國立臺灣大學電機資訊學院電機工程學系
碩士論文
Department of Electrical Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Master Thesis

形式化驗證零知識證明系統編譯器

From a Dependently Typed Language to ZK-SNARKs Circuits: A Formally Verified Compiler

許瑞麟
Ruey-Lin Hsu

指導教授：鄭振牟博士
Advisor: Chen-Mou Cheng, Ph.D.

中華民國 109 年 7 月
July, 2020

doi:10.6342/NTU202003013

誌謝

首先，我要感謝我的指導教授鄭振牟教授，是他指引了我進入了密碼學相關的領域，也是他向說我提出了此論文的零知識證明題目。再來，我要感謝穆信成研究員，是他使我了解了許多與型別理論，程式推導，和許多與程式語言理論有關的東西。並且，穆老師也對我的論文提出了很多有幫助的意見。另外，我要感謝我的口試委員：鄭振牟，楊伯因，穆信成，柯向上，謝謝他們願意撥出時間來當我的口試委員以及參加我的口試。

doi:10.6342/NTU202003013

doi:10.6342/NTU202003013

Acknowledgements

Above all, I want to thank my advisor Professor Chen-Mou Cheng for what he did to introduce me to the field of cryptography and proposed the topic for this dependently typed zkSNARK compiler to me. I also want to thank Researcher Shin-Cheng Mu who helped me understand many programming language theory related topics like type theory and program derivation, and gave a lot of helpful comments regarding my master thesis. Furthermore, I want to thank Chen-Mou Cheng, Bo-Ying Yang, Shin-Cheng Mu, and Hsiang-Shang Ko for being a part of my oral examination committee and taking the time to participate in my oral examination.

doi:10.6342/NTU202003013

doi:10.6342/NTU202003013

摘要

本論文提出一個依值型別的可驗證計算編譯器，並證明其可靠性。此編譯器將一個淺層嵌入於Agda當中，具有依值型別的領域特定語言轉換成一階限制條件。可靠性是轉換正確性的一個部份，表示如果產生出來的限制條件是可被滿足的，那產生出來的限制條件會是正確的。藉由利用柯里-霍華德對應，我們在互動式定理證明器Agda當中建構此編譯器的形式規格以及證明其可靠性。

關鍵字： 依值型別程式設計, 互動式定理證明器, 可驗證計算, 形式化驗證, Agda

doi:10.6342/NTU202003013

doi:10.6342/NTU202003013

Abstract

In this thesis, we will construct and prove the soundness of a dependently typed verifiable computation compiler. The compiler described in this thesis compiles a user program written in a dependently typed shallowly embedded domain specific language in Agda into a set of rank 1 constraints. Soundness is a part of translational correctness that says that if the generated constraints are satisfiable, then the generated constraints are correct. By utilizing the Curry-Howard correspondence, the compiler is formally specified and proved in the interactive theorem prover Agda.

Keywords: dependently typed programming, interactive theorem prover, verifiable computation, formal verification, Agda

doi:10.6342/NTU202003013

Contents

doi:10.6342/NTU202003013

doi:10.6342/NTU202003013

List of Figures

Chapter 1

Introduction

Dependent type theory[11] is a powerful general purpose tool that can be used for both general purpose programming and theorem proving. One of the most fascinating and powerful things about interactive proof assistants based on dependent type theories like Agda[13], Coq[15], and Idris[4] is the abilily to develop programs together with specifications and proofs of their properties in the same language through the Curry-Howard correspondence.

In this thesis, I will describe the construction and formal verification of a verifiable computation compiler that compiles a dependently typed EDSL (embedded domain specific language) in Agda into a set of rank 1 constraints, which is then piped into the zk-SNARK library libsnark.

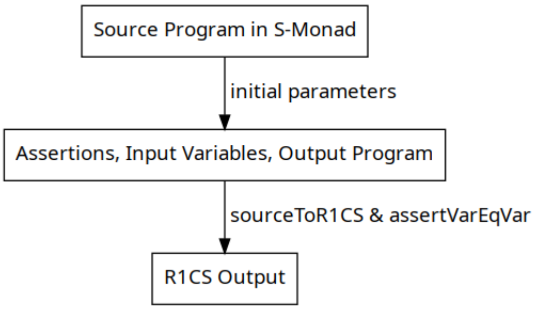

Figure 1.1: Compilation Pipeline

This thesis is an attempt at integrating dependently typed programming into verifiable computation schemes. A verifiable computation scheme can be used to outsource a computation to a potentially untrusted third party, where one only has to examine a small cryptographic proof to know that the computation is performed correctly by a third party.

One way of thinking about type systems in programming languages is that type systems are a way of statically eliminating incorrect programs that might go wrong when executed. Another way to think about type systems in programming languages is that a type tells you what you can expect from a program. Suppose that a program p has type (Int, Int), then the programmer might expect a tuple of integers from the execution of p instead of say, a tuple of strings.

The work of Steward et al.[14] on verifiable computation lets a person write declarative programs living in a Haskell DSL that compiles to verifiable computation constraints. Logically one might think that since dependently typed programming has a long history in interactive theorem proving and functional programming, it would be interesting to see what it is like to have a DSL with a more expressive type system that can compile to verifiable computation constraints. This work is an attempt to explore this question with dependently typed programming.

By using inductive-recursive definitions to encode dependent types à la Tarski[6], it is possible to embed a dependently typed DSL within a dependently typed language itself. This type encoding construction is then used to construct a dependently typed Agda DSL that targets the verifiable computation backend R1CS. One thing that having dependent type allows us to have is branching. Having dependent types in our language allows us to express the possibility of executing different programs with distinct types within our DSL.

The embedded Agda DSL used for composing the source programs is designed to be used together with a state transformation monad that records an unused variable together with a list of input variables (natural numbers) and a list of solver hints and equality constraints over an inductively defined Source datatype (indexed over the type of type codes representing permitted types).

Chapter 2 introduces the background knowledge and ideas used in the compiler, such as type theory, zkSNARK, and verifiable computing. Chapter 3 describes the basic constructions used in the compiler. Chapter 4 describes the source language and the utilities that can be used when writing programs in the source language. In Chapter 5, the construction of the verifiable computation compiler is described, and in Chapter 6, the formal verification of the soundness of the compiler is described.

doi:10.6342/NTU202003013

Chapter 2

Background

2.1 Type Theory

Dependent type theory a la MLTT is a Gentzen style natural deduction system. A derivation in such a system can be seen as an annotated proof tree, and a program can be seen as a microcosm of its corresponding proof tree. Type theory naturally gives rise to the Curry-Howard correspondence: the propositions as types interpretation tells us that a program corresponds to a proof and a type corresponds to a proposition.

The Π type represents universal quantification:

$$\frac{\Gamma \vdash \mathrm{A} : \mathrm{Set} \qquad \Gamma \vdash \mathrm{B} : \mathrm{A} \to \mathrm{Set}}{\Gamma \vdash \Pi_{\mathrm{x}\,:\,\mathrm{A}}\, \mathrm{B}\ \mathrm{x} : \mathrm{Set}}\,\Pi\text{-F}$$

$$\frac{\Gamma, \mathrm{x} : \mathrm{A} \vdash \mathrm{t} : \mathrm{B}\ \mathrm{x}}{\Gamma \vdash \lambda \mathrm{x} : \mathrm{A}.\, \mathrm{t} : \Pi_{\mathrm{x}\,:\,\mathrm{A}} \mathrm{B}\ \mathrm{x}}\,\Pi\text{-I}$$

$$\frac{\Gamma \vdash t_1 : \Pi_{\mathrm{x}\,:\,\mathrm{A}} \mathrm{B}\ \mathrm{x} \qquad \Gamma \vdash \mathrm{t}_2 : \mathrm{A}}{\Gamma \vdash t_1\ t_2 : \mathrm{B}\ t_2}\,\Pi\text{-E}$$

The Σ type represents existential quantification:

$$\frac{\Gamma \vdash \mathrm{A} : \mathrm{Set} \qquad \Gamma \vdash \mathrm{B} : \mathrm{A} \to \mathrm{Set}}{\Gamma \vdash \Sigma_{\mathrm{x}\,:\,\mathrm{A}}\ \mathrm{B}\ \mathrm{x} : \mathrm{Set}}\ \Sigma\text{-F}$$

$$\frac{\Gamma \vdash \mathrm{t}_1 : \mathrm{A} \qquad \Gamma \vdash \mathrm{t}_2 : \mathrm{B}\ \mathrm{t}_1}{\Gamma \vdash (\mathrm{t}_1\ ,\ \mathrm{t}_2) : \Sigma_{\mathrm{x}\,:\,\mathrm{A}}\ \mathrm{B}\ \mathrm{x}}\ \Sigma\text{-I}$$

$$\frac{\Gamma \vdash \mathrm{p} : \Sigma_{\mathrm{x}\,:\,\mathrm{A}}\ \mathrm{B}\ \mathrm{x}}{\Gamma \vdash \mathrm{fst}\ \mathrm{p} : \mathrm{A}}\ \Sigma\text{-fst}$$

$$\frac{\Gamma \vdash \mathrm{p} : \Sigma_{\mathrm{x}\,:\,\mathrm{A}}\ \mathrm{B}\ \mathrm{x}}{\Gamma \vdash \mathrm{snd}\ \mathrm{p} : \mathrm{B}\ (\mathrm{fst}\ \mathrm{p})}\ \Sigma\text{-snd}$$

Non-dependent logical implication and cartesian product are special cases of Π and Σ types respectively.

In type theory, there is the notion of definitional equality, which is a meta-theoretic equality, and which forms the basis of type checking. Terms and types that are definitionally equal are indistinguishable on the object level. Definitional equality can be defined as the equivalence relation generated by a set of structural and equivalence closure rules stating that equal terms are substitutable everywhere and that terms are definitionally identified up to β conversion.

There is also the notion of propositional equality (or equality type), which is an object level equality that expresses the fact that two terms are equal:

$$\frac{\Gamma \vdash \mathrm{A} : \mathrm{Set}}{\Gamma \vdash \_\equiv_A\_ : \mathrm{A} \to \mathrm{A} \to \mathrm{Set}}\ \equiv\text{-F}$$

$$\frac{\Gamma \vdash \mathrm{x} : \mathrm{A}}{\Gamma \vdash \mathrm{refl}\ \mathrm{x} : \mathrm{x} \equiv_A \mathrm{x}}\ \equiv\text{-I}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathrm{C} : (\mathrm{x}\ \mathrm{y} : \mathrm{A}) \to \mathrm{x} \equiv_A \mathrm{y} \to \mathrm{Set} \\ \Gamma \vdash \mathrm{C\text{-}refl} : (\mathrm{x} : \mathrm{A}) \to \mathrm{C}\ \mathrm{x}\ \mathrm{x}\ (\mathrm{refl}\ \mathrm{x}) \\ \Gamma \vdash \mathrm{x} : \mathrm{A} \\ \Gamma \vdash \mathrm{y} : \mathrm{A} \\ \Gamma \vdash \mathrm{p} : \mathrm{x} \equiv_A \mathrm{y}\end{array}}{\Gamma \vdash \equiv\text{-ind C C-refl x y p} : \mathrm{C}\ \mathrm{x}\ \mathrm{y}\ \mathrm{p}}\ \equiv\text{-E(J)}$$

In Agda, the Σ type and the propositional equality type can be roughly translated into their corresponding datatype definitions:

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B fst

data _≡_ {A : Set} : A → A → Set where
  refl : (x : A) → x ≡ x
```

and a Π type corresponds to a function definition in Agda.

Proofs in Agda are written with dependent pattern matching, which was shown to be equivalent to traditional type theory with inductive families plus the addition of axiom K[9][12]. Recent work [5] has also shown that by placing certain restrictions on dependent pattern matching, it's possible to translate programs written with restricted pattern matching rules into traditional type theory with inductive families without the use of axiom K. However, in this thesis, we will be using axiom K to construct the compiler and prove its soundness.

2.2 Verifiable Computation

Verifiable computation can be used to delegate computations to potentially untrusted machines. In this thesis we will focus on a particular approach of verifiable computation –

zkSNARKs. Currently, there are a couple of potential applications for zkSNARKs, like private transactions or smart contracts in cryptocurrencies and verifiable computation. In a zkSNARK, there are two parties, a prover and a verifier. The prover is the party performing the computation, and the one producing a cryptographic proof π that the verifier can use to determine with high probability whether or not the computation is performed correctly.

A zkSNARK consists of a set of probabilistic algorithms: KeyGen ($\mathcal{K}$), Prove ($\mathcal{P}$), and Verify ($\mathcal{V}$). Given a security parameter λ and a program C, a zkSNARK protocol goes as follows:

$$\begin{aligned}(k_p, k_v) &\leftarrow \mathcal{K}(\lambda, \mathcal{C}) \\ \pi &\leftarrow \mathcal{P}(\mathcal{C}, k_p, public, witness) \\ \{true, false\} &\leftarrow \mathcal{V}(\mathcal{C}, k_v, public, \pi)\end{aligned}$$

where k_p is the proving key, k_v is the verifcation key, $public$ is the public variables, $witness$ is the non-public (private) variables, and $\mathcal{C}$ is the program.

In this thesis, the program $\mathcal{C}$ fed to the keygen, the prover and the verifier will be the R1CS constraints (defined below) generated by our compiler. The variables in $\mathcal{C}$ include the input and output variables of the program together with all the intermediate values. And $public$ together with $witness$ constitute the variables in $\mathcal{C}$ (the purpose of $witness$ will be discussed later in this section).

Definition 2.2.1. *A rank 1 constraint system[2] (R1CS) $\mathcal{S}$ over a field $\mathbb{F}$ with N_g constraints is a set of vectors*

$$\{(a_i, b_i, c_i) | i \in [1, N_g], a_i \in \mathbb{F}^{1+N_v}, b_i \in \mathbb{F}^{1+N_v}, c_i \in \mathbb{F}^{1+N_v}\}.$$

and a non-negative integer N_i,

where

- $\mathbb{F}$ *is a field*

doi:10.6342/NTU202003013

- *N_v is the number of variables*

- *$N_i \leq N_v$ is the number of public variables.*

If there is some public values $x \in \mathbb{F}^{N_i}$ and private values (witness) $w \in \mathbb{F}^{N_v - N_i}$ such that

$$\langle a_i, (1, x, w) \rangle \langle b_i, (1, x, w) \rangle = \langle c_i, (1, x, w) \rangle$$

for all $i \in [1, N_g]$ (where $\langle m, n \rangle$ denotes the dot product of m and n, and the left hand side of the equation multiplies the two dot products together), then S is said to be satisfiable with public values x and witness w.

The definition of R1CS can be seen as a charaterization of NP problems as NP-complete problems like SAT can be reduced to R1CS satisfaction problems in polynomial time, and this is used to define what it means to "know" something in the zkSNARK proof of knowledge definition.

Suppose that a programmer A has written a program $Prog(a, b) = (a + b) * b$ where $a, b \in \mathbb{F}$ are the inputs to $Prog$ and A wants to outsource this computation $Prog$ to an untrusted cloud server CS. A can choose to encode $Prog$ as R1CS constraints (which can then be further processed into a quadratic arithmetic program (QAP)[8]).

In this example, $Prog$ can be encoded with a single R1CS constraint with three R1CS variables. Besides a and b, we need another variable out to represent the output of the program, that is, $out = (a + b) * b$. So if we fix the order of the variables (including the constant 1) to be $[1, a, b, out]$, the R1CS constraint would be a singleton set consisting of the element ([0, 1, 1, 0], [0, 0, 1, 0], [0, 0, 0, 1]). What if we require a to be a boolean? This can be accomplised by adding another constraint ([0, 1, 0, 0], [0, 1, 0, 0], [0, 1, 0, 0]) (which says that $a^2 = a$) to the set of constraints. If a is equal to 0, this constraint is satisfied. Otherwise, if $a \neq 0$, we multiple both sides of the equation by a^{-1}, and we get that it must be the case that $a = 1$, and thus, a is a boolean. In this scenario, A wants to know the result out, and the three variables a, b, and out are all public variable in the zkSNARK.

After the program is encoded into R1CS, the resulting constraints are then sent to CS, where a key pair is first generated by A and then the programmer has to decide the input to be fed into $Prog$ (say, fixing $a = 2$ and $b = 3$).

Then the transformed arithmetic constraints along with the fixed inputs 2 and 3 are then sent to CS (which will then try to solve the variable out in the transformed constraints and execute the prover algorithm $\mathcal{P}$ with a, b, and out to generate the proof π). CS then gives π and the variables that are considered to be public to A (which plays the role of the verifier). A then checks if the combination of the public variables and π is valid, and if it is valid, then A can have a high confidence that CS has indeed performed the required computations, and that the output is correct provided that the transformation of $Prog$ is correct.

In some variants of the above scenario, the prover might want to hide some information from the verifier, and this is where the private $witness$ in S comes into play. For example, the prover might want to hide the intermediate values resulting from the execution of a program from the verifier (and only the input and output of the program is made public), and from the zero knowledge guarantee from the zk-SNARK backend, we can know that apart from the public variables, the proof π doesn't tell us additional information about the private $witness$ in VC. The properties that a zkSNARK system has to satisfy is made more precise in the following paragraph.

A zkSNARK[2][3] (which stands for zero knowledge succinct non-interactive argument of knowledge) satisfies the following properties:

- completeness: If $public$ together with $witness$ constitutes a solution to a program $\mathcal{C}$, and

$$(k_p, k_v) \leftarrow \mathcal{K}(\lambda, \mathcal{C})$$

$$\pi \leftarrow \mathcal{P}(\mathcal{C}, k_p, public, witness)$$

  then

$$\mathcal{V}(\mathcal{C}, k_v, public, \pi) = true.$$

- succintness: the size of the proof π generated by $\mathcal{P}$ is $O_\lambda(1)$ (independent of the size of C).

- proof of knowledge: For any probabilistic polynomial time (PPT) adversary A, there is a PPT extractor E such that for every constant $c > 0$, large enough λ, auxiliary input z (where $|z| = poly(\lambda)$) and every program C of size λ^c,

$$\Pr\left[\begin{array}{c|lcl} \mathcal{V}(\mathcal{C}, k_v, public, \pi) = true & (k_p, k_v) & \leftarrow & \mathcal{K}(\lambda, \mathcal{C}) \\ & (public, \pi) & \leftarrow & A(z, p_k, p_v) \\ (public, witness) \text{ not a solution of } \mathcal{C} & witness & \leftarrow & E(z, p_k, p_v, r_A) \end{array}\right]$$

$$\leq negl(\lambda)$$

  where r_A is the random tape of A.

- zero knowledge: meaning that the proof π does not leak any information about $witness$.

Completeness says that someone with a solution $(public, witness)$ to a program C can always produce a proof π that convinces a verifier who follows the zkSNARK protocol. Sunccinctness says that the proof π is small. Proof of knowledge says that if a PPT adversary produces $public$ and a proof π that $\mathcal{V}$ checks to be valid, then with a large enough λ, there is a high probability that there is a PPT knowledge extractor E that can "extract" the knowledge $witness$ from A so that $(public, witness)$ is a solution to $\mathcal{C}$.

Since we will be building the constraints in Agda, following the work of Stewart et al[14], we define the target compilation type R1CS as follows (parameterized over a type *f*):

```
data R1CS : Set where
  IAdd : f → List (f × Var) → R1CS
         -- sums to zero
  IMul : (a : f) → (b : Var) → (c : Var)
                 → (d : f) → (e : Var) → R1CS
```

```
        -- a * b * c = d * e
  Hint : (Map Var ℕ → Map Var ℕ) → R1CS
  Log : String → R1CS
```

where *IAdd* f_1 $((f_2, i_2) :: (f_3, i_3)$ ... $:: [])$ expresses an additive constraint $f_1 + f_2 v_{i_2} + f_3 v_{i_3} ... = 0, f_i \in \mathbb{F}, v_{i_k} \in \mathbb{F}$, and *IMul* f_a b c f_d e expresses a multiplicative constraint $f_a v_b v_c = f_d v_e$ where $f_a, v_b, b_c, f_d, v_e \in \mathbb{F}$. The vectors in an R1CS constraint system are usually sparse and this is why the R1CS datatype is not defined as a tuple of vectors. A list of constraints of type [*R1CS*] in Agda can be easily transformed into the regular tuple of vectors representation. Since a set of R1CS constraints represents an NP-complete problem in general, the definition of R1CS includes hints to help the solver solve these R1CS constraints (and also *Log* to help with debugging).

To date, there have been numerous attempts at compiling existing programming languages like C, or specially designed domain specific languages into zkSNARK systems. ZoKrates[16], SNARKs for C[2], Snårkl[14], and the formally verified compiler made by Fournet et al[7] are all examples of this. However, the source languages of these existing compilers lack an expressive type system. Inspired by the work of Snårkl, we attempt to construct and integrate a dependently typed embedded domain specific language into a verifiable computation system.

Chapter 3

Constructing an Embedded Type Universe

In this chapter, we are going to construct a dependently typed embedded type universe and determine R1CS variable allocation for an element in our embedded type universe. First we are going to describe how the embedded type universe is constructed for our EDSL. The constructed type universe will have the property that each type in the type universe will only have finitely many inhabitants (when instantiated with a finite base type). For such instantiations of our type universe, we can construct an enumeration function *enum* that enumerates elements for any embedded type *u* such that every element in *enum u* only appears once.

Given a function *occ* that counts the number of occurrences of a given element in a list and a comparison function *dec* that tells us whether or not two elements with type *u* are equal, the fact that every element in *enum u* is unique can be expressed as follows: for any element *val* with type *u*, *occ dec val (enum u) ≡ 1.*

Towards the end of this chapter, we will prove that this proposition indeed holds for all type code *u*, and we will show how *enum* can be used to determine the number of R1CS variables that will be allocated in the compilation process for an element of type *u*.

doi:10.6342/NTU202003013

3.1 Type Code

In this section, we are going to define the main datatype for the embedded type codes. Type codes are used to define the types that are allowed in the embedded DSL and is defined as an inductive-recursive definition[6] in Agda. Given any type *f*, the type codes are defined in an Agda module parameterized over *f* as follows:

Definition 3.1.1 (Type Code)**.**

```
data U : Set
⟦_⟧ : U → Set

data U where
  `One : U
  `Two : U
  `Base : U
  `Vec : (S : U) → ℕ → U
  `Σ `Π : (S : U) → (⟦ S ⟧ → U) → U

⟦ `One ⟧ = ⊤
⟦ `Two ⟧ = Bool
⟦ `Base ⟧ = f
⟦ `Vec ty x ⟧ = Vec ⟦ ty ⟧ x
⟦ `Σ fst snd ⟧ = Σ ⟦ fst ⟧ (λ f → ⟦ snd f ⟧)
⟦ `Π fst snd ⟧ = (x : ⟦ fst ⟧) → ⟦ snd x ⟧
```

By interpreting the type codes in *U* through ⟦_⟧, a subset of Agda types that are allowed in our EDSL can be obtained. In this thesis, we will instantiate *f* with types that represent finite fields since we will be compiling the EDSL into finite field constraints. Later on in this section we are going to define what it means for a type together with a set of operators to be a finite field.

For example, suppose that we have an Agda function $\mathit{fromBits} : \{n : \mathbb{N}\} \to \mathit{Vec}\ \mathit{Bool}\ n \to \mathbb{N}$ that transforms an n bit-encoded number into $\mathbb{N}$, then the type code $`\Sigma\ `\mathit{Two}\ (\lambda\, x_1 \to `\Sigma\ `\mathit{Two}\ (\lambda\, x_2 \to \ldots\ `\Sigma\ `\mathit{Two}\ (\lambda\, x_n \to `\mathit{Vec}\ `\mathit{Base}\ (\mathit{fromBits}\ (x_1 :: x_2 :: \ldots :: x_n :: [])))))$ expresses the type of vectors with their lengths encoded in n bits. Similarly, we can have matrices with the number of rows and columns encoded in $m + n$ bits respectively.

Intuitively, one can see that if the type f has only finitely many inhabitants, then for any $u : U$, $[\![\ u\ ]\!]$ also only has finitely many inhabitants, and as such, it is possible to obtain an enumeration of the elements in $[\![\ u\ ]\!]$. Formally, we define a type A to be finite or have finitely many inhabitants if there is an enumeration $l : \mathit{List}\ A$ of elements in A such that for any $x : A$, $x \in l$ and that any x in l only occurs once in l. We now define the pieces (list membership and element counting) that will be put together to form the definition of our finite type in Agda.

3.2 List Membership

Definition 3.2.1 (Any).

```
data Any {A : Set} (P : A → Set)
         : List A → Set where
  here  : ∀ {x xs} (px  : P x)        → Any P (x ∷ xs)
  there : ∀ {x xs} (pxs : Any P xs) → Any P (x ∷ xs)
```

Given a predicate $P : A \to \mathit{Set}$ and a list $l : \mathit{List}\ A$, $\mathit{Any}\ P\ l$ holds if there is at least one element $m : A$ in l such that $P\ m$ holds.

With the *Any* datatype defined, we now proceed to define the membership relation. Given a type A together with an equivalence relation $\_\approx\_$, the membership relation is defined as follows:

Definition 3.2.2 (_∈_).

```
_∈_ : A → List A → Set
x ∈ xs = Any (x ≈_) xs
```

Unless otherwise explicitly stated, _∈_ will be used with propositional equality.

3.3 Counting Number of Occurrences

When proving the soundness of the compiler, some of the steps require us to prove that elements in *enum u* are unique. To facilitate the development of such proofs (and to define what it means for a type to have finitely many inhabitants), we define a function *occ* that counts the number of occurrences of an element in a list.

Defining *occ* requires us to have the ability to determine whether or not propositional equality holds between two inhabitants of a specific type. This is captured with the following definitions:

Definition 3.3.1 (Dec)**.** *Decidability of a proposition P*

```
data Dec (P : Set) : Set where
  yes : (p : P) → Dec P
  no  : (¬p : ¬ P) → Dec P
```

Dec P holds if either *P* is true or ¬ *P* is true.

Definition 3.3.2 (Decidable)**.**

```
Decidable : {A B : Set} → (A → B → Set) → Set
Decidable _~_ = ∀ x y → Dec (x ~ y)
```

If *Decidable* holds for some equality _~_ : A → A → Set, this means that for any elements *x y* : *A*, we can decide whether or not *x* and *y* are propositionally equal.

Equipped with the definition of *Decidable*, *occ* is defined as follows.

Definition 3.3.3 (occ)**.** *Number of times an element appears in a list (up to propositional equality).*

```
occ : ∀ {A : Set}
    → (Decidable {A = A} _≡_) → A → List A → ℕ
occ dec a [] = 0
occ dec a (x ∷ l) with dec a x
... | yes p = suc (occ dec a l)
... | no ¬p = occ dec a l
```

doi:10.6342/NTU202003013

With list membership and element counting defined, we now define *Finite* in the following section.

3.4 Finite Types

Given a type f : *Set*, the predicate *Finite* is defined as an enumeration of all elements of f such that any inhabitant of f only appears in the enumeration once (up to propositional equality).

Definition 3.4.1 (Finite)**.**

```
record Finite (f : Set) : Set where
  field
    elems : List f
    size : ℕ
    a∈elems : (a : f) → a ∈ elems
    occ-1 : (a : f) (dec : Decidable _≡_)
              → occ dec a elems ≡ 1
    size≡len-elems : size ≡ length elems
```

Note that given an arbitrary type f and $a\ b$: *Finite* f, it is not necessarily the case that $a \equiv_{Finite\ f} b$ since the enumeration in a can be a permutation of the enumeration in b.

Our target compilation type R1CS comprises prime field elements and variables. After *Finite* is defined, we now define what it means for a type to be an algebraic field.

3.5 Field

Definition 3.5.1 (Field)**.** *Field f is defined as a record consisting of an addition operator _+_, a multiplication operator _*_, an additive unit zero, a multiplicative unit one, an additive inverse operation -_, and a multiplicative inverse 1/_.*

```
record Field (f : Set) : Set where
  field
```

```
    _+_ _*_ : f → f → f
    -_ : f → f
    1/_ : f → f
    zero : f
    one : f
```

(Note: _+_, _*_, -_, 1/_, zero, one are renamed to be _+F_, _*F_, -F_, 1F/_, zerof, onef respectively in the later chapters of this thesis)

Definition 3.5.2 (IsField)**.** *Field axioms.*

```
record IsField (f : Set) (field' : Field f)
         : Set where
    open Field field'
    field
      +-identityˡ : ∀ x → zero + x ≡ x
      +-identityʳ : ∀ x → x + zero ≡ x
      +-comm : ∀ x y → x + y ≡ y + x
      *-comm : ∀ x y → x * y ≡ y * x
      *-identityˡ : ∀ x → one * x ≡ x
      *-identityʳ : ∀ x → x * one ≡ x
      +-assoc : ∀ x y z → ((x + y) + z)
                             ≡ (x + (y + z))
      *-assoc : ∀ x y z → ((x * y) * z)
                             ≡ (x * (y * z))
      +-invˡ : ∀ x → ((- x) + x) ≡ zero
      +-invʳ : ∀ x → (x + (- x)) ≡ zero
      *-invˡ : ∀ x → ¬ x ≡ zero → (1/ x) * x ≡ one
      *-invʳ : ∀ x → ¬ x ≡ zero → x * (1/ x) ≡ one
      *-distr-+ˡ : ∀ x y z → (x * (y + z))
                                ≡ ((x * y) + (x * z))
```

doi:10.6342/NTU202003013

```
    *-distr-+ʳ : ∀ x y z → ((y + z) * x)
                               ≡ ((y * x) + (z * x))
```

IsField f ops describes the conditions for a type *f* with the field operations *ops* to be a field.

With our embedded type universe and the definition of finite field defined, we now proceed to define list monad before we construct the enumeration function *enum* that enumerates elements of $\llbracket\ u\ \rrbracket$.

3.6 List Monad

Definition 3.6.1 (return)**.** *(List)*

```
return : ∀ {A : Set} → A → List A
return a = [ a ]
```

where [a] denotes the singleton list with only one element a in it.

Definition 3.6.2 (_>>=_)**.** *(List)*

```
_>>=_ : {A B : Set}
    → List A → (A → List B) → List B
[] >>= f = []
(x ∷ ma) >>= f = f x ++ (ma >>= f)
```

where ++ denotes list concatenation.

In order to reason about monadic programs written in list monad later on, we prove a couple of lemmas to fascilitate these proofs.

3.6.1 Properties of List Monad

Suppose that we have $l >\!\!>\!= f : \mathit{List}\ A$ for some A. What is a necessary and sufficient condition for us to know that a particular element y falls inside of $l >\!\!>\!= f$? If there is an element $x \in l$ such that $y \in f\,x$, then we know that it must also fall inside of $l >\!\!>\!= f$:

Lemma 3.6.1 (∈->>=).

```
∈->>= : ∀ {A B : Set} (l : List A)
    (f : A → List B) → ∀ x → x ∈ l
        → ∀ y → y ∈ f x → y ∈ l >>= f
```

Proof. By straightforward induction on the derivation of $x \in l$. ☐

Conversely, we have:

Lemma 3.6.2 (∈->>=⁻).

```
∈->>=⁻ : {A B : Set} (l : List A)
     (f : A → List B) → ∀ y → y ∈ l >>= f
     → ∃ (λ x → x ∈ l × y ∈ f x)
```

Proof. By straightforward induction on l. ☐

Now suppose that we've written the following Agda program to help with enumerating elements of some finite types A and B:

```
makeProducts : {A B : Set}
    → List A → List B → List (A × B)
makeProducts l₁ l₂ = do
  a ← l₁
  b ← l₂
  return (a , b)
```

where _×_ denotes the usual cartesian product type. It's obvious that an element $(x, y) : A \times B$ falls inside of $\mathit{makeProducts}\ l_1\ l_2$ when $x \in l_1$ and $y \in l_2$. The following lemma proves a generalized version of the above statement where B is dependent on A and the domain of _+_ ranges over arbitrary $(m : A)$ and $(n : B\ m)$.

Lemma 3.6.3 (∈l-∈l'-∈r).

```
∈l-∈l'-∈r : ∀ {A : Set} {B : A → Set} {C : Set}
    (l : List A)  (_+_ : (x : A) → B x → C)
    → ∀ x y → x ∈ l → (l' : (x : A) → List (B x))
    → y ∈ l' x → x + y ∈ (l >>= λ r →
                          l' r >>= λ rs →
                          return (r + rs))
```

Proof. Corollary of ∈->>=. ☐

In the later sections of this chapter, we are going to prove that elements in the enumeration produced by *enum* (which will be introduced later in this chapter) are unique. The following lemma is going to help with decomposing the number of occurrences of an element in the subparts of *enum* into simpler parts. Take the program *makeProducts* in Section 3.6.1 for example again, if every element of l_1 in *makeProducts* is unique, then the proposition $\forall\, x_1 \to \neg\, x \equiv x_1 \to \neg\, y \in f\, x_1$ (a premise of *occ*->>=) is satisfied for any $x \in l_1 : List\ A$ and $k : B$ such that $y = (x, k)$ and $f = \lambda\, a \to l_2 >\!\!>\!= \lambda\, b \to (a\ , b)$ (because the first projections of the tuples are distinct). And the following lemma is going to be applied when we are counting occurrences of elements that appear in the subparts of *enum*, which are constructed in a way that is similar to the above *makeProducts* example.

Lemma 3.6.4 (occ->>=).

```
occ->>= : ∀ {A B : Set}
   (decA : Decidable {A = A} _≡_)
   (decB : Decidable {B = B} _≡_)
   (l : List A) (f : A → List B) → ∀ x y →
   (prf : ∀ x₁ → ¬ x ≡ x₁ → ¬ y ∈ f x₁) →
   occ decB y (l >>= f) ≡ (occ decA x l * occ decB y (f x))
```

Proof. By straightforward induction on *l*. ☐

As the embedded DSL is compiled into R1CS, it is necessary to determine the R1CS representation of an element $e : [\![\ u\ ]\!]$. To achieve this, the number of variables that has

doi:10.6342/NTU202003013

to be allocated for e has to be determined. Before we define the function *tySize* that determines how many variables are allocated for an element of some $\llbracket u \rrbracket$, the enumeration function *enum* that enumerates elements of embedded types is defined first as we rely on *enum* to determine R1CS variable allocation.

3.7 Enumerating Elements of Embedded Types

For any finite type f, it is possible to define an enumeration function $\mathit{enum} : (u : U) \to \mathit{List}\ \llbracket u \rrbracket$ that enumerates all elements of $\llbracket u \rrbracket$ exactly once for all u since the base cases are finite. Most cases of *enum* are quite trivial. The only non-trivial case is the case of Π types.

3.7.1 Enumerating Elements of Embedded Pi Types

In order to enumerate elements of embedded Π types, a couple of auxiliary definitions are needed. Observe that when pattern matching is done on u, in the case of Π types (let $u =$ ‘$\Pi\ v\ x$), it is easy to generate a list of pairs of type $\mathit{List}\ (\Sigma\ \llbracket v \rrbracket\ (\lambda\, k \to \mathit{List}\ \llbracket x\ k \rrbracket))$ with the usual list monad by structural recursion that pairs possible inputs to the Π type with the possible outputs together:

```
pairs = enum v >>= \r -> return (r , enum (x r))
```

With these pairs defined, we now define a function *genFunc* that transforms these pairs into $\mathit{List}\ (\mathit{List}\ (\Sigma\ \llbracket u \rrbracket\ (\lambda\, v \to \llbracket x\ v \rrbracket)))$ (which represents a list of functions) that can then be further transformed into a list of functions:

Definition 3.7.1 (genFunc)**.**

```
genFunc : ∀ (u : U) (x : ⟦ u ⟧ → U)
    → List (Σ ⟦ u ⟧ (λ v → List ⟦ x v ⟧))
    → List (List (Σ ⟦ u ⟧ (λ v → ⟦ x v ⟧)))
genFunc u x [] = [ [] ]
genFunc u x (x₁ ∷ l) with genFunc u x l
```

```
... | rec = do
  r ← rec
  choice ← proj₂ x₁
  return ((proj₁ x₁ , choice) ∷ r)
```

To give a sense of what *genFunc* is doing, suppose that we are given a constant type family *fam* over *‘Two* that maps *false* and *true* to *‘Base*[1], and the list $l = [(false\ , [2, 3]), (true\ , [5, 6])]$ of type List (Σ ⟦ *‘Two* ⟧ (λ *v* → *List* ⟦ *fam v* ⟧)). This input list says that the functions that we are building can map *false* to either 2 or 3, and map *true* to either 5 or 6. By feeding these arguments into *genFunc*, we get the list of all possible input output pairs [[(*false*, 2), (*true*, 5)], [(*false*, 3), (*true*, 5)], [(*false*, 2), (*true*, 6)], [(*false*, 3), (*true*, 6)]].

The relationship between the elements in the output of *genFunc* and the input list of *genFunc* is captured by the following relation:

Definition 3.7.2 (FuncInst)**.**

```
data FuncInst (A : Set) (B : A → Set)
     : List (Σ A B) → List (Σ A (λ v → List (B v)))
            → Set where
  InstNil : FuncInst A B [] []
  InstCons : ∀ l l' → (a : A) (b : B a) (ls : List (B a))
        → b ∈ ls → (ins : FuncInst A B l l')
        → FuncInst A B ((a , b) ∷ l) ((a , ls) ∷ l')
```

An instance of *FuncInst A B xs ys* says that if x and y are the i-th elements of xs and ys respectively, then $proj_1\ x$ and $proj_1\ y$ are equal, and $proj_2\ x \in proj_2\ y$.

We show that *FuncInst* ⟦ *u* ⟧ (λ *z* → ⟦ *x z* ⟧) *f l* if and only if *f* ∈ *genFunc u x l* with the following two lemmas.

Lemma 3.7.1 (FuncInst→genFunc)**.**

---

[1] where ‘Base maps to say a prime field or natural numbers

doi:10.6342/NTU202003013

```
FuncInst→genFunc : ∀ (u : U) (x : ⟦ u ⟧ → U)
    (l : List (Σ ⟦ u ⟧ (λ v → List ⟦ x v ⟧)))
    (f : List (Σ ⟦ u ⟧ (λ v → ⟦ x v ⟧)))
    → FuncInst ⟦ u ⟧ (λ z → ⟦ x z ⟧) f l
    → f ∈ genFunc u x l
```

Proof. By induction on the derivation of *FuncInst*. The base case is trivial, and the inductive case is proved by straightforward application of ∈->>=. ☐

Lemma 3.7.2 (genFunc→FuncInst)**.**

```
genFunc→FuncInst : ∀ (u : U) (x : ⟦ u ⟧ → U)
    (l : List (Σ ⟦ u ⟧ (λ v → List ⟦ x v ⟧)))
    (f : List (Σ ⟦ u ⟧ (λ v → ⟦ x v ⟧)))
    → f ∈ genFunc u x l
    → FuncInst ⟦ u ⟧ (λ z → ⟦ x z ⟧) f l
```

Proof. By induction on *l*. ☐

genFunc also satisfies this property (which is captured by *FuncInst*):

Lemma 3.7.3 (genFuncProj₁)**.**

```
genFuncProj₁ : ∀ (u : U) (x : ⟦ u ⟧ → U)
    → (l : List (Σ ⟦ u ⟧ (λ v → List ⟦ x v ⟧)))
    → (x₁ : List (Σ ⟦ u ⟧ (λ v → ⟦ x v ⟧)))
    → x₁ ∈ genFunc u x l
    → map proj₁ x₁ ≡ map proj₁ l
```

Proof. By induction on *l*. The base case is trivial, and the inductive case can be proved with ∈->>=⁻ and IH. ☐

After *genFunc* is defined, we need to construct a function that transforms the output of *genFunc* into a list of actual functions. To do this, we first construct the following *piFromList* function that transforms a list of input output pairs into a partial function (from (*dom* : ⟦ *u* ⟧) to ⟦ *x dom* ⟧) by induction on the derivation of *dom* ∈ *enough*:

Definition 3.7.3 (piFromList)**.**

```
piFromList : ∀ (u : U) (x : ⟦ u ⟧ → U)
     → (enough : List ⟦ u ⟧)
     → (l : List (Σ ⟦ u ⟧ (λ v → ⟦ x v ⟧)))
     → (map proj₁ l ≡ enough)
     → (dom : ⟦ u ⟧)
     → dom ∈ enough → ⟦ x dom ⟧
piFromList u x .(d ∷ _) ((d , v) ∷ l) refl dom
    (here refl) = v
piFromList u x (._ ∷ rest) (x₁ ∷ l) refl dom
    (there dom∈enough)
        = piFromList u x rest l refl dom dom∈enough
```

and provided with a proof ∀ $x \to x \in eu$ that the enumeration of u is complete, we can get total functions out of *piFromList* as demonstrated in the following *listFuncToPi* function when the inputs are in sync. When *listFuncToPi* is given a list l : *List* (*List* (Σ ⟦ *u* ⟧ (λ *v* → ⟦ *x v* ⟧))) representing a list of Π functions, we can get a list of actual functions as its output (given that l is good enough):

Definition 3.7.4 (listFuncToPi)**.**

```
listFuncToPi : ∀ (u : U) (x : ⟦ u ⟧ → U)
     → (eu : List ⟦ u ⟧)
     → (∀ elem → elem ∈ eu)
     → (l : List (List (Σ ⟦ u ⟧ (λ v → ⟦ x v ⟧))))
     → (∀ elem → elem ∈ l → map proj₁ elem ≡ eu)
     → List ⟦ `Π u x ⟧
listFuncToPi u x eu ∈eu [] proj₁l≡eu = []
listFuncToPi u x eu ∈eu (l ∷ l₁) proj₁l≡eu
  = (λ dom → piFromList u x eu l (proj₁l≡eu l (here refl))
                dom (∈eu dom))
```

```
        ∷ listFuncToPi u x eu ∈eu l₁
               (λ m m∈l → proj₁l≡eu m (there m∈l))
```

From the construction of $listFuncToPi$, we can see that the following lemma holds:

Lemma 3.7.4 (f∈listFuncToPi)**.**

```
f∈listFuncToPi :
   ∀ (u : U) (x : ⟦ u ⟧ → U)
   (eu : List ⟦ u ⟧)
   (∈eu : ∀ (elem : ⟦ u ⟧) → elem ∈ eu)
   (funcs : List (List (Σ ⟦ u ⟧ (λ v → ⟦ x v ⟧))))
   (func : List (Σ ⟦ u ⟧ (λ v → ⟦ x v ⟧)))
   (eq : (x₁ : List (Σ ⟦ u ⟧ (λ v → ⟦ x v ⟧))) →
               x₁ ∈ funcs → map proj₁ x₁ ≡ eu)
   (f : ⟦ `Π u x ⟧)
   → (mem : func ∈ funcs)
   → f ≡ (λ d → piFromList u x eu func
             (eq func mem) d (∈eu d))
   → f ∈ listFuncToPi u x eu ∈eu funcs eq
```

Proof. By straightforward induction on the derivation of *func* ∈ *funcs*. ☐

Similarly, we can define a function *piToList* that transforms an element of an embedded Π type back into a list of input/output pairs:

Definition 3.7.5 (piToList)**.**

```
piToList : ∀ (u : U) (x : ⟦ u ⟧ → U)
     → (eu : List ⟦ u ⟧) → (f : ⟦ `Π u x ⟧)
     → List (Σ ⟦ u ⟧ λ v → ⟦ x v ⟧)
piToList u x [] f = []
piToList u x (x₁ ∷ eu) f = (x₁ , f x₁) ∷ piToList u x eu f
```

Having both *piFromList* and *piToList* defined, now we prove that *piFromList* is both a left inverse and a right inverse of *piToList* (under good enough conditions).

In order to prove *piFromList∘piToList≗id*, we first prove an auxilliary lemma *piFromList∘piToList≗idAux* (since if we directly prove *piFromList∘piToList≗id* by induction on *eu*, the premise ∀ *elem* → *elem* ∈ *eu* no longer holds, and the induction fails).

Lemma 3.7.5 (piFromList∘piToList≗idAux)**.**

```
piFromList∘piToList≗idAux : ∀ (u : U) (x : ⟦ u ⟧ → U)
     (eu : List ⟦ u ⟧)
     (f :  ⟦ `Π u x ⟧)
     (p : map proj₁ (piToList u x eu f) ≡ eu)
     (t : ⟦ u ⟧) (t∈eu : t ∈ eu)
     → f t ≡ piFromList u x eu (piToList u x eu f) p t t∈eu
```

Proof. By induction on the derivation of *t* ∈ *eu*. ☐

Corollary 3.7.6 (piFromList∘piToList≗id)**.**

```
piFromList∘piToList≗id : ∀ (u : U) (x : ⟦ u ⟧ → U)
    (eu : List ⟦ u ⟧)
    (∈eu : ∀ elem → elem ∈ eu) (f :  ⟦ `Π u x ⟧)
    (p : map proj₁ (piToList u x eu f) ≡ eu)
    → ∀ (t : ⟦ u ⟧)
    → f t ≡ piFromList u x eu (piToList u x eu f)
                p t (∈eu t)
```

piFromList∘piToList≗id says that *piFromList* is a left inverse of *piToList*.

Proof. Corollary of *piFromList∘piToList≗idAux*. ☐

In order to prove that *piFromList* is a right inverse of *piToList* (under good enough conditions), we need the following lemma:

Lemma 3.7.7 (piFromListLem)**.**

doi:10.6342/NTU202003013

```
piFromListLem : ∀ (u : U) (x : ⟦ u ⟧ → U)
   (dec : ∀ {u} → Decidable {A = ⟦ u ⟧} _≡_)
   (x₁ : ⟦ u ⟧) (x₂ : Σ ⟦ u ⟧ (λ v → ⟦ x v ⟧))
   (px : proj₁ x₂ ≡ x₁)
   (eu :  List ⟦ u ⟧)
   (l : List (Σ ⟦ u ⟧ (λ v → ⟦ x v ⟧)))
   (uniq : occ dec x₁ eu ≡ 1) (p : map proj₁ l ≡ eu)
   → (prf : x₁ ∈ eu) (prf' : x₂ ∈ l)
   → (x₁ , piFromList u x eu l p x₁ prf) ≡ x₂}
```

piFromListLem says that the function that we get from *piFromList* actually corresponds to the input output pairs *l* when all elements of *eu* are unique and the first components of the elements in *l* correspond to *eu*.

Proof. By straightforward induction on the derivation of $x_1 \in eu$ followed by a case analysis on the derivation of $x_2 \in l$. ☐

We first prove an auxiliary lemma *piToList∘piFromList≡idAux* in order to do induction on the list *eu* given to *piToList*.

Lemma 3.7.8 (piToList∘piFromList≡idAux).

```
piToList∘piFromList≡idAux : ∀ (u : U) (x : ⟦ u ⟧ → U)
   (dec : ∀ {u} → Decidable {A = ⟦ u ⟧} _≡_)
   (eu : List ⟦ u ⟧)
   (∈eu : ∀ elem → elem ∈ eu)
   (eu' eu'' : List ⟦ u ⟧)
   (eq : eu'' ++ eu' ≡ eu)
   (l l' l'' : List (Σ ⟦ u ⟧ (λ v → ⟦ x v ⟧)))
   (lenEq : length eu' ≡ length l')
   (eq' : l'' ++ l' ≡ l)
   (uniq : ∀ v → occ dec v eu ≡ 1)
```

```
   (p : map proj₁ l ≡ eu)
   → piToList u x eu'
       (λ dom → piFromList u x eu l p dom (∈eu dom)) ≡ l'
```

Proof. By induction on *eu'* followed by case analysis on *l'*. The inductive case can be proved by applying IH and *piFromListLem*. ☐

After proving the auxiliary lemma *piToList∘piFromList≡idAux*, we can recover *piToList∘piFromList≡id* as a corollary of the auxiliary lemma.

Lemma 3.7.9 (piToList∘piFromList≡id).

```
piToList∘piFromList≡id : ∀ (u : U) (x : ⟦ u ⟧ → U)
   (dec : ∀ {u : U} → Decidable {A = ⟦ u ⟧} _≡_)
   (eu : List ⟦ u ⟧)
   (∈eu : ∀ elem → elem ∈ eu)
   (l : List (Σ ⟦ u ⟧ (λ v → ⟦ x v ⟧)))
   (uniq : ∀ v → occ dec v eu ≡ 1)
   (p : map proj₁ l ≡ eu)
   → piToList u x eu
         (λ dom → piFromList u x eu l p dom (∈eu dom)) ≡ l
```

piToList∘piFromList≡id says that *piFromList* is a right inverse of *piToList*.

Proof. Corollary of *piToList∘piFromList≡idAux*. ☐

After defining *piToList* and *piFromList*, we also need the following lemma *map-proj₁->>=* when defining *enum* to prove that the inputs to *listFuncToPi* are consistent.

Lemma 3.7.10 (map-proj₁->>=).

```
map-proj₁->>= : ∀ {A : Set} {B : A → Set}
    → (l : List A) (f : (x : A) → B x)
    → map proj₁ (l >>= (λ r → (r , f r) ∷ [])) ≡ l
```

Proof. By straightforward induction on l. ☐

In order to obtain an enumeration of the embedded Π types through *listFuncToPi*, we need a proof that the enumeration of u is complete, and this indicates that pattern matching/induction has to be done on a slightly altered goal: $(u : U) \to \Sigma\,(List\ [\![\ u\ ]\!])\,(\lambda\, en \to \forall\, x \to x \in en)$. Alternatively, this kind of definitions in Agda can be defined as mutually recursive definitions (which is what is done in the Agda development). Here we will only show the definition of *enum* without the accompanying proof that the enumerations generated by *enum* are complete. Readers interested in the full definition of *enum* together with the completeness proof can check out Appendix A.

3.7.2 Defining Enumeration of Elements of Embedded Types

Most cases of *enum* are trivial, and in the case of Π types, *enum* is defined through the function *listFuncToPi*.

Definition 3.7.6 (enum)**.**

```
enum : (u : U) → List ⟦ u ⟧
enumComplete : ∀ (u : U) → (x : ⟦ u ⟧) → x ∈ enum u

enum `One = [ tt ]
enum `Two = false ∷ true ∷ []
enum `Base = Finite.elems finite
enum (`Vec u zero) = [ [] ]
enum (`Vec u (suc x)) = do
  r ← enum u
  rs ← enum (`Vec u x)
  return (r ∷ rs)
enum (`Σ u x) = do
  r ← enum u
  rs ← enum (x r)
```

```
  return (r , rs)
enum (`Π u x) =
   let pairs = do
              r ← enum u
              return (r , enum (x r))
       funcs = genFunc _ _ pairs
   in listFuncToPi u x (enum u) (enumComplete u) funcs
         (λ x₁ x₁∈genFunc →
             trans (genFuncProj₁ u x pairs x₁ x₁∈genFunc)
                (map-proj₁->>= (enum u) (enum ∘ x)))
enumComplete = {- definition omitted -}
```

3.7.3 Uniqueness of Elements in enum

Lemma 3.7.11 (enumUnique)**.**

```
enumUnique : ∀ (u : U) → (val : ⟦ u ⟧)
   → (dec : ∀ {u} → Decidable {A = ⟦ u ⟧} _≡_)
   → occ dec val (enum u) ≡ 1
```

Proof. By induction on u. Most cases are trivial and can be solved by applying *occ->>=* and induction hypothesis. The more interesting case is the case of Π types. ☐

In order to prove the case of Π types of *enumUnique*, we need some auxiliary lemmas.

The following lemma *genFuncUnique* says that if

- the list $l : List\ (\Sigma\ [\![\ u\ ]\!]\ (\lambda\ v \to List\ [\![\ x\ v\ ]\!]))$ given to *genFunc* has the property that for all $elem \in l$, every element in $proj_2\ elem$ is unique
- the first projections of l is equal to eu
- and that $piToList\ u\ x\ eu\ f$ is in $genFunc\ u\ x\ l$

then $piToList\ u\ x\ eu\ f$ only occurs once in $genFunc\ u\ x\ l$.

Lemma 3.7.12 (genFuncUnique)**.**

```
genFuncUnique : ∀ (u : U) (x : ⟦ u ⟧ → U)
   (dec : Decidable {A = List (Σ ⟦ u ⟧ (λ v → ⟦ x v ⟧))} _≡_)
   (dec' : ∀ v → Decidable {A = ⟦ x v ⟧} _≡_)
   (eu : List ⟦ u ⟧)
   (f : ⟦ `Π u x ⟧)
   → (l : List (Σ ⟦ u ⟧ (λ v → List ⟦ x v ⟧)))
   → map proj₁ l ≡ eu
   → (∀ elem → elem ∈ l → ∀ (t : ⟦ x (proj₁ elem) ⟧)
           → t ∈ proj₂ elem
           → occ (dec' (proj₁ elem)) t (proj₂ elem) ≡ 1)
   → FuncInst ⟦ u ⟧ (λ v → ⟦ x v ⟧) (piToList u x eu f) l
   → occ dec (piToList u x eu f) (genFunc u x l) ≡ 1
```

Proof. By straightforward induction on *eu*. ☐

occ-listFuncToPi says that the number of occurrences of a function *f* in *listFuncToPi u x eu ∈eu l eq* is equal to the number of occurrences of *piToList u x eu f* in *l* if every element in *eu* is unique.

Lemma 3.7.13 (occ-listFuncToPi)**.**

```
occ-listFuncToPi : ∀ (u : U) (x : ⟦ u ⟧ → U)
   (eu : List ⟦ u ⟧)
   (∈eu : ∀ elem → elem ∈ eu)
   (l : List (List (Σ ⟦ u ⟧ (λ v → ⟦ x v ⟧))))
   (eq : (elem : List (Σ ⟦ u ⟧ (λ v → ⟦ x v ⟧))) →
           elem ∈ l → map proj₁ elem ≡ eu)
   (dec : Decidable {A = ⟦ `Π u x ⟧} _≡_)
   (dec' : Decidable {A = List (Σ ⟦ u ⟧ (λ v → ⟦ x v ⟧))} _≡_)
   (dec'' : ∀ {u} → Decidable {A = ⟦ u ⟧} _≡_)
```

```
    (uniq : (v : ⟦ u ⟧) → occ dec'' v eu ≡ 1)
    (f : ⟦ `Π u x ⟧)
    → occ dec f (listFuncToPi u x eu ∈eu l eq)
          ≡ occ dec' (piToList u x eu f) l
```

Proof. By induction on l followed by case analysis of the following terms:

- *dec val* (λ *dom* → *piFromList u x eu l* (*eq l* (*here refl*)) *dom* (∈*eu dom*))
- *dec'* (*piToList u x eu val*) *l*

Impossible cases can be refuted by applying *piToList*∘*piFromList*. ☐

Lemma 3.7.14 (enumUnique)**.**

```
enumUnique : ∀ (u : U) → (val : ⟦ u ⟧)
  → (dec : ∀ {u : U} → Decidable {A = ⟦ u ⟧} _≡_)
  → occ dec val (enum u) ≡ 1
```

Proof. By induction on u. Most cases are trivial. The ‘Vec and ‘Σ cases are proved with *occ->>=*, and the ‘Π case is proved with *occ-listFuncToPi* and *genFuncUnique*. ☐

3.8 Size of Type Codes

With the enumeration function of the type codes and its correctness defined and proved, the size of a type code representing how “large” the type corresponding to the type code is can now be defined. The size of a type code $u : U$ represents how much “storage” (i.e. variables in R1CS) is needed to store an element of type ⟦ u ⟧ (and not in the sense that the size of Set_i is too large to fit inside Set_i).

The following *tySize* function defines the size of a type code. *tySize* is used to specify the representation for the R1CS variable vector of some (*elem* : ⟦ u ⟧), and an R1CS representation of *elem* would be a vector of length *tySize u*.

```
maxTySizeOver : ∀ {u : U} → List ⟦ u ⟧ → (⟦ u ⟧ → U) → ℕ
tySumOver : ∀ {u : U} → List ⟦ u ⟧ → (⟦ u ⟧ → U) → ℕ
tySize : U → ℕ


tySize `One = 1
tySize `Two = 1
tySize `Base = 1
tySize (`Vec u x) = x * tySize u
tySize (`Σ u x) = tySize u + maxTySizeOver (enum u) x
tySize (`Π u x) = tySumOver (enum u) x

maxTySizeOver [] fam = 0
maxTySizeOver (x ∷ l) fam
    = max (tySize (fam x)) (maxTySizeOver l fam)

tySumOver [] x = 0
tySumOver (x₁ ∷ l) x = tySize (x x₁) + tySumOver l x
```

The size of ‘Σ *u x* is defined as the size of the first component plus the maximum size of *x* over all elements of ⟦ *u* ⟧. The size of ‘Π *u x* is defined as the sum of the size of *x elem* over all possible *elem* : ⟦ *u* ⟧. For example, given the following family of types *fam* over *‘Two*.

```
fam : ⟦ `Two ⟧ → U
fam false = `Vec `Base 5
fam true = `Vec `Two 2
```

The size of the type ‘Σ *‘Two fam* is calculated by summing together the size of the domain *‘Two* (which is 1) and the size of the largest type in the image of *fam* (which is the size of the type *‘Vec ‘Base 5*). And *tySize (‘Σ ‘Two fam)* = *tySize ‘Two* + *tySize (‘Vec ‘Base 5)* = 1

+ 5 = 6. The size of the type ‘Π *‘Two fam* is calculated by summing together the sizes of *fam false* and *fam true*, and *tySize (‘Π ‘Two fam) = tySize (fam false) + tySize (fam true)* = 5 + 2 = 7.

doi:10.6342/NTU202003013

Chapter 4

Source EDSL

We want to define a dependently typed domain specific language that targets R1CS. What should the language be like? Since R1CS allows us to express additive and multiplicative constraints, we also want to allow these in the source expression. By allowing these possibilities, we get the following datatype (parameterized over a type *f* representing a prime field):

```
data Arith : Set where
  Ind : ℕ -> Arith
  Lit : f → Arith
  Add Mul : Arith → Arith → Arith
```

where *Ind* accepts an R1CS variable, *Lit* accepts a literal. *Add* and *Mul* represent additive and multiplicative expressions respectively. Later on, we will allow users to add equality constraints, and so the user equipped with this construction will be able to do things like equal(Ind 10, Add (Lit 5) (Mul (Var 8) (Lit 9))) to express a constraint saying that $v_{10} = 5 + 9v_8$. Since we want dependent types in our language, we proceed to embed the type universe that we built in Chapter 3 into our *Arith* language.

4.1 Source

```
data Source : U → Set where
  Ind : ∀ {u} {m} → m ≡ tySize u → Vec ℕ m → Source u
  Lit : ∀ {u} → ⟦ u ⟧ → Source u
  Add Mul : Source `Base → Source `Base → Source `Base
```

The *Ind* and *Lit* cases are now expanded to allow dependent types. *Ind* is now a vector of variables (of length *tySize u* determined by how many R1CS variables an element of ⟦ *u* ⟧ is compiled into) representing an element of some type *u*, and *Lit* can now represent elements that are allowed by our embedded type universe. The *Add* and *Mul* cases now represent additive and multiplicative expressions over *Source ‘Base*, meaning that they are *Source* expressions over the prime field type of the R1CS constraints.

The *Source* datatype is designed to be used with the *RWS* monad, which is defined in the following section.

4.2 RWS Monad

An *RWS* monad consists of a read-only reader component, a write-only writer component, and a read-write state component. Given a reader type *R*, writer type *W*, state type *S*, an *RWS* monad type is defined as follows:

Definition 4.2.1 (RWSMonad).

```
RWSMonad : Set → Set
RWSMonad A = R × S → (S × W × A)
```

and with a unit element *mempty* : *W* together with a binary operation *mappend* : *W* ⟶ *W* ⟶ *W*, the monadic operations on *RWSMonad A* are defined as follows:

Definition 4.2.2 (_>>=_). *(RWSMonad) monadic bind*

```
_>>=_ : ∀ {A B : Set}
           → RWSMonad A
```

```
      → (A → RWSMonad B)
      → RWSMonad B
m >>= f = λ { (r , s) → let s' , w , a = m (r , s)
                            s'' , w' , b = f a (r , s')
                        in s'' , mappend w w' , b }
```

Definition 4.2.3 (return)**.** *(RWSMonad) Wrap a value into RWSMonad.*

```
return : ∀ {A : Set}
      → A → RWSMonad A
return a = λ { (r , s) → s , mempty , a }
```

get and *put* are used for reading/writing the state component:

Definition 4.2.4 (get)**.** *(RWSMonad) Copy the current state to the result.*

```
get : RWSMonad S
get = λ { (r , s) → s , mempty , s }
```

Definition 4.2.5 (put)**.** *(RWSMonad) Override the current state.*

```
put : S → RWSMonad ⊤
put s = λ _ → s , mempty , tt
```

where ⊤ is the unit type.

tell is used for writing stuff into the writer.

Definition 4.2.6 (tell)**.** *(RWSMonad) Write w to the writer component.*

```
tell : W → RWSMonad ⊤
tell w = λ { (r , s) → s , w , tt }
```

ask is used for accessing the reader:

Definition 4.2.7 (ask)**.** *(RWSMonad) Copy the reader value to the result.*

```
ask : RWSMonad R
ask = λ { (r , s) → s , mempty , r }
```

Definition 4.2.8 (local)**.** *(RWSMonad) Override the reader value of a monadic action with a user provided reader value.*

```
local : {A : Set} → R → RWSMonad A → RWSMonad A
local r p (r' , s) = p (r , s)
```

The following toy program demonstrates what using *RWSMonad* is like with $R = \mathbb{N}$, $W = \text{List}\ (\mathbb{N} \times \mathbb{N})$, $S = \mathbb{N}$ where *mempty* is [] and *mappend* is _++_:

```
{-# TERMINATING #-}
example : RWSMonad ℕ
example = do
  num ← ask
  case num of
    λ { 0 → get
      ; (suc n) → do
        acc ← get
        put (acc * num)
        tell ((num , acc) ∷ [])
        local n example
      }
```

In this example, the accumulating value is stored in the state parameter, the "current" number is stored in the reader component, and the writer component is used to store an execution log of the program. When supplied with an initial reader value of 5 and an initial state value of 1, the program produces the final state 120, the log (5 , 1) ∷ (4 , 5) ∷ (3 , 20) ∷ (2 , 60) ∷ (1 , 120) ∷ [] , and the result 120.

4.3 S-Monad

S-Monad is the main monad in which the user composes their source program. *S-Monad* is defined as an instance of *RWSMonad* where the reader component is instantiated with $\top$, the writer component with $List\ (\exists\ (\lambda\ u \to Source\ u \times Source\ u) \uplus (Map\ Var\ \mathbb{N} \to Map\ Var\ \mathbb{N})) \times List\ \mathbb{N}$ (where $A \uplus B$ is the disjoint union of A and B and $Map\ A\ B$ is the type of partial maps from A to B used in the solver to solve the generated R1CS constraints), the state component with $\mathbb{N}$, *mempty* with ([], []), and *mappend* with $(\lambda\ a\ b \to proj_1\ a +\!+\ proj_1\ b\ , proj_2\ a +\!+\ proj_2\ b)$ where the first component of the writer component stores a list of equality constraints and solver hints, and the second component stores a list of input variables.[1]

Definition 4.3.1 (S-Monad).

```
S-Monad : Set → Set
S-Monad A = ⊤ × ℕ
 → (ℕ × (List ((∃ (λ u → Source u × Source u))
                ⊎ (Map Var ℕ → Map Var ℕ)) × List ℕ) × A)
```

4.3.1 S-Monad Utilities

In order to allow the user to allocate one or more variables, we create the functions *newVar* and *newVars* as follows:

Definition 4.3.2 (newVar).

```
newVar : S-Monad Var
newVar = do
  v ← get
  put (1 + v)
  return v
```

Definition 4.3.3 (newVars).

```
newVars : ∀ (n : ℕ) → S-Monad (Vec Var n)
newVars zero = return []
```

---

[1] In the actual implementation, we used the method described in Hughes[10] to implement linear time list concatenation.

```
newVars (suc n) = do
  v ← newVar
  rest ← newVars n
  return (v ∷ rest)
```

assertEq writes a contraint that says that s_1 is equal to s_2 to the first component of the writer monad.

Definition 4.3.4 (assertEq)**.**

```
assertEq : ∀ {u} → Source u → Source u → S-Monad ⊤
assertEq {u} s₁ s₂
  = tell (inj₁ (u , s₁ , s₂) ∷ [] , [])
```

addHint writes a solver hint to the first component of the writer monad.

Definition 4.3.5 (addHint)**.**

```
addHint : (Map Var ℕ → Map Var ℕ) → S-Monad ⊤
addHint h = tell (inj₂ h ∷ [] , [])
```

new e allocates *tySize e* fresh variables.

Definition 4.3.6 (new)**.** *(S-Monad)*

```
new : ∀ (u : U) → S-Monad (Source u)
new e = do
  vec ← newVars (tySize e)
  return (Ind refl vec)
```

For example, if a programmer *A* wants to allocate two new boolean variables and assert them to be equal, *A* can write the following program to do so:

```
test : S-Monad (Source `Two)
test = do
```

```
  a ← new `Two
  b ← new `Two
  assertEq a b
  return a
```

newI e allocates *tySize e* fresh variables as well as adding these variables to the list of inputs.

```
newI : ∀ (u : U) → S-Monad (Source u)
newI e = do
  vec ← newVars (tySize e)
  tell ([] , toList vec)
  return (Ind refl vec)
```

Given a source program with type *Source* (‘Vec *u x*) and an index *i* : *Fin x* where *Fin* is an inductive family defined as the type of natural numbers less than *x*, we can get the *i*-th element of the vector:

```
getV : ∀ {u : U} {x : ⟦ u ⟧ → U}
    → Source (`Vec u x) → Fin x → Source u
getV {u} {suc x} (Ind refl x₁) f  with splitAt (tySize u) x₁
getV {u} {suc x} (Ind refl x₁) 0F | fst , snd = Ind refl fst
getV {u} {suc x} (Ind refl x₁) (suc f) | fst , snd
    = getV (Ind refl snd) f
getV (Lit (x ∷ x₁)) 0F = Lit x
getV (Lit (x ∷ x₁)) (suc f) = getV (Lit x₁) f
```

where *splitAt* (which splits an Agda vector into two) defined as follows splits a vector into two:

```
splitAt : ∀ {A : Set} → ∀ (m : ℕ){n : ℕ}
     → Vec A (m + n) → Vec A m × Vec A n
splitAt zero vec = [] , vec
```

doi:10.6342/NTU202003013

```
splitAt (suc m) (x ∷ vec)
    with splitAt m vec
... | fst , snd = x ∷ fst , snd
```

Given a function #_ that transforms ℕ into *Fin*, it's also possible to define an iteration function *iterM* (some type casts omitted):

```
iterM : ∀ {A : Set} (n : ℕ)
     → (Fin n → S-Monad A) → S-Monad (Vec A n)
iterM 0 act = return []
iterM (suc n) act = do
  r ← act (# n)
  rs ← iterM n act
  return (r ∷ rs)
```

It is also possible to apply a source expression over Π types. Given an expression of type ‘Π *u x*, in the case of *Lit*, we directly apply the argument to the function literal, and in the case of *Ind*, we return the segment of the R1CS variable vector corresponding to *x val* (the vector *vec* can be seen as the result of concatenating vectors representing elements of type ⟦ *x* e_1 ⟧, ⟦ *x* e_2 ⟧, ... ⟦ *x* e_n ⟧ where $[e_1, .. ,e_n]$ = *enum u*). For example, suppose that we have a type family *fam* over *‘Two* that maps *false* to *‘Vec ‘Base 2* and *true* to *‘Two*, and we have a *Source* expression *exp* = *Ind refl (2 ∷ 3 ∷ 4 ∷ [])* of type *Source* (‘Π *‘Two fam*). By "applying" *false* to *exp*, we get *Ind refl (2 ∷ 3 ∷ [])*, the portion of *exp* corresponding to an element of type *‘Vec ‘Base 2*, and by "applying" *true* to *exp*, we get *Ind refl (4 ∷ [])*, the portion of *exp* corresponding to an element of type *‘Two*.

```
appAux : ∀ {u : U} {x : ⟦ u ⟧ → U} → (eu : List ⟦ u ⟧)
    → (val : ⟦ u ⟧)
    → (mem : val ∈ eu) → Vec ℕ (tySumOver eu x)
    → S-Monad (Source (x val))
appAux {_} {x} .(val ∷ _) val (here refl) vec
    with splitAt (tySize (x val)) vec
```

```
... | fst , _ = return (Ind refl fst)
appAux {_} {x} (x₁ ∷ _) val (there mem) vec
    with splitAt (tySize (x x₁)) vec
... | _ , rest = appAux _ val mem rest

app : ∀ {u : U} {x : ⟦ u ⟧ → U} → Source (`Π u x)
    → (val : ⟦ u ⟧) → S-Monad (Source (x val))
app {u} (Ind refl x₁) val
   = appAux (enum u) val (enumComplete u val) x₁
app (Lit x) val = return (Lit (x val))
```

4.3.2 Examples

In this subsection, numerous examples of programs written in the embedded DSL will be shown.

MatrixMult

This example multiplies a 2 by 4 input matrix by a 4 by 3 input matrix and returns a 2 by 3 matrix where each element of the input matrices is a ‘Base element [14]. The matrix type is simply a vector of vectors in the embedded type universe.

```
`Matrix : U → ℕ → ℕ → U
`Matrix u m n = `Vec (`Vec u n) m

test : S-Monad (Source (`Matrix `Base 2 3))
test = do
  m₁ ← newI (`Matrix `Base 2 4)
  m₂ ← newI (`Matrix `Base 4 3)
  m₃ ← new (`Matrix `Base 2 3)
  iterM 2 (λ m → do
     iterM 3 (λ n → do
```

```
        vec ← iterM 4 (λ o → do
          let fstElem = getMatrix m₁ m o
              sndElem = getMatrix m₂ o n
          return (Mul fstElem sndElem))
        let setElem = getMatrix m₃ m n
        let r = foldl (const (Source `Base)) Add
                   (Lit (fieldElem nPrime 0)) vec
        assertEq r setElem))
  return m₃
```

where *fieldElem nPrime 0* constructs a field element 0 with the proof *nPrime* that the size of the field is prime, and *getMatrix m a b* gives us the element at the *a*-th row and the *b*-th column of *m*.

DependentProdSimple

In this example, we allocate a new vector of R1CS variables m_1 representing an element of type 'Π '*Two f*, and assert m_1 to be equal to the Agda function *func*. We then return the result of "applying" *true* to m_1 as the result of *test*.

```
N = {- a big prime number -}

postulate
  nPrime : Prime N

f : ⟦ `Two ⟧ → U
f t with t ≟B false
f t | yes p = `Two
f t | no ¬p = `Base

func : ⟦ `Π `Two f ⟧
func false = true
```

```
func true = fieldElem nPrime 12345

test : S-Monad (Source `Base)
test = do
  m₁ ← new (`Π `Two f)
  assertEq m₁ (Lit func)
  app m₁ true
```

DependentSumSimple

In this example, we allocate a new vector of R1CS variables m_1 representing an element of type ‘Σ *‘Two f*, and then assert m_1 to be equal to the literal (*false* , *true*). Finally, we return m_1 as the result of *test*.

```
f : ⟦ `Two ⟧ → U
f t with t ≟B false
f t | yes p = `Two
f t | no ¬p = `Vec `One 2

test : S-Monad (Source (`Σ `Two f))
test = do
  m₁ ← new (`Σ `Two f)
  assertEq m₁ (Lit (false , true))
  return m₁
```

Choice

Suppose that we have two possible computations with different inputs and outputs, and we wish to express the possibility that one of the computations is performed. It is possible to encode such a task with Σ types. For example, given a task A that computes the sum of two vectors with the same length and a task B that computes the sum of two ‘*Base* elements, we can define two family of types that tell us what the input and result types look like:

```
inputF : ⟦ `Two ⟧ → U
inputF false = `Vec `Base 2
inputF true = `Vec (`Vec `Base 2) 2

resultF : ⟦ `Two ⟧ → U
resultF false = `Base
resultF true = `Vec `Base 2
```

With these two family of types, we can define the input type to be ‘Σ ‘*Two inputF* and the output type to be ‘Σ ‘*Two resultF*, and assert the first components of the input and the output type to be equal to make sure that the input and output formats are consistent with the computation we chose.

In the *false* branch, we compute the addition of the two elements in the input vector (and fill the rest of the sigma type with zero):

```
computeFalse : Vec Var (maxTySizeOver (enum `Two) inputF)
  →  S-Monad (Vec Var (maxTySizeOver (enum `Two) resultF))
computeFalse vec
    with maxTySplit `Two false inputF vec
... | Σ₂₁ ∷ Σ₂₂ ∷ [] , blank = do
  r ← newVar
  filler ← newVar
  assertEq (Lit zerof) (var filler)
  assertEq (var r) (Add (var Σ₂₁) (var Σ₂₂))
  return (r ∷ filler ∷ [])
```

In the *true* branch, we compute the pairwise addition of the elements of the two input vectors:

```
computeTrue : Vec Var (maxTySizeOver (enum `Two) inputF)
  → S-Monad (Vec Var (maxTySizeOver (enum `Two) resultF))
```

```
computeTrue vec with maxTySplit `Two true inputF vec
... | Σ₂₁ ∷ Σ₂₂ ∷ Σ₂₃ ∷ Σ₂₄ ∷ [] , [] = do
  r₁ ← newVar
  r₂ ← newVar
  assertEq (var r₁) (Add (var Σ₂₁) (var Σ₂₃))
  assertEq (var r₂) (Add (var Σ₂₂) (var Σ₂₄))
  return (r₁ ∷ r₂ ∷ [])
```

We then define the following *assertEqWithCond* that conditionally asserts equality given a variable *cond* that always solves to 1 or 0.

```
assertEqWithCond :
  ∀ {n} → Var → Vec Var n → Vec Var n → S-Monad ⊤
assertEqWithCond cond [] [] = return tt
assertEqWithCond cond (x₁ ∷ vec₁) (x₂ ∷ vec₂) = do
  assertEq (Mul (var cond) (var x₁)) (Mul (var cond) (var x₂))
  assertEqWithCond cond vec₁ vec₂
```

Then we assert the conditional constraints for the *false* branch and the *true* branch in the *Ind* case, and directly compute the result in the *Lit* case:

```
compute : Source (`Σ `Two inputF)
  → S-Monad (Source (`Σ `Two resultF))
compute (Ind refl x₁) with splitAt (tySize `Two) x₁
compute (Ind refl x₁) | fst ∷ [] , snd = do
  result ← newVars (maxTySizeOver (enum `Two) resultF)
  addHint {- solver hint omitted -}
  r₁ ← computeFalse snd
  r₂ ← computeTrue snd
  fst=0 ← lnot fst
  assertEqWithCond fst r₂ result
  assertEqWithCond fst=0 r₁ result
```

```
  return (Ind refl (fst ∷ result))
compute (Lit (false , x₁ ∷ x₂ ∷ []))
  = return (Lit (false , (x₁ + x₂)))
compute (Lit (true , ((x₁₁ ∷ x₁₂ ∷ [])
                        ∷ (x₂₁ ∷ x₂₂ ∷ []) ∷ []))) =
  return (Lit (true , ((x₁₁ + x₂₁) ∷ (x₁₂ + x₂₂) ∷ [])))
```

where *lnot* is defined as the logical negation function:

```
lnot : Var → S-Monad Var
lnot n = do
  v ← S-Monad.newVar
  assertEq (Add (Lit onef) (Mul (Lit (- onef)) (var n))) (var v)
  return v
```

Finally, we put the above pieces together and get a program that depending on the input *choice*, computes either task *A* or task *B* (where $\Sigma\text{-}proj_1 : \forall\ \{u : U\}\ \{x : [\![\ u\ ]\!] \to U\} \to Source\ (`\Sigma\ u\ x) \to Source\ u$ computes the first projection of a Source (‘Σ u x)):

```
test : S-Monad (Source (`Σ `Two resultF))
test = do
  choice ← newI `Two -- v₁
  input ← newI (`Σ `Two inputF)
  result ← new (`Σ `Two resultF)
  assertEq (Σ-proj₁ result) choice
  assertEq (Σ-proj₁ input) choice
  r ← compute input
  assertEq result r
  return result
```

DynamicLengthMatrixMult

In addition to matrix multiplication with fixed lengths, it is also possible to have matrices with dynamic lengths. By stacking $m + n$ layers of ‘Σ, we can get the type of matrices where m bits are used to encode the number of rows and n bits are used to encode the number of columns (where *fromBits : {k : ℕ} → Vec Bool k → ℕ* and *fromBits (false ∷ true ∷ [])* = 01_2 = 1, *fromBits (true ∷ true ∷ [])* = 11_2 = 3 et cetera):

```
`DynMatrix : ℕ → ℕ → U
`DynMatrix m n =
  `Σ `Two (λ r₁ → ... `Σ `Two (λ rₘ →
    `Σ `Two (λ c₁ → ... `Σ `Two (λ rₙ →
      `Vec (`Vec `Base (fromBits (c₁ ∷ ... ∷ cₙ ∷ [])))
          (fromBits (r₁ ∷ ... rₘ ∷ []))))))
```

Suppose that we were given two vectors of R1CS variables that represent dynamically sized matrices a_1 : *Vec* ℕ *(tySize (‘DynMatrix m n))* and a_2 : *Vec* ℕ *(tySize (‘DynMatrix n o))*. In order to multiply these matrices together, we iterate over all possible sizes of a_1 and a_2. For example, suppose that m = n = o = 2, meaning that the number of rows and columns of a_1 and a_2 are encoded with 2 bits. We range over all of these possibilities and add conditional equality constraints (with *assertEqWithCond* in the Choice example) that assert the resulting matrix to be the product of the two input matrices for all possible sizes that can be encoded with the specified bits. In order to assert these conditional constraints, we added solver hints for the result of the dynamic matrix multiplication for the solver to successfully solve the conditional constraints.

doi:10.6342/NTU202003013

Chapter 5

Compiling Programs From Source to R1CS

Recall from Chapter 4 that when a user writes a program in *S-Monad*, the user is essentially building equality constraints over *Source* programs. Our task here is to transform these equality constraints into R1CS constraints.

In this chapter, I will first introduce the basic constructs used for writing the compiler, including the main compiler monad instance, and the basic logic functions. Then we will introduce the main compilation functions.

Since we will be using list builders[10] for linear time list concatenation in the actual implementation of the compiler monad, in order to reason about program properties abstractly on the level of the monadic interface, we chose to impose additional invariants[7][1] on the writer component in the compiler monad as lists are represented as endomorphisms between lists and without additional invariants, arbitrary endomorphisms between lists (and not just list builders) will be allowed (which is undesirable).

Because of the aforementioned problems, we choose to implement a new monad that incorporates this additional invariant in our implementation which will be described in the next section. Readers not interested in the details of how the invariant is embedded in the compiler can choose to skip the next section and proceed to the rest of the chapter.

5.1 RWSInvMonad

Given a reader type R, a writer type W, a state type S, a predicate $P : W \rightarrow Prop$, a unit element $mempty : W$, and a binary operation $mappend : W \rightarrow W \rightarrow W$ such that

- $P\text{-}mempty : P\ mempty$
- $P\text{-}mappend : \forall\ \{a\ b : W\} \rightarrow P\ a \rightarrow P\ b \rightarrow P\ (mappend\ a\ b)$

the RWSInvMonad is defined as follows (Σ* are variants of Σ where some of the components of Σ are inhabitants of types that live in *Prop* instead of *Set*):

Definition 5.1.1 (RWSInvMonad)**.**

```
RWSInvMonad : Set → Set
RWSInvMonad A
   = R × S → Σ′ (S × W × A)
                (λ prod → P (proj₁ (proj₂ prod)))
```

together with the associated operations

Definition 5.1.2 (_>>=_)**.** *(RWSInvMonad)*

```
_>>=_ : ∀ {A B : Set}
              → RWSInvMonad A
              → (A → RWSInvMonad B)
              → RWSInvMonad B
m >>= f = λ { (r , s) →
                let (s' , w , a) , inv = m (r , s)
                     (s'' , w' , b) , inv' = f a (r , s')
                 in (s'' , mappend w w' , b) ,
                          P-mappend inv inv' }
```

Definition 5.1.3 (_>>_)**.** *(RWSInvMonad)*

```
_>>_ : ∀ {A B : Set}
              → RWSInvMonad A
              → RWSInvMonad B
              → RWSInvMonad B
a >> b = a >>= λ _ → b
```

Definition 5.1.4 (return). *(RWSInvMonad)*

```
return : {A : Set}
              → A → RWSInvMonad A
return a = λ { (r , s) → (s , mempty , a) , P-mempty }
```

Definition 5.1.5 (get). *(RWSInvMonad)*

```
get : RWSInvMonad S
get = λ { (r , s) → (s , mempty , s) , P-mempty }
```

Definition 5.1.6 (gets). *(RWSInvMonad)*

```
gets : {A : Set} → (S → A) → RWSInvMonad A
gets f = do
  r ← get
  return (f r)
```

Definition 5.1.7 (put). *(RWSInvMonad)*

```
put : S → RWSInvMonad T
put s = λ _ → (s , mempty , tt) , P-mempty
```

Definition 5.1.8 (tell). *(RWSInvMonad)*

```
tell : (w : W) → (pw : P w) → RWSInvMonad T
tell w pw = λ { (r , s) → (s , w , tt) , pw }
```

Definition 5.1.9 (ask). *(RWSInvMonad)*

```
ask : RWSInvMonad R
ask = λ { (r , s) → (s , mempty , r) , P-mempty }
```

Definition 5.1.10 (asks)**.** *(RWSInvMonad)*

```
asks : {A : Set} → (R → A) → RWSInvMonad A
asks f = do
  r ← ask
  return (f r)
```

Definition 5.1.11 (local)**.** *(RWSInvMonad)*

```
local : ∀ {A : Set} → R
   → RWSInvMonad A → RWSInvMonad A
local r m (r' , s) = m (r , s)
```

The writer invariant that we will use for the compiler is as follows (we will be using two list builders):

Definition 5.1.12 (WriterInvariant)**.**

```
WriterInvariant : (List R1CS → List R1CS) ×
                      (List R1CS → List R1CS)
                  → Set
WriterInvariant = λ builder → ∀ x →
   proj₁ builder x ≡ (proj₁ builder []) ++ x ×
   proj₂ builder x ≡ (proj₂ builder []) ++ x
SquashedWriterInvariant = λ b → Squash (WriterInvariant b)
```

where *Squash* (*WriterInvariant b*) is the propositionally squashed type of *WriterInvariant b* in *Prop*. *Squash* is defined as the following datatype in Agda:

```
data Squash (A : Set) : Prop where
  sq : A → Squash A
```

doi:10.6342/NTU202003013

The invariant expresses the proposition that when a list *x* is applied to *builder*, it is the same as first applying the empty list to *builder*, then appending the resulting list with *x*. Take the following builder function for example: if *build* = (λ a → “hello” ++ a) and we apply a list *x* to *build*, we get “hello” ++ *x*. Which is equal to *build [] ++ x*.

In order to fascilitate the development of the proofs of the compiler, the following projection functions are defined:

Definition 5.1.13 (output)**.**

```
output :
    {S W A : Set} {P : W → Prop}
    → Σ′ (S × W × A) (λ prod → P (proj₁ (proj₂ prod)))
    → A
output ((s , w , a) , _) = a
```

Definition 5.1.14 (writerOutput)**.**

```
writerOutput :
  {S W A : Set} {P : W → Prop}
  → Σ′ (S × W × A) (λ prod → P (proj₁ (proj₂ prod)))
  → W
writerOutput ((s , w , a) , _) = w
```

Definition 5.1.15 (varOut)**.**

```
varOut : ∀ {S W A : Set} {P : W → Prop}
    → Σ′ (S × W × A) (λ prod → P (proj₁ (proj₂ prod)))
    → S
varOut ((s , _ , _) , _) = s
```

Definition 5.1.16 (writerInv)**.**

```
writerInv : {S W A : Set} {P : W → Prop d}
    → (p : Σ′ (S × W × A)
```

```
       (λ prod → P (proj₁ (proj₂ prod))))
  → P (proj₁ (proj₂ (fst p)))
writerInv ((s , w , a) , inv) = inv
```

In the rest of this chapter, we will proceed as if we are using *RWSMonad* to avoid unnecessary clutter.

5.2 SI-Monad

SI-Monad is the main monad that is used for writing the compiler. Since the generated R1CS constraints have to be solved by the solver, the constraints are ordered in a way that our solver can work with. This is done by postponing all type constraints, and adding hints that help the solver solve the contraints. The generated constraints are grouped into two groups: normal constraints and postponed constraints, and the solver tries to solve the constraints generated in normal mode first before trying to solve the constraints generated in postponed mode.

Definition 5.2.1 (WriterMode)**.**

```
data WriterMode : Set where
  NormalMode : WriterMode
  PostponedMode : WriterMode
```

Given a type f : *Set* and functions $fTo\mathbb{N} : f \to \mathbb{N}$, $\mathbb{N}toF : \mathbb{N} \to f$ such that (*field'* : *Field f*) and (*finite* : *Finite f*), *SI-Monad* is defined as an instance of *RWSMonad* where

- *WriterMode* is the first reader component of *SI-Monad*, and is used to choose between adding contraints in normal mode and adding constraints in postponed mode. The second reader component is the prime number chosen for the size of the prime field.

- The writer type used in *SI-Monad* is *List R1CS* × *List R1CS*, the first component of which is used to store the normal mode constraints and the second component is used to store the postponed mode constraints. *mempty* is defined to be a pair of empty lists, and *mappend* is defined to be pairwise list concatenation.

- The state type used in *SI-Monad* is a natural number that indicates the lowest unused variable.

Definition 5.2.2 (SI-Monad).

```
SI-Monad : Set → Set
SI-Monad A =
  (WriterMode × ℕ) × ℕ →
     (ℕ × (List R1CS × List R1CS) × A)
```

With *SI-Monad* defined, the basic utilities used for writing the compiler can now be defined:

5.3 Basic Utilities

Definition 5.3.1 (addWithMode). *(SI-Monad) Produce either a normal mode constraint or a postponed mode constraint.*

```
addWithMode : R1CS → WriterMode → SI-Monad ⊤
addWithMode w NormalMode =
   tell ((λ x → [ w ] ++ x) , id) (sq (λ x → refl , refl))
addWithMode w PostponedMode =
   tell (id , λ x → [ w ] ++ x) (sq (λ x → refl , refl))
```

For example, *addWithMode (IAdd 0 ((5 , 1) ∷ (6 , 2) ∷ [])) NormalMode*[1] adds a constraint in normal mode that says that $5v_1 + 5v_2 = 0$.

Definition 5.3.2 (withMode). *(SI-Monad) Override the execution of an SI-Monad action with a given WriterMode.*

---

[1] Field elements are constructed with fieldElem, but for simplicity's sake, we are using numeric literals to denote field elements here.

```
withMode : ∀ {A : Set} → WriterMode
    → SI-Monad A → SI-Monad A
withMode m act = do
  prime ← asks proj₂
  local (m , prime) act
```

Definition 5.3.3 (add)**.** *(SI-Monad) Produce a constraint with the current WriterMode.*

```
add : R1CS → SI-Monad ⊤
add w' = do
  m ← asks proj₁
  addWithMode w' m
```

Similar to the *newVar* definitions in *S-Monad*, we create simple wrapper functions *new* and *newVarVec* that are used to allocate variables:

Definition 5.3.4 (new)**.** *(SI-Monad) Allocate a new variable.*

```
new : SI-Monad Var
new = do
  v ← get
  put (1 +ℕ v)
  return v
```

where *Var* is defined as ℕ.

Definition 5.3.5 (newVarVec)**.** *Allocate n new variables.*

```
newVarVec : ∀ (n : ℕ) → SI-Monad (Vec Var n)
newVarVec nzero = return []
newVarVec (suc n) = do
  v ← new
  vs ← newVarVec n
  return (v ∷ vs)
```

We define a function that asserts a given R1CS variable to solve to 1.

Definition 5.3.6 (assertTrue)**.** *Assert that the solution of the variable is 1.*

```
assertTrue : Var → SI-Monad ⊤
assertTrue v = add (IAdd one ((- one , v) ∷ []))
```

Our target libsnark sets the value of variable 0 to 1. We follow this convention and always solve the variable 0 to 1 in our solver.

Definition 5.3.7 (trivial)**.** *A trivial utility that returns a variable that solves to 1.*

```
trivial : SI-Monad Var
trivial = do
  return 0
```

With the basic utility functions defined, the basic logic functions are defined as follows:

5.4 Basic Logic Functions

Definition 5.4.1 (neqz)**.** *not equal to zero*

```
neqz : Var → SI-Monad Var
neqz n = do
  v ← new
  v' ← new
  prime ← asks proj₂
  add (Hint (neqzHint prime n v v'))
  add (IMul one v n one v')
  add (IMul one v' n one n)
  return v'
```

neqz checks if the solution of the variable n is not equal to zero. Given a solution map *sol* that satisfies the constraints generated by *neqz*, if the solution of the variable n is zero in *sol*, *neqz* returns a variable that corresponds to 0 in *sol*. Otherwise, it returns a variable that corresponds to 1 in *sol*.

When no confusion arises, phrases like “a variable v solves to l” are used under the assumption that there is a solution S (sometimes also requiring that the variable 0 maps to 1 in S) to the generated constraints in the given context, and that in S, v corresponds to l.

neqzHint is a hint that helps the solver solve the generated *neqz* constraints. The first constraint says that the variable v times the variable n is equal to the variable v'. The second constraint says that the variable v' times n is equal to the variable n. Why is this definition correct? Given a solution *sol*, there are two cases to consider:

- If the solution of n is equal to zero, the solution of v' would also be zero by the first constraint, and the second constraint is vacuously satisfied.
- If the solution of n is not equal to zero, then the second constraint says that the solution of v' must be 1 (whereas the first constraint can be satisfied by picking the solution of v to be the multiplicative inverse of the solution of n.

The following logical operators *lor*, *land*, *lnot*, *limp* assume that the solutions of their respective input variables are boolean.

Given two boolean field elements a and b, a and b is defined as ab. The following definition *land* performs the logical and operation on R1CS variables.

Definition 5.4.2 (land). *logical and*

```
land : Var → Var → SI-Monad Var
land n₁ n₂ = do
  v ← new
  add (IMul one n₁ n₂ one v)
  return v
```

Given two boolean field elements a and b, a or b is defined as $a + b - ab$. The following definition *lor* performs the logical or operation on R1CS variables.

Definition 5.4.3 (lor). *logical or*

```
lor : Var → Var → SI-Monad Var
lor n₁ n₂ = do
  v ← new
  v' ← new
  add (IMul one n₁ n₂ one v)
  add (IAdd zero ((one , n₁) ∷ (one , n₂) ∷
          (- one , v) ∷ (- one , v') ∷ []))
  return v'
```

Given a field element *a*, *not a* is defined as *1 - a*.

Definition 5.4.4 (lnot)**.** *logical not*

```
lnot : Var → SI-Monad Var
lnot n₁ = do
  v ← new
  add (IAdd one ((- one , n₁) ∷ (- one , v) ∷ []))
  return v
```

Given two field elements *a* and *b*, *a* → *b* is defined as *(not a) or b*.

Definition 5.4.5 (limp)**.** *logical implication*

```
limp : Var → Var → SI-Monad Var
limp n₁ n₂ = do
  notN₁ ← lnot n₁
  lor notN₁ n₂
```

Besides the logical functions, we also need a couple of auxiliary compilation functions, which are defined in the following section.

5.5 Auxiliary Compilation Functions

varEqBaseLit checks if a variable is “equal” to a field element.

Definition 5.5.1 (varEqBaseLit).

```
varEqBaseLit : Var → f → SI-Monad Var
varEqBaseLit n l = do
  n-l ← new
  add (IAdd (- l) ((one , n) ∷ (- one , n-l) ∷ []))
  ¬r ← neqz n-l
  r ← lnot ¬r
  return r
```

The variable that *varEqBaseLit* returns solves to 1 if the solution of n is equal to *l* and solves to 0 otherwise.

anyNeqz checks if any variable in the given vector solves to a non-zero entry.

Definition 5.5.2 (anyNeqz).

```
anyNeqz : ∀ {n : ℕ} → Vec Var n → SI-Monad Var
anyNeqz [] = do
  v ← new
  add (IAdd zero ((one , v) ∷ []))
  return v
anyNeqz (x ∷ vec) = do
 r ← neqz x
 rs ← anyNeqz vec
 lor r rs
```

The variable that *anyNeqz* returns solves to 1 if the solution of the vector given to *anyNeqz* contains a nonzero entry and solves to 0 otherwise.

allEqz checks if all variables in the given vector solve to 0.

Definition 5.5.3 (allEqz).

```
allEqz : ∀ {n : ℕ} → Vec Var n → SI-Monad Var
allEqz vec = do
```

```
  ¬r ← anyNeqz vec
  r ← lnot ¬r
  return r
```

The variable that *allEqz* returns solves to 1 if the solution of the input vector variables to *allEqz* solves to a 0 vector.

Now the main compilation functions can be defined:

5.6 Main Compilation Functions

The following functions *piVarEqLit* and *varEqLit* define the comparison operations of a vector of variables to a literal. If the given vector solves to a low level representation of the given literal, the returned variable solves to 1. Otherwise, it solves to 0.

Definition 5.6.1 (piVarEqLit, varEqLit)**.**

```
piVarEqLit : ∀ (u : U) (x : ⟦ u ⟧ → U) (eu : List ⟦ u ⟧)
     → Vec Var (tySumOver eu x) → ⟦ `Π u x ⟧ → SI-Monad Var
varEqLit : ∀ (u : U) → Vec Var (tySize u) → ⟦ u ⟧ → SI-Monad Var

piVarEqLit u x [] vec f = trivial
piVarEqLit u x (x₁ ∷ eu) vec f
    with splitAt (tySize (x x₁)) vec
... | fst , snd = do
  r ← varEqLit (x x₁) fst (f x₁)
  s ← piVarEqLit u x eu snd f
  land r s

varEqLit `One vec lit = allEqz vec
varEqLit `Two vec false = allEqz vec
varEqLit `Two (x ∷ vec) true = varEqBaseLit x one
```

```
varEqLit `Base (x ∷ vec) lit = varEqBaseLit x lit
varEqLit (`Vec u nzero) vec lit = trivial
varEqLit (`Vec u (suc x)) vec (l ∷ lit)
    with splitAt (tySize u) vec
... | fst , snd = do
  r ← varEqLit u fst l
  s ← varEqLit (`Vec u x) snd lit
  land r s
varEqLit (`Σ u x) vec (fst₁ , snd₁)
    with splitAt (tySize u) vec
... | fst , snd with maxTySplit u fst₁ x snd
... | vecₜ₁ , vecₜ₂ = do
  r ← varEqLit u fst fst₁
  s ← varEqLit (x fst₁) vecₜ₁ snd₁
  s' ← allEqz vecₜ₂
  and₁ ← land r s
  land and₁ s'
varEqLit (`Π u x) vec f = piVarEqLit u x (enum u) vec f
```

where *maxTySplit* is defined with *splitAt* (type cast omitted):

```
maxTySplit : ∀ (u : U) (val : ⟦ u ⟧) (x : ⟦ u ⟧ → U)
    → Vec Var (maxTySizeOver (enum u) x)
    → Vec Var (tySize (x val)) ×
      Vec Var (maxTySizeOver (enum u) x - tySize (x val))
maxTySplit u val x vec = splitAt (tySize (x val)) vec
```

In the Σ case of *varEqLit*, the input vector *vec* can be split into three parts: the first part represents the first component of the Σ type, the second part represents the second component of the Σ type, and the third part should be a vector that solves to the 0 vector.

In the Π case of *varEqLit*, given a function $f : \llbracket \text{‘}\Pi\ u\ x \rrbracket$, *varEqLit* (‘$\Pi$ *u x*) *vec* f is logically equivalent to *varEqLit* ($x\ u_1$) $vec_1\ (f u_1)$ && ... && *varEqLit* ($x\ u_k$) $vec_k\ (f u_k)$ where $vec = vec_1$ ++ ... ++ vec_k, *enum u* = [u_1, u_2, ... , u_k].

We will describe functions that generate type constraints in the next subsection.

5.6.1 Generating Type Constraints

In this subsection, we implement the main functions that generate type constraints for a type *u*.

Given a type *u* and *vec* : *Vec Var* (*tySize u*), *tyCond u vec* generates the constraints that tell us whether or not *vec* can be considered as a representation of some *elem* : ⟦ *u* ⟧.

Definition 5.6.2 (enumSigmaCond, enumPiCond, tyCond).

```
enumSigmaCond : ∀ {u : U} → List ⟦ u ⟧ → (x : ⟦ u ⟧ → U)
      → Vec Var (tySize u)
      → Vec Var (maxTySizeOver (enum u) x) → SI-Monad Var
enumPiCond : ∀ {u : U} → (eu : List ⟦ u ⟧) → (x : ⟦ u ⟧ → U)
      → Vec Var (tySumOver eu x) → SI-Monad Var
tyCond : ∀ (u : U) → Vec Var (tySize u) → SI-Monad Var

enumSigmaCond [] x v₁ v₂ = trivial
enumSigmaCond {u} (elem₁ ∷ enum₁) x v₁ v₂
    with maxTySplit u elem₁ x v₂
... | fst , snd = do
  eqElem₁ ← varEqLit u v₁ elem₁
  tyCons ← tyCond (x elem₁) fst
  restZ ← allEqz snd
  tyCons&restZ ← land tyCons restZ
  sat ← limp eqElem₁ tyCons&restZ
  rest ← enumSigmaCond enum₁ x v₁ v₂
  land sat rest
```

```
enumPiCond [] x vec = trivial
enumPiCond (x₁ ∷ eu) x vec
    with splitAt (tySize (x x₁)) vec
... | fst , rest = do
  r ← tyCond (x x₁) fst
  s ← enumPiCond eu x rest
  land r s
tyCond `One vec = allEqz vec
tyCond `Two vec = do
  isZero ← varEqLit `Two vec false
  isOne ← varEqLit `Two vec true
  lor isZero isOne
tyCond `Base vec = trivial
tyCond (`Vec u nzero) vec = trivial
tyCond (`Vec u (suc x)) vec
    with splitAt (tySize u) vec
... | fst , snd = do
  r ← tyCond u fst
  s ← tyCond (`Vec u x) snd
  land r s
tyCond (`Σ u x) vec
    with splitAt (tySize u) vec
... | fst , snd = do
  r ← tyCond u fst
  s ← enumSigmaCond (enum u) x fst snd
  land r s
tyCond (`Π u x) vec = enumPiCond (enum u) x vec
```

The variable that *tyCond u vec* returns solves to 1 if *vec* solves to a low level representation of some *elem* : ⟦ *u* ⟧, and solves to 0 otherwise.

With *tyCond* defined, *indToIR* can now be defined as a function that asserts that the solution of a given vector satisfies the type constraints of some type *u*.

Definition 5.6.3 (indToIR)**.**

```
indToIR : ∀ (u : U) → Vec Var (tySize u)
    → SI-Monad (Vec Var (tySize u))
indToIR u vec = do
  t ← tyCond u vec
  assertTrue t
  return vec
```

5.7 Compiling Source to R1CS

Our strategy for compiling equality constraints is to first transform the two source expressions in an equality constraint into their corresponding R1CS constraints (including the type constraints), then assert the two resulting vector of variables to solve to the same values.

Suppose that we have an equality constraint between the two *Source* expressions *Ind refl (3 ∷ [])* and *Add (Lit 5) (Mul (Lit 6) (Ind refl (7 ∷ [])))* of type *Source ‘Base*. We would transform the literals into *Ind*s by applying *varEqLit* recursively, and the resulting *Ind*s would then be pieced together with additive constraints for the additive expressions, multiplicative constraints for the multiplicative expressions, and finally the equality constraint for the equality of the two *Source* expressions. In the cases other than equality constraints over *‘Base*, we would add the corresponding type constraints for *Ind*s with *indToIR* and assert that the resulting variable of *indToIR* solves to 1 to make sure that the vector of variables in the *Ind*s are representations of a literal of the same type.

First we define the function *sourceToR1CS* that transforms source expressions into R1CS. In the case of literals, we transform them into R1CS variables with *litToInd* by first allocating a new R1CS variable vector, and then asserting that the literal and the vector of R1CS variables are “equal” (with *varEqLit*).

Definition 5.7.1 (litToInd).

```
litToInd : ∀ (u : U) → ⟦ u ⟧ → SI-Monad (Vec Var (tySize u))
litToInd u l = do
  vec ← newVarVec (tySize u)
  add (Hint (litEqVecHint u l vec))
  r ← varEqLit u vec l
  assertTrue r
  return vec
```

The main compilation function *sourceToR1CS* is then defined as follows:

Definition 5.7.2 (sourceToR1CS).

```
sourceToR1CS : ∀ (u : U) → Source u
    → SI-Monad (Vec Var (tySize u))
sourceToR1CS u (Ind refl x)
    = withMode PostponedMode (indToIR u x)
sourceToR1CS u (Lit x) = litToInd u x
sourceToR1CS `Base (Add source source₁) = do
  r₁ ← sourceToR1CS `Base source
  r₂ ← sourceToR1CS `Base source₁
  v ← new
  add (IAdd zero ((one , head r₁) ∷
                        (one , head r₂) ∷ (- one , v) ∷ []))
  return (v ∷ [])
sourceToR1CS `Base (Mul source source₁) = do
  r₁ ← sourceToR1CS `Base source
  r₂ ← sourceToR1CS `Base source₁
```

```
  v ← new
  add (IMul one (head r₁) (head r₂) one v)
  return (v ∷ [])
```

With the *sourceToR1CS* function defined, what is left is of compiling equality constraints is to assert the two components of the equality constraints to solve to the same values. We define the following function *assertVarEqVar* to do so.

Definition 5.7.3 (assertVarEqVar).

```
assertVarEqVar : ∀ (n : ℕ) → Vec Var n → Vec Var n → SI-Monad ⊤
assertVarEqVar .0 [] [] = return tt
assertVarEqVar .(suc _) (x ∷ v₁) (x₁ ∷ v₂) = do
  add (IAdd zero ((one , x) ∷ (- one , x₁) ∷ []))
  assertVarEqVar _ v₁ v₂
```

These components are then composed together into *compAssert* and *compAssertsHints*.

Definition 5.7.4 (compAssert).

```
compAssert : (∃ (λ u → Source u × Source u)) → SI-Monad ⊤
compAssert (u , s₁ , s₂) = do
  r₁ ← sourceToR1CS u s₁
  r₂ ← sourceToR1CS u s₂
  assertVarEqVar _ r₁ r₂
```

Definition 5.7.5 (compAssertsHints).

```
compAssertsHints :
  List (∃ (λ u → Source u × Source u)
           ⊎ (M.Map Var ℕ → M.Map Var ℕ))
  → SI-Monad ⊤
compAssertsHints [] = return tt
compAssertsHints (inj₁ x ∷ ls) = do
```

doi:10.6342/NTU202003013

```
  compAssert x
  compAssertsHints ls
compAssertsHints (inj₂ y ∷ ls) = do
  add (Hint y)
  compAssertsHints ls
```

The whole compilation process is then combined together with *compileSource*.

Definition 5.7.6 (compileSource)**.**

```
compileSource : ∀ (n : ℕ) (u : U) → (S-Monad (Source u))
    → Var × List R1CS × (Vec Var (tySize u) × List ℕ)
compileSource n u source =
  let v , (asserts , input) , output = source (tt , 1)
      ((v' , (cs₁ , cs₂) , outputVars) , inv) = (do
         compAssertsHints (asserts [])
         sourceToR1CS _ output) ((NormalMode , n) , v)
  in v' , cs₁ ++ cs₂ , outputVars , input []
```

Chapter 6

Formal Verification of the Compiler

In this chapter, we will describe how the translational soundness of the compiler is proved. There are two parts to the translational correctness of a compiler: soundness and completeness. Soundness is the property that if the generated constraints are satisfiable, then the solution must be correct, and completeness is the property that the generated constraints are satisfiable. We will only be proving the soundness of the compiler in this thesis.

Recall that in the compilation pipeline, we first execute a program written in S-Monad with some initial states, then the result is piped into *assertVarEqVar* and *sourceToR1CS* to generate the corresponding R1CS constraints. In this chapter, we will describe the formal verification of the soundness of the main compilation functions in detail.

Given functions $\mathbb{N}toF : \mathbb{N} \rightarrow f$ and $fTo\mathbb{N} : f \rightarrow \mathbb{N}$, the soundness of the compiler is proved under the following conditions:

1. _≟F_ : Decidable {A = f} _≡_
2. _≟U_ : ∀ {u} → Decidable {A = ⟦ u ⟧} _≡_
3. field' : Field f
4. isField : IsField f field'
5. finite : Finite f
6. ℕtoF-1≡1 : ℕtoF 1 ≡ onef

7. ℕtoF-0≡0 : ℕtoF 0 ≡ zerof
8. ℕtoF∘fToℕ≡ : ∀ x → ℕtoF (fToℕ x) ≡ x
9. prime : ℕ
10. isPrime : Prime prime
11. onef≠zerof : ¬ onef ≡ zerof
12. The function ℕtoF is additively and multiplicatively homomorphic. i.e.
    (a) ℕtoF-+hom : ∀ x y → ℕtoF (x + y) ≡ (ℕtoF x) + (ℕtoF y)
    (b) ℕtoF-*hom : ∀ x y → ℕtoF (x * y) ≡ (ℕtoF x) * (ℕtoF y)

where *isPrime* is a proof that *prime* is indeed a prime number, *onef* is the multiplicative identity in *field'*, and *zerof* is the additive identity in *field'*.

A solution to a set of R1CS constraints is defined as a partial map *List* (*Var* × ℕ) that maps variables to their corresponding values. Because the variables are mapped to natural numbers, it is necessary to have conversion functions (*fTo*ℕ : *f* → ℕ) (ℕ*toF* : ℕ → *f*) in order to define what it means to say that a *List* (*Var* × ℕ) is a solution to an R1CS constraint.

Before we define the soundness of *sourceToR1CS*, we need a couple of auxiliary definitions, including a semantic function on *Source* expressions, a lookup relation for R1CS variables in a partial map, and a value representation relation between a literal and a vector of natural numbers representing prime field elements. Our *sourceToR1CS* soundness lemma says that given a solution *sol* to the constraints generated by *sourceToR1CS*, then under good enough conditions, the result of the semantic function coincides (meaning that they are related by the value representation relation) with the solution of the vector of variables generated by *sourceToR1CS* in *sol*.

Since we are using natural numbers to represent prime field elements, we first define the following equivalence relation _≈_ that "quotients" elements that are mapped to the same values by the function ℕ*toF*. This relation will then be used to define the solution relation for a partial map of type *List* (*Var* × ℕ) to be considered a solution of an R1CS constraint.

Definition 6.0.1 (_≈_)**.** *_≈_ is the equivalence relation naturally induced by the function ℕtoF.*

```
_≈_ : ℕ → ℕ → Prop
x ≈ y = Squash (ℕtoF x ≡ ℕtoF y)
```

_≈_ is an equivalence relation since the underlying propositional equality is an equivalence relation:

```
≈-refl : ∀ {n} → n ≈ n
≈-sym : ∀ {m n} → m ≈ n → n ≈ m
≈-trans : ∀ {m n o} → m ≈ n → n ≈ o → m ≈ o
```

With _≈_ defined, next we define the lookup relation of one or more variables for a partial map of type *List* (*Var* × ℕ).

Definition 6.0.2 (ListLookup)**.**

```
data ListLookup : Var → List (Var × ℕ) → ℕ → Prop where
  LookupHere : ∀ v l n n' → n ≈ n'
    → ListLookup v ((v , n) ∷ l) n'
  LookupThere : ∀ v l n t
    → ListLookup v l n → ¬ v ≡ proj₁ t
    → ListLookup v (t ∷ l) n
```

Given a variable v : *Var*, a partial map *sol* : *List* (*Var* × ℕ), and a value n : ℕ, *ListLookup v sol n* holds if the first occurrence of v in *sol* maps to n. We generalize the above relation from a single variable to a vector of variables with the following relation *BatchListLookup*.

Definition 6.0.3 (BatchListLookup)**.**

```
data BatchListLookup : {n : ℕ} → Vec Var n → List (Var × ℕ)
     → Vec ℕ n → Prop where
```

doi:10.6342/NTU202003013

```
  BatchLookupNil : ∀ l → BatchListLookup [] l []
  BatchLookupCons : ∀ {len} v n (vec₁ : Vec Var len) vec₂ l
          → ListLookup v l n
          → BatchListLookup vec₁ l vec₂
          → BatchListLookup (v ∷ vec₁) l (n ∷ vec₂)
```

Recall from the definition of *R1CS* in Chapter 2 that an *R1CS* constraint is either an additive constraint (*IAdd*), a multiplicative constraint (*IMul*), a *Hint*, or a *Log*. We define a partial map *sol* : *List* (*Var* × ℕ) to be a solution to an additive constraint *IAdd* f_1 $((f_2, i_2)$ ∷ (f_3, i_3) ... ∷ []) if after looking up the variables i_2, i_3, ... in *sol*, the linear combination sums to zero (in the finite field). The solution of a multiplicative constraint is defined similarly. For the cases *Hint* and *Log*, we define any partial map *sol* : *List* (*Var* × ℕ) to be a solution of a *Hint* or a *Log* since they are not actual constraints. This is formally defined with the Agda definitions in the following section.

6.1 Solution of R1CS Constraints

Given a partial map *sol* : *List* (*Var* × ℕ) and a linear combination of variables *List* (*f* × Var), the value of the linear combination is defined with the following relation *LinearCombVal*.

Definition 6.1.1 (LinearCombVal)**.**

```
data LinearCombVal (sol : List (Var × ℕ)) :
    List (f × Var) → f → Prop where
  LinearCombValBase : LinearCombVal solution [] zerof
  LinearCombValCons : ∀ coeff var varVal {l} {acc}
      → ListLookup var solution varVal
      → LinearCombVal solution l acc
      → LinearCombVal solution ((coeff , var) ∷ l)
              ((coeff *F ℕtoF varVal) +F acc)
```

(where +F, *F are the additive and multiplicative field operations)

A *List* (*Var* × ℕ) is a solution to an R1CS constraint if the following relation holds.

Definition 6.1.2 (R1CSSolution)**.**

```
data R1CSSolution (solution : List (Var × ℕ))
        : R1CS → Prop where
  addSol : ∀ {coeff} {linComb} {linCombVal}
        → LinearCombVal solution linComb linCombVal
        → linCombVal +F coeff ≡ zerof
        → R1CSSolution solution (IAdd coeff linComb)
  multSol : ∀ a b bval c cval d e eval
        → ListLookup b solution bval
        → ListLookup c solution cval
        → ListLookup e solution eval
        → ((a *F (ℕtoF bval)) *F (ℕtoF cval))
              ≡ (d *F (ℕtoF eval))
        → R1CSSolution solution (IMul a b c d e)
  hintSol : ∀ f → R1CSSolution solution (Hint f)
  logSol : ∀ s → R1CSSolution solution (Log s)
```

Since the writer component of *SI-Monad* is defined as a pair of *List R1CS*, in order to facilitate the development of lemmas and proofs related to *SI-Monad*, we define the following relation *ConstraintsSol* that expresses the proposition that every constraint in *xs* ++ *ys* is satisfied by *sol*.

Definition 6.1.3 (ConstraintsSol)**.**

```
ConstraintsSol :
   List R1CS × List R1CS → List (Var × ℕ) → Prop
ConstraintsSol (xs , ys) sol
  = ∀ x → x ∈ (xs ++ ys) → R1CSSolution sol x
```

Next we define the low level representation of a literal in *Source* expressions. This representation relation is used in the main soundness lemma to relate the output of *sourceToR1CS* to the semantics function on *Source* (which will be defined later).

6.2 Literal Representation

Definition 6.2.1 (PiRepr, ValRepr). *ValRepr defines the representation of an element of* ⟦ *u* ⟧ *for a type code u while PiRepr defines the representation of a* Π *type element.*

```
data PiRepr (u : U) (x : ⟦ u ⟧ → U)
        (f : (v : ⟦ u ⟧) → ⟦ x v ⟧)
  : (eu : List ⟦ u ⟧) → Vec ℕ (tySumOver eu x) → Set

data ValRepr : ∀ u → ⟦ u ⟧ → Vec ℕ (tySize u) → Set where
  `OneValRepr : ∀ n → n ≈ 0 → ValRepr `One tt (n ∷ [])
  `TwoValFalseRepr :
      ∀ n → n ≈ 0 → ValRepr `Two false (n ∷ [])
  `TwoValTrueRepr :
      ∀ n → n ≈ 1 → ValRepr `Two true (n ∷ [])
  `BaseValRepr : ∀ {v : f} {v' : ℕ} → (fToℕ v) ≈ v'
      → ValRepr `Base v (v' ∷ [])
  `VecValBaseRepr : ∀ {u} → ValRepr (`Vec u 0) [] []
  `VecValConsRepr :
      ∀ {u} {n} {v₁} {vec₂} {val₁} {val₂} {val₃}
      → ValRepr u v₁ val₁
      → ValRepr (`Vec u n) vec₂ val₂
      → val₁ V++ val₂ ≡ val₃
      → ValRepr (`Vec u (suc n)) (v₁ ∷ vec₂) val₃
  `ΣValRepr :
      ∀ {u} {⟦u⟧} (x : ⟦ u ⟧ → U) {⟦xu⟧} {val⟦u⟧} {val⟦xu⟧}
```

```
        val⟦xu⟧+z {val⟦u⟧+val⟦xu⟧+z}
        {allZ : Vec ℕ (maxTySizeOver (enum u) x
                          - tySize (x ⟦u⟧))}
      → ValRepr u ⟦u⟧ val⟦u⟧
      → ValRepr (x ⟦u⟧) ⟦xu⟧ val⟦xu⟧
      → All (_≈_ 0) allZ
      → val⟦xu⟧+z ≅ val⟦xu⟧ V++ allZ
      → val⟦u⟧ V++ val⟦xu⟧+z ≡ val⟦u⟧+val⟦xu⟧+z
      → ValRepr (`Σ u x) (⟦u⟧ , ⟦xu⟧) val⟦u⟧+val⟦xu⟧+z
  `ΠValRepr :
      ∀ {u} (x : ⟦ u ⟧ → U) {f : (v : ⟦ u ⟧) → ⟦ x v ⟧ } val
      → PiRepr u x f (enum u) val → ValRepr (`Π u x) f val

data PiRepr u x f where
  PiRepNil : PiRepr u x f [] []
  PiRepCons : ∀ {el} {⟦u⟧} {val⟦xu⟧} {vec} {val⟦xu⟧+vec}
      → ValRepr (x ⟦u⟧) (f ⟦u⟧) val⟦xu⟧
      → PiRepr u x f el vec
      → val⟦xu⟧+vec ≡ val⟦xu⟧ V++ vec
      → PiRepr u x f (⟦u⟧ ∷ el) val⟦xu⟧+vec
```

where _≅_ is the usual heterogeneous equality.

The definition of *ValRepr* says that the representation of (up to _≅_)

- *tt* : ⟦ *‘One* ⟧ is (0 ∷ [])
- *false* : ⟦ *‘Two* ⟧ is (0 ∷ [])
- *true* : ⟦ *‘Two* ⟧ is (1 ∷ [])
- *v* : ⟦ *‘Two* ⟧ is (*fTo*ℕ *v* ∷ [])
- *vec* : ⟦ *‘Vec u n* ⟧ is the concatenation of the representations of the elements in *vec*

- *(a , b)* : ⟦ *‘Σ u x* ⟧ is the concatenation of the representations of a, b, and a zero vector that fills up the remaining space

- f : ⟦ ‘Π u x ⟧ is the concatenation of the representations of $f\ u_1$, $f\ u_2$, ... $f\ u_n$ where $[u_1, \ldots, u_n] = enum\ u$.

With *ValRepr* defined, next we define the semantics function for *Source* expressions.

6.3 Semantics Function for Source

Since *Source* expressions can contain R1CS variables, the semantics of a *Source* expression is intrinsically tied to a partial map *store* : *List* (*Var* × ℕ). When evaluating a *Source* expression *exp* together with *store*, we require *store* to map the R1CS variables in a way so that the *Ind*s in *exp* when solved with *store*, give us a representation of some element of the correct type. This is captured with the following definition.

Definition 6.3.1 (SourceStoreRepr). *SourceStoreRepr store u s says that all Inds in s are defined in store and that they point to elements of the correct type.*

```
data SourceStoreRepr (store : List (Var × ℕ))
      : ∀ u → Source u → Set where
  IndStore′ : ∀ {u} {m}
      (vec : Vec Var m) (val : Vec ℕ m) elem
      → (p : m ≡ tySize u)
      → BatchListLookup vec store val
      → ValRepr u elem (subst (Vec ℕ) p val)
      → SourceStoreRepr store u (Ind p vec)
  LitStore′ : ∀ {u} (v : ⟦ u ⟧)
      → SourceStoreRepr store u (Lit v)
  AddStore′ : ∀ (s₁ s₂ : Source `Base)
      → SourceStoreRepr store `Base s₁
      → SourceStoreRepr store `Base s₂
```

```
      → SourceStoreRepr store `Base (Add s₁ s₂)
  MulStore′ : ∀ (s₁ s₂ : Source `Base)
      → SourceStoreRepr store `Base s₁
      → SourceStoreRepr store `Base s₂
      → SourceStoreRepr store `Base (Mul s₁ s₂)
```

Now we are ready to define the semantics function for *Source*.

Definition 6.3.2 (sourceSem)**.** *Semantics function for Source. Under the condition that SourceStoreRepr store u s holds, the semantics function is defined as follows:*

```
sourceSem : ∀ u → (s : Source u) → (store : List (Var × ℕ))
      → SourceStoreRepr store u s → ⟦ u ⟧
sourceSem `One s st ss = tt
sourceSem `Two .(Ind refl vec) st
  (IndStore′ vec val elem refl x x₁) = elem
sourceSem `Two .(Lit v) st (LitStore′ v) = v
sourceSem `Base .(Ind p vec) st
  (IndStore′ vec val elem p x x₁) = elem
sourceSem `Base .(Lit v) st (LitStore′ v) = v
sourceSem `Base .(Add s₁ s₂) st (AddStore′ s₁ s₂ ss ss₁)
   = sourceSem `Base s₁ st ss +F sourceSem `Base s₂ st ss₁
sourceSem `Base .(Mul s₁ s₂) st (MulStore′ s₁ s₂ ss ss₁)
   = sourceSem `Base s₁ st ss *F sourceSem `Base s₂ st ss₁
sourceSem (`Vec u x) .(Ind p vec) st
  (IndStore′ vec val elem p x₁ x₂) = elem
sourceSem (`Vec u x) .(Lit v) st (LitStore′ v) = v
sourceSem (`Σ u x) .(Ind p vec) st
  (IndStore′ vec val elem p x₁ x₂) = elem
sourceSem (`Σ u x) .(Lit v) st (LitStore′ v) = v
sourceSem (`Π u x) .(Ind p vec) st
```

```
  (IndStore′ vec val elem p x₁ x₂) = elem
sourceSem (`Π u x) .(Lit v) st (LitStore′ v) = v
```

Before we define the soundness of *sourceToR1CS*, we first define a weakened notion of *SourceStoreRepr* that says that every R1CS variable in a given source expression points to some value in *store*.

Definition 6.3.3 (SourceStore). *SourceStore store u s says that all Inds in s point to something in store.*

```
data SourceStore (store : List (Var × ℕ))
     : ∀ (u : U) → Source u → Set where
  IndStore : ∀ {u} {m} (vec : Vec Var m) (val : Vec ℕ m)
     → (p : m ≡ tySize u)
     → BatchListLookup vec store val
     → SourceStore store u (Ind p vec)
  LitStore : ∀ {u} (v : ⟦ u ⟧)
     → SourceStore store u (Lit v)
  AddStore : ∀ (s₁ s₂ : Source `Base)
     → SourceStore store `Base s₁
     → SourceStore store `Base s₂
     → SourceStore store `Base (Add s₁ s₂)
  MulStore : ∀ (s₁ s₂ : Source `Base)
     → SourceStore store `Base s₁
     → SourceStore store `Base s₂
     → SourceStore store `Base (Mul s₁ s₂)
```

6.4 Compilation Soundness

Theorem 6.4.1 (sourceToR1CSSound). *Soundness of sourceToR1CS.*

```
sourceToR1CSSound :
  ∀ (r : WriterMode) (u : U)
```

```
→ (s : Source u)
→ (sol : List (Var × ℕ))
→ ListLookup 0 sol 1
→ SourceStore sol u s
→ ∀ init →
let result = sourceToR1CS u s ((r , prime) , init)
in ConstraintsSol (writerOutput result) sol
→ Squash (∃ (λ ⟦u⟧ → ∃ (λ val →
     ValRepr u ⟦u⟧ val × ∃ (λ ss →
       Σ′ (sourceSem u s sol ss ≡ ⟦u⟧)
          (λ _ → BatchListLookup
              (output result) sol val)))))
```

This lemma says that given a *Source* expression $s : \mathit{Source}\ u$ and a partial map $\mathit{sol} : \mathit{List}\ (\mathit{Var} \times \mathbb{N})$ such that

1. *sol* is a solution of the generated constraints from *sourceToR1CS*
2. The variable 0 maps to 1 in *sol*
3. *SourceStore sol u s* holds

then there is

```
⟦u⟧ : ⟦ u ⟧
val : Vec ℕ (tySize u)
ss : SourceStoreRepr store u s
```

such that the following diagram holds

$$\begin{tikzcd}
s \arrow[r, "sourceSem_{ss,sol}"] \arrow[d, "\text{primary output (sourceToR1CS)}"] & {[\![u]\!]} \arrow[d, dash, "ValRepr"] \\
out \arrow[r, dash, "BatchListLookup_{sol}"] & val
\end{tikzcd}$$

for any initial state *init*.

The proof of *sourceToR1CSSound* follows the definition of *sourceToR1CS*. Recall the definition of *sourceToR1CS* from Chapter 5:

```
sourceToR1CS : ∀ (u : U) → Source u
   → SI-Monad (Vec Var (tySize u))
sourceToR1CS u (Ind refl x)
   = withMode PostponedMode (indToIR u x)
sourceToR1CS u (Lit x) = litToInd u x
sourceToR1CS `Base (Add source source₁) = do
  r₁ ← sourceToR1CS `Base source
  r₂ ← sourceToR1CS `Base source₁
  v ← new
  add (IAdd zero ((one , head r₁) ::
                   (one , head r₂) :: (- one , v) :: []))
  return (v :: [])
sourceToR1CS `Base (Mul source source₁) = do
  r₁ ← sourceToR1CS `Base source
  r₂ ← sourceToR1CS `Base source₁
  v ← new
  add (IMul one (head r₁) (head r₂) one v)
```

```
  return (v ∷ [])
```

In the *Ind* case of *sourceToR1CS*, we want *indToIR* to generate the correct type constraints for the R1CS variables, and in the *Lit* case of *sourceToR1CS*, we want *litToInd* to return R1CS variables that solve to the representation of the given literal. In the *Add* and *Mul* cases, *sourceToR1CS* is proved by straightforward induction on the *Source* expression.

What does it mean to say that *indToIR* generates the correct type constraints? The type of *indToIR* is:

```
indToIR : ∀ (u : U)
    → Vec Var (tySize u)
    → SI-Monad (Vec Var (tySize u))
```

Given a type code u and a vector of variables vec, we want *indToIR* to generate enough constraints so that given a good enough solution $sol : List\ (Var \times \mathbb{N})$ to the constraints generated by *indToIR* u vec, vec solves to a representation of some $elem : [\![\ u\ ]\!]$ in sol. This is expressed as the following lemma:

Lemma 6.4.2 (indToIRSound)**.** *Soundness of indToIR.*

```
indToIRSound :
  ∀ (r : WriterMode) (u : U)
  → (vec : Vec Var (tySize u))
  → (val : Vec ℕ (tySize u))
  → (sol : List (Var × ℕ))
  → BatchListLookup vec sol val
  → ListLookup 0 sol 1
  → ∀ init →
  let result = indToIR u vec ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → Squash (∃ (λ elem → ValRepr u elem val))
```

For *litToInd*, we want the constraints generated by *litToInd u l* to ensure that the vector of variables returned by *litToInd u l* solves to a representation of the literal *l* for any good enough solution *sol* : *List* (*Var* × ℕ) to the generated constraints. This is expressed as the following lemma:

Lemma 6.4.3 (litToIndSound)**.** *Soundness of litToInd.*

```
litToIndSound :
  ∀ (r : WriterMode) (u : U)
  → (elem : ⟦ u ⟧)
  → (sol : List (Var × ℕ))
  → ListLookup 0 sol 1
  → ∀ init →
  let result = litToInd u elem ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → Squash (∃ (λ val → Σ′ (ValRepr u elem val)
      (λ _ → BatchListLookup (output result) sol val)))
```

In order to prove *indToIRSound* and *litToIndSound*, we proved that similar soundness properties hold for *tyCond*, *varEqLit*, and *assertTrue*, and to do so, we proved that similar soundness properties hold for the components (including *land*, *lor*, *limp* et cetera) used in the definition of *tyCond*, *varEqLit*, and *assertTrue*. Here we will give an example on what proving these soundness lemmas are like. Readers interested in the details on the lemmas used to prove *indToIRSound* and *litToIndSound* should check out Appendix B for a comprehensive view of the soundness lemmas and the auxiliary definitions used to define and prove these soundness lemmas.

Take the following definition of *limp* from Chapter 5 for example:

```
limp : Var → Var → SI-Monad Var
limp n₁ n₂ = do
  notN₁ ← lnot n₁
  lor notN₁ n₂
```

Given a solution *sol* to the constraints generated by *limp* n_1 n_2 such that n_1 maps to v_1 and n_2 maps to v_2 in *sol* and that $v_1, v_2 \in \{0, 1\}$, the variable that *limp* returns solves to 1 if $v_1 \rightarrow v_2$ (material implication) holds. This is captured with the following definition:

```
limpSound : ∀ (r : WriterMode)
  → (n₁ n₂ : Var) → (v₁ v₂ : ℕ)
  → (sol : List (Var × ℕ))
  → ListLookup n₁ sol v₁
  → ListLookup n₂ sol v₂
  → isBool v₁ → isBool v₂
  → ∀ (init : ℕ) →
  let result = limp n₁ n₂ ((r , prime) , init)
  in BuilderProdSol (writerOutput result) sol
  → ListLookup (output result) sol (limpFunc v₁ v₂)
```

where *limpFunc a b* evaluates to 0 if and only if $a \approx 1$ and $b \approx 0$ and *isBool a* holds if $a \approx 0$ or $a \approx 1$.

Since the inputs of *limp* solve to 0 or 1 in *sol*, the input condition for *lnot* is satisfied, and $notN_1$ must also solve to 0 or 1 in *sol*. Therefore, the input condition for *lor* (both input variables solve to 0 or 1 in *sol*) is satisfied, and the output variable solves to the result calculated by *limpFunc* (up to _≈_) in *sol*.

doi:10.6342/NTU202003013

Chapter 7

Conclusion

In this thesis, I constructed and formally proved the soundness of a dependently typed verifiable computation compiler. By using a tarski style universe in the domain specific language, users are able to write dependently typed domain specific programs that can be directly type checked by the dependently typed language Agda. The result from the Agda programs can then be subsequently piped into the zkSNARK library libsnark to generate the corresponding cryptographic proof. Overall, this was an attempt that I undertook to construct and formally verify a compiler that compiles to rank 1 constraints.

doi:10.6342/NTU202003013

doi:10.6342/NTU202003013

Appendix A

Full Definition of enum

Definition A.0.1 (enum, enumComplete, FuncInstLem)**.**

```
enum : (u : U) → List ⟦ u ⟧
enumComplete : ∀ (u : U) → (x : ⟦ u ⟧) → x ∈ enum u

enum `One = [ tt ]
enum `Two = false ∷ true ∷ []
enum `Base = Finite.elems finite
enum (`Vec u zero) = [ [] ]
enum (`Vec u (suc x)) = do
  r ← enum u
  rs ← enum (`Vec u x)
  return (r ∷ rs)
enum (`Σ u x) = do
  r ← enum u
  rs ← enum (x r)
  return (r , rs)
enum (`Π u x) =
   let pairs = do
                 r ← enum u
```

doi:10.6342/NTU202003013

```
              return (r , enum (x r))
       funcs = genFunc _ _ pairs
   in listFuncToPi u x (enum u) (enumComplete u) funcs
         (λ x₁ x₁∈genFunc →
             trans (genFuncProj₁ u x pairs x₁ x₁∈genFunc)
                (map-proj₁->>= (enum u) (enum ∘ x)))

FuncInstLem : ∀ u x (f : ⟦ `Π u x ⟧) (l : List ⟦ u ⟧)
    → FuncInst ⟦ u ⟧ (λ v → ⟦ x v ⟧) (piToList u x l f)
         (l >>= (λ r → (r , enum (x r)) ∷ []))
FuncInstLem u x f [] = InstNil
FuncInstLem u x f (x₁ ∷ l)
   = InstCons (piToList u x l f)
       (l >>= (λ r → (r , enum (x r)) ∷ []))
       x₁ (f x₁) (enum (x x₁)) (enumComplete (x x₁) (f x₁))
       (FuncInstLem u x f l)

enumComplete `One tt = here refl
enumComplete `Two false = here refl
enumComplete `Two true = there (here refl)
enumComplete `Base x = Finite.a∈elems finite x
enumComplete (`Vec u zero) [] = here refl
enumComplete (`Vec u (suc x₁)) (x ∷ x₂) =
  ∈l-∈l'-∈r (enum u) _∷_ x x₂ (enumComplete u x)
    (λ _ → enum (`Vec u x₁)) (enumComplete (`Vec u x₁) x₂)
enumComplete (`Σ u x₁) (fst , snd) =
  ∈l-∈l'-∈r (enum u) _,_ fst snd (enumComplete u fst)
    (λ r → enum (x₁ r)) (enumComplete (x₁ fst) snd)
enumComplete (`Π u x₁) f =
```

```
let pairs = do
      r ← enum u
      return (r , enum (x₁ r))
    genFuncs = genFunc u x₁ pairs
    fToList = piToList u x₁ (enum u) f
    fToListFuncInstPairs
        = FuncInstLem u x₁ f (enum u)
    fToList∈genFuncs
        = FuncInst→genFunc u x₁ pairs
             fToList fToListFuncInstPairs
    prf = trans
             (genFuncProj₁ u x₁ pairs
                  fToList fToList∈genFuncs)
             (map-proj₁->>= (enum u)
                  (λ x₂ → enum (x₁ x₂)))
    f≗piFromList∘piToList
        = piFromList∘piToList≗id u x₁
             (enum u) (enumComplete u) f prf
in  f∈listFuncToPi u x₁ _ _ genFuncs fToList _
         f fToList∈genFuncs
         (ext f≗piFromList∘piToList)
```

where *ext* is the principle of function extensionality, and *trans* is transitivity for propositional equality.

The reasoning for the Π case of *enumComplete* is that since *enum* (‘Π *u* *x*) is defined with *listFuncToPi*, we show that f : ⟦ ‘Π *u* *x* ⟧ when transformed into *fToList* with *piToList*, must be a member of *genFuncs*, and that *piFromList*∘*piToList* is the identity function.

doi:10.6342/NTU202003013

doi:10.6342/NTU202003013

Appendix B

Additional Formal Verification Lemmas and Definitions

Definition B.0.1 (Vec-≈).

```
data Vec-≈ : ∀ {n} → Vec ℕ n → Vec ℕ n → Prop where
  ≈-Nil : Vec-≈ [] []
  ≈-Cons : ∀ {n} x y {l : Vec ℕ n} {l'} → x ≈ y
       → Vec-≈ l l'
       → Vec-≈ (x ∷ l) (y ∷ l')
```

Definition B.0.2 (isBool).

```
data isBool : ℕ → Set where
  isZero : ∀ n → ℕtoF n ≡ zerof → isBool n
  isOne : ∀ n → ℕtoF n ≡ onef → isBool n
```

isBool n says that ℕ*toF n* is either 1 or 0. If *isBool n* holds for some $(n : \mathbb{N})$, then n is said to be boolean.

Definition B.0.3 (isBoolStrict).

```
data isBoolStrict : ℕ → Set where
  isZeroS : ∀ {n} → n ≡ 0 → isBoolStrict n
  isOneS : ∀ {n} → n ≡ 1 → isBoolStrict n
```

isBoolStrict n says that n is either 1 or 0. If *isBoolStrict n* holds for some (n : ℕ), then n is said to be strictly boolean.

The following lemma says that if a natural number n is strictly boolean, then n is boolean:

Lemma B.0.1 (isBoolStrict→isBool)**.**

```
isBoolStrict→isBool : ∀ {n} → isBoolStrict n → isBool n
```

Proof. Immediate from definition of *isBoolStrict*. □

Lemma B.0.2 (addSound)**.** *addSound says that if there is a solution sol to the constraints generated by add ir, then sol must be a solution to ir.*

```
addSound : ∀ (r : WriterMode)
   → (ir : R1CS)
   → (sol : List (Var × ℕ))
   → ∀ (init : ℕ) →
   let result = add ir ((r , prime) , init)
   in ConstraintsSol (writerOutput result) sol
   → R1CSSolution sol ir
```

Proof. Since *add ir* adds *ir* to the resulting contraints, any solution to the constraints generated by *add ir* must satisfy *ir*. □

Lemma B.0.3 (assertTrueSound)**.** *assertTrueSound says that if there is a solution sol to the constraints generated by assertTrue v, then v is mapped to 1 in sol.*

```
assertTrueSound : ∀ (r : WriterMode)
   → ∀ (v : Var) → (sol : List (Var × ℕ))
   → ∀ (init : ℕ) →
   let result = assertTrue v ((r , prime) , init)
   in
     ConstraintsSol (writerOutput result) sol
   → ListLookup v sol 1
```

Proof. By *addSound*. □

In order to give specifications to the logical functions, functions like the following *neqzFunc* are defined:

Definition B.0.4 (neqzFunc). *Specification of neqz.*

```
neqzFunc : ℕ → ℕ
neqzFunc n with ℕtoF n ≟F zerof
neqzFunc n | yes p = 0
neqzFunc n | no ¬p = 1
```

The following two lemmas are immediate from the definition of neqzFunc:

neqzFuncIsBoolStrict says that *neqzFunc n* is strictly boolean for all *n* : ℕ.

Lemma B.0.4 (neqzFuncIsBoolStrict)**.**

```
neqzFuncIsBoolStrict : ∀ n → isBoolStrict (neqzFunc n)
```

neqzFuncIsBool says that *neqzFunc n* is boolean for all *n* : ℕ.

Lemma B.0.5 (neqzFuncIsBool)**.**

```
neqzFuncIsBool : ∀ n → isBool (neqzFunc n)
```

Since *neqz* is constructed with *add*, we prove the soundness of *neqz* with respect to *neqzFunc* by first applying *addSound*, then we apply the reasoning in Section 5.4 on *neqz*:

Lemma B.0.6 (neqzSound)**.**

```
neqzSound : ∀ (r : WriterMode)
  → ∀ (v : Var) → (val : ℕ) → (sol : List (Var × ℕ))
  → ListLookup v sol val
  → ∀ (init : ℕ) →
  let result = neqz v ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → ListLookup (output result) sol (neqzFunc val)
```

Similarly, we proved that *neqzIsBool* solves to zero or one given a solution map that satisfies the constraints generated by *neqz v* by applying *addSound* and some simple field arithmetic:

Lemma B.0.7 (neqzIsBool)**.**

```
neqzIsBool : ∀ (r : WriterMode)
  → (v : Var)
  → (sol : List (Var × ℕ))
  → ∀ init →
  let result = neqz v ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → Squash (∃ (λ val → Σ′ (isBool val)
              (λ _ → ListLookup (output result) sol val)))
```

This is the leitmotif of the proofs of these lemmas: first we try to obtain the soundness proofs of the underlying components, then we apply a higher level reasoning that uses the subcomponents as basic building blocks.

The following lemma says that given a solution *sol* to the constraints generated by *neqz v* such that *neqz* outputs a variable that solves to 0 in *sol*, the input variable must solve to 0 in *sol*:

Lemma B.0.8 (neqzSound$_0$)**.**

```
neqzSound₀ : ∀ (r : WriterMode)
  → (v : Var)
  → (sol : List (Var × ℕ))
  → ListLookup 0 sol 1
  → ∀ init →
  let result = neqz v ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → ListLookup (output result) sol 0
```

```
  → Squash (∃ (λ val → (Σ′′ (ListLookup v sol val)
                               (λ _ → 0 ≈ val))))
```

By applying similar techniques, we can prove that other basic components used in the construction of the compiler are well-behaved when the generated constraints are satisfiable.

Definition B.0.5 (lorFunc)**.** *Specification of lor.*

```
lorFunc : ℕ → ℕ → ℕ
lorFunc a b with ℕtoF a ≟F zerof
lorFunc a b | yes p with ℕtoF b ≟F zerof
lorFunc a b | yes p | yes p₁ = 0
lorFunc a b | yes p | no ¬p = 1
lorFunc a b | no ¬p = 1
```

Lemma B.0.9 (lorFuncIsBoolStrict)**.**

```
lorFuncIsBoolStrict : ∀ a b → isBoolStrict (lorFunc a b)
```

Lemma B.0.10 (lorFuncIsBool)**.**

```
lorFuncIsBool : ∀ a b → isBool (lorFunc a b)
```

With the specification defined, we proved that *lor* is sound with respect to *lorFunc* when the input variables are mapped to boolean values in the solution.

Lemma B.0.11 (lorSound)**.**

```
lorSound : ∀ (r : WriterMode)
  → (v v' : Var) → (val val' : ℕ)
  → (sol : List (Var × ℕ))
  → ListLookup v sol val
  → ListLookup v' sol val'
  → isBool val → isBool val'
```

```
  → ∀ (init : ℕ) →
  let result = lor v v' ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → ListLookup (output result) sol (lorFunc val val')
```

Given a solution *sol* to the constraints generated by *lor v v'*, if the output variable of *lorFunc* solves to 0, and both inputs to *lor* solve to boolean values, then it must be the case that both input variables solve to 0:

Lemma B.0.12 (lorSound₀)**.**

```
lorSound₀ : ∀ (r : WriterMode)
  → (v v' : Var) (val val' : ℕ)
  → (sol : List (Var × ℕ))
  → ∀ init
  → ListLookup v sol val
  → ListLookup v' sol val'
  → isBool val
  → isBool val' →
  let result = lor v v' ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → ListLookup (output result) sol 0
  → Squash (Σ′′ (ListLookup v sol 0)
                (λ _ → ListLookup v' sol 0))
```

Similarly, for the logical and gate, we can define the following specification:

Definition B.0.6 (landFunc)**.** *Specification of land.*

```
landFunc : ℕ → ℕ → ℕ
landFunc a b with ℕtoF a ≟F zerof
landFunc a b | yes p = 0
landFunc a b | no ¬p with ℕtoF b ≟F zerof
```

```
landFunc a b | no ¬p | yes p = 0
landFunc a b | no ¬p | no ¬p₁ = 1
```

Suppose that *landFunc a b* outputs 1, and that *isBoolStrict a* holds, then *a* must be propositionally equal to 1.

Lemma B.0.13 (landFunc$^{-}_{1}$)**.**

```
landFunc⁻₁ : ∀ {a} {b}
  → isBoolStrict a → landFunc a b ≡ 1 → a ≡ 1
```

Suppose that *landFunc a b* outputs 1, and that *isBoolStrict b* holds, then *b* must be propositionally equal to 1.

Lemma B.0.14 (landFunc$^{-}_{2}$)**.**

```
landFunc⁻₂ : ∀ {a} {b}
  → isBoolStrict b → landFunc a b ≡ 1 → b ≡ 1
```

For arbitrary (*a b* : ℕ), *landFunc a b* is strictly boolean.

Lemma B.0.15 (landFuncIsBoolStrict)**.**

```
landFuncIsBoolStrict : ∀ a b → isBoolStrict (landFunc a b)
```

For arbitrary (*a b* : ℕ), *landFunc a b* is boolean.

Lemma B.0.16 (landFuncIsBool)**.**

```
landFuncIsBool : ∀ a b → isBool (landFunc a b)
```

We proved that given a solution *sol* to the constraints generated by *land v v'* such that the variable 0 maps to 1, and that both inputs to *land* solve to boolean values, then the output of *land* also solves to a boolean value.

Lemma B.0.17 (landIsBool)**.**

```
landIsBool : ∀ r v v' sol val val'
  → ListLookup v sol val
  → ListLookup v' sol val'
  → isBool val
  → isBool val'
  → ListLookup 0 sol 1
  → ∀ init →
  let result = land v v' ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → Squash (∃ (λ val'' → Σ′ (isBool val'')
       (λ _ → ListLookup (output result) sol val'')))
```

The following lemma says that *land* is sound with respect to its specification *landFunc* when the input variables map to boolean values in the solution:

Lemma B.0.18 (landSound)**.**

```
landSound : ∀ (r : WriterMode)
  → (v v' : Var) → (val val' : ℕ)
  → (sol : List (Var × ℕ))
  → ListLookup v sol val
  → ListLookup v' sol val'
  → isBool val → isBool val'
  → ∀ (init : ℕ) →
  let result = land v v' ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → ListLookup (output result) sol (landFunc val val')
```

landSound$_1$, proved with *landSound* and *addSound*, says that given a solution *sol* to the constraints generated by *land v v'* such that the output variable of *land* solves to 1, and that both input variables are boolean in *sol*, then it must be the case that both inputs solve to 1 in *sol*.

Lemma B.0.19 (landSound$_1$)**.**

```
landSound₁ : ∀ (r : WriterMode)
  → (v v' : Var) (val val' : ℕ)
  → (sol : List (Var × ℕ))
  → ∀ init
  → ListLookup v sol val
  → ListLookup v' sol val'
  → isBool val
  → isBool val' →
  let result = land v v' ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → ListLookup (output result) sol 1
  → Squash (Σ′′ (ListLookup v sol 1)
                (λ _ → ListLookup v' sol 1))
```

Similar to the case of *lor* and *land*, we define and prove similar lemmas for the other components used in the compiler:

Definition B.0.7 (lnotFunc)**.** *Specification of lnot.*

```
lnotFunc : ℕ → ℕ
lnotFunc a with ℕtoF a ≟F zerof
lnotFunc a | yes p = 1
lnotFunc a | no ¬p = 0
```

lnotFuncIsBoolStrict says that for any (n : ℕ), *lnotFunc n* is strictly boolean.

Lemma B.0.20 (lnotFuncIsBoolStrict)**.**

```
lnotFuncIsBoolStrict : ∀ n → isBoolStrict (lnotFunc n)
```

For any (*n* : ℕ), *lnotFunc n* is boolean:

Lemma B.0.21 (lnotFuncIsBool)**.**

```
lnotFuncIsBool : ∀ n → isBool (lnotFunc n)
```

lnotSound says that *lnot* is sound with respect to the specification *lnotFunc* when the input variable to *lnot* maps to a boolean in the solution.

Lemma B.0.22 (lnotSound)**.**

```
lnotSound : ∀ (r : WriterMode)
  → (v : Var) → (val : ℕ)
  → (sol : List (Var × ℕ))
  → ListLookup v sol val
  → isBool val
  → ∀ (init : ℕ) →
  let result = lnot v ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → ListLookup (output result) sol (lnotFunc val)
```

lnotSound$_1$ says that given a solution *sol* to the constraints generated by *lnot v*, if the output variable of *lnot* solves to 1 in *sol*, then the input variable solves to 0 in *sol*.

Lemma B.0.23 (lnotSound$_1$)**.**

```
lnotSound₁ : ∀ (r : WriterMode) v val sol init
  → ListLookup v sol val
  → isBool val →
  let result = lnot v ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → ListLookup (output result) sol 1
  → ListLookup v sol 0
```

Definition B.0.8 (limpFunc)**.** *Specification of limp.*

```
limpFunc : ℕ → ℕ → ℕ
limpFunc a b = lorFunc (lnotFunc a) b
```

Lemma B.0.24 (limpFuncImp).

```
limpFuncImp : ∀ {a} {b} → a ≡ 1
    → isBoolStrict b → limpFunc a b ≡ 1 → b ≡ 1
```

limpFuncIsBool says that for any (*a b* : ℕ), *limpFunc a b* must be boolean.

Lemma B.0.25 (limpFuncIsBool).

```
limpFuncIsBool : ∀ a b → isBool (limpFunc a b)
```

limpFuncIsBoolStrict says that for any (*a b* : ℕ), *limpFunc a b* must be strictly boolean.

Lemma B.0.26 (limpFuncIsBoolStrict).

```
limpFuncIsBoolStrict : ∀ a b → isBoolStrict (limpFunc a b)
```

Soundness of the logical implication gate given a solution *sol* to the constraints generated by *limp v v'* that maps input variables to boolean:

Lemma B.0.27 (limpSound).

```
limpSound : ∀ (r : WriterMode)
  → (v v' : Var) → (val val' : ℕ)
  → (sol : List (Var × ℕ))
  → ListLookup v sol val
  → ListLookup v' sol val'
  → isBool val → isBool val'
  → ∀ (init : ℕ) →
  let result = limp v v' ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → ListLookup (output result) sol (limpFunc val val')
```

anyNeqzIsBool says that given a solution *sol* to the constraints generated by *anyNeqz vec*, it must be the case that the output variable is boolean.

Lemma B.0.28 (anyNeqzIsBool)**.**

```
anyNeqzIsBool : ∀ r {n} (vec : Vec Var n) sol init
  → let result = anyNeqz vec ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → Squash (∃ (λ val → Σ′ (isBool val)
     (λ _ → ListLookup (output result) sol val)))
```

anyNeqzSound$_0$ says that given a solution *sol* to the constraints generated by *anyNeqz vec* such that 0 maps to 1 in *sol*, and that the output variable solves to 0, then it must be the case that the input variables all solve to 0.

Lemma B.0.29 (anyNeqzSound$_0$)**.**

```
anyNeqzSound₀ : ∀ (r : WriterMode)
  → ∀ {n} → (vec : Vec Var n)
  → (sol : List (Var × ℕ))
  → ListLookup 0 sol 1
  → ∀ init →
  let result = anyNeqz vec ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → ListLookup (output result) sol 0
  → Squash (∃ (λ val → (Σ′′ (BatchListLookup vec sol val)
      (λ _ → All (_≈_ 0) val))))
```

where *All* is the usual inductively defined predicate that says that all elements in the given vector satisfies the given predicate:

```
data All {A : Set} (P : A → Prop)
    : ∀ {n} → Vec A n → Prop where
  [] : All P []
  _∷_ : ∀ {n x} {xs : Vec A n} (px : P x)
           (pxs : All P xs) → All P (x ∷ xs)
```

Definition B.0.9 (varEqBaseLitFunc)**.** *Specification of varEqBaseLit*

```
varEqBaseLitFunc : ℕ → f → ℕ
varEqBaseLitFunc v l with ℕtoF v ≟F l
varEqBaseLitFunc v l | yes p = 1
varEqBaseLitFunc v l | no ¬p = 0
```

varEqBaseLitSound says that *varEqBaseLit* is sound with respect to the specification *varEqBaseLitFunc* if the input variable to *varEqBaseLit* is mapped to something in the solution.

Lemma B.0.30 (varEqBaseLitSound)**.**

```
varEqBaseLitSound : ∀ (r : WriterMode)
  → (v : Var) → (val : ℕ) → (l : f)
  → (sol : List (Var × ℕ))
  → ListLookup v sol val
  → ∀ (init : ℕ) →
  let result = varEqBaseLit v l ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → ListLookup (output result) sol (varEqBaseLitFunc val l)
```

varEqBaseLitSound$_1$ says that given a solution *sol* to the constraints generated by *varEqBaseLit v l* that maps 0 to 1, and that the output variable of *varEqBaseLit* solves to 1, then it must be the case that the input variable to *varEqBaseLit* maps to l:

Lemma B.0.31 (varEqBaseLitSound$_1$)**.**

```
varEqBaseLitSound₁ : ∀ (r : WriterMode)
  → (v : Var) (l : f)
  → (sol : List (Var × ℕ))
  → ListLookup 0 sol 1
  → ∀ init →
```

```
  let result = varEqBaseLit v l ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → ListLookup (output result) sol 1
  → Squash (∃ (λ val → Σ′ (ℕtoF val ≡ l)
       (λ _ → ListLookup v sol val)))
varEqBaseLitSound₁ r v l sol tri init isSol look
```

Proof. Recall the definition of *varEqBaseList*:

```
varEqBaseLit : Var → f → SI-Monad Var
varEqBaseLit n l = do
  n-l ← new
  add (IAdd (- l) ((one , n) ∷ (- one , n-l) ∷ []))
  ¬r ← neqz n-l
  r ← lnot ¬r
  return r
```

Given a solution *sol* to the constraints generated by *varEqBaseLit v l* such that 0 maps to 1 in *sol*, by *neqzIsBool*, *lnotSound*$_1$, and *neqzSound*$_0$, *n-l* solves to 0 in *sol*. By *addSound*, *n* solves to l in *sol*. ☐

varEqBaseLitIsBool says that given a solution *sol* to the constraints generated by *varEqBaseLit v l*, it must be the case that the output variable of *varEqBaseLit* maps to a boolean value.

Lemma B.0.32 (varEqBaseLitIsBool)**.**

```
varEqBaseLitIsBool : ∀ r (v : Var) (l : f)
   → ∀ sol init →
   let result = varEqBaseLit v l ((r , prime) , init)
   in ConstraintsSol (writerOutput result) sol
   → Squash (∃ (λ val → Σ′ (isBool val)
     (λ _ → ListLookup (output result) sol val)))
```

Proof. By *neqzIsBool*, *lnotFuncIsBool*, and *lnotSound*. □

Definition B.0.10 (anyNeqzFunc)**.** *Specification of anyNeqz.*

```
anyNeqzFunc : ∀ {n} → Vec ℕ n → ℕ
anyNeqzFunc [] = 0
anyNeqzFunc (x ∷ vec) with ℕtoF x ≟F zerof
anyNeqzFunc (x ∷ vec) | yes p = anyNeqzFunc vec
anyNeqzFunc (x ∷ vec) | no ¬p = 1
```

anyNeqzFuncIsBool says that *anyNeqzFunc vec* is boolean for any vector *vec*.

Lemma B.0.33 (anyNeqzFuncIsBool)**.**

```
anyNeqzFuncIsBool : ∀ {n} (vec : Vec ℕ n)
    → isBool (anyNeqzFunc vec)
```

anyNeqzFuncIsBoolStrict says that *anyNeqzFunc vec* is strictly boolean for any vector *vec*.

Lemma B.0.34 (anyNeqzFuncIsBoolStrict)**.**

```
anyNeqzFuncIsBoolStrict : ∀ {n} (vec : Vec ℕ n)
    → isBoolStrict (anyNeqzFunc vec)
```

anyNeqzSound says that given a solution *sol* to the constraints generated by *anyNeqz vec*, and that the input vector *vec* solves to *valVec* in *sol*, then it must be the case that the output variable solves to *anyNeqzFunc valVec* in *sol*.

Lemma B.0.35 (anyNeqzSound)**.**

```
anyNeqzSound : ∀ (r : WriterMode)
  → ∀ {n}
  → (vec : Vec Var n) → (valVec : Vec ℕ n)
  → (sol : List (Var × ℕ))
```

```
 → BatchListLookup vec sol valVec
 → ∀ (init : ℕ) →
 let result = anyNeqz vec ((r , prime) , init)
 in ConstraintsSol (writerOutput result) sol
 → ListLookup (output result) sol (anyNeqzFunc valVec)
```

Definition B.0.11 (allEqzFunc)**.**

```
allEqzFunc : ∀ {n} → Vec ℕ n → ℕ
allEqzFunc [] = 1
allEqzFunc (x ∷ vec) with ℕtoF x ≟F zerof
allEqzFunc (x ∷ vec) | yes p = allEqzFunc vec
allEqzFunc (x ∷ vec) | no ¬p = 0
```

allEqzFuncIsBool says that *allEqzFunc vec* is boolean for all *vec* : Vec ℕ n.

Lemma B.0.36 (allEqzFuncIsBool)**.**

```
allEqzFuncIsBool : ∀ {n} (vec : Vec ℕ n)
  → isBool (allEqzFunc vec)
```

allEqzFuncIsBoolStrict says that *allEqzFunc vec* is strictly boolean for all *vec* : Vec ℕ n.

Lemma B.0.37 (allEqzFuncIsBoolStrict)**.**

```
allEqzFuncIsBoolStrict : ∀ {n} (vec : Vec ℕ n)
  → isBoolStrict (allEqzFunc vec)
```

allEqzIsBool says that given a solution *sol* to the constraints generated by *allEqz vec* that maps 0 to 1, the output variable of *allEqz* solves to a booelan value.

Lemma B.0.38 (allEqzIsBool)**.**

```
allEqzIsBool : ∀ (r : WriterMode)
 → ∀ {n} → (vec : Vec Var n)
```

```
  → (sol : List (Var × ℕ))
  → ListLookup 0 sol 1
  → ∀ init →
  let result = allEqz vec ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → Squash (∃ (λ val → Σ′ (isBool val)
      (λ _ → ListLookup (output result) sol val)))
```

Proof. Recall the definition of *allEqz*:

```
allEqz : ∀ {n} → Vec Var n → SI-Monad Var
allEqz vec = do
  ¬r ← anyNeqz vec
  r ← lnot ¬r
  return r
```

Given a solution *sol* to the constraints generated by *allEqz vec* such that 0 is mapped to 1 in *sol*, by *anyNeqzIsBool* and *lnotSound*, r solves to the result specified by *lnotFunc* in *sol*. The desired result can then be obtained by applying *lnotFuncIsBool*. □

allEqzSound says that given a solution *sol* to the constraints generated by *allEqz vec* such that the input variables map to *valVec*, then it must be the case that the output variable solves to *allEqzFunc valVec*.

Lemma B.0.39 (allEqzSound)**.**

```
allEqzSound : ∀ (r : WriterMode)
  → ∀ {n}
  → (vec : Vec Var n) → (valVec : Vec ℕ n)
  → (sol : List (Var × ℕ))
  → BatchListLookup vec sol valVec
  → ∀ (init : ℕ) →
  let result = allEqz vec ((r , prime) , init)
```

```
  in ConstraintsSol (writerOutput result) sol
  → ListLookup (output result) sol (allEqzFunc valVec)
```

allEqzSound$_1$ says that given a solution *sol* to the constraints generated by *allEqz vec* such that 0 maps to 1 in *sol*, and that the output variable solves to 1, then it must be the case that the entries in the input vector all solve to 0.

Lemma B.0.40 (allEqzSound$_1$)**.**

```
allEqzSound₁ : ∀ (r : WriterMode)
  → ∀ {n} → (vec : Vec Var n)
  → (sol : List (Var × ℕ))
  → ListLookup 0 sol 1
  → ∀ init →
  let result = allEqz vec ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → ListLookup (output result) sol 1
  → Squash (∃ (λ val → (Σ′′ (BatchListLookup vec sol val)
      (λ _ → All (_≈_ 0) val))))
```

Proof. Recall the definition of *allEqz*:

```
allEqz : ∀ {n} → Vec Var n → SI-Monad Var
allEqz vec = do
  ¬r ← anyNeqz vec
  r ← lnot ¬r
  return r
```

Given a solution *sol* to the constraints generated by *allEqz vec* such that 0 maps to 1 in *sol*, by *anyNeqzIsBool* and *lnotSound*$_1$, we get that $\neg r$ solves to 0. Then by *anyNeqzSound*$_0$, *vec* solves to the 0 vector. ☐

Definition B.0.12 (piVarEqLitFunc, varEqLitFunc)**.** *Specification of piVarEqLit and varEqLit.*

```
piVarEqLitFunc : ∀ {u} (x : ⟦ u ⟧ → U) → (eu : List ⟦ u ⟧)
  → (vec : Vec ℕ (tySumOver eu x))
  → ⟦ `Π u x ⟧ → ℕ
varEqLitFunc : ∀ u → Vec ℕ (tySize u) → ⟦ u ⟧ → ℕ
varEqLitFunc `One (x ∷ vec) lit with ℕtoF x ≟F zerof
varEqLitFunc `One (x ∷ vec) lit | yes p = 1
varEqLitFunc `One (x ∷ vec) lit | no ¬p = 0
varEqLitFunc `Two (x ∷ vec) false with ℕtoF x ≟F zerof
varEqLitFunc `Two (x ∷ vec) false | yes p = 1
varEqLitFunc `Two (x ∷ vec) false | no ¬p = 0
varEqLitFunc `Two (x ∷ vec) true  with ℕtoF x ≟F onef
varEqLitFunc `Two (x ∷ vec) true | yes p = 1
varEqLitFunc `Two (x ∷ vec) true | no ¬p = 0
varEqLitFunc `Base (x ∷ vec) lit with ℕtoF x ≟F lit
varEqLitFunc `Base (x ∷ vec) lit | yes p = 1
varEqLitFunc `Base (x ∷ vec) lit | no ¬p = 0
varEqLitFunc (`Vec u zero) vec lit = 1
varEqLitFunc (`Vec u (suc x)) vec (l ∷ lit)
    with splitAt (tySize u) vec
... | fst , snd = landFunc (varEqLitFunc u fst l)
                           (varEqLitFunc (`Vec u x) snd lit)
varEqLitFunc (`Σ u x) vec (fst₁ , snd₁)
    with splitAt (tySize u) vec
... | fst , snd with maxTySplit u fst₁ x snd
... | vecₜ₁ , vecₜ₂
    = landFunc (landFunc
                 (varEqLitFunc u fst fst₁)
                 (varEqLitFunc (x fst₁) vecₜ₁ snd₁))
              (allEqzFunc vecₜ₂)
```

```
varEqLitFunc (`Π u x) vec lit
    = piVarEqLitFunc x (enum u) vec lit

piVarEqLitFunc x [] vec pi = 1
piVarEqLitFunc x (x₁ ∷ eu) vec pi
    with splitAt (tySize (x x₁)) vec
... | fst , snd
    = landFunc (varEqLitFunc (x x₁) fst (pi x₁))
               (piVarEqLitFunc x eu snd pi)
```

varEqLitFuncIsBoolStrict and *piVarEqLitFuncIsBoolStrict* say that *varEqLitFunc* and *piVarEqLitFunc* produce values that are strictly boolean.

Lemma B.0.41 (varEqLitFuncIsBoolStrict, piVarEqLitFuncIsBoolStrict)**.**

```
varEqLitFuncIsBoolStrict : ∀ u vec v
    → isBoolStrict (varEqLitFunc u vec v)
piVarEqLitFuncIsBoolStrict :
    ∀ {u} (x : ⟦ u ⟧ → U) eu vec pi
    → isBoolStrict (piVarEqLitFunc x eu vec pi)
```

varEqLitFuncIsBool and *piVarEqLitFuncIsBool* say that *varEqLitFunc* and *piVarEqLitFunc* produce values that are strictly boolean.

Lemma B.0.42 (varEqLitFuncIsBool, piVarEqLitFuncIsBool)**.**

```
varEqLitFuncIsBool : ∀ u vec v
    → isBool (varEqLitFunc u vec v)
piVarEqLitFuncIsBool : ∀ {u} (x : ⟦ u ⟧ → U) eu vec pi
    → isBool (piVarEqLitFunc x eu vec pi)
```

varEqLitSound says that given a solution *sol* to the constraints generated by *varEqLit u vec l* such that 0 maps to 1 in *sol* and that the input variable vector *vec* maps to *val*, then the output variable solves to the value specified by *varEqLitFunc*. *piVarEqLitSound* says that given a solution *sol* to the constraints generated by *piVarEqLit* that maps 0 to 1 and *vec* to *val*, the output variable solves to the value specified by *piVarEqLitFunc*.

Lemma B.0.43 (varEqLitSound, piVarEqLitSound)**.**

```
varEqLitSound : ∀ (r : WriterMode)
  → ∀ u → (vec : Vec Var (tySize u))
  → (val : Vec Var (tySize u))
  → (l : ⟦ u ⟧)
  → (sol : List (Var × ℕ))
  → BatchListLookup vec sol val
  → ListLookup 0 sol 1
  → ∀ (init : ℕ) →
  let result
         = varEqLit u vec l ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → ListLookup (output result)
       sol (varEqLitFunc u val l)

piVarEqLitSound : ∀ (r : WriterMode)
  → ∀ u (x : ⟦ u ⟧ → U) (eu : List ⟦ u ⟧)
  → (vec : Vec Var (tySumOver eu x))
  → (val : Vec ℕ (tySumOver eu x))
  → (pi : ⟦ `Π u x ⟧)
  → (sol : List (Var × ℕ))
  → BatchListLookup vec sol val
  → ListLookup 0 sol 1
  → ∀ (init : ℕ) →
  let result = piVarEqLit u x eu vec pi ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
```

```
  → ListLookup (output result)
       sol (piVarEqLitFunc x eu val pi)
```

piVarEqLitIsBool and *varEqLitIsBool* say that the respective output variables solve to boolean values given a solution *sol* satisfying the corresponding generated constraints that maps 0 to 1.

Lemma B.0.44 (piVarEqLitIsBool, varEqLitIsBool).

```
piVarEqLitIsBool : ∀ (r : WriterMode)
  → ∀ u x eu vec f sol
  → ListLookup 0 sol 1
  → ∀ init →
  let result = piVarEqLit u x eu vec f ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → Squash (∃ (λ val → Σ′ (isBool val)
      (λ _ → ListLookup (output result) sol val)))

varEqLitIsBool : ∀ (r : WriterMode)
  → ∀ u → (vec : Vec Var (tySize u))
  → (l : ⟦ u ⟧)
  → (sol : List (Var × ℕ))
  → ListLookup 0 sol 1
  → ∀ init →
  let result = varEqLit u vec l ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → Squash (∃ (λ val → Σ′ (isBool val)
      (λ _ → ListLookup (output result) sol val)))
```

Lemma B.0.45 (varEqLitSound$_1$, piVarEqLitSound$_1$).

```
varEqLitSound₁ : ∀ (r : WriterMode)
  → ∀ u → (vec : Vec Var (tySize u))
```

```
  → (l : ⟦ u ⟧)
  → (sol : List (Var × ℕ))
  → ListLookup 0 sol 1
  → ∀ init →
  let result = varEqLit u vec l ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → ListLookup (output result) sol 1
  → Squash (∃ (λ val → Σ′ (ValRepr u l val)
      (λ _ → BatchListLookup vec sol val)))
```

```
piVarEqLitSound₁ : ∀ (r : WriterMode)
  → ∀ u x eu vec f sol
  → ListLookup 0 sol 1
  → ∀ init →
  let result
        = piVarEqLit u x eu vec f ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → ListLookup (output result) sol 1
  → Squash (∃ (λ val → Σ′ (PiRepr u x f eu val)
       (λ _ → BatchListLookup vec sol val)))
```

Definition B.0.13 (tyCondFunc, enumSigmaCondFunc, enumPiCondFunc)**.** *Specification of tyCond, enumSigmaCond and enumPiCond.*

```
tyCondFunc : ∀ u → (vec : Vec ℕ (tySize u)) → ℕ
enumSigmaCondFunc : ∀ u → (eu : List ⟦ u ⟧)
  → (x : ⟦ u ⟧ → U)
  → (val₁ : Vec ℕ (tySize u))
  → (val₂ : Vec ℕ (maxTySizeOver (enum u) x))
  → ℕ
enumPiCondFunc : ∀ u → (eu : List ⟦ u ⟧) → (x : ⟦ u ⟧ → U)
```

```
  → Vec ℕ (tySumOver eu x) → ℕ

tyCondFunc `One (x ∷ vec) with ℕtoF x ≟F zerof
tyCondFunc `One (x ∷ vec) | yes p = 1
tyCondFunc `One (x ∷ vec) | no ¬p = 0
tyCondFunc `Two (x ∷ vec) with ℕtoF x ≟F zerof
tyCondFunc `Two (x ∷ vec) | yes p = 1
tyCondFunc `Two (x ∷ vec) | no ¬p with ℕtoF x ≟F onef
tyCondFunc `Two (x ∷ vec) | no ¬p | yes p = 1
tyCondFunc `Two (x ∷ vec) | no ¬p | no ¬p₁ = 0
tyCondFunc `Base vec = 1
tyCondFunc (`Vec u zero) vec = 1
tyCondFunc (`Vec u (suc x)) vec
    with splitAt (tySize u) vec
... | fst , snd
   = landFunc (tyCondFunc u fst)
              (tyCondFunc (`Vec u x) snd)
tyCondFunc (`Σ u x) vec with splitAt (tySize u) vec
tyCondFunc (`Σ u x) vec | fst₁ , snd₁
   = landFunc (tyCondFunc u fst₁)
         (enumSigmaCondFunc u (enum u) x fst₁ snd₁)
tyCondFunc (`Π u x) vec = enumPiCondFunc u (enum u) x vec

enumPiCondFunc u [] x vec = 1
enumPiCondFunc u (x₁ ∷ eu) x vec
    with splitAt (tySize (x x₁)) vec
... | fst₁ , snd₁
   = landFunc (tyCondFunc (x x₁) fst₁)
              (enumPiCondFunc u eu x snd₁)
```

enumPiCondFuncIsBool says that the value produced by *enumPiCondFunc* must be boolean.

Lemma B.0.46 (enumPiCondFuncIsBool)**.**

```
enumPiCondFuncIsBool : ∀ u eu x vec
  → isBool (enumPiCondFunc u eu x vec)
```

enumPiCondFuncIsBoolStrict says that the value produced by *enumPiCondFunc* must be strictly boolean.

Lemma B.0.47 (enumPiCondFuncIsBoolStrict)**.**

```
enumPiCondFuncIsBoolStrict : ∀ u eu x vec
  → isBoolStrict (enumPiCondFunc u eu x vec)
```

tyCondFuncIsBool says that the value produced by *tyCondFunc* must be boolean.

Lemma B.0.48 (tyCondFuncIsBool)**.**

```
tyCondFuncIsBool : ∀ u vec
  → isBool (tyCondFunc u vec)
```

tyCondFuncIsBoolStrict says that the value produced by *tyCondFunc* must be strictly boolean.

Lemma B.0.49 (tyCondFuncIsBoolStrict)**.**

```
tyCondFuncIsBoolStrict : ∀ u vec
  → isBoolStrict (tyCondFunc u vec)
```

enumSigmaCondFuncIsBool says that the value produced by *enumCondFunc* must be boolean.

Lemma B.0.50 (enumSigmaCondFuncIsBool)**.**

```
enumSigmaCondFuncIsBool : ∀ u eu x val₁ val₂
  → isBool (enumSigmaCondFunc u eu x val₁ val₂)
```

doi:10.6342/NTU202003013

enumSigmaCondFuncIsBoolStrict says that the value produced by *enumCondFunc* must be strictly boolean.

Lemma B.0.51 (enumSigmaCondFuncIsBoolStrict)**.**

```
enumSigmaCondFuncIsBoolStrict : ∀ u eu x val₁ val₂
  → isBoolStrict (enumSigmaCondFunc u eu x val₁ val₂)
```

enumPiCondSound (*tyCondSound*) say that given a solution *sol* to the constraints generated by *enumPiCond eu x vec (tyCond u vec)* such that 0 maps to 1 in *sol* and that the input vector *vec* maps to *val*, then the output variable solves to the value specified by *enumPiCondFunc* (*tyCondFunc*). *enumSigmaCondSound* says that given a solution *sol* of the constraints generated by *enumSigmaCond* such that the input vectors vec_1 and vec_2 map to val_1 and val_2 and that 0 maps to 1, the output variable solves to the value specified by *enumSigmaCondFunc*.

Lemma B.0.52 (enumPiCondSound, tyCondSound, enumSigmaCondSound)**.**

```
enumPiCondSound : ∀ r u → (eu : List ⟦ u ⟧)
   → (x : ⟦ u ⟧ → U)
   → (vec : Vec Var (tySumOver eu x))
   → (val : Vec ℕ (tySumOver eu x))
   → (sol : List (Var × ℕ))
   → BatchListLookup vec sol val
   → ListLookup 0 sol 1
   → ∀ init →
   let result = enumPiCond eu x vec ((r , prime) , init)
   in ConstraintsSol (writerOutput result) sol
   → ListLookup (output result) sol
         (enumPiCondFunc u eu x val)


tyCondSound : ∀ r u
```

```
    → (vec : Vec Var (tySize u))
    → (val : Vec ℕ (tySize u))
    → (sol : List (Var × ℕ))
    → BatchListLookup vec sol val
    → ListLookup 0 sol 1
    → ∀ init →
    let result = tyCond u vec ((r , prime) , init)
    in ConstraintsSol (writerOutput result) sol
    → ListLookup (output result) sol (tyCondFunc u val)

enumSigmaCondSound : ∀ r u → (eu : List ⟦ u ⟧)
    → (x : ⟦ u ⟧ → U)
    → (vec₁ : Vec Var (tySize u))
    → (vec₂ : Vec Var (maxTySizeOver (enum u) x))
    → (val₁ : Vec ℕ (tySize u))
    → (val₂ : Vec ℕ (maxTySizeOver (enum u) x))
    → (sol : List (Var × ℕ))
    → BatchListLookup vec₁ sol val₁
    → BatchListLookup vec₂ sol val₂
    → ListLookup 0 sol 1
    → ∀ init →
    let result
        = enumSigmaCond eu x vec₁ vec₂ ((r , prime) , init)
    in ConstraintsSol (writerOutput result) sol
    → ListLookup (output result) sol
     (enumSigmaCondFunc u eu x val₁ val₂)
```

ValRepr→varEqLit says that *ValRepr* implies *varEqLitFunc*, and *PiRepr→piVarEqLit* says that *PiRepr* implies *piVarEqLit*.

Lemma B.0.53 (ValRepr→varEqLit, PiRepr→piVarEqLit)**.**

```
ValRepr→varEqLit : ∀ u elem val val' → val ≡ val'
   → ValRepr u elem val'
   → Squash (varEqLitFunc u val elem ≡ 1)
PiRepr→piVarEqLit : ∀ u x eu vec vec' f → vec ≡ vec'
   → PiRepr u x f eu vec'
   → Squash (piVarEqLitFunc x eu vec f ≡ 1)
```

enumSigmaCondRestZ says that if *val* : ⟦ *u* ⟧ falls inside of *eu*, that *fst* is the representation of *val*, and that *enumSigmaCondFunc u eu x fst snd* is propositionally equal to 1, then *proj*$_2$ (*maxTySplit u val x snd*) (which corresponds to the third part of a sigma type representation) must be a 0 vector.

Lemma B.0.54 (enumSigmaCondRestZ)**.**

```
enumSigmaCondRestZ : ∀ u eu x fst snd val
  → val ∈ eu → ValRepr u val fst
  → enumSigmaCondFunc u eu x fst snd ≡ 1
  → All (_≈_ 0) (proj₂ (maxTySplit u val x snd))
```

tyCondFuncRepr says that if the specification of *tyCond* holds for some *u* and *vec*, then *vec* must be a representation of some inhabitant of type ⟦ *u* ⟧. *enumSigmaCondFuncRepr* and *piTyCondFuncPartialRepr* are the corresponding representation lemmas for Σ and Π types. These lemmas are proved with mutual recursion in Agda:

Lemma B.0.55 (tyCondFuncRepr, enumSigmaCondFuncRepr. piTyCondFuncPartialRepr)**.**

```
tyCondFuncRepr : ∀ u → (vec : Vec ℕ (tySize u))
  → tyCondFunc u vec ≡ 1
  → Squash (∃ (λ elem → ValRepr u elem vec))

enumSigmaCondFuncRepr : ∀ u eu x elem val₁ val₂
  → ValRepr u elem val₁
```

```
  → elem ∈ eu
  → enumSigmaCondFunc u eu x val₁ val₂ ≡ 1
  → Squash (∃ (λ elem₁ → ValRepr (x elem) elem₁
                 (proj₁ (maxTySplit u elem x val₂))))
piTyCondFuncPartialRepr : ∀ u (x : ⟦ u ⟧ → U) eu
  (prf : ∀ v → v ∈ eu → occ _≟U_ v eu ≡ 1)
  → (vec : Vec ℕ (tySumOver eu x))
  → enumPiCondFunc u eu x vec ≡ 1
  → Squash (∃ (λ f → PiRepr u x f eu vec))
```

indToIRSound says that given a solution *sol* to the constraints generated by *indToIR u vec* such that 0 maps to 1 in *sol* and that *vec* maps to *val* in *sol*, then there must be a high level representation *elem* such that *ValRepr u elem val*.

Lemma B.0.56 (indToIRSound)**.** *Soundness of indToIR.*

```
indToIRSound : ∀ r u
  → (vec : Vec Var (tySize u))
  → (val : Vec ℕ (tySize u))
  → (sol : List (Var × ℕ))
  → BatchListLookup vec sol val
  → ListLookup 0 sol 1
  → ∀ init →
  let result = indToIR u vec ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → Squash (∃ (λ elem → ValRepr u elem val))
```

varEqLitFuncRepr says that *varEqLitFunc* implies *ValRepr*. *piVarEqLitFuncRepr* says that *piVarEqLitFunc* implies *PiRepr*.

Lemma B.0.57 (varEqLitFuncRepr, piVarEqLitFuncRepr)**.**

```
varEqLitFuncRepr : ∀ u val elem
   → varEqLitFunc u val elem ≡ 1
```

```
   → Squash (ValRepr u elem val)
piVarEqLitFuncRepr : ∀ u (x : ⟦ u ⟧ → U) eu vec f
   → piVarEqLitFunc x eu vec f ≡ 1
   → Squash (PiRepr u x f eu vec)
```

litToIndSound says that given a solution *sol* to the constraints generated by *litToInd u elem* such that 0 maps to 1 in *sol*, then there is a low level representation *val* such that *ValRepr u elem val* and that the output variables map to *val* in *sol*.

Lemma B.0.58 (litToIndSound)**.** *Soundness of litToInd.*

```
litToIndSound : ∀ r u
  → (elem : ⟦ u ⟧)
  → (sol : List (Var × ℕ))
  → ListLookup 0 sol 1
  → ∀ init →
  let result = litToInd u elem ((r , prime) , init)
  in ConstraintsSol (writerOutput result) sol
  → Squash (∃ (λ val → Σ′ (ValRepr u elem val)
      (λ _ → BatchListLookup (output result) sol val)))
```

assertVarEqVarSound says that if there is a solution *sol* to the constraints generated by *assertVarEqVar n v v'* such that 0 maps to 1 in *sol*, *v* maps to *val*, and *v'* maps to *val'*, then *Vec-≈ val val'* holds.

Lemma B.0.59 (assertVarEqVarSound)**.** *Soundness of assertVarEqVar.*

```
assertVarEqVarSound : ∀ r n
  → (v v' : Vec Var n)
  → (sol : List (Var × ℕ))
  → (val val' : Vec ℕ n)
  → BatchListLookup v sol val
  → BatchListLookup v' sol val'
```

```
→ ListLookup 0 sol 1
→ ∀ init →
let result = assertVarEqVar n v v' ((r , prime) , init)
in ConstraintsSol (writerOutput result) sol
→ Vec-≈ val val'
```

doi:10.6342/NTU202003013

doi:10.6342/NTU202003013

Bibliography

[1] D. Ahman, C. Fournet, C. Hriţcu, K. Maillard, A. Rastogi, and N. Swamy. Recalling a witness: Foundations and applications of monotonic state. *Proc. ACM Program. Lang.*, 2(POPL), Dec. 2017.

[2] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 90–108, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[3] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC’ 14, page 781−796, USA, 2014. USENIX Association.

[4] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

[5] J. Cockx, D. Devriese, and F. Piessens. Pattern matching without K. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP ’14, pages 257–268, New York, NY, USA, 2014. Association for Computing Machinery.

[6] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *The Journal of Symbolic Logic*, 65(2):525–549, 2000.

[7] C. Fournet, C. Keller, and V. Laporte. A certified compiler for verifiable computing.

In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 268–280, June 2016.

[8] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct nizks without pcps. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 626–645, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[9] H. Goguen, C. McBride, and J. McKinna. *Eliminating Dependent Pattern Matching*, pages 521–540. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[10] R. J. M. Hughes. A novel representation of lists and its application to the function “reverse” . *Inf. Process. Lett.*, 22(3):141-144, Mar. 1986.

[11] P. Martin-Löf. An intuitionistic theory of types. Technical report, University of Stockholm, 1972.

[12] C. McBride. Elimination with a motive. In P. Callaghan, Z. Luo, J. McKinna, R. Pollack, and R. Pollack, editors, *Types for Proofs and Programs*, pages 197–216, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[13] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.

[14] G. Stewart, S. Merten, and L. Leland. Snårkl: Somewhat practical, pretty much declarative verifiable computing in haskell. In F. Calimeri, K. Hamlen, and N. Leone, editors, *Practical Aspects of Declarative Languages*, pages 36–52, Cham, 2018. Springer International Publishing.

[15] The Coq Development Team. The Coq proof assistant, Jan. 2020.

[16] Zokrates Development Team. Zokrates. `https://github.com/Zokrates/ZoKrates`.