern compiler construction.

- Designing the custom language as a **subset of JavaScript syntax** was a pragmatic decision. It allowed the project to leverage `acorn`, an existing, well-tested JavaScript parser. This significantly reduced the development effort that would have been required to build a parser from scratch, enabling a greater focus on the novel aspects of the project, such as the verification framework and the compilation logic.

- The transformation from the generic **ESTree AST (produced by `acorn`) to a custom `CoqCPAST`** is a standard compiler design pattern. The `CoqCPAST` is tailored specifically to the semantics of the custom language and the requirements of the Coq and Solidity code generators. This custom AST provides a more precise and convenient intermediate representation, simplifying the logic in later compilation and validation stages. It directly reflects the language's constructs, such as typed variables, global arrays, distinct instruction types, and module structures.

4.1.2 Topological Sorting of Modules

For a language supporting multi-module projects with inter-dependencies (like cross-module function calls), topological sorting is not just beneficial but essential.

- It ensures that modules are processed in an order where all dependencies are handled before the module that depends on them. This is crucial for the **validation stage**, where cross-module call signatures and dependencies must be resolvable, and for the **compilation stage**, where definitions from one module might be needed to generate code for another.
- The implementation employs a **Depth-First Search (DFS)-based algorithm**, which is a standard and efficient method for topological sorting. This approach also naturally lends itself to cycle detection, a critical aspect of validating module dependencies.

4.1.3 Validation Process

The validation phase is a critical step to ensure the semantic correctness of the source code before attempting formal verification or final code generation.

- Performing validation on **topologically sorted modules** ensures that all necessary inter-module information is available for checking.
- The explicit **cyclic dependency check** using DFS is fundamental because cyclic dependencies typically render a program uncompilable or its semantics ill-defined. Reporting the specific cycle found aids the developer in rectifying the issue.
- The subsequent **per-module validation**—checking array declarations for valid lengths, performing type checking throughout procedure bodies using another DFS traversal, ensuring correct usage of control flow constructs like `break` and `continue` within `range` loops, and validating procedure call signatures against their definitions—collectively enforces the language's static semantics. This detailed analysis catches a wide array of common programming errors early, improving code quality and reducing the likelihood of issues during the more intensive formal verification stage.

4.1.4 Coq Code Generation Strategy

The compilation to Coq is the cornerstone of the formal verification capability, and its implementation details are geared towards producing clear, verifiable Coq code.

- **Systematic Name Sanitization:** Coq has specific rules for identifiers. The implemented sanitization process, which creates unique, Coq-compatible names for functions, variables, and arrays (including handling potential collisions), is essential for generating valid Coq modules.
- **Direct Mapping of Language Structures to Coq Types:** Generating Coq `Inductive` types for `arrayIndex` and `variableIndex`, and helper functions like `arrayType` and `arrays`, directly translates the structural elements of the custom language (global arrays, local variables) into Coq's rich type system. This provides a strong typed foundation for proofs. Furthermore, generating `EqDecision` instances for these types is crucial, as it enables Coq's decision procedures and tactics that rely on decidable equality, simplifying many proof obligations.
- **Adherence to the Monadic Framework:** The consistent use of monadic binding (`>>=`) in the generated Coq code for sequencing operations is a direct

implementation of the effectful computation model described in the methodology. This ensures that side effects are handled in a principled and traceable manner, which is fundamental for reasoning about program states and transitions during formal verification.

- **Recursive AST Traversal for Translation:** Using a DFS-based approach to traverse the `CoqCPAST` and translate each node (literals, operations, control flow, calls) into its corresponding Coq expression or monadic action is a natural and systematic way to ensure comprehensive translation of the program logic.

4.1.5 Solidity Code Generation Strategy

The Solidity code generator aims to produce deployable, efficient, and secure smart contracts that faithfully reflect the behavior of the verified custom language programs.

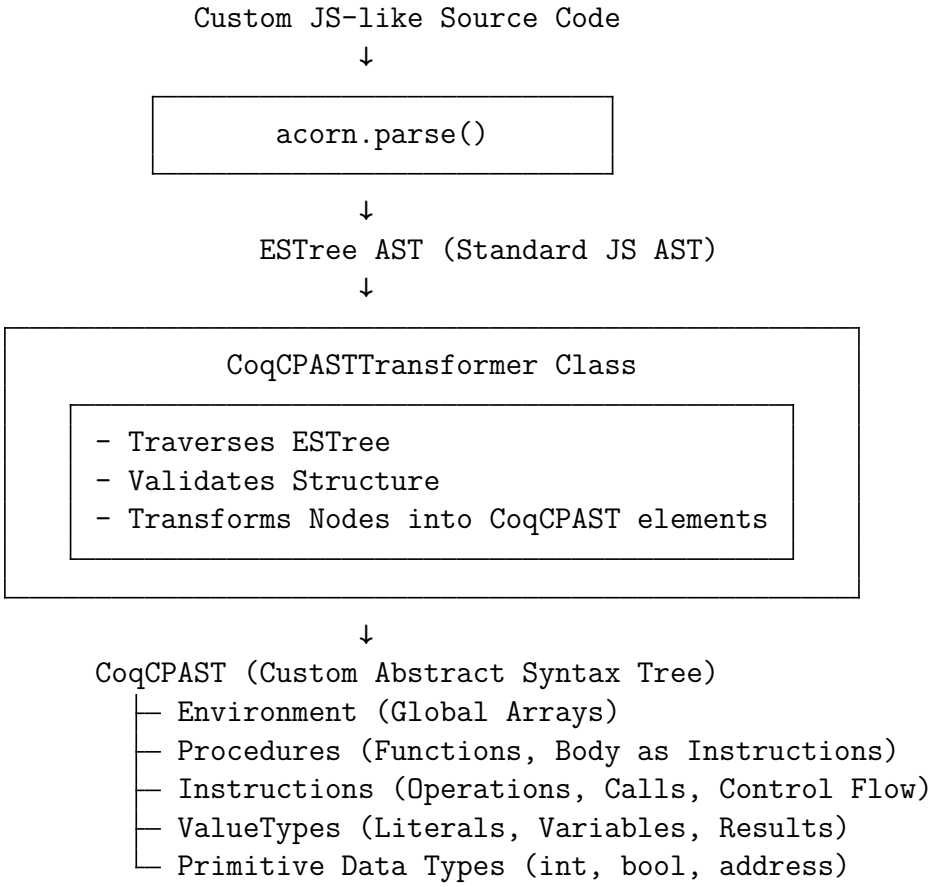
- **Two-Contract Structure for Clarity and Functionality:** The use of a separate `SelfDestructContract` for the `donate` functionality is a standard pattern that provides a clean and gas-efficient way to transfer Ether without invoking the recipient's fallback code. This keeps the main `GeneratedCode` contract focused on the core application logic.
- **Representation of Custom Language Features:**
  - Generating **Solidity structs for tuple types** is a necessary adaptation, as Solidity does not natively support tuples in the same way for state variables or complex function arguments/returns. This ensures that the data structures from the custom language are preserved.
  - The mapping of global arrays to **Solidity storage arrays** and custom language procedures to **private Solidity functions** is a direct and logical translation of the program's structure.
- **Incorporation of Safety and Utility Functions:**
  - The generation of **utility functions for bounds-checked array access and safe arithmetic operations** (like division that reverts on error) is a critical design choice. While the Coq verification proves the logic correct under its own semantic rules (including `Trap` effects for errors), these generated Solidity functions translate those safety assumptions into explicit runtime checks in the deployed contract. This helps prevent common Solidity vulnerabilities and ensures the deployed code behaves more reliably, closer to the verified model.
  - The **constructor for initializing storage arrays** ensures that the contract starts in a predictable, default state, which aligns with the initial conditions often assumed during formal verification.
- **Entry Point and Data Handling:** The use of a **payable `fallback` function as the primary dispatcher** is a flexible way to handle incoming calls and calldata. It processes `msg.data` and routes execution to the appropriate (verified) internal procedure. This design mirrors the custom language's `communication` array mechanism for input/output and is a common pattern for contracts requiring versatile calldata handling. The careful management of the `bytes memory`

`data` buffer, including its return via inline assembly, is standard practice for dynamic data handling in Solidity.

- **Optimization with `unchecked` Blocks:** The strategy of generating procedure bodies within `unchecked` blocks (as noted for procedure functions) is justifiable if, and only if, all necessary safety checks (e.g., division by zero, array bounds) are *explicitly generated by the compiler itself* based on the verified properties and the language semantics. As these checks are indeed systematically included by the compiler, the `unchecked` blocks can offer gas optimizations without sacrificing the safety established by the formal verification and the compiler's safety-guard generation.

Here is a detailed description of the compilation pipeline:

4.2 Parsing



The parser is written in TypeScript. It expects code written in a custom language and returns an Abstract Syntax Tree. The custom language is intentionally designed to be a subset of valid JavaScript syntax, even though the semantics does not match JavaScript semantics at all. This is an intentional design choice to reduce the burden of having to write a custom parser.

With this intentional design choice, we then use an off-the-shelf library called `acorn` to parse the code into a standard ESTree AST. It then traverses this ESTree AST, validating its structure against the expected custom language syntax and transforming it into the CoqCPAST format.

The parser consists of these components.

4.2.1 Primitive Data Types (*PrimitiveType*)

Defines the basic data types supported by the language:

- *'bool'*
- *'int8'*, *'int16'*, *'int32'*, *'int64'*, *'int256'* (various integer sizes)
- *'address'* (suggesting blockchain or distributed system context)

4.2.2 Array Structures

- ***ArrayDeclaration***: Describes a declared array, including:
  - *itemTypes*: An array of *PrimitiveType* defining the types of elements in a tuple (if the array stores tuples).
  - *length*: A literal number specifying the array's fixed size, with its source location.
- ***Environment***: Represents the global environment of the program, primarily containing:
  - *arrays*: A *Map* where keys are array names (strings) and values are *ArrayDeclaration* objects.

4.2.3 Variables and Values

- ***Variable***: Represents a declared variable within a procedure, holding its *PrimitiveType*.
- ***LocalBinder***: Represents a reference to a local variable (or loop variable) by its name.
- ***ValueType***: A discriminated union representing the different kinds of values an expression can evaluate to:
  - *LocalBinder*: An identifier, referring to the current iteration of a loop.
  - *Literal*: A raw numeric, boolean, or string value. Includes *raw* string representation and *location*.
  - *Instruction*: The result of another instruction (as many instructions produce values).
- ***castToInstruction***: A utility function to safely cast a *ValueType* to an *Instruction* if it is one, otherwise *undefined*.

4.2.4 Operations

- ***BinaryOp***: Defines supported binary operations like *add*, *subtract*, *multiply*, *mod*, bitwise operations, boolean operations, shifts, and equality checks.
- ***UnaryOperationInstruction***: Represents operations with a single operand (e.g., *minus*, *plus*, *bitwise not*, *boolean not*).
- ***BinaryOperationInstruction***: Represents operations with two operands.

4.2.5 Instructions (*Instruction*)

This is the core of the AST's executable part. It's a large discriminated union, where each member represents a specific command or expression in the language. Each instruction includes a *location* field for error reporting and debugging. Key instruction types include:

- **Variable/State Access:** *get*, *set*.
- **Array/Storage Operations:** *store*, *retrieve*, *subscript*.
- **Communication & Interaction (Blockchain/System Context):**
  - *COMMUNICATION*: A *Symbol* used as a special name for a communication area.
  - *communication area size*, *invoke*, *donate*, *get sender*, *get money*, *construct address*.
- **Control Flow:** *range*, *condition*, *break*, *continue*, *flush*.
- **Arithmetic & Type Coercion:** *sDivide*, *divide*, *coerceInt8* through *coerceInt256*, *less*, *sLess*.
- **Procedure Calls:** *call*, *cross module call*.

4.2.6 Procedures and Modules

- ***Procedure***: Represents a function or procedure, containing: *name*, *nameLocation*, *variables* (a *Map*), and *body* (an array of *Instruction*s).
- ***CoqCPAST***: The root of the generated AST, containing: *environment*, *procedures*, *moduleName*, and *moduleNameLocation*.

4.2.7 Error Handling

- ***ParseError***: A custom error class that extends *Error*.
- ***Location***: A structure (*start: line, column , end: line, column*) to pinpoint errors.
- ***formatLocation***: Utility function to create a string representation of a *Location*.

4.2.8 The *CoqCPASTTransformer* Class

This class is responsible for the transformation process.

Constructor

```
constructor(code: string) {
  this.ast = acorn.parse(code, {
    sourceType: 'module',
    ecmaVersion: 2023,
    locations: true,
  });
  this.result = new CoqCPAST();
```

```
8 }
```

Listing 4.1: CoqCPASTTransformer Constructor

- Takes the raw JavaScript-like source *code* as a string.
- Uses *acorn.parse()* to generate an ESTree AST. *locations: true* is enabled.
- Initializes an empty *CoqCPAST* object.

***transform()* Method**

This is the main public method that orchestrates the transformation.

- Iterates over top-level statements of the Acorn AST.
- **Top-Level Structure Enforcement:** Expects *ExpressionStatement*s whose expressions are *CallExpression*s. The *callee* must be one of *environment*, *procedure*, or *module*.
  - ***module* expression**: Parses *module(Identifier)*. Sets *this.result.moduleName*.
  - ***environment* expression**: Parses *environment(arrayName: array([...], length))*. Populates *this.result.environment.arrays*.
  - ***procedure* expression**: Parses *procedure("name", vars, () => body)*. Calls *this.transformBodyNode()* for the body. Adds to *this.result.procedures*.
- Returns the populated *CoqCPAST*.

transformBodyNode(bodyNode)

A private helper to transform statements within a block.

- Iterates through statements.
- Handles: *EmptyStatement*, *IfStatement* (to *condition* instruction), *ExpressionStatement* with literals (*"break"*, *"continue"*, *"flush"*), other *ExpressionStatement*s (expected *CallExpression*s for instructions).
- Returns an array of *Instruction*s.

processNode(node: ExtendNode<ESTree.Node>)

Recursively transforms individual ESTree nodes into *ValueType*s.

- Dispatches based on *node.type*:
  - *CallExpression* → *this.processInstruction()*.
  - *Identifier* → *LocalBinder*.
  - *Literal* → *literal ValueType*.
  - *BinaryExpression* → *binaryOp* instruction via *this.processBinaryExpression()*.
  - *UnaryExpression* → *unaryOp* instruction.
  - *MemberExpression* → *subscript* instruction.
  - *LogicalExpression* → *binaryOp* instruction.
- Throws *ParseError* for unrecognized types.

processBinaryExpression(node)

Handles ESTree *BinaryExpression*s.

- Uses *this.getBinaryOperator()* to map ESTree operator string to custom *BinaryOp*.
- Recursively calls *this.processNode()* on operands.
- Returns a *BinaryOperationInstruction*.

getBinaryOperator(operator: string, location: Location)

Maps JavaScript binary operator strings to the language's *BinaryOp* enum values.

4.2.9 processInstruction(name: string, args, location: Location)

The core of instruction parsing, a large *switch* statement based on the instruction *name*.

- For each instruction type:
  - **Argument Validation:** Rigorously checks number and ESTree types of arguments.
  - **Argument Processing:** Recursively calls *this.processNode()* for expression arguments.
  - **Instruction Construction:** Constructs the corresponding *Instruction* object.
  - **Special Cases:**
    - *store* and *retrieve*: Handle forms for named arrays and the *COMMUNICATION* area.
    - *range*: Expects second argument as an arrow function *(loopVar) => { ... }*.
    - *call*: Differentiates between intra-module (2 args) and cross-module (4 args) calls, parsing preset variables and array mappings.
- Throws *ParseError* for mismatches or unknown instructions.

Input Language Syntax

The parser expects a specific, JavaScript-based Domain-Specific Language (DSL):

- **Top-Level Declarations:**

```
module(MyModule);
environment({ arr1: array([int32, bool], 10) });
procedure("myProc", { x: int32 }, () => { /* statements */ });

```

- **Statements within Procedures:** Instructions as function calls (*set("x", 10);*). Nested expressions. Control flow via *if/else* and *range*. Standard JS operators.
- **Literals:** Standard JS numbers, booleans, strings.
- **Identifiers:** For variables, array names, procedure names.

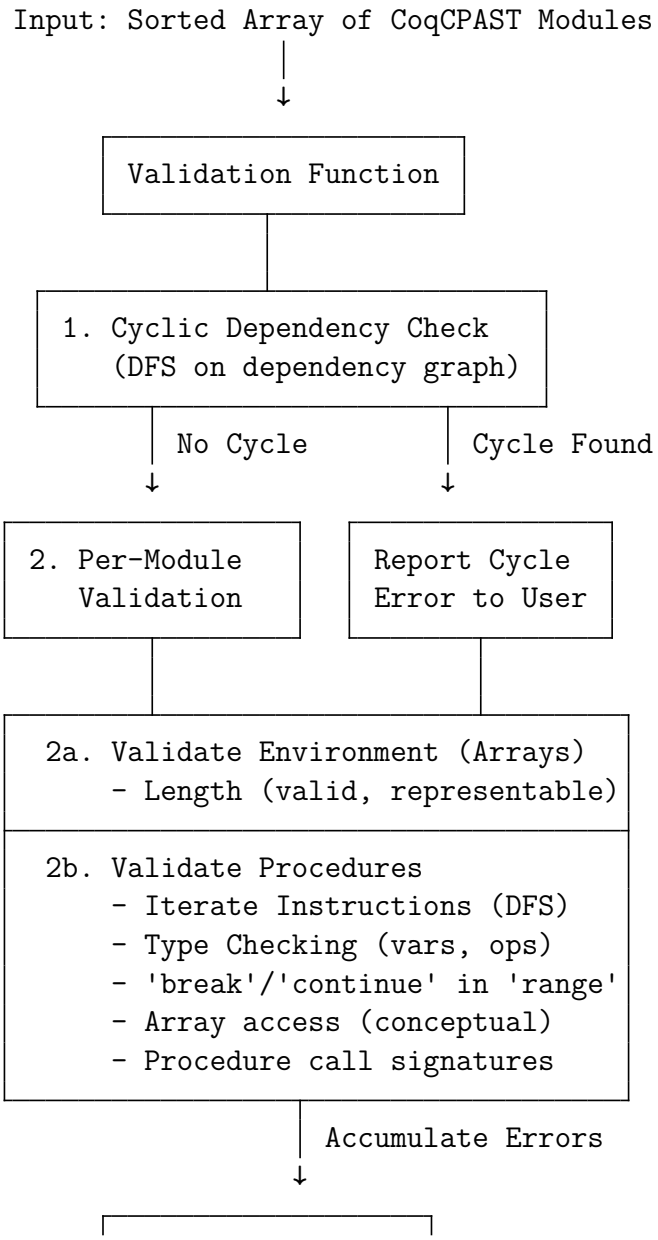
4.3 Topological sorting

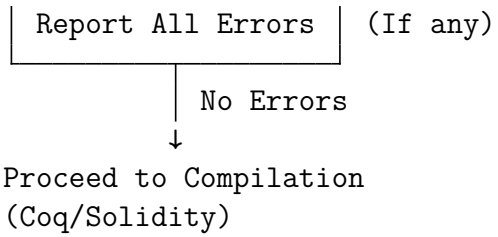
When compiling a computer program with multiple interconnected modules, it is important to determine the processing order of the modules. It is essential to find a suitable order such that when a module is being processed, all modules it depends upon have already been processed before. This is important in resolving cross module calls, which is the main feature that links modules together in the custom programming language.

As the parsing stage processes each module in isolation, it is not necessary to sort the modules to parse them. However, the next stages, which are validation and compilation, do depend on resolving cross-module dependencies and thus, topological sorting is essential.

In section A.1 in the appendix, we describe the topological sorting process in detail.

4.4 Validation





This stage deals with the semantic correctness of the source code. While the parsing stage already does some basic validation checks, it does not have full insight into the meaning of the code. Furthermore, the parser deals with each module in isolation, but modules can be linked together through cross procedure calls. And these cross procedure calls have to be validated.

The validation stage consists of a function that expects an array of sorted modules, and returns a list of validation errors. These validation errors are then reported to the user if the list is nonempty, and otherwise, the compiler proceeds to the next stage of the compilation pipeline.

First, the validation function checks for cyclic dependencies and reports a cycle to the user. This involves a DFS traversal of the dependency graph. While topological sort can detect the presence of cycles, it alone can't pinpoint a concrete cycle to report to the user. And this information is quite important for the user to troubleshoot the cyclic dependency issue.

Refer to figure 4.1 for a diagram of the cycle detection process. We describe it in more detail in section A.2 in the appendix.

4.5 Main Validation Logic

After checking the dependency graph for cycles, the validation stage then proceeds to validating other aspects of the code. Its objective is to perform a comprehensive static analysis, identifying various semantic and type errors within the code defined by these modules.

The operations following the initial duplicate module check and cyclic dependency validation are as follows:

4.5.1 Initialization for Cross-Module and Intra-Module Validation

- `crossModuleProcedureMap = new PairMap<string, string, Procedure>()`: This data structure stores mappings from (`moduleName`, `procedureName`) pairs to `Procedure` objects. It is crucial for resolving and validating calls to procedures defined in other modules.

- `seenModules = new Map<string, CoqCPAST>()`: This map tracks modules that have been processed, storing the module name as the key and its `CoqCPAST` (Abstract Syntax Tree representation) as the value. This facilitates quick lookups of module definitions, particularly for validating cross-module calls.

- `errors: ValidationError[] = []`: An array is initialized to accumulate any validation errors found during the process.

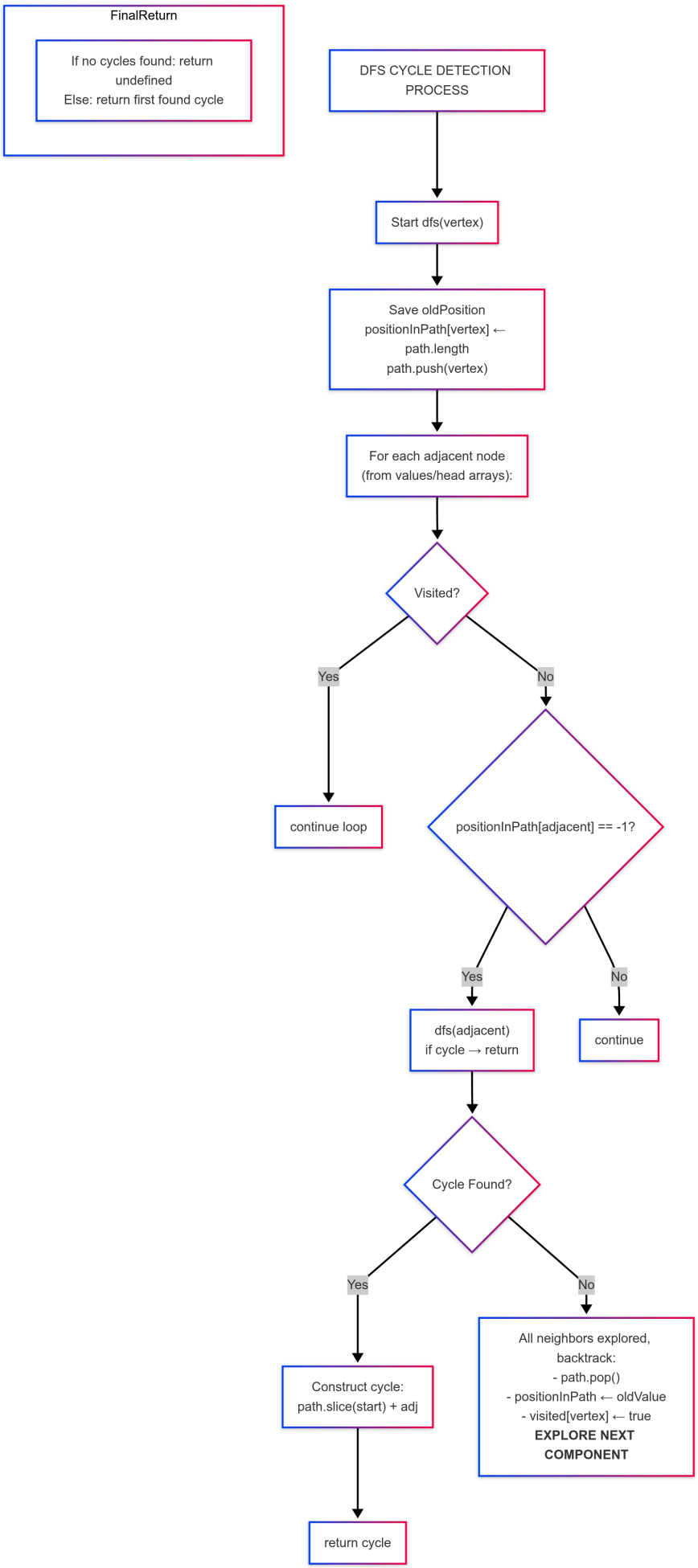


Figure 4.1: Cycle detection process

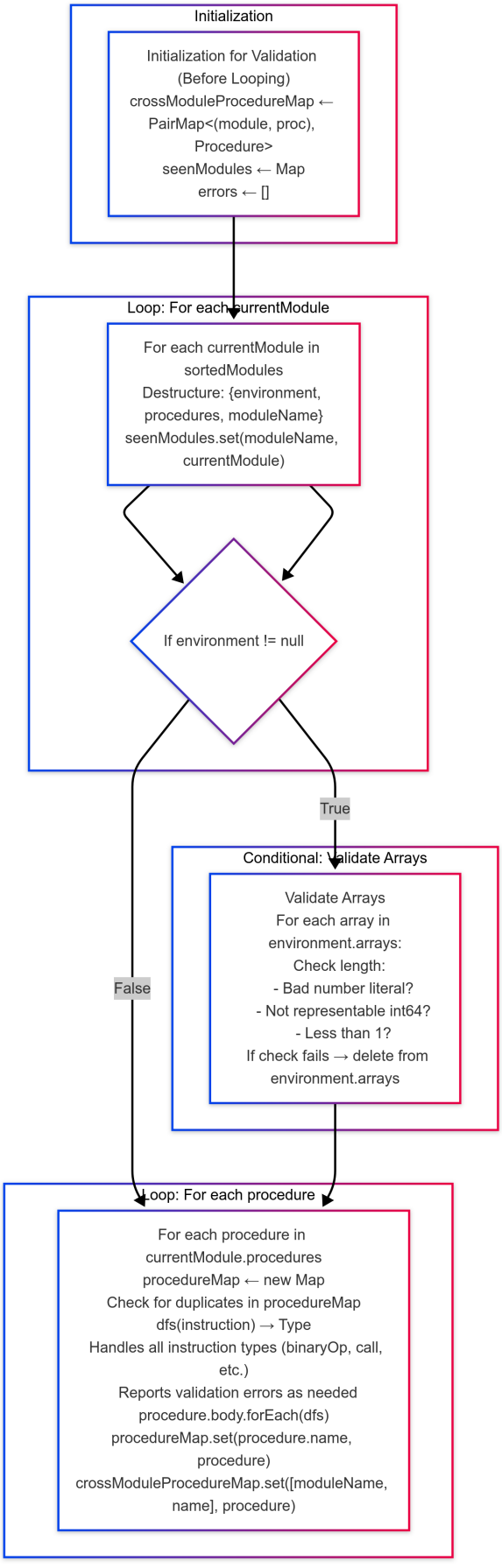


Figure 4.2: Validation process

4.5.2 Iterating Through Each Module

The validator iterates through each `currentModule` in the `sortedModules` list.

- `const { environment, procedures, moduleName } = currentModule;`: Destructures the current module to access its `environment` (global variables and arrays), its list of `procedures`, and its `moduleName`.
- `seenModules.set(moduleName, currentModule);`: Adds the current module to the `seenModules` map.

4.5.3 Validating Module Environment (Array Declarations)

If `currentModule` has an `environment` (`environment !== null`), its array declarations are checked:

- It iterates through each array in `environment.arrays`.
- **Array Length Validation**: For each array's length:
  - `'bad number literal'`: Checks if the `raw` string representation of the length is a valid integer.
  - `'not representable int64'`: Converts the `raw` length to a `BigInt`. If this number is outside the 64-bit integer range, an error is reported.
  - `'array length can't be less than 1'`: Checks if the evaluated array length is less than 1.
- If any check fails, the array is `deleted` from `environment.arrays`.

4.5.4 Iterating Through Procedures Within the Current Module

Each `procedure` within `currentModule.procedures` is processed.

- `const procedureMap = new Map<string, Procedure>()`: A new map is created for each module to store procedures defined within that module, used for resolving intra-module calls.
- **Duplicate Procedure Check**:
  - `if (procedureMap.has(procedure.name))`: Checks if a procedure with the same name already exists in the current module. If so, a `'duplicate procedure'` error is added.
- **Type Checking with Depth-First Search (DFS)**: A local `dfs` function recursively traverses the AST of each instruction within the procedure's body.
  - `type Type = PrimitiveType | 'string' | 'statement' | 'illegal' | PrimitiveType[]`: Defines possible return types for `dfs`. `PrimitiveType` includes types like `int8`, `int64`, `bool`, `address`. `PrimitiveType[]` represents a tuple. `'statement'` is for non-value-evaluating instructions. `'illegal'` signifies a type error.

- `hasSurroundingRangeCommand = false`: A flag to track if instructions like `break` or `continue` are within a `range` command.
- `presentBinderType = new Map<string, 'int64' | 'int8'>()`: Tracks types of loop variables introduced by `range` commands.

The `dfs(instruction: ValueType): Type` Function: Uses a `switch` statement on `instruction.type`.

- **`binaryOp` (e.g., +, -, *, /, ==, !=, &&, ||)**:
  - Recursively calls `dfs` on operands. Reports `'expression no statement'` if operands are statements.
  - *Numeric Operations*: Expects both operands to be numeric and of the same type. Reports `'binary expression expects numeric'` or `'binary expression type mismatch'`. Returns the numeric type.
  - *Equality Operations*: Expects operands of the same type (numeric, `bool`, or `address`). Reports `'binary expression type mismatch'`. Returns `'bool'`.
  - *Boolean Operations*: Expects both operands to be `'bool'`. Reports `'binary expression expects boolean'`. Returns `'bool'`.
- **`break` / `continue`**: Checks `hasSurroundingRangeCommand`. Reports `'no surrounding range command'` if false. Returns `'statement'`.
- **`call` (Intra-Module Procedure Call)**: Looks up procedure in `procedureMap`. Reports `'procedure not found'` if missing. Calls `validateCall` helper which checks for `'variable not present'`, `'variable type mismatch'`, or `'expression no statement'`. Returns `'statement'`.
- **`cross module call`**: Checks target module in `seenModules` (`'module not found'`). Looks up procedure in `crossModuleProcedureMap` (`'procedure not found'`). Calls `validateCall`. Validates array mappings: checks for `'must specify all arrays'`, `'array doesn't exist in procedure module'`, `'array doesn't exist in current module'`, and `'array shape mismatch'`. Returns `'statement'`.
- **`coerceInt8` / `coerceInt16` / `coerceInt32` / `coerceInt64` / `coerceInt256`**: Calls `dfs` on the value. Reports `'expression no statement'` if value is a statement. Returns the corresponding integer type.
- **`condition` (If Statement)**: Recursively calls `dfs` on `body`, `alternate`, and `condition`. Reports `'expression no statement'` if condition is a statement. Checks if condition type is `'bool'` (reports `'condition must be boolean'`). Returns `'statement'`.
- **`divide` / `sDivide` / `less` / `sLess`**: Similar to `binaryOp`. Expects numeric operands of the same type. Reports `'instruction expects numeric'` or `'instruction type mismatch'`. `divide`/`sDivide` return numeric type; `less`/`sLess` return `'bool'`.
- **`get` (Variable Retrieval)**: Looks up variable in `procedure.variables`. Reports `'undefined variable'`. Returns variable's type.
- **`literal`**: Returns `'bool'` for boolean, `'string'` for string. For numbers, validates format (`'bad number literal'`) and int64 range (`'not representable int64'`). Returns `'int64'`.

- **`local binder` (Loop Variable)**: Looks up in `presentBinderType`. Reports `'undefined binder'`. Returns binder's type.
- **`range` (Loop)**: Calls `dfs` on `end` expression. Validates `endType` (`'range end must be int64 or string'`). Manages `hasSurroundingRangeCommand` and `presentBinderType`. Returns `'statement'`.
- **`retrieve` (Array Element Read)**: Validates `index` type (`'instruction expects int64'`). For `COMMUNICATION` array: returns `'int8'`. For regular arrays: looks up array (`'undefined array'`). Returns array's `itemTypes`.
- **`set` (Variable Assignment)**: Calls `dfs` on value. Looks up variable (`'undefined variable'`). Compares types (`'variable type mismatch'`). Returns `'statement'`.
- **`store` (Array Element Write)**: Validates `index` type. For `COMMUNICATION` array: validates value type is `'int8'`. Returns `'statement'`. For regular arrays: type-checks tuple elements. Looks up array (`'undefined array'`). Compares tuple types with array's element types (`'variable type mismatch'`). Returns `'statement'`.
- **`subscript` (Tuple/Array/Address Access)**: Calls `dfs` on `value`. If `value` is a tuple: validates `index` is a literal number and in bounds (`'index must be a number literal'`, `'index out of bounds'`). Returns element type. If `value` is `'address'` and `blockchain`: validates `index` type is `'int64'`. Returns `'int8'`. Else, reports `'instruction expects address or tuple'` / `'instruction expects tuple'`.
- **`unaryOp` (e.g., -, !, bitwise not)**: Calls `dfs` on value. If numeric: `bitwise not`, `minus`, `plus` return `valueType`; `boolean not` reports `'unary operator expects numeric'`. Reports errors for tuples, strings, addresses. If boolean: `boolean not` returns `'bool'`; others report `'unary operator expects boolean'`.
- **Blockchain-Specific Instructions (`get sender`, `communication area size`, `get money`)**: Return respective types: `'address'`, `'int64'`, `'int256'`.
- **`construct address`**: Validates bytes are `'int8'`. Returns `'address'`.
- **`donate`**: Validates `money` is `'int256'` and `address` is `'address'`. Returns `'statement'`.
- **`invoke` (Blockchain Contract Call)**: Validates types: `money` (`'int256'`), `address` (`'address'`), `communicationSize` (`'int64'`). Validates communication `array` is `'[int8]'` (reports `'undefined array'` or `'instruction expects [int8] array'`). Returns `'statement'`.

- After defining `dfs`, `procedure.body.forEach(dfs)` is called.
- `procedureMap.set(procedure.name, procedure)`: The validated procedure is added to the current module's `procedureMap`.
- `crossModuleProcedureMap.set([moduleName, procedure.name], procedure)`: The procedure is added to `crossModuleProcedureMap`.

Finally, the function returns the accumulated list of all validation errors.

4.6 Compilation to Coq

Refer to 3.0.4 for a high-level description of the compilation process. In this section, we talk about the low-level implementation details of the compilation to Coq stage.

4.6.1 Helper Functions

- **`getCoqString(text: string): string`**: Converts a JavaScript string into a Coq string representation. For ASCII-only strings, it uses standard Coq string literals (e.g., `""`), escaping internal double quotes. For strings with non-ASCII characters (UTF-8), it constructs a Coq string by chaining byte values (e.g., `String "099" (String "097" (String "116" EmptyString))` for "cat").
- **`byteLength(x: string): number`**: A utility to calculate the byte length of a string using `TextEncoder`. (Not directly used in the main `coqCodegen` logic).
- **`indent = ' '`**: Defines a two-space string for indenting generated Coq code.
- **`generateInductive(name: string, arms: string[]): string`**: Constructs a Coq `Inductive` type definition. If `arms` is empty, it defines an empty type. Otherwise, it creates an inductive type with the given `name` and constructors from `arms`.
- **`sanitizeName(name: string): string`**: Filters characters in a given `name` to produce a valid Coq identifier, keeping only alphanumeric characters, underscores (_), and single quotes (').

4.6.2 Main Code Generation Logic: `coqCodegen(sortedModules: CoqCPAST[])`

This is the core function orchestrating the translation.

Initialization for Sanitization

- `mapToSanitizedFunc`, `mapToSanitizedVar`, `mapToSanitizedArray`: Instances of `PairMap` used to cache sanitized names, mapping `[moduleName, originalIdentifier]` to a unique, Coq-friendly identifier for functions, variables, and arrays respectively.
- `sanitizedFuncCollisions`, `sanitizedVarCollisions`, `sanitizedArrayCollisions`: Instances of `Map` to track base sanitized name collisions and append a numeric discriminator for uniqueness.

Sanitization Functions

- **`sanitize(moduleName, identifier, namespace, mapToSanitized, sanitizedIdentifierCollisions)`**: The generic sanitization utility.
  1. Checks cache for an existing sanitized name.
  2. If not found, creates a `sanePart` from `moduleName_identifier` via `sanitizeName`.

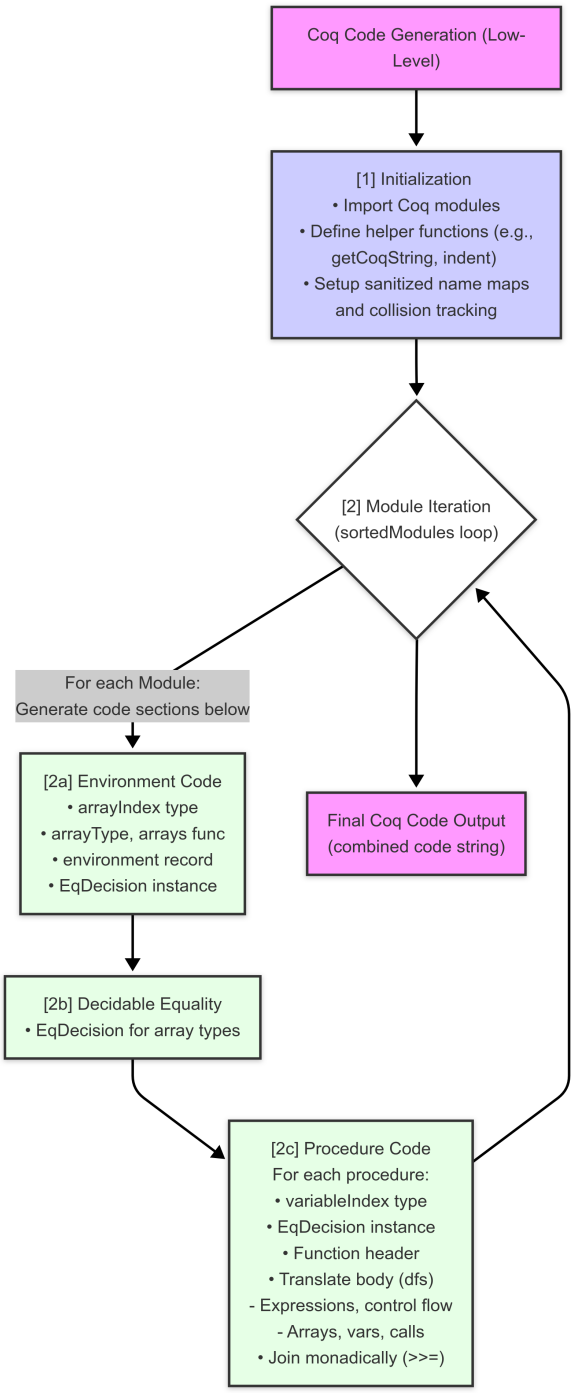


Figure 4.3: Coq compilation procedure

3. Uses a `discriminator` count for the `sanePart` to ensure uniqueness.
4. Forms the final sanitized name as `namespace_discriminator_sanePart` (e.g., `funcdef_0_ModuleA_myFunc`).
5. Caches and returns this name.

- **`sanitizeFunction(moduleName, identifier)`**: Wrapper for `sanitize` for function names (namespace `'funcdef'`).
- **`sanitizeArray(moduleName, identifier)`**: Wrapper for `sanitize` for global array names (namespace `'arraydef'`).
- **`sanitizeVariable(moduleName, functionName, identifier)`**: Wrapper for `sanitize` for local variable names, using `JSON.stringify([functionName, '_-', identifier])` for scoping (namespace `'vardef'`).
- **`sanitizeVariableIndex(moduleName, functionName)`**: Generates a unique name for the Coq `Inductive` type representing local variable names within a function (e.g., `varsfuncdef_0_MyModule_myFunc`).

Initial Coq Code Setup

Starts the Coq code string with necessary imports:

```
From CoqCP Require Import Options Imperative.
From stdpp Require Import numbers list strings.
Require Import Coq.Strings.Ascii.
Open Scope type_scope.
```

Iterating Through Modules

The code iterates through each `module` in `sortedModules`.

Environment Code Generation (`environmentCode` IIFE):

- If no environment or arrays, defines a default empty Coq `Environment`.
- Otherwise:
  1. **`arrayIndex type`**: An `Inductive` type (e.g., `arrayIndex0`) is generated. Constructors are sanitized array names (e.g., `| arraydef_0_ModuleA_arr1`).
  2. **`arrayTypeFunction`**: A Coq function `arrayType` is defined, mapping sanitized array names to the Coq type of their elements (e.g., `Z`, `bool`, or tuples like `Z * bool`).
  3. **`arrayFunction`**: A Coq function `arrays` is defined, mapping sanitized array names to their initial list state (e.g., `repeat 0 length`).
  4. **`Environment Definition`**: A Coq record `environment[moduleIndex]` is defined using the generated functions and types.
  5. **Decidability**: An instance `arrayIndexEqualityDecidable[moduleIndex] : EqDecision arrayIndex[moduleIndex]` is generated using `ltac:( solve_decisio`

Decidable Equality for Array Types (`decidableEquality`): An instance `arrayTypeEqualityDecidable[moduleIndex]` is generated for the types returned by `arrayType`, providing `EqDecision` for array element types.

Procedure Code Generation (`generatedCodeForProcedures`): Iterates over each `procedure` in the module.

- **`variableIndex` type**: An `Inductive` type is generated, with constructors being sanitized local variable names. An `EqDecision` instance is also generated.
- **Function Header**: Defines the Coq function signature (e.g., `Definition sanitizedFuncName (bools : variableIndex -> bool) ... := eliminateLocalVariables ...`). It takes maps for local variables and returns an `Action`.
- **Statement Processing (inner `dfs` function)**: This is the core AST-to-Coq translator.
  - A `localBinderMap` and `binderCounter` manage loop variables.
  - `liftExpression` helper wraps expressions in `liftToWithinLoop` if inside loops.
  - The `dfs(value: ValueType)` function recursively translates AST nodes:
    - **Literals**: JS values to Coq literals (`true`, `false`, `8203510951821098053`, `getCoqString`).
    - **Binary/Unary Ops**: To CoqCP functions (`addInt`, `andBits`, `bool_decide (x = y)`, etc.).
    - **Coercions**: To `coerceInt` or `coerceBool`.
    - **Control Flow**: `break`, `continue`, `if ... then ... else ...`, `loopString`, `loop (Z.to_nat x)`.
    - **Variable Access**: `numberLocalGet`, `booleanLocalGet`, `addressLocalGet`, `numberLocalSet`, etc., using sanitized variable names. Loop binders like `binder_N`.
    - **Array Access**: `retrieve`, `store`. `COMMUNICATION` array uses `readByte`/`setByte`.
    - **Subscripting**: Tuple access via `fst`/`snd`; address byte access via `nthTrap`.
    - **Procedure Calls**:
      - Intra-module (`'call'`): Marshals arguments, calls sanitized Coq function, wrapped in `liftToWithLocalVariables`.
      - Cross-module (`'cross module call'`): Similar, but generates `arrayMappingText` and `congruence`, then calls wrapped in `translateArrays`.
    - **Blockchain Ops**: To CoqCP equivalents (`getSender`, `getMoney`, `invokeWith Arrays`, etc.).
  - **Joining Statements**: Coq expressions for statements are joined monadically using `>>=`, with a final `Done _ _ _ tt`.
- The generated Coq code for the current module (`environmentCode`, `decidableEquality`, `generatedCodeForProcedures`) is appended to the main `code` string.

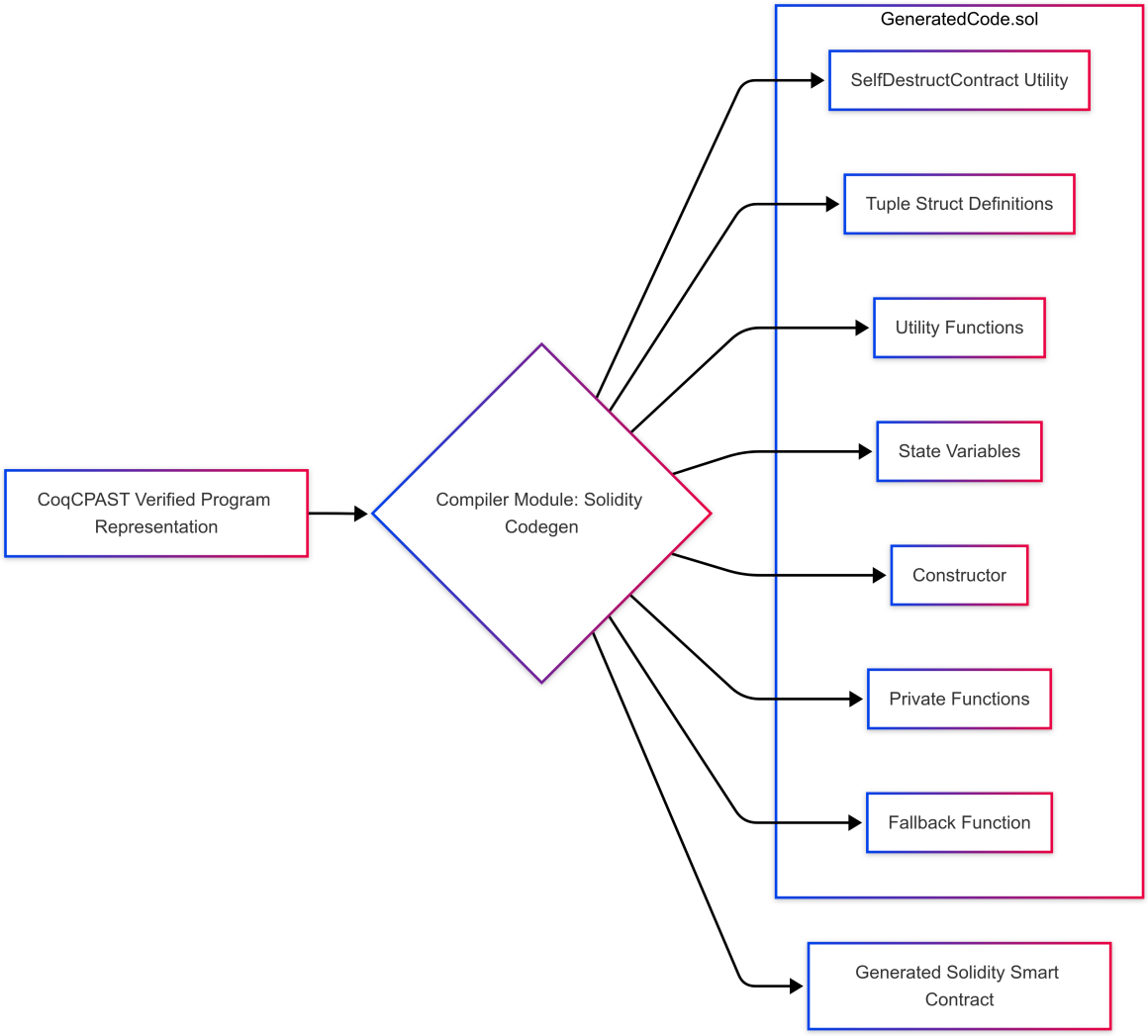


Figure 4.4: Solidity code generation

The function returns the complete `code` string, representing the Coq translation of all input modules.

4.7 Compilation to Solidity

In this section, we describe the process by which the intermediate CoqCPAST representation is translated into a deployable Solidity smart contract using the 'solidity-Codegen' function implemented in TypeScript. We first outline the overall structure of the generated Solidity code, then detail each major component of the translation pipeline.

4.7.1 Overview of the Generated Contract

The output of the compiler is a Solidity source file containing two contracts:

- **SelfDestructContract**: A minimal contract whose constructor immediately transfers its balance to a target address and invokes 'selfdestruct'. This is used to implement the 'donate' primitive in the source language.

- **GeneratedCode**: The main contract encapsulating:
  1. Type definitions (`struct` declarations) for tuple types occurring in the program.
  2. Utility functions for memory safety (bounds-checked array access and updates).
  3. Type coercion and arithmetic wrappers (e.g., safe division, signed/unsigned casts).
  4. Storage declarations mirroring the top-level environment arrays.
  5. A constructor initializing storage arrays with default values.
  6. One private `procedureN` function per CoqCPAST procedure.
  7. A `fallback` function dispatching to the `main` procedure on raw calldata, and returning its output.

4.7.2 Tuple Struct Generation

To represent source-level tuples in Solidity, the compiler:

- Collects each unique combination of primitive types encountered in tuple expressions.
- Assigns each combination a fresh `struct TupleK` declaration, where K is an incrementing counter.
- Emits auxiliary functions `arrayGet` and `arraySet` for each tuple struct to perform bounds-checked access on storage arrays of that struct type.

```
1  struct Tuple0 {
2  uint64 item0;
3  int8  item1;
4  }
5
6  function arrayGet(Tuple0[] storage array, uint64 index) private returns (
       Tuple0 memory) {
7      if (index >= array.length) { assembly { revert(0, 0) } }
8      return array[index];
9  }
10
11 function arraySet(Tuple0[] storage array, uint64 index, Tuple0 memory
       value) private {
12     if (index >= array.length) { assembly { revert(0, 0) } }
13     array[index] = value;
14 }
```

Listing 4.2: Tuple struct generation in Solidity

4.7.3 Contract Scaffolding and Utilities

The compiler emits a `pragma` header and declares:

- A helper `constructAddress` to build an ‘address‘ from 20 bytes.

- A ‘shoot‘ wrapper to deploy `SelfDestructContract` with payable value, realizing the ‘donate‘ primitive.
- Functions `communicationGet` and `communicationSet` to index the `bytes` buffer representing the communication area.
- Signed and unsigned division functions (`sdivint8,16,32,64,256` and ‘divint8,...,256‘) that revert on overflow or division by zero.

4.7.4 Type and Operator Mapping

Primitive types in CoqCPAST are mapped to Solidity types as follows:

- `bool` ↦ `bool`
- `int8` ↦ `uint8`
- `int16` ↦ `uint16`
- `int32` ↦ `uint32`
- `int64` ↦ `uint64`
- `int256` ↦ `uint256`
- `address` ↦ `address`

Binary and unary operators are translated to their Solidity equivalents, for instance:

- `add` ↦ `+`, `subtract` ↦ `-`, `equal` ↦ `==`
- `bitwise and` ↦ `&`, `boolean and` ↦ `&&`
- `minus` ↦ unary `-`, `boolean not` ↦ `!`, etc.

4.7.5 Environment Storage and Constructor

For the main (empty-name) module, each top-level array in the CoqCPAST environment becomes a ‘storage‘ array field:

```
Tuple0[] private environment0;
Tuple3[] private environment1;
```

A constructor loops over the declared length of each array and pushes zero-initialized tuples:

```
constructor() {
for (uint i = 0; i < N0; i++)
environment0.push(Tuple0(0,0));
for (uint i = 0; i < N1; i++)
environment1.push(Tuple3(0,0,0,0));
}
```

4.7.6 Procedure Functions

Each CoqCPAST procedure is emitted as a private Solidity function `procedureK`, whose parameters consist of:

1. Storage references for each environment array (`TupleX[] storage environmentM`).
2. Local variables as parameters (`uint64 local0, bool local1`, etc.).
3. A `bytes memory communication` buffer.

The function body is generated in an 'unchecked' block and translates each CoqCPAST instruction to a corresponding Solidity statement or expression, respecting control flow constructs:

- `get`/`set` instructions map to local variable reads/writes.
- `store`/`retrieve` map to `arraySet`/`arrayGet` calls.
- `invoke` produces an external `call`, handling calldata copying and return data storage.
- `range` and `condition` generate `for` loops and `if` statements.
- `donate` emits `shoot(payable(addr), value)`

4.7.7 Fallback Function

Finally, the contract defines a payable `fallback()` method that:

1. Reads `msg.data` into a 'bytes memory data'.
2. Calls `procedureN_main` with default local arguments and the calldata buffer.
3. Returns the possibly modified buffer via inline assembly:

```
fallback() external payable {
bytes memory data = msg.data;
procedure0(environment0, ..., data);
assembly {
return(add(data, 0x20), mload(data))
}
}
```

The function returns the complete `code` string, representing the Solidity translation of all input modules.

Chapter 5
RESULT

This chapter delves into the example contracts that are formalized with our framework.

5.1 Disjoint Set Union contract

Related files

1. https://github.com/huynhtrankhanh/CoqCP/blob/main/programs/DisjointSetUnion.js
2. https://github.com/huynhtrankhanh/CoqCP/blob/main/programs/DisjointSetUnion.module.json
3. https://github.com/huynhtrankhanh/CoqCP/blob/main/theories/DisjointSetUnion.v
4. https://github.com/huynhtrankhanh/CoqCP/blob/main/theories/DisjointSetUnionCode.v
5. https://github.com/huynhtrankhanh/CoqCP/blob/main/theories/DisjointSetUnionCode2.v
6. https://github.com/huynhtrankhanh/CoqCP/blob/main/theories/DisjointSetUnionCode3.v

This is the source code of the contract.

```
environment({
  dsu: array([int8], 100),
  hasBeenInitialized: array([int8], 1),
  result: array([int8], 1),
})

procedure('ancestor', { vertex: int8, work: int8 }, () => {
  set('work', get('vertex'))
  range(100, (_) => {
    if (sLess(retrieve('dsu', coerceInt64(get('work')))[0], coerceInt8(0)
   )) {
      ;('break')
    }
    set('work', retrieve('dsu', coerceInt64(get('work')))[0])
  })
  store('result', 0, [get('work')])
  set('work', get('vertex'))
  range(100, (_) => {
    if (sLess(retrieve('dsu', coerceInt64(get('work')))[0], coerceInt8(0)
   )) {
      ;('break')
    }
    set('vertex', retrieve('dsu', coerceInt64(get('work')))[0])
```

```
    store('dsu', coerceInt64(get('work')), [retrieve('result', 0)[0]])
    set('work', get('vertex'))
  })
})

procedure('unite', { u: int8, v: int8, z: int8 }, () => {
  call('ancestor', { vertex: get('u') })
  set('u', retrieve('result', 0)[0])
  call('ancestor', { vertex: get('v') })
  set('v', retrieve('result', 0)[0])
  if (get('u') != get('v')) {
    if (
      sLess(
        retrieve('dsu', coerceInt64(get('u')))[0],
        retrieve('dsu', coerceInt64(get('v')))[0]
      )
    ) {
      set('z', get('u'))
      set('u', get('v'))
      set('v', get('z'))
    }
    store('dsu', coerceInt64(get('v')), [
      retrieve('dsu', coerceInt64(get('u')))[0] +
        retrieve('dsu', coerceInt64(get('v')))[0],
    ])
    store('dsu', coerceInt64(get('u')), [get('v')])
    donate(
      getSender(),
      coerceInt256(-retrieve('dsu', coerceInt64(get('v')))[0])
    )
  }
})

procedure('main', {}, () => {
  if (retrieve('hasBeenInitialized', 0)[0] == coerceInt8(0)) {
    store('hasBeenInitialized', 0, [coerceInt8(1)])
    range(100, (i) => {
      store('dsu', i, [coerceInt8(-1)])
    })
  }
  call('unite', { u: retrieve(0), v: retrieve(1) })
  // gotta reset
  store('result', 0, [coerceInt8(0)])
})
```

Listing 5.1: Disjoint Set Union contract

The main correctness claim of the contract is **the maximum withdrawal amount is 5049 wei**.

5.1.1 Inspection of Contract's Correctness

Before moving on to the proof details, we first go into how the framework and Coq's logical infrastructure enables the end user to quickly check whether the contract is logically correct or not, without having to trust the person who wrote the correctness proof.

1. **Check for Compilation Errors:** Run the framework's build scripts, including

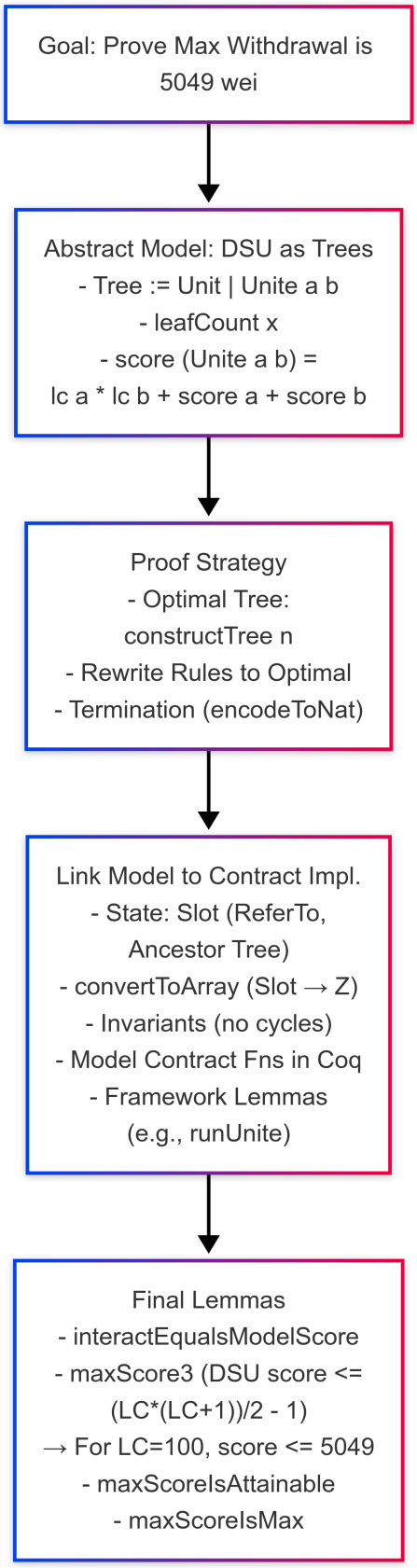


Figure 5.1: Overall strategy for DSU contract

the compilation pipeline to Solidity and Coq and Coq's verifier. If there are no errors, we can proceed with the next step.

2. **Check What the Proof Proves:** Read the theorem statements and the definitions. There are very few things that the user needs to read compared to the size of the proof. In the case of this smart contract, here are the key claims:

```
1 Lemma maxScoreIsAttainable : interact state (map (fun x => (0%Z, Z.
    of_nat x)) (seq 1 99)) = 5049%Z.
2
3 Lemma maxScoreIsMax (x : list (Z * Z)) (hN : forall a b, In (a, b) x
     -> Z.le 0 a /\ Z.lt a 256 /\ Z.le 0 b /\ Z.lt b 256) : (interact
     state x <= 5049)%Z.
4
```

Listing 5.2: Key claims of the correctness proof of the DSU contract

These key claims together state that it is possible to withdraw 5049 wei from the smart contract, and it is impossible to exceed that amount.

3. **Conclusion:** Absent bugs in the framework, when the framework says that everything is logically consistent, and the key claims match what the user wants from the smart contract, the user can trust the smart contract is correct.

5.1.2 Proof Details

Abstract Model: Interactions as Trees

The fundamental approach models DSU operations as the construction of trees.

1. Tree Representation An inductive type `Tree` is defined:

```
Inductive Tree := | Unit | Unite (a b : Tree).
```

`Unit` signifies a singleton set (a leaf in the tree), and `Unite a b` represents a new set formed by merging two pre-existing sets (sub-trees) `a` and `b`.

2. Quantifying Withdrawals: Leaf Count and Score

- `leafCount x`: This function calculates the number of `Unit` leaves in a tree `x`. It's proven that $1 \leq$ `leafCount a` for any tree `a`.
- `score x`: This function defines the total wei withdrawn, corresponding to the tree `x`:

```
Fixpoint score (x : Tree) := match x with
| Unit => 0
| Unite a b => leafCount a + leafCount b + score a + score b end.
```

When sets `a` and `b` merge, `leafCount a` + `leafCount b` (the size of the new set) is withdrawn. The terms `score a` and `score b` account for wei withdrawn during the formation of `a` and `b` respectively.

Optimal Strategy: Tree Construction and Transformation

The proof identifies an optimal tree structure that maximizes the score and shows how any tree can be transformed into this optimal form.

1. Optimal Tree Construction (`constructTree`) A function `constructTree (n : nat)` is defined to build a tree with $n + 1$ leaves that yields the highest possible score:

```
Fixpoint constructTree (n : nat) : Tree :=
match n with
| O => Unit | S n => Unite (constructTree n) Unit
end.
```

This function represents a strategy of always merging the largest accumulated set with a new singleton set. Key properties:

- `leafCount (constructTree n)` = `S n` (i.e., $n + 1$).
- `score (constructTree n)` = $\frac{(n+1)(n+2)}{2} - 1$.

2. Rewrite Rules for Optimization An exchange argument based on three rewrite rules is used to prove that `constructTree` is optimal. These rules transform any tree into the `constructTree` form without decreasing its score.

- **Rule 1:** `Unite Unit (Unite a b)` → `Unite (Unite a b) Unit`
  - Score remains unchanged (`rewriteRule1`).
  - `encodeToNat` (a measure for termination, see below) strictly decreases (`rule1_a1`).
- **Rule 2:** If `leafCount a` ≥ `leafCount d`, then:
  `Unite (Unite a b) (Unite c d)` → `Unite (Unite (Unite a b) c) d`
  - Score does not decrease (`rewriteRule2`).
  - `encodeToNat` strictly decreases (`rule2_a1`).
- **Rule 3:** If `leafCount a` < `leafCount d`, then:
  `Unite (Unite a b) (Unite c d)` → `Unite (Unite (Unite d c) b) a`
  - Score strictly increases (`rewriteRule3`).

3. Canonical Form (`noThreeRules`) Functions `hasRule1`, `hasRule2`, `hasRule3` check for the applicability of these rules. `replaceRule1`, `replaceRule2`, `replaceRule3` apply the transformations. The `noThreeRules` lemma states that if none of these three rewrite rules can be applied to a tree `x`, then `x` is identical to `constructTree`(`leafCount x`− 1).

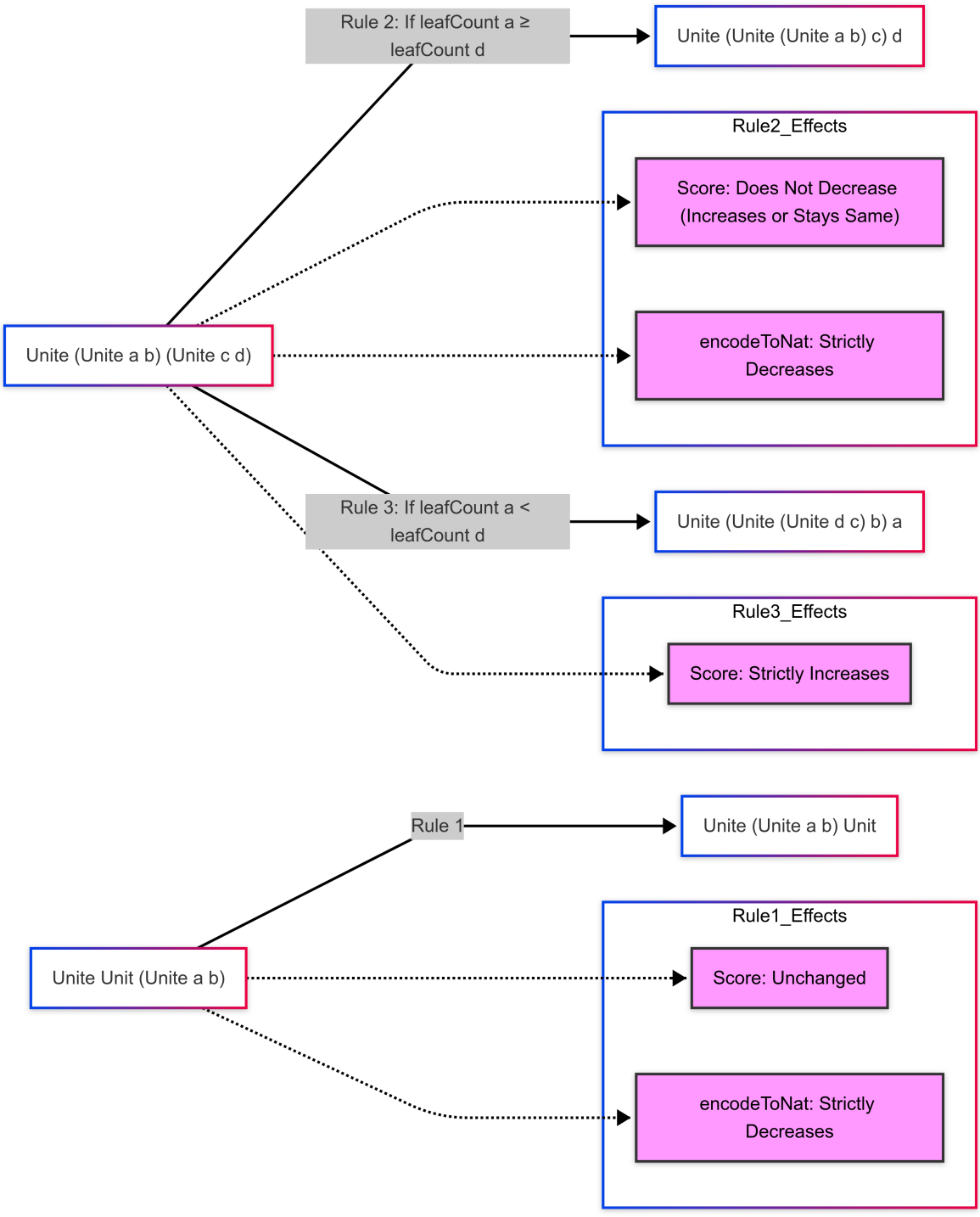


Figure 5.2: Rewrite rules for optimization

4. Termination of Rewriting (`turnIntoOptimalTree`) To prove that any optimal tree can be transformed into the `constructTree` form, the termination of this rewriting process is crucial.

- **Encoding:** Trees are encoded into binary strings (`encodeToList`) and then to natural numbers (`listToNat`, `encodeToNat`).
- **Termination Measure:** A well-founded induction is used with the measure: $2^{(2 \cdot \texttt{leafCount x} - 1)} \cdot (\texttt{leafCount x} \cdot \texttt{leafCount x} - \texttt{score x}) + \texttt{encodeToNat x}$.
  - $\texttt{score x} \leq (\texttt{leafCount x})^2$ is an established upper bound (`scoreUpperBound`).
  - For Rules 1 and 2, `encodeToNat x` decreases while `score x` does not decrease (or $\texttt{leafCount x} \cdot \texttt{leafCount x} - \texttt{score x}$ does not decrease). The overall measure decreases.
  - For Rule 3, `score x` strictly increases, causing $\texttt{leafCount x} \cdot \texttt{leafCount x} - \texttt{score x}$ to decrease. Even if `encodeToNat x` increases, the multiplicative factor $2^{(2 \cdot \texttt{leafCount x} - 1)}$ ensures the overall measure decreases.
- The `turnIntoOptimalTree` lemma formally proves that for any optimal tree `x`, `constructTree`(`leafCount x` $-$ 1) is also optimal. This relies on the fact that applying the rewrite rules maintains or improves optimality and eventually reaches the canonical `constructTree` form.
- `constructTreeIsOptimal (n : nat)`: This lemma finally asserts the optimality of the `constructTree` structure for any `n`.

Linking Abstract Model to Smart Contract Implementation

A significant portion of the proof involves demonstrating that the abstract tree model accurately reflects the smart contract's DSU operations.

1. State Representation (`Slot`, `convertToArray`)

- `Slot`: An inductive type to represent elements in the DSU array within the proof:

  ```
  Inductive Slot := | ReferTo (x : nat) | Ancestor (x : Tree).
  ```

  `ReferTo x` denotes a pointer to index `x`. `Ancestor x_tree` indicates a root element, where `x_tree` is the `Tree` structure representing the set rooted at this element.

- `convertToArray (x : list Slot) : list Z`: This function translates the proof's `list Slot` representation to a `list Z` (list of integers), mirroring the contract's DSU array.
  - A `ReferTo y` slot becomes `Z.of_nat y`.
  - An `Ancestor x_tree` slot becomes `(Z.sub 256 (Z.of_nat (leafCount x_tree)))`. This reflects how the contract stores the size of a set at its root: typically as a negative number. Given an 8-bit representation, this is equivalent to $256 - \text{size}$. The `nthConvert` lemma formalizes this correspondence.
  - Helper lemmas like `insertConvertReferTo` and `insertConvertAncestor` ensure that updates to `list Slot` correctly translate to updates in `list Z`.

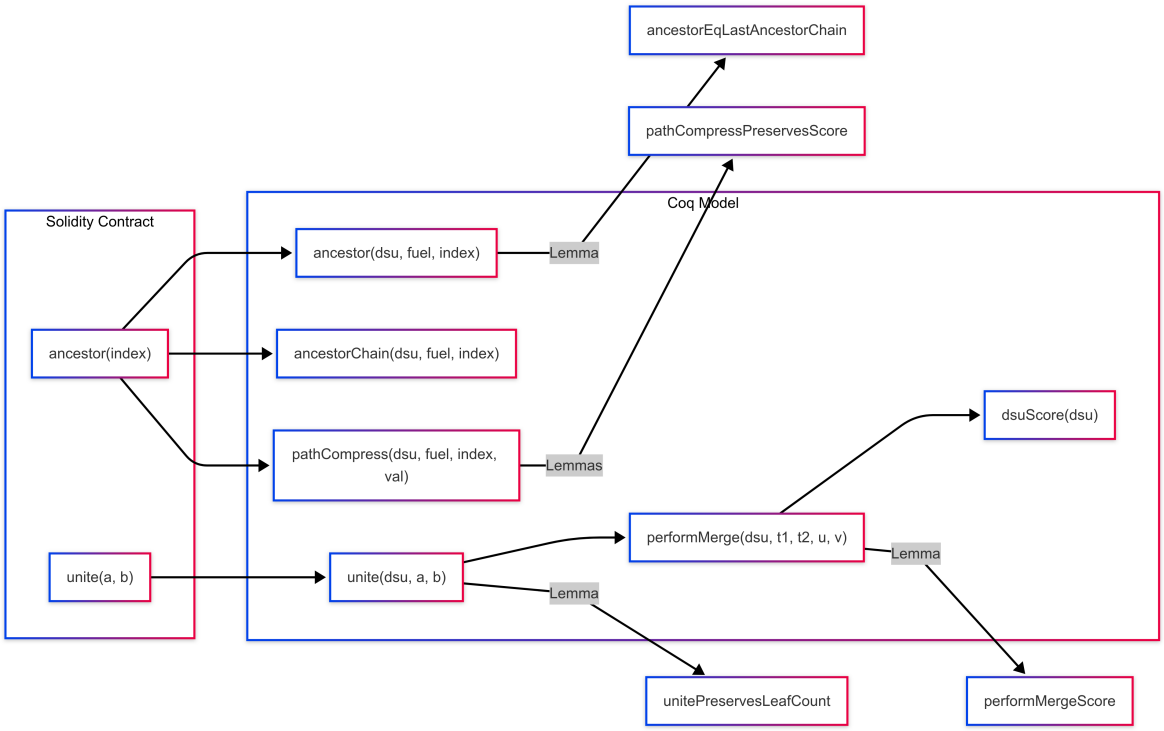


Figure 5.3: Abstract model and DSU contract implementation

2. DSU Invariants

- `noIllegalIndices dsu`: Ensures all `ReferTo` pointers are valid array indices (i.e., $y <$ `length dsu`).
- `withoutCyclesN dsu n`: Guarantees that following parent pointers from any element $x < n$ eventually leads to an `Ancestor` node, preventing cycles. This is also related to `withoutCyclesBool`.

3. Modeling Contract Functions

- **`ancestor` function:**
  - Coq's `ancestor dsu fuel index` mimics the contract's find operation.
  - `ancestorChain dsu fuel index` returns the list of nodes visited during an ancestor search.
  - Lemmas like `ancestorEqLastAncestorChain` connect these definitions.
- **Path Compression (`pathCompress`):**
  - Coq's `pathCompress dsu fuel index ancestor_val` models the contract's path compression logic.
  - `pathCompressPreservesLength`, `pathCompressPreservesScore`, and `pathCompress PreservesLeafCount` show it maintains key DSU properties.
  - Crucially, `pathCompressPreservesNoIllegalIndices` and `pathCompressPreserves WithoutCycles` ensure invariants are upheld. Proving the latter

involves lemmas about `validChain`, `validChainToAncestor`, and properties of `ancestorChain`, including `ancestorInsert` which reasons about how `ancestor` calls behave after an array modification.

- **`unite` function (Contract's Core Logic):**
  - Coq's `unite dsu a b` function models the entire union operation, including finding ancestors (with path compression) and merging.
  - It's proven that `unitePreservesLength`, `unitePreservesLeafCount`, `unitePreservesNoIllegalIndices`, and `unitePreservesWithoutCycles` hold.

4. Connecting to Generated Code (Framework Lemmas) The proof uses a framework for reasoning about code generated from the smart contract.

- `runCompressLoop`: This lemma (from `DisjointSetUnionCode2.v`) proves that the path compression loop in the generated code for the `ancestor` function correctly implements the `pathCompress` Coq function.
- `runAncestor`: This lemma proves that the entire `ancestor` procedure in the contract (which involves two loops: one for finding the root and one for path compression) behaves like the Coq `ancestor` function followed by `pathCompress`.
- `mergingLogic` / `mergingLogic3`: These lemmas connect the contract's unite operation steps (after ancestors are found and path compression is done) to the Coq `performMerge` function and `dsuScore` update. It shows that the contract correctly updates the DSU array and the total wei withdrawn matches the increase in `dsuScore`.
- `runUnite` / `runUnite3`: These lemmas prove that a full call to the contract's `unite` function (which includes calls to `ancestor`) corresponds to the behavior of the Coq `unite` function in terms of DSU state and score.

5. Integer Representation and Overflow The contract uses 8-bit integers. Root nodes store the negative of their set size. `coerceInt val 8` and `toSigned val 8` are used in proofs to model this. An important aspect of `mergingLogic` is proving that adding two (negative) set sizes does not cause an underflow that would change a root node into a non-root node (pointer). Since the maximum sum of leaf counts is 100, the sum of two root values (represented as $-(\text{size}_1)$ and $-(\text{size}_2)$) will be at least $-(100)$. For an `int8` type (range -128 to 127), this addition (e.g., $(256 - \text{size}_1) + (256 - \text{size}_2) \pmod{256}$) correctly results in $256 - (\text{size}_1 + \text{size}_2)$, which is the correct representation for the new root, without underflowing to become a positive pointer value. This is crucial for correctness. The `sumTwoAncestors` lemma shows `leafCount u` + `leafCount v` $\leq$ `Z.to_nat`(`dsuLeafCount dsu`), bounding the sum of sizes.

DSU Score and Total Withdrawal

The proof culminates by relating the abstract DSU score to the total wei withdrawn.

1. **DSU-wide Score and Leaf Count**

   - `dsuScore (dsu : list Slot)`: Sum of `score` for all `Ancestor` trees in the
   - `dsu_list`.
   - `dsuLeafCount (dsu : list Slot)`: Sum of `leafCount` for all `Ancestor` trees.
   - `dsuLeafCount` is invariant and equals DSU size (100).

2. **Bounding the DSU Score**

   - `roll (dsu_list : list Slot) : option Tree`: An auxiliary function that conceptually combines all trees in the DSU into a single tree.
     - `rollPreservesLeafCount` shows `Z.to_nat(dsuLeafCount dsu_list)` equals `leafCount (roll dsu_list)` if not `None`.
   - `maxScore (dsu_list : list Slot)`: Lemma stating `Z.to_nat(dsuScore dsu_list) <= score (default Unit (roll dsu_list))`.
   - `maxScore3 (dsu_list : list Slot)`: This key lemma bounds the total DSU score: `dsuScore dsu_list <= (dsuLeafCount dsu_list) * (dsuLeafCount dsu_list + 1) / 2 - 1`.
   - Since `dsuLeafCount` is 100, `dsuScore` is 100 * 101 / 2 - 1 = 5049

3. **Score Increase on Merge (`performMergeScore`)**

   - `performMerge dsu t1 t2 u v`: Models setting `u` to point to `v`, and `v` to become `Ancestor (Unite t2 t1)`.
   - The `performMergeScore` lemma is critical:

```
(dsuScore (performMerge dsu t1 t2 u v) = dsuScore dsu + Z.of_nat (
    leafCount t1) + Z.of_nat (leafCount t2))%Z
```

   This shows the `dsuScore` increases by the sum of the leaf counts of the merged trees, which is exactly the amount of wei transferred by the contract (size of the newly merged set).

4. **Blockchain State and Interaction Modeling**

   - `state`: Defines the initial blockchain state, including the contract with 100,000 wei and an empty DSU (implicitly, as it's initialized on first interaction or represented by `repeat (Ancestor Unit) 100` initially in proofs).
   - `stateAfterInteractions arrays money`: Defines the state after some interactions, tracking the contract's DSU array and the money withdrawn (which accumulates in an EOA).
   - `interact state interactions_list`: Simulates a list of `(u, v)` interactions with the contract, returning the total money withdrawn.

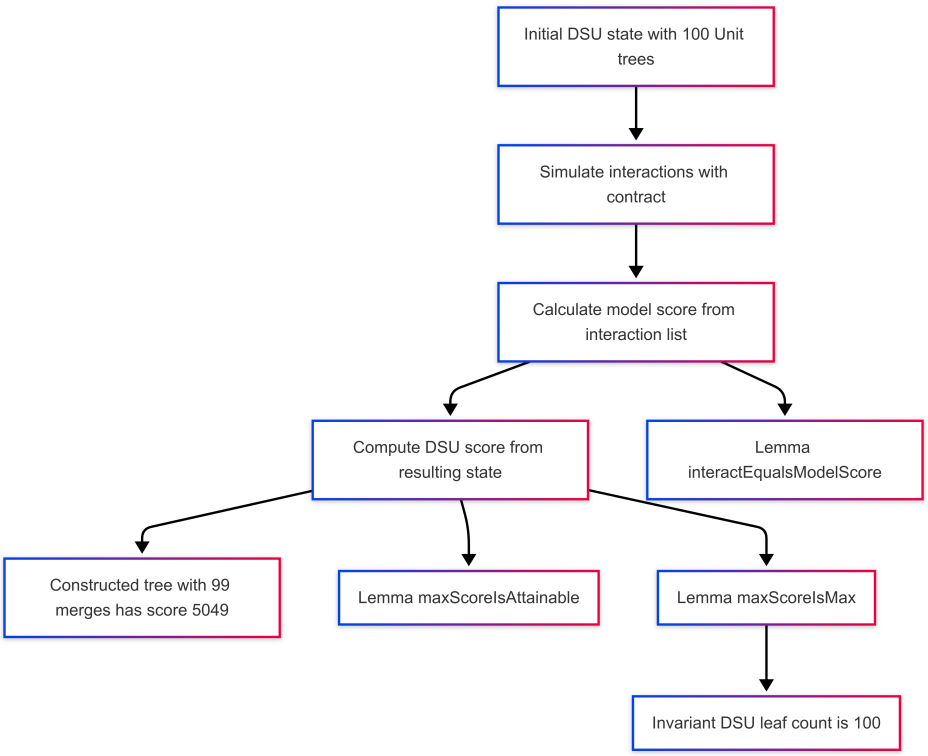


Figure 5.4: Establishing the upper bound of 5049

- `modelScore interactions_list`: Calculates the `dsuScore` after the same sequence of interactions applied to the abstract `list Slot` model, starting from `repeat (Ancestor Unit) 100`.

- `interactEqualsModelScore`: This lemma proves that the actual wei withdrawn by interacting with the contract (`interact`) equals the score calculated by the abstract model (`modelScore`). This is a crucial bridge between the contract execution and the abstract DSU scoring.

Final Conclusion: Maximum Withdrawal is 5049 Wei

The final lemmas in `DisjointSetUnionCode3.txt` tie everything together:

- **`maxScoreIsAttainable`**: Demonstrates that a score of 5049 is indeed achievable by providing a specific sequence of interactions (merging elements $1, \ldots, 99$ sequentially, which corresponds to the `constructTree` strategy).

```
Lemma maxScoreIsAttainable : interact state (map (fun x => (0%Z, Z.
    of_nat x)) (seq 1 99)) = 5049%Z.
```

- **`maxScoreIsMax`**: Proves that for any sequence of valid interactions, the total wei withdrawn (`interact state x`) is less than or equal to 5049.

```
Lemma maxScoreIsMax (x : list (Z * Z)) (hN : forall a b, In (a, b) x
    -> Z.le 0 a /\ Z.lt a 256 /\ Z.le 0 b /\ Z.lt b 256) : (interact
    state x <= 5049)%Z.
```

The proof of `maxScoreIsMax` relies on:

1. `interactEqualsModelScore`: Equating contract interaction result with the model's score.
2. `maxScore3`: The upper bound on `dsuScore`.
3. An assertion that `dsuLeafCount nx = 100` after any sequence of interactions, where `nx` is the DSU state derived from interactions. This is proven by induction on the list of interactions, using lemmas like `uniteP-reservesLeafCount`.

These lemmas collectively establish that 5049 wei is the provable maximum withdrawal amount. The proof also handles out-of-bounds interactions, ensuring they don't lead to withdrawals or violate DSU invariants (`outOfBoundsInteractionNAC`, `outOfBound-sInteractionNBC`, etc.

Rationale for the DSU Proof Strategy

The verification of the Disjoint Set Union (DSU) contract, particularly its maximum withdrawal property, employs a multi-layered proof strategy. This approach is chosen for its effectiveness in bridging the gap between a high-level conceptual understanding of the DSU mechanism and the concrete, low-level behavior of the smart contract implementation. The rationale behind this strategy is as follows:

- **Why the Abstract Tree Model?** The DSU structure, with its sets and union operations, naturally maps to a forest of trees, where each tree represents a set and union operations correspond to linking trees. Modeling interactions as the construction and scoring of a single abstract `Tree` (representing the history of merges) allows for a clear and mathematically tractable definition of the "score" or total wei withdrawn. This abstraction simplifies reasoning about the core problem: maximizing this score under the rules of DSU operations. It allows us to analyze the profit generation mechanism independently of implementation details.

- **Why Optimal Tree Construction and Rewrite Rules for Maximization?** To prove a maximum withdrawal amount, it's not enough to show one profitable scenario; we must demonstrate that no other scenario yields more. The strategy of defining an `constructTree` function that builds a tree in a specific, regular way (always merging with a new singleton set) provides a candidate for this optimal structure. The core of the maximization proof lies in demonstrating that any valid DSU interaction sequence (any tree) can be transformed into this canonical `constructTree` form using a set of score-preserving or score-increasing rewrite rules. The termination of this rewriting process, proven using a well-founded measure (`encodeToNat` combined with the score), is crucial to guarantee that the optimal `constructTree` form is always reachable and that its score represents the maximum. This is a robust method for proving optimality.

- **Why Link the Abstract Model to the Concrete Contract Implementation?** A proof solely at the abstract tree level would not guarantee the correctness of the actual smart contract. Therefore, a significant portion of the effort is dedicated to rigorously linking this abstract model to the contract's implementation details as compiled into Coq. This involves:

- **State Representation Mapping:** Defining a `Slot` type and a `convertToArray` function to formally relate the abstract tree structures (specifically, `Ancestor` nodes storing `Tree`s) to the contract's array-based representation in `list Z`. This ensures that the state manipulated by the abstract model corresponds faithfully to the state manipulated by the contract code.
- **Proving DSU Invariants:** Establishing and proving crucial DSU invariants for the contract's state, such as `noIllegalIndices` and `withoutCycles`, ensures the integrity of the DSU structure throughout any operation. These invariants are fundamental to the correct functioning of the DSU algorithm itself.
- **Modeling Contract Functions:** Formalizing the behavior of the contract's procedures (`ancestor`, `pathCompress`, `unite`) within Coq and proving that these Coq functions accurately model the execution of the compiled contract code (e.g., via lemmas like `runCompressLoop`, `runAncestor`, `mergingLogic`, and `runUnite`). This step uses the framework's monadic infrastructure and `invokeContractAux` simulator to reason about the step-by-step execution of contract instructions. Special attention is given to integer representation and potential overflows/underflows to ensure arithmetic correctness within the contract's 8-bit integer constraints.

- **How This Strategy Achieves Comprehensive Verification:** This layered strategy effectively dissects the verification challenge. First, it solves the core optimization problem (maximum score) in an abstract, mathematically convenient domain (trees). Then, it meticulously demonstrates that the smart contract's behavior, when executed according to the framework's formal semantics, precisely implements this abstract model and adheres to its constraints and properties. The final lemmas, `interactEqualsModelScore`, `maxScoreIsAttainable`, and `maxScoreIsMax`, bring these layers together to conclude that the proven maximum score of the abstract model is indeed the maximum wei that can be withdrawn from the actual smart contract. This approach ensures that the verification is not superficial but deeply tied to both the algorithmic logic and the faithful implementation thereof. The Coq proof assistant is indispensable in managing the complexity of these definitions, inductive proofs, and numerous case analyses involved in such a comprehensive verification.

5.2 Knapsack contract

Related files

1. `https://github.com/huynhtrankhanh/CoqCP/blob/main/theories/Knapsack.v`
2. `https://github.com/huynhtrankhanh/CoqCP/blob/main/theories/KnapsackCode.v`
3. `https://github.com/huynhtrankhanh/CoqCP/blob/main/theories/KnapsackCode2.v`
4. `https://github.com/huynhtrankhanh/CoqCP/blob/main/programs/Knapsack.js`

5. https://github.com/huynhtrankhanh/CoqCP/blob/main/programs/Knapsack.module.json

This is the source code of the contract.

```
environment({
  dp: array([int64], 1000000),
  message: array([int32], 1),
  n: array([int64], 1),
})

procedure('get weight', { index: int64 }, () => {
  store('message', 0, [
    coerceInt32(retrieve(4 * get('index'))) * coerceInt32(1 << 24) +
      coerceInt32(retrieve(4 * get('index') + 1)) * coerceInt32(1 << 16) +
      coerceInt32(retrieve(4 * get('index') + 2)) * coerceInt32(1 << 8) +
      coerceInt32(retrieve(4 * get('index') + 3)),
  ])
})

procedure('get value', { index: int64 }, () => {
  store('message', 0, [
    coerceInt32(retrieve(4 * retrieve('n', 0)[0] + 4 * get('index'))) *
      coerceInt32(1 << 24) +
      coerceInt32(retrieve(4 * retrieve('n', 0)[0] + 4 * get('index') + 1)) *
        coerceInt32(1 << 16) +
      coerceInt32(retrieve(4 * retrieve('n', 0)[0] + 4 * get('index') + 2)) *
        coerceInt32(1 << 8) +
      coerceInt32(retrieve(4 * retrieve('n', 0)[0] + 4 * get('index') + 3)),
  ])
})

procedure('store result', { x: int64 }, () => {
  store(0, coerceInt8(get('x') >> 56))
  store(1, coerceInt8((get('x') >> 48) & 255))
  store(2, coerceInt8((get('x') >> 40) & 255))
  store(3, coerceInt8((get('x') >> 32) & 255))
  store(4, coerceInt8((get('x') >> 24) & 255))
  store(5, coerceInt8((get('x') >> 16) & 255))
  store(6, coerceInt8((get('x') >> 8) & 255))
  store(7, coerceInt8(get('x') & 255))
})

procedure('get limit', {}, () => {
  store('message', 0, [
    coerceInt32(retrieve(8 * retrieve('n', 0)[0])) * coerceInt32(1 << 24) +
      coerceInt32(retrieve(8 * retrieve('n', 0)[0] + 1)) *
        coerceInt32(1 << 16) +
      coerceInt32(retrieve(8 * retrieve('n', 0)[0] + 2)) * coerceInt32(1 << 8) +
      coerceInt32(retrieve(8 * retrieve('n', 0)[0] + 3)),
  ])
})

procedure('main', { limit: int64, weight: int64, value: int64 }, () => {
```

```
  store('n', 0, [divide(communicationSize() - 4, 8)])
  call('get limit', {})
  set('limit', coerceInt64(retrieve('message', 0)[0]))
  store('message', 0, [coerceInt32(0)])
  range(retrieve('n', 0)[0] + 1, (i) => {
    if (i == 0) {
      ;('continue')
    }
    range(get('limit') + 1, (cap) => {
      call('get weight', { index: i - 1 })
      set('weight', coerceInt64(retrieve('message', 0)[0]))
      call('get value', { index: i - 1 })
      set('value', coerceInt64(retrieve('message', 0)[0]))
      if (less(cap, get('weight'))) {
        store('dp', i * (get('limit') + 1) + cap, [
          retrieve('dp', (i - 1) * (get('limit') + 1) + cap)[0],
        ])
      } else {
        if (
          less(
            retrieve('dp', (i - 1) * (get('limit') + 1) + cap)[0],
            retrieve(
              'dp',
              (i - 1) * (get('limit') + 1) + (cap - get('weight'))
            )[0] + get('value')
          )
        ) {
          store('dp', i * (get('limit') + 1) + cap, [
            retrieve(
              'dp',
              (i - 1) * (get('limit') + 1) + (cap - get('weight'))
            )[0] + get('value'),
          ])
        } else {
          store('dp', i * (get('limit') + 1) + cap, [
            retrieve('dp', (i - 1) * (get('limit') + 1) + cap)[0],
          ])
        }
      }
      set('weight', 0)
      set('value', 0)
    })
    store('message', 0, [coerceInt32(0)])
  })
  call('store result', {
    x: retrieve(
      'dp',
      retrieve('n', 0)[0] * (get('limit') + 1) + get('limit')
    )[0],
  })
})
```

Listing 5.3: Knapsack contract

The main correctness claim of the contract is **it computes the maximum total value** that can be obtained by selecting items from the list such that their total weight does not exceed the limit.

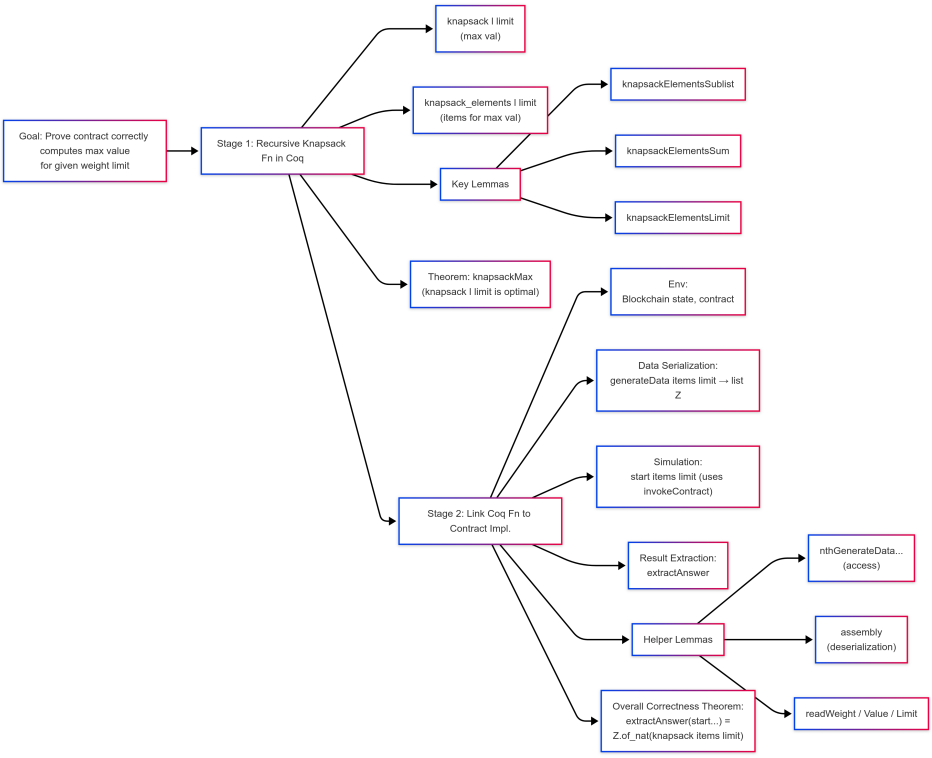


Figure 5.5: Overall strategy for knapsack contract

5.2.1 Inspection of Contract's Correctness

Before moving on to the proof details, we first go into how the framework and Coq's logical infrastructure enables the end user to quickly check whether the contract is logically correct or not, without having to trust the person who wrote the correctness proof.

1. **Check for Compilation Errors:** Run the framework's build scripts, including the compilation pipeline to Solidity and Coq and Coq's verifier. If there are no errors, we can proceed with the next step.

2. **Check What the Proof Proves:** Read the theorem statements and the definitions. There are very few things that the user needs to read compared to the size of the proof. In the case of this smart contract, here are the key claims:

```
1 Lemma knapsackMax (l : list (nat * nat)) (limit : nat) : isMaximum (
    knapsack l limit) (fun x => exists choice, sublist choice l /\
    fold_right (fun x acc => snd x + acc) 0 choice = x /\ fold_right
    (fun x acc => fst x + acc) 0 choice <= limit).
2
3 Lemma extractAnswerEq (items : list (nat * nat)) (notNil : items <>
    []) (limit : nat) (hp : ((limit + 1%nat) * (length items + 1%nat)
     <= 1000000%nat)%nat) (a32 : forall x, (fst (nth x items (0%nat
    ,0%nat)) < 2^32)%nat) (b32 : forall x, (snd (nth x items (0%nat
    ,0%nat)) < 2^32)%nat) : extractAnswer (start items limit) = Z.
    of_nat (knapsack items limit).
4
```

Listing 5.4: Key claims of the correctness proof of the knapsack contract

The first claim, `knapsackMax`, says that the `knapsack` function, defined somewhere in the proof, correctly computes the maximum value. The first claim doesn't talk about the code of the smart contract.

The second claim, `extractAnswerEq`, ties that `knapsack` function to the actual source code of the smart contract. It says that the source code of the smart contract implements the `knapsack` function in the proof.

These two claims, taken together, mean that the smart contract correctly computes the maximum value that can be obtained from picking items.

3. **Conclusion:** Absent bugs in the framework, when the framework says that everything is logically consistent, and the key claims match what the user wants from the smart contract, the user can trust the smart contract is correct.

5.2.2 Proof Details

The proof is split into two stages. In the first stage, we model a recursive knapsack function in Coq, and prove that the function is correct. In the second stage, we connect the knapsack function to the actual smart contract implementation.

Defining the Knapsack Function

`knapsack` This function calculates the maximum value that can be obtained from a list of items (pairs of weight and value) given a weight limit.

```
Fixpoint knapsack (l : list (nat * nat)) (limit : nat) :=
  match l with
  | [] => 0
  | (weight, value) :: tail =>
    if decide (limit < weight) then knapsack tail limit
    else knapsack tail limit `max` (value + knapsack tail (limit - weight))
  end.
```

Idea:

- **Base Case:** If the list of items `l` is empty, the value is 0.
- **Recursive Step:** For an item `(weight, value)`:
  - If the current `limit` is less than the item's `weight` (i.e., `limit < weight`), the item cannot be included. The function recurses on the `tail` of the list with the same `limit`.
  - Otherwise, there are two choices:
    1. **Exclude the item:** The value is `knapsack tail limit`.
    2. **Include the item:** The value is `value + knapsack tail (limit - weight)`.

    The function takes the maximum (`max`) of these two options.

knapsack_elements This function returns the list of items that achieve the maximum value calculated by the knapsack function.

```
Fixpoint knapsack_elements (l : list (nat * nat)) (limit : nat) : list (nat * nat) :=
  match l with
  | [] => []
  | (weight, value) :: tail =>
    if decide (limit < weight) then knapsack_elements tail limit
    else
      let without := knapsack_elements tail limit in
      let with_item := (weight, value) :: knapsack_elements tail (limit - weight) in
      if decide ((fold_right (fun x acc => snd x + acc) 0 without) <
                 (fold_right (fun x acc => snd x + acc) 0 with_item))
      then with_item
      else without
  end.
```

Idea:

- **Base Case:** If `l` is empty, the list of chosen items is empty.
- **Recursive Step:** For an item `(weight, value)`:
  - If `limit < weight`, the item cannot be included. Recurse on `tail` with the same `limit`.
  - Otherwise:
    1. `without`: Recursively find the best set of items from `tail` without the current item.
    2. `with_item`: Recursively find the best set of items from `tail` assuming the current item is taken (and thus `limit` is reduced by `weight`), then prepend the current item `(weight, value)` to this set.
    3. Compare the total value of `without` and `with_item` (using `fold_right` to sum the `snd` components, which are values). Return the list that yields a higher total value.

The proof details of the knapsack recursive function are available in section A.3 in the appendix. Refer to the diagrams from figure 5.6 onwards for a summary of the steps of the proof.

Tying the Knapsack Function to the Actual Implementation

1. Defining the Environment We now need to define:

- The structure of the input data required by the knapsack contract function.
- The initial state of the blockchain and the contract.
- How to simulate an invocation of the contract.
- How to interpret the result returned by the contract.
- Lemmas proving basic properties (specifically data length) about the serialization process.

Figure 5.6: Proving sublist property

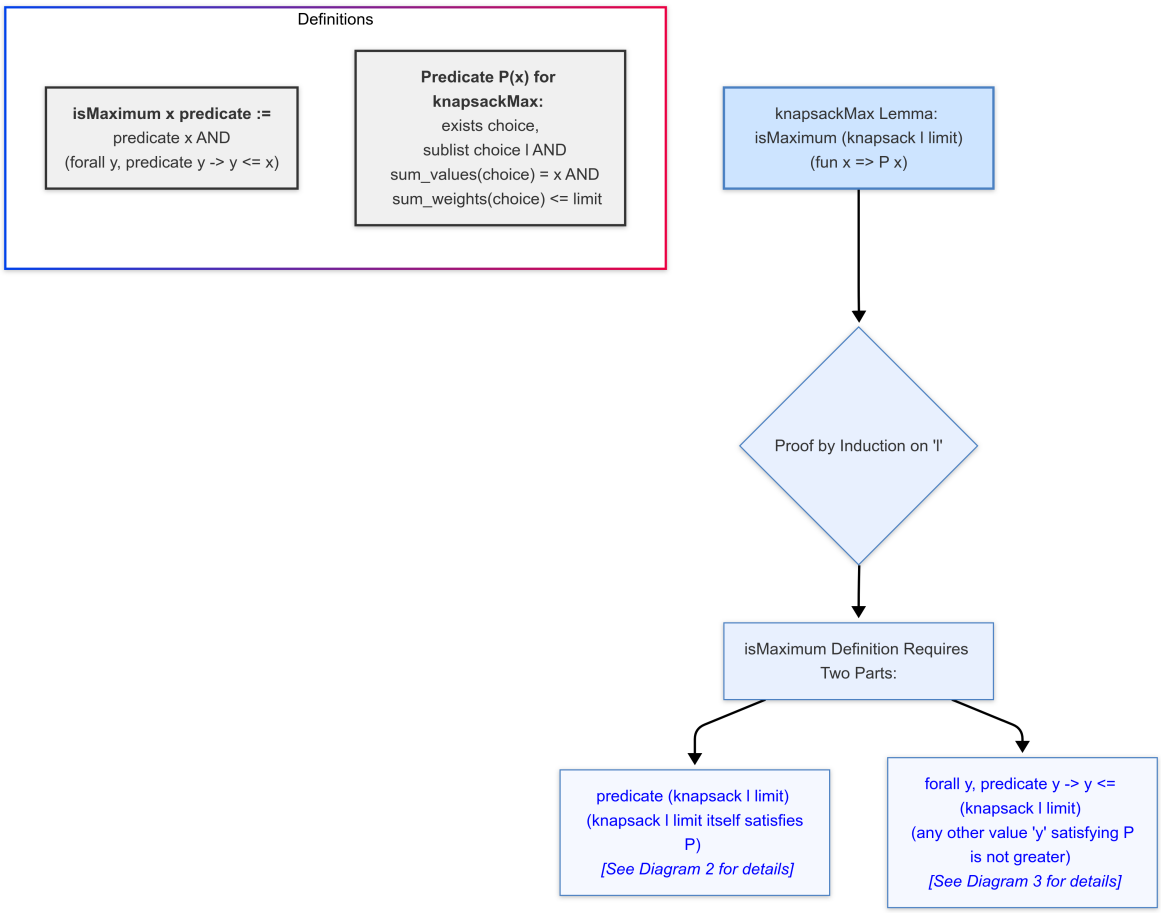


Figure 5.7: Building the definition

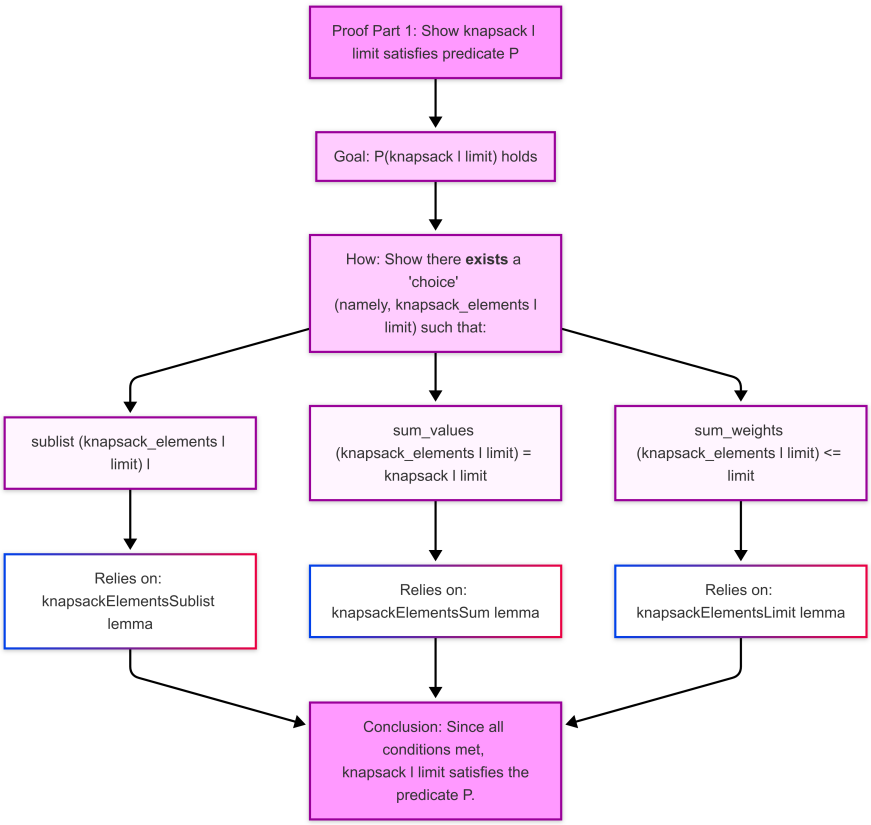


Figure 5.8: Function satisfies the predicate (Diagram 2)

2. Blockchain State Definitions `state`: This definition describes the initial state of the blockchain before any interaction with the knapsack contract. It specifies:

- A contract deployed at the address `0x0...0` (20 zero bytes). This contract has an initial balance of 100000 units, associated storage (`arrays _ environment0`), and its code is defined by `funcdef_0__main`.
- All other addresses are initially Externally Owned Accounts (EOAs) with a balance of 0.

3. Data Serialization The code defines functions to serialize the knapsack problem parameters (items and capacity limit) into a list of integers (`list Z`), suitable for use as calldata in a smart contract invocation. `Z` represents arbitrary-precision integers.

- `to32 (x : nat)`: This helper function converts a natural number (`nat`) `x` into a list of four `Z` integers. It effectively breaks down the number into four 8-bit chunks (bytes), representing a 32-bit unsigned integer encoding (big-endian).
  - `Lemma to32Length`: Proves that the output list of `to32` always has a length of 4. The proof is straightforward by unfolding the definition.
- `serializeWeights (items : list (nat * nat))`: This function takes a list of items, where each item is a pair `(weight, value)`, and recursively serializes *only* the weights. It applies `to32` to each weight and concatenates the resulting 4-byte lists.

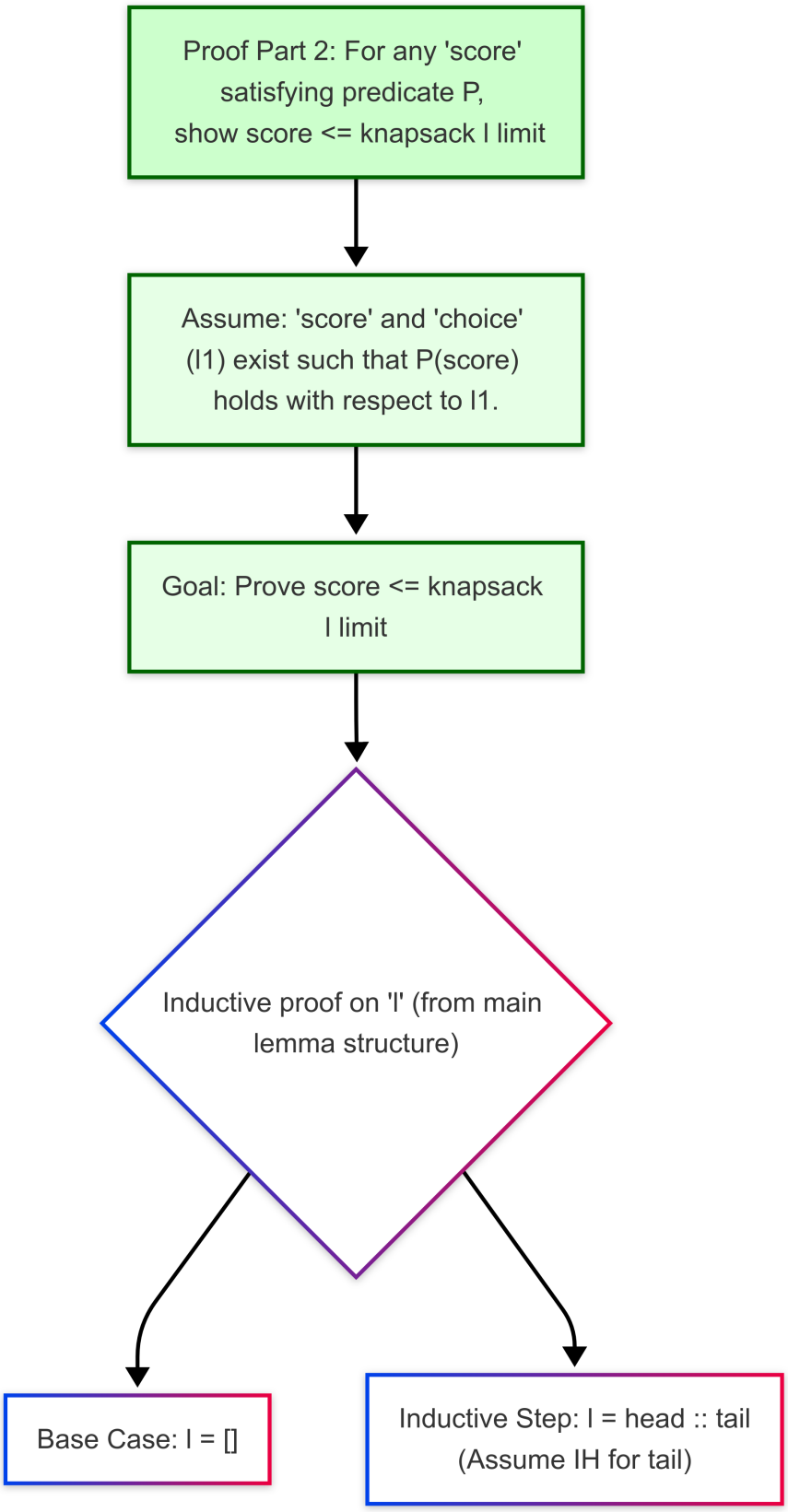


Figure 5.9: Overview of inductive proof (Diagram 3)

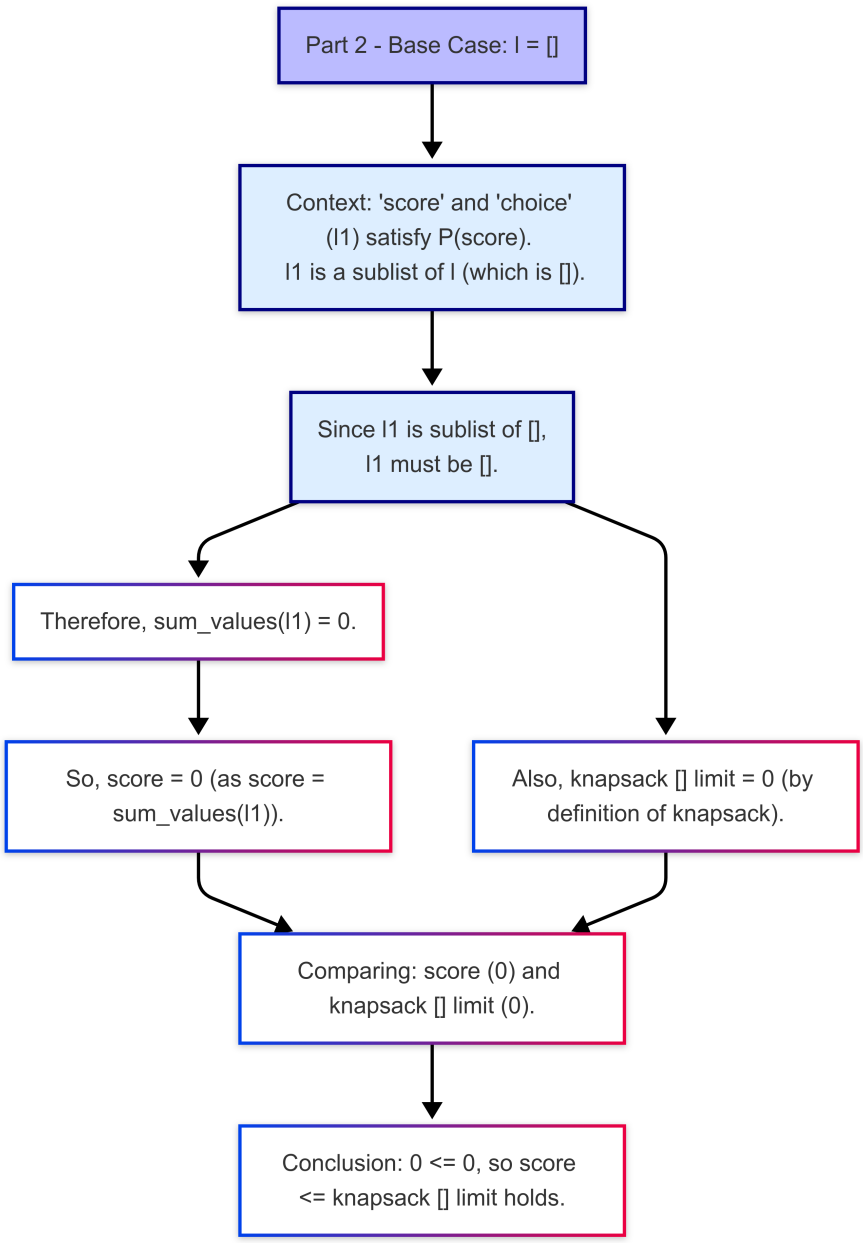


Figure 5.10: Base case (Diagram 4)

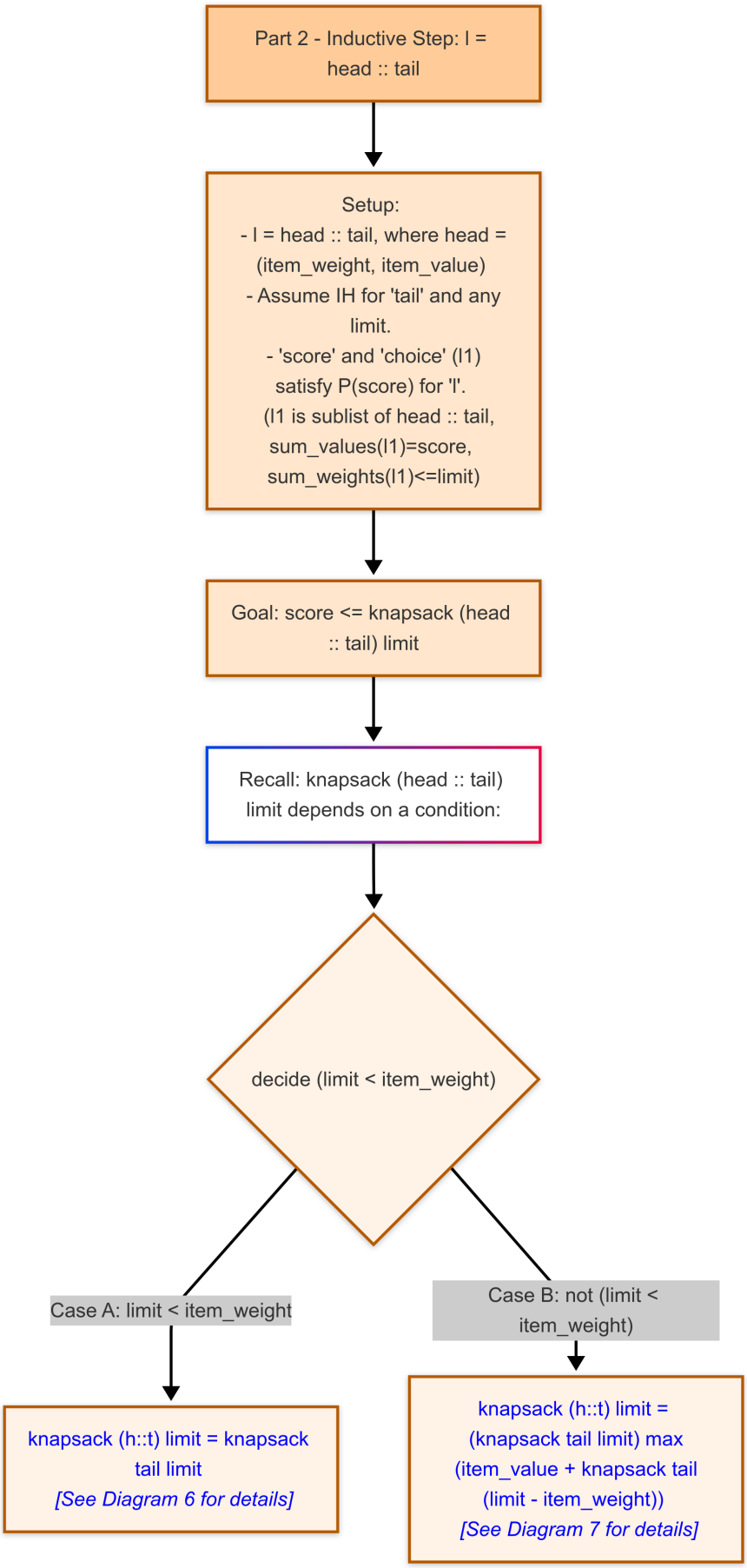


Figure 5.11: Inductive step (Diagram 5)

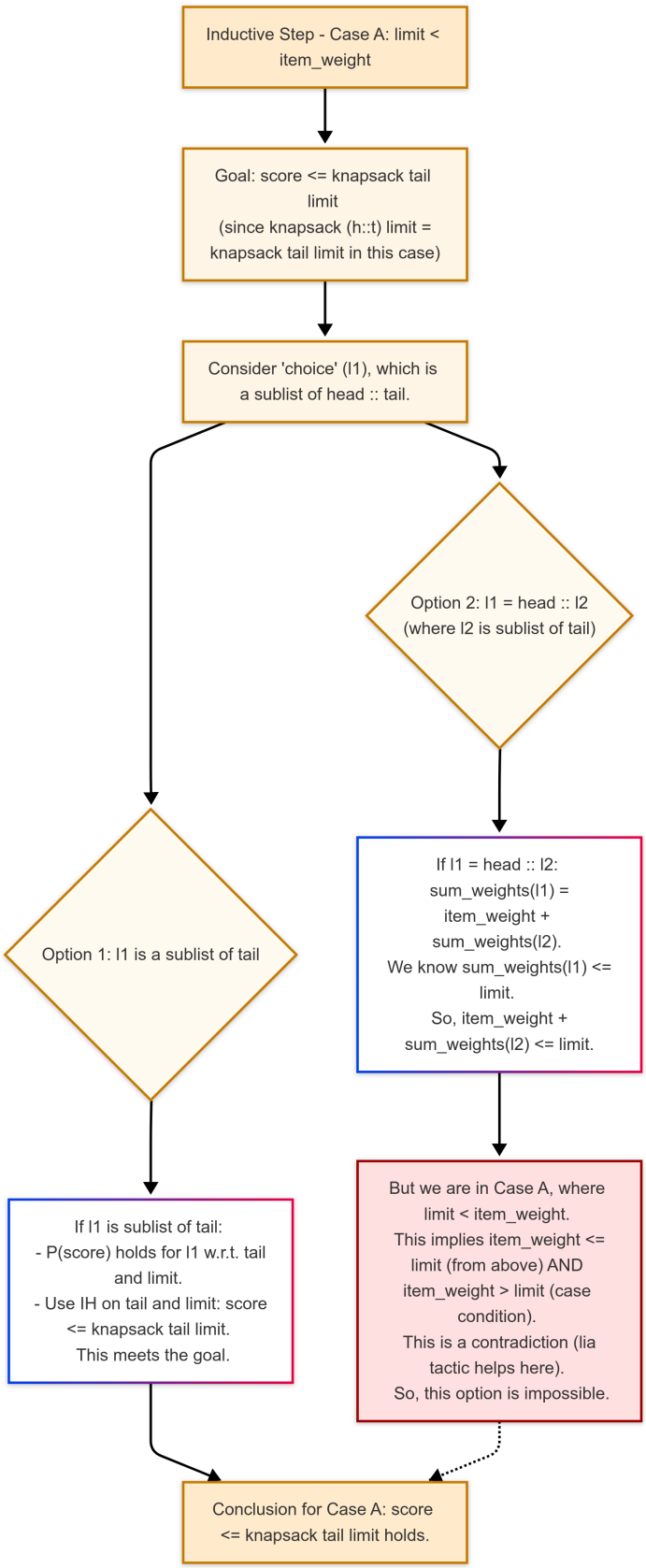


Figure 5.12: Case A (Diagram 6)

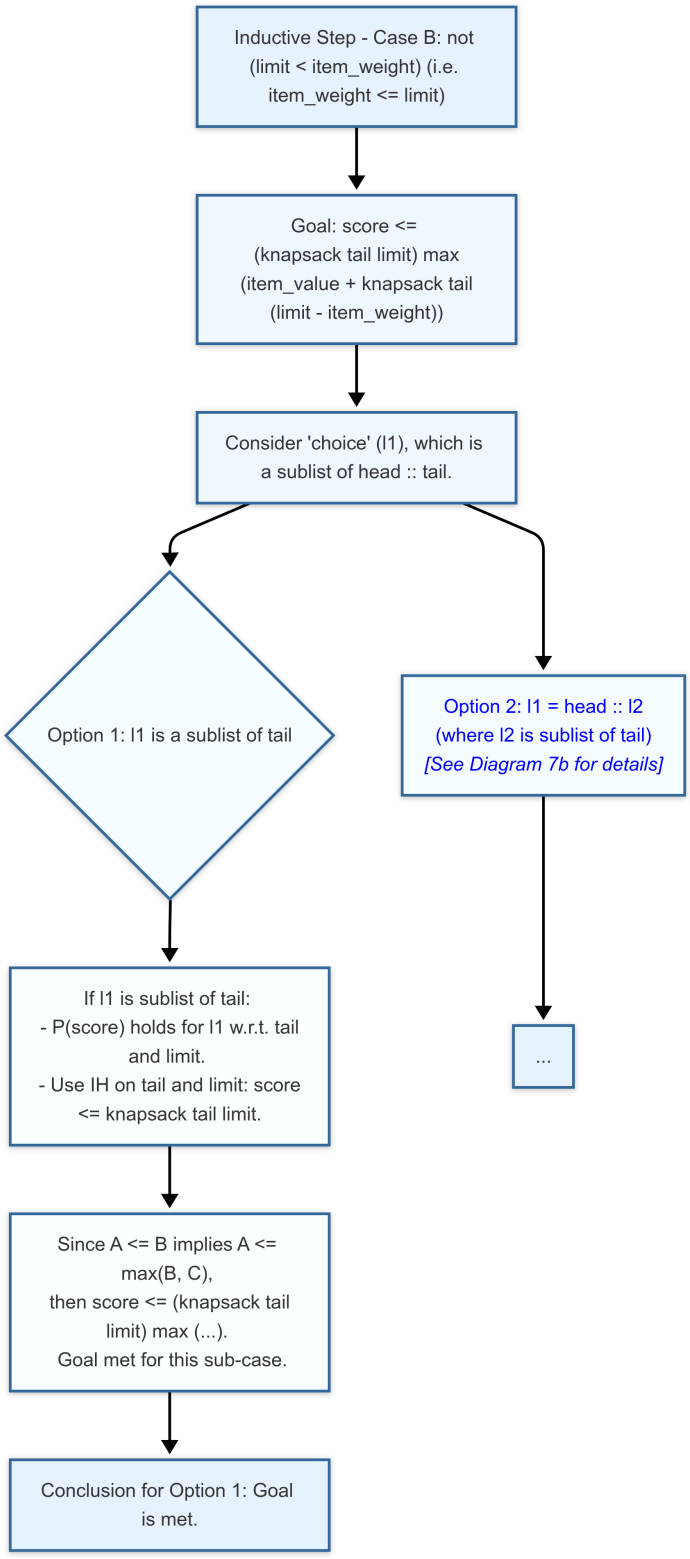


Figure 5.13: Case B (Diagram 7)

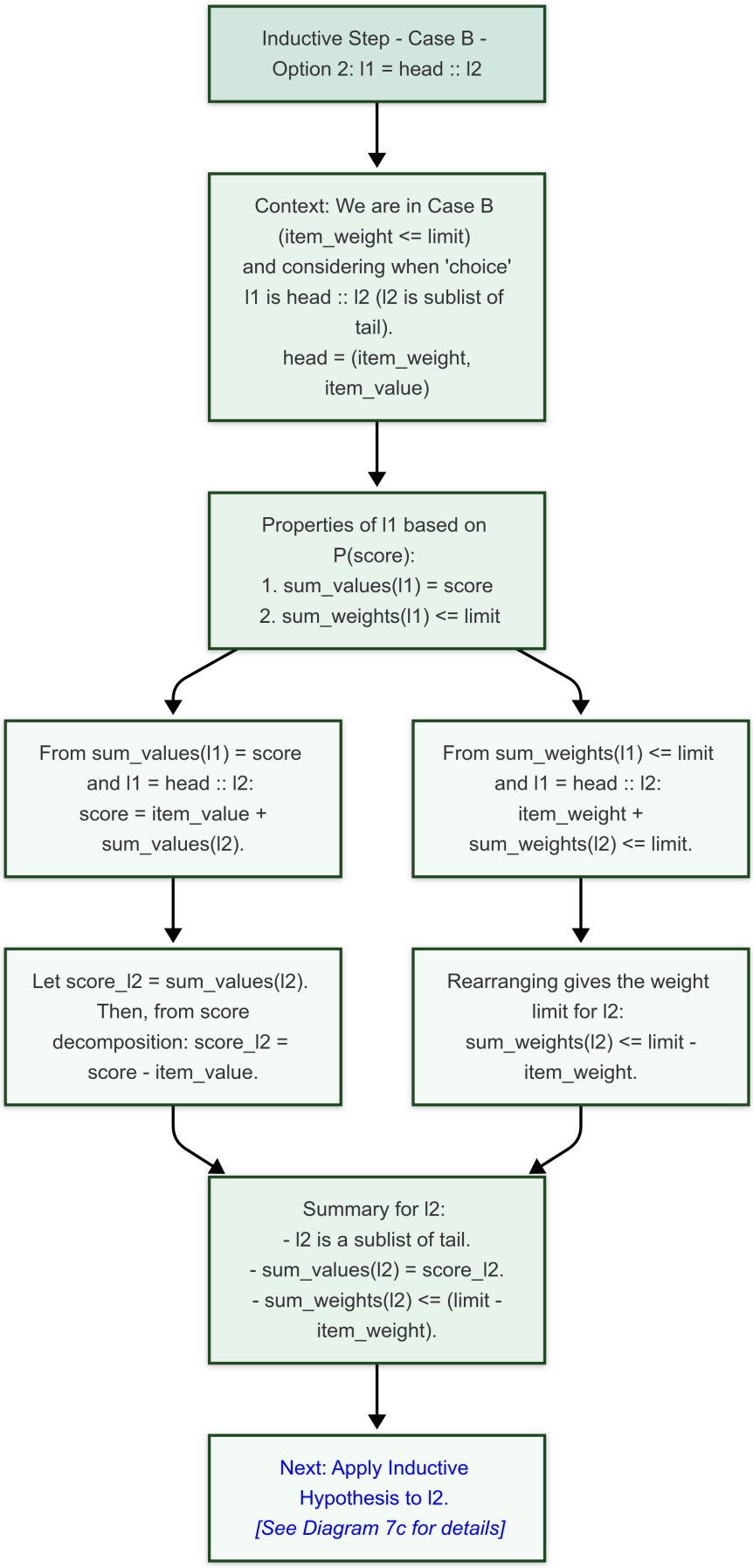


Figure 5.14: Case B (Diagram 7b)

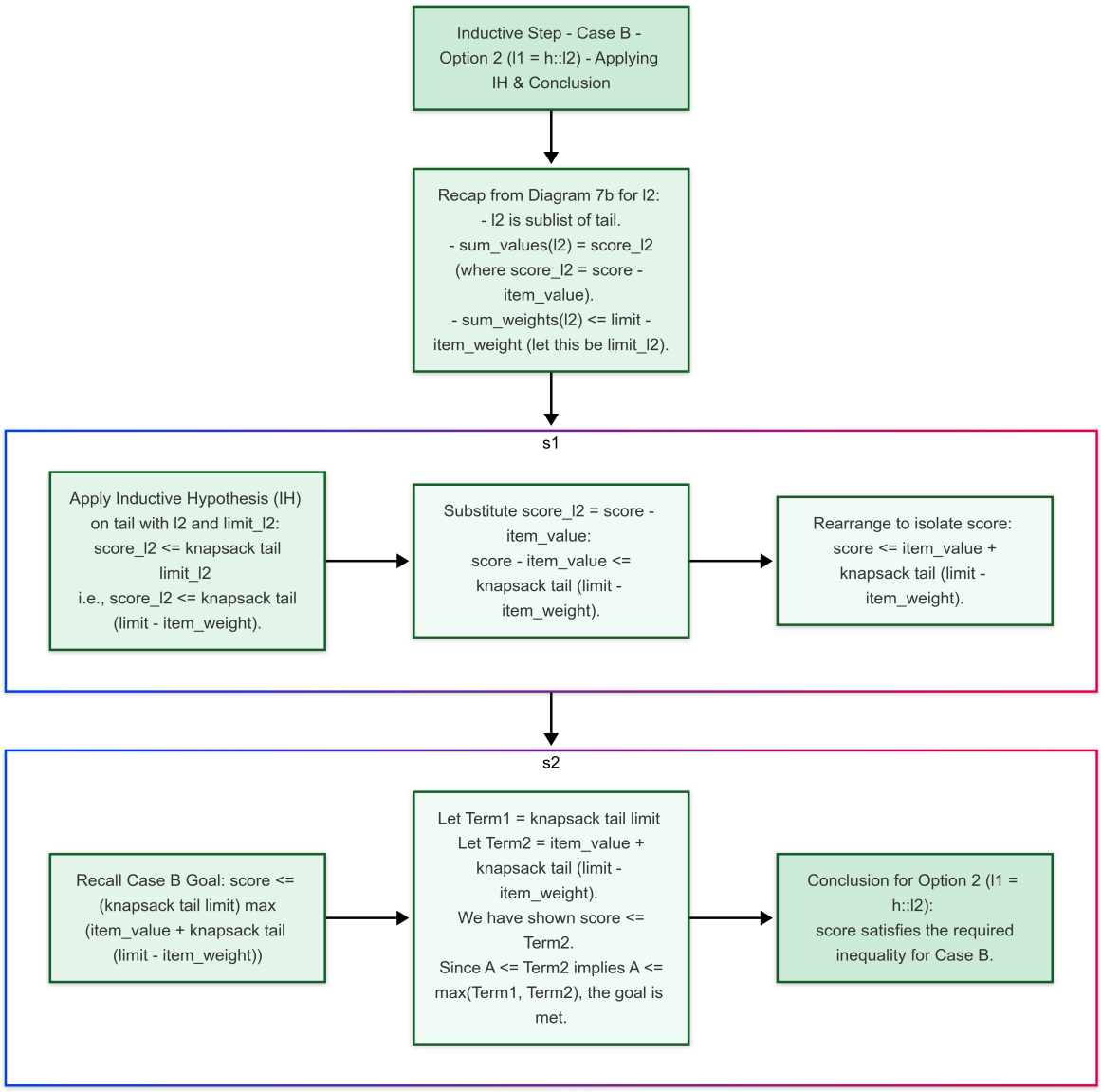


Figure 5.15: Case B (Diagram 7c)

- Lemma serializeWeightsLength: Proves by induction on the items list that the length of the serialized weights list is exactly $4 \times$ length items.

- serializeValues (items : list (nat * nat)): Similar to serializeWeights, but it serializes *only* the values from the items list using to32 for each.

  - Lemma serializeValuesLength: Proves by induction that the length of the serialized values list is also $4 \times$ length items.

- generateData (items : list (nat * nat)) (limit : nat): This function constructs the final calldata payload. It concatenates the serialized weights, the serialized values, and the serialized capacity limit (using to32).

  - Lemma dataLength: Proves that the total length of the generated data is $(4 \times \texttt{length items}) + (4 \times \texttt{length items}) + 4 = 8 \times \texttt{length items} + 4$. The proof uses the previous length lemmas and properties of list concatenation and arithmetic (lia).

4. Contract Invocation Simulation

- start (items : list (nat * nat)) (limit : nat): This definition simulates calling the smart contract. It uses the invokeContract function (presumably from CoqCP).

  - Caller: 0x1...1 (the EOA address).
  - Contract Address: 0x0...0.
  - Value Sent: 0 (no Ether/currency sent with the call).
  - Initial State: Uses the predefined state.
  - Input Data: Uses generateData items limit to provide the serialized knapsack problem instance.

  The result of invokeContract is an option (list Z * BlockchainState), representing either failure (None) or success (Some) with the return data and the final blockchain state.

5. Result Extraction

- extractAnswer (x : option (list Z * BlockchainState)): This function decodes the result returned by the simulated contract call x.

  - If x is Some (answer, _) (successful call), it takes the returned data answer (a list Z) and reconstructs a single Z integer. It assumes the contract returns an 8-byte (64-bit) integer, encoded similarly to to32 but across 8 elements, and reconstructs it using powers of 2 (effectively treating answer as bytes of a big-endian 64-bit number). It uses nth 0 l d to safely access list elements, returning default 0 if the index is out of bounds. The reconstruction is equivalent to:
    $\texttt{nth}\ 0\ \texttt{answer} \times 2^{56} + \texttt{nth}\ 1\ \texttt{answer} \times 2^{48} + \texttt{nth}\ 2\ \texttt{answer} \times 2^{40} + \texttt{nth}\ 3\ \texttt{answer} \times 2^{32} + \texttt{nth}\ 4\ \texttt{answer} \times 2^{24} + \texttt{nth}\ 5\ \texttt{answer} \times 2^{16} + \texttt{nth}\ 6\ \texttt{answer} \times 2^{8} + \texttt{nth}\ 7\ \texttt{answer}\ 0$

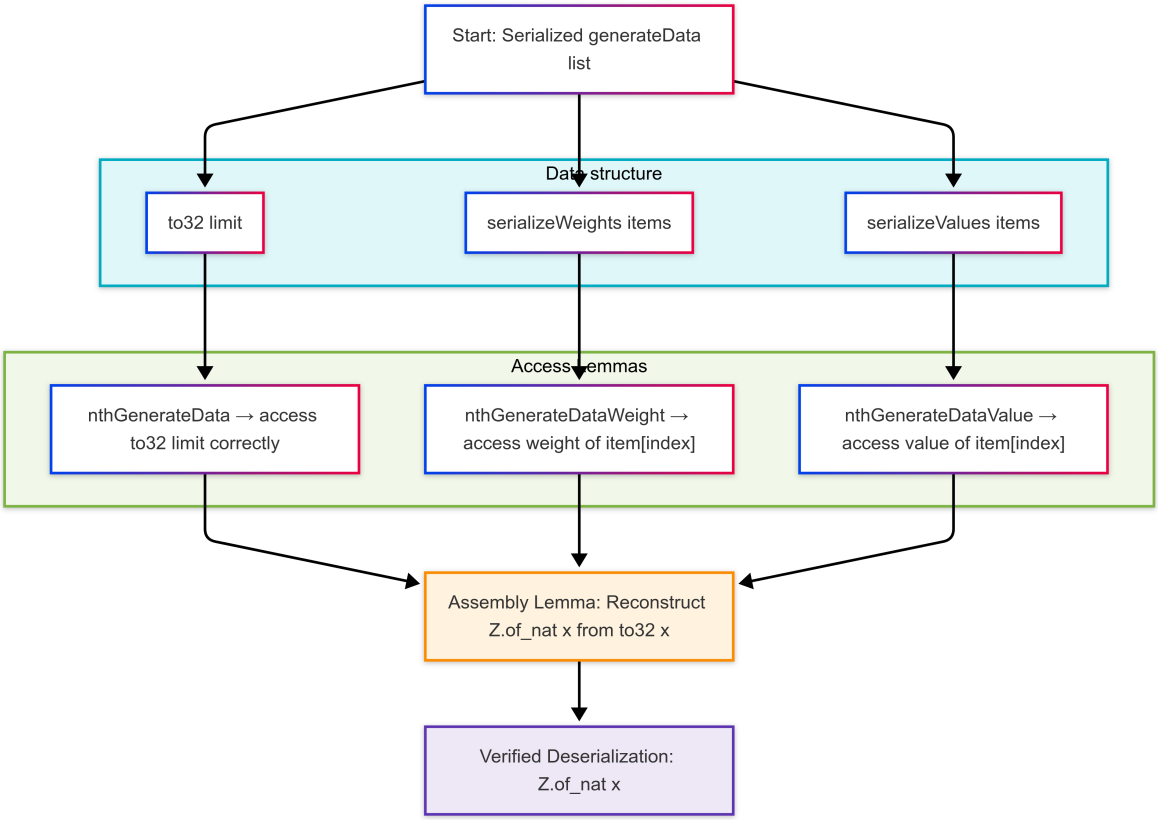


Figure 5.16: Detailed data access lemmas

– If `x` is `None` (failed call), it returns 0.

After defining the environment, we then prove the execution of the smart contract under the defined environment yields the expected result. We follow the following outline:

- Prove lemmas about accessing specific parts (limit, item weights, item values) of the `generateData` output using `nth`.
- Prove a crucial `assembly` lemma, demonstrating that the byte-wise reconstruction of a 32-bit integer (as would happen in a contract) from its `to32` representation yields the original integer.
- Prove lemmas (`readWeight`, `readValue`, `readLimit`) showing that simulated contract helper functions for reading weights, values, and the limit correctly retrieve and process these values from the calldata/storage.
- State the main `correctness` theorem, asserting that the end-to-end simulation of calling the knapsack contract and extracting its answer yields the same result as the reference `knapsack` algorithm.

6. Detailed Data Access Lemmas These lemmas verify that individual components of the knapsack problem input can be correctly retrieved from the serialized `generateData` list.

- `nthGenerateData (items : list (nat * nat)) (limit x : nat)`: This lemma proves that accessing the x-th element of the `to32 limit` part within the larger `generateData items limit` list is equivalent to directly accessing the x-th element of `to32 limit`. The proof involves unfolding `generateData` and repeatedly applying `nth_lookup` and `lookup_app_r` (lemmas for accessing elements in concatenated lists) along with the previously proven length lemmas (`serializeWeightsLength`, `serializeValuesLength`, `to32Length`) and arithmetic reasoning (`lia`).

- `nthGenerateDataWeight (items : list (nat * nat)) (limit x index : nat) (hx : (x < 4)%nat) (hIndex : (index < length items)%nat)`: This lemma proves that accessing the x-th byte of the serialized weight for the `index`-th item in `generateData` correctly corresponds to the x-th byte of `to32 (fst (nth index items (0%nat,0%nat)))`. The proof proceeds by induction on the `items` list.

  - The base case (empty list) is trivial.
  - The inductive step destructs the `head` item and the `index` (`n`).
    * If `n` is 0, it means we are accessing the first item's weight. The proof uses `lookup_app_l` to focus on the `to32 weight` part and simplifies.
    * If `n` is `S n'`, it means we are accessing a weight in the `tail`. The proof uses `lookup_app_r` to shift focus to `serializeWeights tail` and applies the induction hypothesis.

  Arithmetic (`lia`) and properties of `nth` and `to32` are used throughout.

- `nthGenerateDataValue (items : list (nat * nat)) (limit x index : nat) (hx : (x < 4)%nat) (hIndex : (index < length items)%nat)`: This lemma is analogous to `nthGenerateDataWeight` but for item values. It proves that accessing the x-th byte of the serialized value for the `index`-th item in `generateData` corresponds to the x-th byte of `to32 (snd (nth index items (0%nat,0%nat)))`. The proof structure is very similar, using induction and properties of list operations.

7. Data Deserialization (Assembly) Lemma

- `assembly (x : nat) (hx : Z.of_nat x < 2^32)`: This is a critical lemma demonstrating that if a natural number x (less than 2^{32}) is serialized using `to32`, the process of reading these four byte `Z` integers and reassembling them using shifts and additions correctly yields `Z.of_nat x`. The proof involves:

  1. Asserting properties like `Z.of_nat (2^24) = 2^24`.
  2. Using `Z.div_le_mono` to establish bounds on the results of divisions (e.g., `Z.of_nat x / 2^24 <= 2^8 - 1`).
  3. Asserting and proving that `coerceInt (nth i (to32 x) 0) 32` correctly extracts the i-th byte value. This relies on `Z.mod_small` and `Nat2Z.inj_div/inj_mod`.
  4. Rewriting with these extracted byte values and the definitions of `coerceInt` for multiplication and addition (which involve `mod 2^32`).

5. Finally, using `Z.div_mod_eq_decomp` (or a similar decomposition like $x = (x/d) * d + x \pmod d$) repeatedly to show that the sum of scaled bytes reconstructs `Z.of_nat x`. The `lia` tactic resolves the arithmetic goal.

8. Contract Helper Function Simulation Lemmas These lemmas concern the behavior of specific helper functions within the knapsack smart contract, as simulated by `invokeContractAux`. They show that these functions correctly retrieve and process their respective data from the contract's state/calldata.

- `readWeight`, `readValue`, `readLimit`: These lemmas follow a similar pattern. For example, `readWeight` proves that if `funcdef_0__getweight` (which takes an `index`) is called within the contract simulation, the effect is equivalent to directly updating the contract's message storage with the `Z.of_nat` of the actual weight of the item at that `index`. The proofs involve:

  1. Unfolding the definition of the specific getter function (e.g., `funcdef_0__getweight`).
  2. Unfolding definitions of operations like `addInt`, `multInt`, `shiftLeft`, `numberLocalGet`, `readByte`, `store`. Many of these are monadic operations, so `bindAssoc` and `leftIdentity` (for `Done`) are used extensively to simplify the action sequence.
  3. Using `unfoldInvoke_S_ReadByte` (and similar for `Store`, `Retrieve`) which likely describe how these low-level operations interact with the simulated state and calldata.
  4. Applying the `nthGenerateDataWeight` (or `Value`/`Data`) lemmas to show that `readByte` on the calldata at calculated offsets retrieves the correct byte from the `to32` representation.
  5. Using the `assembly` lemma to show that the byte-wise reconstruction within the simulated function yields the correct `Z.of_nat` value.
  6. Simplifying the state update using functional extensionality (`functional_extensionality_dep`) and properties of record/array updates (`<[ ... ]>`).

9. Overall Correctness Theorem

- `correctness (items : list (nat * nat)) (limit : nat) ...`: This is the main theorem of the development. It states that `extractAnswer (start items limit) = Z.of_nat (knapsack items limit)`, under certain preconditions (e.g., `items` is not nil, `limit` and `length items` are within bounds, item weights/values are within 2^{32}). This theorem claims that the result obtained by:

  1. Generating calldata for the given `items` and `limit`.
  2. Simulating the invocation of the main contract function (`start`, which uses `invokeContract`).
  3. Extracting the answer from the simulation's return data.

  is equal to the result of applying the reference (purely functional and verified) `knapsack` algorithm.

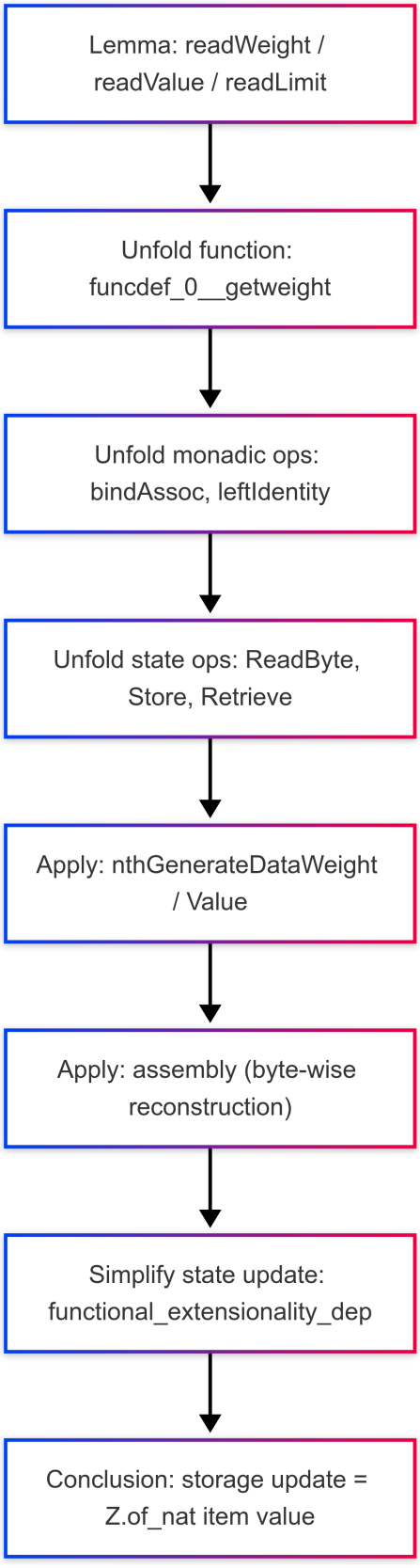


Figure 5.17: Contract simulation lemmas

Rationale for the Knapsack Proof Strategy

The formal verification of the Knapsack contract, which aims to compute the maximum total value of items that can be carried given a weight limit, follows a two-stage proof strategy. This methodology is chosen to manage complexity and ensure a thorough verification, from the abstract algorithm down to the concrete smart contract implementation. The rationale is as follows:

- **Why Define and Prove a Recursive Knapsack Function in Coq First?** The Knapsack problem has a well-known optimal substructure and overlapping subproblems, making it amenable to a recursive (or dynamic programming) solution.
  - **Clear Specification:** Defining the `knapsack` and `knapsack_elements` functions directly in Coq provides an unambiguous, formal specification of what it means to solve the knapsack problem correctly. This functional definition serves as the "gold standard" against which the smart contract's implementation is compared.
  - **Algorithmic Correctness:** Proving fundamental correctness lemmas about these Coq functions—such as `knapsackElementsSublist` (chosen items are valid), `knapsackElementsSum` (chosen items' value sum equals the `knapsack` function's output), `knapsackElementsLimit` (chosen items' weight sum is within the limit), and culminating in `knapsackMax` (the `knapsack` function indeed finds the maximum possible value satisfying the conditions)—establishes the correctness of the underlying algorithm itself, independently of any implementation details. This separation of concerns is a powerful proof technique.
- **Why Meticulously Link the Abstract Algorithm to the Smart Contract Implementation?** Simply proving the Coq `knapsack` function correct is insufficient; we must demonstrate that the smart contract, when executed, faithfully implements this verified algorithm. This involves several critical steps:
  - **Data Serialization and Deserialization:** Smart contracts interact with the external world via serialized calldata. The proof strategy includes formally defining how the input (list of items and capacity limit) is serialized into `list Z` (calldata) and, crucially, proving that the contract's internal logic for deserializing this data is correct. Lemmas like `nthGenerateData`, `nthGenerateDataWeight`, `nthGenerateDataValue`, and the `assembly` lemma are vital. They show that the contract correctly reconstructs the intended numerical values (weights, values, limit) from the byte-array representation it receives. This ensures the contract operates on the correct inputs.
  - **Simulation of Contract Execution:** The framework's capability to simulate contract execution via `invokeContractAux` is leveraged to prove the correctness of the contract's main logic and its helper functions (like `get_weight`, `get_value`, `get_limit` as compiled into Coq, e.g., `funcdef_0__getweight`). These proofs demonstrate that the step-by-step execution of the contract code, including its loops, conditional statements, array accesses (to the `dp` table), and arithmetic operations, correctly implements the dynamic programming approach inherent in the recursive `knapsack` definition.

- **End-to-End Correctness (`extractAnswerEq`):** The final key lemma, `extractAnswerEq`, ties everything together. It asserts that the entire process—serializing inputs, simulating the contract call using `start`, and then extracting and deserializing the 64-bit integer result from the contract's return data—yields a value identical to `Z.of_nat (knapsack items limit)`. This demonstrates that the observable behavior of the smart contract matches the proven-correct abstract algorithm.

- **How This Strategy Ensures Comprehensive Verification:** This two-stage strategy, combined with meticulous attention to data representation and execution semantics, ensures that the verification is holistic. It doesn't just check for isolated properties but validates that the contract correctly performs its intended complex computation from input to output. The formalization within Coq, supported by the custom language's design for verifiability and the framework's effect system, makes it possible to reason precisely about array indexing, loop invariants (implicitly through the recursive structure of proofs about loops), and arithmetic operations, which are critical for the correctness of a dynamic programming implementation like the one used in the Knapsack contract. The successful proof of `extractAnswerEq` provides high confidence in the functional correctness of the smart contract.

Chapter 6
DISCUSSION

6.1 The Burden of Verification

Formal verification makes it easier for end users of a smart contract to check whether the contract is bug free, or is free of vulnerabilities. This is because the proof written by the expert has to comply with the logical and mathematical rules of the framework as well as Coq. End users, with some basic knowledge of Coq and the framework, can easily check if the proof written by the expert proves what they want to know about the smart contract.

However, the effort needed for the expert to write the code to prove the smart contract's properties is very high, and requires a lot of mathematical knowledge. The expert has to be fluent in basic mathematical concepts taught in high school and the principles of discrete mathematics taught in college. In addition to that, the expert has to be highly familiar with how Coq works in order to write the proofs for the smart contract properties. Also, proving the smart contract properties requires writing a lot of code, a lot of intermediary lemmas and definitions. This means that the process of writing the correctness proof is highly labor intensive.

But this doesn't mean that the work is entirely impractical. Rather, it is important to see this work as a component in the larger formal verification ecosystem. Currently, Chinese AI firms are actively working on large language models that can write formal verification code in languages such as Lean and Coq. Currently, these efforts are mostly focused on high school competition math—the kind of math that is often seen in olympiad problems. They have yet to tackle program verification. It is important to note that tackling olympiad problems is an important stepping stone in making AIs that can verify software correctness in the future, as olympiad problems exhibit many problem-solving patterns that are also present in software verification.

This work, when combined with recent developments in AI, could eventually make smart contract verification a routine thing, or at least a thing that is done often enough to increase people's confidence in smart contracts.

6.2 Expansion to Other Domains

This work focuses on the verification of smart contracts. However, there are many things that can benefit from formal verification, such as web applications, mobile apps, and embedded systems.

Despite the focus on smart contracts, the framework has the potential to expand to these other domains. In every field in programming, there are conditional statements, loops, variables, arrays, and side effects. Conditional statements, loops, variables and arrays exist and are the same everywhere. But the side effects can be different depending on domain.

For web applications:

Side effects primarily involve interactions with the client-side environment (browser) and the server-side environment. Key side effects include:

- **Database Operations:** Reading from or writing to databases (e.g., SQL, NoSQL).
- **Network Requests:** Making HTTP requests to external APIs or microservices.
- **DOM Manipulation:** Modifying the structure, content, or style of the web page presented to the user (client-side).
- **Browser Storage:** Reading from or writing to cookies, `localStorage`, or `sessionStorage`.
- **User Session Management:** Creating, modifying, or destroying user sessions on the server.
- **File System Access:** Reading or writing files on the server.
- **Sending Emails or Notifications:** Triggering external communication.

For mobile apps:

Side effects involve interactions with the mobile operating system, device hardware, network resources, and local storage. Key side effects include:

- **Hardware Interaction:** Accessing device sensors (GPS, accelerometer, camera, microphone), Bluetooth, Wi-Fi.
- **Operating System Services:** Interacting with contacts, calendar, notifications, location services, background tasks.
- **Local Storage:** Reading/writing files or accessing local databases (e.g., SQLite, Realm).
- **Network Requests:** Communicating with backend servers or third-party APIs.
- **UI Updates:** Modifying the user interface elements displayed on the screen.
- **Inter-App Communication:** Sending data to or receiving data from other applications (e.g., via Intents on Android or URL Schemes on iOS).
- **Permission Handling:** Requesting and managing user permissions for accessing sensitive resources.

For embedded systems:

Side effects are heavily focused on direct hardware interaction and real-time constraints. Key side effects include:

- **Hardware Register Manipulation:** Reading from or writing to memory-mapped registers to control peripherals (GPIO, timers, ADC, DAC, communication interfaces like UART, SPI, I2C, CAN).

- **Sensor Reading:** Acquiring data from connected sensors.
- **Actuator Control:** Sending signals to control physical components (motors, LEDs, relays, valves).
- **Communication Interface Interaction:** Sending or receiving data over hardware communication protocols.
- **Interrupt Handling:** Responding to hardware interrupts, which alters program flow based on external events.
- **Non-Volatile Memory Access:** Writing data (e.g., configuration, logs) to persistent storage like EEPROM or Flash memory.
- **Timing Operations:** Interacting with hardware timers, managing real-time deadlines, and controlling task scheduling (especially in RTOS environments).

The framework's monadic framework for effects (section 3.0.4) provides a strong foundation to support multiple platforms. With the single, extensible Action monad, every side effect can be modeled within the framework. The existing code for parsing the custom language and validating types can be reused, and much of the code that compiles the language to Coq can be reused. To implement support for new platforms, it is only necessary to write additional code to traverse the AST and generate the code for the new platforms, and specify the semantics of additional side effects in Coq.

There is only so much that can be done in a bachelor's thesis, but given the extensible nature of the framework, the framework has great potential to adapt to other fields.

Chapter 7
CONCLUSION

This project proposes an alternative way of checking smart contracts to make sure they are safe and work right. We check these smart contracts using formal verification. This method uses Coq, which helps prove with math that the code is correct. This provides a stronger guarantee that the code is correct than code audits.

The project involves the creation of a custom programming language, which is based on JavaScript syntax and is intentionally simplified to reduce verification effort. We built a compiler for the custom language to translate the language into both Coq and Solidity. The Coq code is then used in proofs to prove that the program is functionally correct. The Solidity code can be deployed on a blockchain. We had to set up the basic rules and semantics in Coq using a hierarchy of effects to describe exactly how these smart contracts work and what they do.

To show that our system works well in real situations, we used it to very carefully check two smart contracts that were not simple. One was called a Disjoint Set Union (DSU) contract, and the other was a Knapsack contract. For these, we manually wrote proofs in Coq. Coq then checked our proofs to make sure they were correct. In the DSU contract, we proved the maximum amount of money that could be taken out is 5049. For the Knapsack contract, it meant proving it correctly found the best answer to the knapsack problem. We also proved that important rules about how the contracts store their information are always followed. And we proved that common mistakes or weaknesses, like math errors with numbers or trying to use parts of a list that aren't there, were not in the code.

This system gives more power to the people who use smart contracts. Because the Coq tool checks the proofs by itself, people don't have to just trust the human checkers or the people who wrote the proofs. Instead, they can trust the Coq tool, which is well-tested, and the clear math rules that were used for the proof. This fits better with the main idea of blockchain systems, which is about not having to blindly trust a single person or group.

Writing these math proofs still takes a lot of work and special skills. But this project shows it's possible to use a special language, our compiler, and a proof tool like Coq to get strong promises that smart contracts are safe and correct. In the future, new things like Artificial Intelligence (AI) might help write these proofs. Also, our system's way of handling different actions in the code is flexible. This means it could be used to check other types of computer programs too, like those for websites or phones, not just smart contracts. This shows there's a way to apply this system to other fields.

In conclusion, this project offers a real way to make smart contracts more trustworthy and reliable. It's a more thorough and strict way to check code than what's usually done. This helps with the bigger aim of making safer computer programs that are decentralized (meaning they don't rely on one single point of control). The system we built and the checks we showed prove how powerful interactive theorem proving (using tools like Coq) can be. It can bring math-level certainty to the tricky software used in the world of blockchain.

References

[1] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: Effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*, (New York, NY, USA), pp. 1–4, ACM, 2020.

[2] ConsenSys Diligence, "MythX Smart Contract Security Analysis Platform." Web Platform, approx. 2018–2024. MythX was a smart contract security analysis platform, sunset effective March 31st, 2024. Its components included Mythril (open source) and Harvey (evolved into Diligence Fuzzing). Archival information may be available at the URL. URL: https://mythx.io/.

[3] B. Mueller, "Smashing ethereum smart contracts for fun and real profit." Presentation/Whitepaper, ConsenSys Diligence & HITB SECCONF Amsterdam. Supplemental materials available at: `https://github.com/muellerberndt/smashing-smart-contracts`, 2018. This document introduces Mythril and its symbolic execution backend LASER-Ethereum.

[4] Protofire, "Solhint: Solidity Linter." GitHub repository and npm package, 2017. An open-source linter for Solidity code, providing security and style guide validations. Maintained by Protofire since 2017, as stated on their website and developer resources. Official website: https://protofire.io/solhint.

[5] R. Dua and contributors, "Ethlint (formerly Solium): Linter for Solidity." GitHub repository, 2016. A customizable linter for Ethereum Solidity, originally named Solium. The file history in the 'duaraghav8/Ethlint' repository suggests an initial development around 2016. The 'solium' npm package was not to receive updates after December 2019.

[6] L. Alt, M. Blicha, A. E. J. Hyvärinen, and N. Sharygina, "SolCMC: Solidity compiler's model checker," in *Computer Aided Verification (CAV '22), Part I*, vol. 13371 of *Lecture Notes in Computer Science*, pp. 325–338, Springer, Cham, 2022. SolCMC is the model checker module integrated into the Solidity compiler, and was formerly referred to as SMTChecker. This paper describes its architecture, functionality, and usage.

[7] Certora, "Certora Prover." GitHub repository and Commercial Tool/Service, approx. 2017-Present. A formal verification tool for smart contracts, supporting EVM-compatible chains, Solana, and Stellar. The Prover was open-sourced in late 2024 or early 2025. Official website: https://www.certora.com/.

[8] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, "KEVM: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pp. 204–217, IEEE, July 2018.

[9] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Sţefanescu, and G. Roşu, "KEVM: A complete formal semantics of the ethereum virtual machine," in *Principles of Security and Trust*

- *ETAPS 2018*, vol. 10804 of *Lecture Notes in Computer Science*, pp. 217–242, Springer International Publishing, 2018.

[10] The Coq Development Team, "The Coq Proof Assistant, version 8.20," September 2024.

Appendix A
TECHNICAL DETAILS

A.1 Topological sorting

We detail the procedure used to topologically sort an array of *CoqCPAST* modules based on their inter-dependencies. Topological sorting arranges modules in a linear order such that for every directed edge from module U to module V (meaning V depends on U), module U comes before module V in the ordering. This is essential for processes like compilation, linking, or loading, where prerequisites must be handled first.

The primary function orchestrating this is *sortModules*, which relies on several helper functions to identify dependencies, represent them as a graph, and then apply a Depth-First Search (DFS) based topological sort algorithm.

A.1.1 The Sorting Procedure: *sortModules*

The *sortModules* function takes an array of *CoqCPAST* objects and returns a new array of these modules sorted topologically.

```
export function sortModules(modules: CoqCPAST[]): CoqCPAST[] {
  // Step 1: Create index map from module name to numerical index
  const indexMap = createIndexMap(modules);

  // Step 2: Identify dependencies and create an edge list for the graph
  const { edgeList } = createEdgeListAndMentionLocation(modules, indexMap);

  // Step 3: Perform topological sort using the edge list
  const sortedIndices = topologicalSort(modules.length, edgeList);

  // Step 4: Rearrange modules according to sorted indices
  const sortedModules = sortedIndices.map((index) => modules[index]);

  return sortedModules;
}
```

Listing A.1: The *sortModules* function overview

The procedure involves the following distinct steps:

Step 1: Assigning Numerical Indices to Modules

Function: *createIndexMap(modules: CoqCPAST[])*

Graph algorithms typically operate on numerical indices for nodes. This initial step maps each module's string name (*moduleName*) to a unique integer. This integer corresponds to the module's original index in the input *modules* array.

```
export const createIndexMap = (modules: CoqCPAST[]) => {
  const indexMap = new Map(); // Maps: moduleName (string) -> index (number)
  const existingModuleNames = modules.map((x) => x.moduleName);
  for (const [index, name] of existingModuleNames.entries()) {
    indexMap.set(name, index);
  }
  return indexMap;
};
```

Listing A.2: Function *createIndexMap*

This *indexMap* is crucial for translating module name dependencies into numerical edge representations for the graph algorithm.

Step 2: Identifying Dependencies and Constructing Edges

Functions: *createEdgeListAndMentionLocation(...)* and *findDependencies(...)*

This step identifies the direct dependencies between modules and translates them into a list of graph edges. An edge is represented as a pair of numerical indices *[fromNumber, toNumber]*, signifying that the module *fromNumber* is a prerequisite for module *toNumber* (i.e., *toNumber* depends on *fromNumber*).

Discovering Dependencies: *findDependencies*

The *findDependencies* function scans a single *CoqCPAST* module to find all other modules it directly references via *'cross module call'* instructions.

```
// Inside findDependencies recursive AST traversal:
function findValueTypeDependencies(valueType: ValueType): void {
  switch (valueType.type) {
    // ... other cases
    case 'cross module call':
      dependencies.push({
        dependencyName: valueType.module, // Name of the module being called
        mention: valueType.location,
      });
      valueType.presetVariables.forEach((value) => findValueTypeDependencies(value));
      break;
    // ... other cases
  }
}
// ... setup and iteration ...
// returns: { dependencyName: string; mention: Location }[]
```

Listing A.3: Relevant part of *findDependencies*

For each *'cross module call'* encountered, the name of the target module (*valueType.module*) is recorded as a dependency.

Creating the Edge List: *createEdgeListAndMentionLocation*

This function iterates through all input modules. For each module (*currentModule*, with index *toNumber*), it calls *findDependencies* to get its list of dependent-on module names. For each *dependencyName*:

1. The numerical index of *dependencyName* is retrieved from *indexMap* (this is *fromNumber*).
2. An edge *[fromNumber, toNumber]* is added to *edgeList*.

This edge indicates a dependency: *fromNumber* → *toNumber*.

```
export const createEdgeListAndMentionLocation = (
  modules: CoqCPAST[],
  indexMap: Map<string, number>
) => {
  const edgeList: [number, number][] = [];
```

```
  // ... mentionLocation part for error reporting
  for (const [toNumber, module] of modules.entries()) {
    const dependencies = findDependencies(module).filter(
      (x) => x.dependencyName !== module.moduleName
    );
    for (const { dependencyName, mention } of dependencies) {
      const fromNumber = indexMap.get(dependencyName);
      if (fromNumber === undefined) continue; // External, unlisted dependency
      edgeList.push([fromNumber, toNumber]); // Edge: dependency -> current module
    }
  }
  return { edgeList, mentionLocation };
};
```

Listing A.4: Function *createEdgeListAndMentionLocation*

Step 3: Building the Graph Representation

Function: *edgeListToGraph(edgeList: [number, number][])*

The list of edges *edgeList* is converted into an adjacency list representation. This is a more convenient structure for graph traversal algorithms like DFS. The resulting *graph* is a *Map* where each key is a module's numerical index (*src*), and its value is an array of indices of modules (*dest*) that depend on *src*.

```
function edgeListToGraph(edgeList: [number, number][]): Map<number, number[]> {
  const graph = new Map<number, number[]>();
  for (const [src, dest] of edgeList) { // Edge [src, dest] means src is a prerequisite for
     dest
    if (!graph.has(src)) {
      graph.set(src, []);
    }
    graph.get(src)?.push(dest); // src -> dest
  }
  return graph;
}
```

Listing A.5: Function *edgeListToGraph*

So, if module 0 is a prerequisite for modules 1 and 2, *graph.get(0)* would yield *[1, 2]*.

Step 4: Performing Topological Sort using DFS

Function: *topologicalSort(nodeCount: number, edgeList: [number, number][])*

This function implements the Kahn's algorithm or a DFS-based topological sort. The provided code uses a DFS-based approach.

```
export function topologicalSort(
  nodeCount: number,
  edgeList: [number, number][]
) {
  const graph = edgeListToGraph(edgeList); // Adjacency list

  const visited = new Set<number>(); // Tracks visited nodes
  const result: number[] = [];      // Stores sorted node indices (in reverse initially)

  function dfs(node: number) {
    visited.add(node);
    const neighbors = graph.get(node) || []; // Nodes that depend on `node`
    for (const neighbor of neighbors) {
      if (!visited.has(neighbor)) {
        dfs(neighbor); // Recursively visit dependents
      }
    }
    // After all nodes that depend on `node` (and their further dependents)
    // have been visited and added to `result`, add `node` itself.
    result.push(node);
  }
```

```
22
23    // Iterate through all nodes to handle disconnected components (if any)
24    // or to start DFS from nodes with no incoming edges from other processed components.
25    for (let node = 0; node < nodeCount; node++) {
26      if (!visited.has(node)) {
27        dfs(node);
28      }
29    }
30
31    // The `result` list is currently in reverse topological order.
32    // A node is added after its dependents, so prerequisites appear later.
33    return result.reverse();
34  }
```

Listing A.6: Function *topologicalSort* (DFS-based)

- **Initialization:** An empty *Set visited* tracks visited nodes to prevent reprocessing and detect cycles (though cycle detection is separate in *findCycle*). An empty array *result* will store the sorted indices.
- **DFS Traversal (*dfs(node)*)**:
  1. Mark the current *node* as *visited*.
  2. For each *neighbor* of the current *node* (i.e., for each module that depends on the current *node*):
     - If the *neighbor* has not been visited, recursively call *dfs(neighbor)*.
  3. After all neighbors (dependents) have been visited (meaning they, and all modules that depend on them, have been added to *result*), add the current *node* to the *result* array.
- **Main Loop:** The algorithm iterates through all module indices. If a module hasn't been visited, it initiates a DFS traversal from that module. This ensures all modules are processed, even if the dependency graph has multiple disconnected components.
- **Reversing the Result:** Because a node is added to *result* only *after* all nodes that depend on it have been processed and added, the *result* array is effectively in reverse topological order. For example, if A $\rightarrow$ B (B depends on A), B would be added to *result* before A. Thus, *result.reverse()* is called to obtain the correct topological order, where prerequisites (dependencies) appear before the modules that depend on them.

Step 5: Reconstructing the Sorted Module List

The final step in *sortModules* uses the *sortedIndices* array (which contains the original numerical indices of modules, now in topological order) to build the final sorted list of *CoqCPAST* modules.

```
1   // sortedIndices contains original indices in topological order
2   const sortedModules = sortedIndices.map((index) => modules[index]);
3   return sortedModules;
```

Listing A.7: Final step in *sortModules*

This is a simple mapping operation: for each index in *sortedIndices*, the corresponding module from the original *modules* array is selected.

A.2 Cycle detection

A.2.1 The *dfs* Function for Detecting Cycles

The recursive *dfs(vertex)* function explores the graph starting from a given *vertex*.

1. **Path Tracking**: Upon entering *dfs(vertex)*, the current *vertex* is marked as part of the active exploration path. Its original *positionInPath* value is saved (as *oldPosition*), then *positionInPath[vertex]* is set to the current length of the *path* array, and *vertex* is appended to *path*. *let oldPosition = positionInPath[vertex]; positionInPath[vertex] = path.length; path.push(vertex);*

2. **Neighbor Exploration**: The function iterates through all *adjacent* nodes of the current *vertex*. These neighbors are retrieved from the *values* array, using indices derived from the *head* array: *for (let i = head[vertex]; i < head[vertex + 1]; i++)*.

    - **If *adjacent* is not in the current path** (*positionInPath[adjacent] === -1*): A recursive call, *dfs(adjacent)*, is made to continue exploration. If this sub-exploration finds a cycle (*dfsResult !== undefined*), that cycle information is immediately propagated upwards by returning *dfsResult*. The line *if (visited[vertex]) continue;* appears before the recursive call. If *visited[vertex]* is true at this stage, it means this vertex's full exploration from a previous DFS initiation point has completed without finding cycles through its remaining unvisited neighbors from that prior context. This line would skip further processing of neighbors for an already fully explored *vertex* if it's re-entered as a neighbor.
    - **If *adjacent* is already in the current path** (*positionInPath[adjacent] !== -1*): This condition signifies that a back edge has been found, and thus a cycle exists. The *adjacent* node is an ancestor of the current *vertex* in the DFS tree for this path. The cycle is then constructed. The variable *start* is assigned *positionInPath[adjacent]*, which is the index in the *path* array where *adjacent* was first encountered in the current active path. The cycle is formed by taking a slice of the *path* from this *start* index to the end, and then appending *adjacent* to explicitly close the loop: *[...path.slice(start), adjacent]*. This array representing the cycle is then returned.

3. **Backtracking**: If the loop through all neighbors of *vertex* completes without returning a cycle, the function backtracks. The *vertex* is removed from the current *path* (*path.pop()*), its *positionInPath* is restored to *oldPosition*, and *visited[vertex]* is set to *true*. This *visited[vertex] = true* signifies that all reachable paths from *vertex* have been explored for the current DFS traversal without finding a cycle involving *vertex* as a point of cycle completion for paths explored from it.

Initiating DFS Traversals

To ensure all parts of the graph (including disconnected components) are checked, the *dfs* function is called iteratively for each node from *0* to *n-1*: *for (let i = 0; i < n; i++) ...* If a node *i* has already been *visited* (i.e., fully processed by a previous DFS call), it is skipped (*if (visited[i]) continue;*). Otherwise, *dfs(i)* is initiated. If any such call returns a cycle, this cycle is immediately returned by the main *findCycle* function.

Return Value

If the main loop completes without any *dfs* call detecting and returning a cycle, the *findCycle* function returns *undefined*, indicating the graph is acyclic. Otherwise, it returns the array of nodes forming the first detected cycle. Then, each node is then mapped to a "call implicated in cycle" error that is then reported to the user.

After the cycle detection phase, the validation process moves on to the next phase: validating the types, function calls, array accesses and other aspects of the code.

A.3 Knapsack proof details

Helper Lemmas for `fold_right`

These lemmas simplify reasoning about `fold_right` when summing either the first or second components of pairs in a list.

`foldrSum9`

```
Lemma foldrSum9 (l : list (nat * nat)) (a b : nat) :
  fold_right (fun x acc => snd x + acc) 0 ((a, b) :: l) =
  b + fold_right (fun x acc => snd x + acc) 0 l.
```

Idea: This proves that summing the second components (`snd x`, typically values) of a list starting with `(a,b)` is equivalent to `b` plus the sum of the second components of the rest of the list `l`. **Proof Idea:** `unfold foldr. simpl. reflexivity.` This is straightforward by the definition of `fold_right`.

`foldrSum11`

```
Lemma foldrSum11 (l : list (nat * nat)) (a b : nat) :
  fold_right (fun x acc => fst x + acc) 0 ((a, b) :: l) =
  a + fold_right (fun x acc => fst x + acc) 0 l.
```

Idea: Similar to `foldrSum9`, but for the first components (`fst x`, typically weights). Summing the first components of a list starting with `(a,b)` is `a` plus the sum of the first components of `l`. **Proof Idea:** `unfold foldr. simpl. reflexivity.`

Key Correctness Lemmas for the Knapsack Function

`knapsackElementsSublist`

```
Lemma knapsackElementsSublist (l : list (nat * nat)) (limit : nat) :
  sublist (knapsack_elements l limit) l.
```

Idea: This lemma proves that any item included in the result of `knapsack_elements` must have been present in the original list `l`. **Proof Idea:** By induction on `l`.

- **Base Case (`l = []`):** `knapsack_elements [] limit` is `[]`. `[]` is a sublist of `[]`. Proven by `easy`.
- **Inductive Step (`l = head :: tail`):** Let `head` be `(item_weight, item_value)`. The proof proceeds by cases based on the definition of `knapsack_elements`:

1. **If `limit < item_weight`:** `knapsack_elements` calls itself on `tail`. The result `knapsack_elements tail limit` is a sublist of `tail` by the inductive hypothesis (IH). Since `tail` is a sublist of `head :: tail`, the property holds (using `sublist_cons`). A snippet from the proof:

   ```
   (* After simpl, intro limit, destruct head, case_decide as h1. *)
   (* This is the branch where h1: limit < weight is true *)
   - apply sublist_cons.
     apply IH.
   ```

2. **If `not (limit < item_weight)`:** `knapsack_elements` chooses between taking `head` or not. This involves another `case_decide` (named `h2` in the proof script).
   - **If taking `head` is better (first branch of `h2`):** The result is `head :: knapsack_elements tail (limit - item_weight)`. By IH, `knapsack_elements tail (limit - item_weight)` is a sublist of `tail`. Thus, `head :: (sublist of tail)` is a sublist of `head :: tail` (using `sublist_skip`).

     ```
     (* This is the branch where h1: not (limit < weight) is true *)
     (* And h2: taking the item is better *)
     + apply sublist_skip.
       apply IH.
     ```

   - **If not taking `head` is better or equal (second branch of `h2`):** The result is `knapsack_elements tail limit`. By IH, this is a sublist of `tail`. Thus, it's also a sublist of `head :: tail` (using `sublist_cons`).

     ```
     (* This is the branch where h1: not (limit < weight) is true *)
     (* And h2: not taking the item is better or equal *)
     + apply sublist_cons.
       apply IH.
     ```

`knapsackElementsSum`

```
Lemma knapsackElementsSum (l : list (nat * nat)) (limit : nat) :
  fold_right (fun x acc => snd x + acc) 0 (knapsack_elements l limit) =
  knapsack l limit.
```

Idea: The sum of values of the items chosen by `knapsack_elements` is equal to the optimal value computed by `knapsack`. **Proof Idea:** By induction on `l`.

- **Base Case (`l = []`):** Both sides are `0`. `easy`.
- **Inductive Step (`l = head :: tail`):** Let `head` be `(item_weight, item_value)`. The proof considers two main scenarios based on `decide (limit < item_weight)` (`h1`):

  1. **If `limit < item_weight` (`h1` is true):** Both `knapsack_elements (head :: tail) limit` and `knapsack (head :: tail) limit` simplify to their respective calls on `tail limit`. The goal becomes `fold_right ... (knapsack_elements tail limit) = knapsack tail limit`, which is exactly the IH.

```
- (* Case: limit < weight *)
  apply IH.
```

2. **If `not (limit < item_weight)` (`h1` is false):** The proof then considers the inner `if` in `knapsack_elements` (whether taking the item is better, condition `h2`).
   - **If taking `head` is better (first branch of `h2`):** The LHS becomes `fold_right ... (head :: knapsack_elements tail (limit - item_weight))`. By `foldrSum9` and IH, this is `item_value + knapsack tail (limit - item_weight)`. The RHS is `knapsack tail limit` `` `max` `` `(item_value + knapsack tail (limit - item_weight))`. Given the condition for this branch (rewritten using IH in `h2` as `knapsack tail limit < item_value + knapsack tail (limit - item_weight)`), the `max` resolves to `item_value + knapsack tail (limit - item_weight)`. Both sides are equal.

```
+ (* Case: taking item is better, after rewriting h2 with IH
     *)
  rewrite (ltac:(lia) : (* rewrite for max *)), foldrSum9,
   IH.
  reflexivity.

```

   - **If not taking `head` is better or equal (second branch of `h2`):** The LHS becomes `fold_right ... (knapsack_elements tail limit)`. By IH, this is `knapsack tail limit`. The RHS, given the condition for this branch (rewritten using IH in `h2` as `not (knapsack tail limit < item_value + knapsack tail (limit - item_weight))`), the `max` resolves to `knapsack tail limit`. Both sides are equal.

```
+ (* Case: not taking item is better/equal, after rewriting
    h2 with IH *)
  rewrite (ltac:(lia) : (* rewrite for max *)), IH.
  reflexivity.

```

`knapsackElementsLimit`

```
Lemma knapsackElementsLimit (l : list (nat * nat)) (limit : nat) :
  fold_right (fun x acc => fst x + acc) 0 (knapsack_elements l limit) <= limit.
```

Idea: The sum of weights of items chosen by `knapsack_elements` does not exceed the given `limit`. **Proof Idea:** By induction on `l`.

- **Base Case (`l = []`):** `fold_right ... []` is `0`. `0 <= limit` is true. Solved by `lia`.
- **Inductive Step (`l = head :: tail`):** Let `head` be `(item_weight, item_value)`.
  1. **If `limit < item_weight`:** `knapsack_elements (head :: tail) limit` becomes `knapsack_elements tail limit`. The goal is `fold_right ... (knapsack_elements tail limit) <= limit`, which is `IH limit`.

```
{ apply IH. }
```

2. **If `not (limit < item_weight)`:**
   - **If taking `head` is better:** We need to show `fold_right ... (head :: knapsack_elements tail (limit - item_weight)) <= limit`. By `foldrSum11`, this is `item_weight + fold_right ... (knapsack_elements tail (limit - item_weight))`. By `IH (limit - item_weight)`, we know `fold_right ... (knapsack_elements tail (limit - item_weight)) <= limit - item_weight`. So the expression is `item_weight + (something <= limit - item_weight)`, which is `<= limit`. Proven by `lia`.

     ```
     { rewrite foldrSum11.
       pose proof IH (limit - weight). lia. }
     ```

   - **If not taking `head` is better or equal:** We need to show `fold_right ... (knapsack_elements tail limit) <= limit`. This is `IH limit`.

     ```
     { pose proof IH limit. lia. }
     ```

Main Correctness Theorem: `knapsackMax`

`isMaximum` Definition First, a helper definition:

```
Definition isMaximum (x : nat) (predicate : nat -> Prop) :=
  predicate x /\ (forall y, predicate y -> y <= x).
```

Idea: `x` is the maximum value satisfying `predicate` if `x` itself satisfies `predicate`, and any other value `y` that also satisfies `predicate` is less than or equal to `x`.

`knapsackMax` Lemma

```
Lemma knapsackMax (l : list (nat * nat)) (limit : nat) :
  isMaximum (knapsack l limit)
            (fun x => exists choice,
   sublist choice l /\
   fold_right (fun x acc => snd x + acc) 0 choice = x /\
   fold_right (fun x acc => fst x + acc) 0 choice <= limit).
```

Idea: This is the main theorem. It states that `knapsack l limit` is the maximum value (`x`) such that there `exists` a `choice` of items which:

1. Is a `sublist` of the original items `l`.
2. The sum of values (`snd`) of `choice` equals `x`.
3. The sum of weights (`fst`) of `choice` is less than or equal to `limit`.

Proof Idea: By induction on `l`. The proof involves two main parts for `isMaximum`:

1. **Show that `knapsack l limit` itself satisfies the predicate:** This means showing there exists a `choice` (namely, `knapsack_elements l limit`) such that the three conditions above hold. This relies on `knapsackElementsSublist`, `knapsackElementsSum`, and `knapsackElementsLimit`. The Coq proof constructs this part as:

```
(* In the inductive step, first part of proving isMaximum *)
- exists (knapsack_elements (head :: tail) limit).
  constructor; [| constructor];
    [apply knapsackElementsSublist |
     apply knapsackElementsSum |
     apply knapsackElementsLimit].
```

2. **Show that for any other `score` satisfying the predicate, `score <= knapsack l limit`:**

   - **Base Case (`l = []`):** If `choice` is a sublist of `[]`, then `choice` must be `[]`. The sum of values is `0`. `knapsack [] limit` is also `0`. So `0 <= 0`.
   - **Inductive Step (`l = head :: tail`):** Assume we have a `score` and a `choice` (named `l1` in the proof) from `head :: tail` satisfying the conditions. Let `head` be `(item_weight, item_value)`. The goal is `score <= knapsack (head :: tail) limit`. The structure of the RHS depends on `decide (limit < item_weight)`.
     - **Case A: `limit < item_weight`:** The goal becomes `score <= knapsack tail limit`. Since `l1` is a sublist of `head :: tail`, `l1` is either a sublist of `tail` or `l1 = head :: l2` where `l2` is a sublist of `tail`.
       * If `l1` is a sublist of `tail`: The properties of `l1` allow use of `IH tail limit` (its second part, `bb`) to show `score <= knapsack tail limit`.
       * If `l1 = head :: l2`: Then `sum_weights(l1) = item_weight + sum_weights(l2)`. We know `sum_weights(l1) <= limit`. But `item_weight > limit`. This leads to a contradiction (`lia`), making this case impossible.
     - **Case B: `not (limit < item_weight)`:** The goal is `score <= (knapsack tail limit `max` (item_value + knapsack tail (limit - item_weight)))`. Again, `l1` is a sublist of `tail` or `l1 = head :: l2`.
       * If `l1` is a sublist of `tail`: Using `IH tail limit`, we get `score <= knapsack tail limit`. This implies the main goal.
       * If `l1 = head :: l2`: `score = item_value + sum_values(l2)`. `sum_weights(l1) = item_weight + sum_weights(l2) <= limit`, so `sum_weights(l2) <= limit - item_weight`. Using `IH tail (limit - item_weight)` on `l2` (whose value sum is `score - item_value`), we get `score - item_value <= knapsack tail (limit - item_weight)`. Thus, `score <= item_value + knapsack tail (limit - item_weight)`. This implies the main goal.

     The `lia` tactic is crucial for resolving arithmetic inequalities.