FORMAL PROOF OF THE FUNDAMENTAL THEOREM OF DECORATED INTERVAL ARITHMETIC

FORMAL PROOF OF THE FUNDAMENTAL THEOREM OF DECORATED INTERVAL ARITHMETIC

By

BINGZHOU ZHENG, M.Sc., B.Eng.

A Thesis

Submitted to the Department of Computing and Software

and the School of Graduate Studies

of McMaster University

in Partial Fulfilment of the Requirements

for the Degree of

Doctor of Philosophy

Doctor of Philosophy (2013) McMaster University

(Computing and Software) Hamilton, Ontario, Canada

TITLE:	Formal Proof of the Fundamental Theorem of Decorated Interval Arithmetic
AUTHOR:	Bingzhou Zheng M.Sc. (Computer Science) McMaster University, Hamilton, Canada B.Eng. (Control Theory and Technology) Taiyuan University of Technology, Taiyuan, China
SUPERVISOR:	Dr. Ned Nedialkov
NUMBER OF PAGES:	v, 123

To my parents, Mindi Zheng and Huiru Liu

Abstract

Interval arithmetic is used to enclose roundoff, truncation, and modeling errors in interval methods, thus obtaining numerical methods with automatic verification of the results. The Fundamental Theorem of Interval Arithmetic (FTIA) shows that, when evaluating an expression using interval arithmetic, the computed result contains the mathematically correct value of the expression.

Decorations were introduced in the IEEE P1788 working group for standardizing interval arithmetic. Their role is to help track properties of interval evaluations. That is, we wish to say if a function is defined, undefined, or continuous in its inputs. Moreover, decorations act as local exception flags and do not lead to interruption of the computations. The FTIA plus the decoration system is expanded into the Fundamental Theorem of Decorated Interval Arithmetic (FTDIA).

Several versions of this theorem are formulated and proved by J. Pryce. This thesis formalizes and proves the core of this theorem (version 3.0 of the IEEE-P1788 proposal) using the theorem prover Coq. Namely, we prove it for the common case where all the inputs to a function are non-empty intervals.

There are two distinctive features of our formalization and proof. First, we define the semantics of an interval as a set of real numbers (including the empty set), and we do not impose any other restrictions on such a set, except that models of this interval can decide if the set is empty or not. For example, an interval need not be closed and

bounded, as in traditional interval arithmetic. Second, our formalization and proof do not rely on specific interval operations: it works with any interval operation that satisfies the requirements for decorated interval library operations.

As the FTDIA is central to the IEEE-P1788 proposal, the correctness of the FTDIA is crucial. Our mechanized proof can give the research community in interval computations much confidence in its correctness. The current version of the FTDIA (in P1788 version 8.0) is slightly different from the theorem proved here. Modifying our proof to reflect this is left as future work.

Acknowledgments

I would like to thank first my supervisor Dr. Ned Nedialkov and my committee member Dr. Wolfram Kahl. Without their help this thesis would have been very difficult to accomplish. Dr. Nedialkov gave me strong support and instructive guidance for my research. He urged me to go ahead when I was indecisive and encouraged me during difficult times. His detailed comments and suggestions helped improve this thesis greatly. Dr. Kahl introduced me to logic, programming languages, and formal specifications, and directed me to formalize and prove the core of the FTDIA. His vision of computer formalized mathematics deeply influenced this thesis.

I would like to thank my committee members: Drs. William Farmer, Spencer Smith, and Emil Sekerinski, and my external examiner Dr. Nathalie Revol for valuable suggestions and numerous questions on this work.

I would also like to thank Dr. John Pryce for suggesting this problem and providing detailed comments on one of the thesis drafts. I am thankful to Dr. Russell O'Connor for teaching me Coq and dependent type theory.

Last but not least, I would like to thank my parents and my wife, Ke Tang, for their support, and my son, Haozhen, for the happiness he brings to me.

This work is supported in part by the National Science and Engineering Council of Canada (NSERC).

Contents

Chapter 1

Introduction

Digital computers use floating-point numbers with finite precision rather than real numbers to perform numerical computations. However, no floating-point system with base two can represent exactly irrational numbers e.g. $\sqrt{2}$, and some simple rational numbers like $1/10$ [9, 19]. Thus, when computers need to manipulate real numbers, they usually carry out correct roundings to replace these real numbers by their nearest floating-point numbers. Roundoff errors are introduced in this process. In addition, scientific computation often begins with initial data obtained from measurements, which are seldom 100% accurate. Measurement errors of these initial data normally result from finite precision of measurement instruments, and have upper bounds supplied by these instruments [26]. If roundoff errors and measurement errors are handled improperly, disasters like the failure of the Patriot missile defense system in the 1991 Gulf War may happen [9].

One way to deal with such errors is to use interval arithmetic. Compared to performing a point version computation in the context of errors and obtaining a result with no guarantee how far it is from the true answer, outwardly rounded interval arithmetic generates an interval rigorously enclosing the exact solution [21, 24, 25].

That is, the computed interval is guaranteed to contain the exact result. This property is proved by the Fundamental Theorem of Interval Arithmetic (FTIA), which is the central theorem of interval arithmetic [24, 25, 31].

However, there is an implicit assumption that input data are always in the domain of a real function in the FTIA. Neglecting to check if this assumption is satisfied may make an interval computation fail. For example, what is the computed interval of the computation $\sqrt{[-1,1]}$? In her presentations [33, 34], Revol points out that the FTIA is not valid without exception handling, which is used to deal e.g. with undefinedness. We may add exception handling similar to the IEEE-754 into interval arithmetic, although this mechanism is expensive. For this reason, decorations, that is, local flags attached to intervals [29, 30, 33], are introduced to interval arithmetic in the IEEE-P1788 proposal draft for the standardization of interval arithmetic. Decorations can help track properties of interval evaluations, such as definedness and continuity of real functions, and do not interrupt the flow of computations [29, 30]. The FTIA plus the decoration system is expanded into the Fundamental Theorem of Decorated Interval Arithmetic (FTDIA), which asserts the correctness of both the property that the FTIA shows and the properties about functions, such as definedness, continuity, etc.

The main contribution of this thesis is the formal proof of the core of the FTDIA using the theorem prover Coq [2]. The traditional, pen-and-paper proof of this theorem is given by John Pryce in [29]. As it is central to the IEEE-P1788 proposal, the correctness of the FTDIA is crucial. A mechanized proof can give the research community in interval computations greater confidence in its correctness. Since there is an ongoing effort to produce an IEEE standard for interval arithmetic, we believe this is a timely result.

Our formal proof is based on a formalization of the decorated interval system in

the IEEE-P1788 proposal draft (version 3.0). Two important contributions resulting from our formalization are as follows.

First, we define the semantics of an interval as *a set of real numbers* (including the empty set), and we do not impose any other restrictions on such a set except that models of this interval can decide if the set is empty or not. For example, an interval need not be closed and bounded, as in traditional interval arithmetic.

Second, our formalization and proof do not rely on specific interval operations: it works with any interval operation that satisfies the requirements for decorated interval library operations.

The traditional proof by J. Pryce of the FTDIA has three base cases. This proof fails in a particular base case when an input interval vector to a function has a component, which is an empty interval, and this component is not used when evaluating this function.

Another base case is when the input is an empty box, that is, at least one of its components is an empty interval, and all components are used when evaluating the function. Trivially, the result is an empty interval, but this case makes the formalization and the proof of the FTDIA in Coq difficult.

In our formalization, we assume we can decide if the corresponding set of an interval is empty or not. Thus, we can screen out the cases of empty boxes, and produce the proof for the case of a non-empty box. This proof captures all non-trivial parts of the FTDIA. We shall refer to the FTDIA with the assumption of non-empty input boxes as FTDIA_NonEmptyBox.

The structure of this thesis is summarized as follows.

- In Chapter 2, we review related work.
- In Chapter 3, we first present the basic concepts of interval arithmetic and the

decoration system, and then introduce the FTIA and FTDIA.

- In Chapter 4, we outline the proof roadmap of the FTDIA_NonEmptyBox. The proof itself is then presented, with explanations, in Chapters 5–12.
- In Chapter 13, we finish with conclusions and discuss future work.

Since all the lemmas and theorems in this thesis are checked by Coq, we only give summaries of the mechanized proofs of some core lemmas and theorems, and omit the descriptions of the rest of the auxiliary lemmas.

Coq overview. Here, we give a brief overview of our formal verification tool. Coq [2, 6, 7] is an interactive theorem prover written mainly in Objective Caml (OCaml) and with some C. The underlying logic in Coq is the *Calculus of Inductive Constructions*, which is a kind of Intuitionistic Logic and is a very expressive variation of typed λ-calculus [8]. Coq has a specification language, called Gallina, in which types or specifications can be represented, and has a command language, called Vernacular, in which proofs can be obtained in a semi-automatic way [8, 10]. Coq supports writing new proof procedures in OCaml and in the domain-specific language Ltac. Such proof procedures cannot lead to invalid proofs.

Coq is an LCF-style (Logic of Computable Functions) theorem prover [28]. It has a small kernel and satisfies *de Bruijn criterion* [10, 17]. This feature makes it a good candidate to verify theorems that are supposed to achieve absolute trust [5, 8]. In addition, types correspond to propositions, and values or members in these types correspond to proofs in Coq. This is called *Curry-Howard correspondence* [8, 35]. The small proof-checking kernel, dependent types, and the availability of good documentation were factors for choosing Coq as our formal verification tool for the FTDIA. Moreover, the Coq community was very helpful during this work.

In Coq, a theorem is normally formalized in the form $[A_1;\ \ldots;\ A_n] \vdash B$, where $A_1, \ldots, A_n, B$ are formulas. This theorem asserts that the goal B can be deduced from the hypotheses $A_1, \ldots, A_n$. To prove a theorem in Coq, we can build proofs for new hypotheses from the proofs of its original hypotheses until one of the new hypotheses is just the goal of the theorem. This is a forward proof. We can also reduce the goal to simpler subgoals until all their subgoals are solved. This is a backward proof. We use both forward and backward proofs in this thesis.

Chapter 2

Related Work

In engineering applications, we often need to compute bounds or check inequalities on numerical expressions and verify whether the results are reliable. Since interval arithmetic is becoming a standard tool to handle this kind of problems, several approaches formalize intervals and develop companion interval libraries for elementary functions in interactive theorem provers. All these tools provide guaranteed formal proofs of numerical problems, and usually demand little knowledge of interactive theorem provers. Hence, their users can formally verify interval arithmetic with minimum effort to learn these theorem provers.

For example, Muñoz et al. [11, 13] present a pragmatic way to certify automatically real number computation using rational bounds in PVS. Melquiond develops the Gappa [12, 14, 15, 23] tool to find bounds on numerical programs handling floating-point or fixed-point arithmetic and generate proof scripts, which can be mechanically checked by Coq. Hölzl [20] introduces an automatic proof method to verify formally real-valued inequalities with bounded variables in Isabelle/HOL. In addition, Zumkeller [3, 36] formalizes Taylor models [27] in Coq using interval arithmetic based on constructive real numbers. This model can be used to provide formal proofs of

numerical inequalities, whose solutions have requirements for precision, performance, and scalability.

We give an overview of these tools here.

2.1 Real Number Calculations in PVS

Muñoz et al. [11, 13] formally certify real number calculations on elementary functions using interval arithmetic in PVS. They formalize the endpoint interval model as a record with two fields: lower bound and upper bound, which are two rational numbers. They define exact bounds for the four basic interval operations (i.e. addition, subtraction, multiplication, and division) and negative, absolute value, square, and power functions. In addition, they establish bounds of adjustable tightness for the square root function and trigonometric functions using series expansions. They also develop a library for rational interval arithmetic in PVS. This library includes several strategies, which first calculate the bound of a real expression, and then prove automatically numerical inequalities on this expression. To obtain tighter bounds, these strategies use a factorization algorithm, splitting techniques, and Taylor's series expansions. However, compared to other tools [15, 36], this tool focuses on automation and pragmatism, rather than computing very tight bounds.

In this interval library, the strategies use built-in real numbers, which belong to an uninterpreted type real, so the fundamental theorem of interval arithmetic (FTIA) is described as a meta theorem, which cannot be formalized and proved in PVS [11, page 10]. However, Muñoz et al. show inclusions in particular cases using a strategy implemented in this library.

2.2 Certification of Bounds on Ranges and Round-off Errors Using Gappa

Melquiond develops a high-level tool Gappa [12, 14, 15, 23] to certify formally tight bounds on ranges and errors in numerical programs dealing with floating-point or fixed-point arithmetic. Gappa has two components [12]: a program written in the C++ language and a companion library for a proof assistant, which performs the proof checking. This program first uses interval arithmetic and forward error analysis to bound ranges of floating-point expressions and their absolute or relative errors. Then, it generates the proof obligations and proof scripts for these bounds, which can be automatically checked by any proof assistant (as claimed) owning a companion library. Now Melquiond provides this library for Coq and HOL-light.

Gappa has several advantages. Its input grammar is similar to the C language, so C code that needs validating can be easily translated into Gappa code. This tool uses a batch of theorems about real arithmetic and rewriting rules to obtain automatically bounds on values of mathematical expressions and their errors. If Gappa cannot find a tight bound for a proof goal, users have to provide extra rewriting rules for its proof engine. Finally, Gappa generates a formal proof for the proof goal, which contains normally thousands to hundreds of thousands of lines [12], and can be automatically checked by a proof assistant like Coq. Thus, its users do not necessarily need to learn about interactive theorem provers. In addition, the facility of generating proof scripts automatically helps its users avoid the tedious work of reproving completely (e.g. if changes in the C code are made).

Besides Gappa, Melquiond also develops Coq.interval [22] library, which includes some tactics with fine-tuning paremeters. These tactics can be used to simplify the proofs of real-valued inequalities for Coq.

2.3 Proving Real-Valued Inequalities by Computation in Isabelle/HOL

Hölzl develops an automatic proof method in Isabelle/HOL [20], which uses interval arithmetic to prove formally real-valued inequalities. He utilizes Taylor's series expansions to approximate most elementary functions and uses Horner's scheme to compute bounds for these elementary functions. This automatic proof method translates real number inequalities into interval arithmetic computations on floating-point numbers and then verifies these computations completely in Isabelle/HOL. All these steps are packed into an automatic tactic with a parameter specifying precision that the computations should achieve.

Chapter 3

Basic Definitions

We introduce concepts and theorems related to interval arithmetic in this chapter. In §3.1, we first define basic concepts about interval arithmetic. Then, we present the Fundamental Theorem of Interval Arithmetic. In §3.2, we first give the reason for introducing the decoration system, and then we present concepts related to the decoration system. Finally, we introduce the Fundamental Theorem of Decorated Interval Arithmetic.

3.1 Interval Arithmetic and the Fundamental Theorem of Interval Arithmetic

Interval arithmetic can help enclose roundoff, truncation and modeling errors. It can be used e.g. to compute rigorous bounds on ranges of real functions [21], help prove existence of solutions to linear and nonlinear systems, bound solutions to initial value problems for ordinary differential equations, and construct global optimization algorithms [16, 21]. In addition, interval arithmetic is also used in mathematical proofs, such as Hales's proof of the Kepler's conjecture [18, 25, 32].

3.1.1 Definitions

Most of the definitions below follow the IEEE-P1788 draft [1, 29].

Interval. For the purpose of the proof of the FTDIA, we consider an interval as an arbitrary set of real numbers in this thesis. Note that we do not assume e.g. closed sets.

Box. A box, or an interval vector, is an n-tuple $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$, whose components $\mathbf{x}_i$ are intervals. Usually, a box $\mathbf{X}$ is identified with the Cartesian product $\mathbf{x}_1 \times \ldots \times \mathbf{x}_n$ of its components. In particular, a vector $x \in \mathbf{X}$, for $x \in \mathbb{R}^n$, means by definition $x_i \in \mathbf{x}_i$ for all $i = 1, \ldots, n$.

Point function. A point function is a (possibly partial) multivariate real function mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$ for some natural numbers $n \geq 0$ and $m > 0$. If we denote the domain of f as $\mathsf{domain}\, f$, we define the range of f over box $\mathbf{X}$ as

$$\text{range}(f, \mathbf{X}) := \{f(x) \mid x \in \mathbf{X} \cap \mathsf{domain}\, f\}.$$

Interval-extension. Given an n-variable scalar point function f, an interval extension of f is a mapping $\mathbf{f}$ from n-dimensional boxes to intervals such that $f(x) \in \mathbf{f}(\mathbf{X})$, whenever $x \in \mathbf{X}$ and $f(x)$ is defined, equivalently $\mathbf{f}(\mathbf{X}) \supseteq \text{range}(f, \mathbf{X})$ for any box $\mathbf{X}$.

Arithmetic operation. A function that is provided by an implementation.

Interval evaluation. An expression is some symbolic form to define a function. When evaluated using intervals, we obtain an interval evaluation.

3.1.2 Fundamental Theorem of Interval Arithmetic (FTIA)

The FTIA is the core theorem of interval arithmetic. It asserts the correctness of interval arithmetic, that is, the computed interval is guaranteed to contain every possible result.

Theorem 3.1.1 (FTIA)**.** *Given a function* $y = f(x_1, \ldots, x_n)$ *and an input box* $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$*, if*

(i) a point evaluation of f *over* $\mathbf{X}$ *obtains* $\text{range}(f, \mathbf{X})$*, and*

(ii) an interval evaluation of $\mathbf{f}$ *over* $\mathbf{X}$ *gives an interval result* $\mathbf{y}$*,*

then $\text{range}(f, \mathbf{X}) \subseteq \mathbf{y}$.

Example. We give an example to illustrate an interval computation and the fact that the computed interval includes the exact result.

In this example, we use the standard definition of intervals, that is, an interval $\mathbf{x} = [\underline{x}, \overline{x}]$ is a set of real numbers given by $[\underline{x}, \overline{x}] = \{x \in \mathbb{R} : \underline{x} \leq x \leq \overline{x}\}$. The basic interval arithmetic operations can be evaluated by these formulae:

$$
\begin{aligned}
\mathbf{x} + \mathbf{y} &= [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \\
\mathbf{x} - \mathbf{y} &= [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \\
\mathbf{x} \times \mathbf{y} &= [\min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}, \max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}] \\
1/\mathbf{x} &= [1/\overline{x}, 1/\underline{x}] \quad \textit{if } \underline{x} > 0 \text{ or } \overline{x} < 0 \\
\mathbf{x}/\mathbf{y} &= \mathbf{x} \times 1/\mathbf{y}
\end{aligned}
$$

Now, given the function $f(x) = x(x-1)$, if we replace x with the interval $[0, 1]$

and perform interval evaluation, we obtain,

$$\mathbf{f}(\mathbf{X}) \quad = \quad [0,1]([0,1]-[1,1]) = [0,1][-1,0] = [-1,0],$$

which contains the range $[-0.25, 0]$ of f over $[0, 1]$.

3.2 Decorated Interval Arithmetic

Decorations are data attached to intervals [29]. They act not like global flags, such as IEEE-754 flags for division by zero etc., but like local flags, which present the properties of how intervals are computed rather than the properties of intervals themselves. Thus, the flow of computation does not need to be interrupted, while information about an evaluation is collated. Such information can be inspected after a computation terminates. Compared with the global exception handling mechanism, using decorations has several advantages as follows [30].

- First, some interval algorithms, say graphics rendering algorithms, need to grasp knowledge about if a real function is defined or continuous on a box.
- In addition, several functions can be computed in parallel for their decorations do not interfere with one another. This feature can be used in Single Instruction Multiple Data (SIMD) machines.
- Finally, code that is executed, but whose results are not used is always handled correctly for its decoration does not impact the final result [30].

3.2.1 Decoration System

We consider the decoration system of the draft 3.0 of IEEE-P1788.

Decorations. The decoration system, which we consider, has five distinct decorations: saf, def, con, emp, and ill. Each decoration represents a set of pairs $(f, \mathbf{X})$, where f is a function, and $\mathbf{X}$ is a box. If $(f, \mathbf{X}) \in d$, then the predicate $p_d(f, \mathbf{X})$ is valid as follows:

- $p_{\mathsf{saf}}(f, \mathbf{X})$: $\mathbf{X}$ is a nonempty subset of the domain of f, and f is continuous and bounded on $\mathbf{X}$;
- $p_{\mathsf{def}}(f, \mathbf{X})$: $\mathbf{X}$ is a nonempty subset of the domain of f;
- $p_{\mathsf{con}}(f, \mathbf{X})$: always true;
- $p_{\mathsf{emp}}(f, \mathbf{X})$: $\mathbf{X}$ is disjoint from the domain of f; and
- $p_{\mathsf{ill}}(f, \mathbf{X})$: the domain of f is empty.

Since we can regard decorations as sets, the decorations are partially ordered by strength as

$$\mathsf{saf} \subseteq \mathsf{def} \subseteq \mathsf{con} \quad \text{and} \quad \mathsf{ill} \subseteq \mathsf{emp} \subseteq \mathsf{con}, \tag{3.1}$$

where saf and ill are strongest in their respective orderings, and con is the weakest in both orderings.

The decorations are also given a total quality ordering as

$$\mathsf{ill} < \mathsf{emp} < \mathsf{con} < \mathsf{def} < \mathsf{saf}, \tag{3.2}$$

where ill is the worst, and saf is the best.

Strongest decoration. For a function f and a box $\mathbf{X}$, we define $\mathrm{dec}(f, \mathbf{X})$, the decoration of f over $\mathbf{X}$, to be the strongest decoration d, for which $p_d(f, \mathbf{X})$ is true

as follows:

$$\mathrm{dec}(f,\mathbf{X}) = \begin{cases} \mathsf{saf} & \text{if } p_{\mathsf{saf}}(f,\mathbf{X}) \text{ holds;} \\ \mathsf{def} & \text{else if } p_{\mathsf{def}}(f,\mathbf{X}) \text{ holds;} \\ \mathsf{ill} & \text{else if } p_{\mathsf{ill}}(f,\mathbf{X}) \text{ holds;} \\ \mathsf{emp} & \text{else if } p_{\mathsf{emp}}(f,\mathbf{X}) \text{ holds;} \\ \mathsf{con} & \text{otherwise.} \end{cases}$$

The $\mathrm{dec}(f,\mathbf{X})$ is always unique for each $(f,\mathbf{X})$, as it cannot have both a decoration with good quality, i.e. saf or def and a decoration with bad quality, i.e. ill or emp.

Decorated interval. We define a decorated interval as a pair $(\mathbf{x}, d)$, where $\mathbf{x}$ is an interval and d is a decoration.

Decorated interval containment order $\subseteq$. For any two decorated intervals $(\mathbf{x},\ d)$ and $(\mathbf{x}',\ d')$,

$$(\mathbf{x},\ d) \subseteq (\mathbf{x}',\ d') \quad \text{iff} \quad \mathbf{x} \subseteq \mathbf{x}' \text{ and } d \subseteq d'. \tag{3.3}$$

Requirements for decorated interval library operations. Given a function $f(x_1,\ldots,x_n)$, if each intermediate step of evaluating f depends on the input vector $(x_1,\ldots,x_n)$, then each step can be viewed as a function of this vector. Thus, one has, at each step, a (theoretical) intermediate function $u(x_1,\ldots,x_n)$ and a (computed) intermediate interval $\mathbf{u}$ in interval evaluation. The function f can be defined in terms of previous intermediate functions, say $u_1,\ldots,u_k$, by

$$f(x_1,\ldots,x_n) = \phi\big(u_1(x_1,\ldots,x_n),\ldots,u_k(x_1,\ldots,x_n)\big),$$

where ϕ is a k-argument arithmetic operation, and the corresponding interval $\mathbf{y}$ containing $f(x_1, \ldots, x_n)$ is computed by applying an interval version of ϕ to the corresponding intermediate intervals $\mathbf{u}_1, \ldots, \mathbf{u}_k$, where $u_i(x_1, \ldots, x_n) \in \mathbf{u}_i$.

Given a function f, if its decorated interval version $\mathbf{f}$ is a decorated interval library operation, then it should satisfy

$$\mathbf{y} \supseteq \operatorname{range}(\phi, \mathbf{U}), \quad \text{and} \tag{3.4}$$

$$d = \min(c_0, c_1, \ldots, c_k), \tag{3.5}$$

where $c_0 \supseteq \operatorname{dec}(\phi, \mathbf{U})$ and c_i is the decoration of $\mathbf{u}_i$.

Decorated interval evaluation. When the expression defining a function is evaluated according to (3.4–3.5), we obtain a decorated interval evaluation of this function.

3.2.2 Fundamental Theorem of Decorated Interval Arithmetic

Theorem 3.2.1 (FTDIA)**.** *Let a function $y = f(x_1, \ldots, x_n)$ be evaluated in decorated interval mode, and let the interval parts of its inputs be given intervals $\mathbf{x}_1, \ldots, \mathbf{x}_n$, defining the input box $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$. Suppose*

(i) each $\mathbf{x}_i$ is given the decoration $d_i = \mathsf{saf}$ if $\mathbf{x}_i$ is nonempty and bounded, $d_i = \mathsf{def}$ if $\mathbf{x}_i$ is nonempty and unbounded, and $d_i = \mathsf{emp}$ if $\mathbf{x}_i$ is empty;

(ii) the decorated interval library operations satisfy (3.4) *and* (3.5)*.*

Then the decorated interval result $(\mathbf{y}, d)$ encloses the decorated range $\big(\operatorname{range}(f, \mathbf{X}), \operatorname{dec}(f, \mathbf{X})\big)$ in the sense of (3.3)*.*

The FTDIA not only asserts that the computed interval includes the actual result, but also states that the decoration attached in the final interval can never make a

false assertion about definedness, continuity, etc.

Example. We give a simple example to illustrate a decorated interval computation. We want to compute an enclosure and a decoration of $f(x, y) = \sqrt{x - y} + x$, where $x \in [1, 4]$, and $y \in [-1, 2]$. We initialize both $\mathbf{x}$ and $\mathbf{y}$ as saf, meaning nonempty and bounded, and compute the enclosure and decoration of $f(x, y)$ as follows:

$$\begin{aligned}
\mathbf{f}(\mathbf{X}) &= \sqrt{([1,3],\mathsf{saf}) - ([-1,2],\mathsf{saf})} + ([1,3],\mathsf{saf}) \\
&= \sqrt{([-1,4],\mathsf{saf})} + ([1,3],\mathsf{saf}) \\
&= ([0,2],\mathsf{con}) + ([1,3],\mathsf{saf}) \\
&= ([1,5],\mathsf{con}).
\end{aligned}$$

Since subtraction is everywhere saf, $([1,3],\mathsf{saf}) - ([-1,2],\mathsf{saf}) = ([-1,4],\mathsf{saf})$. However, $[-1,4]$ is not a subset of the domain $[0,+\infty)$ of the square root function, and it is also not disjoint from $[0,+\infty)$, thus $\sqrt{([-1,4],\mathsf{saf})} = ([0,2],\mathsf{con})$. Finally, although addition is everywhere saf, $([0,2],\mathsf{con}) + ([1,3],\mathsf{saf}) = ([1,5],\mathsf{con})$ for the decoration of one of its arguments is con, which is worse than saf.

Chapter 4

Proof Outline

If we evaluate a function f on a nonempty input box $\mathbf{X}$ in decorated interval mode, and obtain the result $(\mathbf{y}, d)$, to prove the FTDIA_NonEmptyBox, we need to show

(i) the FTIA, that is, $\mathbf{y} \supseteq \text{range}\ (f, \mathbf{X})$, and also

(ii) the decoration enclosure property, that is, $d \supseteq \text{dec}(f, \mathbf{X})$.

The flow of our proof is shown in Figure 4.1. We outline how the next chapters refer to this figure.

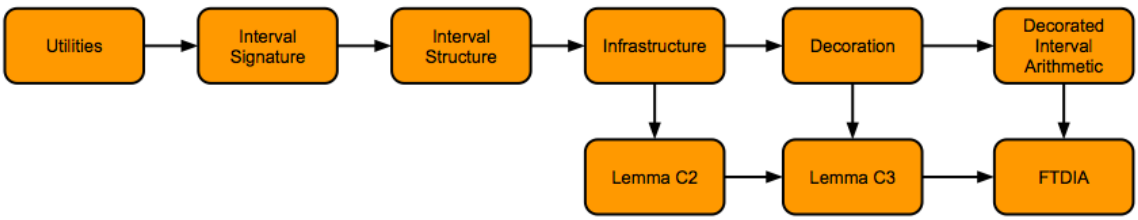

Figure 4.1: Proof Roadmap

In Chapter 5, we define some utilities with generic types, such as partial function, set, vector etc., and prove their properties. These utilities and properties help formalize and prove the FTDIA_NonEmptyBox.

In Chapter 6, we give the specification of the intervals used in our proof system, and introduce an abstract interval model into our proof system.

In Chapter 7, we define some operations and predicates on real numbers, and prove their properties. All these definitions and properties are closely related to the formalization and proof of the FTDIA_NonEmptyBox and its auxiliary lemmas.

In Chapter 8, we first introduce the decoration system. Then, we relate the predicates defined in this chapter to the predicates defined in Chapter 7.

In Chapter 9, we present Lemma C2 [29]. We formalize the three cases of this lemma and give the essential parts of their mechanized proofs.

In Chapter 10, we first describe Lemma C3 [29]. Then, we formalize and prove two auxiliary lemmas C3_pCmin_NonEmptyBox and decfX_StrongerThan_cMin, which can be combined to prove Lemma C3.

In Chapter 11, we introduce and formalize some concepts about the decorated interval system, and prove their properties, which are used to prove the theorem FTDIA_NonEmptyBox.

In Chapter 12, we first formalize and prove the FTIA. Then we formalize and prove the FTDIA_NonEmptyBox, which asserts that the FTDIA is correct when an input box is not empty.

Chapter 5

Utilities

In this chapter, we define functions and predicates with generic types and show their properties. These utilities and properties underlie the formalization and proofs of the FTDIA_NonEmptyBox and its auxiliary lemmas.

The functions and predicates, which deal with definedness and undefinedness, and their properties, are defined and proved in §5.1. The facilities and properties related to set theory are implemented and shown in §5.2. The concepts and properties about partial functions are introduced and proved in §5.3. Vector operations and properties are presented and shown in §5.4. Other utilities, which are related to real numbers, are introduced in §7.

We use the following conventions.

- Capitals A, B, C, and T denote types.
- $a : A$ denotes that a is of type A.
- $A \nrightarrow B$ denotes a partial function from A to B. For example, for the square root function, we write $\sqrt{x} : \mathbb{R} \nrightarrow \mathbb{R}$.

5.1 Utilities for Checking the Result of a Partial Function

In this section, we first explain how Coq deals with partial functions and undefinedness. Then, we define functions and predicates about definedness and undefinedness and show their properties. These definitions and lemmas supplement the utilities provided by Coq and facilitate our proof.

5.1.1 Basic Concepts

We start with an explanation of some basic concepts.

- In Coq, *option* is used to model partial functions. Namely, for any types A and B, the partial function type $A \nrightarrow B$ is modeled as $A \rightarrow option\ B$.
- If A is a type, then $option\ A$ is the extension of A with an extra element *None*, which is used to denote an undefined result of a partial function. In addition, to extend $a : A$ to an element in $option\ A$, we use $Some\ a$.
- For any function $f : A \rightarrow option\ B$,
  - if f is defined at $a : A$, then there exists some $b : B$, such that $f\ a = Some\ b$
  - if f is undefined at $a : A$, then $f\ a = None$

5.1.2 Definitions and Lemmas

We list definitions and lemmas along with short explanations.

isSome Given $x : option\ A$, then $isSome\ x$ has a proof if $x = Some\ a$, where $a : A$, or $isSome\ x$ has no proof if $x = None$.

```
Definition isSome {A : Type} (x : option A) : Prop :=
 match x with
   | None ⇒ False
   | Some a ⇒ True
 end.
```

optionCase shows that either $x : option\ A$ is $None$, or there exists an $a : A$ such that $x = Some\ a$.

```
Lemma optionCase :
 ∀ {A : Type} (x : option A), x = None ∨ ∃ a : A , x = Some a.
```

optionBind defines that if $x = None$, $optionBind\ f\ x$ returns $None$. Otherwise $x = Some\ a$, where $a : A$, $optionBind\ f\ x$ returns $f\ a$.

```
Definition optionBind {A B : Type} (f : A → option B)
                                   (x : option A) : option B :=
 match x with
   | None ⇒ None
   | Some a ⇒ f a
 end.
```

optionBind_Some shows that, if $optionBind\ f\ x = Some\ b$, then there exists an $a : A$, such that $x = Some\ a$ and $f\ a = Some\ b$.

```
Lemma optionBind_Some :
 ∀ {A B : Type} {f : A → option B} {x : option A} {b : B},
   optionBind f x = Some b → ∃ a : A, x = Some a ∧ f a = Some b.
```

optionMap2 has three explicit arguments, a binary function $f : A \to B \to C$, $x :$ *option* A, and $y :$ *option* B. The result of applying *optionMap2* to its arguments is as follows.

- If $x =$ *None* or $y =$ *None*, then *optionMap2* $f\ x\ y =$ *None*.
- Otherwise, $x =$ *Some* a and $y =$ *Some* b, where $a : A$ and $b : B$, and *optionMap2* $f\ x\ y =$ *Some* c, where $c : C$, *Some* $c :$ *option* C, and $c = f\ a\ b$.

```
Definition optionMap2 {A B C : Type} (f : A → B → C)
                                     (x : option A) (y : option B) : option C :=
  match x with
    | None ⇒ None
    | Some a ⇒ option_map (f a) y
  end.
```

Here, *option_map* is a built-in function in Coq. It has two arguments, a unary function $f : A \to B$, and $o :$ *option* A. The result of applying *option_map* to its arguments is as follows.

- If $o :$ *None*, then *option_map* $f\ o =$ *None*.
- Otherwise, $o =$ *Some* a, where $a : A$, then *option_map* $f\ o =$ *Some* b, where $b : B$, *Some* $b :$ *option* B, and $b = f\ a$.

optionMap2_eq_Some shows that, given a function $f : A \to B \to C$, $x :$ *option* A, $y :$ *option* B, and $c : C$, if *optionMap2* $f\ x\ y =$ *Some* c, then there exist $a : A$ and $b : B$, such that $x =$ *Some* a, $y =$ *Some* b, and $c = f\ a\ b$.

```
Lemma optionMap2_eq_Some:
  ∀ {A B C : Type} {f : A → B → C} {x : option A} {y : option B} {c : C},
```

$optionMap2\ f\ x\ y = Some\ c \rightarrow$

$\exists\ a : A, x = Some\ a \wedge \exists\ b : B, y = Some\ b \wedge c = f\ a\ b.$

isSome_extract Given $x :\ option\ A$, if the proposition $isSome\ x$ has a proof x_some, that is, there exists an $a : A$ such that $x = Some\ a$, then $isSome\_extract$ extracts the a from x_some.

```
Definition isSome_extract : ∀ {A : Type} {x : option A} (x_some : isSome x), A.
```

isSome_eq Given that $isSome\ x$ has a proof x_some, then $isSome\_eq$ shows that $x = Some\ (isSome\_extract\ x\_some)$, that is, there exists an $a : A$, such that $a = isSome\_extract\ x\_some$ and $x = Some\ a$.

```
Lemma isSome_eq :
  ∀ {A : Type} {x : option A} (x_some : isSome x),
    x = Some (isSome_extract x_some).
```

Some_injective proves that the constructor $Some$ is injective.

```
Lemma Some_injective : ∀ {A : Type} {x y : A}, Some x = Some y → x = y.
```

5.2 Set Theory

In this section, we formalize some concepts from set theory, such as power set, subset, empty set and nonempty set, and some lemmas, such as the transitivity property of the subset relation. Since the different definitions for the same notion can give diffent views on this concept [8], we redefine some of the concepts, which already exist in the Coq library, to adapt them to our formalization and proofs.

set T is defined as the power set of type T.

```
Definition set (T : Type) := Ensemble T.
```

subset defines the subset relation.

```
Definition subset {T : Type} : set T → set T → Prop := Included T.
```

subsetTrans shows that the subset relation is transitive.

```
Lemma subsetTrans :
 ∀ {T : Type} {X Y Z : set T},
   subset X Y → subset Y Z → subset X Z.
```

emptySet Given a set X, then *emptySet* X asserts that X is empty.

```
Definition emptySet {T : Type} (X : set T) : Prop := ∀ x : T, not (X x).
```

nonEmptySet Given a set X, then *nonEmptySet* X asserts that X is nonempty.

```
Definition nonEmptySet {T : Type} (X : set T) : Prop := ∃ x : T, X x.
```

intersection Given two sets X and Y, then *intersection* X Y generates their intersection.

```
Definition intersection {T : Type} (X Y: set T) : set T :=
 fun x: T ⇒ X x ∧ Y x.
```

disjoint Given two sets X and Y, then *disjoint* X Y asserts that the two sets are disjoint.

```
Definition disjoint {T : Type} (X Y: set T) : Prop :=
 ∀ x : T, not ((intersection X Y) x).
```

setVecProd Given a vector $X = (X_1, \ldots, X_n)$, where X_i is a subset of T, and a vector $x = (x_1, \ldots, x_n)$, where $x_i \in T$, then *setVecProd* X x asserts that $x \in X_1 \times \ldots \times X_n$, which is defined as (meaning) $x_i \in X_i$ for $i = 1, \ldots, n$.

```
Fixpoint setVecProd {T : Type} {n : nat} (X : Vec (set T) n)
                                  (x : Vec T n) {struct X} : Prop :=
```

```
(match X with
   | nil ⇒ fun _ ⇒ True
   | cons X_head n' X_tail ⇒
                fun x : Vec T (S n') ⇒
                 X_head (hd x) ∧ setVecProd X_tail (tl x)
 end) x.
```

setVecProd_cons Given two vectors $X = (X_1, \ldots, X_n)$ and $x = (x_1, \ldots, x_n)$, then *setVecProd_cons* shows that $x \in X$ iff $x_i \in X_i$ for all $i = 1, \ldots, n$.

```
Lemma setVecProd_cons:
 ∀ {T : Type} {n : nat}
   {X_head : set T} {X_tail : Vec (set T) n}
   {x_head : T} {x_tail : Vec T n},
 setVecProd (cons (set T) X_head n X_tail) (cons T x_head n x_tail) =
 (X_head x_head ∧ setVecProd X_tail x_tail).
```

5.3 Partial Functions and Their Properties

In this section, we introduce a partial function type $T^n \nrightarrow T$, define *domain* function, *range* function, and *emptyDomain* predicate, and show the well-definedness and extensional equality of partial functions with the type $T^n \nrightarrow T$.

partialFunType defines a partial function type $T^n \nrightarrow T$.

```
Definition partialFunType (T : Type) (n : nat): Type := Vec T n → option T.
```

domain Given a partial function $f : T^n \nrightarrow T$, then *domain f* returns the domain of f, that is, the set of vectors on which f is defined.

```
Definition domain {T : Type} {n : nat}
                  (f : partialFunType T n) : set (Vec T n) :=
  fun x : Vec T n ⇒ isSome (f x).
```

emptyDomain asserts that a partial function $f : T^n \nrightarrow T$ has an empty domain.

```
Definition emptyDomain {T : Type} {n : nat}
                       (f : partialFunType T n) : Prop :=
  ∀ (x : Vec T n), f x = None.
```

range Given a partial function $f : T^n \nrightarrow T$ and an n-vector $X = (X_1, \ldots, X_n)$, where X_i is a subset of T for all $i = 1, \ldots, n$, then *range f X* returns the range of f on $X_1 \times \ldots \times X_n$.

```
Definition range {T : Type} {n : nat} (f : partialFunType T n)
                 (X : Vec (set T) n) : set T :=
  fun y : T ⇒ ∃ x : Vec T n, setVecProd X x ∧ Some y = f x.
```

range_argExt shows that, if two partial functions $f : T^n \nrightarrow T$ and $g : T^n \nrightarrow T$ satisfy $f\ x = g\ x$ for each argument $x \in T^n$, then the range of f on X is equal to the range of g on X, where $X = (X_1, \ldots, X_n)$ and $X_1 \times \ldots \times X_n$ can be any subset of T^n.

```
Lemma range_argExt:
  ∀ {T : Type} {n : nat} {f g : partialFunType T n} (X : Vec (set T) n),
    (∀ x : Vec T n, f x = g x) → range f X = range g X.
```

partialFun_wellDefFun shows that a partial function $f : T^n \nrightarrow T$ is well defined. That is, if $f\ v = Some\ x$ and $f\ v = Some\ y$, then $x = y$.

```
Lemma partialFun_wellDefFun :
  ∀ {T : Type} {n : nat} {f : partialFunType T n} {v : Vec T n} {x y: T},
```

$f\ v = Some\ x \to f\ v = Some\ y \to x = y.$

fi_in_f Given a vector of functions $f = (f_1, \ldots, f_n)$, then *fi_in_f* shows that f_i is a component of f, for any $i = 1, \ldots, n$.

```
Lemma fi_in_f :
  ∀ {A B : Type} {n : nat} (f : Vec (A → B) n) (i : Fin.t n), Vector.In (nth f i) f.
```

- *Fin.t n* denotes the finite set $\{1, 2, \ldots, n\}$
- *Vector.In* (*nth f i*) *f* means that f_i is a component of f

5.4 Vector Operations and Their Properties

In §5.4.1, we define two abbreviated names for the finite set type and vector type in Coq, and show two rewriting rules commonly used for proving properties on vectors. In §5.4.2, we introduce our own scanning predicates *vecAll* and *vecAll2*, which are more suitable for our formalization and proofs. In §5.4.3, besides showing several rewriting rules for the built-in mapping functions *map*, *map2* in Coq, we develop two new mapping functions *mapApp* and *mapBoxApp*. In §5.4.4, we define a generalized mapping function *vecAll2_map*. In §5.4.5, we show several rewriting rules for the projection function *nth*. In §5.4.6, we introduce the function *vecOp2OpVec* to deal with vectors resulting from partial function evaluation. In §5.4.7, we define the predicates *vecSetIncluded* and *vecSetIncluded2* about inclusion relation between two vectors, whose components are sets.

5.4.1 Definition and Rewriting Rules

Fin is an abbreviated name for the finite set type in Coq.

`Definition` *Fin* := *Fin.t*.

Vec is an abbreviated name for the vector type in Coq.

`Definition` *Vec* := *Vector.t*.

V0_eq shows that a vector of length 0 is *nil*.

`Lemma` *V0_eq* : ∀ {*A* : `Type`} (*v* : *Vec A* 0), *v* = *nil A*.

VSn_eq shows that a vector $v : A^{n+1}$ is equal to (v_head, v_tail), where $v\_head : A$ and $v\_tail : A^n$.

`Lemma` *VSn_eq* :

∀ {*A* : `Type`} {*n* : *nat*} (*v* : *Vec A* (*S n*)), *v* = *cons A* (*hd v*) _ (*tl v*).

In the above,

- *hd v* denotes *v_head*
- *tl v* denotes *v_tail*
- *cons A* (*hd v*) _ (*tl v*) denotes (v_head, v_tail), where "_" denotes the length of *v_tail*, which is a subterm that can be automatically inferred by Coq

5.4.2 Scanning Predicates

vecAll Given a predicate p and a vector $x = (x_1, \ldots, x_n)$, *vecAll p x* asserts that each component of x satisfies p.

`Fixpoint` *vecAll* {*T* : `Type`} {*n* : *nat*} (*p* : *T* → `Prop`) (*x* : *Vec T n*) : `Prop` :=

```
match x with
  | nil ⇒ True
  | cons x_head n' x_tail ⇒ p x_head ∧ vecAll p x_tail
end.
```

vecAll_cons shows that given a predicate p and a vector $x = (x_1, \ldots, x_{n+1})$, *vecAll p x* is the same as $(p\ x_1) \wedge \ldots \wedge (p\ x_{n+1})$.

```
Lemma vecAll_cons :
 ∀ {T : Type} {n : nat} (p : T → Prop) (x_head : T) (x_tail : Vec T n),
   vecAll p (cons T x_head n x_tail) = (p x_head ∧ vecAll p x_tail).
```

vecAll_nth shows that given a predicate p and a vector $x = (x_1, \ldots, x_n)$, *vecAll p x* implies that x_i satisfies p for any $i = 1, \ldots, n$.

```
Lemma vecAll_nth :
 ∀ {A : Type} {n : nat} {p : A → Prop} {x : Vec A n} (i : Fin.t n),
 vecAll p x → p (nth x i).
```

vecAll_const shows that given an $x : T$, if $p\ x$ is true, then each component of the n-vector $(x, \ldots, x)$, whose components are all equal to x, also satisfies p.

```
Lemma vecAll_const :
 ∀ {T : Type} {n : nat} {p : T → Prop} {x : T}, p x → vecAll p (const x n).
```

---

const x n generates an n-ary vector, whose components are all equal to x

vecAll2 Given two vectors $x = (x_1, \ldots, x_n) : A^n$ and $y = (y_1, \ldots, y_n) : B^n$, and a predicate p, *vecAll2 x y* asserts that each (x_i, y_i), for $i = 1, \ldots, n$, satisfies p.

```
Fixpoint vecAll2 {A B: Type} {n : nat} (p : A → B → Prop)
                        (x : Vec A n) (y : Vec B n) : Prop :=
 (match x with
```

```
  | nil ⇒ fun _ ⇒ True
  | cons x_head n' x_tail ⇒
        fun y' : Vec B (S n') ⇒
          (p x_head (hd y')) ∧ vecAll2 p x_tail (tl y')
  end) y.
```

vecAll2_cons shows that, given two vectors $x = (x_1, \ldots, x_{n+1}) : A^{n+1}$ and $y = (y_1, \ldots, y_{n+1}) : B^{n+1}$ and a predicate p, *vecAll2 p x y* holds iff each pair (x_i, y_i), for $i = 1, \ldots, n+1$, satisfies p.

```
Lemma vecAll2_cons :
  ∀ {A B: Type} {n : nat} (p : A → B → Prop)
    (x_head : A) (y_head : B) (x_tail : Vec A n) (y_tail : Vec B n),
  vecAll2 p (cons A x_head n x_tail) (cons B y_head n y_tail) =
  (p x_head y_head ∧ vecAll2 p x_tail y_tail).
```

vecAll2_nth shows that, given two vectors $(x_1, \ldots, x_n) : A^n$, $(y_1, \ldots, y_n) : B^n$, and an index i, *vecAll2 p x y* implies that each pair (x_i, y_i) satisfies p, for $i = 1, \ldots, n$.

```
Lemma vecAll2_nth :
  ∀ {A B: Type} {n : nat} {p : A → B → Prop}
    {x : Vec A n} {y : Vec B n} (i : Fin.t n),
  vecAll2 p x y → p (nth x i) (nth y i).
```

vecAll2_from_nth shows that, given two vectors $(x_1, \ldots, x_n) : A^n$ and $(y_1, \ldots, y_n) : B^n$, if any (x_i, y_i) for $i = 1, \ldots, n$ satisfies p, *vecAll2 p x y* holds.

```
Lemma vecAll2_from_nth :
  ∀ {A B: Type} {n : nat} {p : A → B → Prop} {x : Vec A n} {y : Vec B n},
    (∀ (i : Fin.t n), p (nth x i) (nth y i)) → vecAll2 p x y.
```

5.4.3 Mapping Functions

map_cons shows that applying a unary function $f : A \to B$ to a vector $(x_1, \ldots, x_{n+1}) : A^{n+1}$ is equal to applying f to each component x_i, for $i = 1, \ldots, n+1$. That is, $map\, f\, (x_1, \ldots, x_{n+1}) = \big(f(x_1), \ldots, f(x_{n+1})\big)$, where $\big(f(x_1), \ldots, f(x_{n+1})\big) : B^{n+1}$.

```
Lemma map_cons :
  ∀ {A B : Type} {n : nat} {f : A → B} {x_head : A} {x_tail : Vec A n},
  map f (cons A x_head n x_tail) = cons B (f x_head) n (map f x_tail).
```

The built-in function *map* takes a unary function $f : A \to B$ and a vector $(x_1, \ldots, x_n) : A^n$ to produce a vector of function applications. That is,
$map\, f\, (x_1, \ldots, x_n) = \big(f(x_1), \ldots, f(x_n)\big) : B^n$.

map2_cons shows that applying a binary function $f : A \to B \to C$ to two vectors $(x_1, \ldots, x_{n+1}) : A^{n+1}$ and $(y_1, \ldots, y_{n+1}) : B^{n+1}$ is the same as applying f to each pair (x_i, y_i), for $i = 1, \ldots, n+1$. That is, $map2\ f\ (x_1, \ldots, x_{n+1})\,(y_1, \ldots, y_{n+1}) = \big(f(x_1, y_1), \ldots, f(x_{n+1}, y_{n+1})\big)$, where $\big(f(x_1, y_1), \ldots, f(x_{n+1}, y_{n+1})\big) : C^{n+1}$, and *map2* is a built-in function.

```
Lemma map2_cons :
  ∀ {A B C: Type} {n : nat} (f : A → B → C)
    (x_head : A) (x_tail : Vec A n) (y_head : B) (y_tail : Vec B n),
  map2 f (cons A x_head n x_tail) (cons B y_head n y_tail) =
   cons C (f x_head y_head) n (map2 f x_tail y_tail).
```

mapApp applies a vector of functions to a single argument. Specifically, it applies each component $f_i : A \to B$ of $f = (f_1, \ldots, f_n)$ to an argument x and returns a vector $\big(f_1(x), \ldots, f_n(x)\big) : B^n$.

```
Fixpoint mapApp {A B : Type} {n : nat}
```

$$(f : Vec\ (A \to B)\ n)\ (x : A) : Vec\ B\ n :=$$

```
match f with
  | nil ⇒ nil B
  | cons f_head n' f_tail ⇒ cons B (f_head x) n' (mapApp f_tail x)
end.
```

mapApp_cons shows that applying a vector of functions $f = (f_1, \ldots, f_{n+1})$ to an argument x is the same as applying each component f_i to x. That is, $mapApp\ (f_1, \ldots, f_{n+1})\ x = \big(f_1(x), \ldots, f_{n+1}(x)\big)$, where $\big(f_1(x), \ldots, f_{n+1}(x)\big) : B^{n+1}$.

```
Lemma mapApp_cons :
∀ {A B : Type} {n : nat}
  {f_head : A → B} {f_tail : Vec (A → B) n} {x : A},
mapApp (cons (A → B) f_head n f_tail) x =
 cons B (f_head x) n (mapApp f_tail x).
```

mapApp_map Given a binary function $f : A \to B \to C$, a vector $(v_1, \ldots, v_n) : A^n$, and $y : B$, *mapApp_map* shows that the result of the first evaluation path

- $map\ f\ v$, that is, $f(v_1, \ldots, v_n) = \big(f(v_1), \ldots, f(v_n)\big) : (B \to C)^n$
- $mapApp\ (map\ f\ v)\ y$, that is, $\big(f(v_1), \ldots, f(v_n)\big)\, y = \big((f\ v_1\ y), \ldots, (f\ v_n\ y)\big) : C^n$

and that of the second evaluation path

- `fun` $x \Rightarrow f\ x\ y$, that is, $(\lambda\, x.\ f\ x\ y) : A \to C$
- map (`fun` $x \Rightarrow f\ x\ y$) v, that is, $(\lambda\, x.\ f\ x\ y)(v_1, \ldots, v_n) = \big((f\ v_1\ y), \ldots, (f\ v_n\ y)\big) : C^n$

are equal.

```
Lemma mapApp_map :
 ∀ {A B C : Type} {n : nat} {f : A → B → C} {v : Vec A n} {y : B} ,
   mapApp (map f v) y = map (fun x ⇒ f x y) v.
```

mapBoxApp Given a vector of partial functions $f = (f_1, \ldots, f_k)$, where $f_i : T^n \nrightarrow T$ for $i = 1, \ldots, k$, and an n-vector of sets $V = (V_1, \ldots, V_n)$, where $V_j \subseteq T$ for $j = 1, \ldots, n$, *mapBoxApp* applies f on V, and returns a vector of ranges. That is, *mapBoxApp* $f\ V = \big((range\ f_1\ V), \ldots, (range\ f_k\ V)\big)$.

```
Definition mapBoxApp {T : Type} {n k : nat} (f : Vec (partialFunType T n) k)
                                            (V : Vec (set T) n) : Vec (set T) k :=
 mapApp (map range f) V.
```

mapBoxApp_cons shows that applying a vector of partial functions $f = (f_1, \ldots, f_{k+1})$ to an n-vector of sets $V = (V_1, \ldots, V_n)$ is the same as applying each function $(range\ f_i)$, for $i = 1, \ldots, k+1$, to V. That is, *mapBoxApp* $f\ V = \big((range\ f_1\ V), \ldots, (range\ f_{k+1}\ V)\big)$.

```
Lemma mapBoxApp_cons :
 ∀ {T : Type} {n k : nat}
   (f_head : partialFunType T n) (f_tail : Vec (partialFunType T n) k)
   (V : Vec (set T) n),
 mapBoxApp (cons (partialFunType T n) f_head k f_tail) V =
  cons (set T) (range f_head V) k (mapBoxApp f_tail V).
```

5.4.4 Generalized Mapping Functions

tabulate constructs a vector with type T^n, given a function from indices belonging to *Fin* n to elements with type T. This is an isomorphism between functions of type

Fin $n \to T$ and vectors of type *Vec* T n, and its inverse is the function *nth*. One could also say that *tabulate* is the inverse of the function *nth* considered as mapping vectors of type *Vec* T n to functions of type *Fin* $n \to T$.

```
Fixpoint tabulate {T : Type} {n : nat} : (Fin n → T) → Vec T n :=
  match n with
    | O ⇒ fun _ ⇒ nil T
    | S n' ⇒ fun f ⇒ cons T (f Fin.F1) n' (tabulate (compose f Fin.FS))
  end.
```

Fin_zero_elim is a special case for empty type elimination: if we have an element of *Fin* 0, then anything is true.

```
Definition Fin_zero_elim : ∀ {T : Type} (i : Fin O), T.
```

vecAll2_map can be thought of the generalization of *map2*. *vecAll2_map* takes a function $q : \forall\ (a : A)\ (b : B),\ p\ a\ b \to T$ and two n-vectors $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$, such that each (x_i, y_i) satisfies the predicate p for $i = 1, \ldots, n$, then returns an n-vector of type T^n.

We define *vecAll2_map* via *vecAll2_map_F* and *tabulate*.

vecAll2_map_F

Given

- a function $q : \forall\ (a : A)\ (b : B),\ p\ a\ b \to T$, which takes two arguments $a : A$ and $b : B$, and also a proof that the pair (a, b) satisfies the predicate p, and returns an element of type T;
- two vectors $x = (x_1, \ldots, x_n) : A^n$ and $y = (y_1, \ldots, y_n) : B^n$; and
- a proof *va* of *vecAll2* p x y, that is (x_i, y_i), for any $i = 1, \ldots, n$, satisfies p.

vecAll2_map_F q va returns a function from indices belonging to *Fin n* to elements of *T*.

```
Definition vecAll2_map_F {A B T: Type} {p : A → B → Prop}
                         (q : ∀ (a : A) (b : B), p a b → T)
                         {n : nat} : ∀ {x : Vec A n} {y : Vec B n} ,
                         vecAll2 p x y → Fin n → T :=
  match n with
    | O ⇒ fun _ _ _ i ⇒ Fin_zero_elim i
    | S n' ⇒ fun x y va i ⇒ q (nth x i) (nth y i) (vecAll2_nth i va)
  end.
```

vecAll2_map can now be defined from *vecAll2_map_F* via *tabulate*.

```
Definition vecAll2_map {A B T: Type} {p : A → B → Prop}
                       (q : ∀ (a : A) (b : B), p a b → T)
                       {n : nat} {x : Vec A n} {y : Vec B n} :
                       vecAll2 p x y → Vec T n :=
  fun va ⇒ tabulate (vecAll2_map_F q va).
```

5.4.5 Properties of the Projection Function

tabulate_nth is one of the inverse properties of the isomorphisms *tabulate* and *nth*. It shows that, if we construct a vector, given a function f from indices to elements, then the i-th component of the vector (*tabulate f*) is equal to the result $f(i)$, where $1 \leq i \leq n$.

```
Lemma tabulate_nth :
  ∀ {A : Type} {n : nat} {f : Fin n → A} {i : Fin n}, nth (tabulate f) i = f i.
```

vecAll2_map_nth: Given a vector ($vecAll2\_map\ q\ va$), its i-th component is the application of the argument function q to the i-th components of the argument vectors x and y with the corresponding subproof $p\ x_i\ y_i$ extracted from va, where $1 \leq i \leq n$.

That is, $vecAll2\_map\_nth\ q\ va\ i$ shows that the i-th component of the vector ($vecAll2\_map\ q\ va : T^n$) is equal to the result of applying the function q to the i-th component of the vector x, the i-th component of the vector y, and the proof that (x_i, y_i) satisfies p.

```
Lemma vecAll2_map_nth :
  ∀ {A B T: Type} {p : A → B → Prop} (q : ∀ (a : A) (b : B), p a b → T)
    {n : nat} {x : Vec A n} {y : Vec B n} (va : vecAll2 p x y) (i : Fin.t n),
    nth (vecAll2_map q va) i = q (nth x i) (nth y i) (vecAll2_nth i va).
```

nth_F1 shows that the first component of the vector x is the head of x.

```
Lemma nth_F1 :
  ∀ {A : Type} {n : nat} {x : Vec A (S n)}, nth x Fin.F1 = hd x.
```

nth_FS shows that the $(i+1)$-th component of the vector x is the i-th component of the tail of x.

```
Lemma nth_FS :
  ∀{A : Type} {n : nat} {x : Vec A (S n)} (i : Fin.t n),
    nth x (Fin.FS i) = nth (tl x) i.
```

map_nth shows that the i-th component of the vector $\big(f(x_1), \ldots, f(x_n)\big)$ is equal to $f(x_i)$, where $1 \leq i \leq n$.

```
Lemma map_nth:
  ∀ {A B : Type} {n : nat} {f : A → B} {i : Fin.t n} {x : Vec A n},
    nth (map f x) i = f (nth x i).
```

map2_nth shows that the i-th component of the vector $\big(f(x_1, y_1), \ldots, f(x_n, y_n)\big)$ is $f(x_i, y_i)$, where $1 \leq i \leq n$.

```
Lemma map2_nth :
  ∀ {A B C: Type} {n : nat} {f : A → B → C} {i : Fin.t n}
    {x : Vec A n} {y : Vec B n},
   nth (map2 f x y) i = f (nth x i) (nth y i).
```

mapApp_nth shows that the i-th component of the vector $f = \big(f_1(x), \ldots, f_n(x)\big)$ is $f_i(x)$, where $1 \leq i \leq n$.

```
Lemma mapApp_nth :
  ∀ {A B : Type} {n : nat} {f : Vec (A → B) n} {i : Fin.t n} {x : A},
   nth (mapApp f x) i = (nth f i) x.
```

const_nth shows that, given an n-vector, whose components are all equal to x, then its i-th component is equal to x, where $1 \leq i \leq n$.

```
Lemma const_nth :
  ∀ {A : Type} {n : nat} {i : Fin.t n} {x : A}, nth (const x n) i = x.
```

5.4.6 Utilities for Partial Vector Functions

vecOp2OpVec maps a vector $(Some\ v_1, \ldots, Some\ v_n)$ to $Some\ (v_1, \ldots, v_n)$ and maps a vector containing at least one *None* to *None*. Below, *nil T* is an empty vector.

```
Fixpoint vecOp2OpVec {T : Type} {n : nat} (v : Vec (option T) n) :
                                          option (Vec T n) :=
  match v with
    | nil ⇒ Some (nil T)
    | cons v_head n' v_tail ⇒
```

optionMap2 (`fun` $x \Rightarrow$ *cons T x n'*) *v_head* (*vecOp2OpVec v_tail*)

`end`.

vecOp2OpVec_Some shows that, if we apply *vecOp2OpVec* to $\big(f_1(x), \ldots, f_{k+1}(x)\big)$, where each f_i, for $i = 1, \ldots, k+1$, is a partial function of type $T^n \nrightarrow T$, and obtain *Some* $(y_1, \ldots, y_{k+1})$, then *Some* $y_1 = f_1(x) \wedge \ldots \wedge$ *Some* $y_{k+1} = f_{k+1}(x)$.

`Lemma` *vecOp2OpVec_Some* :

$\forall$ $\{T :$ `Type`$\}$ $\{n\ k :$ *nat*$\}$

(*f_head* : *partialFunType T n*) (*f_tail* : *Vec* (*partialFunType T n*) *k*)

(*x* : *Vec T n*) (*y_head* : *T*) (*y_tail* : *Vec T k*),

vecOp2OpVec (*mapApp* (*cons* (*partialFunType T n*) *f_head k f_tail*) *x*) =

Some (*cons T y_head k y_tail*) $\rightarrow$

Some y_head = *f_head x* $\wedge$ *Some y_tail* = *vecOp2OpVec* (*mapApp f_tail x*).

vecOp2OpVec_Some2 shows that, if we apply *vecOp2OpVec* to a vector $\big(f(x_1), \ldots, f(x_{n+1})\big)$, where f is a partial function of type $A \nrightarrow B$, and obtain *Some* $(y_1, \ldots, y_{n+1})$, then *Some* $y_1 = f(x_1) \wedge \ldots \wedge$ *Some* $y_{n+1} = f(x_{n+1})$.

`Lemma` *vecOp2OpVec_Some2* :

$\forall$ $\{A\ B :$ `Type`$\}$ $\{n :$ *nat*$\}$ $\{f : A \rightarrow$ *option B*$\}$

$\{$*x_head* : *A*$\}$ $\{$*x_tail* : *Vec A n*$\}$ $\{$*y_head* : *B*$\}$ $\{$*y_tail* : *Vec B n*$\}$,

vecOp2OpVec (*map f* (*cons A x_head n x_tail*)) =

Some (*cons B y_head n y_tail*) $\rightarrow$

Some y_head = *f x_head* $\wedge$ *Some y_tail* = *vecOp2OpVec* (*map f x_tail*).

vecOp2OpVec_defined shows that, given

- a vector $x = (x_1, \ldots, x_n) : T^n$;
- an n-vector of sets $X = (X_1, \ldots, X_n)$, where X_i is a subset of T for $i = 1, \ldots, n$;

- a vector of partial functions $f = (f_1, \ldots, f_k)$, where f_i is a partial function of type $T^n \nrightarrow T$ for $i = 1, \ldots, k$; and
- a vector $y = (y_1, \ldots, y_k) : T^k$;

if we know that x is an element of $X_1 \times \ldots \times X_n$, and each f_i of f is defined at x, such that $Some\ y_1 = f_1(x), \ldots, Some\ y_k = f_k(x)$, then y is in the range of f on X.

```
Lemma vecOp2OpVec_defined :
 ∀ {T : Type} {n k : nat} {f : Vec (partialFunType T n) k}
    {x : Vec T n} {X : Vec (set T) n} {y : Vec T k},
    setVecProd X x → Some y = vecOp2OpVec (mapApp f x) →
 setVecProd (mapApp (map range f) X) y.
```

vecOp2OpVec_Some_AllDefined shows that, given a vector of partial functions $f = (f_1, \ldots, f_{k+1})$, where each $f_i : T^n \nrightarrow T$ for $i = 1, \ldots, k+1$, if we apply *vecOp2OpVec* to a vector $\big(f_1(x), \ldots, f_{k+1}(x)\big)$, and obtain $Some\ y$, where $y = (y_1, \ldots, y_{k+1})$, then each f_i of f is defined on x, that is, $f_i(x) = Some\ y_i$.

```
Lemma vecOp2OpVec_Some_AllDefined :
 ∀ {T : Type} {n k : nat} (f : Vec (partialFunType T n) (S k))
    (x : Vec T n) (y : Vec T (S k)),
    vecOp2OpVec (mapApp f x) = Some y →
 ∀ f_i, In f_i f → ∃ y_i, f_i x = Some y_i.
```

vecOp2OpVec_wellDefFun shows that *vecOp2OpVec* is a well defined function.

```
Lemma vecOp2OpVec_wellDefFun :
 ∀ {T : Type} {n : nat} {x : Vec (option T) n} {v1 v2 : Vec T n},
   vecOp2OpVec x = Some v1 → vecOp2OpVec x = Some v2 → v1 = v2.
```

5.4.7 Utilities for Inclusion Relations

vecSetIncluded Given $X = (X_1, \ldots, X_n)$ and $Y = (Y_1, \ldots, Y_n)$, where X_i and Y_i are subsets of T, *vecSetIncluded X Y* asserts that $X_i \subseteq Y_i$, for $i = 1, \ldots, n$.

```
Definition vecSetIncluded {T : Type} {n : nat} (X Y : Vec (set T) n) : Prop :=
  vecAll2 subset X Y.
```

vecSetIncluded2 Given $X = (X_1, \ldots, X_n)$ and $Y = (Y_1, \ldots, Y_n)$, where X_i and Y_i are subsets of T, *vecSetIncluded2 X Y* asserts that all elements in $X_1 \times \ldots \times X_n$ are also in $Y_1 \times \ldots \times Y_n$.

```
Definition vecSetIncluded2 {T : Type} {n : nat}
                                    (X Y : Vec (set T) n) : Prop :=
  subset (setVecProd X) (setVecProd Y).
```

vecSetIncluded_vecSetIncluded2 Later in our proofs, we use the fact that *vecSetIncluded X Y* implies *vecSetIncluded2 X Y*, which is shown by

```
Lemma vecSetIncluded_vecSetIncluded2 :
 ∀ {T : Type} {n : nat} {X Y : Vec (set T) n},
   vecSetIncluded X Y → vecSetIncluded2 X Y.
```

Chapter 6

Interval Signature and Structure

In this Chapter, we give the specification of intervals used in our proof system. This specification includes a type symbol *Interval*, its semantic interpreter *iSem*, and several predicates about intervals. We define the concept *Box*, that is, a vector of intervals, based on *Interval*, its semantic interpreter *bSem*, and the predicate *isEmptyBox* on boxes. Finally, this specification has an axiom *isEmpty_dec*, which assumes that given an interval, we can decide if its semantic set is empty.

In our formalization, an interval is interpreted as a set of real numbers. As a consequence, this includes e.g. closed bounded intervals, unbounded intervals, etc. However, if one needs to work with a particular interval model (e.g. end-point interval representation) of this specification, the type *Interval* and its semantic interpreter *iSem* need to be implemented accordingly, and *isEmpty_dec* must be proved in the model.

We first introduce three predicates *pBoundedSet*, *pUnboundedSetL*, and *pUnboundedSetU*.

pBoundedSet Given a real number set X, *pBoundedSet* X asserts that X is bounded.

```
Definition pBoundedSet (X : set R) : Prop :=
  ∃ lb ub : R, ∀ x : R, X x → lb ⩽ x ⩽ ub.
```

pUnboundedSetL Given a real number set X, *pUnboundedSetL X* asserts that X has no lower bound.

```
Definition pUnboundedSetL (X : set R) : Prop :=
  ∀ x : R, X x → ∃ y : R, X y ∧ Rlt y x.
```

pUnboundedSetU Given a real number set X, *pUnboundedSetU X* asserts that X has no upper bound.

```
Definition pUnboundedSetU (X : set R) : Prop :=
  ∀ x : R, X x → ∃ y : R, X y ∧ Rlt x y.
```

```
Module Type INTERVAL.
```

Interval is declared as a type symbol.

```
Parameter Interval : Type.
```

iSem interprets an interval as a set of real numbers.

```
Parameter iSem : Interval → set R.
```

isEmpty asserts that, given an interval i, the real number set *iSem i* is empty.

```
Definition isEmpty (i : Interval) : Prop := emptySet (iSem i).
```

isNonEmpty asserts that, given an interval i, the real number set *iSem i* is nonempty.

```
Definition isNonEmpty (i : Interval) : Prop := nonEmptySet (iSem i).
```

isEmptyAndNotEmpty shows that, given an interval i, if both *isEmpty i* and *isNonEmpty i* are true, then they can lead to a contradiction.

```
Lemma isEmptyAndNotEmpty :
  ∀ {i : Interval}, isEmpty i → isNonEmpty i → False.
```

isEmpty_dec is an assumption, which states that, given an interval i, we can decide if the set $iSem\ i$ is empty. Normally, this assumption is too strong. However, for the commonly used interval models, such as the endpoint model and the midpoint-radius model, it can be decided if the real number set denoted by an interval is empty. This assumption will help prove the FTDIA by case analysis on input interval vectors.

```
Axiom isEmpty_dec : ∀ (i : Interval), {isEmpty i} + {isNonEmpty i}.
```

isBounded asserts that, given an interval i, the real number set $iSem\ i$ is bounded.

```
Definition isBounded (i : Interval) : Prop := pBoundedSet (iSem i).
```

isUnboundedL asserts that, given an interval i, the real number set $iSem\ i$ has no lower bound.

```
Definition isUnboundedL (i : Interval) : Prop := pUnboundedSetL (iSem i).
```

isUnboundedU asserts that, given an interval i, the real number set $iSem\ i$ has no upper bound.

```
Definition isUnboundedU (i : Interval) : Prop := pUnboundedSetU (iSem i).
```

isUnbounded asserts that, given an interval i, the real number set $iSem\ i$ is unbounded.

```
Definition isUnbounded (i : Interval) : Prop :=
  isUnboundedL i ∨ isUnboundedU i.
```

Box is a vector of intervals.

```
Definition Box (n : nat) := Vec Interval n.
```

bSem specifies that a $box = (i_1, \ldots, i_n)$, where i_j is an interval for $j = 1, \ldots, n$, is a set of n-vectors that belong to $iSem\ i_1 \times \ldots \times iSem\ i_n$.

```
Definition bSem {n : nat} (box : Box n) : set (Vec R n) :=
```

setVecProd (*map iSem box*).

isEmptyBox Given a $box = (i_1, \ldots, i_n)$, if any real number set $iSem\ i_j$, where $j = 1, \ldots, n$, is empty, *isEmptyBox box* is true; otherwise, *isEmptyBox box* is false.

```
Fixpoint isEmptyBox {n : nat} (box : Box n) : Prop :=
 match box with
   | Vector.nil ⇒ False
   | Vector.cons b_head n' b_tail ⇒ isEmpty b_head ∨ isEmptyBox b_tail
 end.
```

isEmptyBox_dec shows that, given a *box*, *isEmptyBox box* is either true or false.

```
Lemma isEmptyBox_dec :
 ∀ {n : nat} (box : Box n), {isEmptyBox box} + { ¬ (isEmptyBox box)}.
```

```
End INTERVAL.
```

Now we declare a structure for the interval signature.

```
Declare Module I : INTERVAL.
```

```
Export I.
```

Chapter 7

Infrastructure for the Lemmas and the FTDIA

In this chapter, we define some operations and predicates on real numbers and prove their properties. In Section 7.1, we define two functions for constructing partial vector functions and composing partial functions. In Section 7.2, we define some vector operations on real numbers. In Section 7.3, we define predicates asserting that a function is defined, undefined, continuous, or bounded on an input box. All these definitions and properties are closely related to the formalization and proofs of the FTDIA_NonEmptyBox and its auxiliary lemmas.

The relation between the notation in this proof and in IEEE P1788 is as follows.

notation in this proof	**notation in IEEE P1788**
$\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n) : \overline{\mathbb{IR}}^n$	$\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n) \in \overline{\mathbb{IR}}^n$
$\mathbf{U} = (\mathbf{u}_1, \ldots, \mathbf{u}_{k+1}) : \overline{\mathbb{IR}}^{k+1}$	$\mathbf{u} = (\mathbf{u}_1, \ldots, \mathbf{u}_{k+1}) \in \overline{\mathbb{IR}}^{k+1}$
$f = (f_1, \ldots, f_{k+1}) : \mathbb{R}^n \nrightarrow \mathbb{R}^{k+1}$	$(u_1, \ldots, u_{k+1}) : \mathbb{R}^n \nrightarrow \mathbb{R}^{k+1}$
$f_i : \mathbb{R}^n \nrightarrow \mathbb{R}$	$u_i : \mathbb{R}^n \nrightarrow \mathbb{R}$
$\phi \circ f : \mathbb{R}^n \nrightarrow \mathbb{R}$	$v = \phi(u_1, \ldots, u_{k+1}) : \mathbb{R}^n \nrightarrow \mathbb{R}$

We adopt the following conventions.

- Bold denotes intervals and interval vectors (boxes)
- $x \in \mathbf{X}$ means that a vector x is an element of the Cartesian product $\mathbf{x}_1 \times \ldots \times \mathbf{x}_n$
- We use f to denote both a vector of partial functions and a partial vector function generated from a vector of partial functions

7.1 Constructing and Composing Partial Vector Functions

vecFun Given a vector of partial functions $f = (f_1, \ldots, f_{k+1})$, where each component $f_i : \mathbb{R}^n \nrightarrow \mathbb{R}$ is a partial function for $i = 1, \ldots, k+1$, then *vecFun f* creates a partial vector function with the type $\mathbb{R}^n \nrightarrow \mathbb{R}^{k+1}$.

```
Definition vecFun {n k : nat} (f : Vec (partialFunType R n) (S k)) :
                                  Vec R n → option (Vec R (S k)) :=
  fun (x : Vec R n) ⇒ vecOp2OpVec (mapApp f x).
```

composition composes a partial vector function $f : \mathbb{R}^n \nrightarrow \mathbb{R}^{k+1}$ and a partial function $\phi : \mathbb{R}^{k+1} \nrightarrow \mathbb{R}$ to generate a partial function $\phi \circ f : \mathbb{R}^n \nrightarrow \mathbb{R}$.

```
Definition composition {n k : nat} (f : Vec (partialFunType R n) (S k))
                                   (phi : partialFunType R (S k))
                                   (v : Vec R n) : option R :=
  optionBind phi (vecFun f v).
```

7.2 Vector Operations

vecRabs applies the *Rabs* function to a vector $(v_1, \ldots, v_n)$ and returns $(|\ v_1\ |, \ldots, |\ v_n\ |)$.

```
Definition vecRabs {n : nat} (v : Vec R n ) : Vec R n := map Rabs v.
```

vecRminus applies *Rminus* function to two vectors $(u_1, \ldots, u_n)$ and $(v_1, \ldots, v_n)$, and returns $(u_1 - v_1, \ldots, u_n - v_n)$.

```
Definition vecRminus {n : nat} (u v : Vec R n) : Vec R n := map2 Rminus u v.
```

vecRlt Given two vectors $u = (u_1, \ldots, u_n)$ and $v = (v_1, \ldots, v_n)$, *vecRlt u v* asserts that $u_i < v_i$, for all $i = 1, \ldots, n$.

```
Definition vecRlt {n : nat} (u v : Vec R n) := vecAll2 Rlt u v.
```

vecMin Given two vectors $u = (u_1, \ldots, u_n)$ and $v = (v_1, \ldots, v_n)$, *vecMin u v* returns the component-wise minimum of two vectors, that is, $\big(\min(u_1, v_1), \ldots, \min(u_n, v_n)\big)$.

```
Fixpoint vecMin {n : nat} (u v : Vec R n) : Vec R n :=
 (match u with
    | Vector.nil ⇒ fun _ ⇒ Vector.nil R
    | Vector.cons u_head n' u_tail ⇒
        fun v : Vec R (S n') ⇒
          match Rle_dec u_head (hd v) with
            | left _ ⇒ Vector.cons R u_head n' (vecMin u_tail (tl v))
            | right _ ⇒ Vector.cons R (hd v) n' (vecMin u_tail (tl v))
          end
 end) v.
```

vecMin_lcons Given two vectors $u = (u_1, \ldots, u_{n+1})$ and $v = (v_1, \ldots, v_{n+1})$, *vecMin_lcons* shows that if $u_1 \leqslant v_1$, then *vecMin* $(u_1, \ldots, u_{n+1})$ $(v_1, \ldots, v_{n+1}) =$

$\big(u_1, vecMin(u_2, \ldots, u_{n+1})(v_2, \ldots, v_{n+1})\big)$.

```
Lemma vecMin_lcons :
 ∀ {n : nat} (u_head v_head : R) (u_tail v_tail : Vec R n),
  u_head ⩽ v_head →
   vecMin (Vector.cons R u_head n u_tail)
          (Vector.cons R v_head n v_tail) =
   Vector.cons R u_head n (vecMin u_tail v_tail).
```

vecMin_rcons Given two vectors $u = (u_1, \ldots, u_{n+1})$ and $v = (v_1, \ldots, v_{n+1})$, *vecMin_rcons* shows that if $u_1 > v_1$, then $vecMin\ (u_1, \ldots, u_{n+1})\ (v_1, \ldots, v_{n+1}) = \big(v_1, vecMin(u_2, \ldots, u_{n+1})(v_2, \ldots, v_{n+1})\big)$.

```
Lemma vecMin_rcons :
 ∀ {n : nat} (u_head v_head : R) (u_tail v_tail : Vec R n),
  ¬ (u_head ⩽ v_head) →
   vecMin (Vector.cons R u_head n u_tail)
          (Vector.cons R v_head n v_tail) =
   Vector.cons R v_head n (vecMin u_tail v_tail).
```

O_LT_VecMin Given two vectors $u = (u_1, \ldots, u_n)$ and $v = (v_1, \ldots, v_n)$, where $0 < u_i$ and $0 < v_i$, for all $i = 1, \ldots, n$, then *O_LT_VecMin* shows that $0 < \min(u_i, v_i)$ for all $i = 1, \ldots n$.

```
Lemma O_LT_VecMin :
 ∀ {n : nat} (u v : Vec R n),
   vecAll (Rlt 0) u → vecAll (Rlt 0) v →
  vecAll (Rlt 0) (vecMin u v).
```

vecMinVecAVecB_LE_VecA proves that given four vectors $u = (u_1, \ldots, u_n)$, $v = (v_1, \ldots, v_n)$, $\delta = (\delta_1, \ldots, \delta_n)$, $\gamma = (\gamma_1, \ldots, \gamma_n)$, if $\mid v_i - u_i \mid < \min(\delta_i, \gamma_i)$, for all

$i = 1, \ldots, n$, then $\mid v_i - u_i \mid < \delta_i$, for all $i = 1, \ldots, n$.

```
Lemma vecMinVecAVecB_LE_VecA :
  ∀ {n : nat} (u v deltas gammas : Vec R n),
     vecRlt (vecRabs (vecRminus v u)) (vecMin deltas gammas) →
  vecRlt (vecRabs (vecRminus v u)) deltas.
```

vecMinVecAVecB_LE_VecB shows that given four vectors $u = (u_1, \ldots, u_n)$, $v = (v_1, \ldots, v_n)$, $\delta = (\delta_1, \ldots, \delta_n)$, $\gamma = (\gamma_1, \ldots, \gamma_n)$, if $\mid v_i - u_i \mid < \min(\delta_i, \gamma_i)$, for all $i = 1, \ldots, n$, then $\mid v_i - u_i \mid < \gamma_i$, for all $i = 1, \ldots, n$.

```
Lemma vecMinVecAVecB_LE_VecB :
  ∀ {n : nat} (u v deltas gammas : Vec R n),
     vecRlt (vecRabs (vecRminus v u)) (vecMin deltas gammas) →
  vecRlt (vecRabs (vecRminus v u)) gammas.
```

xInX_xnInXn shows that, given a box $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ and a vector $x = (x_1, \ldots, x_n)$, if $x \in \mathbf{X}$, then $x_i \in \mathbf{x}_i$ for any $i = 1, \ldots, n$.

```
Lemma xInX_xnInXn :
  ∀ {n : nat} {X : Box n} {x : Vec R n} (t : Fin.t n),
    setVecProd (map iSem X) x → iSem (nth X t) (nth x t).
```

7.3 Predicates for Defined, Undefined, Continuous and Bounded

7.3.1 Defined and Undefined

pDefinedOnBox Given a partial function $f : \mathbb{R}^n \nrightarrow R$ and a box $\mathbf{X}$, *pDefinedOnBox f X* asserts that f is defined on $\mathbf{X}$.

```
Definition pDefinedOnBox {n : nat} (f : partialFunType R n)
                                   (X : Box n) : Prop :=
  subset (setVecProd (map iSem X)) (domain f).
```

pNowhereDefinedOnBox Given a partial function $f : \mathbb{R}^n \nrightarrow R$ and a box $\mathbf{X}$, *pNowhereDefinedOnBox f X* asserts that f is nowhere defined on X.

```
Definition pNowhereDefinedOnBox {n : nat} (f : partialFunType R n)
                                          (X : Box n) : Prop :=
  ∀ x : Vec R n, setVecProd (map iSem X) x → f x = None.
```

7.3.2 Continuous

We use Weierstrass's definition [4] (epsilon-delta) of continuity.

pContinuousAtPoint Given a partial function $f : \mathbb{R}^n \nrightarrow \mathbb{R}$, a vector x, and a box $\mathbf{X}$, *pContinuousAtPoint f X x* asserts that f is continuous at x.

```
Definition pContinuousAtPoint {n : nat} (f : partialFunType R n)
                                        (X : Box n) (x : Vec R n) : Prop :=
  ∀ epsilon : R, Rlt 0 epsilon →
    ∃ deltas : Vec R n, vecAll (Rlt 0) deltas ∧
      (∀ (x' : Vec R n), bSem X x' →
        vecRlt (vecRabs (vecRminus x' x)) deltas →
          ∃ y : R, f x = Some y ∧
            ∃ y' : R, f x' = Some y' ∧
              Rlt (Rabs (Rminus y' y)) epsilon).
```

pContinuousOnBox Given a partial function $f : \mathbb{R}^n \nrightarrow \mathbb{R}$ and a box $\mathbf{X}$, *pContinuousOnBox f X* asserts that f is continuous at every $x \in \mathbf{X}$.

```
Definition pContinuousOnBox {n : nat} (f : partialFunType R n)
                                    (X : Box n) : Prop :=
  ∀ (x : Vec R n), bSem X x → pContinuousAtPoint f X x.
```

pContinuousAtPoint_vec Given a partial vector function $f : \mathbb{R}^n \nrightarrow \mathbb{R}^{k+1}$, a vector x, and a box $\mathbf{X}$, *pContinuousAtPoint_vec f X x* asserts that f is continuous at x.

```
Definition pContinuousAtPoint_vec {n k : nat}
                                    (f : Vec (partialFunType R n) (S k))
                                    (X : Box n) (x : Vec R n) : Prop :=
  ∀ vEpsilon : Vec R (S k), vecAll (Rlt 0) vEpsilon →
    ∃ deltas : Vec R n, vecAll (Rlt 0) deltas ∧
      (∀ (x' : Vec R n), bSem X x' →
        ∃ y : Vec R (S k), (vecFun f) x = Some y ∧
        ∃ y': Vec R (S k), (vecFun f) x' = Some y' ∧
          (vecRlt (vecRabs (vecRminus x' x)) deltas →
            vecRlt (vecRabs (vecRminus y' y)) vEpsilon)).
```

pContinuousOnBox_vec Given a partial vector function $f : \mathbb{R}^n \nrightarrow \mathbb{R}^{k+1}$, and a box $\mathbf{X}$, *pContinuousOnBox_vec f X* asserts that f is continuous at every $x \in \mathbf{X}$.

```
Definition pContinuousOnBox_vec {n k : nat}
                                    (f : Vec (partialFunType R n) (S k))
                                    (X : Box n) : Prop :=
  ∀ (x : Vec R n), bSem X x → pContinuousAtPoint_vec f X x.
```

7.3.3 Bounded

pBounded Given a partial function f and an n-vector of sets $X = (X_1, \ldots, X_n)$, *pBounded* f X asserts that the range of f on X is bounded.

```
Definition pBounded {n : nat} (f : partialFunType R n)
                              (X : Vec (set R) n) : Prop :=
  ∃ lb : R, ∃ ub : R, ∀ x : Vec R n, setVecProd X x →
    ∀ r : R, f x = Some r → lb ⩽ r ∧ r ⩽ ub.
```

pBoundedSet_vec Given an n-vector of sets $X = (X_1, \ldots, X_n)$, *pBoundedSet_vec* X asserts that each X_i, for $i = 1, \ldots, n$, is bounded.

```
Definition pBoundedSet_vec {n : nat} (X : Vec (set R) n) : Prop :=
  vecAll pBoundedSet X.
```

pBoundedSet_vec_cons shows that a vector of sets $X = (X_1, \ldots, X_{k+1})$ is bounded iff each component X_i, for $i = 1, \ldots, k+1$, of X is bounded.

```
Lemma pBoundedSet_vec_cons:
  ∀ {k : nat} (X_head : set R) (X_tail : Vec (set R) k),
    pBoundedSet_vec (Vector.cons (set R) X_head k X_tail) ↔
      pBoundedSet X_head ∧ pBoundedSet_vec X_tail.
```

Chapter 8

Decoration System

In this chapter, we formalize the decoration system given in [29]. First, we define the decorations and their corresponding predicates. Second, we define two properties, strength and quality, which are used to order decorations. Then, we prove lemmas describing implication relations between the predicates of decorations and those of intervals. Finally, we define functions returning a decoration with a worst quality and also prove their properties.

8.1 Decorations and Related Predicates

Decoration is an enumerated type. It contains five distinct decorations.

```
Inductive Decoration : Type :=
  | Saf : Decoration
  | Def : Decoration
  | Con : Decoration
  | Emp : Decoration
  | Ill : Decoration.
```

Decoration_dec shows that, given a decoration d, then it must be one of the five decorations saf, def, con, emp, or ill.

```
Lemma Decoration_dec :
 ∀ d : Decoration, d = Saf ∨ d = Def ∨ d = Con ∨ d = Emp ∨ d = Ill.
```

pDecoration is an abbreviation name for the type of the five predicates corresponding to the five decorations.

```
Definition pDecoration : Type :=
 ∀ (n : nat), partialFunType R n → Box n → Prop.
```

pSaf is a predicate corresponding to the decoration saf. Given a partial function f and a box $\mathbf{X}$, then $pSaf\ f\ X$ asserts that the nonempty box $\mathbf{X}$ is a subset of the domain of f, and f is continuous and bounded on $\mathbf{X}$.

```
Definition pSaf : pDecoration :=
 fun (n : nat) (f : partialFunType R n) (X : Box n) ⇒
  nonEmptySet (bSem X) ∧ subset (bSem X) (domain f) ∧
   pContinuousOnBox f X ∧ pBounded f (map iSem X).
```

pDef corresponds to the decoration def. Given a partial function f and a box $\mathbf{X}$, then $pDef\ f\ X$ asserts that the nonempty box $\mathbf{X}$ is a subset of the domain of f.

```
Definition pDef : pDecoration :=
 fun (n : nat) (f : partialFunType R n) (X : Box n) ⇒
  nonEmptySet (bSem X) ∧ subset (bSem X) (domain f).
```

pCon corresponds to the decoration con. Given a partial function f and a box $\mathbf{X}$, then $pCon\ f\ X$ is always true.

```
Definition pCon : pDecoration :=
 fun (n : nat) (f : partialFunType R n) (X : Box n) ⇒ True.
```

pEmp corresponds to the decoration emp. Given a partial function f and a box $\mathbf{X}$, then $pEmp\ f\ X$ asserts that $\mathbf{X}$ is disjoint with the domain of f.

```
Definition pEmp : pDecoration :=
  fun (n : nat) (f : partialFunType R n) (X : Box n) ⇒
    disjoint (bSem X) (domain f).
```

pIll corresponds to the decoration ill. Given a partial function f and a box $\mathbf{X}$, then $pEmp\ f\ X$ asserts that the domain of f is empty.

```
Definition pIll : pDecoration :=
  fun (n : nat) (f : partialFunType R n) (X : Box n) ⇒ emptyDomain f.
```

DecToPred maps a decoration into its corresponding predicate.

```
Definition DecToPred (dec : Decoration) : pDecoration :=
  match dec with
    | Saf ⇒ pSaf
    | Def ⇒ pDef
    | Con ⇒ pCon
    | Emp ⇒ pEmp
    | Ill ⇒ pIll
  end.
```

8.2 Strength and Quality of Decorations

strictStronger implements the strict strength ordering given in (3.1).

```
Definition strictStronger (d1 d2 : Decoration) : Prop :=
  match d1, d2 with
    | Saf, Def ⇒ True
```

```
    | Saf, Con ⇒ True
    | Def, Con ⇒ True
    | Ill, Emp ⇒ True
    | Ill, Con ⇒ True
    | _ , _ ⇒ False
  end.
```

stronger_equal implements the strength ordering given in (3.1).

```
Definition stronger_equal (d1 d2 : Decoration) : Prop :=
  d1 = d2 ∨ strictStronger d1 d2.
```

DecToQuality implements the quality ordering given in (3.2). Here, $\%nat$ means that 0, 1, 2, 3, 4 are natural numbers.

```
Definition DecToQuality (dec : Decoration) : nat :=
  (match dec with
     | Ill ⇒ 0
     | Emp ⇒ 1
     | Con ⇒ 2
     | Def ⇒ 3
     | Saf ⇒ 4
   end)%nat.
```

HasStrongestDec Since the function $\mathrm{dec}(f, \mathbf{X})$ given in [29, eqn. (6)] is undecidable, we define a predicate *HasStrongestDec* in our proof system. Given a partial function f and a box $\mathbf{X}$, then d is the strongest decoration for $(f, \mathbf{X})$, if it makes *HasStrongestDec X f d* true.

```
Definition HasStrongestDec {n : nat} (X : Box n) (f : partialFunType R n) :=
```

```
fun dec : Decoration ⇒
 DecToPred dec n f X ∧
  ∀ dec' : Decoration, DecToPred dec' n f X → stronger_equal dec dec'.
```

8.3 Implication Relations Between Predicates

pSaf_impl_pDef shows that, given a partial function f and a box $\mathbf{X}$, if $(f, \mathbf{X})$ satisfies *pSaf*, then it also satisfies *pDef*.

```
Lemma pSaf_impl_pDef :
 ∀ (n : nat) (f : partialFunType R n) (X : Box n), pSaf n f X → pDef n f X.
```

pSaf_impl_pDef_vec shows that, given a vector of partial functions $f = (f_1, \ldots, f_{k+1})$ and a box $\mathbf{X}$, if each $(f_i, \mathbf{X})$, where $i = 1, \ldots, k+1$, satisfies *pSaf*, then it also satisfies *pDef*.

```
Lemma pSaf_impl_pDef_vec :
 ∀ {n k : nat} {f : Vec (partialFunType R n) (S k)} {X : Box n},
    vecAll (fun f_i : partialFunType R n ⇒ pSaf n f_i X) f →
 vecAll (fun f_i : partialFunType R n ⇒ pDef n f_i X) f.
```

pSaf_impl_pContinousOnBox_vec proves that, given a vector of partial functions $f = (f_1, \ldots, f_{k+1})$ and a box $\mathbf{X}$, if each $(f_i, \mathbf{X})$, where $i = 1, \ldots, k+1$, satisfies *pSaf*, then it also satisfies *pContinuousOnBox*.

```
Lemma pSaf_impl_pContinousOnBox_vec :
 ∀ {n k : nat} {f : Vec (partialFunType R n) (S k)} {X : Box n},
    vecAll (fun f_i : partialFunType R n ⇒ pSaf n f_i X) f →
 vecAll (fun f_i : partialFunType R n ⇒ pContinuousOnBox f_i X) f.
```

pSaf_impl_pBounded_vec shows that, given a vector of partial functions $f = (f_1, \ldots, f_{k+1})$ and a box $\mathbf{X}$, if each $(f_i, \mathbf{X})$, where $i = 1, \ldots, k+1$, satisfies *pSaf*, then it also satisfies *pBounded*.

```
Lemma pSaf_impl_pBounded_vec :
 ∀ {n k : nat} {f : Vec (partialFunType R n) (S k)} {X : Box n},
    vecAll (fun f_i : partialFunType R n ⇒ pSaf n f_i X) f →
 vecAll (fun f_i : partialFunType R n ⇒ pBounded f_i (map iSem X)) f.
```

pSaf_impl_pContinuousOnBox_phiU proves that, given a partial function ϕ and a box $\mathbf{U}$, if $(\phi, \mathbf{U})$ satisfies *pSaf*, then it also satisfies *pContinuousOnBox*.

```
Lemma pSaf_impl_pContinuousOnBox_phiU :
 ∀ {k : nat} {phi : partialFunType R (S k)} {U : Box (S k)},
   pSaf (S k) phi U → pContinuousOnBox phi U.
```

pSaf_impl_pBounded_phiU proves that, given a partial function ϕ and a box $\mathbf{U}$, if $(\phi, \mathbf{U})$ satisfies *pSaf*, then it also satisfies *pBounded*.

```
Lemma pSaf_impl_pBounded_phiU :
 ∀ {k : nat} {phi : partialFunType R (S k)} {U : Box (S k)},
   pSaf (S k) phi U → pBounded phi (map iSem U).
```

pDef_impl_pDefinedOnBox shows that, given a partial function f and a box $\mathbf{X}$, if $(f, \mathbf{X})$ satisfies *pDef*, then it also satisfies *pDefinedOnBox*.

```
Lemma pDef_impl_pDefinedOnBox :
 ∀ {n : nat} {f : partialFunType R n} {X : Box n},
   pDef n f X → pDefinedOnBox f X.
```

pDef_impl_pDefinedOnBox_vec proves that, given a vector of partial functions $f = (f_1, \ldots, f_{k+1})$ and a box $\mathbf{X}$, if each $(f_i, \mathbf{X})$, where $i = 1, \ldots, k+1$, satisfies *pDef*,

then it also satisfies *pDefinedOnBox*.

```
Lemma pDef_impl_pDefinedOnBox_vec :
 ∀ {n k : nat} {f : Vec (partialFunType R n) (S k)} {X : Box n},
    vecAll (fun f_i : partialFunType R n ⇒ pDef n f_i X) f →
 vecAll (fun f_i : partialFunType R n ⇒ pDefinedOnBox f_i X) f.
```

8.4 Worst Quality Functions

minDecoration Given two decorations d_1 and d_2, *minDecoration* returns the smaller in the quality order.

```
Definition minDecoration (d1 d2 : Decoration) : Decoration :=
  match le_lt_dec (DecToQuality d1) (DecToQuality d2) with
    | left _ ⇒ d1
    | right _ ⇒ d2
  end.
```

minDecoration_dec Given two decorations d_1 and d_2, *minDecoration_dec* d_1 d_2 asserts that the decoration with a worse quality can be either d_1 or d_2.

```
Lemma minDecoration_dec :
 ∀ (d1 d2 : Decoration),
  minDecoration d1 d2 = d1 ∨ minDecoration d1 d2 = d2.
```

minVecDecoration returns a decoration with a worst quality from a vector of decorations $cs = (c_1, \ldots, c_n)$.

```
Fixpoint minVecDecoration {n : nat} (cs : Vec Decoration n) : Decoration :=
  (match n as n0 return t Decoration n0 → Decoration with
     | O ⇒ fun cs ⇒ Saf
```

```
    | S n' ⇒ fun cs ⇒ minDecoration (hd cs) (minVecDecoration (tl cs))
  end) cs.
```

minVecDecoration_cons proves that, given a vector of decorations $cs = (c_1, \ldots, c_{n+1})$, if we know that a worst quality decoration of the vector *cs_tail*, that is $(c_2, \ldots, c_{n+1})$, is *minVecDecoration cs_tail*, then a worst quality decoration of *cs* is the same as the worse quality decoration of *cs_head*, that is c_1, and *minVecDecoration cs_tail*.

```
Lemma minVecDecoration_cons :
 ∀ {n : nat} (cs_head : Decoration) (cs_tail : Vec Decoration n),
   minVecDecoration (Vector.cons Decoration cs_head n cs_tail) =
   minDecoration cs_head (minVecDecoration cs_tail).
```

minDecorationSaf_componentsSaf shows that, given two decorations d_1 and d_2, if *minDecoration* d_1 d_2 returns saf, then both d_1 and d_2 are saf.

```
Lemma minDecorationSaf_componentsSaf :
 ∀ {d1 d2 : Decoration},
    minDecoration d1 d2 = Saf → d1 = Saf ∧ d2 = Saf.
```

componentsSaf_minDecorationSaf shows that, given two decorations d_1 and d_2, if both d_1 and d_2 are saf, then *minDecoration* d_1 d_2 is equal to saf.

```
Lemma componentsSaf_minDecorationSaf :
 ∀ {d1 d2 : Decoration},
    d1 = Saf → d2 = Saf → minDecoration d1 d2 = Saf.
```

minVecDecorationSaf_componentsSaf proves that, if a worst quality decoration of the whole decoration vector *cs* is saf, then each decoration component of *cs* is saf.

```
Lemma minVecDecorationSaf_componentsSaf :
 ∀ {k : nat} {cs : Vec Decoration k},
```

$$minVecDecoration\ cs = Saf \rightarrow vecAll\ (\texttt{fun}\ d : Decoration \Rightarrow d = Saf)\ cs.$$

minDecorationDef_componentsSafDef shows that, given two decorations d_1 and d_2, if $minDecoration\ d_1\ d_2$ is equal to def, then at least one of two decorations is def.

```
Lemma minDecorationDef_componentsSafDef :
 ∀ {d1 d2 : Decoration},
   minDecoration d1 d2 = Def →
   (d1 = Saf ∧ d2 = Def) ∨ (d1 = Def ∧ d2 = Saf) ∨ (d1 = Def ∧ d2 = Def).
```

minVecDecorationDef_componentsSafDef proves that, if a worst quality decoration of the whole decoration vector cs is def, then each decoration component of cs is either saf or def.

```
Lemma minVecDecorationDef_componentsSafDef :
 ∀ {k : nat} {cs : Vec Decoration (S k)},
   minVecDecoration cs = Def →
   vecAll (fun d : Decoration ⇒ d = Saf ∨ d = Def) cs.
```

minDecorationEmp_cases shows that, given two decorations d_1 and d_2, if $minDecoration\ d_1\ d_2$ is equal to emp, then d_1 and d_2 should satisfy one of the three cases:

- d_1 is emp, and d_2 can be any decoration except ill; or
- d_1 can be any decoration except ill, and d_2 is emp; or
- both d_1 and d_2 are emp.

```
Lemma minDecorationEmp_cases :
 ∀ {d1 d2 : Decoration},
   minDecoration d1 d2 = Emp →
```

$(d1 = Emp \wedge not\ (d2 = Ill)) \vee (not\ (d1 = Ill) \wedge d2 = Emp) \vee$

$(d1 = Emp \wedge d2 = Emp)$.

minDecorationNotIll proves that, given two decorations d_1 and d_2, if *minDecoration* d_1 d_2 is not equal to ill, then neither d_1 nor d_2 is ill.

```
Lemma minDecorationNotIll :
```

$\forall\ \{d1\ d2 : Decoration\},$

$minDecoration\ d1\ d2 \neq Ill \rightarrow d1 \neq Ill \wedge d2 \neq Ill.$

minVecDecorationNotIll shows that, if a worst quality decoration of a vector *cs* is not ill, then none of the components of *cs* can be ill.

```
Lemma minVecDecorationNotIll :
```

$\forall\ \{k : nat\}\ \{cs : Vec\ Decoration\ k\},$

$minVecDecoration\ cs \neq Ill \rightarrow vecAll\ (\texttt{fun}\ d : Decoration \Rightarrow d \neq Ill)\ cs.$

minVecDecorationEmp_impl_NotIll proves that, if a worst quality decoration of a vector *cs* is emp, then a worst quality decoration of *cs* cannot be ill.

```
Lemma minVecDecorationEmp_impl_NotIll :
```

$\forall\ \{k : nat\}\ \{cs : Vec\ Decoration\ k\},$

$minVecDecoration\ cs = Emp \rightarrow minVecDecoration\ cs \neq Ill.$

minVecDecorationEmp_componentsNotIll shows that, if a worst quality decoration of a vector *cs* is emp, then each component of *cs* cannot be ill.

```
Lemma minVecDecorationEmp_componentsNotIll :
```

$\forall\ \{k : nat\}\ \{cs : Vec\ Decoration\ (S\ k)\},$

$minVecDecoration\ cs = Emp \rightarrow vecAll\ (\texttt{fun}\ d : Decoration \Rightarrow \neg\ d = Ill)\ cs.$

minVecDecorationEmp_existEmp proves that, if a worst quality decoration of a vector *cs* is emp, then there exists a component c_i in *cs* such that $c_i = Emp$.

Lemma *minVecDecorationEmp_existEmp* :

$\forall\ \{k : nat\}\ \{cs :\ Vec\ Decoration\ (S\ k)\},$

$minVecDecoration\ cs = Emp \rightarrow \exists\ ci : Decoration,\ Vector.In\ ci\ cs \wedge ci = Emp.$

minVecDecorationEmp_componentsEmpNotIll shows that, if a worst quality decoration of a vector *cs* is emp, then we can deduce the following

- *cs_head* = *Emp*, and each component of *cs_tail* must not be ill, or
- *cs_head* is not ill, and each component of *cs_tail* is not ill, and there exists a *ci* in *cs* such that *ci* = *Emp*.

Lemma *minVecDecorationEmp_ComponentsEmpNotIll* :

$\forall\ \{k : nat\}\ \{cs\_tail :\ Vec\ Decoration\ (S\ k)\}\ \{cs\_head : Decoration\},$

$minDecoration\ cs\_head\ (minVecDecoration\ cs\_tail) = Emp \rightarrow$

$cs\_head = Emp \wedge vecAll\ (\texttt{fun}\ d \Rightarrow d \neq Ill)\ cs\_tail \vee$

$cs\_head \neq Ill \wedge vecAll\ (\texttt{fun}\ d \Rightarrow d \neq Ill)\ cs\_tail \wedge$

$\exists\ ci,\ Vector.In\ ci\ cs\_tail \wedge ci = Emp.$

minDecorationIll_componentsIll shows that, given two decorations d_1 and d_2, if *minDecoration* d_1 d_2 is equal to ill, then at least one of d_1 and d_2 is ill.

Lemma *minDecorationIll_componentsIll* :

$\forall\ \{d1\ d2 : Decoration\},$

$minDecoration\ d1\ d2 = Ill \rightarrow d1 = Ill \vee d2 = Ill.$

minVecDecorationIll_existIll proves that if the worst quality decoration of the whole decoration vector *cs* is ill, then there exists a component c_i in *cs* such that c_i = *Ill*.

Lemma *minVecDecorationIll_existIll* :

$$\forall\ \{k : nat\}\ \{cs :\ Vec\ Decoration\ (S\ k)\},$$

$$minVecDecoration\ cs = Ill \rightarrow \exists\ ci,\ Vector.In\ ci\ cs \wedge ci = Ill.$$

Chapter 9

Lemma C2

In this chapter, we formalize Lemma *C2* (stated below), and formally prove its cases (a), (b) and (e) in Sections 9.2, 9.3, and 9.4, respectively. Cases (c) and (d) are proved as part of auxiliary lemmas *C3a_pSaf* A.2.1 and *C3a_pEmp* A.2.3 needed for the proof of Lemma *C3*.

9.1 Statement of Lemma C2

We have

- a partial vector function $f = (f_1, \ldots, f_{k+1}) : \mathbb{R}^n \nrightarrow \mathbb{R}^{k+1}$, where $f_i : \mathbb{R}^n \nrightarrow \mathbb{R}$, for $i = 1, \ldots, k+1$;
- a partial function $\phi : \mathbb{R}^{k+1} \nrightarrow \mathbb{R}$;
- a box $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n) : \overline{\mathbb{IR}}^n$, where $\mathbf{x}_i : \overline{\mathbb{IR}}$, for $i = 1, \ldots, n$, is an interval; and
- a box $\mathbf{U} = (\mathbf{u}_1, \ldots, \mathbf{u}_{k+1}) : \overline{\mathbb{IR}}^{k+1}$, where $\mathbf{u}_i : \overline{\mathbb{IR}}$, for $i = 1, \ldots, k+1$, satisfies $\mathbf{u}_i \supseteq \text{range}(f_i, \mathbf{X})$.

We prove the following

(a) If each f_i in f is everywhere defined on $\mathbf{X}$, and ϕ is everywhere defined on $\mathbf{U}$, then the composition $\phi \circ f : \mathbb{R}^n \nrightarrow \mathbb{R}$ is everywhere defined on $\mathbf{X}$.

(b) If each f_i in f is everywhere defined and continuous on $\mathbf{X}$, and ϕ is everywhere defined and continuous on $\mathbf{U}$, then $\phi \circ f : \mathbb{R}^n \nrightarrow \mathbb{R}$ is continuous on $\mathbf{X}$.

(c) If each f_i in f is bounded on $\mathbf{X}$, and ϕ maps bounded sets, which are subsets of $\mathbb{R}^{k+1}$, to bounded sets, which are subsets of $\mathbb{R}$, then $\phi \circ f : \mathbb{R}^n \nrightarrow \mathbb{R}$ is bounded on $\mathbf{X}$.

(d) If any f_i in f is nowhere defined on $\mathbf{X}$, or ϕ is nowhere defined on $\mathbf{U}$, then $\phi \circ f : \mathbb{R}^n \nrightarrow \mathbb{R}$ is nowhere defined on $\mathbf{X}$.

(e) If any f_i in f has an empty domain, or ϕ has an empty domain, then $\phi \circ f : \mathbb{R}^n \nrightarrow \mathbb{R}$ has an empty domain.

9.2 Lemma C2a

Lemma *C2a* shows that if each f_i of f, for $i = 1, \ldots, k+1$, is everywhere defined on $\mathbf{X}$, and ϕ is everywhere defined on $\mathbf{U}$, then $\phi \circ f$ is everywhere defined on $\mathbf{X}$. We first state this lemma. Then we formalize it in Coq and explain its specification. Finally, we give a summary of the mechanized proof of this lemma.

Lemma 9.2.1 (*C2a*)**.** *If*

(i) *each* $f_i : \mathbb{R}^n \nrightarrow \mathbb{R}$ *in* $f : \mathbb{R}^n \nrightarrow \mathbb{R}^{k+1}$*, for* $i = 1, \ldots, k+1$*, is everywhere defined on* $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$*;*

(ii) $\phi : \mathbb{R}^{k+1} \nrightarrow \mathbb{R}$ *is everywhere defined on* $\mathbf{U} = (\mathbf{u}_1, \ldots, \mathbf{u}_{k+1})$*; and*

(iii) $\text{range}(f_i, \mathbf{X}) \subseteq \mathbf{u}_i$, *for* $i = 1, \ldots, k+1$,

then $\phi \circ f$ *is everywhere defined on* $\mathbf{X}$.

```
Lemma C2a :
 ∀ {n k : nat}
   {f : Vec (partialFunType R n) (S k)} {X : Box n}
   {phi : partialFunType R (S k)} {U : Box (S k)},
   vecAll (fun f_i : partialFunType R n ⇒ pDefinedOnBox f_i X) f →
   pDefinedOnBox phi U →
   vecAll2 subset (mapBoxApp f (map iSem X)) (map iSem U) →
 pDefinedOnBox (composition f phi) X.
```

For the reader's convenience, we explain the above specification and give references to the corresponding definitions.

- *partialFunType R n* (§5.3) is the partial function type $\mathbb{R}^n \nrightarrow \mathbb{R}$
- *f : Vec (partialFunType R n) (S k)* is a $(k+1)$-vector of partial functions $(f_1, \ldots, f_{k+1})$
- *X : Box n* (§6) is an n-vector of intervals $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$
- *pDefinedOnBox f_i X* (§7.3.1) asserts that f_i is everywhere defined on $\mathbf{X}$
- *vecAll* (`fun` *f_i : partialFunType R n ⇒ pDefinedOnBox f_i X*) *f* (§5.4.2) asserts that each f_i in f is everywhere defined on $\mathbf{X}$
- *iSem* (§6) interprets an interval as a set of real numbers
- *map iSem X* interprets a box as an n-vector of real number sets

- *mapBoxApp f (map iSem X)* (§5.4.3) returns a $(k+1)$-vector of ranges for each f_i on $\mathbf{X}$, that is, $\big(\text{range}(f_1, \mathbf{X}), \ldots, \text{range}(f_{k+1}, \mathbf{X})\big)$

- *vecAll2 subset (mapBoxApp f (map iSem X)) (map iSem U)* asserts that

$$\text{range}(f_i, \mathbf{X}) \subseteq \mathbf{u}_i \quad \text{for } i = 1, \ldots, k+1$$

- *composition f phi* (§7.1) constructs the composition of ϕ and f, that is $\phi \circ f : \mathbb{R}^n \twoheadrightarrow \mathbb{R}$

Essential parts of the mechanized proof

1. If each f_i is everywhere defined on $\mathbf{X}$, then the partial vector function f, which is denoted by *vecFun f* (§7.1) in Coq, is everywhere defined on $\mathbf{X}$ as well. This fact is shown by Lemma *AllfiDefined_to_fDefined* (A.1.1). Thus, for each $x \in \mathbf{X}$,

$$(\mathit{vecFun}\, f)x = \mathit{Some}(u_1, \ldots, u_{k+1}),$$

   where $f_i(x) = \mathit{Some}\ u_i$, $u_i : \mathbb{R}$ and $\mathit{Some}\ u_i : \mathit{option}\,\mathbb{R}$.

2. Since $\text{range}(f_i, \mathbf{X}) \subseteq \mathbf{u}_i$, for $i = 1, \ldots, k+1$, then $u_i \in \mathbf{u}_i$. If we define $u = (u_1, \ldots, u_{k+1})$, then $u : \mathbf{U}$.

3. Since ϕ is defined for all $u : \mathbf{U}$, we know that $\phi(u) = \mathit{Some}\, v$, where $v : \mathbb{R}$.

This shows that $\phi \circ f$ is everywhere defined on $\mathbf{X}$.

9.3 Lemma C2b

Lemma *C2b* shows that if each f_i of f, for $i = 1, \ldots, k+1$, is everywhere defined and continuous on $\mathbf{X}$, and ϕ is everywhere defined and continuous on $\mathbf{U}$, then $\phi \circ f$ is continuous on $\mathbf{X}$. We first list the notation used in the proof of *C2b*. Then we state Lemma *C2b*. Finally, we formalize and prove it in Coq.

Notation For n-vectors u and v, we define

- $\min(u, v) = \big(\min(u_1, v_1), \ldots, \min(u_n, v_n)\big)$
- $u < v \Leftrightarrow u_i < v_i$, for $i = 1, \ldots, n$
- $\mid u \mid = (\mid u_1 \mid, \ldots, \mid u_n \mid)$

Lemma 9.3.1 (*C2b*)**.** *In addition to the hypotheses of Lemma 9.2.1 (C2a), we have the hypotheses*

(i) each $f_i : \mathbb{R}^n \nrightarrow \mathbb{R}$ *in* $f : \mathbb{R}^n \nrightarrow \mathbb{R}^{k+1}$*, for* $i = 1, \ldots, k+1$*, is continuous on* $\mathbf{X}$*, and*

(ii) $\phi : \mathbb{R}^{k+1} \nrightarrow \mathbb{R}$ *is continuous on* $\mathbf{U}$*.*

Then $\phi \circ f$ *is continuous on* $\mathbf{X}$*. This is proved by*

```
Lemma C2b :
 ∀ {n k : nat}
   {f : Vec (partialFunType R n) (S k)} {X : Box n}
   {phi : partialFunType R (S k)} {U : Box (S k)},
   vecAll (fun f_i : partialFunType R n ⇒ pDefinedOnBox f_i X) f →
```

vecAll (`fun` *f_i*: *partialFunType R n* ⇒ *pContinuousOnBox f_i X*) *f* →
vecAll2 subset (*mapBoxApp f* (*map iSem X*)) (*map iSem U*) →
pContinuousOnBox phi U →
pContinuousOnBox (*composition f phi*) *X*.

This specification is like that of Lemma *C2a* with these additions:

- *pContinuousOnBox f_i X* (§7.3.2) asserts that f_i is continuous on $\mathbf{X}$
- *vecAll* (`fun` *f_i*: *partialFunType R n* ⇒ *pContinuousOnBox f_i X*) *f* asserts that each f_i is continuous on $\mathbf{X}$

Essential parts of the mechanized proof

We use the ϵ–δ definition in our mechanized proof.

Since $f_1, \ldots, f_{k+1}$ are everywhere defined and continuous on $\mathbf{X}$, the vector function *vecFun f* is everywhere defined and continuous on $\mathbf{X}$, which is shown in A.1.1 and A.1.2. Then,

$$\begin{aligned} \forall\, x_f \in \mathbf{X}.\, \forall\, \epsilon_f \in \mathbb{R}^{k+1}.\epsilon_f > 0.\, \exists\, \delta_f \in \mathbb{R}^n.\delta_f > 0.\, \forall\, {x_f}' \in \mathbf{X}. \\ \mid {x_f}' - x_f \mid < \delta_f \Rightarrow \mid f({x_f}') - f(x_f) \mid < \epsilon_f \end{aligned} \tag{9.6}$$

From the continuity of ϕ on $\mathbf{U}$,

$$\begin{aligned} \forall\, u \in \mathbf{U}.\, \forall\, \epsilon_\phi \in \mathbb{R}.\epsilon_\phi > 0.\, \exists\, \delta_\phi \in \mathbb{R}^n.\delta_\phi > 0.\, \forall\, u' \in \mathbf{U}. \\ \mid u' - u \mid < \delta_\phi \;\Rightarrow \mid \phi(u') - \phi(u) \mid < \epsilon_\phi \end{aligned} \tag{9.7}$$

Our goal is to show that

$$\begin{aligned}&\forall\, x \in \mathbf{X}.\, \forall\, \epsilon \in \mathbb{R}.\epsilon > 0.\, \exists\, \delta \in \mathbb{R}^n.\delta > 0.\, \forall\, x' \in \mathbf{X}.\\ &\quad \mid x' - x \mid < \delta \;\Rightarrow \mid (\phi \circ f)(x') - (\phi \circ f)(x) \mid < \epsilon\end{aligned} \tag{9.8}$$

Our proof follows the following steps.

1. We fix x and ϵ in (9.8). Then we have the new goal

$$\exists\, \delta \in \mathbb{R}^n.\delta > 0.\, \forall\, x' \in \mathbf{X}.\, \mid x' - x \mid < \delta \;\Rightarrow \mid (\phi \circ f)(x') - (\phi \circ f)(x) \mid < \epsilon \tag{9.9}$$

2. Let $u = f(x)$ and $\epsilon_\phi = \epsilon$ in (9.7). Then $u \in \mathbf{U}$, since $\mathbf{U}$ contains the range of f over $\mathbf{X}$, and there exists $\delta_\phi > 0$, such that

$$\forall\, u' \in \mathbf{U}.\, \mid u' - f(x) \mid < \delta_\phi \Rightarrow \mid \phi(u') - \phi(f(x)) \mid < \epsilon \tag{9.10}$$

3. Let $x_f = x$ and $\epsilon_f = \delta_\phi$ in (9.6). Then there exists $\delta_f > 0$, such that

$$\forall\, {x_f}' \in \mathbf{X}.\, \mid {x_f}' - x \mid < \delta_f \Rightarrow \mid f({x_f}') - f(x) \mid < \delta_\phi \tag{9.11}$$

4. Let $\delta = \delta_f$ and fix x' in (9.9). Then our new goal becomes

$$\mid x' - x \mid < \delta_f \;\Rightarrow \mid (\phi \circ f)(x') - (\phi \circ f)(x) \mid < \epsilon \tag{9.12}$$

5. Let $u' = f(x')$ in (9.10). Then $u' \in \mathbf{U}$, because $\mathbf{U}$ contains the range of f over $\mathbf{X}$. We have

$$\mid f(x') - f(x) \mid < \delta_\phi \Rightarrow \mid \phi(f(x')) - \phi(f(x)) \mid < \epsilon \tag{9.13}$$

6. Let $x_f{}' = x'$ in (9.11). Then

$$| x' - x | < \delta_f \Rightarrow | f(x') - f(x) | < \delta_\phi \tag{9.14}$$

7. From (9.14) and (9.13), we obtain

$$| x' - x | < \delta_f \Rightarrow | f(x') - f(x) | < \delta_\phi \Rightarrow | (\phi \circ f)(x') - (\phi \circ f)(x) | < \epsilon$$

which shows (9.12).

9.4 Lemma C2e

Lemma *C2e* shows that if any f_i of f or ϕ has an empty domain, then $\phi \circ f$ has an empty domain. In this section, we first introduce Lemma *C2e* and outline its proof. Then, we formalize Lemma *C2e* and its two auxiliary lemmas, and give a summary of their mechanized proofs.

Lemma 9.4.1 (*C2e*)**.** *If*

(i) any f_i in f, where $i = 1, \ldots, k+1$, has an empty domain, or

(ii) ϕ has an empty domain,

then $\phi \circ f$ has an empty domain.

Proof outline

We prove *C2e* by two cases

- if any f_i of f has an empty domain, then $\phi \circ f$ has an empty domain (Lemma *C2ea*)

- if ϕ has an empty domain, then $\phi \circ f$ has an empty domain (Lemma *C2eb*)

Lemma 9.4.2 (*C2ea*). *If any f_i in f, where $i = 1, \ldots, k+1$, has an empty domain, then $\phi \circ f$ has an empty domain.*

```
Lemma C2ea :
 ∀ {n k : nat}
   {f : Vec (partialFunType R n) (S k)} {phi : partialFunType R (S k)},
   (∃ f_i : partialFunType R n, Vector.In f_i f ∧ emptyDomain f_i) →
 emptyDomain (composition f phi).
```

- *emptydomain* f_i (§5.3) asserts that f_i has an empty domain
- $\exists\ f_i,\ Vector.In\ f_i\ f\ \wedge\ emptyDomain\ f_i$ asserts that there exists an f_i in f, such that f_i has an empty domain.

Essential parts of the mechanized proof

If any f_i of f has an empty domain, then *vecFun f* has empty domain, i.e. for any $x \in \mathbb{R}^n$, $(vecFun\ f)\ x = None$. This is shown by lemma *f_iEmptyDomain_fEmptyDomain* (A.1.3). Thus, $(\phi \circ f)\ x = \phi\big((vecFun\ f)\ x\big) = \phi(None) = None$.

This shows that $\phi \circ f$ has an empty domain.

Lemma 9.4.3 (*C2eb*). *If ϕ has an empty domain, then $\phi \circ f$ has an empty domain.*

```
Lemma C2eb :
 ∀ {n k : nat}
   {f : Vec (partialFunType R n) (S k)}
   {phi : partialFunType R (S k)}, emptyDomain phi →
 emptyDomain (composition f phi).
```

Essential parts of the mechanized proof

We prove this lemma by case analysis.

1. If *vecFun f* has a nonempty domain in $\mathbb{R}^n$, and ϕ has an empty domain in $\mathbb{R}^{k+1}$, then for any $x : \mathbb{R}^n$,

   - if *vecFun f* is defined at x, and $f(x) = y$, then $(\phi \circ f)\ x = \phi\big((vecFun\ f)\ x\big) = \phi(y) = None$;
   - if *vecFun f* is undefined at x, then $(\phi \circ f)x = \phi\big((vecFun\ f)\ x\big) = \phi(None) = None$.

2. If both *vecFun f* and ϕ have an empty domain, then for any $x : \mathbb{R}^n$,
   $(\phi \circ f)\ x = \phi\big((vecFun\ f)\ x\big) = \phi(None) = None$.

   This shows that $\phi \circ f$ has an empty domain.

Proof of Lemma C2e

The Coq formalization of Lemma *C2e* follows. We prove it using Lemma *C2ea* and Lemma *C2eb*.

```
Lemma C2e :
  ∀ {n k : nat}
    {f : Vec (partialFunType R n) (S k)} {phi : partialFunType R (S k)},
    (∃ f_i : partialFunType R n, Vector.In f_i f ∧ emptyDomain f_i) ∨
    emptyDomain phi →
  emptyDomain (composition f phi).
```

Chapter 10

Lemma C3

Lemma *C3* proves the containment order of decorations in *FTDIA_NonEmptyBox*. It shows that, if we evaluate a function on a box, the decoration obtained from the interval-mode evaluation includes the strongest decoration.

In Section 10.1, we summarize Lemma *C3* [29]. In the next two sections, we formalize and prove two auxiliary lemmas *C3_pCmin_NonEmptyBox* and *decfX_StrongerThan_cMin*, which can be combined to prove Lemma *C3*.

Notation. Given an arbitrary decoration c, we use p_c to denote its corresponding predicate. For example, if $c = \mathsf{saf}$, then p_c denotes *pSaf*.

10.1 Lemma C3

Lemma 10.1.1 (*C3*)**.** *If*

(i) $c_i \subseteq \mathrm{dec}(f_i, \mathbf{X})$, *for* $i = 1, \ldots, k$,

(ii) $c_0 \subseteq \mathrm{dec}(\phi, \mathbf{U})$, *and*

(iii) $d = \mathrm{dec}(\phi \circ f, \mathbf{X})$,

then $d \subseteq c_{\min} = \min(c_0, \ldots, c_k)$.

Proof outline

Instead of proving $d \subseteq c_{\min}$ directly, we show two lemmas *C3_pCmin_NonEmptyBox*, which proves that $(\phi \circ f, \mathbf{X})$ satisfies $p_{c_{\min}}$ and *decfX_StrongerThan_cMin*, which proves that for any $d = \text{dec}(\phi \circ f, \mathbf{X})$, $p_{c_{\min}}(\phi \circ f, \mathbf{X})$ implies that $d \subseteq c_{\min}$. Then, we can prove $d \subseteq c_{\min}$ by combining these two lemmas.

10.2 Lemma C3_pCmin_NonEmptyBox

In this section, we first introduce Lemma *C3_pCmin_NonEmptyBox*. Then we formalize and prove it in Coq. Finally, we give the essential parts of its mechanized proof.

Lemma 10.2.1 (*C3_pCmin_NonEmptyBox*)**.** *Given a vector of functions* $f = (f_1, \ldots, f_k)$, *where* $f_i : \mathbb{R}^n \nrightarrow \mathbb{R}$, *for* $i = 1, \ldots, k$, *if*

(i) $\mathbf{X}$ *is a nonempty box,*

(ii) each $(f_i, \mathbf{X})$ *satisfies* p_{c_i}, *for* $i = 1, \ldots, k$,

(iii) $(\phi, \mathbf{U})$ *satisfies* p_{c_0},

(iv) $\text{range}(f_i, \mathbf{X}) \subseteq \mathbf{u}_i$, *for* $i = 1, \ldots, k$, *and*

(v) $c_{\min} = \min(c_0, c_1, \ldots, c_k)$,

then $(\phi \circ f, \mathbf{X})$ *satisfies* $p_{c_{\min}}$.

`Lemma` *C3_pCmin_NonEmptyBox* :

```
∀ {n k: nat}
  (f : Vec (partialFunType R n) k) (X : Box n) (cs : Vec Decoration k)
  (phi : partialFunType R k) (U : Box k) (c0 : Decoration),
  nonEmptySet (bSem X) →
  vecAll2 (fun fi di ⇒ DecToPred di _ fi X) f cs →
  DecToPred c0 _ phi U →
  vecAll2 subset (mapBoxApp f (map iSem X)) (map iSem U) →
  let c_min := minDecoration c0 (minVecDecoration cs) in
DecToPred c_min n
        (fun v : Vec R n ⇒ optionBind phi (vecOp2OpVec (mapApp f v))) X.
```

Here,

- *minDecoration* (§8.4) returns a decoration with a worse quality from its two arguments
- *minVecDecoration* (§8.4) returns a decoration with a worst quality from a vector of decorations
- $c_{\min} :=$ *minDecoration* $c0$ (*minVecDecoration* cs) states that we assign to $c_{\min}$ a worst quality decoration of $c_0, c_1, \ldots, c_k$, where $(f_i, \mathbf{X})$ satisfies p_{c_i}, and $(\phi, \mathbf{U})$ satisfies p_{c_0}
- *DecToPred* $c_{\min}$ returns the predicate $p_{c_{\min}}$ (§8.1)
- *DecToPred* $c_{\min}$ n

  (*fun* v : *Vec* R n => *optionBind phi* (*vecOp2OpVec* (*mapApp* f v))) X

  (§8.1) asserts that $(\phi \circ f, \mathbf{X})$ satisfies $p_{c_{\min}}$, where n is the first argument of $p_{c_{\min}}$

Essential parts of the mechanized proof

We prove *C3_pCmin_NonEmptyBox* by induction on k, which is the length of the vector of functions $f = (f_1, \ldots, f_k)$.

1. Base Case. When $k = 0$, Lemma *pCmin_O* (A.2.5) shows that $(\phi \circ f, \mathbf{X})$ satisfies $p_{c_{\min}}$, that is, p_{c_0}.

2. Induction Case. When $k \geq 1$, Lemma *pCmin_Sk* (A.2.5) shows that $(\phi \circ f, \mathbf{X})$ satisfies $p_{c_{\min}}$ using case analysis on $\min(c_0, c_1, \ldots, c_{k+1})$, as follows.

   i. If $\min(c_0, c_1, \ldots, c_{k+1}) = \mathsf{saf}$, then we show that $(\phi \circ f, \mathbf{X})$ satisfies *pSaf*. From the definition of *pSaf*, we prove that

      - $\phi \circ f$ is everywhere defined on $\mathbf{X}$ by Lemma *C3a_pSaf_inDomain* (A.2.1)
      - $\phi \circ f$ is continuous on $\mathbf{X}$ by Lemma *C3a_pSaf_pContinuousOnBox* (A.2.1)
      - $\phi \circ f$ is bounded on $\mathbf{X}$ by Lemma *C3a_pSaf_pBounded* (A.2.1)
      - $\mathbf{X}$ is nonempty by Lemma *C3a_pSaf_nonEmptySet* (A.2.1)

   ii. If $\min(c_0, c_1, \ldots, c_{k+1}) = \mathsf{def}$, then we show that $(\phi \circ f, \mathbf{X})$ satisfies *pDef*. From the definition of *pDef*, we prove that

      - $\phi \circ f$ is everywhere defined on $\mathbf{X}$ by Lemma *C3a_pDef_inDomain* (A.2.2)
      - $\mathbf{X}$ is nonempty by Lemma *C3a_pDef_nonEmptySet* (A.2.2)

   iii. If $\min(c_0, c_1, \ldots, c_{k+1}) = \mathsf{con}$, then we show that $(\phi \circ f, \mathbf{X})$ satisfies *pCon*, which is always true.

   iv. If $\min(c_0, c_1, \ldots, c_{k+1}) = \mathsf{emp}$, then we show that $(\phi \circ f, \mathbf{X})$ satisfies *pEmp*. From the definition of *pEmp*, we prove that

- either there exists some f_i in f, such that the domain of f_i is disjoint from $\mathbf{X}$ by Lemma *C3a_pEmp_ciEmp* (A.2.3), or
- the domain of ϕ is disjoint from $\mathbf{U}$ by Lemma *C3a_pEmp_c0Emp* (A.2.3)

v. If $\min(c_0, c_1, \ldots, c_{k+1}) = \mathsf{ill}$, then we show that $(\phi \circ f, \mathbf{X})$ satisfies *pIll*. From the definition of *pIll*, we prove that

- either there exists some f_i in f such that the domain of f_i is empty by Lemma *C3a_pIll_ciIll* (A.2.4), or
- the domain of ϕ is empty by Lemma *C2eb* (§9.4)

Hence, we can conclude that $(\phi \circ f, \mathbf{X})$ satisfies $p_{c_{\min}}$.

10.3 Lemma d_StrongerThan_cMin

Lemma 10.3.1 (d_StrongerThan_cMin)**.** *For any decoration c, if $p_c(f, \mathbf{X})$ is true, then for any strongest decoration $d = dec(f, \mathbf{X})$, $d \subseteq c$.*

```
Lemma d_StrongerThan_cMin :
 ∀ {n : nat} {f : partialFunType R n} {X : Box n} {cMin : Decoration},
   DecToPred cMin n f X →
    ∀ d, HasStrongestDec X f d → stronger_equal d cMin.
```

---

stronger_equal d cMin (§8.2) means that $d \subseteq cMin$. That is, d is stronger than *cMin*.

Essential parts of the mechanized proof

We prove Lemma *d_StrongerThan_cMin* by case analysis on the decoration *c*.

1. If $c = \mathsf{saf}$, then the strongest decoration d for the pair $(f, \mathbf{X})$ must be saf. By the definition of the strength ordering (§8.2), we can prove $\mathsf{saf} \subseteq \mathsf{saf}$. ($\mathsf{saf} \subseteq \mathsf{saf}$ is trivial, but we still have to prove it in Coq).

2. If $c = \mathsf{def}$, then the strongest decoration d can be either saf or def. By the definition of the strength ordering, we can prove $\mathsf{saf} \subseteq \mathsf{def}$ and $\mathsf{def} \subseteq \mathsf{def}$.

3. If $c = \mathsf{con}$, then by the definition of the strength ordering, we can always prove $d \subseteq \mathsf{con}$ for any decoration d.

4. If $c = \mathsf{emp}$, then the strongest decoration d can be either emp or ill. By the definition of the strength ordering, we can prove both $\mathsf{emp} \subseteq \mathsf{emp}$ and $\mathsf{ill} \subseteq \mathsf{emp}$.

5. If $c = \mathsf{ill}$, then the strongest decoration d must be ill. By the definition of the strength ordering, we can prove $\mathsf{ill} \subseteq \mathsf{ill}$.

Chapter 11

Decorated Interval Arithmetic

In this chapter, we first formalize the concepts of the decorated interval arithmetic, such as decorated intervals, initial decorations, initial decorated intervals, and decorated interval library operations. Then we define the syntax and semantics of decorated interval arithmetic. Finally, we show some auxiliary lemmas, which are used in the proof of the *FTDIA_NonEmptyBox*.

11.1 Basic Concepts

We formalize the concepts of decorated intervals, initial decorated intervals, and decorated interval library operations in this section.

11.1.1 Decorated Interval

A *decorated interval* consists of:

- an interval *ivl* and
- a decoration *dec*,

such that the two satisfy the *invariant property* (IP) [29].

IP states that not all combinations of intervals and decorations are permitted in semantics. The possible combinations are:

- if an interval *ivl* is nonempty, then its decoration *dec* is in $\{\mathsf{saf}, \mathsf{def}, \mathsf{con}\}$, or
- if an interval *ivl* is empty, then its decoration *dec* is in $\{\mathsf{emp}, \mathsf{ill}\}$.

`Definition` *IP* (*ivl* : *Interval*) (*dec* : *Decoration*) :=

(*isEmpty ivl* ∧ (*dec* = *Emp* ∨ *dec* = *Ill*)) ∨

(*isNonEmpty ivl* ∧ (*dec* = *Saf* ∨ *dec* = *Def* ∨ *dec* = *Con*)).

With this definition of the IP, we formalize a decorated interval as a record with three fields, where the third field is a proof that *IP ivl dec* is true.

`Record` *DecIvl* := *DI*

{ *ivl* : *Interval*

; *dec* : *Decoration*

; *invariant* : *IP ivl dec*

}.

containOrder defines the decorated interval containment order (3.3), that is,

$$(\mathbf{x}, d) \subseteq (\mathbf{x}', d') \text{ iff } \mathbf{x} \subseteq \mathbf{x}' \text{ and } d \subseteq d'.$$

`Definition` *containOrder* (*di1 di2* : *DecIvl*) : `Prop` :=

subset (*iSem* (*ivl di1*)) (*iSem* (*ivl di2*)) ∧ *stronger_equal* (*dec di1*) (*dec di2*).

11.1.2 Initial Decoration and Initial Decorated Interval

InitialDec Given an interval $\mathbf{x}$ and a decoration d, *InitialDec* x d asserts that d is an appropriate initial decoration attached to $\mathbf{x}$.

The hypothesis (i) of the FTDIA in §3.2.1 states that each interval $\mathbf{x}_i$ is given an initial decoration d_i, where

- $d_i = \mathsf{saf}$, if $\mathbf{x}_i$ is nonempty and bounded;
- $d_i = \mathsf{def}$, if $\mathbf{x}_i$ is nonempty and unbounded; and
- $d_i = \mathsf{emp}$, if $\mathbf{x}_i$ is empty.

When formalizing the hypothesis (i), we do not need to assign an initial decoration d_i for each $\mathbf{x}_i$, but only assume that each $(\mathbf{x}_i, d_i)$ satisfies the predicate *InitialDec*. We define *InitialDec* via *NEInitialDec*.

NEInitialDec (NonEmpty Initial Decoration) Given an interval $\mathbf{x}$ and a decoration d, *NEInitialDec* x d asserts that

- $\mathbf{x}$ is a nonempty interval, and
- either $\mathbf{x}$ is bounded and $d = \mathsf{saf}$, or $\mathbf{x}$ is unbounded and $d = \mathsf{def}$.

```
Definition NEInitialDec (x : Interval) (d : Decoration) : Prop :=
  isNonEmpty x ∧ ((isBounded x ∧ d = Saf) ∨ (isUnbounded x ∧ d = Def)).
```

NEInitialDec_IvlInv shows that, for any interval $\mathbf{x}$ and decoration d, if $(\mathbf{x}, d)$ satisfies *NEInitialDec*, then it also satisfies the IP.

```
Lemma NEInitialDec_IvlInv :
  ∀ {x : Interval} {d : Decoration}, NEInitialDec x d → IP x d.
```

NEInitialDecIvl Given an interval **x**, a decoration d, and a proof *neInitDec*, which shows that *NEInitialDec x d* is true, then *NEInitialDecIvl x d neInitDec* can generate a decorated interval *DI x d* (*NEInitialDec_IvlInv neInitDec*).

```
Definition NEInitialDecIvl (x : Interval) (d : Decoration)
                           (neInitDec : NEInitialDec x d) : DecIvl :=
  DI x d (NEInitialDec_IvlInv neInitDec).
```

Now we can define *InitialDec* using *NEInitialDec*.

```
Definition InitialDec (x : Interval) (d : Decoration) : Prop :=
  (isEmpty x ∧ d = Emp) ∨ NEInitialDec x d.
```

InitialDec_IvlInv shows that, if (**x**, d) satisfies the initial decoration predicate *InitialDec*, then it must also satisfy the IP.

```
Lemma InitialDec_IvlInv:
  ∀ {x : Interval} {d : Decoration}, InitialDec x d → IP x d.
```

InitialDecIvl constructs an initial decorated interval using (**x**, d), which satisfies the initial decoration predicate *InitialDec*.

```
Definition InitialDecIvl (x : Interval) (d : Decoration)
                         (isInitDec : InitialDec x d) : DecIvl :=
  DI x d (InitialDec_IvlInv isInitDec).
```

11.1.3 Decorated Interval Library Operation

DILO stands for Decorated Interval Library Operation. For any function symbol, its corresponding *DILO* (defined below) instance contains both its point version semantic function *fn* and its interval version semantic function *ifn*, and shows that these two

term semantic functions satisfy the compatibility properties *contains* and *properDec* (defined below).

For an n-ary point-version function f, if there are k intermediate steps of evaluating f, and each intermediate step depends on the input vector $x = (x_1, \ldots, x_n)$, then we write $f = \phi(f_1, \ldots, f_k)$, where each f_i is an n-ary point-version function, and ϕ is a k-ary point-version function, such that $f(x) = \phi\big(f_1(x), \ldots, f_k(x)\big)$.

For its corresponding interval-version function $\mathbf{f}$, we also write $\mathbf{f} = \boldsymbol{\phi}(\mathbf{f}_1, \ldots, \mathbf{f}_k)$, where each $\mathbf{f}_i$ and $\boldsymbol{\phi}$ are the corresponding interval-version functions of f_i and ϕ.

In a decorated interval environment, given a box $\mathbf{X}$, we obtain

- a box $\mathbf{U} = (\mathbf{u}_1, \ldots, \mathbf{u}_k)$, where $\mathbf{u}_i = \mathbf{f}_i(\mathbf{X})$ for $i = 1, \ldots, k$, and a list of decorations $c_1, \ldots, c_k$, where $c_i \supseteq \mathrm{dec}(f_i, \mathbf{X})$, and
- final interval $\mathbf{y} = \boldsymbol{\phi}(\mathbf{U})$ and a decoration d.

We require that any decorated interval library operation satisfies

- $\mathbf{y} \supseteq \mathrm{range}(\phi, \mathbf{U})$, and
- $d = \min(c_0, c_1, \ldots, c_k)$, where $c_0 \supseteq \mathrm{dec}(\phi, \mathbf{U})$.

Any *DILO* instance holds a point version function and its interval extension, such that the two functions should satisfy the properties

- *contains*, that is, $\mathbf{y} \supseteq \mathrm{range}(\phi, \mathbf{u})$
- *properDec*, that is, $d = \min(c_0, c_1, \ldots, c_k)$, where $c_0 \supseteq \mathrm{dec}(\phi, \mathbf{U})$

```
Record DILO (k : nat) :=
{ fn : partialFunType R k
; ifn : Vec DecIvl k → DecIvl
```

; *contains* : $\forall$ *u* : *Vec DecIvl k*,

subset (*range fn* (*map* (*compose iSem ivl*) *u*)) (*iSem* (*ivl* (*ifn u*)))

; *properDec*: $\forall$ *u* : *Vec DecIvl k*, $\exists$ *c0*,

dec (*ifn u*) = *minDecoration c0* (*minVecDecoration* (*map dec u*)) $\wedge$

DecToPred c0 k fn (*map ivl u*)

}.

- *range fn* (*map* (*compose iSem ivl*) *u*) corresponds to range($\phi, \mathbf{u}$)
- *iSem* (*ivl* (*ifn u*)) corresponds to $\mathbf{y}$
- *subset* (*range fn* (*map* (*compose iSem ivl*) *u*)) (*iSem* (*ivl* (*ifn u*))) asserts that $\mathbf{y} \supseteq$ range($\phi, \mathbf{u}$)
- *dec* (*ifn u*) = *minDecoration c0* (*minVecDecoration* (*map dec u*)) asserts that $d = \min(c_0, c_1, \ldots, c_k)$
- *DecToPred c0 k fn* (*map ivl u*) asserts that $c_0 \supseteq \text{dec}(\phi;\ \mathbf{u})$

11.2 Expressions of Decorated Interval Arithmetic

Signature consists of a set of function symbols and their arity information.

An instance of *Signature* has two fields:

- *funSymb* includes all function symbols existing in this signature
- *arity* denotes the arities of the function symbols in this signature

For example, a signature may have $\{+, -, \times, \sin, \cos \ldots\}$. However, we do not fix a particular set of function symbols.

```
Record Signature := Sig
{ funSymb : Set
; arity : funSymb → nat
}.
```

Term is either a variable or applying an n-ary function symbol to n terms.

```
Inductive Term (Sigma : Signature) (n : nat) : Set :=
  | Var : Fin.t n → Term Sigma n
  | App : ∀ (f : funSymb Sigma),
            Vec (Term Sigma n) (arity f) → Term Sigma n.
```

11.3 Semantics of Decorated Interval Arithmetic

DILibrary Given a *Signature* instance *Sigma*, that is, a set of function symbols and their arity information, *DILibrary Sigma* is the type of combined interpretation of *Sigma*. In other words, *DILibrary Sigma* maps each function symbol in *Sigma* to both its point version semantics function *fn* and its interval version semantics function *ifn*, and shows that these two functions satisfy the compatibility properties *contains* and *properDec*.

```
Definition DILibrary (Sigma : Signature) : Type :=
  ∀ f : funSymb Sigma, DILO (arity f).
```

pEval evaluates an expression, i.e. a term, in point mode. It either interprets a variable as a component of its input vector, or recursively interprets the outermost function symbol as a point version function and applies this function to its arguments.

```
Fixpoint pEval {Sigma : Signature} (Lib : DILibrary Sigma)
                  {n : nat} (term : Term Sigma n) (x : Vec R n) : option R :=
  match term with
    | Var i ⇒ Some (nth x i)
    | App f args ⇒ optionBind (fn (Lib f))
                      (vecOp2OpVec (map (fun term ⇒ pEval Lib term x) args))

  end.
```

iEval evaluates an expression in interval mode. That is, it either interprets a variable as an interval of its input box, or recursively interprets the outermost function symbol as an interval version function, and applies this function to its arguments.

```
Fixpoint iEval {Sigma : Signature} (Lib : DILibrary Sigma) {n : nat}
                  (term : Term Sigma n) (X : Vec DecIvl n) : DecIvl :=
  match term with
    | Var i ⇒ nth X i
    | App f args ⇒ ifn (Lib f) (map (fun term ⇒ iEval Lib term X) args)
  end.
```

Chapter 12

Proof of the FTDIA_NonEmptyBox

We first prove the *FTIA* in Section 12.1, and then we prove the *FTDIA_NonEmptyBox* in Section 12.2.

12.1 Proof of the FTIA

FTIA. We show the property range$(f, \mathbf{X}) \subseteq \mathbf{y}$ of Theorem 3.1.1.

```
Theorem FTIA :
 ∀ {Sigma : Signature} {Lib : DILibrary Sigma} {n : nat}
   (t : Term Sigma n) {X : Box n} {D : Vec Decoration n}
   (iniDec : vecAll2 InitialDec X D),
  let iEval' := fun t ⇒ iEval Lib t (vecAll2_map InitialDecIvl iniDec) in
  let yd := iEval' t
  in subset (range (pEval Lib t) (map iSem X)) (iSem (ivl yd)).
```

- *Sigma* is an instance of *Signature* (§11.2)
- *Lib* maps any function symbol into a point version function *fn* and its interval extension *ifn*, such that they satisfy the two properties of DILO
- *vecAll2_map InitialDecIvl iniDec* creates a vector of initial decorated intervals
- *iEval Lib t* (*vecAll2_map InitialDecIvl iniDec*) evaluates the term t in decorated interval mode
- *yd* denotes the final decorated interval result $(\mathbf{y}, d)$
- *iSem* (*ivl yd*) denotes $\mathbf{y}$
- *range* (*pEval Lib t*) (*map iSem X*) denotes $\text{range}(f, \mathbf{X})$
- *subset* (*range* (*pEval Lib t*) (*map iSem X*)) (*iSem* (*ivl yd*)) asserts $\text{range}(f, \mathbf{X}) \subseteq \mathbf{y}$

Essential parts of the mechanized proof

1. Base Case: Given a projection function π_i, for $i = 1, \ldots, n$, and a box $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$,
   - if any $\mathbf{x}_j$, for $j = 1, \ldots, n$, is an empty interval, then $\mathbf{X}$ is an empty box. Since $\text{range}(\pi_i, \mathbf{X})$ is empty, $\text{range}(\pi_i, \mathbf{X}) \subseteq \mathbf{y}$ holds trivially;
   - if each $\mathbf{x}_j$, for $j = 1, \ldots, n$, is a nonempty interval, then $\text{range}(\pi_i, \mathbf{X}) = \mathbf{x}_i = \mathbf{y}$
2. Induction Case: We have $f = \phi(f_1, \ldots, f_k)$, where each f_i is an n-ary point-version function, and ϕ is a k-ary point-version function, such that

$f(x) = \phi\big(f_1(x), \ldots, f_k(x)\big)$. For each $i = 1, \ldots, k$, there is a corresponding computed intermediate interval $\mathbf{u}_i$ that encloses range$(f_i, \mathbf{X})$ by the induction hypothesis. Thus, we have $\mathbf{U} = (\mathbf{u}_1, \ldots, \mathbf{u}_k)$. Since the decorated interval version of ϕ satisfies DILO, then range$(\phi, \mathbf{U}) \subseteq \mathbf{y}$.

Thus, we show the *FTIA*.

12.2 Proof of the FTDIA_NonEmptyBox

In the proof of the *FTDIA* [29], Pryce shows that the five decoration system fails at the base case (c), which says that if we have

1. a projection function π_k, which projects the k-th component from an n-vector, and
2. a box $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$, where
   - for some i, $\mathbf{x}_i = (\emptyset, \mathsf{emp})$, and
   - for some k, $\mathbf{x}_k$ is a nonempty set with the decoration saf,

then we know that the box $\mathbf{X}$ is empty and dec$(\pi_k, \mathbf{X}) = \mathsf{emp}$. However, if we evaluate the projection π_k on $\mathbf{X}$ in decorated interval-mode, we obtain an interval $\mathbf{x}_k$ with the decoration saf, since the projection function is not a DILO instance. The base case (c) fails for saf does not enclose emp.

However, based on the assumption of our formal system, we can simply screen out the base case (c) by checking if any of the components of an input box is empty. Hence, we formalize and prove the theorem *FTDIA_NonEmptyBox* instead, which accounts for all non-trivial parts of the *FTDIA*.

```
Theorem FTDIA_NonEmptyBox :
 ∀ {Sigma : Signature} {Lib : DILibrary Sigma} {n : nat}
   (t : Term Sigma n) {X : Box n} {D : Vec Decoration n}
   (iniDec : vecAll2 NEInitialDec X D) (nonEmptyBox : ¬ isEmptyBox X)
   {decfX : Decoration}
   (decfXStrongest : HasStrongestDec X (pEval Lib t) decfX),
  let yd := iEval Lib t (vecAll2_map NEInitialDecIvl iniDec)
  in subset (range (pEval Lib t) (map iSem X)) (iSem (ivl yd)) ∧
     stronger_equal decfX (dec yd).
```

- *HasStrongestDec X (pEval Lib t) decfX* asserts that *decfX* is the strongest decoration for $(f, \mathbf{X})$, that is $decfX = \text{dec}(f, \mathbf{X})$
- *stronger_equal decfX (dec yd)* assert $decfX \subseteq d$

Essential parts of the mechanized proof

To prove the *FTDIA_NonEmptyBox*, we need to prove two subgoals

(i) $\text{range}(f, \mathbf{X}) \subseteq \mathbf{y}$, and

(ii) $\text{dec}(f, \mathbf{X}) \subseteq d$.

The first subgoal $\text{range}(f, \mathbf{X}) \subseteq \mathbf{y}$ is proved by the lemma *FTIA_NonEmptyBox*. The second subgoal $\text{dec}(f, \mathbf{X}) \subseteq d$ can be solved in two steps. We first show that $(f, \mathbf{X})$ satisfies the predicate p_d using the lemma *pCmin_NonEmptyBox*. Then, we use the lemma *d_StrongerThan_cMin* to prove that for any strongest decoration $d' = \text{dec}(f, \mathbf{X})$, if $(f, \mathbf{X})$ satisfies the predicate p_d, then $d' \subseteq d$ is always true. From the two steps above, we prove $\text{dec}(f, \mathbf{X}) \subseteq d$.

Chapter 13

Conclusion

We provide a formal proof of the FTDIA (as given in draft 3.0 of the IEEE P1788 proposal) for the common case, where the input box is non-empty. To cover the case of a box with components that are empty sets, and all components are used in a function evaluation, we should define some predicate to assert that an empty interval in a box is used in the decorated interval evaluation. However, it appears difficult to formalize and prove the FTDIA that supports the case of an empty input box. Although obvious, an empty input box results in an empty output interval with a decoration emp or ill.

During the development of our formal proof, the P1788 drafts evolved, now at draft 8.1, which contains decorations as follows:

Decoration	Short description	Property	Definition
com	common	$p_{\mathsf{com}}(f,x)$	x is a bounded, nonempty subset of $\mathrm{dom}(f)$; f is continuous at each point of x; and the computed interval $f(x)$ is bounded.
dac	defined & continuous	$p_{\mathsf{dac}}(f,x)$	x is a nonempty subset of $\mathrm{dom}(f)$, and the restriction of f to x is continuous;
def	defined	$p_{\mathsf{def}}(f,x)$	x is a nonempty subset of $\mathrm{dom}(f)$;
trv	trivial	$p_{\mathsf{trv}}(f,x)$	always true (so gives no information);
ill	ill-formed	$p_{\mathsf{ill}}(f,x)$	Not an Interval; formally $\mathrm{dom}(f)=\emptyset$

The decorations in the FDTIA_NonEmptyBox we proved compare to the new set of decorations as:

current proof	version 8.1
—	com
saf (safe)	—
—	dac
def (defined)	def
con (always true)	trv
emp (empy)	—
ill	ill

To provide a proof of the most up-to-date version of the FTDIA, one needs to change the formalization of the decoration system and modify the proof of Lemma C3 by case analysis on the new decoration system. We do not expect other proofs to change. This is left as future work, along with the formalization of the case when a box contains as components the empty sets.

In general, to prove the FTDIA with a set of given decorations, one needs to

- formalize the new decoration system
- provide the equivalent of Lemma C3, and
- possibly update Lemma C2, depending on the properties of the new decorations.

That is, the machinery behind the boxes marked as Lemma C2, Lemma C3, and Decoration in Figure 4.1 needs re-developing.

In the course of the development of our formal proof, we learn some lessons. First, we notice that there are often more than one way to formalize a concept and its properties. An appropriate formalization can greatly simplify the proof related to the concept and its properties. Second, it can be helpful to give different formal specifications for the same notion and show the equivalence between them. That is, different definitions can give different points of view about a concept. Finally, handling partial functions required considerable effort due to dealing with both defined and undefined cases, where the latter can be involved. In particular, we needed to deal with the undefined cases of partial functions in the proofs of Lemma C2 and Lemma C3.

Coq Scripts

The Coq scripts of this thesis can be downloaded as the compressed file `FTDIA.zip` from `http://www.cas.mcmaster.ca/~nedialk/FTDIA/FTDIA.zip`. When this file is uncompressed, the directory `FTDIA` with subdirectories `src` and `latex` are created. The files `*.v` in `src` contain both Coq scripts and LaTeX in a literate programming style.

Typing `make coq` in `FTDIA` compiles and checks the proof scripts using `coqc`. Successful compilations ensures that our proof is correct according to Coq.

Typing `make doc` extracts from each `*.v` file a LaTeX file, which is placed in subdirectory `latex`. Then `make` compiles the main file `FTDIA.tex` to produce `FTDIA.pdf`, which is this document.

Executing `make` has the effect of executing `make coq` and then `make doc`.

Appendix A

Auxiliary Lemmas

A.1 Auxiliary Lemmas for Lemma C2

A.1.1 Lemmas for C2a

AllfiDefined_to_fDefined shows that, if each f_i in f is everywhere defined on $\mathbf{X}$, then the partial vector function f is everywhere defined on $\mathbf{X}$.

```
Lemma AllfiDefined_to_fDefined :
 ∀ {n k : nat}
   {f : Vec (partialFunType R n) k}
   {x : Vec R n} {X : Vec Interval n},
   vecAll (fun f_i : partialFunType R n ⇒
               ∀ x : Vec R n, setVecProd (map iSem X) x →
                 isSome (f_i x)) f →
   setVecProd (map iSem X) x →
  isSome (vecOp2OpVec (mapApp f x)).
```

AllfiDefined_to_fDefined2 also proves that, if each f_i of f is everywhere defined

on $\mathbf{X}$, then the partial vector function f is everywhere defined on $\mathbf{X}$.

```
Lemma AllfiDefined_to_fDefined2 :
 ∀ {n k : nat}
   {f : Vec (partialFunType R n) (S k)} {X : Box n},
   vecAll (fun f_i : partialFunType R n ⇒ pDefinedOnBox f_i X) f →
 ∀ (x : Vec R n), bSem X x →
  ∃ y : Vec R (S k), vecOp2OpVec (mapApp f x) = Some y.
```

A.1.2 Lemmas for C2b

fX_in_U shows that, given

- a vector of functions $f = (f_1, \ldots, f_{k+1})$,
- two boxes $\mathbf{X}$ and $\mathbf{U}$, such that $\mathbf{u}_i \supseteq \text{range}(f_i, \mathbf{X})$, for $i = 1, ..., k+1$,

for any $x \in \mathbf{X}$, if *vecOp2OpVec (mapApp f x) = Some y*, then $y \in \mathbf{U}$.

```
Lemma fX_in_U :
 ∀ {n k : nat}
   {f : Vec (partialFunType R n) (S k)}
   {x : Vec R n} {X : Box n} {U : Box (S k)},
   vecAll2 subset (mapBoxApp f (map iSem X)) (map iSem U) → bSem X x →
 ∀ y : Vec R (S k), vecOp2OpVec (mapApp f x) = Some y → bSem U y.
```

The continuity of partial vector function f

If each f_i in f is continuous on $\mathbf{X}$, then f is obviously continuous on $\mathbf{X}$. This fact is shown in Coq by Lemma *fContinuous*. We prove *fContinuous* by induction. The

base case is shown by Lemma *fContinousBaseCase*, and the induction case is shown by Lemma *fContinousInductionCase*.

```
Lemma fContinousBaseCase :
 ∀ {n : nat}
   (f : Vec (partialFunType R n) 1) (X : Box n),
   vecAll (fun f_i : partialFunType R n ⇒ pDefinedOnBox f_i X) f →
   vecAll (fun f_i: partialFunType R n ⇒ pContinuousOnBox f_i X) f →
 pContinuousOnBox_vec f X.

Lemma fContinousInductionCase :
 ∀ {n k : nat}
   (f : Vec (partialFunType R n) (S (S k))) (X : Box n),
   vecAll (fun f_i : partialFunType R n ⇒ pDefinedOnBox f_i X) f →
   vecAll (fun f_i: partialFunType R n ⇒ pContinuousOnBox f_i X) f →
   (∀ f : Vec (partialFunType R n) (S k),
       vecAll (fun f_i : partialFunType R n ⇒ pDefinedOnBox f_i X) f →
       vecAll (fun f_i : partialFunType R n ⇒ pContinuousOnBox f_i X) f →
    pContinuousOnBox_vec f X) →
 pContinuousOnBox_vec f X.

Lemma fContinuous :
 ∀ {n k : nat}
   (f : Vec (partialFunType R n) (S k)) (X : Box n),
   vecAll (fun f_i : partialFunType R n ⇒ pDefinedOnBox f_i X) f →
   vecAll (fun f_i: partialFunType R n ⇒ pContinuousOnBox f_i X) f →
 pContinuousOnBox_vec f X.
```

- *pContinuousOnBox f_i X* §7.3.2 asserts that f_i is continuous on $\mathbf{X}$
- *pContinuousOnBox_vec f X* §7.3.2 asserts that the partial vector function f is continuous on $\mathbf{X}$

Essential parts of the mechanized proof

Assuming that each $f_i : \mathbb{R}^n \nrightarrow \mathbb{R}$ in f, for $i = 1, \cdots, k+1$, is defined and continuous on the box $\mathbf{X}$, then the partial vector function f is continuous on $\mathbf{X}$.

We prove this goal by induction.

Base case ($k = 1$). f is just f_1. From the hypothesis, we know f_1 is defined and continuous on $\mathbf{X}$, so is f. The base case is proved in Lemma *fContinousBaseCase*.

Induction case ($k > 1$). We assume that each f_i in f is defined and continuous on $\mathbf{X}$, and the induction hypothesis is true, that is, $f' = (f_2, \cdots, f_{k+2})$ is continuous on $\mathbf{X}$. Then,

f_1 is continuous on $\mathbf{X}$,

$$\begin{aligned} &\forall\, x_{f_1} \in \mathbf{X}. \forall\, \epsilon_{f_1} \in \mathbb{R}. \epsilon_{f_1} > 0. \exists\, \delta_{f_1} \in \mathbb{R}^n. \delta_{f_1} > 0. \forall\, {x_{f_1}}' \in \mathbf{X}. \\ &\qquad\qquad | \, {x_{f_1}}' - x_{f_1} \, | < \delta_{f_1} \Rightarrow | \, f_1({x_{f_1}}') - f_1(x_{f_1}) \, | < \epsilon_{f_1} \end{aligned} \tag{A.15}$$

f' is continuous on $\mathbf{X}$,

$$\begin{aligned} &\forall\, x_{f'} \in \mathbf{X}. \forall\, \epsilon_{f'} \in \mathbb{R}^{k+1}. \epsilon_{f'} > 0. \exists\, \delta_{f'} \in \mathbb{R}^n. \delta_{f'} > 0. \forall\, {x_{f'}}' \in \mathbf{X}. \\ &\qquad\qquad | \, {x_{f'}}' - x_{f'} \, | < \delta_{f'} \Rightarrow | \, f'({x_{f'}}') - f'(x_{f'}) \, | < \epsilon_{f'} \end{aligned} \tag{A.16}$$

Our goal is:

$$\begin{aligned} &\forall\, x \in \mathbf{X}. \forall\, \epsilon \in \mathbb{R}^{k+2}. \epsilon > 0. \exists\, \delta \in \mathbb{R}^n. \delta > 0. \forall\, x' \in \mathbf{X}. \\ &\qquad\qquad | \, x' - x \, | < \delta \Rightarrow | \, f(x') - f(x) \, | < \epsilon \end{aligned} \tag{A.17}$$

Our proof follows the following steps.

1. We fix x and ϵ in (A.17), where $\epsilon = (\epsilon_1, \cdots, \epsilon_{k+2})$. Then our goal becomes

$$\exists\, \delta \in \mathbb{R}^n . \delta > 0 . \forall\, x' \in \mathbf{X} . \mid x' - x \mid < \delta \Rightarrow \mid f(x') - f(x) \mid < \epsilon \tag{A.18}$$

2. Let $x_{f_1} = x$ and $\epsilon_{f_1} = \epsilon_1$ in (A.15). Then there exists a $\delta_{f_1} > 0$, such that

$$\forall\, {x_{f_1}}' \in \mathbf{X} . \mid {x_{f_1}}' - x \mid < \delta_{f_1} \Rightarrow \mid f_1({x_{f_1}}') - f_1(x) \mid < \epsilon_1 \tag{A.19}$$

3. Let $x_{f'} = x$ and $\epsilon_{f'} = (\epsilon_2, \cdots, \epsilon_{k+2})$ in (A.16). Then there exists a $\delta_{f'} > 0$, such that

$$\forall\, {x_{f'}}' \in \mathbf{X} . \mid {x_{f'}}' - x \mid < \delta_{f'} \Rightarrow \mid f'({x_{f'}}') - f'(x) \mid < (\epsilon_2, \cdots, \epsilon_{k+2}) \tag{A.20}$$

4. Let $\delta = \min(\delta_{f_1}, \delta_{f'})$ and fix x' in (A.18). Then our new goal is

$$\mid x' - x \mid < \min(\delta_{f_1}, \delta_{f'}) \Rightarrow \mid f(x') - f(x) \mid < \epsilon \tag{A.21}$$

5. Let ${x_{f_1}}' = x'$ in (A.19)

$$\mid x' - x \mid < \delta_{f_1} \Rightarrow \mid f_1(x') - f_1(x) \mid < \epsilon_1 \tag{A.22}$$

6. Let ${x_{f'}}' = x'$ in (A.20)

$$\mid x' - x \mid < \delta_{f'} \Rightarrow \mid f'(x') - f'(x) \mid < (\epsilon_2, \cdots, \epsilon_{k+2}) \tag{A.23}$$

7. From the hypothesis of (A.21), (A.22), and (A.23), we obtain

$$\begin{aligned} &\mid x' - x \mid < \min(\delta_{f_1}, \delta_{f'}) \Rightarrow \\ (\mid f_1(x') - f_1(x) \mid < \epsilon_1 \wedge &\mid f'(x') - f'(x) \mid < (\epsilon_2, \cdots, \epsilon_{k+2})) \Rightarrow \\ &\mid f(x') - f(x) \mid < \epsilon \end{aligned} \tag{A.24}$$

which shows (A.21).

Induction case is proved in Lemma *fContinousInductionCase*.

A.1.3 Lemma for C2e

f_iEmptyDomain_fEmptyDomain shows that, if any f_i of f has empty domain, then the partial vector function f has empty domain.

```
Lemma f_iEmptyDomain_fEmptyDomain :
 ∀ {n k : nat}
    {f : Vec (partialFunType R n) (S k)},
    (∃ f_i : partialFunType R n, Vector.In f_i f ∧ emptyDomain f_i) →
 ∀ x : Vec R n, vecOp2OpVec (mapApp f x) = None.
```

A.2 Auxiliary Lemmas for Lemma C3

A.2.1 C3a_pSaf

AllfiXSaf_AllfiXpSaf shows that, if each $(f_i, \mathbf{X})$ has the strongest decoration saf for $i = 1, \ldots, k+1$, then all of them satisfy *pSaf*.

```
Lemma AllfiXSaf_AllfiXpSaf :
```

$\forall\ \{n\ k : nat\}$

$(f : Vec\ (partialFunType\ R\ n)\ (S\ k))\ (X : Box\ n)\ (cs : Vec\ Decoration\ (S\ k)),$

$vecAll2\ (\texttt{fun}\ fi\ di \Rightarrow DecToPred\ di\ \_\ fi\ X)\ f\ cs \rightarrow$

$vecAll\ (\texttt{fun}\ c\_i : Decoration \Rightarrow c\_i = Saf)\ cs \rightarrow$

$vecAll\ (\texttt{fun}\ f\_i \Rightarrow pSaf\ n\ f\_i\ X)\ f.$

AllfiXSaf_AllfiDefinedOnX shows that, if each $(f_i, \mathbf{X})$ has the strongest decoration saf for $i = 1, \ldots, k+1$, then each f_i is defined on $\mathbf{X}$.

Lemma *AllfiXSaf_AllfiDefinedOnX* :

$\forall\ \{n\ k : nat\}$

$(f : Vec\ (partialFunType\ R\ n)\ (S\ k))\ (X : Box\ n)\ (cs : Vec\ Decoration\ (S\ k)),$

$vecAll2\ (\texttt{fun}\ f\_i\ d\_i \Rightarrow DecToPred\ d\_i\ \_\ f\_i\ X)\ f\ cs \rightarrow$

$vecAll\ (\texttt{fun}\ d\_i : Decoration \Rightarrow d\_i = Saf)\ cs \rightarrow$

$vecAll\ (\texttt{fun}\ f\_i : partialFunType\ R\ n \Rightarrow pDefinedOnBox\ f\_i\ X)\ f.$

C3a_pSaf_nonEmpty shows that, if each $(f_i, \mathbf{X})$ has the strongest decoration saf for $i = 1, \ldots, k+1$, then $\mathbf{X}$ is not empty.

Lemma *C3a_pSaf_nonEmptySet* :

$\forall\ \{n\ k : nat\}$

$(f : Vec\ (partialFunType\ R\ n)\ (S\ k))\ (X : Box\ n)\ (cs : Vec\ Decoration\ (S\ k)),$

$vecAll2\ (\texttt{fun}\ f\_i\ d\_i \Rightarrow DecToPred\ d\_i\ \_\ f\_i\ X)\ f\ cs \rightarrow$

$vecAll\ (\texttt{fun}\ c\_i : Decoration \Rightarrow c\_i = Saf)\ cs \rightarrow$

$nonEmptySet\ (bSem\ X)\ .$

C3a_pSaf_inDomain shows that, if each $(f_i, \mathbf{X})$ has the strongest decoration saf and $\text{range}(f_i, \mathbf{X}) \subseteq \mathbf{u}_i$ for $i = 1, \ldots, k+1$, and $(\phi, \mathbf{U})$ has the strongest decoration saf, then $\mathbf{X} \subseteq \text{domain}(\phi \circ f)$.

Lemma *C3a_pSaf_inDomain* :

$\forall\ \{n\ k : nat\}$

$(f : Vec\ (partialFunType\ R\ n)\ (S\ k))\ (X : Box\ n)\ (cs : Vec\ Decoration\ (S\ k))$

$(phi : partialFunType\ R\ (S\ k))\ (U : Box\ (S\ k))\ (c0 : Decoration),$

$vecAll2\ (\texttt{fun}\ fi\ di \Rightarrow DecToPred\ di\ \_\ fi\ X)\ f\ cs \rightarrow$

$DecToPred\ c0\ \_\ phi\ U \rightarrow$

$vecAll\ (\texttt{fun}\ c\_i : Decoration \Rightarrow c\_i = Saf)\ cs \rightarrow c0 = Saf \rightarrow$

$vecAll2\ subset\ (mapBoxApp\ f\ (map\ iSem\ X))\ (map\ iSem\ U) \rightarrow$

$subset\ (bSem\ X)\ (domain\ (composition\ f\ phi)).$

C3a_pSaf_pContinuousOnBox shows that, if each $(f_i, \mathbf{X})$ has the strongest decoration saf and range$(f_i, \mathbf{X}) \subseteq \mathbf{u}_i$ for $i = 1, \ldots, k+1$, and $(\phi, \mathbf{U})$ has the strongest decoration saf, then $\phi \circ f$ is continuous on $\mathbf{X}$.

Lemma *C3a_pSaf_pContinuousOnBox* :

$\forall\ \{n\ k : nat\}$

$(f : Vec\ (partialFunType\ R\ n)\ (S\ k))\ (X : Box\ n)\ (cs : Vec\ Decoration\ (S\ k))$

$(phi : partialFunType\ R\ (S\ k))\ (U : Box\ (S\ k))\ (c0 : Decoration),$

$vecAll2\ (\texttt{fun}\ fi\ di \Rightarrow DecToPred\ di\ \_\ fi\ X)\ f\ cs \rightarrow$

$DecToPred\ c0\ \_\ phi\ U \rightarrow$

$vecAll\ (\texttt{fun}\ c\_i : Decoration \Rightarrow c\_i = Saf)\ cs \rightarrow c0 = Saf \rightarrow$

$vecAll2\ subset\ (mapBoxApp\ f\ (map\ iSem\ X))\ (map\ iSem\ U) \rightarrow$

$pContinuousOnBox\ (composition\ f\ phi)\ X.$

phiUSaf_phiUpBounded shows that, if $(\phi, \mathbf{U})$ satisfies *pSaf*, and each $u_i \in \mathbf{U}$ is bounded, for $i = 1, \ldots, k+1$, then the range of ϕ on $\mathbf{U}$ is bounded.

Lemma *phiUSaf_phiUpBounded* :

$\forall\ \{k : nat\}\ (phi : partialFunType\ R\ (S\ k))\ (U : Box\ (S\ k)),$

$DecToPred\ Saf\ \_\ phi\ U \rightarrow$

pBoundedSet_vec (*map iSem U*) →

pBounded phi (*map iSem U*).

C3a_pSaf_pBounded shows that, if each $(f_i, \mathbf{X})$ has the strongest decoration saf and range$(f_i, \mathbf{X}) \subseteq \mathbf{u}_i$ for $i = 1, \ldots, k+1$, and $(\phi, \mathbf{U})$ has the strongest decoration saf, then the range of $\phi \circ f$ on $\mathbf{X}$ is bounded.

```
Lemma C3a_pSaf_pBounded :
  ∀ {n k : nat}
    (f : Vec (partialFunType R n) (S k)) (X : Box n) (cs : Vec Decoration (S k))
    (phi : partialFunType R (S k)) (U : Box (S k)) (c0 : Decoration),
    vecAll2 (fun fi di ⇒ DecToPred di _ fi X) f cs →
    DecToPred c0 _ phi U →
    vecAll (fun c_i : Decoration ⇒ c_i = Saf) cs → c0 = Saf →
    vecAll2 subset (mapBoxApp f (map iSem X)) (map iSem U) →
  pBounded (composition f phi) (map iSem X).
```

C3a_pSaf shows that, if a worst quality decoration of $(c_0, c_1, \ldots, c_{k+1})$ is saf, each $(f_i, \mathbf{X})$ satisfies p_{c_i} and range$(f_i, \mathbf{X}) \subseteq \mathbf{u}_i$, for $i = 1, \ldots, k+1$, and $(\phi, \mathbf{U})$ satisfies p_{c_0}, then $(\phi \circ f, \mathbf{X})$ satisfies *pSaf*.

```
Lemma C3a_pSaf :
  ∀ {n k : nat}
    (f : Vec (partialFunType R n) (S k)) (X : Box n) (cs : Vec Decoration (S k))
    (phi : partialFunType R (S k)) (U : Box (S k)) (c0 : Decoration),
    vecAll2 (fun fi di ⇒ DecToPred di _ fi X) f cs →
    DecToPred c0 _ phi U →
    minDecoration c0 (minVecDecoration cs) = Saf →
    vecAll2 subset (mapBoxApp f (map iSem X)) (map iSem U) →
```

pSaf n (composition f phi) X.

A.2.2 C3a_pDef

AllfiXSafDef_AllfiXpDef shows that, if each $(f_i, \mathbf{X})$ has the strongest decoration saf or def, for $i = 1, \ldots, k+1$, then all of them satisfy *pDef*.

```
Lemma AllfiXSafDef_AllfiXpDef :
 ∀ {n k : nat}
   (f : Vec (partialFunType R n) (S k)) (X : Box n) (cs : Vec Decoration (S k)),
   vecAll2 (fun fi di ⇒ DecToPred di _ fi X) f cs →
   vecAll (fun d : Decoration ⇒ d = Saf ∨ d = Def) cs →
 vecAll (fun f_i ⇒ pDef n f_i X) f.
```

AllfiXSafDef_AllfiXpDefinedOnBox shows that, if each $(f_i, \mathbf{X})$ has the strongest decoration saf or def, for $i = 1, \ldots, k+1$, then each f_i is defined on $\mathbf{X}$.

```
Lemma AllfiXSafDef_AllfiXpDefinedOnBox :
 ∀ {n k : nat}
   (f : Vec (partialFunType R n) (S k)) (X : Box n) (cs : Vec Decoration (S k)),
   vecAll2 (fun fi di ⇒ DecToPred di _ fi X) f cs →
   vecAll (fun d : Decoration ⇒ d = Saf ∨ d = Def) cs →
 vecAll (fun f_i : partialFunType R n ⇒ pDefinedOnBox f_i X) f.
```

C3a_pDef_nonEmptySet shows that, if a worst quality decoration of the tuple $(c_0, c_1, \ldots, c_{k+1})$ is def, and each $(f_i, \mathbf{X})$ satisfies p_{c_i}, for $i = 1, \ldots, k+1$, then $\mathbf{X}$ is not empty.

```
Lemma C3a_pDef_nonEmptySet :
 ∀ {n k : nat}
```

$(f : Vec\ (partialFunType\ R\ n)\ (S\ k))\ (X : Box\ n)$

$(cs : Vec\ Decoration\ (S\ k))\ (c0 : Decoration)$,

$vecAll2\ (\texttt{fun}\ fi\ di \Rightarrow DecToPred\ di\ \_\ fi\ X)\ f\ cs \rightarrow$

$minDecoration\ c0\ (minVecDecoration\ cs) = Def \rightarrow$

$nonEmptySet\ (bSem\ X)$.

C3a_pDef_inDomain shows that, if a worst quality decoration of $(c_0, c_1, \ldots, c_{k+1})$ is def, each $(f_i, \mathbf{X})$ satisfies p_{c_i} and $\text{range}(f_i, \mathbf{X}) \subseteq \mathbf{u}_i$, for $i = 1, \ldots, k+1$, and $(\phi, \mathbf{U})$ satisfies p_{c_0}, then $\mathbf{X} \subseteq \text{domain}(\phi \circ f)$.

Lemma *C3a_pDef_inDomain* :

$\forall\ \{n\ k : nat\}$

$(f : Vec\ (partialFunType\ R\ n)\ (S\ k))\ (X : Box\ n)\ (cs : Vec\ Decoration\ (S\ k))$

$(phi : partialFunType\ R\ (S\ k))\ (U : Box\ (S\ k))\ (c0 : Decoration)$,

$vecAll2\ (\texttt{fun}\ fi\ di \Rightarrow DecToPred\ di\ \_\ fi\ X)\ f\ cs \rightarrow$

$DecToPred\ c0\ \_\ phi\ U \rightarrow$

$minDecoration\ c0\ (minVecDecoration\ cs) = Def \rightarrow$

$vecAll2\ subset\ (mapBoxApp\ f\ (map\ iSem\ X))\ (map\ iSem\ U) \rightarrow$

$subset\ (bSem\ X)\ (domain\ (composition\ f\ phi))$.

C3a_pDef shows that, if a worst quality decoration of $(c_0, c_1, \ldots, c_{k+1})$ is def, each $(f_i, \mathbf{X})$ satisfies p_{c_i} and $\text{range}(f_i, \mathbf{X}) \subseteq \mathbf{u}_i$, for $i = 1, \ldots, k+1$, and $(\phi, \mathbf{U})$ satisfies p_{c_0}, then $(\phi \circ f, \mathbf{X})$ satisfies *pDef*.

Lemma *C3a_pDef* :

$\forall\ \{n\ k : nat\}$

$(f : Vec\ (partialFunType\ R\ n)\ (S\ k))\ (X : Box\ n)\ (cs : Vec\ Decoration\ (S\ k))$

$(phi : partialFunType\ R\ (S\ k))\ (U : Box\ (S\ k))\ (c0 : Decoration)$,

$vecAll2\ (\texttt{fun}\ fi\ di \Rightarrow DecToPred\ di\ \_\ fi\ X)\ f\ cs \rightarrow$

DecToPred c0 _ phi U →

minDecoration c0 (minVecDecoration cs) = Def →

vecAll2 subset (mapBoxApp f (map iSem X)) (map iSem U) →

pDef n (composition f phi) X.

A.2.3 C3a_pEmp

C3a_pEmp_c0Emp shows that, given a vector of functions $f = (f_1, \ldots, f_{k+1})$, if range$(f_i, \mathbf{X}) \subseteq \mathbf{u}_i$, for $i = 1, \ldots, k+1$, and $(\phi, \mathbf{U})$ satisfies *pEmp*, then $(\phi \circ f, \mathbf{X})$ satisfies *pEmp*.

Lemma *C3a_pEmp_c0Emp* :

∀ {*n k* : *nat*}

(*f* : *Vec* (*partialFunType R n*) (*S k*)) (*X* : *Box n*)

(*phi* : *partialFunType R* (*S k*)) (*U* : *Box* (*S k*)) (*c0* : *Decoration*),

DecToPred c0 _ phi U → c0 = Emp →

vecAll2 subset (*mapBoxApp f* (*map iSem X*)) (*map iSem U*) →

pEmp n (*composition f phi*) *X.*

ciEmp shows that, given a vector of decorations $cs = (c_1, \ldots, c_{k+1})$, if there is a decoration emp in *cs*, then there exists some i, such that the i-th component of *cs* is emp.

Lemma *ciEmp* :

∀ {*k* : *nat*} (*cs* : *Vec Decoration* (*S k*)), *Vector.In Emp cs →*

∃ *i*, *nth cs i = Emp.*

ciEmp_fiXpEmp shows that, given a vector of functions $f = (f_1, \ldots, f_{k+1})$ and a vector of decorations $cs = (c_1, \ldots, c_{k+1})$, if each $(f_i, \mathbf{X})$ has the strongest decoration

c_i, for $i = 1, \ldots, k+1$, and there is a decoration emp in cs, then there exists some f_i in f, such that $(f_i, \mathbf{X})$ satisfies *pEmp*.

```
Lemma ciEmp_fiXpEmp :
  ∀ {n k : nat}
    (f : Vec (partialFunType R n) (S k)) (X : Box n) (cs : Vec Decoration (S k)),
    vecAll2 (fun fi di ⇒ DecToPred di _ fi X) f cs → Vector.In Emp cs →
  ∃ f_i : partialFunType R n, Vector.In f_i f ∧ pEmp n f_i X.
```

C3a_pEmp_ciEmp shows that, given a vector of functions $f = (f_1, \ldots, f_{k+1})$ and a vector of decorations $cs = (c_1, \ldots, c_{k+1})$, if each $(f_i, \mathbf{X})$ has the strongest decoration c_i, for $i = 1, \ldots, k+1$, and there is no decoration ill but emp in cs, then $(\phi \circ f, \mathbf{X})$ satisfies *pEmp*.

```
Lemma C3a_pEmp_ciEmp :
  ∀ {n k : nat}
    (f : Vec (partialFunType R n) (S k)) (X : Box n) (cs : Vec Decoration (S k))
    (phi : partialFunType R (S k)),
    vecAll2 (fun fi di ⇒ DecToPred di _ fi X) f cs →
    vecAll (fun c_i : Decoration ⇒ c_i ≠ Ill) cs ∧
     (∃ c_i : Decoration, Vector.In c_i cs ∧ c_i = Emp) →
  pEmp n (composition f phi) X.
```

C3a_pEmp shows that, given a vector of functions $f = (f_1, \ldots, f_{k+1})$, if a worst quality decoration of $(c_0, c_1, \ldots, c_{k+1})$ is emp, each $(f_i, \mathbf{X})$ satisfies p_{c_i} and range$(f_i, \mathbf{X}) \subseteq \mathbf{u}_i$, for $i = 1, \ldots, k+1$, and $(\phi, \mathbf{U})$ satisfies p_{c_0}, then $(\phi \circ f, \mathbf{X})$ satisfies *pEmp*.

```
Lemma C3a_pEmp :
  ∀ {n k : nat}
    (f : Vec (partialFunType R n) (S k)) (X : Box n) (cs : Vec Decoration (S k))
```

(*phi* : *partialFunType R* (*S k*)) (*U* : *Box* (*S k*)) (*c0* : *Decoration*),

vecAll2 (fun *fi di* ⇒ *DecToPred di* _ *fi X*) *f cs* →

DecToPred c0 _ *phi U* →

minDecoration c0 (*minVecDecoration cs*) = *Emp* →

vecAll2 subset (*mapBoxApp f* (*map iSem X*)) (*map iSem U*) →

pEmp n (*composition f phi*) *X*.

A.2.4 C3a_pIll

ciIll shows that, given a vector of decorations $cs = (c_1, \ldots, c_{k+1})$, if there is a decoration ill in cs, then there exists some i, such that the i-th component of cs is ill.

Lemma *ciIll* :

∀ {*k* : *nat*} (*cs* : *Vec Decoration* (*S k*)), *Vector.In Ill cs* → ∃ *i*, *nth cs i* = *Ill*.

ciIll_fiXpIll shows that, given a vector of functions $f = (f_1, \ldots, f_{k+1})$ and a vector of decorations $cs = (c_1, \ldots, c_{k+1})$, if each $(f_i, \mathbf{X})$ has the strongest decoration c_i, for $i = 1, \ldots, k+1$, and there is a decoration ill in cs, then there exists some f_i in f, such that $(f_i, \mathbf{X})$ satisfies *pIll*.

Lemma *ciIll_fiXpIll* :

∀ {*n k* : *nat*}

(*f* : *Vec* (*partialFunType R n*) (*S k*)) (*X* : *Box n*) (*cs* : *Vec Decoration* (*S k*)),

vecAll2 (fun *fi di* ⇒ *DecToPred di* _ *fi X*) *f cs* → *Vector.In Ill cs* →

∃ *f_i* : *partialFunType R n*, *Vector.In f_i f* ∧ *pIll n f_i X*.

C3a_pIll_ciIll shows that, given a vector of functions $f = (f_1, \ldots, f_{k+1})$ and a vector of decorations $cs = (c_1, \ldots, c_{k+1})$, if each $(f_i, \mathbf{X})$ has the strongest decoration c_i, for $i = 1, \ldots, k+1$, and a worst quality decoration of cs is ill, then there exists some f_i

in f, such that f_i has an empty domain.

```
Lemma C3a_pIll_ciIll :
 ∀ {n k : nat}
    {f : Vec (partialFunType R n) (S k)} {X : Box n}
    (cs : Vec Decoration (S k)),
    vecAll2 (fun fi di ⇒ DecToPred di _ fi X) f cs →
    minVecDecoration cs = Ill →
 (∃ f_i : partialFunType R n, Vector.In f_i f ∧ emptyDomain f_i).
```

C3a_pIll shows that, given a vector of functions $f = (f_1, \ldots, f_{k+1})$, if a worst quality decoration of $(c_0, c_1, \ldots, c_{k+1})$ is ill, each $(f_i, \mathbf{X})$ satisfies p_{c_i} and range$(f_i, \mathbf{X}) \subseteq \mathbf{u}_i$, for $i = 1, \ldots, k+1$, and $(\phi, \mathbf{U})$ satisfies p_{c_0}, then $(\phi \circ f, \mathbf{X})$ satisfies *pIll*.

```
Lemma C3a_pIll :
 ∀ {n k : nat}
    (f : Vec (partialFunType R n) (S k)) (X : Box n) (cs : Vec Decoration (S k))
    (phi : partialFunType R (S k)) (U : Box (S k)) (c0 : Decoration),
    vecAll2 (fun fi di ⇒ DecToPred di _ fi X) f cs →
    DecToPred c0 _ phi U →
    minDecoration c0 (minVecDecoration cs) = Ill →
    vecAll2 subset (mapBoxApp f (map iSem X)) (map iSem U) →
 pIll n (composition f phi) X.
```

A.2.5 Auxiliary Lemmas for Lemma C3_pCmin_NonEmptyBox

pCmin_O shows that, if f is an empty vector, which does not contain any function component, $\mathbf{X}$ is a nonempty box, and $(\phi, \mathbf{U})$ satisfies p_{c_0}, then $(\phi \circ f, \mathbf{X})$ also satisfies p_{c_0}.

```
Lemma pCmin_O :
  ∀ {n : nat}
    (f : Vec (partialFunType R n) 0) (X : Box n)
    (phi : partialFunType R 0) (U : Box 0) (c0 : Decoration),
    nonEmptySet (bSem X) →
    DecToPred c0 _ phi U →
  DecToPred c0 n
            (fun v : Vec R n ⇒ optionBind phi (vecOp2OpVec (mapApp f v))) X.
```

pCmin_Sk shows that, if

- each $(f_i, \mathbf{X})$, for $i = 1, \ldots, k+1$, satisfies p_{c_i},
- $\text{range}(f_i, \mathbf{X}) \subseteq \mathbf{u}_i$, for $i = 1, \ldots, k+1$,
- $(\phi, \mathbf{U})$ satisfies p_{c_0}, and
- $c_{\min}$ is a worst quality decoration of $c_0, c_1, \ldots, c_{k+1}$,

then $(\phi \circ f, \mathbf{X})$ satisfies $p_{c_{\min}}$.

```
Lemma pCmin_Sk :
  ∀ {n k : nat}
    (f : Vec (partialFunType R n) (S k)) (X : Box n) (cs : Vec Decoration (S k))
    (phi : partialFunType R (S k)) (U : Box (S k)) (c0 : Decoration),
    vecAll2 (fun fi di ⇒ DecToPred di _ fi X) f cs →
    DecToPred c0 _ phi U →
    vecAll2 subset (mapBoxApp f (map iSem X)) (map iSem U) →
    let c_min := minDecoration c0 (minVecDecoration cs) in
  DecToPred c_min n (composition f phi) X.
```

A.3 Auxiliary Lemmas for the FTDIA

isEmptyBox_cons shows that a box is empty iff its head interval is empty or there is an empty interval in its tail.

```
Lemma isEmptyBox_cons :
 ∀ {n : nat} {box : Box (S n)},
   isEmptyBox (cons Interval (hd box) n (tl box)) ↔
   isEmpty (hd box) ∨ isEmptyBox (tl box).
```

vectorInNonemptyBox shows that, given a vector $x = (x_1, \ldots, x_n)$ and a box $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$, then $x \in (\mathbf{x}_1 \times \ldots \times \mathbf{x}_n)$ implies $x_i \in \mathbf{x}_i$, for $i = 1, \ldots, n$.

```
Lemma vectorInNonemptyBox :
 ∀ {n : nat} {X : Box n} {x : Vec R n} (i : Fin.t n),
   setVecProd (map iSem X) x → iSem (nth X i) (nth x i).
```

nonEmptyBox_xInX shows that, if a box $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ is not empty, then there exists a vector $x = (x_1, \ldots, x_n)$, such that $x \in (\mathbf{x}_1 \times \ldots \times \mathbf{x}_n)$.

```
Lemma nonEmptyBox_xInX :
 ∀ {n : nat} {X : Box n}, ¬ isEmptyBox X →
 ∃ x : Vec R n, setVecProd (map iSem X) x.
```

pContinuous_projection shows that, given a box $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ and a projection function π_i, where $i \in \{1, \ldots, n\}$, then π_i is continuous on $\mathbf{X}$.

```
Lemma pContinuous_projection :
 ∀ {n : nat} {X : Box n} {i : Fin.t n},
   pContinuousOnBox (fun x : Vec R n ⇒ Some (nth x i)) X.
```

pSaf_projectionOnNonemptyBox shows that, given a nonempty box $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ and a projection function π_i, where $i \in \{1, \ldots, n\}$, if $\mathbf{x}_i$ is nonempty

and bounded, then $(\pi_i, \mathbf{X})$ satisfies the predicate *pSaf*.

```
Lemma pSaf_projectionOnNonemptyBox :
 ∀ {n : nat} {X : Box n} {i : Fin.t n},
  ¬ isEmptyBox X → isNonEmpty (nth X i) → isBounded (nth X i) →
 pSaf n (fun x : Vec R n ⇒ Some (nth x i)) X.
```

pDef_projectionOnNonemptyBox shows that, given a nonempty box $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ and a projection function π_i, where $i \in \{1, \ldots, n\}$, if $\mathbf{x}_i$ is nonempty and unbounded, then $(\pi_i, \mathbf{X})$ satisfies the predicate *pDef*.

```
Lemma pDef_projectionOnNonemptyBox :
 ∀ {n : nat} {X : Box n} {i : Fin.t n},
  ¬ isEmptyBox X → isNonEmpty (nth X i) → isUnbounded (nth X i) →
 pDef n (fun x : Vec R n ⇒ Some (nth x i)) X.
```

nonEmptyBox_nonEmptySet shows that, if a box $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ is not empty, then *bSem X* is a nonempty set.

```
Lemma nonEmptyBox_nonEmptySet :
 ∀ {n : nat} {X : Box n}, ¬ isEmptyBox X →
 nonEmptySet (bSem X).
```

pEmp_projectionEmptyBox shows that, given an empty box $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ and a projection function π_i, where $i \in \{1, \ldots, n\}$, then $(\pi_i, \mathbf{X})$ satisfies the predicate *pEmp*.

```
Lemma pEmp_projectorEmptyBox :
 ∀ {n : nat} {X : Box n} {t : Fin.t n}, isEmptyBox X →
  pEmp n (fun x : Vec R n ⇒ Some (nth x t)) X.
```

TermInd Term induction at arbitrary `Type`.

```
Fixpoint TermInd
  {Sigma : Signature} {n : nat} {P : Term Sigma n → Type}
  (Base: ∀ (i : Fin.t n), P (Var Sigma n i))
  (Step: ∀ (f : funSymb Sigma) (v : Vec (Term Sigma n) (arity f)),
            (∀ (i : Fin.t (arity f)) , P (nth v i)) →
          P (App Sigma n f v))
  (t : Term Sigma n) : P t :=
 match t as t0 return (P t0) with
 | Var i ⇒ Base i
 | App f v ⇒ Step f v (fun i : Fin.t (arity f) ⇒ TermInd Base Step (nth v i))
 end.
```

TermInd_vecAll Term induction at `Prop`, with simpler *Step*.

```
Lemma TermInd_vecAll :
 ∀ {Sigma : Signature} {n : nat} {P : Term Sigma n → Prop}
   (Base: ∀ (i : Fin.t n), P (Var Sigma n i))
   (Step: ∀ (f : funSymb Sigma) (v : Vec (Term Sigma n) (arity f)),
             (vecAll P v) →
          P (App Sigma n f v))
 (t : Term Sigma n) , P t.
```

pEval_App is a rewriting rule, which shows that *pEval Lib* (*App Sigma n f ts*) *x* is equivalent to *optionBind* (*fn* (*Lib f*)) (*vecOp2OpVec* (*map* (`fun` *t* ⇒ *pEval Lib t x*) *ts*)).

```
Lemma pEval_App :
 ∀ {Sigma : Signature} (Lib : DILibrary Sigma) {n : nat}
   (f : funSymb Sigma) (ts : Vec (Term Sigma n) (arity f)) (x : Vec R n),
```

pEval Lib (App Sigma n f ts) x =

optionBind (fn (Lib f)) (vecOp2OpVec (map (`fun` *t ⇒ pEval Lib t x) ts)).*

pEval_App_ext is a rewriting rule, which shows that *pEval Lib (App Sigma n f ts)* is equivalent to (`fun` *x ⇒ optionBind (fn (Lib f)) (vecOp2OpVec (map (*`fun` *t ⇒ pEval Lib t x) ts))).*

`Lemma` *pEval_App_ext :*

∀ {Sigma : Signature} (Lib : DILibrary Sigma) {n : nat}

(f : funSymb Sigma) (ts : Vec (Term Sigma n) (arity f)),

pEval Lib (App Sigma n f ts) =

(`fun` *x ⇒ optionBind (fn (Lib f)) (vecOp2OpVec (map (*`fun` *t ⇒ pEval Lib t x) ts))).*

pEval_App_ext' is a rewriting rule, which shows that *pEval Lib (App Sigma n f ts)* is equivalent to (`fun` *x ⇒ optionBind (fn (Lib f)) (vecOp2OpVec (mapApp (map (pEval Lib) ts) x))).*

`Lemma` *pEval_App_ext' :*

∀ {Sigma : Signature} (Lib : DILibrary Sigma) {n : nat}

(f : funSymb Sigma) (ts : Vec (Term Sigma n) (arity f)),

pEval Lib (App Sigma n f ts) =

(`fun` *x ⇒ optionBind (fn (Lib f)) (vecOp2OpVec (mapApp (map (pEval Lib) ts) x))).*

iEval_App is a rewriting rule, which shows that *iEval Lib (App Sigma n f ts) X* is equivalent to *ifn (Lib f) (map (*`fun` *t ⇒ iEval Lib t X) ts).*

`Lemma` *iEval_App :*

∀ {Sigma : Signature} (Lib : DILibrary Sigma) {n : nat}

(f : funSymb Sigma) (ts : Vec (Term Sigma n) (arity f)) (X : Vec DecIvl n),

iEval Lib (App Sigma n f ts) X = ifn (Lib f) (map (`fun`* t ⇒ iEval Lib t X) ts).*

FTIA_NonEmptyBox. For convenience of the proof of the *FTDIA_NonEmptyBox*, we also prove Lemma *FTIA_NonEmptyBox*, which shows that the FTIA is true when the input box is not empty. Since its proof is similar to the FTIA, we omit its proof here.

`Theorem` *FTIA_NonEmptyBox* :

∀ {*Sigma* : *Signature*} {*Lib* : *DILibrary Sigma*} {*n* : *nat*}
(*t* : *Term Sigma n*) {*X* : *Box n*} {*D* : *Vec Decoration n*}
(*iniDec* : *vecAll2 NEInitialDec X D*),
`let` *iEval'* := `fun` *t* ⇒ *iEval Lib t* (*vecAll2_map NEInitialDecIvl iniDec*) `in`
`let` *yd* := *iEval' t*
`in` *subset* (*range* (*pEval Lib t*) (*map iSem X*)) (*iSem* (*ivl yd*)).

pCmin_NonEmptyBox shows that, if a box $\mathbf{X}$ is not empty, then $\big((pEval\ Lib\ t), \mathbf{X}\big)$ satisfies the predicate p_d, where d is the second component of $yd = (\mathbf{y}, d)$, that is, a decoration.

`Lemma` *pCmin_NonEmptyBox* :

∀ {*Sigma* : *Signature*} {*Lib* : *DILibrary Sigma*} {*n* : *nat*}
(*t* : *Term Sigma n*) {*X* : *Box n*} {*D* : *Vec Decoration n*}
(*iniDec* : *vecAll2 NEInitialDec X D*)
(*nonEmptyBox* : ¬ *isEmptyBox X*),
`let` *iEval'* := `fun` *t* ⇒ *iEval Lib t* (*vecAll2_map NEInitialDecIvl iniDec*)
`in let` *yd* := *iEval' t*
`in` *DecToPred* (*dec yd*) *n* (*pEval Lib t*) *X*.

- *Sigma* is an instance of *Signature* (§11.2)

- *Lib* maps any function symbol to a point version function *fn* and its interval extension *ifn*, such that they satisfy the two properties of DILO
- *yd* denotes the final decorated interval result $(\mathbf{y}, d)$

Bibliography

[1] *IEEE Interval Standard Working Group - P1788.* `http://grouper.ieee.org/groups/1788/`.

[2] *The Coq Proof Assistant.* `http://coq.inria.fr/`.

[3] *The Flyspeck Project.* `https://code.google.com/p/flyspeck/`.

[4] H. Anton, I. Bivens, and S. Davis, *Calculus Single Variable*, Wiley, 2012.

[5] H. Barendregt and H. Geuvers, *Proof-assistants using dependent type systems*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001, pp. 1149–1238.

[6] Y. Bertot, *Coq in a Hurry*, CoRR, abs/cs/0603118 (2006).

[7] ——, *A Short Presentation of Coq*, in TPHOLs, 2008, pp. 12–16.

[8] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science, Springer Verlag, 2004.

[9] E. W. Cheney and D. R. Kincaid, *Numerical Mathematics and Computing*, Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 6th ed., 2007.

[10] A. J. Chlipala, *Certified Programming with Dependent Types*, 2011.

[11] M. Daumas, D. Lester, and C. Munoz, *Verified Real Number Calculations: A Library for Interval Arithmetic*, IEEE Trans. Comput., 58 (2009), pp. 226–237.

[12] M. Daumas and G. Melquiond, *Certification of bounds on expressions involving rounded operators*, ACM Trans. Math. Softw., 37 (2010), pp. 2:1–2:20.

[13] M. Daumas, G. Melquiond, and C. Munoz, *Guaranteed Proofs Using Interval Arithmetic*, in Proceedings of the 17th IEEE Symposium on Computer Arithmetic, ARITH-17, Cape Cod, Massachusetts, 2005.

[14] F. de Dinechin, C. Lauter, and G. Melquiond, *Certifying the Floating-Point Implementation of an Elementary Function Using Gappa*, IEEE Trans. Comput., 60 (2011), pp. 242–253.

[15] F. de Dinechin, C. Q. Lauter, and G. Melquiond, *Assisted verification of elementary functions using Gappa*, in Proceedings of the 2006 ACM symposium on Applied Computing, SAC '06, New York, NY, USA, 2006, ACM, pp. 1318–1322.

[16] W. W. Edmonson and G. Melquiond, *IEEE Interval Standard Working Group - P1788: Current Status*, in IEEE Symposium on Computer Arithmetic, 2009, pp. 231–234.

[17] H. Geuvers, *Proof assistants: History, ideas and future*, Sadhana, 34 (2009), pp. 3–25.

[18] T. C. Hales, J. Harrison, S. McLaughlin, T. Nipkow, S. Obua, and R. Zumkeller, *A revision of the proof of the Kepler conjecture*, ArXiv e-prints, (2009).

[19] B. Hayes, *COMPUTING SCIENCE A LUCID INTERVAL*, American Scientist, Vol.91, No. 6 (2003), pp. 484–488.

[20] J. Hölzl, *Proving Inequalities over Reals with Computation in Isabelle/HOL*, in Proceedings of the ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS'09), G. D. Reis and L. Théry, eds., Munich, August 2009, pp. 38–45.

[21] R. B. Kearfott, *Interval Computations: Introduction, Uses, and Resources*, Euromath Bulletin, 2 (1996), pp. 95–112.

[22] G. Melquiond, *Interval package for Coq.* `https://www.lri.fr/~melquion/soft/coq-interval/`.

[23] ——, *User's Guide for Gappa.* `http://gappa.gforge.inria.fr/`.

[24] R. Moore, *Interval analysis*, Prentice-Hall series in automatic computation, Prentice-Hall, 1966.

[25] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*, SIAM, 2009.

[26] N. S. Nedialkov, V. Kreinovich, and S. A. Starks, *Interval Arithmetic, Affine Arithmetic, Taylor Series Methods: Why, What Next?*, 2003.

[27] M. Neher, *From interval analysis to Taylor models — an Overview*, Proc. IMACS, (2005).

[28] L. C. Paulson, *Lessons Learned from LCF: A Survey of Natural Deduction Proofs*, Comput. J., 28 (1985), pp. 474–479.

[29] J. Pryce, *IEEE Std P1788 Draft 03.0*, 2011.

[30] ——, *P1788 Interval Standard Working Group: What you always wanted to know about decorated intervals.* `http://www.dagstuhl.de/mat/Files/11/11371/11371.PryceJohn.Slides.pdf`, 2011.

[31] J. D. Pryce, George, and F. Corliss, *Interval arithmetic with containment sets*, Computing, 78, pp. 251–276.

[32] N. Revol, *Validation for scientific computations interval arithmetic.* `http://perso.ens-lyon.fr/nathalie.revol/teaching/ValCS06-07/lecture-notes7.pdf`, 2007.

[33] ——, *IEEE 1788 Working Group for the Standardization of Interval Arithmetic: a brief overview*, in Dagstuhl Seminar 11371: Uncertainty modeling and analysis with intervals: Foundations, tools, applications, Dagstuhl, Germany, Sept. 2011.

[34] ——, *IEEE-1788 standardization of interval arithmetic: work in progress (a personal view)*, in IFIP Working Group 2.5 on Numerical Software, Santander, Espagne, July 2012.

[35] S. Thompson, *Type Theory and Functional Programming*, 1991.

[36] R. Zumkeller, *Formal global optimisation with Taylor models*, in Proceedings of the Third international joint conference on Automated Reasoning, IJCAR'06, Berlin, Heidelberg, 2006, Springer-Verlag, pp. 408–422.