Circular lists in Iris $*$ deduction rules of $\triangleright$

Herman Bergwerf

March 11, 2022

Introduction

One concern of theoretical computer science is to prove the correctness of algorithms, for example the implementation of datastructures that are at the core of almost all programs. This becomes especially complicated when concurrency is involved in a non-trivial way, such that multiple parts of memory that logically belong together are modified in parallel.

Proof Assistants are computer programs that enable logical reasoning in a strict formal proof system. Developing proofs inside a such a proof assistant requires extreme rigour. As long as your definitions are not flawed, it is practically impossible to miss edge cases. The proof assistant we use is called Coq [18].

Coq is not readily usable to reason about programs that use mutable resources, but the *Iris project* has built a framework inside Coq that offers a convenient language and toolbox for this [19]. Iris uses separation logic and modal logic to reason about memory, concurrency, and potentially non-terminating recursion.

Contributions:

1. **Circular lists in Iris** We formalize a circular doubly linked list in Coq/Iris using separation logic. We only verified synchronous list operations. This chapter demonstrates the use of separation logic to reason about locations stored on a heap. To our knowledge this is the first interactive verification of a circular list using separation logic.

2. **Deduction rules of $\triangleright$** We study the model of plain step-indexed propositions without resources, a basic interpretation of the $\triangleright$ modality. We determine a complete set of deductions, and formalize an algorithm that checks a finite number of potential counterexamples in Coq, proving decidability and completeness.

Chapter 1

Circular lists in Iris

A circular doubly linked list is a linked list where each node has a pointer to the previous and next node. The first and last node are also connected as shown schematically in Figure 1.1.

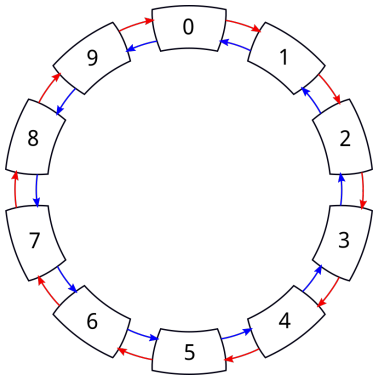

Figure 1.1: The conceptual layout of a circular list with ten nodes. The red arrows represent pointers to the next node, and the blue arrows to the previous node.

Our goal is to implement the algorithms to create and modify such a list, and prove that they are correct. This is also called *verification*. When is an algorithm *correct*? This depends on the context, and the requirements it needs to satisfy. We will define this using pre- and postconditions describing the memory structure, and show that the algorithms follow these definitions. More specifically we prove *total correctness*, meaning that each verified function, given an input satisfying the precondition, terminates without crashing with an output satisfying the postcondition. A stronger sense of validity can be achieved when considering asynchronous execution, for example parallel inserts at different points in the list. Here we only consider synchronous execution.

1.1 Separation logic

In 1969 Hoare introduced a logic to reason about computer programs [4]. This logic uses so called *Hoare triples* to specify program behavior. A Hoare triple contains a *precondition* P, a program (or *command*) C, and a *postcondition* Q, and is written as $\{P\}C\{Q\}$. The pre- and postcondition are formulae specifying properties of the state before and after the program is executed.

To reason about programs that use mutable memory locations, the *points to* predicate was later added. A points to predicate is written as $l \mapsto v$, and asserts that *location* l points to value v.

When verifying even slightly complex programs, overlap between locations becomes a critical issue. In the case of our list, it will be essential to require that each node is stored at a separate location in memory. Instead of writing this down in the formula explicitly, the *separating conjunction* was introduced to indicate that two formulae hold with respect to two separate parts of the heap, giving rise to separation logic [13]. The separating conjunction of two formulas P and Q is written as $P * Q$. Here the points to predicates in P are disjoint from those in Q; both formulas describe a separate part of the heap. Writing $P * Q$ is similar to saying "the heap can be split into two parts, such that P holds for one part, and Q for the other part".

1.2 Working with Iris

Iris is a framework to carry out program verification. It is built for the Coq Proof Assistant, and requires no additional plugins. The design of Iris has multiple layers of abstraction, the top-level layer implements a small language called HeapLang.

HeapLang HeapLang is a small untyped language. It supports the following values: a unit element, integers, memory locations, functions, products (*e.g.* pairs), and sums. The language contains the following constructs: recursive lambda expressions, function application, if statements, basic integer operations, and heap modification (allocating, storing, loading, and freeing). There are a few other constructs, for example for concurrency, but these are not relevant to this chapter.

Hoare triples Iris allows writing program specifications using Hoare triples with a notation that roughly looks as follows:

```
[[{ P }]] program [[{ x, RET x; Q x }]]
```

Here `P` and `Q x` are Iris predicates with type `iProp`, `program` is a HeapLang expression, and `x` is variable representing a HeapLang value. The `RET x` captures the value that is returned by the program. The above expression states that if the input satisfies `P`, then the program will terminate with output `x` satisfying `Q x`.

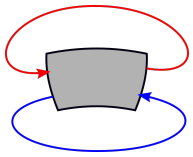

Figure 1.2: A circular list containing only a single dummy node. This is the result of `make`.

Verification To verify the basic circular list operations, we will implement them in HeapLang (Section 1.3) and define suitable predicates describing the structure of the circular list (Section 1.4). We then formulate pre- and postconditions that describe the behavior of each list operation and prove that these are satisfied with an interactive Coq proof (Section 1.5).

1.3 Implementation

In this section we look at the HeapLang implementation of basic circular list operations. Nodes are encoded as a tuple of the form (p, n, v) where p and n are the location of the previous and next node, respectively, and v is the value of the node. We store values either as `NONE` or `SOME v` to support dummy nodes holding no value. These dummy nodes cannot be deleted or modified, and will be used to represent empty lists. When a new list is created it will consist of a single dummy node, as shown in Figure 1.2.

Empty list A new list contains one dummy node pointing to itself.

```
Definition make : val :=
  λ: <>,
    let: "node" := ref NONE in
    "node" ← (("node", "node"), NONE);;
    "node".
```

Getters and setters We implement basic getters and setters for convenience.

```
Notation prev' node := (Fst (Fst node)).
Notation next' node := (Snd (Fst node)).
Notation value' node := (Snd node).

Definition get_prev : val := λ: "node", prev' !"node".
Definition get_next : val := λ: "node", next' !"node".

Definition set_prev : val :=
  λ: "node" "prev",
    let: "node_v" := !"node" in
    "node" ← ("prev", next' "node_v", value' "node_v").

Definition set_next : val :=
  λ: "node" "next",
    let: "node_v" := !"node" in
    "node" ← (prev' "node_v", "next", value' "node_v").
```

Insert a node Insert a new node after the `"prev"` node with `"v"` as value.

```
Definition insert : val :=
  λ: "prev" "v",
    let: "next" := get_next "prev" in
    let: "node" := ref ("prev", "next", SOME "v") in
    set_next "prev" "node";;
    set_prev "next" "node";;
    "node".
```

Delete a node Delete a given `"node"`. Dummy nodes cannot be deleted.

```
Definition delete : val :=
  λ: "node",
    let: "node_v" := !"node" in
    match: value' "node_v" with
      NONE ⇒ NONE |
      SOME "value" ⇒
        let: "prev" := prev' "node_v" in
        let: "next" := next' "node_v" in
        set_next "prev" "next";;
        set_prev "next" "prev";;
        Free "node";;
        SOME "value"
    end.
```

Deque operations Deque is an abbreviation for "double-ended queue". A double ended queue supports push and pop operations and both the front and the back. Figure 1.3 illustrates the structure of our deque.

```
Definition push_front : val := λ: "dq" "v", insert "dq" "v";; #().
Definition push_back  : val := λ: "dq" "v", insert (get_prev "dq") "v";; #().
Definition pop_front  : val := λ: "dq", delete (get_next "dq").
Definition pop_back   : val := λ: "dq", delete (get_prev "dq").
```

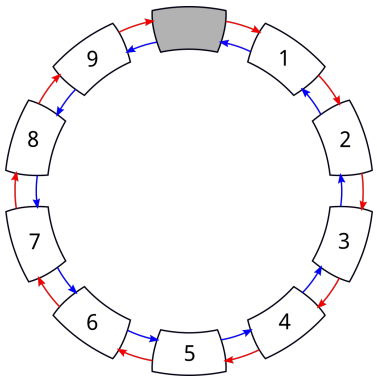

Figure 1.3: A circular list with one dummy node in gray. The `push_back` and `push_front` operation insert nodes before and after the dummy node.

1.4 List predicates

The list predicates that we define relate a standard Coq list to a correctly laid out circular list on the heap. This relation will allow us to specify the list operations in terms of modifications to the Coq list.

We define three predicates; `dseg` describes an arbitrary list segment given a Coq list of locations and associated values, `dlist` describes a full circular list segment given a Coq list of locations and values, and `deque` describes a circular list using a Coq list of just values and the location of the initial dummy node. The predicates build on top of each other.

It turned out useful to use as little existential quantification as possible. Instead of asserting the existence of node locations in `dseg` they are given as input. This way we can carry out case analysis in Coq by destructing the input list, instead of destructing Iris hypotheses.

A list segment Segments are specified by a Coq list of locations with associated values. The function `convert_val` maps values of the type `option val` to the corresponding HeapLang value (options are implemented using HeapLang sum values). The `dseg` predicate states that the given `nodes : list (loc * option val)` form a segment of doubly linked nodes. The location of the node before the first node and after the last node is given by `prev after : loc`.

```
Fixpoint dseg (prev after : loc) (nodes : list (loc * option val)) : iProp :=
  match nodes with
  | [] ⇒ True
  | [(l, v)] ⇒ l ↦ (#prev, #after, convert_val v)
  | (l, v) :: ((next, _) :: _) as nodes' ⇒
    l ↦ (#prev, #next, convert_val v) ∗
    dseg l after nodes'
  end.
```

A circular list A circular list consists of a single segment where the first and last nodes are joined together. The function `dlast` retrieves the location of the last node. The location of the first node, `n.1`, is provided as a default argument.

```
Notation dlast l nodes := (last l (map fst nodes)).

Definition dlist (nodes : list (loc * option val)) : iProp :=
  match nodes with
  | [] ⇒ True
  | n :: _ ⇒ dseg (dlast n.1 nodes) n.1 nodes
  end.
```

A value list For the final deque specification we hide the node locations. The `deque` predicate only requires the location of the dummy node, `l : loc`, and the values that are in the list, `vs : list val`. For the predicate to hold there must be a list of node locations to which those values belong.

```
Definition deque (l : loc) (vs : list val) : iProp :=
  ∃ ls, ⌜length ls = length vs⌝ ∗
    dlist ((l, None) :: zip ls (map Some vs)).
```

1.5 Specifications and proofs

Splitting and gluing segments It is useful to split and glue segments satisfying the `dseg` predicate. We prove that a bigger segment can be split into two adjacent segments, and that two adjacent segments can again be combined into a single segment.

```
Lemma dseg_split lA lB vB lC vC lD ns1 ns2 :
  dseg lA lD (ns1 ++ (lB, vB) :: (lC, vC) :: ns2) -∗
  dseg lA lC (ns1 ++ [(lB, vB)]) ∗ dseg lB lD ((lC, vC) :: ns2).

Lemma dseg_glue lA lB vB lC vC lD ns1 ns2 :
  dseg lA lC (ns1 ++ [(lB, vB)]) ∗ dseg lB lD ((lC, vC) :: ns2) -∗
  dseg lA lD (ns1 ++ (lB, vB) :: (lC, vC) :: ns2).
```

Rotating a circular list With the help of these lemmas we can prove a very useful property of `dlist`. We can move elements at the front of the list to the back, and vice versa.

```
Lemma dlist_step n ns :
  dlist (n :: ns) ⊣⊢ dlist (ns ++ [n]).
```

List operations The specification of list creation, insertion, and deletion uses the `dlist` predicate. The `delete_none_spec` states that `delete` does not delete dummy nodes.

```
Lemma make_spec :
  [[{ True }]]
    make #()
  [[{ l, RET #l; dlist [(l, None)] }]].

Lemma insert_spec l0 v0 v ns :
  [[{ dlist ((l0, v0) :: ns) }]]
    insert #l0 v
  [[{ l, RET #l; dlist ((l0, v0) :: (l, Some v) :: ns) }]].

Lemma delete_none_spec l ns :
  [[{ dlist ((l, None) :: ns) }]]
    delete #l
  [[{ RET NONEV; dlist ((l, None) :: ns) }]].

Lemma delete_some_spec l v ns :
  [[{ dlist ((l, Some v) :: ns) }]]
    delete #l
  [[{ RET (SOMEV v); dlist ns }]].
```

Our proof of `insert_spec` uses a distinction between the initial list containing just one node pointing to itself, and a list containing multiple nodes. Our proof of `delete_some_spec` uses a distinction between the initial list containing just one node (destroying the list), containing two nodes (leaving one node connected to itself), and containing three or more nodes.

Deque operations The deque operations push/pop front/back are specified using the `deque` predicate, because the underlying node locations are not important. The `make_deque_spec` is a version of `make_spec` using the `deque` predicate.

```
Lemma make_deque_spec :
  [[{ True }]]
    make #()
  [[{ l, RET #l; deque l [] }]].

Lemma push_front_spec l v vs :
  [[{ deque l vs }]]
    push_front #l v
  [[{ RET #(); deque l (v :: vs) }]].

Lemma pop_front_spec l v vs :
  [[{ deque l (v :: vs) }]]
    pop_front #l
  [[{ RET (SOMEV v); deque l vs }]].

Lemma push_back_spec l v vs :
  [[{ deque l vs }]]
    push_back #l v
  [[{ RET #(); deque l (vs ++ [v]) }]].

Lemma pop_back_spec l v vs :
  [[{ deque l (vs ++ [v]) }]]
    pop_back #l
  [[{ RET (SOMEV v); deque l vs }]].
```

These specifications use the `dlist_step` lemma to "rotate" the underlying `dlist` predicate, and then apply the specifications of the list operations. Our proof of `push_back_spec` also uses a distinction between an empty and a non-empty list of values.

1.6 Related work

Iris uses the logic of bunched implications, which was proposed over two decades ago by O'Hearn and Pym [9]. The usage of list predicates such as our `dlist` is standard. Such a predicate for ordinary doubly linked lists was already given in Reynold's seminal paper on separation logic from 2002 [13]. Various (early) works on separation logic investigate the verification of a linked list (for example [11] section 6). To our knowledge this work is the first formal verification of a circular doubly linked list using separation logic in a proof assistant.

A paper from 2008 describes the automatic verification of a range of linked datastructures that are implemented in Java, including a circular list [15][16]. Their specifications are written in a special format inside comments in the Java code[1]. For the circular list their program relied mainly on MONA [8], a tool implementing a decision procedure for monadic second-order logic over strings and trees. This procedure is based on automata.

[1] The source of their circular list can be found here: https://github.com/epfl-lara/jahob/blob/master/examples/datastructures/encap/CircularList/CircularList.java

Chapter 2

Deduction rules of ▷

Recursion is a fundamental technique to express algorithms. In order to compute a result such an algorithm executes itself on a subproblem. These algorithms may not always be terminating. Sometimes infinite loops are used intentionally, for example to put a concurrent program in a waiting state. To prove that a recursive algorithm satisfies a postcondition we use induction: Given that the results of all subcomputations satisfy the condition, we show that the result of the whole computation satisfies the condition.

Induction normally uses some kind of decreasing measure. A straightforward choice is the number of computation steps. To avoid the need to calculate an upper bound on the number of computation steps needed by the algorithm, and to reason about potentially non-terminating algorithms, we prove that a program satisfies the desired postcondition once it terminates, and that it doesn't crash, for any fixed number of *available* computation steps.

The later modality, written as ▷, serves as a syntactic tool to hide the computation step index. Instead of explicitly doing induction on the number of allowed steps, we can use an induction principle for this modality which is often referred to as *Löb induction*[1]. Instead of an induction hypothesis that guarantees the postcondition for recursive calls with a smaller number of computation steps, it will give the postcondition with an added ▷. This modal operator can only be removed by applying computation steps, such that in effect we can only apply the induction hypothesis to recursive calls that use less computation steps. The later modality hides the step indexing.

A more comprehensive introduction to the later modality can be found in [14]. It plays an important role in Iris [19]. In the rest of this chapter we are not concerned with program verification, but with the theory of the later modality.

[1] Löb induction is named after the German mathematician Martin Hugo Löb. Löb proved a theorem about provability in Peano Arithmetic that resembles this principle.

Contribution We study the model of step-indexed propositions, which provides a basic interpretation of propositional logic with the later modality. We introduce this model formally in Section 2.2. In Section 2.3 we introduce standard deduction rules that also hold in this model. Then we wonder; is this set of rules complete? Can we derive every formula that is a tautology in the step-indexing model? We show that the model is essentially a subset of linear integer arithmetic in Section 2.4. In Section 2.5 we introduce the *comparison rule*, which completes the deduction system. We will explain the intuition behind the completeness proof in Section 2.6, and discuss the Coq formalization of this proof in Section 2.7. In Section 2.8 we briely reflect on these results.

2.1 Definitions

Formulas Let $\Sigma = \{P_0, P_1, ...\}$ be a set of proposition letters. The language of formulas is defined by the following grammar:

$$\varphi_0, \varphi_1 \in \mathcal{L}_\triangleright ::= \top \mid \bot \mid P_i \in \Sigma \mid \varphi_0 \wedge \varphi_1 \mid \varphi_0 \vee \varphi_1 \mid \varphi_0 \Rightarrow \varphi_1 \mid \triangleright\varphi_0$$

Remarks:

- We sometimes use the term *variables* to refer to proposition letters, and *variable index* to refer to the index of proposition letters. For example the index of P_3 is 3 (we assume Σ is denumerable).
- Conjunction ($\wedge$) has a lower precedence than the other operators, such that $\varphi_0 \Rightarrow \varphi_1 \wedge \varphi_0 \vee \varphi_1$ is equivalent to $(\varphi_0 \Rightarrow \varphi_1) \wedge (\varphi_0 \vee \varphi_1)$.
- We will use $\varphi_0 \Leftrightarrow \varphi_1$ as an abbreviation of $\varphi_0 \Rightarrow \varphi_1 \wedge \varphi_1 \Rightarrow \varphi_0$.
- We will use $\triangleright^n$ as an abbreviation of n successive $\triangleright$'s.

Models A model $\mathfrak{A}$ must define:

- A domain $\dot{\mathfrak{A}}$ (denoted with a dot)
- A binary relation $\sqsubseteq_{\mathfrak{A}} \subseteq \dot{\mathfrak{A}} \times \dot{\mathfrak{A}}$
- A denotation $\mathfrak{A}[\![\varphi]\!](\Gamma) \in \dot{\mathfrak{A}}$, where $\varphi \in \mathcal{L}_\triangleright$ and $\Gamma : \Sigma \to \dot{\mathfrak{A}}$

A *valuation* $\Gamma : \Sigma \to \dot{\mathfrak{A}}$ assigns a value to each proposition letter. Let $\varphi_0, \varphi_1 \in \mathcal{L}_\triangleright$ be formulas, we say that φ_1 is a consequence of φ_0 in $\mathfrak{A}$ if for all valuations Γ we have $\mathfrak{A}[\![\varphi_0]\!](\Gamma) \sqsubseteq_{\mathfrak{A}} \mathfrak{A}[\![\varphi_1]\!](\Gamma)$. We will write this as $\varphi_0 \models_{\mathfrak{A}} \varphi_1$:

$$\varphi_0 \models_{\mathfrak{A}} \varphi_1 := \forall \Gamma.\ \mathfrak{A}[\![\varphi_0]\!](\Gamma) \sqsubseteq_{\mathfrak{A}} \mathfrak{A}[\![\varphi_1]\!](\Gamma)$$

We will sometimes loosely refer to this as a tautology.

2.2 The step-indexing model

We now define the step-indexing model $\mathfrak{B}$. This model is similar to the model of Iris[2] (see Figure 8 and 10 of [19]) omitting resources. The model of Iris is based on other research, which we briefly discuss in Section 2.9.

Elements of the step-indexing model are binary sequences that are downwards closed. This means that every α in the domain satisfies the formula $\forall i \forall j \leq i.\ \alpha(i) \to \alpha(j)$, *i.e.* a 0 cannot occur before a 1. We now define $\mathfrak{B}$.

$$
\begin{aligned}
\dot{\mathfrak{B}} &:= \{\alpha : \mathbb{N} \to \{0,1\} \mid \forall i \forall j \leq i.\ \alpha(i) \to \alpha(j)\} \\
\alpha \sqsubseteq_{\mathfrak{B}} \beta &:= \forall i.\ \alpha(i) \to \beta(i) \\
\mathfrak{B}[\![P_i]\!](\Gamma) &:= \Gamma(P_i) \\
\mathfrak{B}[\![\bot]\!](\Gamma) &:= \lambda i.\ 0 \\
\mathfrak{B}[\![\top]\!](\Gamma) &:= \lambda i.\ 1 \\
\mathfrak{B}[\![\varphi_0 \wedge \varphi_1]\!](\Gamma) &:= \lambda i.\ \mathfrak{B}[\![\varphi_0]\!](\Gamma)(i) \wedge \mathfrak{B}[\![\varphi_1]\!](\Gamma)(i) \\
\mathfrak{B}[\![\varphi_0 \vee \varphi_1]\!](\Gamma) &:= \lambda i.\ \mathfrak{B}[\![\varphi_0]\!](\Gamma)(i) \vee \mathfrak{B}[\![\varphi_1]\!](\Gamma)(i) \\
\mathfrak{B}[\![\varphi_0 \Rightarrow \varphi_1]\!](\Gamma) &:= \lambda i.\ \forall j \leq i.\ \mathfrak{B}[\![\varphi_0]\!](\Gamma)(j) \to \mathfrak{B}[\![\varphi_1]\!](\Gamma)(j) \\
\mathfrak{B}[\![\triangleright \varphi]\!](\Gamma) &:= \lambda i.\ \textbf{if } i = 0 \textbf{ then } 1 \textbf{ else } \mathfrak{B}[\![\varphi]\!](\Gamma)(i-1)
\end{aligned}
$$

Downward closed Why are sequences downwards closed? We can make sense out of this in the context of program verification and computation steps. Suppose a program is valid (does not crash and, upon termination, satisfies some condition) for n steps, then it is also valid for less than n steps.

2.3 The deduction system

In the previous section we saw a model for our logic, which gives it a semantic meaning. We now define a syntactic meaning by defining a deduction system to derive entailments using deduction trees. We use $\vdash$ to denote a syntactic entailment between two formulas. Writing $\varphi_0 \vdash \varphi_1$ means that a closed (*i.e.* without hypotheses) deduction tree exists with this entailment as conclusion. We also call this a derivation of $\varphi_0 \vdash \varphi_1$.

Figure 2.1 introduces the set of standard[3] deduction rules. Rules for the standard logical connectives are equivalent to intuitionistic propositional logic. Löb induction is given as the $\triangleright$-Löb rule.

In our deduction system the context (the formula left of the $\vdash$) is a single formula instead of a list or set. The minimal formulation of rules like $\wedge$-elim-l requires that transitivity of $\vdash$ is included as a basic rule; it is not possible to derive a deduction theorem. We used this approach to simplify our formalization.

[2] The Iris repository also contains a formalization of "plain" step-indexed propositions, see: https://gitlab.mpi-sws.org/iris/iris/-/tree/master/iris/si_logic.

[3] These rules are very similar to the Gödel-Löb logic given by Appel et al. (Figure 4 of [14]). Variants appear throughout the literature.

$$\frac{}{\varphi \vdash \varphi}\ \text{refl} \qquad \frac{\varphi_0 \vdash \varphi_1 \quad \varphi_1 \vdash \varphi_2}{\varphi_0 \vdash \varphi_2}\ \text{trans}$$

$$\frac{}{\varphi \vdash \top}\ \top\text{-intro} \qquad \frac{}{\bot \vdash \varphi}\ \bot\text{-elim}$$

$$\frac{\sigma \vdash \varphi_0 \quad \sigma \vdash \varphi_1}{\sigma \vdash \varphi_0 \wedge \varphi_1}\ \wedge\text{-intro} \qquad \frac{\varphi_0 \vdash \varphi_2 \quad \varphi_1 \vdash \varphi_2}{\varphi_0 \vee \varphi_1 \vdash \varphi_2}\ \vee\text{-elim}$$

$$\frac{}{\varphi_0 \wedge \varphi_1 \vdash \varphi_0}\ \wedge\text{-elim-l} \qquad \frac{}{\varphi_0 \vdash \varphi_0 \vee \varphi_1}\ \vee\text{-intro-l}$$

$$\frac{}{\varphi_0 \wedge \varphi_1 \vdash \varphi_1}\ \wedge\text{-elim-r} \qquad \frac{}{\varphi_1 \vdash \varphi_0 \vee \varphi_1}\ \vee\text{-intro-r}$$

$$\frac{\sigma \wedge \varphi_0 \vdash \varphi_1}{\sigma \vdash \varphi_0 \Rightarrow \varphi_1}\ \Rightarrow\text{-intro} \qquad \frac{\sigma \vdash \varphi_0 \Rightarrow \varphi_1 \quad \sigma \vdash \varphi_0}{\sigma \vdash \varphi_1}\ \Rightarrow\text{-elim}$$

$$\frac{}{\varphi_0 \vdash \triangleright\varphi_0}\ \triangleright\text{-intro} \qquad \frac{}{\triangleright\varphi_0 \wedge \triangleright\varphi_1 \vdash \triangleright(\varphi_0 \wedge \varphi_1)}\ \triangleright\text{-conj}$$

$$\frac{\top \vdash \triangleright\varphi}{\top \vdash \varphi}\ \triangleright\text{-elim} \qquad \frac{\varphi_0 \vdash \varphi_1}{\triangleright\varphi_0 \vdash \triangleright\varphi_1}\ \triangleright\text{-mono} \qquad \frac{\triangleright\varphi \vdash \varphi}{\top \vdash \varphi}\ \triangleright\text{-Löb}$$

Figure 2.1: Standard deduction rules.

Example Below is an example deduction tree to illustrate the way these rules can be composed. The left branch is not completely spelled out, but is trivial to complete. A derivation of the strong-Löb rule is given in Appendix A.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\ldots \vdash \varphi_0 \Rightarrow \varphi_1} \quad \dfrac{\dfrac{}{\ldots \vdash \triangleright\varphi_1 \Rightarrow \varphi_0} \quad \dfrac{}{\ldots \vdash \triangleright\varphi_1}}{\ldots \vdash \varphi_0}\ \Rightarrow\text{-elim}}{\triangleright\varphi_1 \Rightarrow \varphi_0 \wedge \varphi_0 \Rightarrow \varphi_1 \wedge \triangleright\varphi_1 \vdash \varphi_1}\ \Rightarrow\text{-elim}}{\triangleright\varphi_1 \Rightarrow \varphi_0 \wedge \varphi_0 \Rightarrow \varphi_1 \vdash \triangleright\varphi_1 \Rightarrow \varphi_1}\ \Rightarrow\text{-intro} \qquad \dfrac{}{\triangleright\varphi_1 \Rightarrow \varphi_1 \vdash \varphi_1}\ \text{strong-Löb}}{\triangleright\varphi_1 \Rightarrow \varphi_0 \wedge \varphi_0 \Rightarrow \varphi_1 \vdash \varphi_1}\ \text{trans}$$
$$\dfrac{\triangleright\varphi_1 \Rightarrow \varphi_0 \wedge \varphi_0 \Rightarrow \varphi_1 \vdash \varphi_1}{\triangleright\varphi_1 \Rightarrow \varphi_0 \vdash (\varphi_0 \Rightarrow \varphi_1) \Rightarrow \varphi_1}\ \Rightarrow\text{-intro}$$

Soundness The model $\mathfrak{B}$ from the previous section realizes all deductions. The proof proceeds by induction on the deduction tree.

Theorem 1. *(soundness of $\mathfrak{B}$) If $\varphi_0 \vdash \varphi_1$, then $\varphi_0 \models_{\mathfrak{B}} \varphi_1$.*

2.4 The arithmetic model

We strip the step-indexing model $\mathfrak{B}$ down to its essence; the number of 1's in the sequence. Every sequence contains either a finite number of 1's followed by only 0's, or it contains an infinite number of 1's. Hence the domain of our new model is $\mathbb{N} \cup \{\omega\}$, where ω denotes infinity. We define a model that is isomorphic[4] to $\mathfrak{B}$, by mapping every sequence to its prefix length:

$$\alpha \in \dot{\mathfrak{B}} \mapsto \begin{cases} n & \alpha = 1^n 0^\omega \\ \omega & \alpha = 1^\omega \end{cases}$$

Below is a precise definition of $\mathfrak{N}$. The definition of $\leq$, min, max is extended to include ω in the obvious way.

$$\begin{aligned}
\dot{\mathfrak{N}} &:= \mathbb{N} \cup \{\omega\} \\
p \sqsubseteq_{\mathfrak{N}} q &:= p \leq q \\
\mathfrak{N}[\![P_i]\!](\Gamma) &:= \Gamma(P_i) \\
\mathfrak{N}[\![\bot]\!](\Gamma) &:= 0 \\
\mathfrak{N}[\![\top]\!](\Gamma) &:= \omega \\
\mathfrak{N}[\![\varphi_0 \wedge \varphi_1]\!](\Gamma) &:= \min\{\mathfrak{N}[\![\varphi_0]\!](\Gamma),\ \mathfrak{N}[\![\varphi_1]\!](\Gamma)\} \\
\mathfrak{N}[\![\varphi_0 \vee \varphi_1]\!](\Gamma) &:= \max\{\mathfrak{N}[\![\varphi_0]\!](\Gamma),\ \mathfrak{N}[\![\varphi_1]\!](\Gamma)\} \\
\mathfrak{N}[\![\varphi_0 \Rightarrow \varphi_1]\!](\Gamma) &:= \textbf{if } \mathfrak{N}[\![\varphi_0]\!](\Gamma) \leq \mathfrak{N}[\![\varphi_1]\!](\Gamma) \textbf{ then } \omega \textbf{ else } \mathfrak{N}[\![\varphi_1]\!](\Gamma) \\
\mathfrak{N}[\![\triangleright\varphi]\!](\Gamma) &:= \textbf{if } \mathfrak{N}[\![\varphi]\!](\Gamma) = n \in \mathbb{N} \textbf{ then } n+1 \textbf{ else } \omega
\end{aligned}$$

Note that given a formula φ, we have $\top \models_{\mathfrak{N}} \varphi$ when[5] $\mathfrak{N}[\![\varphi]\!](\Gamma) = \omega$ for every valuation Γ. Also note that this model is indeed ismorphic to $\mathfrak{B}$:

Theorem 2. *$\varphi_0 \models_{\mathfrak{B}} \varphi_1$ if and only if $\varphi_0 \models_{\mathfrak{N}} \varphi_1$*

Proof. Let $f : \dot{\mathfrak{B}} \to \dot{\mathfrak{N}}$ be the isomorphism given at the start of this section.
Show $\forall \alpha, \beta \in \dot{\mathfrak{B}}.\ \alpha \sqsubseteq_{\mathfrak{B}} \beta \leftrightarrow f(\alpha) \sqsubseteq_{\mathfrak{N}} f(\beta)$.
Show $\forall \varphi \in \mathcal{L}_\triangleright.\ f(\mathfrak{B}[\![\varphi]\!](\Gamma)) = \mathfrak{N}[\![\varphi]\!](\Gamma)$. □

With this model we have translated our logic to linear integer arithmetic. It is known that linear integer arithmetic is decidable using quantifier elimination (first demonstrated by Presburger [1] and later improved by Cooper [5]), or using finite automata [2][7].

[4] This isomorphism is non-constructive since it needs to distinguish between sequences that only contain 1's and sequences that contain a 0. This classical principle is known as LPO (see page 21 of [20]).

[5] Can you see why the law of excluded middle, $P \vee \neg P$, is not a tautology of $\mathfrak{N}$?

2.5 The comparison rule

Since the models $\mathfrak{B}$ and $\mathfrak{N}$ are isomorphic, and $\mathfrak{B}$ is sound, $\mathfrak{N}$ is also sound. The deduction rules in Figure 2.1 all correspond to arithmetic tautologies via the translation in the previous section. For example the $\triangleright$-intro rule roughly[6] translates to $\forall n.\ n \leq n+1$, the $\triangleright$-mono rule to $\forall\ m\ n.\ n \leq m \to n+1 \leq m+1$, and the $\bot$-elim rule to $\forall n.\ 0 \leq n$.

We now face the following question: what deduction rules can we add to our logic such that every tautology is provable? Can we perhaps find such a rule as a translation of another arithmetic tautology? It turns out we need just one more rule, which in arithmetic states the linear ordering of natural numbers: $\forall\ m\ n.\ m \leq n \vee m > n$. We can express this rule in $\mathcal{L}_\triangleright$ and add it as an axiom. We call this the *comparison rule*:

$$\frac{}{\top \vdash \varphi_0 \Rightarrow \varphi_1 \vee \triangleright\varphi_1 \Rightarrow \varphi_0}\ \text{compare}$$

Figure 2.2: The comparison rule.

Note that this rule holds in both $\mathfrak{N}$ and $\mathfrak{B}$ (so Theorem 1 still holds). Lets see how we can make more complex case distinctions using this rule. Since the full proof tree is large and contains many trivial steps, we give an informal desription.

Theorem 3. $\top \vdash \triangleright\varphi_0 \Rightarrow \varphi_1 \vee \varphi_0 \Leftrightarrow \varphi_1 \vee \triangleright\varphi_1 \Rightarrow \varphi_0$

Proof. Eliminate $\top \vdash \varphi_0 \Rightarrow \varphi_1 \vee \triangleright\varphi_1 \Rightarrow \varphi_0$. (case A) Suppose $\varphi_0 \Rightarrow \varphi_1$. Eliminate $\top \vdash \varphi_1 \Rightarrow \varphi_0 \vee \triangleright\varphi_0 \Rightarrow \varphi_1$. (case A1) Suppose $\varphi_1 \Rightarrow \varphi_0$. Now we have $\varphi_0 \Leftrightarrow \varphi_1$. (case A2) Now we have $\triangleright\varphi_0 \Rightarrow \varphi_1$. (case B) Now we have $\triangleright\varphi_1 \Rightarrow \varphi_0$. □

Note that under the arithmetic translation Theorem 3 corresponds to the *law of trichotomy*, which states $\forall\ m\ n.\ m < n \vee m = n \vee m > n$. We can add more eliminations of the comparison rule to obtain more specific case distinctions, for example one corresponding to the following disjunction:

$$\forall\ m\ n.\ m+2 \leq n \vee m+1 = n \vee m = n \vee n+1 = m \vee n+2 \leq m$$

The next section will explain how we prove the completeness and decidability of the deduction system resulting from adding the comparison rule to the rules in Figure 2.1.

[6] We are ignoring the role of ω.

2.6 Eliminating connectives

We prove completeness and decidability together by giving an algorithm that, given a formula φ, constructs a derivation of φ (*i.e.* a deduction $\top \vdash \varphi$), or outputs a valuation Γ such that φ is not realized (*i.e.* $\mathfrak{N}[\![\varphi]\!](\Gamma) \neq \omega$). This section outlines the intuition behind this algorithm, and the next section describes its precise formalization in Coq.

Analogy with truth-tables Our process is somewhat reminiscent of the method of truth-tables for classical propositional logic; by checking all rows of a truth-table one can determine that the formula is a tautology, which can be derived with repeated use of the law of excluded middle, or one finds a row (*i.e.* a valuation) such that the formula is false. But in contrast with 2-valued logic, our "$\mathbb{N} \cup \{\omega\}$"-valued logic has an infinite number of possible valuations!

The key insight is that we do not have to check all of these valuations; it suffices to check just a finite number of them. All other valuations are in some sense similar enough to these valuations such that we can omit them. The reason for this will hopefully become clear over the course of this section.

Reduction rules Below are six reduction rules that will aid our algorithm. Rules 1 to 5 are straightforward, and also hold for plain intuitionistic logic. Rule 6 is interesting because deriving it requires the Löb induction rule. We derived the left to right implication of this rule in Section 2.3.

$$\varphi_0 \Rightarrow \varphi_1 \vdash (\varphi_0 \wedge \varphi_1) \Leftrightarrow \varphi_0 \tag{1}$$
$$\varphi_1 \Rightarrow \varphi_0 \vdash (\varphi_0 \wedge \varphi_1) \Leftrightarrow \varphi_1 \tag{2}$$

$$\varphi_0 \Rightarrow \varphi_1 \vdash (\varphi_0 \vee \varphi_1) \Leftrightarrow \varphi_1 \tag{3}$$
$$\varphi_1 \Rightarrow \varphi_0 \vdash (\varphi_0 \vee \varphi_1) \Leftrightarrow \varphi_0 \tag{4}$$

$$\varphi_0 \Rightarrow \varphi_1 \vdash (\varphi_0 \Rightarrow \varphi_1) \Leftrightarrow \top \tag{5}$$
$$\triangleright\varphi_1 \Rightarrow \varphi_0 \vdash (\varphi_0 \Rightarrow \varphi_1) \Leftrightarrow \varphi_1 \tag{6}$$

Example Recall the disjunction we proved in Theorem 3. We will show how this disjunction, combined with the above reduction rules, produces a derivation the formula $(\triangleright P_0 \Rightarrow P_1 \wedge P_1 \Rightarrow P_0) \Rightarrow P_0$. We prove this by showing that is is equivalent to $\top$:

$$((\triangleright P_0 \Rightarrow P_1 \wedge P_1 \Rightarrow P_0) \Rightarrow P_0) \Leftrightarrow \top$$

Theorem 4. $\top \vdash ((\triangleright P_0 \Rightarrow P_1 \wedge P_1 \Rightarrow P_0) \Rightarrow P_0) \Leftrightarrow \top$

Proof. Eliminate $\top \vdash \triangleright P_0 \Rightarrow P_1 \vee P_0 \Leftrightarrow P_1 \vee \triangleright P_1 \Rightarrow P_0$ (Theorem 3).

- **Case 1:** Suppose $\triangleright P_0 \Rightarrow P_1$. We apply reductions until $\top$ is left.

  $$(\triangleright P_0 \Rightarrow P_1 \wedge P_1 \Rightarrow P_0) \Rightarrow P_0$$

  Using 5 replace $\triangleright P_0 \Rightarrow P_1$ with $\top$, and using 6 replace $P_1 \Rightarrow P_0$ with P_0:

  $$(\top \wedge P_0) \Rightarrow P_0$$

  Note that $P_0 \Rightarrow \top$ follows from $\top$-intro. Using 2 replace $\top \wedge P_0$ with P_0:

  $$P_0 \Rightarrow P_0$$

  Note that $P_0 \Rightarrow P_0$ follows from refl. Using 5 replace $P_0 \Rightarrow P_0$ with $\top$:

  $$\top$$

  We have shown that under the assumption $\triangleright P_0 \Rightarrow P_1$, the following holds:

  $$((\triangleright P_0 \Rightarrow P_1 \wedge P_1 \Rightarrow P_0) \Rightarrow P_0) \Leftrightarrow \top$$

- **Case 2:** Suppose $P_0 \Leftrightarrow P_1$.

  $$(\triangleright P_0 \Rightarrow P_1 \wedge P_1 \Rightarrow P_0) \Rightarrow P_0$$

  Note $\triangleright P_1 \Rightarrow \triangleright P_0$ using $\triangleright$-mono. Use 6 and 5:

  $$(P_1 \wedge \top) \Rightarrow P_0$$

  Use 1 and then 5:

  $$\top$$

- **Case 3:** Suppose $\triangleright P_1 \Rightarrow P_0$.

  $$(\triangleright P_0 \Rightarrow P_1 \wedge P_1 \Rightarrow P_0) \Rightarrow P_0$$

  Note $P_1 \Rightarrow P_0$ using $\triangleright$-intro. Use 6 and 5:

  $$(P_1 \wedge \top) \Rightarrow P_1$$

  The rest is analogous to *Case 2*.

□

In the above proof we could also have eliminated $\top \vdash P_1 \Rightarrow P_0 \vee \triangleright P_0 \Rightarrow P_1$, generating just two cases. However eliminating $\top \vdash P_0 \Rightarrow P_1 \vee \triangleright P_1 \Rightarrow P_0$ would *not* work! With the assumption $P_0 \Rightarrow P_1$ we *cannot* reduce $P_1 \Rightarrow P_0$, since neither $P_1 \Rightarrow P_0$ nor $\triangleright P_0 \Rightarrow P_1$ follows from $P_0 \Rightarrow P_1$. Making a case distinction that is "detailled" enough is an important aspect of our algorithm.

The reduction theorem We will call formulas like $\triangleright P_0 \Rightarrow P_1$, $P_0 \Leftrightarrow P_1$, and $\triangleright P_1 \Rightarrow P_0$ in Theorem 4 *case formulas*. To always finish the reduction process we need the case formulas to contain enough information. For this we need a way to define what subformulas we can possibly encounter during the reduction process. First we define the set of variables (FV) and the modal depth (MD) of a formula. Below the $\square$ is a placeholder for any of the three binary connectives $\wedge, \vee, \Rightarrow$.

$$\begin{aligned} \mathrm{FV}(P_i) &:= \{P_i\} \\ \mathrm{FV}(\triangleright\varphi) &:= \mathrm{FV}(\varphi) \\ \mathrm{FV}(\varphi_0 \mathbin{\square} \varphi_1) &:= \mathrm{FV}(\varphi_0) \cup \mathrm{FV}(\varphi_1) \end{aligned}$$

$$\begin{aligned} \mathrm{MD}(P_i) &:= 0 \\ \mathrm{MD}(\triangleright\varphi) &:= 1 + \mathrm{MD}(\varphi) \\ \mathrm{MD}(\varphi_0 \mathbin{\square} \varphi_1) &:= \max\{\mathrm{MD}(\varphi_0), \mathrm{MD}(\varphi_1)\} \end{aligned}$$

The elimination of connectives is carried out bottom-up (starting at the leafs of the formula), and will at each step yield a formula of the form $\triangleright^n x$. Here x is either $\top$, $\bot$, or a proposition letter P_i. Note that the $\triangleright$'s may stack up, but their number will never exceed the modal depth of the formula we are reducing.

With these ideas in mind we define *atomic formulas* of φ. For added clarity we will now use τ to denote atomic formulas, and σ to denote case formulas.

Definition 5. *A formula τ is a φ-atomic formula if there exists an $n \leq \mathrm{MD}(\varphi)$ and an $x \in \{\top, \bot\} \cup \mathrm{FV}(\varphi)$ such that $\tau = \triangleright^n x$.*

We can now define what it means for a case formula to contain "enough information" to carry out a full reduction. We call this property *exhaustive*.

Definition 6. *A formula σ is exhaustive for φ if for every two φ-atomic formulas τ_0 and τ_1 either $\sigma \vdash \tau_0 \Rightarrow \tau_1$ or $\sigma \vdash \triangleright\tau_1 \Rightarrow \tau_0$.*

Using these definitions we can define the *reduction theorem*, which generalizes the reduction process that we used in the example.

Theorem 7. *If σ is exhaustive for φ then there exists a φ-atomic formula τ such that $\sigma \vdash \varphi \Leftrightarrow \tau$.*

We proved this theorem in Coq, so we will encounter it again in the next section.

Cases Note that the number of atomic formulas of a formula φ is finite. We can use the comparison rule to make a deduction of a disjunction of exhaustive case formulas, just like we did in Theorem 3. In practice this disjunction includes all possible permutations of $\mathrm{FV}(\varphi)$ combined with different possible $\triangleright$-*offsets* between these variables. Because the notion of offset will return in the discription of our Coq formalization, we will now define it.

Definition 8. *The offset between τ_0 and τ_1 in σ is at least n if $\sigma \vdash \triangleright^n \tau_0 \Rightarrow \tau_1$, and at most n if $\sigma \vdash \tau_1 \Rightarrow \triangleright^n \tau_0$.*

Valuations It is straightforward to determine a valuation that realizes a given case formula. To see this it is useful to again translate formulas to arithmetic. Consider the case formula $\triangleright P_1 \Rightarrow P_0$ that we encountered in Theorem 4. This corresponds to $p_1 + 1 \leq p_0$ in arithmetic, where p_i represents the value assigned to P_i. To satisfy this inequality we could pick $p_1 = 0$ and $p_0 = 1$. The formula $P_0 \Leftrightarrow P_1$ corresponds to $p_0 = p_1$, and we could pick $p_0 = p_1 = 0$. The case formulas are constructed in such a way that a valuation can easily be determined. Moreover, we can *always* pick a valuation not containing ω.

Completeness If for some case formula, the input formula φ reduces to an atomic formula that is not equivalent to $\top$, then that case formula is a *counterexample*. There exists a valuation Γ such that $\mathfrak{N}[\![\varphi]\!](\Gamma) \neq \omega$. However if φ is a tautology, *i.e.* $\top \models_{\mathfrak{N}} \varphi$, then there cannot be any counterexamples. Therefore φ should reduce to $\top$ under every case formula. Together with a deduction of the disjunction of case formulas this produces a deduction of φ.

Theorem 9. *(completeness) If $\varphi_0 \models_{\mathfrak{N}} \varphi_1$ then $\varphi_0 \vdash \varphi_1$.*

We also proved this theorem in Coq. Together with the soundness of $\mathfrak{N}$ it shows that deductions in our system correspond exactly with the tautologies of $\mathfrak{N}$.

Second example We look at how reduction yields a counterexample.

$$\varphi := (\triangleright P_0 \Rightarrow \triangleright P_1) \Rightarrow P_0 \Rightarrow P_1$$

Note that by definition $\bot$ is also a φ-atomic formula, and so it must be taken into account. The formula $\sigma := \bot \Leftrightarrow P_1 \wedge \triangleright P_1 \Leftrightarrow P_0$ is exhaustive for φ. Note that $\sigma \vdash \triangleright P_1 \Rightarrow P_0$. Applying the reduction algorithm yields $\sigma \vdash \varphi \Leftrightarrow P_1$. This is a counterexample because P_1 is not equivalent to $\top$. The following valuation realizes σ:

$$\Gamma : \Sigma \to \mathbb{N} \cup \{\omega\} := \{(P_0, 1), (P_1, 0)\}$$

Now $\mathfrak{N}[\![\varphi]\!](\Gamma) = \mathfrak{N}[\![P_1]\!](\Gamma) = 0 \neq \omega$, demonstrating that φ is *not* a tautology.

Decidability The process we have described is an algorithm. We can effectively search for a counterexample, and construct a deduction if there is none. Therefore our result also shows that deductions are decidable.

2.7 The Coq formalization

We have worked out a proof of Theorem 9 in Coq. This section will outline the definitions and intermediary lemmas of this formalization. We used list and set utilities from the `Coq-std++` library [21].

Formulas The definition of formulas is straightforward. Formulas have a dynamic type for terms, such that we can use both syntactic terms ($\top$, $\bot$, and P_i) and model values. This is useful as intermediate type for evaluation. The type of syntactic formulas is written as `form term`.

Deductions Deductions are implemented as an inductive predicate. The deduction rules are equivalent to those given in Figure 2.1 and 2.2.

```
Inductive deduction : form term → form term → Prop :=
  | d_refl p           : p ⊢ p
  | d_trans p q r      : p ⊢ q → q ⊢ r → p ⊢ r
  | d_true_intro p     : p ⊢ ⊤
  | d_false_elim p     : ⊥ ⊢ p
  | d_conj_intro c p q : c ⊢ p → c ⊢ q → c ⊢ p `∧` q
  | d_conj_elim_l p q  : p `∧` q ⊢ p
  | d_conj_elim_r p q  : p `∧` q ⊢ q
  | d_disj_intro_l p q : p ⊢ p `∨` q
  | d_disj_intro_r p q : q ⊢ p `∨` q
  | d_disj_elim p q r  : p ⊢ r → q ⊢ r → p `∨` q ⊢ r
  | d_impl_intro c p q : c `∧` p ⊢ q → c ⊢ p ⟹ q
  | d_impl_elim c p q  : c ⊢ p ⟹ q → c ⊢ p → c ⊢ q
  | d_later_intro p    : p ⊢ ▷p
  | d_later_elim p     : ⊢ ▷p → ⊢ p
  | d_later_fix p      : ▷p ⊢ p → ⊢ p
  | d_later_mono p q   : p ⊢ q → ▷p ⊢ ▷q
  | d_later_conj p q   : ▷p `∧` ▷q ⊢ ▷(p `∧` q)
  | d_compare p q      : ⊢ p ⟹ q `∨` ▷q ⟹ p
  where "p ⊢ q" := (deduction p q) and "⊢ q" := (⊤ ⊢ q).
```

N-ary operators We defined a function to create a conjunction or a disjunction of a list of formulas. Deductions can be made with the following lemmas.

```
Lemma d_big_conj_intro c ps : (∀ p, p ∈ ps → c ⊢ p) → c ⊢ ⋀ ps.
Lemma d_big_conj_elim p ps : p ∈ ps → ⋀ ps ⊢ p.
Lemma d_big_disj_intro p q qs : p ∈ ps → p ⊢ ⋁ ps.
Lemma d_big_disj_elim ps q : (∀ p, p ∈ ps → p ⊢ q) → ⋁ ps ⊢ q.
```

The reference model We will show that all tautologies of $\mathfrak{N}$ have a deduction, so we have to define $\mathfrak{N}$ in Coq as well. In our formalization the domain of $\mathfrak{N}$ has type `Sω`, which refers to the ordinal $S(\omega)$.

```
Inductive Sω := Finite (n : nat) | Infinite.
```

We define a number of simple operations on `Sω` to build an evaluation function called `eval` corresponding to the denotation of $\mathfrak{N}$.

```
Fixpoint eval (f : form Sω) : Sω :=
  match f with
  | ($x) ⇒ x
  | (▷p) ⇒ Sω_succ (eval p)
  | (p `∧` q) ⇒ Sω_min (eval p) (eval q)
  | (p `∨` q) ⇒ Sω_max (eval p) (eval q)
  | (p ⟹ q) ⇒ if Sω_leb (eval p) (eval q) then Infinite else eval q
  end.
```

A function called `interp` maps terms to `Sω` using a valuation `Γ : nat → Sω` that assigns values to proposition letters. The value of $\bot$ is `Finite 0` and the value of $\top$ is `Infinite`. Using this we define realization within $\mathfrak{N}$.

```
Definition realizes (Γ : nat → Sω) (p q : form term) : Prop :=
  Sω_le (eval (interp Γ <$> p)) (eval (interp Γ <$> q)).
```

Note that `realizes Γ p q` means the same as $p \models_{\mathfrak{N}} q$. We prove that that a deduction is realized for every valuation (*i.e.* that $\mathfrak{N}$ is sound). This proof proceeds by induction on the deduction tree.

```
Theorem deduction_sound Γ p q :
  p ⊢ q → realizes Γ p q.
```

Formula reduction We prove the reduction theorem (Theorem 7) given a case formula `ctx : form term`, a set of variables `fv : gset nat`, and a modal depth `md : nat`. The `atom` predicate is analogous to Definition 5. We assume that `ctx` is exhaustive (Definition 6) for the given variables and modal depth.

```
Hypothesis exhaustive : ∀ a b,
  atom fv md a → atom fv md b →
  ctx ⊢ a ⟹ b ∨ ctx ⊢ ▷b ⟹ a.
```

The reduction is formulated as follows.

```
Theorem d_reduce_to_atom f :
  FV f ⊆ fv → MD f ≤ md →
  ∃ a, atom (FV f) (MD f) a ∧ ctx ⊢ f ⇔ a.
```

Implication permutations Now we need to find a deduction for a disjunction of cases, such that each case is exhaustive. The first step is to look at the order in which the variables imply each other. We define a function `adj` that returns a list of adjacent pairs of elements in a list. For example:

```
adj (x0 :: x1 :: x2 :: xs) = (x0, x1) :: (x1, x2) :: adj (x2 :: xs)
```

Using this utility we define a formula representing one such ordering. Here `xs : list nat` is some permutation of variable indices, `p.1` is a notation to extract the first element of a pair, and `#i` is a notation for the formula P_i.

```
Definition perm_form xs := ⋀ ((λ p, #p.1 ⟹ #p.2) <$> adj xs).
```

We prove that a disjunction of all possible permutations can be derived.

```
Theorem d_permutations (xs : list nat) :
  ⊢ ⋁ (perm_form <$> permutations xs).
```

Later offsets Given two formulas `p, q : form term` such that `ctx ⊢ p ⟹ q`, we prove that a disjunction of all possible offsets (Definition 8) between 0 and n can be deduced. The remaining case states that the offset is at least n: `ctx ⊢ n*▷p ⟹ q`. Here `n*▷p` is a notation for $\triangleright^n p$.

```
Lemma d_offsets ctx p q n :
  ctx ⊢ p ⟹ q →
  ctx ⊢ n*▷p ⟹ q `v` ⋁ ((λ i, i*▷p ⟺ q) <$> seq 0 n).
```

Case listing We can combine the permutations and offsets into a disjunction of exhaustive cases. Each case is generated by a `list (nat * nat)` that specifies the variable permutation and offsets. The `list_cases` function generates all cases given a set of variables and a modal depth.

```
Definition list_cases (fv : gset nat) (md : nat) : list (list (nat * nat)) :=
  let skips := seq 0 (2 + md) in
  let perms := permutations (elements fv) in
  xs ← perms; mapM (λ i, pair i <$> skips) xs.
```

The `case_form` function generates a formula for each case. We prove that each `case_form` is indeed exhaustive[7], and that a disjunction of case formulas based on `list_cases` can be deduced.

```
Lemma d_list_cases fv md :
  ⊢ ⋁ (case_form md ⊥ <$> list_cases fv md).
```

Case realization Given the configuration of one case, we assign a number to each variable index `i : nat` using the `case_val` function.

```
Fixpoint case_val (case : list (nat * nat)) (i : nat) : nat :=
  match case with
  | [] ⇒ 0
  | (j, n) :: case' ⇒ n + if i =? j then 0 else case_val case' i
  end.
```

When composed with the `Finite` constructor of `Sω` this becomes a valuation. We prove that this valuation realizes the case formula. The condition `NoDup case.*1` requires that there are no duplicate variable indices in the case list.

```
Lemma case_val_realizes_case_form md case :
  NoDup case.*1 →
  realizes (Finite ∘ case_val case) ⊤ (case_form md ⊥ case).
```

[7] This lemma is omitted since it is quite verbose.

Counterexample search Using the case valuation we can compute if a formula `f : form term` is true for a given case. This is computed by the `eval_case` function, which internally uses `eval`.

```
Definition eval_case (f : form term) (case : list (nat * nat)) : bool :=
  Sω_leb Infinite (eval (interp (Finite ∘ case_val case) <$> f)).
```

Now we can implement a function that decides if a formula is a tautology. It uses `list_cases` to get a complete list of cases, and `eval_case` to evaluate each case. If the result is false for one case, then that case is a counterexample. The first counterexample is returned, or `None` if the formula is a tautology.

```
Definition counterexample (f : form term) : option (list (nat * nat)) :=
  find (negb ∘ eval_case f) (list_cases (FV f) (MD f)).
```

When a counterexample is found the formula is *not* realized with the corresponding case valuation (*i.e.* `counterexample` is sound).

```
Theorem counterexample_sound f case :
  counterexample f = Some case →
  ¬realizes (Finite ∘ case_val case) ⊤ f.
```

Completeness If the formula is true for a given case, then it must reduce to $\top$. We use this to prove the completeness of `eval_case`. The `case.*1 ≡ₚ elements (FV f)` condition requires that the case variables are a permutation of the formula variables.

```
Lemma eval_case_complete f case :
  case.*1 ≡ₚ elements (FV f) →
  eval_case f case = true →
  case_form (MD f) ⊥ case ⊢ f.
```

Finally we prove that if there is no counterexample, then there is a deduction of the formula. This requires the `d_list_cases` lemma.

```
Theorem counterexample_complete f :
  counterexample f = None → ⊤ ⊢ f.
```

Since $\varphi_0 \vdash \varphi_1$ if and only if $\top \vdash \varphi_0 \Rightarrow \varphi_1$ we can generalize this result. We explicitly show that deductions are decidable.

```
Theorem deduction_decidable p q :
  { p ⊢ q } + { ∃ Γ, ¬ realizes Γ p q }.
```

From this the completeness of deductions (Theorem 9) directly follows.

```
Corollary deduction_complete p q :
  (∀ Γ, realizes Γ p q) → p ⊢ q.
```

2.8 Conclusion

We have seen how propositional logic with the $\triangleright$ modality translates to a subset of linear integer arithmetic, and how the comparison rule completes our deduction system by enabling a truth-table like algorithm. It is not clear under what extensions, such as quantification or a separating conjunction with mutable resources, the comparison rule remains valid. The author suspects the scope of the comparison rule is rather limited.

The Disjunction Property The Disjunction Property states that for every deduction $\varphi_0 \vdash \varphi_1 \vee \varphi_2$ with $\vee$ not occuring in φ_0, there is either a deduction $\varphi_0 \vdash \varphi_1$ or $\varphi_0 \vdash \varphi_2$. Intuitionistic logics satisfy the Disjunction Property[6], but our comparison rule cearly does not. Therefore we consider it anti-intuitionistic.

2.9 Related work

Models such as the one used by Iris have been studied a lot. A key paper about this is by Appel et al. [14]. This paper develops the later modality, which is analogous to the *approximation modality* developed by Nakano in 2000 [10]. Nakano proved the Kripke-completeness[8] and decidability of the modal logic corresponding to the typing rules for his approximation modality [12]. His typing rules[9] are based on $\lambda\mu$, the simply typed lambda calculus with recursive types. It is perhaps unsurprising that his logic has no analog to the comparison rule, since Kripke-completeness is commonly used for intuitionistic logics.

Van Doorn proved the completeness of a classical deduction system for propositional logic in Coq [17]. This proof proceeds by converting formulas to Conjunctive Normal Form, and proving that they are tautologies precisely when every clause contains either $\neg\bot$ or $p \vee \neg p$. Similar to our formalization van Doorn uses inductive types to represent a deduction system.

There is a range of publications about the later modality and related type theories. We did not investigate this literature in detail, though it appears the completeness result presented in this chapter has not been discussed before.

[8] Kripke-completeness refers to derivability with respect to Kripke models: the logic is Kripke-complete when a formula is derivable if and only if it is true in every Kripke model for this logic. Intuïtionistic natural deduction is Kripke-complete [3].

[9] See Figure 3 of [12].

Bibliography

[1] Mojżesz Presburger. "Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt". In: *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves* (1929), pp. 92–101, 395.

[2] J. Richard Büchi. "Weak Second-Order Arithmetic and Finite Automata". In: *Mathematical Logic Quarterly* 6.1-6 (1960), pp. 66–92. DOI: https://doi.org/10.1002/malq.19600060105.

[3] Saul A. Kripke. "Semantical Analysis of Intuitionistic Logic I". In: *Formal Systems and Recursive Functions*. Ed. by J.N. Crossley and M.A.E. Dummett. Vol. 40. Studies in Logic and the Foundations of Mathematics. Elsevier, 1965, pp. 92–130. DOI: 10.1016/S0049-237X(08)71685-9.

[4] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259.

[5] David C Cooper. "Theorem proving in arithmetic without multiplication". In: *Machine intelligence* 7.91-99 (1972), p. 300.

[6] Alexander Chagrov and Michael Zakharyashchev. "The Disjunction Property of Intermediate Propositional Logics". In: *Studia Logica: An International Journal for Symbolic Logic* 50.2 (1991), pp. 189–216. ISSN: 00393215, 15728730. URL: http://www.jstor.org/stable/20015573.

[7] Alexandre Boudet and Hubert Comon. "Diophantine equations, Presburger arithmetic and finite automata". In: *Trees in Algebra and Programming — CAAP '96*. Ed. by Hélène Kirchner. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 30–43. ISBN: 978-3-540-49944-2.

[8] Jacob Elgaard, Nils Klarlund, and Anders Møller. "MONA 1.x: New Techniques for WS1S and WS2S". English. In: *Computer Aided Verification*. Ed. by Alan J. Hu and Moshe Y. Vardi. Lecture Notes in Computer Science. 10th International Conference on Computer Aided Verification. CAV 1998 ; Conference date: 28-06-1998 Through 02-07-1998. Netherlands: Springer, 1998, pp. 516–520. DOI: 10.1007/BFb0028773.

[9] Peter W. O'Hearn and David J. Pym. "The Logic of Bunched Implications". In: *The Bulletin of Symbolic Logic* 5.2 (1999), pp. 215–244. ISSN: 10798986. URL: http://www.jstor.org/stable/421090.

[10] Hiroshi Nakano. "A modality for recursion". In: *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*. 2000, pp. 255–266. DOI: 10.1109/LICS.2000.855774.

[11] John C. Reynolds. "Intuitionistic Reasoning about Shared Mutable Data Structure". In: *Millennial Perspectives in Computer Science*. Palgrave, 2000, pp. 303–321.

[12] Hiroshi Nakano. "Fixed-Point Logic with the Approximation Modality and Its Kripke Completeness". In: *Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software*. TACS '01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 165–182. ISBN: 3540427368.

[13] J.C. Reynolds. "Separation logic: a logic for shared mutable data structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[14] Andrew W. Appel et al. "A Very Modal Model of a Modern, Major, General Type System". In: *SIGPLAN Not.* 42.1 (Jan. 2007), pp. 109–122. ISSN: 0362-1340. DOI: 10.1145/1190215.1190235.

[15] Karen Zee, Viktor Kuncak, and Martin Rinard. "Full Functional Verification of Linked Data Structures". In: *SIGPLAN Not.* 43.6 (June 2008), pp. 349–361. ISSN: 0362-1340. DOI: 10.1145/1379022.1375624.

[16] Karen Zee. "Verification of Full Functional Correctness for Imperative Linked Data Structures". PhD thesis. Massachusetts Institute of Technology, Sept. 2010. URL: http://hdl.handle.net/1721.1/58078.

[17] Floris van Doorn. *Propositional Calculus in Coq*. 2015. arXiv: 1503.08744 [math.LO].

[18] The Coq Development Team. *The Coq Proof Assistant, version 8.7.0*. Version 8.7.0. Oct. 2017. DOI: 10.5281/zenodo.1028037.

[19] Ralf Jung et al. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 28 (2018), e20. DOI: 10.1017/S0956796818000151.

[20] Wim Veldman. "Intuitionism: An Inspiration?" In: *Jahresbericht der Deutschen Mathematiker-Vereinigung* 123 (Apr. 2021). DOI: 10.1365/s13291-021-00230-8.

[21] The Coq-std++ Team. *An extended "standard library" for Coq*. Available online at https://gitlab.mpi-sws.org/iris/stdpp. 2022.

Appendix A

The strong Löb rule

We derive the strong Löb induction rule, which is the following entailment:

$$\triangleright\varphi_0 \Rightarrow \varphi_0 \vdash \varphi_0$$

To fit this deduction tree on a page we first prove several lemmas.

A.1

$$\dfrac{\dfrac{}{\sigma \wedge \varphi_0 \vdash \sigma}\ \wedge\text{-elim-l} \qquad \sigma \vdash \varphi_0 \Rightarrow \varphi_1}{\sigma \wedge \varphi_0 \vdash \varphi_0 \Rightarrow \varphi_1}\ \text{trans} \qquad \dfrac{\dfrac{}{\sigma \wedge \varphi_0 \vdash \varphi_0}\ \wedge\text{-elim-r} \qquad \sigma \wedge \varphi_0 \vdash \varphi_0 \Rightarrow \varphi_1}{\sigma \wedge \varphi_0 \vdash \varphi_1}\ \Rightarrow\text{-elim}$$

A.2

$$\dfrac{\dfrac{}{\varphi_0 \vdash \top}\ \top\text{-intro} \qquad \top \vdash \varphi_0 \Rightarrow \varphi_1}{\varphi_0 \vdash \varphi_0 \Rightarrow \varphi_1}\ \text{trans} \qquad \dfrac{\dfrac{}{\varphi_0 \vdash \varphi_0}\ \text{refl} \qquad \varphi_0 \vdash \varphi_0 \Rightarrow \varphi_1}{\varphi_0 \vdash \varphi_1}\ \Rightarrow\text{-elim}$$

A.3

$$\dfrac{\dfrac{\dfrac{}{\varphi_0 \wedge \varphi_1 \vdash \varphi_1}\ \wedge\text{-elim-r} \qquad \dfrac{}{\varphi_0 \wedge \varphi_1 \vdash \varphi_0}\ \wedge\text{-elim-l}}{\varphi_0 \wedge \varphi_1 \vdash \varphi_1 \wedge \varphi_0}\ \wedge\text{-intro} \qquad \varphi_1 \wedge \varphi_0 \vdash \varphi_2}{\varphi_0 \wedge \varphi_1 \vdash \varphi_2}\ \text{trans}$$

A.4

$$\dfrac{\dfrac{}{\triangleright\varphi_0 \wedge \triangleright\varphi_1 \vdash \triangleright(\varphi_0 \wedge \varphi_1)}\ \triangleright\text{-conj} \qquad \dfrac{\varphi_0 \wedge \varphi_1 \vdash \varphi_2}{\triangleright(\varphi_0 \wedge \varphi_1) \vdash \triangleright\varphi_2}\ \triangleright\text{-mono}}{\triangleright\varphi_0 \wedge \triangleright\varphi_1 \vdash \triangleright\varphi_2}\ \text{trans}$$

A.5

$$\dfrac{\dfrac{}{\varphi_0 \vdash \triangleright\varphi_0}\ \triangleright\text{-intro} \qquad \triangleright\varphi_0 \vdash \varphi_1}{\varphi_0 \vdash \varphi_1}\ \text{trans}$$

A.6

$$\dfrac{\dfrac{}{\varphi_0 \wedge \varphi_1 \Rightarrow \varphi_2 \vdash \varphi_1 \Rightarrow \varphi_2}\ \wedge\text{-elim-r} \qquad \varphi_0 \wedge \varphi_1 \Rightarrow \varphi_2 \vdash \varphi_1}{\varphi_0 \wedge \varphi_1 \Rightarrow \varphi_2 \vdash \varphi_2}\ \Rightarrow\text{-elim}$$

A.7

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\triangleright\varphi_0 \Rightarrow \varphi_0 \wedge (\triangleright\varphi_0 \Rightarrow \varphi_0) \Rightarrow \varphi_0 \vdash \triangleright\varphi_0 \Rightarrow \varphi_0}\ \wedge\text{-elim-l}}{\triangleright\varphi_0 \Rightarrow \varphi_0 \wedge (\triangleright\varphi_0 \Rightarrow \varphi_0) \Rightarrow \varphi_0 \vdash \triangleright\varphi_0 \Rightarrow \varphi_0}\ \text{A.6}}{\triangleright\varphi_0 \Rightarrow \varphi_0 \wedge (\triangleright\varphi_0 \Rightarrow \varphi_0) \Rightarrow \varphi_0 \vdash \varphi_0}\ \text{A.4}}{\triangleright(\triangleright\varphi_0 \Rightarrow \varphi_0) \wedge \triangleright((\triangleright\varphi_0 \Rightarrow \varphi_0) \Rightarrow \varphi_0) \vdash \triangleright\varphi_0}\ \Rightarrow\text{-intro}}{\triangleright(\triangleright\varphi_0 \Rightarrow \varphi_0) \vdash \triangleright((\triangleright\varphi_0 \Rightarrow \varphi_0) \Rightarrow \varphi_0) \Rightarrow \triangleright\varphi_0}\ \text{A.5}}{\triangleright\varphi_0 \Rightarrow \varphi_0 \vdash \triangleright((\triangleright\varphi_0 \Rightarrow \varphi_0) \Rightarrow \varphi_0) \Rightarrow \triangleright\varphi_0}\ \text{A.1}}{\triangleright\varphi_0 \Rightarrow \varphi_0 \wedge \triangleright((\triangleright\varphi_0 \Rightarrow \varphi_0) \Rightarrow \varphi_0) \vdash \triangleright\varphi_0}\ \text{A.3}}{\triangleright((\triangleright\varphi_0 \Rightarrow \varphi_0) \Rightarrow \varphi_0) \wedge \triangleright\varphi_0 \Rightarrow \varphi_0 \vdash \triangleright\varphi_0}\ \text{A.6}}{\triangleright((\triangleright\varphi_0 \Rightarrow \varphi_0) \Rightarrow \varphi_0) \wedge \triangleright\varphi_0 \Rightarrow \varphi_0 \vdash \varphi_0}\ \Rightarrow\text{-intro}}{\triangleright((\triangleright\varphi_0 \Rightarrow \varphi_0) \Rightarrow \varphi_0) \vdash (\triangleright\varphi_0 \Rightarrow \varphi_0) \Rightarrow \varphi_0}\ \triangleright\text{-Löb}}{\top \vdash (\triangleright\varphi_0 \Rightarrow \varphi_0) \Rightarrow \varphi_0}\ \text{A.2}}{\triangleright\varphi_0 \Rightarrow \varphi_0 \vdash \varphi_0}$$