Arithmetization of Σ_1^1 relations with polynomial bounds in Halo 2

Anthony Hart and Morgan Thomas

Orbis Labs
team@orbislabs.com

August 26, 2022

Abstract

Previously [4], Orbis Labs presented a method for compiling ("arithmetizing") relations, expressed as Σ_1^1 formulas in the language of rings, into Halo 2 [1, 2, 3] arithmetic circuits. In this research, we extend this method to support polynomial quantifier bounds, in addition to constant quantifier bounds. This allows for more efficient usage of rows in the resulting circuit.

Contents

1 Introduction **1**

2 Σ_1^1 formulas with polynomial quantifier bounds **3**

3 Semicircuits **10**

4 Translating Σ_1^1 Formulas into Semicircuits **14**

5 Circuits and logic circuits **16**

6 Compiling semicircuits to logic circuits **19**

7 Compiling logic circuits to arithmetic circuits **24**

1 Introduction

A proof is traditionally defined as a sound argument. A sound argument is a valid argument with true premises. An argument is a series (or sometimes a tree or another structure) of statements which relates a set of zero or more premises to a conclusion. A valid argument is an argument which is truth-preserving, in the sense that in any "model" or possible reality where the premises of the argument are all true, its conclusion is also true. As a corollary of these definitions, the conclusion of a sound argument is true.

Proof theories state rules for constructing valid arguments. A proof theory is a game with symbols which describes processes of truth-preserving inference.

Model theories state rules for interpreting the meaning of a statement. Using model theory, the meaning of a statement can be understood as its truth conditions, or the set of "models" or possible realities where

the statement is true. A model determines a unique truth value (e.g., "true" or "false") for each statement in the language of the model.

The concept of soundness can be defined formally using model theory. The general idea is as follows. A sound argument is an argument A such that for all models M, if all of the premises of A are true in M, then the conclusion of A is true in M.

Probabilistic proof systems differ from traditional proof systems in that they use a different definition of soundness. In the context of probabilistic proof systems, soundness is the property that every computationally feasible algorithm has a negligible probability of finding a proof of a false statement.

Why would one use a probabilistic proof system, which does not confer absolute certainty, when traditional proof systems provide certainty? Probabilistic proof systems confer benefits that traditional proof systems do not: in particular, succinct proofs, succinct verification, and information hiding.

A succinct proving system produces proofs which are in many cases much shorter than the shortest proof of the same statement would be using a traditional proof theory. Furthermore, they allow for the proof size to be decoupled from the amount of information required to construct the proof in the first place. The ideal of succinct proofs is that proofs are O(1) in size, meaning that a constant number of bits suffice to represent proofs of all provable statements.

We can quantify succinctness by considering maximum proof length as a function of statement length. Given a proof system S, consider the function $f_S : \mathbb{N} \to \mathbb{N}$ defined as follows. $f_S(n)$ is the largest number y such that for some statement ϕ of n symbols or less, y is the minimum length of a proof of ϕ in system S. For an ideally succinct proof system S, f_S is $O(1)$. If f_S is $O(n)$, that also qualifies as a succinct proof system.

Gödel's speed-up theorem demonstrates that traditional proof theories capable of representing the statements and proving the theorems of Peano arithmetic are unable to be succinct. Quite the opposite is the case.

The proof of Gödel's speed-up theorem relies on constructing a self-referential statement, such as "this statement cannot be proved in system S in less than a googolplex symbols." If such a statement could be proved in system S in less than a googolplex symbols, then it would be provable in system S but not true, and thus S would be unsound, and in fact S would be logically inconsistent. Therefore, if S is consistent, then the statement is true, and it can be proven in S by enumerating all proofs of less than a googolplex symbols and checking that none of them are a proof of the statement. Therefore, S is not succinct; there is a fairly short statement which it can prove, such that the shortest proof in system S of that statement is no less than a googolplex symbols.

The speed-up theorem shows that f_S is a mind bogglingly fast-growing function when S is a proof system that is absolutely sound and capable of representing the statements and proving the theorems of Peano arithmetic. In such cases f_S is not $O(1)$, nor $O(n)$, nor $O(g(n))$ for any computable function g; it grows faster than all of these. To see this, consider $\phi(n)$ as the self-referential statement "this statement is not provable in system S in less than n symbols." How large can the shortest proof of $\phi(n)$ be, as a function of the length of $\phi(n)$? This just depends on the largest number we can refer to in a given number of symbols.

We could allow for a particular self-referential statement $\phi(n)$ to be proven succinctly by defining a system S' which consists of S plus the axiom $\phi(n)$. However, Gödel's speed-up theorem also applies to system S', meaning there is another statement which can be proven in system S' but not in less than a googolplex symbols.

In contrast to traditional proof systems which are absolutely sound, probabilistic proof systems are able to be succinct. By relaxing the requirement for absolute soundness to the requirement for probabilistic soundness, we obtain the capability of succinct proofs.

Probabilistic proof systems are also able to have succinct verification, meaning that proofs can be verified efficiently and in some cases with O(1) time and space complexity. This is not possible with

traditional proof theories, because by nature, proof checking needs to look at each symbol in the proof.

Finally, probabilistic proof systems are able to have information hiding, meaning that they do not reveal certain information required to construct the proof. This is most often expressed as "zero knowledge," which is roughly the property that having a proof p of a statement x does not reduce the computational difficulty of finding a secret s which was used as an input to the process which resulted in p.

For all of these reasons, probabilistic proof systems are useful in various applications. For example, the Zcash cryptocurrency protocol uses succinct, zero-knowledge probabilistic proofs to verify private transactions. The latest and greatest version of Zcash's probabilistic proving is based on Halo 2. [1, 2, 3]

Orbis Labs' previous research on Σ_1^1 arithmetization [4] demonstrated the concept of translating a bounded-quantifier formula of second-order arithmetic to a Halo 2 circuit denoting an equivalent relation.

Orbis Labs' ongoing research and development on the Orbis Specification Language (OSL) builds on Σ_1^1 arithmetization by providing a high level specification language which can be compiled into second order arithmetic formulas. [6]

The previous research on Σ_1^1 arithmetization presented two practical difficulties: it is defined in a way that is complex and difficult to work with, and it is inefficient in its usage of rows. To see how its row usage is inefficient, consider quantifying over the elements of a list. All quantifiers are bounded by constants, and the constant in this case will be the maximum length of the list. Arithmetizing such a formula will require at least as many rows as the maximum length of the list. Now suppose that the list is a list of lists. Each list in the list of lists may be of a different length. Now suppose that in addition to quantifying over elements of the list, the formula quantifies over elements of elements of the list. That inner quantifier will be bound by a constant. If the bound on the length of the list is b_0, and the bound on the length of any element of the list is b_1, then such a formula will require at least $b_0 \cdot b_1$ rows to arithmetize. However, we could achieve more efficient row usage if instead the number of rows was a bound on the sum of the lengths of the elements of the list.

This paper provides a new definition of Σ_1^1 arithmetization. The goals of this research are twofold:

1. Allow for efficient usage of rows when nested quantifiers are present, by allowing for quantifier bounds to be variable instead of always constant.

2. Provide a clearer and simpler definition of Σ_1^1 arithmetization which is more amenable to formalization.

No effort is made in this paper to prove that this definition of Σ_1^1 arithmetization is correct. Formalization [5] work on Σ_1^1 arithmetization is ongoing. That formalization work will define and prove the correctness of Σ_1^1 arithmetization.

2 Σ_1^1 formulas with polynomial quantifier bounds

This section gives a new definition of Σ_1^1 formulas over the language of rings with polynomial quantifier bounds (henceforth just "Σ_1^1 formulas"), as well as a denotational semantics for these formulas. Apart from the polynomial bounds, the definitions in this section are otherwise similar to the definitions given in [4], Section 3.

The first step in defining Σ_1^1 formulas is to define a language of terms. A term is a syntactic object which denotes a number, given a context which bestows values to the variables it contains. Terms are defined by the following recursive definition.

1. For each positive integer i, x_i is a term. x_i is called a first-order variable.

2. For each positive integer i and all sequences of terms $\tau_1, ..., \tau_n$, $f_i^n(\tau_1, ..., \tau_n)$ is a term. $f_i^n(\tau_1, ..., \tau_n)$ is called a function application. Notice that in this definition, second order variables contain their arity as part of their name.

3. For all terms τ, μ:

   (a) $(\tau + \mu)$ is a term.

   (b) $(\tau \cdot \mu)$ is a term.

   (c) $\text{ind}_<(\tau, \mu)$ is a term. $\text{ind}_<$ is called the comparison indicator function; it returns 1 when τ is less than μ and 0 otherwise.

4. 0 is a term. 1 is a term. -1 is a term. These are called constant symbols.

A "polynomial term," by definition, is a term which is constructed using only rules 1, 2, 3a, 3b, and 4 above: in other words, it is built out of variables, function applications, and constants using only addition and multiplication.

First-order formulas (over the language of rings) are defined by the following recursive definition.

1. For all terms τ, μ,
$$(\tau = \mu) \tag{1}$$
is a first-order formula. $\tau = \mu$ is called an atomic formula or an equation.

2. For all first-order formulas ϕ,
$$\neg\phi \tag{2}$$
is a first-order formula. $\neg\phi$ is called a negation.

3. For all first-order formulas ϕ, ψ,
$$(\phi \wedge \psi) \tag{3}$$
is a first-order formula. $(\phi \wedge \psi)$ is called a conjunction.

4. For all first-order formulas ϕ, ψ,
$$(\phi \vee \psi) \tag{4}$$
is a first-order formula. $(\phi \vee \psi)$ is called a disjunction.

5. For all first-order formulas ϕ, ψ,
$$(\phi \to \psi) \tag{5}$$
is a first-order formula. $(\phi \to \psi)$ is called an implication.

6. For all first-order formulas ϕ and polynomial terms β,
$$\forall < \beta.\ \phi \tag{6}$$
is a first-order formula. $\forall$ is called the universal quantifier. β is called the quantifier bound.

7. For all first-order formulas ϕ and polynomial terms β,
$$\exists < \beta.\ \phi \tag{7}$$
is a first-order formula. $\exists$ is called the first-order existential quantifier. β is called the quantifier bound.

Σ_1^1 formulas (over the language of rings) are defined by the following recursive definition.

1. Every first-order formula is a Σ_1^1 formula.
2. For all Σ_1^1 formulas ϕ and polynomial terms γ and non-empty sequences of polynomial terms $\beta_1, ..., \beta_n$,

$$\exists_f < \gamma(< \beta_1, ..., < \beta_n).\ \phi \tag{8}$$

   is a Σ_1^1 formula. $\exists_f$ is called the second-order existential quantifier. n is called the arity of the function. $\beta_1, ..., \beta_n$ are called the bounds of the dimensions of the domain of the function. γ is called the bound of the codomain of the function.

Without loss of generality, we will assume that all formulas under consideration reference each quantified variable that they contain.

It is often useful to know the set of variables which are free in a given formula. How is this set defined when variables are denoted by de Bruijn indices, which can denote different variables in different contexts? Even two free occurrences of the same de Bruijn index in the same formula can denote different variables if they are in different contexts with different levels of quantifier nesting. The trick here is to define the set of free variables as the de Bruijn indices, relative to the outermost scope of the formula (outside of all the quantifiers), of all the free variables referenced in the program, regardless of which de Bruijn indices denote those variables in the context(s) in which those variables are referenced. The following recursive equations define the set of free variables in a Σ_1^1 formula or a term. $\mathcal{P}$ denotes the power set function: $\mathcal{P}(X) := \{Y : \text{Set} \mid Y \subseteq X\}$.

$$\text{free} : \{\phi \mid \phi \text{ is a } \Sigma_1^1 \text{ formula}\} \oplus \{\tau \mid \tau \text{ is a term}\} \to \mathcal{P}(\{x_i\}_{i \in \mathbb{Z}^+} \oplus \{f_i\}_{i \in \mathbb{Z}^+}) \tag{9}$$

$$\text{free}(x_i) := \{x_i\} \tag{10}$$

$$\text{free}(f_i^n(\tau_1, ..., \tau_n)) := \{f_i^n\} \cup \bigcup_{i \in [n]} \text{free}(\tau_i) \tag{11}$$

$$\text{free}(\tau + \mu) := \text{free}(\tau) \cup \text{free}(\mu) \tag{12}$$

$$\text{free}(\tau \cdot \mu) := \text{free}(\tau) \cup \text{free}(\mu) \tag{13}$$

$$\text{free}(\text{ind}_(\tau, \mu)) := \text{free}(\tau) \cup \text{free}(\mu) \tag{14}$$

$$\text{free}(0) = \text{free}(1) := \text{free}(-1) = \emptyset \tag{15}$$

$$\text{free}(\tau = \mu) := \text{free}(\tau) \cup \text{free}(\mu) \tag{16}$$

$$\text{free}(\neg\phi) := \text{free}(\phi) \tag{17}$$

$$\text{free}(\phi \wedge \psi) := \text{free}(\phi) \cup \text{free}(\psi) \tag{18}$$

$$\text{free}(\phi \vee \psi) := \text{free}(\phi) \cup \text{free}(\psi) \tag{19}$$

$$\text{free}(\phi \to \psi) := \text{free}(\phi) \cup \text{free}(\psi) \tag{20}$$

$$\text{free}(\forall < \beta.\ \phi) := \text{free}(\beta) \cup \{x_i \mid x_{i+1} \in \text{free}(\phi)\} \cup \{f_i^n \mid f_i^n \in \text{free}(\phi)\} \tag{21}$$

$$\text{free}(\exists < \beta.\ \phi) := \text{free}(\beta) \cup \{x_i \mid x_{i+1} \in \text{free}(\phi)\} \cup \{f_i^n \mid f_i^n \in \text{free}(\phi)\} \tag{22}$$

$$\text{free}(\exists_f < \gamma(< \vec{\beta}).\ \phi) := \text{free}(\gamma) \cup \left(\bigcup_i \text{free}(\beta_i)\right) \cup \{x_i \mid x_i \in \text{free}(\phi)\} \cup \{f_i^n \mid f_{i+1}^n \in \text{free}(\phi)\} \tag{23}$$

Relative to a suitable model, it is possible to define whether any given Σ_1^1 formula is true or false. For this context, a model, by definition, is a tuple

$$M = (R, \cdot, +, 0, 1, -1, <, F, S), \tag{24}$$

where:

1. $(R, \cdot, +, 0, 1, -1)$ is a ring:
   (a) R is a set.
   (b) $\cdot : R \times R \to R$ is a binary operation.
   (c) $+ : R \times R \to R$ is a binary operation.
   (d) $0, 1, -1 \in R$.
   (e) $+$ is associative and commutative, meaning, for all $x, y, z \in R$,

$$(x + y) + z = x + (y + z), \tag{25}$$

$$x + y = y + x. \tag{26}$$

   (f) Each element of R has an additive inverse, meaning, for each $x \in R$ there is $y \in R$ such that $x + y = 0$.
   (g) $\cdot$ is associative, meaning, for all $x, y, z \in R$,

$$(x \cdot y) \cdot z = x \cdot (y \cdot z) \tag{27}$$

   (h) 0 is the additive identity, meaning, for each $x \in R$,

$$x + 0 = x = 0 + x. \tag{28}$$

   (i) 1 is the multiplicative identity, meaning, for each $x \in R$,

$$x \cdot 1 = x = 1 \cdot x. \tag{29}$$

   (j) -1 is the additive inverse of 1, meaning $1 + (-1) = 0$.

(k) The distributive law holds, meaning, for all $x, y, z \in R$,

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z), \tag{30}$$

$$(y + z) \cdot x = (y \cdot x) + (z \cdot x). \tag{31}$$

2. $< \subseteq R \times R$ is a strict total ordering relation on R with a least element:

   (a) $<$ is antisymmetric, meaning there is no $x \in R$ such that $x < x$.

   (b) $<$ is transitive, meaning for all $x, y, z \in R$, if $x < y$ and $y < z$ then $x < z$.

   (c) $<$ is connected, meaning for all $x, y \in R$, if $x \neq y$ then either $x < y$ or $y < x$.

   (d) $<$ has a least element, meaning there exists a $z \in R$ such that for all $x \in R$, if $x \neq z$ then $z < x$. (Such a z is necessarily unique.)

3. $F : \mathbb{Z}^+ \to R$ is a partial function mapping de Bruijn indices naming first-order variables to their corresponding values (if any).

4. $S : \mathbb{Z}^+ \to \sum_{i \in \mathbb{Z}^+} (R^i \to R)$ is a partial function mapping de Bruijn indices naming second-order variables to their corresponding values (if any). The values are partial functions of variable arity from R to R. Here $\sum_{i \in \mathbb{Z}^+}$ denotes the indexed coproduct or disjoint union operation with i ranging over the positive integers.

By definition, M is an "integral model" when

$$(R, +, 0, 1, -1) = (\mathbb{Z}, +, 0, 1, -1) \tag{32}$$

is the standard ring of integers and $<$ puts the integers into the following order:

$$1, 2, ..., 0, -1, -2, ... \tag{33}$$

Let $M = (R, \cdot, +, 0, 1, -1, <, F, S)$ be a model. Let $y \in R$. Define the model

$$M[x_1 \mapsto y] = (R, \cdot, +, 0, 1, <, F', S) \tag{34}$$

by letting

$$\begin{array}{rcl} F'(1) & = & y, \\ F'(n+1) & = & F(n). \end{array} \tag{35}$$

Let n be a positive integer. Let $g : R^n \to R$ be a partial function. Define the model

$$M[f_1 \mapsto g] = (R, \cdot, +, 0, 1, F, S') \tag{36}$$

by letting

$$\begin{array}{rcl} S'(1) & = & g, \\ S'(n+1) & = & S(n). \end{array} \tag{37}$$

The definitions just given of $M[x_1 \mapsto y]$ and $M[f_1 \mapsto g]$ explain how to update a model with a new variable mapping at the least de Bruijn index, pushing up all the existing de Bruijn index mappings. These operations are useful for dealing with quantification in the denotational semantics which follows below.

Also helpful for defining the denotational semantics will be an extension of the $[i]$ notation previously defined. Previously, $[i]$ was defined as $\{1, ..., i\}$ for a positive integer i. Generalizing this to a ring R for an

arbitrary model M, let $[i]$ be defined as $\{x \in R \mid z \leq x \leq i\}$, where z is the least element of R under $<$ and $\leq$ is the non-strict version of $<$.

Given a model,

$$M := (R, \cdot, +, 0, 1, -1, <, F, S), \tag{38}$$

it is possible to define the denotations of terms and formulas, such as by the following recursive definition. The denotation of a term is an element of R, whereas the denotation of a formula is a truth value. The set of truth values is the set $\{0, 1\}$, where 0 represents false and 1 represents true. Due to the partiality of F, S, and the functions in the codomain of S, not every term or formula has a denotation in every model. The following recursive clauses define the denotation $\delta_M(\tau)$ for each term τ which has a denotation in M and the denotation $\delta_M(\phi)$ for each term ϕ which has a denotation in M.

1. For all positive integers i,
$$\delta_M(x_i) := F(i), \tag{39}$$
if $F(i)$ is defined.

2. For all positive integers i and non-empty sequences of terms $\tau_1, ..., \tau_n$,
$$\delta_M(f_i(\tau_1, ..., \tau_n)) := S(i)(\delta_M(\tau_1), ..., \delta_M(\tau_n)), \tag{40}$$
if $S(i)$ is defined, and $\delta_M(\tau_i)$ is defined for each i, and
$$S(i)(\delta_M(\tau_1), ..., \delta_M(\tau_n)) \tag{41}$$
is defined.

3. For all terms τ, μ,
$$\delta_M(\tau + \mu) := \delta_M(\tau) + \delta_M(\mu), \tag{42}$$
if $\delta_M(\tau)$ and $\delta_M(\mu)$ are defined.

4. For all terms τ, μ,
$$\delta_M(\tau \cdot \mu) := \delta_M(\tau) \cdot \delta_M(\mu), \tag{43}$$
if $\delta_M(\tau)$ and $\delta_M(\mu)$ are defined.

5. For all terms τ, μ,
$$\delta_M(\mathrm{ind}_<(\tau, \mu)) := \begin{cases} 1 & \delta_M(\tau) < \delta_M(\mu), \\ 0 & \text{otherwise.} \end{cases} \tag{44}$$

6. $\delta_M(0) := 0$.

7. $\delta_M(1) := 1$.

8. $\delta_M(-1) := -1$.

9. For all terms τ, μ,
$$\delta_M(\tau = \mu) := 1 \tag{45}$$
if $\delta_M(\tau)$ and $\delta_M(\mu)$ are defined and
$$\delta_M(\tau) = \delta_M(\mu). \tag{46}$$

10. For all terms τ, μ,
$$\delta_M(\tau = \mu) := 0 \tag{47}$$
if $\delta_M(\tau)$ and $\delta_M(\mu)$ are defined and
$$\delta_M(\tau) \neq \delta_M(\mu). \tag{48}$$

11. For all first-order formulas ϕ,
$$\delta_M(\neg\phi) := 1 - \delta_M(\phi), \tag{49}$$
if $\delta_M(\phi)$ is defined.

12. For all first-order formulas ϕ, ψ,
$$\delta_M(\phi \wedge \psi) := \min\{\delta_M(\phi), \delta_M(\psi)\}, \tag{50}$$
if $\delta_M(\phi)$ and $\delta_M(\psi)$ are defined.

13. For all first-order formulas ϕ, ψ,
$$\delta_M(\phi \vee \psi) := \max\{\delta_M(\phi), \delta_M(\psi)\}, \tag{51}$$
if $\delta_M(\phi)$ and $\delta_M(\psi)$ are defined.

14. For all polynomial terms β and first-order formulas ϕ,
$$\delta_M(\forall < \beta.\ \phi) := \min(\{\delta_{M[x_1 \mapsto i]}(\phi) | i \in [\delta_M(\beta)]\} \cup \{1\}), \tag{52}$$
if $\delta_{M[x_1 \mapsto i]}(\phi)$ is defined for each i.

15. For all polynomial terms β and first-order formulas ϕ,
$$\delta_M(\exists < \beta.\ \phi) := \max(\{\delta_{M[x_1 \mapsto i]}(\phi) | i \in [\delta_M(\beta)]\} \cup \{0\}), \tag{53}$$
if $\delta_{M[x_1 \mapsto i]}(\phi)$ is defined for each i.

16. For all polynomial terms γ and non-empty sequences of polynomial terms $\beta_1, ..., \beta_n$ and Σ^1_1 formulas ϕ,
$$\delta_M(\exists_f < \gamma(< \beta_1, ..., < \beta_n).\ \phi) := \max_{g \in [\delta_M(\gamma)]^{\prod_{j \in [n]} [\delta_M(\beta_j)]}} \delta_{M[f_1 \mapsto g]}(\phi), \tag{54}$$
if $\delta_{M[f_1 \mapsto g]}(\phi)$ is defined for each g.

Note: this truth condition cannot assign a truth value to the formula when $\delta_M(\gamma) < 1$. In such a case, the only value that g can take is the empty set, and if the formula ϕ references f_1, then $\delta_{M[f_1 \mapsto \emptyset]}(\phi)$ is undefined. Since we do not care about cases where ϕ does not reference all quantified variables, it is not important for our purposes that this truth condition fails to apply when $\delta_M(\gamma) < 1$. The same comments apply when $\delta_M(\beta_i) < 1$ for some i.

3 Semicircuits

For the purposes of simplifying the presentation of the arithmetization, we introduce an intermediate representation called a "semcircuit" which disintegrates the structure of Σ_1^1 formulas while maintaining abstractness.

For any $i \in \mathbb{N}$, the notation $[i]$ denotes the set $\{j \in \mathbb{N} \mid 1 \leq j \leq i\}$. $\mathbb{Z}^+$ denotes the set of positive integers.

Given a model,

$$M := (R, \cdot, +, 0, 1, -1, <, F, S), \tag{55}$$

a semcircuit consists of the following data:

1. A non-negative integer n, the number of first-order free variables.
2. A vector $\vec{a}$ of positive integers, the vector of arities of second-order free variables.
3. A non-negative integer m, the number of first-order existentially quantified variables.
4. A vector $\vec{b}$ of positive integers, the vector of arities of second-order existentially quantified variables.
5. A non-negative integer u, the number of universally quantified variables.
6. A (non-strictly) increasing vector $\vec{\nu} = \nu_1, ..., \nu_m$ of non-negative integers such that $\forall i, 0 \leq \nu_i \leq u$. This stores the number of universal quantifiers mentioned prior to a particular existential quantifier.
7. A vector $\vec{r} = r_1, ..., r_{|\vec{a}|}$ of non-negative integers representing the number of function calls to each free function.
8. A vector $\vec{q} = q_1, ..., q_{|\vec{b}|}$ of non-negative integers representing the number of function calls to each existentially quantified function.
9. A vector $\vec{\mathcal{P}} = \mathcal{P}_1, ..., \mathcal{P}_{|\vec{\mathcal{P}}|}$ of syntactic descriptions of ring terms. These include expressions appearing as arguments to functions and bounds. The elements of $\vec{\mathcal{P}}$ are generated by the following grammar;
   $\langle \mathrm{P} \rangle ::= 0 \mid 1 \mid -1 \mid \langle \mathrm{P} \rangle + \langle \mathrm{P} \rangle \mid \langle \mathrm{P} \rangle \cdot \langle \mathrm{P} \rangle \mid \mathrm{ind}_<(\langle \mathrm{P} \rangle, \langle \mathrm{P} \rangle) \mid \vec{v}_n \mid \vec{u}_{\langle \mathrm{u} \rangle} \mid \vec{s}_{\langle \mathrm{s} \rangle} \mid \vec{o}_{\langle \mathrm{o} \rangle} \mid \vec{\sigma}_{\langle \sigma \rangle}$
   $\langle \mathrm{v} \rangle := [n],\ \langle \mathrm{u} \rangle := [u],\ \langle \mathrm{s} \rangle := [m],\ \langle \mathrm{o} \rangle := \{(i,j) \mid i \in [\,|\vec{a}|\,] \wedge j \in [r_i]\},\ \langle \sigma \rangle := \{(i,j) \mid i \in [\,|\vec{b}|\,] \wedge j \in [q_i]\}$
   $\vec{v}_{\langle \mathrm{v} \rangle}$, $\vec{u}_{\langle \mathrm{u} \rangle}$, $\vec{s}_{\langle \mathrm{s} \rangle}$, $\vec{o}_{\langle \mathrm{o} \rangle}$, and $\vec{\sigma}_{\langle \sigma \rangle}$ act as references to the addresses of pre-computed values stored elsewhere in the circuit.
10. A vector of vectors of vectors $\vec{w} = w_{1,1,1}, ..., w_{1,1,a_1}, ..., w_{1,r_1,a_1}, ..., w_{|\vec{a}|,r_{|\vec{a}|},a_{|\vec{a}|}}$ of elements taken from $[\,|\vec{\mathcal{P}}|\,]$. $w_{i,j,k}$ corresponds to the kth argument to the jth function call to the ith function. These link argument calls to **free functions** to their corresponding ring terms.
11. A vector of vectors of vectors $\vec{\omega} = \omega_{1,1,1}, ..., \omega_{1,1,b_1}, ..., \omega_{1,s_1,b_1}, ..., \omega_{|\vec{b}|,s_{|\vec{b}|},b_{|\vec{s}|}}$ of elements taken from $[\,|\vec{\mathcal{P}}|\,]$. $\omega_{i,j,k}$ corresponds to the kth argument to the jth function call to the ith function. These link argument calls to **existentially quantified functions** to their corresponding ring terms.
12. A vector $\vec{V} = V_1, ..., V_u$ of elements taken from $[\,|\vec{\mathcal{P}}|\,]$. These link the bounds for each universally quantified variable to their corresponding ring term.

13. A vector $\vec{S} = S_1, ..., S_m$ of elements taken from $[\,|\vec{\mathcal{P}}|\,]$. These link the bounds for each existentially quantified first-order variable to their corresponding ring term.

14. A vector $\vec{G} = G_1, ..., G_{|\vec{b}|}$ of elements taken from $[\,|\vec{\mathcal{P}}|\,]$. These link the bounds for the outputs of each existentially quantified second-order variable to their corresponding ring term.

15. A vector of vectors $\vec{B} = B_{1,1}, ..., B_{1,b_1}, ..., B_{|\vec{b}|,b_{|\vec{b}|}}$ of elements taken from $[\,|\vec{\mathcal{P}}|\,]$. These link the bounds for the inputs of each existentially quantified second-order variable to their corresponding ring term.

16. A set $\mathcal{S}$ which is either empty or contains exactly one element generated by the following grammar: $\langle \mathrm{S} \rangle ::= \neg\langle \mathrm{S} \rangle \mid \langle \mathrm{S} \rangle \wedge \langle \mathrm{S} \rangle \mid \langle \mathrm{S} \rangle \vee \langle \mathrm{S} \rangle \mid \langle \mathrm{S} \rangle \to \langle \mathrm{S} \rangle \mid \langle P \rangle = \langle P \rangle$

Let $\mathcal{C} = (n, \vec{a}, m, \vec{b}, u, \vec{\nu}, \vec{r}, \vec{q}, \vec{\mathcal{P}}, \vec{w}, \vec{\omega}, \vec{V}, \vec{S}, \vec{G}, \vec{B}, \mathcal{S})$ be a semicircuit. An instance value for $\mathcal{C}$ consists of:

1. A vector $\vec{v} = v_1, ..., v_n$ of R elements which act as instance values for all n free first-order variables.

2. A vector $\vec{f} = f_1, ..., f_{|\vec{a}|}$ of sets such that

$$\forall i \in [\,|\vec{a}|\,], f_i \subseteq (R^{a_i} \times R) \wedge \forall t \in R^{a_i}, \forall e_1 \in R, \forall e_2 \in R, (t, e_1) \in f \wedge (t, e_2) \in f_i \to e_1 = e_2. \tag{56}$$

   In other words, each f_i is a partial function of arity a_i over R. These act as lookup table instances for all free second-order functions.

Advice values for $\mathcal{S}$ consists of:

1. A vector $\vec{g} = g_1, ..., g_{|\vec{b}|}$ of sets such that

$$\forall i \in [\,|\vec{b}|\,], g_i \subseteq (R^{b_i} \times R) \wedge \forall t \in R^{b_i}, \forall e_1 \in R, \forall e_2 \in R, (t, e_1) \in f \wedge (t, e_2) \in g_i \to e_1 = e_2. \tag{57}$$

   In other words, each g_i is a partial function of arity b_i over R. These act as lookup table instances for all existentially quantified second-order functions.

2. A set $U \subseteq R^u$ of u-tuples. Conceptually, U must contain all and only the possible combinations of values which the universal variables may take. From this, one may be tempted to define U uniquely upfront, but this leads to circular definitions stemming from polynomial constraints on universally quantified variables needing to reference other bound variables. As such, U will be forced to be a unique set later by constraint 67. The order of each entry within an element of U corresponds to the order in which each corresponding universally quantified variable appears in the original formula.

3. A vector $\vec{s} = s_1, ..., s_m$ of functions taken from $U \to R$, where m is the number of existentially quantified first-order variables. Each s can be thought of as something like a Skolem function in spirit, though skolemization is not actually possible on Σ_1^1 formulas with polynomial bounds. [1] The ordering of $\vec{s}$ corresponds to the order in which each corresponding existentially quantified variable appears in the original formula.

[1] To see this, consider the following. We first replace the first-order existential quantifiers with second-order existential quantifiers asserting the existence of a Skolem function. These second-order quantifiers will share bounds with the universal quantifiers they expect as arguments. These bounds may reference other universally quantified variables. This prevents us from moving the second-order quantifiers out of scope of the first-order universal quantifiers, thus preventing the skolemization from completing. One cannot, in general, specify the domain of the Skolem functions without referencing universally quantified variables.

4. A vector of vectors $\vec{o} = o_{1,1}, ..., o_{1,r_1}, ..., o_{|\vec{a}|,r_{|\vec{a}|}}$ of functions taken from $U \to R$. $o_{i,n}$ corresponds to the nth function call to the **free function** corresponding to f_i. These correspond to the **output** values of each function call.

5. A vector of vectors $\vec{\sigma} = \sigma_{1,1}, ..., \sigma_{1,q_1}, ..., \sigma_{|\vec{b}|,q_{|\vec{b}|}}$ of functions taken from $U \to R$. $\sigma_{i,n}$ corresponds to the nth function call to the **existentially quantified function** corresponding to g_i. These correspond to the **output** values of each function call.

Before the semantics can be defined, evaluation functions for the syntactic descriptions are required. From a syntactic description of ring terms, we may produce a function $U \to R$ via;

$\text{eval}_R(0) := \lambda x.0$

$\text{eval}_R(1) := \lambda x.1$

$\text{eval}_R(-1) := \lambda x.-1$

$\text{eval}_R(a + b) := \lambda x.\text{eval}_R(a)(x) + \text{eval}_R(b)(x)$

$\text{eval}_R(a \cdot b) := \lambda x.\text{eval}_R(a)(x) \cdot \text{eval}_R(b)(x)$

$\text{eval}_R(\text{ind}_<(a, b)) := \lambda x.\text{ind}_<(\text{eval}_R(a)(x), \text{eval}_R(b)(x))$

$\text{eval}_R(\vec{v}_i) := \lambda x.v_i$

$\text{eval}_R(\vec{u}_i) := \lambda x.x_i$

$\text{eval}_R(\vec{s}_i) := s_i$

$\text{eval}_R(\vec{o}_{(i,j)}) := o_{i,j}$

$\text{eval}_R(\vec{\sigma}_{(i,j)}) := \sigma_{i,j}$

From a syntactic description of propositions, we may produce a function $U \to \mathbb{B}$ via;

$\text{eval}_B(\neg a) := \lambda x.\neg\text{eval}_B(a)(x)$

$\text{eval}_B(a \wedge b) := \lambda x.\text{eval}_B(a)(x) \wedge \text{eval}_B(b)(x)$

$\text{eval}_B(a \vee b) := \lambda x.\text{eval}_B(a)(x) \vee \text{eval}_B(b)(x)$

$\text{eval}_B(a \to b) := \lambda x.\text{eval}_B(a)(x) \to \text{eval}_B(b)(x)$

$\text{eval}_B(a = b) := \text{eval}_R(a) = \text{eval}_R(b)$

Let

$$\mathcal{C} = (n, \vec{a}, m, \vec{b}, u, \vec{\nu}, \vec{r}, \vec{q}, \vec{\mathcal{P}}, \vec{w}, \vec{\omega}, \vec{V}, \vec{S}, \vec{G}, \vec{B}, \mathcal{S}) \tag{58}$$

be a semicircuit. Let

$$X = (\vec{v}, \vec{f}) \tag{59}$$

be an instance value for $\mathcal{C}$. We say that X is true, or in other words $\mathcal{C}$ is satisfiable on X, if and only if there exists

$$Y = (\vec{g}, U, \vec{s}, \vec{o}, \vec{\sigma}), \tag{60}$$

an advice value for $\mathcal{C}$, such that the following hold:

1. The main proposition must be true;

$$\forall x \in U, \forall e \in \mathcal{S}, \text{eval}_B(e)(x) = 1 \tag{61}$$

2. Each call to every free function must be a valid input-output pair.

$$\forall x \in U, \forall i \in [\,|\vec{f}|\,], \forall j \in [r_i], ((\text{eval}_R(\mathcal{P}_{w_{i,j,1}})(x), ..., \text{eval}_R(\mathcal{P}_{w_{i,j,a_i}})(x)), o_{i,j}(x)) \in f_i \tag{62}$$

3. Each call to every existentially quantified function must be a valid input-output pair.

$$\forall x \in U, \forall i \in [\,|\vec{g}|\,], \forall j \in [q_i], ((\text{eval}_R(\mathcal{P}_{\omega_{i,j,1}})(x), ..., \text{eval}_R(\mathcal{P}_{\omega_{i,j,b_i}})(x)), \sigma_{i,j}(x)) \in g_i \tag{63}$$

4. Every existentially quantified first-order variable must be within its declared bounds.

$$\forall x \in U, \forall i \in [\,|\vec{s}|\,], s_i(x) < \text{eval}_R(\mathcal{P}_{S_i})(x) \tag{64}$$

5. Every possible output of each second-order existentially quantified function must be within its declared bounds.

$$\forall i \in [\,|\vec{g}|\,], \forall k \in g_i, \forall x \in U, \pi_2(k) < \text{eval}_R(\mathcal{P}_{G_i})(x) \tag{65}$$

6. Every possible input of each second-order existentially quantified function must be within its declared bounds.

$$\forall i \in [\,|\vec{g}|\,], \forall j \in [b_i], \forall k \in g_i, \forall x \in U, \pi_1(k)_j < \text{eval}_R(\mathcal{P}_{B_{i,j}})(x) \tag{66}$$

7. U contains exactly those combinations of values which are within each universally quantified variable's declared bounds.

$$U = \{t \in R^u \mid \forall i \in [u], t_i < \text{eval}_R(\mathcal{P}_{V_i})(t)\} \tag{67}$$

8. Values of existentially quantified variables should be constant with respect to universally quantified variables appearing later in a formula.

$$\forall i \in [\,|\vec{s}|\,], \forall m \in R^{\nu_i}, \exists c \in R, \forall n \in R^{u-\nu_i}, m^\frown n \in U \rightarrow s_i(m^\frown n) = c \tag{68}$$

where $m^\frown n$ denotes the concatenation of the tuples m and n.

Given instance values and some of our advice values, we can derive the rest of our required advice automatically. The advice which needs to be provided is called the "nondeterministic" advice. Only $\vec{g}$ and $\vec{s}$ are nondeterministic. All other advice can be derived automatically and are called "deterministic." The deterministic advice consists only of U, $\vec{o}$, and $\vec{\sigma}$.

Assuming our circuit was constructed from a Σ_1^1 formula, U's contents can be constructed in pieces based on the bounds of the universal quantifiers. We start with $u = 0$ and $U = \{\}$, which makes condition 67 hold trivially. We can note that the polynomials appearing within the bounds of universal quantifiers must be constant with respect to later universally quantified variables. This allows us to treat each polynomial bound as a function of only the universally quantified variables appearing before the one we are currently adding. Assuming U already incorporates all the quantifiers prior to the uth, we can incorporate the uth by the replecement

$$U := \{(t_1, ..., t_{u-1}, x) \mid (t_1, ..., t_{u-1}) \in U \wedge x < \text{eval}_R(\mathcal{P}_{V_u})(t_1, ..., t_{u-1})\} \tag{69}$$

Note that $t = (t_1, ..., t_{u-1})$ being within the old U implies that $\forall i \in [u-1], t_i < \text{eval}_R(\mathcal{P}_{V_i})(t)$. To satisfy $\forall i \in [u], t'_i < \text{eval}_R(\mathcal{P}_{V_i})(t')$ where $t' = (t_1, ..., t_{u-1}, x)$ we only need to additionally satisfy $x <$

$\text{eval}_R(\mathcal{P}_{V_u})(t')$, which is guaranteed as a consequence of the second conjunct in the formula defining U's replacement. This ensures that condition 67 holds at each step. We continue this procedure until all universal quantifiers are incorporated into U.

The values for $\vec{o}$ and $\vec{\sigma}$ can be calculated automatically by setting

$$o_{i,j}(x) := f_i(\text{eval}_R(\mathcal{P}_{w_{i,j,1}})(x), ..., \text{eval}_R(\mathcal{P}_{w_{i,j,a_i}})(x)) \tag{70}$$

and

$$\sigma_{i,j}(x) := g_i(\text{eval}_R(\mathcal{P}_{\omega_{i,j,1}})(x), ..., \text{eval}_R(\mathcal{P}_{\omega_{i,j,b_i}})(x)) \tag{71}$$

These are the unique values which guarantee that conditions 62 and 63 are satisfied.

4 Translating Σ_1^1 Formulas into Semicircuits

Assume, without loss of generality, that our Σ_1^1 formula is of the form

$$\Sigma\Phi\psi \tag{72}$$

where Σ represents the second-order existential quantifiers, Φ represents the first order (existential and universal) quantifiers, and ψ represents the internal (quantifier-free) proposition of the formula. This form allows us to separate the translation into stages.

Before the first stage, we must start with an empty semicircuit. This circuit contains the following;

1. The number, n, of free first-order variables consistent with those appearing in $\Sigma\Phi\psi$.
2. The vector, $\vec{a}$, of arities of free second-order variables consistent with those appearing in $\Sigma\Phi\psi$.
3. $m = 0$, $u = 0$, and $\vec{b}$, $\vec{\nu}$, $\vec{r}$, $\vec{q}$, $\vec{\mathcal{P}}$, $\vec{w}$, $\vec{\omega}$, $\vec{V}$, $\vec{S}$, $\vec{G}$, $\vec{B}$, $\mathcal{S}$ are all empty.

We next need to define how to add a ring/poly term into an already existing semicircuit. We will define a procedure which will manipulate and access the existing semicircuit, as a global variable called $\mathcal{C}$. The procedure for ring/poly terms will be referred to as "RingProc."

Given a ring/poly term, we will need to build up a syntactic representation of that term to store in $\vec{\mathcal{P}}$. We define this recursively in a mostly obvious way;

$\text{encode}_R(0) := 0$

$\text{encode}_R(1) := 1$

$\text{encode}_R(-1) := -1$

$\text{encode}_R(a + b) := \text{encode}_R(a) + \text{encode}_R(b)$

$\text{encode}_R(a \cdot b) := \text{encode}_R(a) \cdot \text{encode}_R(b)$

$\text{encode}_R(\text{ind}_<(a, b)) := \text{ind}_<(\text{encode}_R(a), \text{encode}_R(b))$

$\text{encode}_R(x_i) :=$
if x_i is free
then $\vec{v}_{|\vec{v}|-(i-|\vec{s}|)}$
else if x_i is universally quantified
then let j be the index of x_i within the tuple of universally quantified variables in $\vec{u}_j$
else if x_i is existentially quantified then $\vec{s}_{|\vec{s}|-i-\nu_{|\vec{s}|-i}}$

encode$_R(f_i^n(\tau_1, ..., \tau_n))$:=
run RingProc on τ_i for all i
let $t_i \in [\,|\vec{\mathcal{P}}|\,]$ be the index of the ring term for τ_i
if f_i^n is free
then
increment $r_{|\vec{f}|-(i-|\vec{g}|)}$
append $(t_1, ..., t_2)$ to $w_{|\vec{f}|-(i-|\vec{g}|)}$
$\vec{f}_{|\vec{f}|-(i-|\vec{g}|)}$
else if f_i^n is existentially quantified then
increment $\sigma_{|\vec{g}|-i}$
append $(t_1, ..., t_2)$ to $\omega_{|\vec{g}|-i}$
$\vec{g}_{|\vec{g}|-i}$

Notice that encode$_R$ will modify $\mathcal{C}$ and make a call to the (yet to be defined) RingProc procedure if it encounters a function variable. We can now define RingProc as.

$$\text{RingProc}(r) := \text{append encode}_R(r) \text{ to } \vec{\mathcal{P}}$$

Note that RingProc will end up running encode$_R$ and therefore modify $\mathcal{C}$ prior to the append operations completing. With this in hand we can describe the first stage of translation which incorporates second-order quantifiers into the circuit. We start with an empty semicircuit and run the following procedure;

2ndOrderProc($\exists_f < \gamma(< \beta_1, ..., < \beta_n).\phi$) := append n to $\vec{b}$
run RingProc(γ)
let $p_\gamma \in [\,|\vec{\mathcal{P}}|\,]$ be the index of the ring term for γ
append p_γ to $\vec{G}$
run RingProc(β_i) for each i
let $p_{\beta i} \in [\,|\vec{\mathcal{P}}|\,]$ be the index of the ring term for β_i
append $(p_{\beta 1}, ..., p_{\beta n})$ to $\vec{B}$
2ndOrderProc(ϕ)
2ndOrderProc(ϕ) := if ϕ is a first-order formula then 1stOrderProc(ϕ)

The second stage of translation incorporates first-order quantifiers into the circuit.

$$
\begin{aligned}
\text{1stOrderProc}(\exists < \gamma.\phi) := \;& \text{increment } m \\
& \text{run RingProc}(\gamma) \\
& \text{let } p_\gamma \in [\,|\vec{\mathcal{P}}|\,] \text{ be the index of the ring term for } \gamma \\
& \text{append } p_\gamma \text{ to } \vec{S} \\
& \text{append } u \text{ to } \vec{\nu} \\
& \text{1stOrderProc}(\phi) \\
\text{1stOrderProc}(\forall < \gamma.\phi) := \;& \text{increment } u \\
& \text{run RingProc}(\gamma) \\
& \text{let } p_\gamma \in [\,|\vec{\mathcal{P}}|\,] \text{ be the index of the ring term for } \gamma \\
& \text{append } p_\gamma \text{ to } \vec{V} \\
& \text{1stOrderProc}(\phi) \\
\text{1stOrderProc}(\phi) := \;& \text{if } \phi \text{ is a quantifier free order formula then PropProc}(\phi)
\end{aligned}
$$

The final stage of the translation incorporates the quantifier-free sub-formula into the circuit. The procedure, PropProc, which does this is defined as;

$$\text{PropProc}(\phi) := \text{append encode}_P(\phi) \text{ to } \mathcal{S} \tag{73}$$

It just makes a call to encode_P, which will convert ϕ into a syntactical representation.
encode_P can be defined recursively as

$\text{encode}_P(\neg\phi) := \neg\text{encode}_P(\phi)$

$\text{encode}_P(a \wedge b) := \text{encode}_P(a) \wedge \text{encode}_P(b)$

$\text{encode}_P(a \vee b) := \text{encode}_P(a) \vee \text{encode}_P(b)$

$\text{encode}_P(a \to b) := \text{encode}_P(a) \to \text{encode}_P(b)$

$\text{encode}_P(x = y) := \text{encode}_R(x) = \text{encode}_R(y)$

This completes the algorithm translating Σ_1^1 formulas into semicircuits.

5 Circuits and logic circuits

In order to simplify the presentation, we will look at the arithmetization process in stages. In the first stage, a Σ_1^1 formula gets transformed into a semicircuit. In the second stage, a semicircuit is converted into an arithmetic-circuit-like structure with an enriched constraint language; we call these structures "logic circuits." In arithmetic circuits, gate constraints are *local*: they apply to each row and they involve variables in the same row and nearby rows. Also, in arithmetic circuits, constraints are *polynomial equations*. In logic circuits, gate constraints are still local, but they are generalized from polynomial equations to polynomial equations and inequalities combined by logical connectives.

In order to simplify defining arithmetic circuits and logic circuits, we first look at a definition of what they both are, which we call an *abstract circuit*. An abstract circuit is an instantiation of an *abstract circuit scheme*. These notions will be defined after some intervening definitions.

Let F, A, I, E, and N be distinct constants. These constants are used to symbolize column types. A column is either fixed, advice, or instance. Separately, a column is either equality constrainable or not equality constrainable. Thus, the set of column types is:

$$\text{ColType} := \{F, A, I\} \times \{E, N\}. \tag{74}$$

Let ColTypes denote the set of vectors of column types.

Let PolyBound denote the set of polynomial degree bounds:

$$\text{PolyBound} := \mathbb{N}. \tag{75}$$

By definition, an "abstract circuit scheme" is a tuple $(D, \text{Constraint}, \text{Sat})$, where:

1. D is a ring. D is called the domain.
2. Constraint is a set family parameterized by a ring, and element from PolyBound, and another set acting as formal variables. In practice, Constraint will be a set of syntactical descriptions of constraints involving elements from the ring and variables which can be filled in from elsewhere.
3. $\text{Sat}_n^d : \text{Constraint}(D, d, [n] \times \mathbb{Z}) \to D^{[n]\times[r]} \to \mathbb{B}$

   is a function interpreting a constraint as a predicate over matrices. The formal variables $(j, k) \in [n]\times\mathbb{Z}$ act as coordinates within the matrix with j as a column index and k as a relative row reference.

Let $\mathcal{S} = (D, \text{Constraint}, \text{Sat})$ be an arbitrary abstract circuit scheme. An "$\mathcal{S}$-circuit," by definition, is a tuple

$$\mathcal{C} := (\vec{c}, d, \vec{g}, \vec{\ell}, r, \vec{e}, X), \tag{76}$$

where:

1. $\vec{c} = c_1, ..., c_n$ is a vector of column types.
2. $d \in$ PolyBound.
3. $\vec{g} = g_1, ..., g_m$ is a vector of gate constraints; $g_i \in \text{Constraint}(\vec{c}, d)$ for each $i \in [m]$.
4. $\vec{\ell} = \ell_1, ..., \ell_k$ is a vector of lookup constraints. For all $i \in [k]$, ℓ_i is a tuple $(\vec{x}, \vec{a})$, where for some positive integer q:
   (a) $\vec{x} = x_1, ..., x_q$, where for all $j \in [q]$, x_j is a polynomial of degree $\leq d$ with coefficients in D, where variables are pairs $(k, l) \in [n] \times \mathbb{Z}$ where k is a column index and l is a relative row reference. $\vec{x}$ is called the input expression. $l = 0$ refers to the current row, and in general, if i is the current row index, then l refers to row $i + l$.
   (b) $\vec{a} = a_1, ..., a_q$, where for all $j \in [q]$, a_j is a column index: $a_j \in [n]$. $\vec{a}$ is called the vector of lookup table columns.

   $\vec{\ell}$ is called the sequence of lookup arguments.
5. r is a positive integer. r is called the number of rows.
6. $\vec{e} = e_1, ..., e_t$, where for all $i \in [t]$, $e_i \in ([n] \times [r])^2$ is a pair of pairs of indices, representing an equality constraint in the form of absolute cell references.
7. $X \in D^{[u]\times[r]}$, where u is the number of fixed columns, contains the values of each fixed column in each row.

The denotational semantics for abstract circuits is defined as follows. An $\mathcal{S}$-circuit $\mathcal{C}$ denotes a relation, a subset of $D^{[v]\times[r]}$, where v is the number of instance columns. This relation is the set of all $Z \in D^{[v]\times[r]}$ such that there exists a $Y \in D^{[n]\times[r]}$ such that:

1. Let $i \in [v]$. Let j be the index of the ith instance column. For all $k \in [r]$,
$$Y_{j,k} = Z_{i,k}. \tag{77}$$
2. Let $i \in [u]$. Let j be the index of the ith fixed column. For all $k \in [r]$,
$$Y_{j,k} = X_{i,k}. \tag{78}$$
3. For all $i \in [m]$,
$$\mathrm{Sat}_n^d(g_i, Y) = 1. \tag{79}$$
4. Let $i \in [k]$. Let $((x_1, ..., x_q), \vec{a}) = \ell_i$. For all $j \in [r]$,
$$(x_1(Y, j), ..., x_q(Y, j)) \in \{(Y_{a_1,k}, ..., Y_{a_q,k}) \mid k \in [r]\}. \tag{80}$$
5. Let $i \in [t]$. Let $((x_0, y_0), (x_1, y_1)) = e_i$. Then:
$$Y_{x_0,y_0} = Y_{x_1,y_1}. \tag{81}$$

By definition, an "arithmetic circuit scheme" is an abstract circuit scheme $(\mathbb{F}, \mathrm{Constraint}, \mathrm{Sat})$ with a fixed $d \in \mathrm{PolyBound}$ such that:

1. $\mathbb{F}$ is a finite field.
2. For all D, d, and V, $\mathrm{Constraint}(D, d, V)$ is the set $D[V]$ of polynomials of degree $\leq d$.
3. For all d, n, and p, $\mathrm{Sat}_n^d(p)$ maps $Y \in \mathbb{F}^{[n]\times[r]}$ (for any $r \in \mathbb{N}$) to the truth value for $\forall i \in [r], p(Y, i) = 0$.

Here $p(Y, i)$ denotes the result of evaluating the polynomial p over the matrix Y at row i. This notation can be defined recursively, for all polynomials, as follows:

1. For a constant c, $c(Y, i) := c$.
2. For a variable (j, k), $(j, k)(Y, i) := Y_{j,i+k}$.
3. For a polynomial $x + y$, $(x + y)(Y, i) := x(Y, i) + y(Y, i)$.
4. For a polynomial $x \cdot y$, $(x \cdot y)(Y, i) := x(Y, i) \cdot y(Y, i)$.

For every field $\mathbb{F}$ there is a unique arithmetic circuit scheme $(\mathbb{F}, \mathrm{Constraint}, \mathrm{Sat})$. We refer to this arithmetic circuit scheme as $\mathrm{Arith}(\mathbb{F})$.

By definition, an "arithmetic circuit" is a tuple
$$\mathcal{C} := (\mathbb{F}, \vec{c}, d, \vec{p}, \vec{\ell}, r, \vec{e}, X), \tag{82}$$
such that $\mathbb{F}$ is a finite field and $(\vec{c}, d, \vec{p}, \vec{\ell}, r, \vec{e}, X)$ is an $\mathrm{Arith}(\mathbb{F})$-circuit.

Defining logic circuits is a bit more involved, because the gate constraint type first needs to be defined. The logic gate constraint type is parameterized by D, the type of constants, V, the type of variables (which are relative cell indices when this definition is used in context), and d, the polynomial degree bound.

Let D be a ring. Let V be a set. Let $d \in \mathrm{PolyBound}$. By definition, a "(D, V, d) logic constraint" is any expression generated by the following recursive clauses, where we shorten "(D, V, d) logic constraint" to just "constraint" since context makes it clear.

1. $\tau = \mu$ and $\tau < \mu$ are constraints, where τ and μ are polynomials of degree $\leq d$ with coefficients in D and variables in V.

2. $\neg\phi$ is a constraint, where ϕ is a constraint.

3. $\phi \wedge \psi$, $\phi \vee \psi$, and $\phi \to \psi$ are constraints, where ϕ and ψ are constraints.

By definition, a "logic circuit scheme" is an abstract circuit scheme $(D, \text{Constraint}, \text{Sat})$ such that:

1. For all D, d, and V, $\text{Constraint}(D, d, V)$ is the set of (D, V, d) logic constraints.

2. For all d, n, and g, $\text{Sat}_n^d(g)$ maps a matrix $Y \in D^{[n]\times[r]}$ (for any $r \in \mathbb{N}$) to the truth value for $\forall i \in [r], g(Y, i) = 1$.

The notation $g(Y, i)$ denotes the truth value of g at row i over the matrix Y. This can be defined recursively for all $(D, [n] \times \mathbb{Z}, d)$ logic constraints as follows:

$$(\tau = \mu)(Y, i) := \begin{cases} 1 & \tau(Y, i) = \mu(Y, i), \\ 0 & \text{otherwise.} \end{cases} \tag{83}$$

$$(\tau < \mu)(Y, i) := \begin{cases} 1 & \tau(Y, i) < \mu(Y, i), \\ 0 & \text{otherwise.} \end{cases} \tag{84}$$

$$(\neg\phi)(Y, i) := 1 - \phi(Y, i). \tag{85}$$

$$(\phi \wedge \psi)(Y, i) := \phi(Y, i) \cdot \psi(Y, i). \tag{86}$$

$$(\phi \vee \psi)(Y, i) := \phi(Y, i) + \psi(Y, i) - (\phi(Y, i) \cdot \psi(Y, i)). \tag{87}$$

$$(\phi \to \psi)(Y, i) := (\neg\phi \vee \psi)(Y, i). \tag{88}$$

For every ring D there is a unique logic circuit scheme $(D, \text{Constraint})$. We refer to this logic circuit scheme as $\text{Logic}(D)$.

By definition, a "logic circuit" is a tuple

$$\mathcal{C} := (D, \vec{c}, d, \vec{g}, \vec{\ell}, r, \vec{e}, X), \tag{89}$$

such that D is a ring and $(\vec{c}, d, \vec{g}, \vec{\ell}, r, \vec{e}, X)$ is a $\text{Logic}(D)$-circuit.

As defined in this paper, Σ_1^1 arithmetization consists of two steps: compiling a Σ_1^1 formula to a logic circuit, and then compiling the logic circuit to an arithmetic circuit. These translations are required to be semantics preserving.

6 Compiling semicircuits to logic circuits

This section describes a semantics preserving process for compiling semicircuits to logic circuits over $\mathbb{Z}$. The process is semantics preserving in the sense that the source formula which produced the semicircuit is true on a given input if and only if the resulting logic circuit is satisfiable on the instance corresponding to the input.

An input to a semicircuit, $\mathcal{C}$, consists of a value for each free variable. A value for a first-order free variable is an integer, while a value for a second-order free variable is a function table which defines all inputs to the function that are used in evaluating the truth value of $\mathcal{C}$. These correspond to one instance column for each free first-order variable, for each entry in $\vec{v}$, and $n + 1$ instance columns for each free second-order variable, for each entry in $\vec{f}$, of arity n. In this instance, the value of a free first-order variable

is duplicated at each row in its instance column, and the value of a free second-order variable is stored in its instance columns, with duplicate rows as needed to fill up the function table in the instance.

Every logic circuit has a fixed number of rows. This limits the inputs for which we can have semantics preservation to those where the function tables fit into the instance table.

The advice columns of a circuit compiling a semicircuit to a logic circuit hold the values of the universally and existentially quantified variables as well as precomputed values for some subexpressions. For each first-order quantified variable, we have one advice column. For each n-ary second-order (existentially) quantified variable in $\vec{g}$, we have $(n+1)$ advice columns, holding the function table of the witness. U will be converted into u advice columns containing all the tuples within U stored in lexicographic order. Each term depending on a tuple $t \in U$ will become one advice column with the value corresponding to t being stored in the same row as the tuple t. This applies to $\vec{s}$, $\vec{o}$, and $\vec{\sigma}$. This allows the values within $\vec{w}$, $\vec{\omega}$, $\vec{G}$, $\vec{B}$, $\vec{S}$, and $\vec{V}$ to be replaced with matrix coordinates/relative row references. These data, along with $\vec{\mathcal{P}}$ and $\vec{\mathcal{S}}$ will be incorporated into the circuit constraints. Additionally, every call to $\text{ind}_<$ within any term in $\vec{\mathcal{P}}$ will also need an additional column for its precomputed values.

The columns for U, which will later be called $\vec{u}_i$ for each universally quantified variable i, are not generally going to be the contents of U on the nose. Instead, it may also contain additional "dummy" rows which facilitate completeness checking. These dummy rows will contain values which produce a bound of 0. Such cases denote empty quantifier occurrences. There will also be an additional column, denoted $\vec{\delta}$, containing a flag indicating if a row is a dummy.

We can construct the $\vec{u}_i$s and $\vec{\delta}$ algorithmically. First, we need a function which can lexicographically increment a prefix of us.

$$\text{inc}_1((u_1)) := (u_1 + 1) \tag{90}$$

$$\begin{aligned} \text{inc}_j((u_1, ..., u_j)) := \quad & \text{if } u_j + 1 = \text{eval}(\mathcal{P}_{V_j})(u_1, ..., u_{j-1}) \\ & \text{then } \text{inc}_{j-1}((u_1, ..., u_{j-1}))^\frown(0) \\ & \text{else } (u_1, ..., u_j + 1) \end{aligned} \tag{91}$$

We start with $u_{i,1} = 0$ for all i. If row k has already been added and we are adding row $k+1$, we do one of the following;

1. If j is the smallest index such that $\text{eval}(\mathcal{P}_{V_j})(u_{1,k}, ..., u_{j,k}) = 0$, then set $(u_{1,k+1}, ..., u_{j-1,k+1}) := \text{inc}(u_{1,k}, ..., u_{j-1,k})$ and set $u_{l,k+1} := 0$ for all l between j and u, inclusive.

2. If none of the bounds are 0, we simply set $(u_{1,k+1}, ..., u_{u,k+1}) := \text{inc}(u_{1,k}, ..., u_{u,k})$.

If we reach a point where we need to call $\text{inc}_1((u_1))$ while $u_1 + 1 = \text{eval}(\mathcal{P}_{V_1})$ then we are done filling out us.

We fill out $\vec{\delta}$ by checking at each row if there is an index j such that $\text{eval}(\mathcal{P}_{V_j})(u_{1,k}, ..., u_{j,k}) = 0$. If so, $\delta_k = 1$, otherwise $\delta_k = 0$.

Whenever we have a dummy row, the entries for $\vec{o}$ and $\vec{\sigma}$ get filled with a default value of 0. $\vec{s}$ is a bit more complicated as its correctness criterion relates values between subsequent rows. For $\vec{s}_i$, all non-dummy rows will be equipped with values such that $\forall k, l \in [r], (\forall j < \nu_i, \vec{u}_{j,k} = \vec{u}_{j,l}) \to \vec{s}_{i,k} = \vec{s}_{i,l}$. Given a dummy row k and non-dummy row l satisfying $\forall j < \nu_i, \vec{u}_{j,k} = \vec{u}_{j,l}$, s_k will have the value of s_l. Any dummy rows not covered by this will have their value filled in with 0.

Define a logic circuit $\mathcal{C}$, which arithmetizes the semicircuit $\mathcal{C}$, as follows:

1. The number of instance columns of $\mathcal{C}$ is

$$|\vec{v}| + \sum_{i \in |\vec{f}|} (a_i + 1). \tag{92}$$

The instance columns are named as follows:

$$\begin{array}{rl} \vec{v}_i & \text{column of values of } v_i \\ \vec{a}_{i,j} & j\text{th input column of } f_i \\ \vec{b}_i & \text{output column of } f_i \end{array} \tag{93}$$

This naming scheme assumes a total ordering of the free variables, so their arities and de Bruijn indices can be indexed by indices. Any total ordering will do.

2. The number of advice columns of $\mathcal{C}$ is

$$\sum_{i \in |\vec{g}|} (b_i + 1) + u + 1 + |\vec{s}| + |\vec{o}| + |\vec{\sigma}| + l. \tag{94}$$

Where l denotes the number of calls to $\mathrm{ind}_<$.

The advice columns are named as follows:

$$\begin{array}{rl} \vec{c}_{i,j} & j\text{th input column of } g_i \\ \vec{d}_i & \text{output column of } g_i \\ \vec{u}_i & \text{column of values of } i\text{th universal variable} \\ \vec{\delta} & \text{column of flags indicating if we are at a dummy row} \\ \vec{s}_i & \text{column of values of } i\text{th existential variable } s_i \\ \vec{o}_{i,j} & \text{column corresponding to the free function application subterms } o_{i,j} \\ \vec{\sigma}_{i,j} & \text{column corresponding to the existential function application subterms } \sigma_{i,j} \\ \vec{l}_i & \text{output column of } i\text{th } \mathrm{ind}_< \text{ application subterm} \end{array} \tag{95}$$

3. There is one fixed column equal to the zero vector and written as $\vec{0}$. There is one fixed column equal to one at all rows and written as $\vec{1}$. There is one fixed column equal to one at all rows except for the last row, where it is equal to one; this is written as ι.

4. The row count is r. Choose r such that there are enough rows to represent witnesses for all true instances for which semantics preservation is required.

5. The equality constrainable columns are $\{\vec{0}\} \cup \{\vec{u}_i\}_{i=1}^{|\vec{u}|}$.

6. The equality constraints set $u_{i,1} = 0$ for all $i \in [m]$.

7. There are the following gate constraints checking that our data have the correct structure;

   (a) *Instance function tables define functions.* For all $i \in [1, n]$ and $j \in [r]$,

$$\begin{array}{ll} & \iota_j = 1 \\ \vee & (a_{i,1,j} = a_{i,1,j+1} \wedge \cdots \wedge a_{i,k_i,j} = a_{i,k_i,j+1} \wedge b_{i,j} = b_{i,j+1}) \\ \vee & \mathrm{Lex}_<((a_{i,1,j}, ..., a_{i,k_i,j}), (a_{i,1,j+1}, ..., a_{i,k_i,j})) \end{array} \tag{96}$$

   The relation $\mathrm{Lex}_<$ is defined, by recursion on the length of the input vectors, as follows:

$$\mathrm{Lex}_<((x), (y)) = x < y \tag{97}$$

$$\mathrm{Lex}_<((w, \vec{x}), (y, \vec{z})) = w < y \vee (w = y \wedge \mathrm{Lex}_<(\vec{x}, \vec{z})). \tag{98}$$

(b) *Existential function tables define functions.* For all $i \in [1, m]$ and $j \in [r]$,

$$\begin{array}{ll} & \iota_j = 1 \\ \vee & (c_{i,1,j} = c_{i,1,j+1} \wedge \cdots \wedge c_{i,k_i,j} = c_{i,k_i,j+1} \wedge d_{i,j} = d_{i,j+1}) \\ \vee & \text{Lex}_<((c_{i,1,j}, ..., c_{i,k_i,j}), (c_{i,1,j+1}, ..., c_{i,k_i,j})) \end{array} \tag{99}$$

(c) *First-order instance variable column values are uniform.* For all $i \in [\,|\vec{v}|\,]$ and $j \in [r]$,

$$v_{i,j} = v_{i,j+1}. \tag{100}$$

8. Every expression in $\mathcal{P}$ gets converted into a polynomial

$\text{eval}_k(0) := 0$
$\text{eval}_k(1) := 1$
$\text{eval}_k(-1) := -1$
$\text{eval}_k(a + b) := \text{eval}_k(a) + \text{eval}_k(b)$
$\text{eval}_k(a \cdot b) := \text{eval}_k(a) \cdot \text{eval}_k(b)$
$\text{eval}_k(\text{ind}_<(a, b)) := \vec{l}_{w,k}$ where w is the column associated with this respective $\text{ind}_<$ call.
$\text{eval}_k(\vec{v}_i) := \vec{v}_{i,k}$
$\text{eval}_k(\vec{u}_i) := \vec{u}_{i,k}$
$\text{eval}_k(\vec{s}_i) := \vec{s}_{i,k}$
$\text{eval}_k(\vec{o}_{(i,j)}) := \vec{o}_{i,j,k}$
$\text{eval}_k(\vec{\sigma}_{(i,j)}) := \vec{\sigma}_{i,j,k}$

9. $\mathcal{S}$ gets converted into a logic constraint through the map

$\text{eval}_k(\neg a) := \neg\text{eval}_k(a)$
$\text{eval}_k(a \wedge b) := \text{eval}_k(a) \wedge \text{eval}_k(b)$
$\text{eval}_k(a \vee b) := \text{eval}_k(a) \vee \text{eval}_k(b)$
$\text{eval}_k(a \to b) := \text{eval}_k(a) \to \text{eval}_k(b)$
$\text{eval}_k(x = y) := \text{eval}_k(x) = \text{eval}_k(y)$

Constraint 61 stating that the quantifier-free proposition is true becomes the gate constraint

$$\forall k \in [r], \delta_k = 1 \vee \text{eval}_k(\mathcal{S}) = 1 \tag{101}$$

10. We need a constraint declaring that the values of $\vec{\delta}$ are correct.

$$\forall k \in [r], (\delta_k = 0 \wedge \neg \bigvee_{i=1}^{u} \text{eval}_k(\vec{\mathcal{P}}_{V_i}) = 0) \vee (\delta_k = 1 \wedge \bigvee_{i=1}^{u} \text{eval}_k(\vec{\mathcal{P}}_{V_i}) = 0) \tag{102}$$

11. We introduce the additional gate constraint guaranteeing that precomputed values for the calls to $\text{ind}_<$ are correct.

$$\forall i \in [\,|\vec{l}|\,], \forall k \in [r], (\vec{l}_{i,k} = 0 \wedge \neg(\text{eval}_k(\mathcal{I}_i^1) < \text{eval}_k(\mathcal{I}_i^2))) \vee (\vec{l}_{i,k} = 1 \wedge \text{eval}_k(\mathcal{I}_i^1) < \text{eval}_k(\mathcal{I}_i^2)) \tag{103}$$

where $\mathcal{I}_i^1$ and $\mathcal{I}_i^2$ are the syntactical descriptions of the left and right arguments, respectively, to the ith call to $\text{ind}_<$.

12. Constraint 62 stating that the precomputed input and output values for every free function call is a valid input-output pair becomes the lookup argument

$$\forall k \in [r], \forall i \in [\,|\vec{f}|\,], \forall j \in [\,|o_i|\,], \delta_k = 1 \vee (\mathrm{eval}_k(\vec{\mathcal{P}}_{w_{i,j,1}}), ..., \mathrm{eval}_k(\vec{\mathcal{P}}_{w_{i,j,b_i}}), \vec{o}_{i,j,k}) \in (\vec{a}_{i,1}, ..., \vec{a}_{i,a_i}, \vec{b}_i) \tag{104}$$

13. Constraint 63 stating that the precomputed input and output values for every existentially quantified function call is a valid input-output pair becomes the lookup argument

$$\forall k \in [r], \forall i \in [\,|\vec{f}|\,], \forall j \in [\,|o_i|\,], \delta_k = 1 \vee (\mathrm{eval}_k(\vec{\mathcal{P}}_{\omega_{i,j,1}}), ..., \mathrm{eval}_k(\vec{\mathcal{P}}_{\omega_{i,j,b_i}}), \vec{\sigma}_{i,j,k}) \in (\vec{c}_{i,1}, ..., \vec{c}_{i,b_i}, \vec{d}_i) \tag{105}$$

14. Constraint 64 stating that every existentially quantified first-order variable is in bounds becomes the gate constraint

$$\forall k \in [r], \forall i \in [\,|\vec{s}|\,], \delta_k = 1 \vee s_{i,k} < \mathrm{eval}_k(\vec{\mathcal{P}}_{S_i}) \tag{106}$$

15. Constraint 65 stating that every output of every existentially quantified second-order function is within bounds becomes the gate constraint

$$\forall k \in [r], \forall i \in [\,|\vec{g}|\,], \vec{d}_{i,k} < \mathrm{eval}_k(\vec{\mathcal{P}}_{G_i}) \tag{107}$$

16. Constraint 66 stating that every input of every existentially quantified second-order function is within bounds becomes the gate constraint

$$\forall k \in [r], \forall i \in [\,|\vec{g}|\,], \forall j \in [b_i], \vec{c}_{i,j,k} < \mathrm{eval}_k(\vec{\mathcal{P}}_{B_{i,j}}) \tag{108}$$

17. Constraint 67, stating that the values for universally quantified variables are exactly those within bounds, requires a relation stating that some prefix of us, $(\vec{u}_{1,k}, \vec{u}_{2,k}, ..., \vec{u}_{j,k})$ is lexicographically one less than $(\vec{u}_{1,k+1}, \vec{u}_{1,k+1}, ..., \vec{u}_{j,k+1})$.

$$\mathrm{Next}_{1,k} := \vec{u}_{1,k} + 1 = \vec{u}_{1,k+1} \tag{109}$$

$$\mathrm{Next}_{j,k} := \left(\left(\bigwedge_{i=1}^{j-1} \vec{u}_{i,k} = \vec{u}_{i,k+1} \right) \wedge \vec{u}_{j,k} + 1 = \vec{u}_{j,k+1} \right) \vee (\vec{u}_{j,k} + 1 = \mathrm{eval}_k(\vec{\mathcal{P}}_{V_j}) \wedge \vec{u}_{j,k+1} = 0 \wedge \mathrm{Next}_{j-1,k}) \tag{110}$$

This is essentially equations 90 and 91 recast as a relation. We can use this to check the correctness of the U table via

$$\forall k \in [r], \iota_k = 1 \vee \left(\bigvee_{i=1}^{u} \mathrm{eval}_k(\vec{\mathcal{P}}_{V_i}) = 0 \wedge \mathrm{Next}_{i-1,k} \wedge \bigwedge_{l=i}^{u} \vec{u}_{l,k} = \vec{u}_{l,k+1} = 0 \right) \vee \mathrm{Next}_{u,k} \tag{111}$$

18. Constraint 68 stating that values of existentially quantified variables do not depend on universal quantifiers later in the formula becomes the gate constraint for all $k \in [r]$ and all $i \in [\,|\vec{s}|\,]$

$$\iota_k = 1 \vee \left(\left(\bigwedge_{j=1}^{\nu_i} \vec{u}_{j,k} = \vec{u}_{j,k+1} \right) \rightarrow \vec{s}_{i,k} = \vec{s}_{i,k+1} \right). \tag{112}$$

7 Compiling logic circuits to arithmetic circuits

This section describes a semantics preserving process for compiling logic circuits over $\mathbb{Z}$ to arithmetic circuits over a sufficiently large field. This translation is semantics preserving for instances within defined bounds.

This process uses lookup tables and byte decompositions to model truth tables for comparison operations, which allows for logical operations to be defined as polynomials, embedding Boolean algebra into the field algebra. Each subformula of the form $\tau = \mu$ or $\tau < \mu$ in a gate constraint of the source circuit gives rise to advice columns in the resulting circuit:

1. A vector of advice columns holding the byte decompositions of the values of $\mu - \tau$ at each row.
2. A vector of advice columns holding the truth values of the statements $(\mu - \tau)_i = 0$ at each row, where $(\mu - \tau)_i$ denotes the ith byte of the byte decomposition of $\mu - \tau$.

These advice columns are sufficient to compute the truth value of each subformula of each gate constraint as a polynomial. In addition to polynomial gate constraints expressing that each logic gate constraint in the source circuit is satisfied based on the mentioned advice columns, there are polynomial gate and lookup constraints expressing that the values of the mentioned advice columns are consistent and correct. The correctness of byte decompositions is checked using a polynomial gate constraint, and range checks implemented using lookup constraints. The correctness of the $(\mu - \tau)_i = 0$ truth value advice columns is checked using a truth table implemented using a lookup constraint.

All fixed, instance, and advice columns from the source circuit carry over into the resulting circuit. All equality and lookup constraints from the source circuit carry over into the resulting circuit. The number of rows stays the same, while the polynomial degree bound increases. That sums up the construction.

Let

$$\mathcal{C} := (\mathbb{Z}, (c_1, ..., c_n), d, (g_1, ..., g_m), (\ell_1, ..., \ell_k), r, (e_1, ..., e_t), X) \tag{113}$$

be a logic circuit, the source circuit.

Choose a word size W. The resulting arithmetic circuit will use $(W+1)$-bit signed words (with a W-bit unsigned part and a sign bit). Choose a byte size B. The resulting arithmetic circuit will use B-bit bytes. Let $N = W/B$. The resulting arithmetic circuit will use W-bit words.

The word size W needs to be chosen to be large enough that it provides enough headroom to hold all possible values of $\mu - \tau$ for all subformulas of the form $\tau = \mu$ or $\tau < \mu$ in all gate constraints of the source circuit, when the instance and advice values from the source circuit are small enough. What does small enough mean? For instance values, it means within bounds that define which instances we need semantics preservation for. For advice values, it means within some bounds that are large enough to guarantee the existence of in-bounds satifying advice values for all satisfiable instances.

Choose bounds $B_I : [v] \to \mathbb{N}^2$, where v is the number of instance columns. These bounds define which instances we require semantics preservation for. Choose bounds $B_A : [w] \to \mathbb{N}^2$, where w is the number of advice columns. Choose B_A such that, for all instances $Z \in \mathbb{Z}^{[v] \times [r]}$, if for all $i \in [v], j \in [r]$, $Z_{i,j} \in (-\pi_1(B_I(i)), \pi_2(B_I(i)))$, and Z is in the denotation of $\mathcal{C}$, then there is some $Y \in \mathbb{Z}^{[n] \times [r]}$ witnessing this fact, such that for all $i \in [w]$, letting j be the index of the ith advice column, for all $k \in [r]$, $Y_{j,k} \in (-\pi_1(B_A(i)), \pi_2(B_A(i)))$.

Choose a finite field $\mathbb{F}$. Choose $\mathbb{F}$ such that $|\mathbb{F}| > 2^{W+1}$, so that each $(W+1)$-bit word has a distinct representation. Also choose $\mathbb{F}$ so that $|\mathbb{F}|$ has enough headroom to compute all intermediate results in gate constraints without arithmetic overflow.

Create a total ordering of the subformulas $\tau = \mu$ or $\tau < \mu$ in the gate constraints $\vec{g}$, so that the polynomials of each such subformula are assigned indices, e.g. $\tau_i = \mu_i$ or $\tau_i < \mu_i$. Let s be the total number of such subformulas. Therefore we have two sequences of formulas, $\vec{\tau}$ and $\vec{\mu}$, each of length s.

Define the resulting circuit $\mathcal{D}$ as follows.

1. The domain of $\mathcal{D}$ is $\mathbb{F}$.
2. The number of rows of $\mathcal{D}$ is r (same as $\mathcal{C}$).
3. The number of instance columns of $\mathcal{D}$ is v (same as $\mathcal{C}$).
4. The fixed columns of $\mathcal{D}$ are those in $\mathcal{C}$ (values given by X), plus the fixed columns required for the byte decomposition range checks and equality statement truth tables:
   (a) $\vec{\beta}_0 := 0, 1, 2, ..., 2^B - 1, 2^B - 1, 2^B - 1, ...$
   This column enumerates all the numbers 0 through $2^B - 1$, and then repeats $2^B - 1$, $r - 2^B$ times.
   (b) $\vec{\beta}_1 := 1, 0, 0, 0, ...$
   This column has a one in row zero and a zero in all other rows.
5. The advice columns of $\mathcal{D}$ are those in $\mathcal{C}$, plus, for each subformula $\tau_i = \mu_i$ or $\tau_i < \mu_i$ in a gate constraint in $\mathcal{C}$:
   (a) $N + 1$ advice columns, $\vec{\sigma}_i, \vec{\delta}_{i,0}, ..., \vec{\delta}_{i,N-1}$. These are supposed to hold the byte decomposition of the value of $\mu_i - \tau_i$ at each row. $\vec{\sigma}_i$ holds the sign bit (-1 or 1), and $\vec{\delta}_{i,j}$ holds the jth byte of the unsigned part (in big endian notation).
   (b) N advice columns, $\vec{\gamma}_{i,0}, ..., \vec{\gamma}_{i,N-1}$. These are supposed to be set as follows, for all $j \in [N], k \in [r]$:
   $$\gamma_{i,j,k} = \begin{cases} 1 & \vec{\delta}_{i,j,k} = 0, \\ 0 & \text{otherwise.} \end{cases} \tag{114}$$
6. The gate constraints are as follows:
   (a) For all $i \in [s]$,
   $$(\mu_i - \tau_i) = \vec{\sigma}_i \cdot \sum_{j=0}^{N} 2^j \cdot \vec{\delta}_{i,N-1-j}. \tag{115}$$
   Here $\mu_i - \tau_i$ is interpreted as a polynomial where the variables denote column vectors: the column vectors referred to by the variable names, rotated by the row offsets in the variable names.
   This constraint checks the correctness of the byte decompositions.
   (b) For all $i \in [m]$,
   $$\text{eval}(g_i) = \vec{1}, \tag{116}$$
   where eval is defined for all subformulas of the source gate constraints $\vec{g}$, by the following recursive clauses:
      i. $\text{eval}(\tau_i = \mu_i) := (\frac{1}{2} \cdot (\vec{\sigma}_i + 1)) \cdot \prod_{k \in [N]} \vec{\gamma_{i,k}}$.
      ii. $\text{eval}(\tau_i < \mu_i) := (\frac{1}{2} \cdot (\vec{\sigma}_i + 1)) \cdot \text{some}(\vec{\gamma}_{i,0}, ..., \vec{\gamma}_{i,N-1})$.
      iii. $\text{eval}(\neg\phi) = 1 - \text{eval}(\phi)$.
      iv. $\text{eval}(\phi \wedge \psi) := \text{eval}(\phi) \cdot \text{eval}(\psi)$.
      v. $\text{eval}(\phi \vee \psi) := \text{eval}(\phi) + \text{eval}(\psi) - (\text{eval}(\phi) \cdot \text{eval}(\psi))$.
      vi. $\text{eval}(\phi \to \psi) := \text{eval}(\neg\phi \vee \psi)$.

The function some, which takes any number of column vectors and returns a polynomial, is defined on arguments by recursion on the number of arguments:

$$\text{some}(\vec{x}) := \vec{x}. \tag{117}$$

$$\text{some}(\vec{x}, \vec{y}) := \vec{x} + \text{some}(\vec{y}) - (\vec{x} \cdot \text{some}(\vec{y})). \tag{118}$$

7. The lookup constraints are $\vec{\ell}$ (those of $\mathcal{C}$), plus the following additional lookup constraints. For all $i \in [s], j \in [r]$:

   (a) For all $i \in [s], j \in [r]$,

   $$\frac{1}{2}(\sigma_{i,j} + 1) \in \beta_1. \tag{119}$$

   This ensures that $\sigma_{i,j} \in \{-1, 1\}$.

   (b) For all $i \in [s], j \in [N], k \in [r]$,

   $$(\delta_{i,j,k}, \gamma_{i,j,k}) \in (\beta_0, \beta_1). \tag{120}$$

   This is the equality statement truth table check and the byte range check, combined into one constraint.

8. The equality constraints of $\mathcal{D}$ are $\vec{e}$ (same as $\mathcal{C}$).

References

[1] The Electric Coin Company. *The halo2 Book.* 2021. `https://zcash.github.io/halo2/index.html`

[2] The Electric Coin Company. *halo2.* 2022. `https://github.com/zcash/halo2`

[3] Sean Bowe, Jack Grigg, and Daira Hopwood. *Recursive Proof Composition without a Trusted Setup.* IACR Cryptol. ePrint Arch., 2019, #1021. `https://eprint.iacr.org/2019/1021`

[4] Morgan Thomas (Orbis Labs). *Arithmetization of Σ_1^1 relations in Halo 2.* IACR Cryptol. ePrint Arch., 2022, #777. `https://eprint.iacr.org/2022/777`

[5] Orbis Labs. *Coq-Arithmetization.* GitHub. `https://github.com/Orbis-Tertius/coq-arithmetization`

[6] Morgan Thomas (Orbis Labs). *Orbis Specification Language: a type theory for zk-SNARK programming.* IACR Cryptol. ePrint Arch., 2022, #1003. `https://eprint.iacr.org/2022/1003`