NONDETERMINISM IN CONSTRUCTIVE Z

HASSAN HAGHIGHI AND SEYYED-HASSAN MIRIAN-HOSSEINABADI

Abstract. The abstraction inherent in most specifications and the need to specify nondeterministic programs are two well-known sources of nondeterminism in formal specifications. In this paper, we present a formalism, including the notion of multi-schema and a new set of schema calculus operations, by which one can specify bounded, unbounded, loose, strict, erratic, angelic, demonic, singular, and plural nondeterminism in the CZ formal specification language. CZ is a Z-style notation that is based upon a constructive set theory, namely, CZ set theory. While our definitions can be modified slightly to be used in the Z notation, we have chosen CZ, instead of Z, because of its constructive basis that allows us to investigate the notion of nondeterminism from the formal program development point of view. More precisely, we formally construct functional programs from CZ specifications and then probe the effects of the initially specified nondeterminism on final programs. Our investigation will show that without specifying nondeterminism explicitly, the effects of the nondeterminism involved in initial specifications will not be preserved in final programs. On the other hand, we will interpret all the nondeterministic constructs of the formalism, proposed by this paper, in CZ and then prove that using this formalism for writing nondeterministic specifications leads to programs that preserve the initially specified modalities of nondeterminism. We will show the practicability of the new formalism via several examples.

1. Introduction

Traditional methods for developing software systems basically rely on program testing to demonstrate that the program works successfully and meets the specification. Such an approach has long been criticized: testing must be complete in covering every possible operational situation. For real systems, this is either expensive or impossible to achieve [15]. Thus, it is not a good vehicle for the development of reliable software systems [3].

As a solution to increase the level of reliability of software systems, formal methods have been proposed which are based on mathematics [58, 75]. Usually these methods are introduced to the software life-cycle by adding the formal specification stage to the stages of software projects. At this stage, we describe what has to be done instead of how it has to be done. Formal specifications should be written in a soundly based specification language [75]. Z [63, 75] is an excellent specification notation that is based upon a well-known set theory, namely, Z set theory, and the first order predicate logic. Together, they make up a mathematical language that is easy to learn and to apply [75]. Also, the set theoretical specification in Z of a problem is as abstract as possible; hence, it makes no commitment to operational details. Another advantage of Z pertains to the schema calculus. This provides a systematic tool for structuring and organizing specifications of large systems.

Key words and phrases. formal specification, formal program development, constructive Z, Martin-Löf’s theory of types, nondeterminism, modalities of nondeterminism.
Having specified a problem formally, the question is how a program can be developed so as to guarantee that it meets the specified requirements. There are a number of approaches in the literature for developing programs from Z specifications. One approach is to refine an abstract specification to code using the rules suggested by the refinement calculus: via a series of steps the specification is gradually transformed into a program, usually an imperative one [50]. While involving a well-developed program development calculus, refinement does not smoothly integrate with Z since the concepts in the two regimes are rather different. The main difficulty is knowing which rules to apply; it does highlight the large gulf that exists between the pure set theoretical specification and the imperative target code.

The second branch of activities for developing programs from Z specifications consists of those activities that employ a functional target language and attempt to translate Z specifications directly to code. A survey of such activities can be found in [47]. Unfortunately, none of these activities provide a complete algorithmic translation between Z specifications and programs. In addition to the two mentioned methods, there are approaches, based upon constructive mathematics, that try to derive programs from correctness proofs of formal specifications. In this area, Martin-Löf’s theory of types [42] is a well-known formalism for program development; the proof rules of this theory can be used to derive a correct program from the correctness proof of a specification as well as to verify that a given program has a certain property [55]. Furthermore, this theory allows us to express both specifications and programs within the same formalism. However, its infrastructure for the organization and manipulation of specifications is rather meager.

To employ both the facilities of Z as a specification medium and the abilities of constructive theories in program development, in [47] and [48], the CZ (Constructive Z) formal specification language has been introduced. CZ is a Z-style notation which is interpreted by a constructive set theory, called CZ set theory. Indeed, the difference between CZ and Z notations lies in the interpretation and not the appearance; the intention for developing CZ is to retain the whole of the Z notation including the schema calculus but provide it with a different interpretation, namely, one in a constructive theory. In [47] and [48], the CZ set theory has been interpreted in Martin-Löf’s theory of types.

1.1. Nondeterminism in formal specifications. It is crucial that a specification specifies exactly the desired result, not less and not more [20]. Obviously, if it specifies less, then we might end up with a program that satisfies the specification, but is still not what is wanted. If the specification over-specifies, then the programmer is deprived of some routes to take in developing the program. In this way, some appropriate implementations might be excluded, or in the worst case, all implementations might be excluded. Using abstraction in specifications helps not to over-specify. However, abstraction causes formal specifications to determine several behaviors of final programs, and thus nondeterminism remains an inherent part of them [23, 28]. The next example shows how abstraction can lead to nondeterministic specifications.

Example 1. Consider the sorting problem. For convenience, we confine ourselves to sorting elements whose keys are natural numbers. An informal specification of this problem is:
Sorting problem: for any sequence of elements whose keys are natural numbers, there exists a permutation in which the keys are in the increasing order [20].

This abstract specification is nondeterministic since for a sequence having different elements with equal keys, it determines more than one permutation in which the keys are in the increasing order 1.

In addition to abstraction, another source of nondeterminism in formal specifications is the need to specify nondeterministic programs. A program is nondeterministic if for at least one input, it produces more than one output. For example, consider a program controlling a drinks machine that serves tea and coffee. When asked whether they would like tea or coffee, indecisive people reply Either. For them, the program may have a nondeterministic behavior. Concurrent and parallel systems are familiar nondeterministic programs: in a multiprogramming environment, at several points of the execution, such as process (or thread) creation, synchronization and communication, two or more entities (programs, processes, threads, or expressions) may want to use a common resource (e.g., a lock, an address space, a communication network, or a shared variable) at the same time [9, 56, 61]. To resolve this competition, a choice should be made whose result is not necessarily deterministic; hence, various approaches to the theory and specification of concurrent systems, such as CSP [34], ACP [5] and CCS [46], include the phenomenon of nondeterminism.

The next example 2 shows an instance of nondeterminism involved in concurrent systems. The example indicates that when we use a lock-based method to handle concurrency in a database management system, at several points of the execution, two or more transactions may compete for acquiring a lock on a common data item. This competition can be resolved by a nondeterministic choice among the competing transactions. In subsection 2.2, we will refer to the next example by writing a CZ specification of the concurrency control in a database management system. In subsection 3.5, we will change this specification to bring nondeterminism explicitly into its structure.

Example 2. In a database management system, some operations are executed on data items among which the most primitive ones are the following atomic operations:

- \( a = \text{read}(x) \): Reads the value of the data item \( x \) and stores it in \( a \).
- \( \text{write}(a, x) \): Writes the value of \( a \) to \( x \).

A transaction is a sequence of the above operations that satisfies the ACID (Atomicity, Consistency, Isolation and Durability) properties [17]. The goal of a concurrent database management system is to increase efficiency by allowing several transactions to execute concurrently. Of course, the effect should be the same as if the transactions were executed in some serial order. In this way, data items involved in operations are left in a consistent state. A usual way to achieve this goal is to

---

1 We will refer to the sorting problem several times throughout the paper. In subsections 2.2 and 2.3, we will write set theoretical and type theoretical specifications of this problem in CZ and Martin-Löf’s theory of types, respectively. In subsection 2.4, we will produce a functional program from the initial CZ specification of the problem. Again, in section 3, we will change the previous CZ specification of this problem to explicitly bring nondeterminism into its structure. Then, we will extract a new program from the explicitly nondeterministic specification.

2 This example comes from [53].
consider a scheduler that schedules conflicting operations. Two operations conflict if and only if they are involved in different transactions, at least one of them is the write operation, and both operate on the same data item. The scheduling can take place through locking mechanisms. Locking is the oldest, and still most widely used, form of concurrency control: when a transaction needs access to a data item, it tries to acquire a lock on it. When it no longer needs the item, it releases the lock. The job of the scheduler is to grant and release locks in a way that guarantees valid schedules. Sometimes the scheduler encounters more than one transaction whose requests for acquiring a lock are valid. The scheduler must resolve such a competition by making a nondeterministic choice.

We have informally described the sorting problem in example 1. In example 3, we will give a formal specification of this problem in CZ. Although this specification corresponds to more than one implementation, the specification structure does not imply such a nondeterministic behavior. In other words, nondeterminism does not come into the specification explicitly. In this paper, we use the terminology implicitly nondeterministic specification to denote such specifications. As we will show in section 3, at the end of deriving a program from a correctness proof of an implicitly nondeterministic specification, only one of the possible behaviors is extracted from the proof tree and appears in the final program. In other words, the effects of the initially specified nondeterminism will be lost in the final program. Such a result seems not to be wrong in a situation where the nondeterminism is due to the abstraction involved in the specification while we still want the final implementation to be deterministic (e.g., in the sorting problem); however, in such situations, we encounter another problem: the programmer is deprived of some routes to take in developing the program or some good implementations might be excluded.

The above problem seems to be more serious when we consider specifications of nondeterministic programs. If the nondeterminism involved in such programs is not specified explicitly, it will be lost during the development of programs from their formal specifications, and thus will not exist in final programs. Let us investigate this problem in the specification and development of concurrent systems. Concurrency is usually modelled by the nondeterministic interleaving of atomic operations [23]. For instance, Evans in a series of his papers (for example, [21], [22], and [23]) has tried to describe the concurrent behavior of a system in terms of its allowable computations that result from the interleaved execution of concurrent processes. However, an Evans’s specification of a concurrent system is implicitly nondeterministic and results in a program provided with only one of the possible interleaved executions of concurrent processes. In example 4, we will give a formal specification of a lock-based method for controlling concurrency in a database management system. In subsection 3.1, we will show that while this specification implies the competition between concurrent transactions, such a competition will not be preserved in the resulting program.

To solve this problem, it seems that we must directly specify all possible behaviors which are obtained from nondeterminism. As we will show in section 3, although this method solves the problem, it yields longer and more complicated specifications which are harder to read and write. Also, using this idea, the writer of a nondeterministic specification him/herself must take care that his/her
method to specify all possible behaviors is suitable at the same time that s/he describes the functional properties of the system. Both of these issues become more serious when we need to specify modalities of nondeterminism such as strict, angelic, demonic, singular, and plural that each of them is useful for some applications in the real world (We will give some examples in section 3. For further examples, see [2, 33, 56, 71]).

To address the above mentioned issues, a number of attempts have been undertaken to introduce nondeterministic constructs into different formal specification languages. A survey of these attempts can be found in [25]. Using these constructs, we can specify modalities of nondeterminism at the same time that we apply our usual procedures to specify functional properties of the system. In this way, nondeterminism comes into our specifications without decreasing our focus on describing the functional properties of the system. On the other hand, since these constructs do not change the main structure of a specification, they do not lead to longer and more complicated specifications.

To produce nondeterministic programs from specifications which involve nondeterministic constructs, one of the following approaches can be adopted:

1. Extending current formal program development frameworks in somehow to transform nondeterministic constructs of specifications into final programs.
2. Interpreting the nondeterministic constructs in the specification language itself. Having such an interpretation, there is no need to change the current frameworks for formal program development.

Now we are ready to say what this paper is intended to do.

1.2. The contribution of the paper. The aim of this paper is to present a CZ-based formalism by which one can bring bounded, unbounded, loose, strict, erratic, angelic, demonic, singular, and plural nondeterminism into formal specifications. The new formalism includes a new notion of schemas, called multi-schema, and a set of new operations for the schema calculus. We will interpret all the new constructs in CZ itself; hence one can use the current translation of CZ into Martin-Löf’s theory of types [47, 48] to extract programs from nondeterministic specifications written in the new formalism.

While our definitions can be modified slightly to be used in the Z notation, we have chosen CZ, instead of Z, because of its constructive basis, namely, CZ set theory, that allows us to probe the effects of the initially specified nondeterminism on constructed programs. In this way, we will prove that specifications written in the new formalism lead to functional programs that are provided with all possible behaviors according to the initially specified modalities of nondeterminism. More precisely, by the new formalism, one can write his/her own nondeterministic specifications and then interpret these specifications in CZ. Using the current translation of the CZ set theory into Martin-Löf’s theory of types, the resulting CZ specifications can be transformed into their type theoretical counterparts. Finally, functional programs, which satisfy the initial nondeterministic specifications, can be extracted from correctness proofs of the type theoretical specifications.

Our work is close to that of [53] in which some nondeterministic constructs have been added to the CZ specification language to model strict, unbounded, erratic, angelic, and demonic nondeterminism. In [53], although all proposed nondeterministic constructs have been interpreted in CZ, the constructive basis of CZ has not
been regarded, and thus the notion of nondeterminism has not been investigated from the formal program development point of view; In order to overcome this drawback, in [26], we have interpreted the nondeterministic constructs of CZ [53] in Martin-Löf’s theory of types and have thus introduced a way to construct programs from CZ nondeterministic specifications. In this paper, we extend/modify our previous work in three ways:

1. In addition to the strict, unbounded, erratic, angelic, and demonic modalities of nondeterminism, which have been considered in the previous work, we will model bounded, loose, singular, and plural nondeterminism in CZ. Also, unlike [53] and [26], which only regard the non-abortion of programs to distinguish angelic and demonic nondeterminism, we will introduce a general approach for modelling the angelic and demonic choices. This approach also covers game-like situations in which the combination of both modalities of nondeterminism are used to simulate conflicting agents. To model singular and plural nondeterminism in CZ, we can adopt two alternative ideas (see subsection 3.3.3); however, only one of these ideas can be supported in the specification phase. Therefore, We will interpret singular and plural nondeterminism in terms of this idea. The issues related to the other one will be stated, but the complete investigation will be delegated to future work.

2. Since the operations of the schema calculus will no longer work in the presence of the new nondeterministic constructs, we will give a new set of schema calculus operations which properly operate on schemas having proposed nondeterministic constructs.

3. Our method in [26] is based on adding nondeterministic constructs to type theory, and thus extending the current interpretation of CZ in type theory [47, 48] to translate nondeterministic constructs of CZ into their counterparts in type theory. In this paper, however, we interpret all the new constructs in CZ itself. Of course, we use the current translation of CZ into Martin-Löf’s theory of types to investigate the notion of nondeterminism from the formal program development point of view. Our investigation will show that without specifying nondeterminism explicitly, the effects of the nondeterminism involved in initial specifications will not be preserved in final programs. The proposed formalism in this paper helps us to develop programs which are provided with all possible behaviors in terms of the initially specified modalities of nondeterminism. We will show the applicability of the new formalism by constructing programs from nondeterministic specifications of simple problems, such as the sorting problem, and probing the effects of the specified nondeterminism on final programs. Also, as a case study of concurrent systems, in subsection 3.5, we will apply the new formalism to a familiar concurrent problem, the strict 2PL algorithm. In this way, we will demonstrate that our formalism can be considered as a starting point for specifying concurrent systems and developing them such that the developed programs preserve the initially specified nondeterminism.

The paper is organized in the following way. In section 2, we give a brief overview of the CZ set theory, the CZ specification language, Martin-Löf’s theory of types, and the interpretation of CZ in type theory. In section 3, we explore the notion of
nondeterminism in CZ and introduce a number of nondeterministic constructs and a new operations of the schema calculus to model modalities of nondeterminism in set theoretical specifications of CZ. The last section is devoted to the conclusion and directions for future work.

The contribution of the paper begins from section 3 when we present a CZ-like (or Z-like) formalism by which one can bring modalities of nondeterminism into formal specifications. However, since we want to investigate the notion of nondeterminism from the formal program development point of view, we should introduce the formal framework which is intended to develop programs from CZ specifications. This includes two main theories, namely, the CZ set theory and Martin-Löf’s theory of types, and the translation of the former into the latter. To make the paper self contained, we dedicate numerous pages for reviewing these materials in detail; it substantially leads to a long paper. Nevertheless, readers who are familiar with the mentioned theories or only interested in following the main contribution of the paper, can skip the corresponding subsections.

2. Preliminaries

In this section, we review the constructive framework for formal program development which will be used in section 3 to investigate the notion of nondeterminism from the program development point of view. This includes a relatively detailed description of the CZ set theory, the CZ formal specification language, and Martin-Löf’s theory of types in subsections 2.1, 2.2, and 2.3, respectively. The interpretation of CZ in Martin-Löf’s theory of types is summarized in subsection 2.4.

2.1. CZ set theory. As in classical, axiomatic set theories, in Constructive set theories, a few simple axioms about the primitive notions of set and membership are formulated, and then basic set theoretical principles are obtained from these axioms. However, Constructive set theories have been defined to discover formalisms which isolate the essential principles of constructive mathematics in the same way that Zermelo-Fraenkel (ZF) set theory isolates the principles underlying classical mathematics. Constructive set theories originated with Myhill’s paper [54] that attempted to formalize the content of Bishop’s constructive mathematics [6, 7]. Aczel in [1] has given the description of the system CZF (Constructive Zermelo-Fraenkel) of the constructive set theory that is an extension of Myhill’s system and a subsystem of the classical Zermelo-Fraenkel set theory.

In [47] and [48], another constructive set theory, called CZ set theory, has been introduced that is a weaker version of CZF and is intended to give constructive interpretations for Z-style specifications. The language of the CZ set theory is the same as that of CZF. We adopt the following BNF syntax for the language of the CZ set theory:

well-formed formulas: \( \phi ::= \alpha \mid \phi \lor \phi \mid \phi \land \phi \mid \phi \Rightarrow \phi \mid \forall x \cdot \phi \mid \exists x \cdot \phi \)

atomic formulas: \( \alpha ::= t \in t \mid t = t \mid \Omega \)

terms: \( t ::= x \)

where \( \Omega \) is absurdity, and \( x \) is a variable name. Negation is defined in terms of absurdity and implication in the usual manner:

\( \neg \phi \equiv \phi \Rightarrow \Omega \)
We employ \( \iff \) for the bi-implication and \( y \subseteq x \) for \( y \) is a subset of \( x \), both defined in the standard way. *Bounded quantification* is introduced by definition as follows:

\[
\forall x \in t \cdot \phi \iff \forall x \cdot x \in t \Rightarrow \phi \\
\exists x \in t \cdot \phi \iff \exists x \cdot x \in t \land \phi
\]

All proof rules of the classical set theory ZF can be used in CZ except the rule of classical negation which is given in the following:

\[
\neg \phi \vdash \Omega \\
\phi
\]

The reason for discarding the rule of classical negation in CZ is as follows. In the classical mathematics, a proposition is thought of as being *true* or *false* independently of whether we can prove or disprove it. On the other hand, a proposition is constructively *true* only if we have a method of proving it, and *false* if the assumption on its correctness results in contradiction. For example, classically the law of excluded middle, namely, \( A \lor \neg A \), is *true* since the proposition \( A \) is either *true* or *false*. Constructively, however, a disjunction is *true* only if we can prove one of the disjuncts. Since we have no method of proving or disproving an arbitrary proposition \( A \), we have no proof of \( A \lor \neg A \), and thus the law of excluded middle is not constructively valid [55]. For this reason, in constructive set theories, the axiom of excluded middle and other axioms and rules which can be derived from this axiom, such as the rule of classical negation, must be discarded.

All the axioms of CZ shadow those of the classical theory; indeed, most axioms remain intact. However, three axioms including separation, foundation and power set have to be modified to satisfy constructive scruples. In contrast to the axioms of CZF, the axioms of CZ do not include the *Collection* axiom. Also, modifying the power set axiom yields a new axiom concerning the cartesian product set constructor. In the following, we describe the set theoretical axioms of CZ:

**Extensionality:** \( \forall x \cdot \forall y \cdot (x = y \iff (\forall z \cdot z \in x \iff z \in y)) \)

This axiom states that two sets are equal if they have exactly same members.

**Empty set:** \( \exists x \cdot \forall y \cdot \neg(y \in x) \)

This axiom guarantees the existence of the empty set. By the Extensionality axiom, this set is unique. \( \emptyset \) is used as the symbol of this unique set.

**Bounded separation:** \( \forall x \cdot \exists z \cdot \forall y \cdot y \in z \iff (y \in x \land \phi[y]) \) (\( \phi \) is bounded \(^4\))

Like the axiom of Separation in ZF, the Bounded separation axiom enables us to define any subset of a set which satisfies the formula \( \phi \). Unlike the Separation axiom, the requirement in the Bounded separation axiom, that \( \phi \) is bounded, shows that the condition part in the definition of a certain set must only refer to sets which were or might have been defined previously. In this way, CZ follows the strictures of constructivism and only permits bounded formulas to determine subsets.

\(^3\) In CZF, there exists the following axiom, called *Collection*:

\[
(\forall x \in u \cdot \exists y \cdot \phi[x, y]) \Rightarrow \exists v \cdot \forall x \in u \cdot \exists y \in v \cdot \phi[x, y]
\]

In this axiom, if \( \phi[x, y] \) is read as \( x \) nominates \( y \), then the axiom guarantees the existence of the set of all nominees when each member of a set \( u \) nominates at least one object. This axiom is equivalent to the axiom of Replacement of ZF. The reason for ignoring the Collection axiom in CZ is the same as the reason for using Z set theory (ZF set theory without the axiom of Replacement) instead of ZF as the basis for the Z notation. For a detailed description, see [47].

\(^4\) A formula \( \phi \) is bounded if it is constructed from atomic formulas using only conjunction, disjunction, implication and bounded quantification.

---

8 Hassan Haghighi and Seyyed-Hassan Mirian-Hosseinabadi
Pairing: $\forall x \cdot \forall y \cdot \exists z \cdot x \in z \land y \in z$

This axiom expresses that for every two sets, there is a set that they belong to. Let $x$ and $y$ be two sets and let $z$ be the set guaranteed by Pairing. Applying the Bounded separation axiom to $z$ can yield a set whose elements are precisely $x$ and $y$ ($\{u \in z \mid u = x \vee u = y\}$), which is unique by Extensionality. We use $\{x, y\}$ for the set whose members are exactly $x$ and $y$ and use $\{x\}$ for the singleton set whose only member is $x$. We also employ the following standard definition of ordered pairs:

$$(x, y) \equiv \{\{x\}, \{x, y\}\}$$

Union: $\forall x \cdot \exists z \cdot \forall y \in x \cdot \forall u \in y \cdot u \in z$

This axiom defines how to build a set from the members of other sets. By the Bounded separation and Extensionality axioms, there is a unique set whose members are precisely of members of $x$; we employ $\bigcup x$ for this guaranteed set which we call union. In particular, we write $a \cup b$ for $\bigcup \{a, b\}$.

Set induction: $$(\forall y \cdot (\forall x \in y \cdot \phi(x)) \Rightarrow \phi[y]) \Rightarrow \forall x \cdot \phi[x]$$

The Foundation axiom in ZF is the only axiom of this theory which implies the law of excluded middle for bounded formulas (For a detailed description, see [48]). This axiom has therefore been replaced with the axiom of Set induction.

Infinity: $\exists x \cdot \forall y \cdot y \in x \Leftrightarrow y = \emptyset \lor \exists u \in x \cdot y = \{u\}$

This axiom guarantees the existence of an infinite set. By the Set induction axiom, this guaranteed set is unique. We write $N$ for this unique set, i.e. the set of natural numbers. We employ $0$ for $\emptyset$ and $s(n)$ for $\{n\}$.

Decidable power set: $\forall x \cdot \exists z \cdot \forall y \in z \Leftrightarrow y \subseteq x$

In the above axiom, the relation $y \subseteq x$ indicates that $y$ is a decidable subset of $x$. This relation is defined as follows:

Definition 2.1.1. $y$ is a decidable subset of $x$ iff $y \subseteq x$ and $\forall u \in x \cdot u \in y \lor \neg(u \in y)$

In the classical set theory ZF, the Power set axiom is as follows:

Power set: $\forall x \cdot \exists z \cdot \forall y \cdot y \in z \Leftrightarrow y \subseteq x$

In the above axiom, the power set is not restricted: any kind of subset is permitted, not just the decidable ones. It is the most important difference between CZ and the classical set theory ZF. CZ only permits subsets which can be constructed in the sense that we can determine their membership relative to their superset. Intuitively, the decidable subsets can be identified with decision procedures which test for membership. Indeed, this is precisely how they are introduced in [7]. such a constraint is more in keeping with constructive scruples which reject the Power set axiom in its traditional form because of its impredicativity.

Our main way of forming subsets is via bounded separation: which of these are decidable? The following notion gives a partial answer to this question; it will be useful in determining which subsets of a set are decidable.

Definition 2.1.2. A well-formed formula $\phi$ is decidable relative to a set $a$ iff $\forall x \in a \cdot \phi \lor \neg\phi$.

Decidable, well-formed formulas (relative to a given set) are closed under the propositional connectives: if $\phi$ and $\psi$ are decidable relative to $a$ then
so are their negations, conjunction, disjunction, and implication. Now if \( b \) is a decidable subset of \( a \), and \( \phi \) (bounded) is decidable relative to \( a \), then \( \{ x \in b \mid \phi(x) \} \) is also a decidable subset of \( a \). This follows directly from the definitions 2.1.1 and 2.1.2. The following notion of sets with decidable equality will also be useful in the paper.

**Definition 2.1.3.** A set \( a \) has decidable equality iff \( \forall x \in a \cdot \forall y \in a \cdot x = y \lor \neg (x = y) \).

*Cartesian product:* \( \forall x \cdot \forall y \cdot \exists z \cdot \forall u \in x \cdot \forall v \in y \cdot (u, v) \in z \)

The usual way of defining the *cartesian product* of two sets, in the presence of the power set, is as follows:

\[
x \times y \cong \{ z \in P(P(x \cup y)) \mid \exists u \in x \cdot \exists v \in y \cdot z = (u, v) \}
\]

Since the decidable version of the power set is used, it is not sensible to use this definition, and it is necessary to add the *Cartesian product* of two sets as an axiom. By the Bounded separation axiom, we have:

\[
\forall x \cdot \forall y \cdot \exists z \cdot \forall w \cdot (w \in z \iff \exists u \in x \cdot \exists v \in y \cdot w = (u, v))
\]

For every two sets \( x \) and \( y \), the uniqueness of \( z \) (see the above proposition) is guaranteed by the Extensionality axiom. \( z \) is called the cartesian product of \( x \) and \( y \) and is written as \( x \times y \).

In contrast to the Z set theory, which is a classical theory, the CZ set theory can be considered as a *constructive* interpretation of the Z language. Specially, replacing instances of the power set by decidable ones provides a way for determining whether specifications specify decidable problems or not. In the next subsection, we show that CZ set theory is enough for the purposes of program specification in the style of Z. To achieve this goal, we indicate how some basic infrastructure of Z is represented in CZ. Therefore, the difference between the CZ and the Z notations lies in the interpretation and not the appearance. The CZ set theory has an interpretation in a constructive type theory, namely, Martin-Löf’s theory of types (see subsection 2.4), and does not permit the specification of non-computable functions. In other words, while non-computable functions are definable in Z, they will no longer be definable in the CZ specification language.

### 2.2. CZ formal specification language

In this subsection, we introduce the CZ formal specification language as a Z-style formalism for writing specifications based on the CZ set theory. We show that the common set theoretical constructions employed in Z are available in CZ. Specifications in CZ find their meanings as operations upon sets. Hence, as in the Z notation, a set is a primitive notion in CZ. The existence of sets, in particular the empty set (\( \emptyset \)) and the set of natural numbers (\( N \)), are guaranteed by the set theoretical axioms of the CZ set theory stated in the previous subsection. The definitions of operations on sets, namely, *Union* (\( \cup \)), *Generalized Union* (\( \bigcup \)), *Intersection* (\( \cap \)), *Generalized Intersection* (\( \bigcap \)) and *Difference* (\( \setminus \)), and the definition of *Decidable Subset* (\( \subseteq \)) are derived from the axioms of the CZ set theory (For a detailed description, see [47]). The function \( # \) is defined to count the number of members of finite sets.

The CZ specification language is a strongly typed language; it means that using the type inference rules of CZ, any CZ expression must be assigned one and only one type (To see the list of type inference rules of CZ, refer to [47]). Types are
maximal sets to which variables and expressions belong. The type of a variable is
declared in the same way as the membership relation between the variable and its
type.

In the next subsection, we introduce some types and type constructors of CZ.

2.2.1. The type system of CZ. In the CZ language, the definitions of types (as
maximal sets) are obtained from the axioms of the constructive set theory CZ.
The type of natural numbers (N) is a basic type in this language, derived from
the Infinity axiom. Given set is the other basic type in CZ. Additional types can
be created using the decidable power set (P) and the cartesian product (A × B)
constructors. In the following, we describe the primary types and type constructors
of CZ:

• Given type: A given set can be used as a given type. This type is a set
whose internal structure is invisible. We may introduce elements of such a
set, and associate properties with them, but we can assume nothing about
the set itself. Names of given types are identified within square brackets at
the first stage of writing a specification without any further assumptions
about their internal structures. In example 4, at the beginning of the
specification of a concurrent database management system, we will define
data types, respectively:

- [DatumItem, Value]
- DataItem
- Value

It is assumed that all given types have no elements in common and have
decidable equality (see definition 2.1.3).

• Cartesian product: The cartesian product of two types A and B is the type
A × B consisting of all ordered pairs whose first elements are in A and
whose second elements are in B.

• Relation: As in Z, we can define a relation R between two types A and B
as a set of ordered pairs (x, y) where x ∈ A and y ∈ B. The relation R is
shown as R ∈ A ⇔ B where A ⇔ B ≡ P(A × B): unlike Z, the class of
relations is restricted to decidable subsets of A × B. Using the axiom of
Bounded separation (see subsection 2.1), various sets and operations
can be defined on relations. We give the definitions of those sets and operations
that we will use in this paper:

1. domain : If R ∈ A ⇔ B, then
dom R ≜ \{x ∈ A | ∃y ∈ B \cdot (x, y) ∈ R\}
2. range : If R ∈ A ⇔ B, then
ran R ≜ \{y ∈ B | ∃x ∈ A \cdot (x, y) ∈ R\}
3. domain restriction : If R ∈ A ⇔ B and u ⊆ A, then
u ↾ R ≜ \{(x, y) ∈ A × B | x ∈ u \land (x, y) ∈ R\}
4. range restriction : If R ∈ A ⇔ B and u ⊆ B, then
u ⇿ R ≜ \{(x, y) ∈ A × B | y ∈ u \land (x, y) ∈ R\}
5. domain anti-restriction : If R ∈ A ⇔ B and u ⊆ A, then
u ⊢ R ≜ \{(x, y) ∈ A × B | x \notin u \land (x, y) ∈ R\}
6. range anti-restriction : If R ∈ A ⇔ B and u ⊆ B, then
u ⊢ R ≜ \{(x, y) ∈ A × B | y \notin u \land (x, y) ∈ R\}

• Function: A relation f ∈ A ⇔ B is a partial function if for every x in A,
there is at most one y in B such that (x, y) ∈ f. If, in addition, we know
that for every x in A, there is a unique y in B such that (x, y) ∈ f, then f is
called a total function. Other types of functions, namely, partial injection, total injection, partial surjection, total surjection, bijection, finite partial function, and finite partial injection are defined similar to the definitions of their counterparts in Z [63, 75]. However, unlike Z, CZ does not support all classical functions: the restriction of the class of subsets to decidable ones also affects the class of functions. Since functions are special kind of relations, all operations defined over relations can also be used on functions. Moreover, Functional Overriding [63, 75] with the operator ⊕ is defined as follows:

**Functional Overriding:** functional overriding: If \( f \) and \( g \) are two functions from \( A \) to \( B \), then
\[
f \oplus g \equiv ((\text{dom } g) \subseteq f) \cup g
\]

- **Sequence:** Sequences are used to model ordered collections of objects. In a sequence, some elements may appear more than once. For any set \( X \), \( \text{seq } X \) denotes the type of sequences including elements of \( X \). Formally, \( \text{seq } X \) is the set of all partial functions \( f \) from \( N \) to \( X \) such that \( \exists n \in N \cdot \text{dom } f = N_i \), where \( N_i \equiv \ldots, i \). Sequences are written enclosed in angle brackets, i.e. \( \langle \) and \( \rangle \), with their elements separated by commas. The empty sequence is written as \( \langle \rangle \); a non-empty sequence \( l \) is written as \( h :: t \) where \( h \) and \( t \) are the head and the tail of \( l \), respectively. The \( i \)th element of the sequence \( S \) is written as \( S.i \). Since a sequence is a function or, in general, a set, we can use all set or function operations to manipulate sequences. There are four more functions, namely, head, tail, front and last, which are defined as usual. The concatenation operator, \( \land \), is used to join two sequences together by placing one in front of the other.

- **Free type:** As in Z, we use free types as sets with explicit structuring information to model enumerated collections, compound objects, and recursively defined structures. To begin with free types, consider the special case in which the set to be introduced has a small number of elements. In example 4, we will introduce the new commands commit and abort to the database management system in addition to the commands read and write, defined in example 2. We will define a free type Command as the set of commands of the database management system by introducing its four distinct elements:

\[
\text{Command} ::= \text{Read} | \text{Write} | \text{Commit} | \text{Abort}
\]

Once this definition has been made, we may infer that Command is the smallest set containing the four distinct elements Read, Write, Commit, and Abort. The order in which these elements are introduced is unimportant [75].

We may include copies of other sets as part of a free type, using constructor functions. The notation

\[
\text{FreeType} ::= \text{constructor}(\langle \text{source} \rangle)
\]

introduces a collection of constants, one for each element of the set source. constructor is an injective function whose target is the set FreeType. Constants and constructor functions may be used together in the same definition, as in the following free type:

\[
\text{FreeType} ::= \text{constant} | \text{constructor}(\langle \text{source} \rangle)
\]

What is more, the source type of a constructor function may refer to the free type being defined. The result is a recursive type definition: FreeType
is defined in terms of itself. For example, in the following, we define a recursive data structure resembling the natural numbers:

\[ \text{nat} := \text{zero} \mid \text{succ}(\langle \text{nat} \rangle) \]

Every element of \( \text{nat} \) is either \text{zero} or the successor of a natural number. \text{zero} is not a successor, and every element of \( \text{nat} \) has a unique successor. The set \( \text{nat} \) is the smallest set containing the following collection of distinct elements: \text{zero}, \text{succ(\text{zero})}, \text{succ(succ(\text{zero}))}, \text{and so on.} As another example, we may define a free type of binary trees, in which every element is either a leaf or a branching point:

\[ \text{BinaryTree} := \text{leaf}(\langle N \rangle) \mid \text{branch}(\langle \text{BinaryTree} \times \text{BinaryTree} \rangle) \]

Each leaf contains a number; each branching point joins a pair of sub-trees.

In the next subsection, we show how the language of CZ can be used to specify programs.

2.2.2. Specification construction units. In CZ, the specification of an operation can be written as a proposition in the following general form:

\[ \forall x_1 \in A_1, x_2 \in A_2, \ldots, x_m \in A_m \cdot (\phi \Rightarrow \exists y_1 \in B_1, y_2 \in B_2, \ldots, y_n \in B_n \cdot \psi) , \]

where \( \phi \) and \( \psi \) are the pre- and postconditions of the operation, respectively. Also, \( x_i (i:1..m) \) are input (or before state) variables and \( y_j (j:1..n) \) are output (or after state) variables. To distinguish between after and before state variables, and input and output variables, we use the same conventions as in Z. That is, plain variable names are used for before state variables while primed variable names are used for after state ones, and the variable names ending with ? represent the input variables while the variable names ending with ! are used for output variables.

As in Z, we use the schema language to structure and compose descriptions: collating pieces of information, encapsulating them, and naming them for reuse. A schema consists of two parts: a declaration of variables; and a predicate constraining their values. We write the text of a schema in the following form:

\[ \text{Name} \equiv \{ \text{Declaration} \mid \text{Predicate} \} \]

The declaration and predicate are separated by a vertical bar, and the schema text is delimited by brackets. When no predicate constrains the schema variables, the predicate part can be removed.

We can use schemas as types when we require a composite type, one with a variety of different components. A schema type differs from a cartesian product in that the components are stored not by position but by name. In example 4, we will use the schema below which corresponds to a composite data type with three components: a data item of the database called dataItem, the identifier of a transaction called transId, and a \text{Read} or \text{Write} command, called command. These components together show that the command \text{command} of a transaction with the identifier \text{transId} has granted a lock for the data item dataItem.

\[ \text{Lock} \equiv \{ \text{dataItem } \in \text{DataItem}, \text{transId } \in \text{N}, \text{command } \in \text{Command} \mid \text{command } \in \{ \text{Read, Write} \} \} \]

The schema type \text{Lock} is the set of all bindings in which \text{dataItem}, \text{transId}, and \text{command} are bound to a data item, a natural number, and a command, respectively. Of course, we are limited to only those bindings in which \text{command} is a member of the set \{ \text{Read, Write} \}. As in Z, to write an object of a schema type in extension, we list the component names and the values to which they are bound. For example, for the schema type \text{Lock} above, we use \[ \text{dataItem } \mapsto x, \text{transId } \mapsto 2, \text{command } \mapsto \text{Write} \] to show a binding in which \text{dataItem}, \text{transId}, and \text{command}
are bound to $x$, 2, and $Write$, respectively. We introduce elements of the schema type $Lock$ in the usual way: the declaration $lock \in Lock$ introduces an object $lock$ of the schema type $Lock$. In other words, $lock$ is declared to be a binding that meets the constraint part of the schema $Lock$. To refer to a particular component, we employ the selection operator ".". For example, we write $lock.command$ to denote the command component of $lock$. Two bindings are equal if they bind their component names to equivalent values.

A schema may also be used whenever a declaration is expected: in a set comprehension, in another schema, or following a logical quantifier. The effect is to introduce the variables mentioned in the declaration part of the schema, under the constraint of the predicate part. As in $Z$, when we conjoin two schemas by including one in the declaration part of the other, the declarations are merged and the predicates conjoined. If the same variable is declared in both schemas, then the types must match. Otherwise, the resulting schema will be undefined. In example 3, we will use two schemas $Increasing$ and $Perm$ in the declaration part of the schema $Sort$.

The notion of state schema is used to describe system states. The form of a state schema is similar to that of schema types given earlier:

$$State \equiv [Declaration | Predicate]$$

where the declared variables correspond to the components of the specified state. The $Predicate$ part gives the properties of, and relationships between, the declared variables, and thus describes the invariant properties of the state. When the state has no such properties, the $Predicate$ part can be removed. Each object of the schema type $State$ represents a valid state: a binding of schema variables in which $Predicate$ is satisfied. We say that $Predicate$ forms part of the state invariant for the system: a constraint that must always be satisfied.

The notion of operation schema is used to describe an operation upon the state of a system. An operation schema has the following form:

$$Operation \equiv [Declaration | Precondition, Postcondition]$$

We describe an operation by including two copies of the state schema in the Declaration part of the operation schema: one representing the state before the operation; the other representing the state afterwards. To distinguish between the two, we decorate the components of the second schema, adding a single prime to each name. Some operations involve either input to the system or output from it. To model such operations, in addition to before (plain) and after (primed) state variables, we include input (end with ?) and output (end with !) variables in the Declaration part of the operation schema. The predicate part of an operation schema is divided into two parts, namely, $Preconditions$ and $Postconditions$\(^5\). Both predicates together describe the relationship between the schema variables, and thus characterize the operation: the $Precondition$ part states what must be true of the state if the effect of the operation is to be fully defined; the $Postcondition$ part describes the effect of the operation upon the values of the state variables.

Separating preconditions and postconditions is one of the major differences between operation schemas of CZ and those of $Z$. In fact, we follow the B Method or VDM in this regards. Having separate pre- and postconditions encourages the

\(^5\)Notice that the separation of preconditions and postconditions occurs only in operation schemas. In state schemas, schemas as types, and schemas as declarations, we have still a single predicate part.
There is a convention for including two copies of the same schema, one of them decorated with a prime. If \( \text{State} \) describes the state of a system, then \( \Delta \text{State} \) is a schema including both \( \text{State} \) and \( \text{State}' \): that is,

\[
\Delta \text{State} \equiv [\text{State}, \text{State}']
\]

This schema could be included whenever we wish to describe an operation that may change the state of the system. As in Z, within an operation schema, we use \( \theta \text{State} \) to indicate the before state binding. Frequently, we will wish to equate \( \theta \text{State} \) to the after state binding, thereby insisting that nothing has changed. To do this, we use a decorated version of the same binding. In other words, the decorated binding \( \theta \text{State}' \) associates the components of \( \text{State} \) with the values of decorated variables. Equating decorated and undecorated bindings of \( \text{State} \) is thus a way of insisting that the before state variables are equal to their corresponding after state variables: each component has been left with the same value. Again, there is a convention: we write \( \Xi \text{State} \) to denote the schema that includes \( \text{State} \) and \( \text{State}' \) and equates their bindings:

\[
\Xi \text{State} \equiv [\Delta \text{State} | \theta \text{State} = \theta \text{State}']
\]

This schema could be included whenever we wish to describe an operation that does not change the state of the system. Finally, as in Z, we may include a description of the initial state of the system being specified. This may be seen as the result of an operation, some form of initialization, that does not refer to the state beforehand. The initial state of a system may be described by a decorated copy of the state schema, representing the state after initialization:

\[
\text{StateInit} \equiv [\text{State}' | \text{true}, P]
\]

The predicate \( P \) describes the initial constraints upon the components of the state.

The schema calculus operations are useful in CZ as well as in the Z notation for reducing the length of a specification, modularizing it, and enabling reuse. In Z, all propositional connectives (\( \land, \lor, \Rightarrow, \Leftrightarrow \) and \( \neg \) ), \textit{Hiding} operators (\( \setminus, \forall, \) and \( \exists \) ), \textit{Sequential Composition} (\( ; \)), and \textit{Renaming} convention are defined in the world of schemas \([63, 75]\). The schema calculus of CZ consists of the operations \textit{Negation} (\( \neg_c \)), \textit{Conjunction} (\( \land_c \)), \textit{Disjunction} (\( \lor_c \)), \textit{Existential Quantifier} (\( \exists_c \)), \textit{Universal Quantifier} (\( \forall_c \)), and \textit{Sequential Composition} (\( ;_c \)), which are selected from the Z schema calculus operations. In \([47]\), all the above mentioned operators except \( \neg_c \) have been defined on both state schemas and operation schemas; \textit{Negation} has been only defined on state schemas. Here, we give the semantics of these operators when applied to operation schemas:

**Definition 2.2.1.** Let \([d | \phi, \psi]\), \([d_1 | \phi_1, \psi_1]\) and \([d_2 | \phi_2, \psi_2]\) be operation schemas. Then we have:

\[
(1) \quad [d_1 | \phi_1, \psi_1] \land_c [d_2 | \phi_2, \psi_2] \equiv [d_1, d_2 | \phi_1 \land \phi_2, \psi_1 \land \psi_2]
\]
In definition 2.2.1, as in Z, it has been assumed that the operands of the binary operators are in their normalized form and are type compatible. In [47], it has been proved that the above presented definitions of the schema calculus operations are sound, by showing that schemas constructed from the schema calculus operations in CZ are refinements of their counterparts in Z.

In examples 1.1 and 1.2, we have given informal specifications of the sorting problem and concurrency control in a database management system, respectively. In examples 2.1 and 2.2, we will write formal specifications of the two mentioned problems in the CZ notation. We will see that in the constructive approach the formal specifications remain as in the classical approach. Indeed, the difference between CZ and Z specifications lies in the interpretation and not the appearance. Therefore, when we write specifications in CZ, we can proceed as we do in Z since CZ involves the whole of the Z toolkit. We only must take care that CZ specifications are interpreted in a constructive set theory. This has important implications for the program development stage. For example, all functions and relations we use in CZ are restricted to the decidable ones which induce program developments.

In section 3, we will back to the specifications given in the next two examples by explicitly specifying the nondeterminism involved in them.

Example 3. Consider a special sorting problem: we want to sort students identifiers in an increasing order in terms of their scores in a course. The following CZ schema specifies such an operation formally. The input of the schema, i.e., \( i_n ? \), is a sequence of ordered pairs whose components are natural numbers. Each ordered pair corresponds to one student: the first component of the ordered pair is the student (unique) identifier and the second one is the student score. To develop a
program from this specification more simply later, we have considered both components of type \(N\). Hence, as has been assumed in example 1, the sequence being sorted has elements whose keys are natural numbers.

Sort \(\cong [\text{Increasing}, \text{Perm}, \text{in}\? \in \text{seq}(N \times N), \text{out}! \in \text{seq}(N \times N)] \cdot \text{true}, \text{out}! \in \text{increasing} \wedge (\text{in}?, \text{out}!) \in \text{perm}\)

This schema includes two other schemas, namely, Increasing and Perm, which will be defined later. The precondition is true and the postconditions define the intention of the sort operation, i.e., the output has to be an increasing permutation of the input sequence.

Increasing \(\cong [\text{increasing} \in P(\text{seq}(N \times N))] Φ N\sum_{i \in \text{dom} s \cdot j \in \text{dom} s \cdot i < j \Rightarrow s \cdot i \leq s \cdot j \leq 2]\)

In the above schema, increasing is the set of all sequences of ordered pairs of natural numbers that are increasingly sorted in terms of their second components. Since Increasing is not an operation schema, it includes a single predicate part.

Perm \(\cong [\text{perm} \in \text{seq}(N \times N) \leftrightarrow \text{seq}(N \times N)] \cdot \forall s \in \text{seq}(N \times N) \cdot \forall t \in \text{seq}(N \times N) \cdot (s, t) \in \text{perm} \leftrightarrow (\forall x \in \text{ran} s \cdot \text{occ}(x, s) = \text{occ}(x, t)) \wedge (\forall x \in \text{ran} t \cdot \text{occ}(x, s) = \text{occ}(x, t))\]

Two sequences are permutations of each other when the number of occurrences of every element in both sequences are equal. In the schema Perm, occ is a function that counts the number of occurrences of a given ordered pair in a given sequence of ordered pairs:

\[
\text{occ} : ((N \times N) \times \text{seq}(N \times N)) \rightarrow N
\]

(occ(\text{x}, h :: t) = occ(x, t) + 1 ∧ x = h) \lor (occ(x, h :: t) = occ(x, t) ∧ x \neq h)

All the above defined schemas are similar to their Z counterparts in appearance. In other words, they are syntactically indistinguishable from their counterparts in Z. However, they differ in their intended interpretation. For example, in the schema Increasing, the operator \(P\) denotes the decidable power set which only allows decidable subsets. A similar argument exists for the notion of relation \((\text{seq}(N \times N) \leftrightarrow \text{seq}(N \times N))\) employed in the schema Perm.

Example 4. In example 2, we described how lock-based methods control concurrency in a database management system. A well-known lock-based method is the two-phase-locking (2PL) protocol. In this protocol, before reading or writing a data item, a lock must be obtained. Once the scheduler has released any lock on behalf of a transaction \(T\), it will never grant another lock on behalf of \(T\), regardless of which data item \(T\) is requesting the lock for. A variant of 2PL is strict two-phase locking which adds the restriction that all locks must be released after the transaction commits or aborts. Now we give a CZ specification of strict 2PL. At first, we introduce two new commands of the database management system in addition to the commands read and write, defined in example 2:

- commit: Finishes a transaction and asks for finalizing the results.
- abort: Finishes a transaction and asks to abort the results.

Now we formalize the above description.

[DataItem, Value]

Command ::= Read | Write | Commit | Abort

Database \(\cong [\text{database} \in \text{DataItem} \rightarrow \text{Value}]\)
Database state is a function assigning values to all data items. A transaction is modelled as a sequence of operations in which there exists one and only one commit or abort operation at the end.

The schema Lock shows that a data item is locked for a command of a transaction determined by the identifier transId. The schema LockBase indicates the set of current locks. The invariant condition for the schema is that no two conflicting locks exist in this set.

CCDB is the state schema of the system. In this schema, activeTransactions denotes a sequence of current transactions whereas runningTransactions indicates remaining operations of each active transaction. For example, activeTransactions.i (runningTransactions.i) is associated with a binding of the schema type Transaction in which operations is bound to a sequence of all (the remaining) operations of the transaction with identifier i. To prevent the operations of a transaction from being undone when it aborts (e.g., because of deadlock), we assume that each transaction has its local copy of the database. All transactions apply operations to their local copies of the database. When a transaction sends its commit operation, its local copy is applied to the main database. localDBs.i denotes the local copy of the database maintained for the transaction with identifier i. The domains of the variables activeTransactions, runningTransactions, and localDBs correspond to the set of identifiers of all active transactions, and thus are equal to each other.

Before we give the operation schemas of the system, we write the initialization schema of the system:

Now we present the operation schemas of the system in turn. The following axiomatic definition will be used in the first operation schema.
conflictLock : (N × Transaction) → LockBase

\forall id ∈ N, transaction ∈ Transaction, lockBase ∈ LockBase
((id, transaction), lockBase) ∈ conflictLock ⇔
(∃lock ∈ lockBase.locks ⋅ (head(transaction.operations)).dataItem = lock.dataItem ∧
 id ≠ lock.transId ∧
((lock.command = Read ∧ (head(transaction.operations)).command = Write) ∨
 (lock.command = Write ∧ (head(transaction.operations)).command ∈ {Read, Write})))

Using the relation conflictLock defined by the above axiomatic definition, we can check whether the first operation of the transaction transaction whose identifier is id conflicts with the locks existing in a particular lockBase.

AcquireLock ≡ |ΔCCDB, transId? ∈ N | true,
transId? ∈ dom activeTransactions ∧
{⟨{transId?}⟩ ⋙ runningTransactions} ∩ conflictLock(lockBase) = ∅) ∧
(lockBase'.locks = lockBase.locks) ∪
{|[transId ↦ transId?], command ↦ FirstCommand(runningTransactions.transId?),
dataItem ↦ FirstDataItem(runningTransactions.transId?)|})

FirstCommand and FirstDataItem, which are not specified here, return the first command and the first data item of their input transactions, respectively. The above schema specifies the selection of a transaction that can acquire a lock for its first operation. If such a transaction exists, the lock related to its first operation is added to the set of current locks, and its identifier is returned. This schema is nondeterministic since more than one transaction may satisfy its postconditions.

Read ≡ |ΔCCDB, transId? ∈ N, value! ∈ Value | true,
FirstCommand(runningTransactions.transId?) = Read,
value! = (localDBs.transId?).database(FirstDataItem(runningTransactions.transId?)) ∧
runningTransactions' = runningTransactions ⊕
{|{transId?, |operations ↦ tail(runningTransactions.transId?.operations)|}|}

Using functional overriding (⊕) in the above schema, we specify that the first operation of the transaction with the identifier transId? is removed from the sequence of its operations.

Write ≡ |ΔCCDB, transId? ∈ N |
FirstCommand(runningTransactions.transId?) = Write,
(runningTransactions' = runningTransactions ⊕
{|{transId?, |operations ↦ tail(runningTransactions.transId?.operations)|}|})∧
(localDBs' = localDBs ⊕
{|{transId?, |database ↔ localDBs.transId?.database ⊕
{((FirstDataItem(runningTransactions.transId?),
FirstValue(runningTransactions.transId?)))}|}|)

Using ⊕ in the above schema, it has been shown that the first operation of the transaction with the identifier transId? overwrites the copy of the database maintained for transId?. FirstValue, which is not specified here, returns the value related to the first operation of its input transaction.

Commit ≡ |ΔCCDB, transId? ∈ N |
FirstCommand(runningTransactions.transId?) = Commit,
runningTransactions' = transId? ⩾ runningTransactions ∧
activeTransactions' = transId? ⩾ activeTransactions ∧
localDBs' = transId? ⩾ localDBs ∧
dataBase'.database = dataBase.database ⊕ (localDBs.transId?).database]
Abort $\simeq [\Delta CCDB, \text{transId?} \in N]$

FirstCommand$(\text{runningTransactions} \cdot \text{transId?}) = \text{Abort},$

runningTransactions$' = \text{transId?} \leq \text{runningTransactions} \land$

activeTransactions$' = \text{transId?} \leq \text{activeTransactions} \land$

localDBs$' = \text{transId?} \leq \text{localDBs}$

The schemas Commit and Abort define operations which remove the transaction with identifier transId? from active and running transactions. Due to the conditions defined in the state schema CCDB, locks acquired by transId? are removed from the set of current locks. In addition to the above mentioned operations, the schema Commit overwrites the main database by the local database of transId?.

The final specification of the strict 2PL algorithm is as follows:

STwoPL $\equiv \text{AcquireLock}_c (\text{Read} \lor \text{Write} \lor \text{Commit} \lor \text{Abort})$

STwoPL is the sequential composition of acquiring the requested lock for a transaction transId? and sending transId? for execution to one of the operation schemas Read, Write, Commit, or Abort.

As we have stated in example 2, the concurrent database management system specified formally in the above example is a nondeterministic system: at each time of the execution, there exists a set (possibly containing more than one element) of transactions which ask for a lock and do not conflict with the set of current locks; hence, the scheduler may be forced to select a transaction between several alternatives and deliver a lock to it. In section 3, we will show that a program constructed from a correctness proof of an implicitly nondeterministic specification, such as the one presented in the above example, is provided with only one of the possible behaviors. For example, our specification of the concurrent database management system results in a program that is provided with only one of the possible interleaved executions of concurrent transactions. In other words, the final program does not involve the initially specified competition among transactions.

A framework for formal specification and formal program development must guarantee that it produces all possible implementations of a nondeterministic specification. Later in the implementation phase, the program makes a choice in terms of the implementation considerations. For example, in a distributed database management system, the program may select a transaction, according to the current load of the network, from the set of those transactions that can acquire their required locks. In example 16, using the nondeterministic constructs that will be introduced in section 3, we will revise the current specification of the strict 2PL method to address the above mentioned issue.

2.3. Martin-Löf’s theory of types. Constructive type theory was originally developed as a formalism for the precise codification of the constructive mathematics by Per Martin-Löf [42]. He, in [43], has explained why the constructive theory of types may equally well be viewed as a programming language, and it no longer seems possible to distinguish the discipline of programming from constructive mathematics. Since the publication of Martin-Löf’s original paper, there has been rapidly growing literature on type theory and its application in formal specification and programming. A survey of these attempts has been given in [47].

Martin-Löf’s theory of types is well suited as a theory for program construction; the proof rules of this theory can be used to derive a correct program from the correctness proof of a specification as well as to verify that a given program has a certain property. Furthermore, this theory allows us to express both specifications
and programs within the same formalism. As a programming language, type theory is similar to typed functional languages, such as Hope and ML [55]. The relationship between type theory and functional programming has been shown in [65]. Also, in [67], Turner has given a constructive foundation for functional languages.

As it has been stated earlier, type theory was originally developed with the aim of being a clarification of constructive mathematics, but unlike most other formalizations of mathematics, type theory is not based on the first order predicate logic; instead, the predicate logic can be interpreted within type theory through the correspondence between propositions and types 9. A proposition is interpreted as a type whose elements represent the proofs of the proposition. A type cannot only be viewed as a proposition; it is also possible to see a type as a problem description, and the elements of the type as the possible solutions to the problem. Similarly, if we see the type as a specification of a programming problem, the elements are the programs that satisfy the specification. Hence, set membership and program correctness are the same problem in type theory; because all programs terminate in type theory, correctness means total correctness [55].

The language for types in type theory is similar to the type system in programming languages except that the language is much more expressive. In addition to the usual set forming operations which are found in type systems of programming languages, such as $\text{Bool}, A + B, A \rightarrow B, A \times B$, and $\text{List}(A)$, there are operations which make it possible to express properties of programs or write specifications using the usual connectives in the predicate logic. Suppose that $A$ and $B$ are types.

Then:

- The form $A \Rightarrow B$ is the type of functions $\lambda x \cdot t[x]$ such that $t[x] \in B$ for $x \in A$. This type is called function space, the type theoretical equivalent of the proposition $A \Rightarrow B$.
- The form $A \otimes B$ is a type called cartesian product of two types. It is the type of pairs $(x, y)$ such that $x \in A$ and $y \in B$. If $z$ is the ordered pair $(x, y)$, then $\text{fst}(z)$ and $\text{snd}(z)$ are equal to $x$ and $y$, respectively. This form is the type theoretical equivalent of the proposition $A \land B$.
- The form $A \oplus B$ is a type called disjoint union of two types. It is the type of objects of the form $\text{inl}(x)$ with $x \in A$ or $\text{inr}(y)$ with $y \in B$. This form is the type theoretical equivalent of the proposition $A \lor B$.
- The form $I[A, x, y]$ is a type provided that $x$ and $y$ are elements of the type $A$. It has an element $e$ if $x = y$. It can be viewed as the following proposition:

  $x$ and $y$ are identical elements of the type $A$

- The form $N$ is the type of natural numbers. It is the type of objects of the form $0$ and $s(n)$ (The successor of $n$), provided that $n$ is an element of $N$. In some of the literature (for example, [65]), for each nonnegative integer $n$, a form $N_n$ has been considered as the type containing the $n$ objects $1, 2, \ldots, n$. $N_0$ is the empty type ($\emptyset$) or the type theoretical equivalent of the

---

9. The Curry-Howard interpretation of propositions as types is one of the basic ideas behind Martin-Löf’s theory of types [65]. However, another source for type theory is the proof theory. Using the identification of propositions and types, normalizing a derivation is closely related to computing the value of the proof term which corresponds to the derivation [55].

10. For convenience and ease of reference, we adopt the presentation of the constructive type theory given in [66].
The language to express the elements of types in type theory constitutes a typed functional programming language with lazy evaluation order. The program forming operations are divided into constructors and selectors. Constructors are used to construct objects in a type from other objects, examples are 0, s (The successor function), inl, inr and λ. Selectors are used as a generalized pattern matching, examples are fst and snd [55].

In this paper, we use an intensional type theory which has been strengthened with the type U (universe) and the type constructor W (well-founded). Intensional means that the judgmental equality is understood as definitional equality; in particular, the equality is decidable (see definition 2.1.3). The existence of the intensional equality is a prerequisite for developing computerized tools that support the construction of proofs within type theory [55]. In the following paragraphs, we will illustrate the intuition behind the type U and the type constructor W. Both of these play an important role in interpreting the CZ set theory in Martin-Löf’s theory of types.

All types that have so far been introduced are called small types since their elements are not themselves types. With only small types in place, type theory cannot handle the following situations for example [65]:

- We may wish to make an element depend upon a type parameter.
- We might want to assert the existence of a type with certain properties: this is the content of an abstract type definition.
- Some functions are most naturally defined over the collection of all objects of all types.

For this reason and generally to have a richer notion of type, we can see the merit of introducing a type U which is called the universe and whose elements are themselves types together with the reflection principle. Roughly speaking, this principle says that whatever we can do with types can also be done inside the universe U. For instance, this principle leads us to have

$$\emptyset \in U,$$
$$N \in U,$$
$$A \oplus B \in U,$$
$$I[A, x, y] \in U, (x \text{ and } y \text{ are elements of the type } A),$$
$$(\Pi x \in A.B) \in U; \text{ and}$$
$$(\Sigma x \in A.B) \in U,$$
where $A$ and $B$ are types in $U$. Note that we do not wish to have $U \in U$, because it would mean, intuitively, that $U$ is the universe of all types, from which a paradox can be derived [66].

![Figure 1. Two natural numbers 0 and 2, and the sequence $\langle 1, 0 \rangle$ [65].](image1)

![Figure 2. A binary numeric tree [65].](image2)

Now we give the intuition behind the constructor $W$. To incorporate recursive data types like lists and trees into type theory, Martin-Löf has proposed well-founded types. They are types over which we can define objects by recursion and prove properties by induction. Informally, for this to be possible, we need to be sure that when we make a recursive definition (or an inductive proof), we never encounter an infinite sequence of activities [65]. In general, we can think of the elements of any recursive type as trees. In Fig. 1, we see three trees representing the natural numbers 0 and 2 (i.e. $\text{suc}(\text{suc}(0))$), and the sequence $\langle 1, 0 \rangle$. Fig. 2 shows an example from binary numeric trees.

The general form of elements of well-founded types is that each node is built from a certain collection of predecessors of the same type. Considering the type of binary numeric trees, null nodes, illustrated by black discs in Fig. 2, have no predecessors, whereas other nodes, shown in white, have two predecessors. For a general recursive type, we will have a type $A$ of sorts of node. In the case of binary numeric trees, this type is best thought of as a disjoint union type, $T \oplus N$, where $T$ is the one element type, for the null node, and $N$ is for the non-null nodes, which carry numbers.

Different kinds of nodes have different numbers of predecessors. For a particular kind of node $a \in A$, we specify what form the predecessors of the node take by supplying a type $B(a)$, which we can think of as the type of names of predecessors places. For a particular node of that sort, we specify the collection of predecessors of the node by a function from $B(a)$ to the type in question. Considering the particular case of the type binary numeric trees, since the null node has no predecessors, we say $B(\text{null}) \equiv_{df} \emptyset$, and for a binary node $a$, we have two predecessors, so we define $B(a) \equiv_{df} N_2$. 
The type we build is determined by the class of sorts of node, $A$, and the family determining the nature of the set of predecessors of each sort of node, $B(x)$. The type thus constructed is $Wx \in A.B$ whose elements denoted by $sup(a,f)$, where $a \in A$ and $f$ is a function from $B[a]$ to $Wx \in A.B$. $W$ is used as a reminder that the type is well-founded. An important property of the type constructor $W$ is that we can always, for each element $\alpha \in (Wx \in A.B)$, recover a node $a \in A$ and a function $f \in (B[a] \Rightarrow (Wx \in A.B))$, such that $\alpha = sup(a,f)$ [66].

After illustrating the language of types, specially the type $U$ and the type constructor $W$ in type theory, now we briefly review the program development process in this theory: in type theory, specifications are expressed by mathematical propositions, and program development process consists of functional specification of a problem, by giving its type, and then constructing an object of that type, using the inference rules of the logic. This object, if existing, can be viewed as a program which satisfies the specification. The inference rules in type theory are presented in Gentzen’s natural deduction style. For each type constructor in type theory, there are four types of inference rules:

- **Formation rule**: This rule states under what conditions $A$ is a type.
- **Introduction rule**: The constructors of a type, i.e. operators that construct elements of the type, are introduced by this rule which corresponds to the introduction rules of natural deduction systems for predicate logic. In other words, this rule states how to form the canonical elements of a type $A$ and when two canonical elements are equal. Intuitively, the canonical elements are the final values of programs whose evaluations cannot proceed. Examples of canonical elements in type theory are $3$, true, $(3,4)$, and $\lambda x \cdot x$ and examples of non-canonical elements are $3 + 5$, $(\lambda x \cdot x + 1)(12 + 13)$, and $if \ 3 = 4 \ then \ fst((3,4)) \ else \ snd((3,4))$. 

Figure 3. The inference rules for the type constructor $\oplus$ [65].
For instance, we show the inference rules for the type constructor \( \oplus \) in the second, by applying \( gC \) of \( \text{inr} \) having the form \( A \) be a (tagged) member of \( \text{inr} \) (see Fig. 3), can be described as follows: we know that the object \( r \) should either be a (tagged) member of \( A \), having the form \( \text{init}(p) \) or be a (tagged) member of \( B \), having the form \( \text{inr}(q) \). The functions \( f \) and \( g \) are sufficient to give us a member of \( C \) in either case: in the first case, we get a member of \( C \) by applying \( f \) to \( p \), and in the second, by applying \( g \) to \( q \). This computational information is expressed by the computational rule for \( \oplus \) (see Fig. 3).

Before ending this subsection, in the next example, we write a type theoretical specification of the sorting problem that we have given its CZ specification in example 3. By this specification, those readers who are familiar with set theoretical specification languages can see how a specification in Martin-Löf’s theory of types looks like. Also, we can compare this specification with the specification that will be derived in the next subsection from the interpretation of the CZ specification in type theory.

**Example 5.** The type theoretical specification of the sorting problem, which was specified in example 3, is as follows:

\[
\text{Sort} \cong \Pi l_1 \in \text{List}(N \otimes N) \cdot \Sigma l_2 \in \text{List}(N \otimes N) \cdot \text{Increasing}(l_2) \otimes \text{Perm}(l_1, l_2)
\]

For convenience, we assume the existence of the type constructor \( \text{List} \), as a special case of \( W \), throughout the paper. We use \( \text{List} \) for modelling ordered collections of objects. The inference rules for this type can be found in [47]. The empty list is shown as \([\ ]\); a non-empty list \( l \) is written as \( h \mathbin{::} t \) where \( h \) and \( t \) are the head and the tail of \( l \), respectively. The \( i \)th element of the list \( l \) is written as \( l(i) \). Two predicate functions \( \text{Increasing} \) and \( \text{Perm} \) specify the intention of the sort operation, i.e., the output has to be an increasing permutation of the input sequence. We define these two functions as follows:

\[
\text{Increasing}(l) \cong \Pi i \in N \cdot \Pi j \in N \cdot \left( \Sigma m \in N \cdot \Pi [N, j, i + n] \Rightarrow (\Sigma m \in N \cdot I[N, \text{snd}(l(j)), \text{snd}(l(i)) + n]) \right)
\]

\[
\text{Perm}(l_1, l_2) \cong \Pi \pi x \in \pi l_1 \cdot I[N, \text{occ}(x, l_1), \text{occ}(x, l_2)] \otimes \Pi \pi x \in \pi l_2 \cdot I[N, \text{occ}(x, l_1), \text{occ}(x, l_2)]
\]

Two sequences are permutations of each other when the number of occurrences of every element in both sequences are equal. In the above definition, \( \text{occ} \) is a function that counts the number of occurrences of a given ordered pair in a given sequence of ordered pairs. We define \( \text{occ} \) as follows:

\[
\text{occ} = \lambda n \cdot \lambda s \cdot \text{brec}(s, 0, \lambda h \cdot \lambda t \cdot \lambda r \cdot \text{if } n = h \text{ then } r + 1 \text{ else } r)
\]

\( \text{brec} \) is the symbol of recursion on lists. Indeed, it is the selector of type \( \text{List} \), derived from the elimination rule for this type. You can see the rule in [47]. However, we present the definition of \( \text{brec} \) here:

\[
\text{brec}(\cdot, e, f) = e
\]

\[
\text{brec}(h :: t, e, f) = f(h, t, \text{brec}(t, e, f))
\]

Therefore, \( \text{occ} \) is the following recursive function:
\[
\text{occ} : ((N \otimes N) \otimes \text{seq}(N \otimes N)) \Rightarrow N
\]
\[
\text{occ}(x, \emptyset) = 0
\]
\[
\text{occ}(x, h :: t) = \text{occ}(x, t) + 1 \quad \text{if } x = h
\]
\[
\text{occ}(x, h :: t) = \text{occ}(x, t) \quad \text{if } x \neq h
\]

In example 6, we will extract a program from a correctness proof of the type theoretical specification of the sorting problem.

2.4. Interpretation of CZ in Martin-Löf’s theory of types. In [47], to give a constructive meaning to the set theoretical notions of CZ, their interpretation were justified into Martin-Löf’s theory of types. Informally, the idea behind the interpretation is to find a constructive version of the classical, iterative notion of sets. In order to interpret the language of the CZ set theory, it is necessary to have a type (or set) which can be used as the universe of sets and the model for CZ. More precisely, the intention is to build a model \( \nu = <V, \varepsilon, \infty> \) of CZ in Martin-Löf’s theory of types\(^{11}\) in which each set is associated with a pair consisting of a base type together with a family of types, i.e. its elements. This is furnished as follows:

\[
V \cong Wx \in U.x,
\]
where \( U \) is the universe, and \( W \) is the type constructor for recursive types (see subsection 2.3). As has been described in subsection 2.3, each \( \alpha \in V \) can be split into two components \( \alpha^- \) and \( \alpha^\sim \) such that \( \alpha = \text{sup}(\alpha^-, \alpha^\sim) \) where \( \alpha^- \in U \) and \( \alpha^\sim \in \alpha^- \Rightarrow V \) (Notice that in the definition of \( V \) above, \( B(x) \) is equal to \( x \). Thus, \( B(\alpha^-) = \alpha^- \) here). Therefore, \( V \) is a tree whose nodes are members of \( U \), and thus are themselves types having a predecessor for each of their elements (because of \( B(x) = x \)). In other words, each node of \( V \) is a type whose elements (or predecessors of the node) are themselves types. In this way, we obtain an interpretation of the iterative notion of sets in the constructive type theory.

To complete the description of the model, we need to define the two binary relations \( \varepsilon \) and \( \infty \). It is necessary to distinguish between \( = \) and \( \in \) on the one hand, which are primitive symbols in type theory, and \( \varepsilon \) and \( \infty \), which have been defined as the type theoretical interpretations of \( = \) and \( \in \) from CZ, respectively. The equality \( \varepsilon \) between \( \alpha \) and \( \beta \) of \( V \) is defined as follows:

\[
\alpha \varepsilon \beta \cong (\Pi x \in \alpha^- \cdot \alpha^\sim x \in \beta) \otimes (\Pi x \in \beta^- \cdot \beta^\sim x \in \alpha)
\]

Thus, the equality between sets is explained in terms of the equality between their elements. In other words, the above definition corresponds to the extensional equality in set theories, stated by the Extensionality axiom. Indeed, this definition can be considered as the interpretation of the Extensionality axiom of the CZ set theory in type theory (for a detailed description, see [47]). The membership \( \infty \) between \( \alpha \) and \( \beta \) is now explained as:

\[
\alpha \in \beta \cong \Sigma x \in \beta^- \cdot \beta^\sim x = \alpha
\]

The symbols \( = \) and \( \in \) have been defined by simultaneous recursion. if \( \Phi[x] \) is a proposition for \( x \in V \), then by definition we have:

\[
\forall x \in \alpha \cdot \Phi[x] \cong \forall x \in \alpha^- \cdot \Phi[\alpha^\sim x]
\]
\[
\exists x \in \alpha \cdot \Phi[x] \cong \exists x \in \alpha^- \cdot \Phi[\alpha^\sim x]
\]

Now using the model \( \nu \), we interpret the set theoretical notions of CZ in type theory. We begin by giving the interpretation of the empty set and the set of natural numbers.

\(^{11}\)This approach for model construction has been adopted from that given in [1].
Follows \[65\]:

We can therefore take \( R_0 \in \emptyset \to V \)\(^{12}\), we certainly have \( \sup(\emptyset, R_0) \), which we shall abbreviate \( \Theta \), as the interpretation of the empty set and the natural number 0. Having the interpretation of the natural number 0, we can use the von Neumann definition for natural numbers to interpret other natural numbers:

\[ 0 \cong \emptyset \]
\[ 1 \cong \{0\} = \{\emptyset\} \]
\[ 2 \cong \{0, 1\} = \{\emptyset, \{\emptyset\}\} \]
\[ 3 \cong \{0, 1, 2\} = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \]

Thus, if \( s(n) \) denotes the successor of \( n \), then we have \( s(n) = n \cup \{n\} \).

According to the above discussion, if in the model \( \nu \), boldface numbers represent natural numbers, and \( \alpha^+ \) is the successor of \( \alpha^{13} \); we have the following definitions:

\[ 0 \cong \Theta = \sup(\emptyset, R_0) \]
\[ \alpha^+ \cong \sup(\alpha \cong \{\alpha\}^{14}, \lambda u \cdot \text{case } u \alpha^{-} \alpha) \]

where \( u \in \alpha^+ \cong \{\alpha\} \), \( \alpha^{-} \in (\alpha^{-} \Rightarrow V) \), and \( \alpha \) is a constant function here. For example, for the natural number 1, we can write:

\[ 1 \cong \emptyset^0 = \sup(0^0 \cong \emptyset^0, \text{case } u \emptyset^0 \emptyset) = \sup(\emptyset \cong \emptyset, \text{case } u R_0 \emptyset) \]

Similarly, we can treat the natural numbers 2, 3, \ldots. Now let \( \omega \cong \{0, 1, 2, \ldots\} \) be the set of natural numbers in \( \nu \); hence, it is a member of \( V \), and we can define it as follows:

\[ \omega \cong \sup(N, \text{natrec}(n, \Theta, \lambda x \cdot \lambda y \cdot y^+)) \]

where \( n \in N \), and \( \text{natrec} \) is the symbol of recursion on natural numbers. Indeed, \( \text{natrec} \) is the selector of type \( N \), derived from the elimination rule for this type.

You can see the rule in [47]. However, we present the definition of \( \text{natrec} \) here:

\[
\text{natrec}(0, e, f) = e
\]
\[
\text{natrec}(s(u), e, f) = f(u, \text{natrec}(u, e, f))
\]

It is easy to see that \( \omega^0 = 0, \omega^1 = 1, \omega^2 = 2, \ldots \). For example, for \( \omega^2 \), we have:

\[
\omega^2 = \text{natrec}(2, \Theta, \lambda x \cdot \lambda y \cdot y^+) = sf(1, \text{natrec}(1, \Theta, sf)) = (\text{natrec}(1, \Theta, sf))^+ = (sf(0, \text{natrec}(0, \Theta, sf))^+) = ((\text{natrec}(0, \Theta, sf))^+)^+ = (\Theta^+)^+ = (0^+)^+ = \Theta^+ = 2
\]

where we have used \( sf \) as an abbreviation for the function \( \lambda x \cdot \lambda y \cdot y^+ \).

After interpreting the empty set and the set of natural numbers in type theory, in [47], an assignment function \( \xi \) has been defined which assigns elements of \( V \) to well-formed formulas and atomic formulas of the CZ set theory (See the BNF syntax for the language of CZ presented in subsection 2.1). The following equalities define this function:

\[
[\Omega]_\xi = \Omega
\]
\[
[x = y]_\xi = [\xi(x) = \xi(y)]
\]
\[
[x \in y]_\xi = [\xi(x) \in \xi(y)]
\]
\[
[\phi \wedge \psi]_\xi = [\phi]_\xi \wedge [\psi]_\xi
\]
\[
[\phi \lor \psi]_\xi = [\phi]_\xi \lor [\psi]_\xi
\]

\(^{12}\)For any type \( T \), there is a function from the type \( \emptyset \) to \( T \), given by the \textit{abort} construct as follows [65]:

\[ \textit{afun} \equiv_{df} \lambda x \cdot \text{abort} T x \]

We can therefore take \( R_0 \equiv_{df} \lambda x \cdot \text{abort} V x \) here.

\(^{13}\)In the subsequent sections, we use the more familiar notation \( s(\alpha) \) to denote the successor of \( \alpha \) in the model \( \nu \). Here we dedicate a new notation to distinguish it from its counterpart in CZ.

\(^{14}\)Indeed, \( \alpha^{-} \cong (\alpha^{-}) \) is \( s(\alpha^{-}) \) which itself is a natural number in CZ, the successor of \( \alpha^{-} \).
which will be required in our subsequent examples and proofs in the paper: V shown how each construct of the CZ set theory, which is related to an axiom, is or as a part of a declaration.

28 HASSAN HAGHIGHI AND SEYYED-HASSAN MIRIAN-HOSSEINABADI

... postconditions of operations, respectively. Also, \( x_j(j : 1..m) \) are input (or before state) variables, and \( y_k(k : 1..n) \) are output (or
after state) variables. Now we extend the function $\xi$ to translate the schemas $\text{Schema}_1$ and $\text{Schema}_2$ above into two elements of $V$ as follows:

$$[\text{Schema}_1]_\xi = \Sigma \alpha_1 \in (\xi(S_1))^-, \alpha_2 \in (\xi(S_2))^-, \ldots, \alpha_1 \in (\xi(S_1))^- \cdot ([\theta]_\xi)[(\xi(S_1))\alpha_1/x_1]$$

$$[\text{Schema}_2]_\xi = \Pi \alpha_1 \in (\xi(A_1))^-, \alpha_2 \in (\xi(A_2))^-, \ldots, \alpha_m \in (\xi(A_m))^- \cdot (\xi(B_1))^- \cdot \beta_1 \in (\xi(B_2))^- \cdot \ldots \cdot \beta_n \in (\xi(B_n))^- \cdot (\psi)\xi)[(\xi(\Sigma A_i))\alpha_1/x_1][(\xi(B_i))\beta_1/y_1]$$

By the definition of the function $\xi$ and its recent extension, we can conclude the following equalities:

$$[\text{Schema}_1]_\xi = [\exists s_1 \in S_1, s_2 \in S_2, \ldots, s_1 \in S_1 \cdot \theta]_\xi$$

$$[\text{Schema}_2]_\xi = [\forall x_1 \in A_1, x_2 \in A_2, \ldots, x_m \in A_m \cdot (\phi \Rightarrow \exists y_1 \in B_1, y_2 \in B_2, \ldots, y_n \in B_n \cdot \psi)]_\xi$$

The above equalities introduce a constructive interpretation of schemas which induces proof obligations which facilitates the translation of CZ schemas into type theory and thereby yields implementations of these schemas. Now, given a specification in CZ, we can use the function $\xi$ to translate the specification into a type in type theory and then extract a program (a term in type theory) which meets the specification (more precisely, meets its representation in type theory). To illustrate this approach, in the next example, we employ the CZ specification of the sorting problem given in example 3. We show that our method of program development extracts a deterministic program from this specification whereas the specification is itself nondeterministic. In the next section, we will investigate this problem in detail.

**Example 6.** We use the extended version of the function $\xi$ to translate the operation schema $\text{Sort}$, given in the CZ specification of the sorting problem (see example 3). For convenience, we do not translate two schemas $\text{Increasing}$ and $\text{Perm}$ which are included in the schema $\text{Sort}$. To consider these schemas in the translation process, we can bring their declaration and predicate parts into $\text{Sort}$ before we begin the translation.

$$[\text{Sort}]_\xi = \Pi \alpha \in (\xi([seq(N \times N)]))^- \cdot \Sigma \beta \in (\xi([seq(N \times N)]))^-$$

$$= \Pi \alpha \in (\xi([seq(N \times N)]))^- \cdot \Sigma \beta \in (\xi([seq(N \times N)]))^-$$

Since $\alpha$ and $\beta$ are themselves elements of $V$, we have

$$[\xi([seq(N \times N)])]\alpha = \alpha$$

$$[\xi([seq(N \times N)])]\beta = \beta$$

Thus, the following equality holds:

$$[\text{Sort}]_\xi = \Pi \alpha \in (\xi([seq(N \times N)]))^- \cdot \Sigma \beta \in (\xi([seq(N \times N)]))^- \cdot \beta \in [\xi(\text{Increasing}) \otimes (\alpha, \beta)]_\xi$$

To continue the translation, we use the interpretations of the set of natural numbers and the cartesian product constructor in type theory, stated in this subsection. On the other hand, in [47], $\text{seq}X$ was translated into a set of finite boolean-valued functions whose domains are decidable subsets of $N \otimes (\xi(X))^-$. This interpretation then was proved to be equivalent to the semantics of the type constructor $\text{List}$ in type theory (For a detailed description, see [47]). Thus, we can easily map $\text{seq}X$ of CZ to $\text{List}(X)$ of type theory. Using this mapping and the interpretations of the set of natural numbers and the cartesian product constructor in type theory, we can conclude the following equality:

$$[\text{Sort}]_\xi = \Pi \alpha \in \text{List}(N \otimes N) \cdot \Sigma \beta \in \text{List}(N \otimes N) \cdot \beta \in [\xi(\text{Increasing}) \otimes (\alpha, \beta)]_\xi$$
Notice that the resulting type theoretical specification of the sorting problem is similar to the one which was directly written in example 5. We now derive a program from a correctness proof of the resulting specification. An initial part of such a proof is shown in Fig. 4 (To see the complete proof, refer to [47]). In the proof tree, we have used two conventions \( IN = \xi(\text{increasing}) \) and \( P = \xi(\text{perm}) \). Also, throughout the paper, the dashed lines in proof trees indicate those parts of the proof that must proceed, but are not shown in the proof tree; by a solid line, however, we transfer a part of the proof to another space because of the lack of the space.

The extracted program is as follows:

\[
\text{sort} = \lambda \alpha \cdot \text{rec}(\alpha, (\lambda h_1, r), \lambda \alpha \cdot \lambda x \cdot \lambda h_1 \cdot (\text{Insert}(\alpha, \text{fst}(h_1)), q))
\]

We first give an abstract illustration of the program. The readers interested in details of the program can see them later in this example. The program \text{sort} is a recursive function that for each input sequence of ordered pairs, results in a permutation in which the elements are in the increasing order in terms of their second components:
sort : List(N ⊗ N) ⇒ List(N ⊗ N)
sort(()) = ()
sort(h :: t) = Insert(h, sort(t)),
where the function Insert(n, s) results in a permutation of ⟨n⟩ ⊲ s whose elements are in the increasing order in terms of their second components, provided that the elements of s are themselves in the increasing order:

Insert : (N ⊗ N) ⊗ List(N ⊗ N) ⇒ List(N ⊗ N)
Insert(n, ⟨⟩) = ⟨⟩
Insert(n, h :: t) = if snd(n) ≤ snd(h) then ⟨n⟩ ⊲ (h :: t) else ⟨h⟩ ⊲ Insert(n, t)

A detailed description of the program sort is as follows: r and q are the proof objects of ⟨⟩ and Insert(a, fst(h1)), respectively (see Fig. 4). lrec is the symbol of recursion on lists defined in the previous subsection (see example 5 where the function occ has been defined). Thus, sort is a recursive function which results in ⟨⟩ for the input α = ⟨⟩. In the recursive step, the program tries to compute an output for the input α = h :: t while it has generated an output for the input α = t. Since the output for α = t, i.e. h1, is an ordered pair whose second element is the proof object of the first one, applying fst to it removes the proof part. Thus, we can conclude that sort(h :: t) = Insert(h, sort(t)).

The rules that yield v = Insert(a, fst(h1)) have not been shown in Fig. 4. Nevertheless, these rules can validate the following equality [47]:

Insert = λn · λs · lrec(s, ⟨n⟩), λh · λt · λr · if snd(n) ≤ snd(h) then ⟨n⟩ ⊲ (h :: t) else ⟨h⟩ ⊲ r

The program sort extracted in example 6 is correct according to the initial specification of the sorting problem given in CZ: for each input sequence of ordered pairs, this program produces a permutation in which the elements are in the increasing order in terms of their second components. However, it has a deterministic behavior and produces only one of the possible outputs for each input sequence whereas the initial specification is nondeterministic: for input sequences having different elements with identical keys, the operation schema Sort allows more than one sorted permutation. In the next section, we will look for the origin of this problem by investigating our program development method when applied to a typical nondeterministic CZ specification. We will show that the initial nondeterminism, specified implicitly in CZ specifications, will be lost during the process of program development. Such a result seems not to be wrong in a situation where the abstraction involved in the specification leads to nondeterminism while we still want the final implementation to be deterministic (e.g., in the sorting problem); however, in such situations, we still encounter a problem: the programmer is deprived of some routes to take in developing the program or some good implementations might be excluded. The above problem seems to be more serious when we consider specifications of nondeterministic programs (See our description given for the specification of a concurrent database management system at the end of example 4).

In the next section, we will see that the above mentioned problem could be addressed by specifying all possible behaviors directly in a nondeterministic specification. However, it yields longer and more complicated specifications which are harder to read and write. Also, the writer of a nondeterministic specification him/herself must take care that his/her method to specify all possible behaviors is suitable at the same time that s/he describes the functional properties of the system. Both
of these issues become more serious when we need to specify modalities of nondeterminism. Therefore, we will introduce a number of nondeterministic constructs and a set of new operations for the schema calculus by which we can specify modalities of nondeterminism easily at the same time that we use our usual procedures to specify functional properties. On the other hand, these constructs enter into specifications without changing their main structures. Thus, they do not yield longer and more complicated specifications. We will interpret all the new constructs in CZ itself; hence, using the current translation of CZ into Martin-Löf’s theory of types [47], we can develop programs which are provided with all possible implementations according to the initially specified modalities of nondeterminism.

3. Nondeterminism in CZ

In this section, we first explore the notion of nondeterminism in CZ specifications from a formal program development point of view. We then introduce a mathematical toolkit to specify modalities of nondeterminism in CZ.

3.1. Exploring nondeterminism. At first, it is necessary to formalize the notion of nondeterminism in CZ. To achieve this goal, as in [53], we regard the existence of several after state valuations for a single before state binding in an operation schema as a clear notion of nondeterminism in CZ. The following definition formalizes this notion:

**Definition 3.1.1.** An operation schema $Op$ is nondeterministic iff $\exists b_1, b_2 \in Op \cdot b_1 \neq b_2 \land b_1^{bi} = b_2^{bi}$.

For a binding $b \in Op$, we use the notation $b^{bi}$ to denote that part of $b$ in which before state or input variables of $Op$ are bound to some values. By the above definition, an operation schema is nondeterministic iff there exists a combination of values of before state and input variables that with two or more different combinations of values of after state and output variables satisfy the schema predicates.

**Example 7.** According to definition 3.1.1, the operation schema $Sort$ given in example 3 is nondeterministic since there are two different bindings

\[
\begin{align*}
\text{in?} & \mapsto \{(1, 16), (2, 18), (3, 16)\}, \quad \text{out?} \mapsto \{(1, 16), (3, 16), (2, 18)\} \in Sort \\
\text{in?} & \mapsto \{(1, 16), (2, 18), (3, 16)\}, \quad \text{out?} \mapsto \{(3, 16), (1, 16), (2, 18)\} \in Sort
\end{align*}
\]

and $\text{in?} \mapsto \{(1, 16), (2, 18), (3, 16)\} \in Sort$, and $\text{in?} \mapsto \{(1, 16), (2, 18), (3, 16)\} \in Sort$. Notice that a specification may be deterministic whereas it involves a nondeterministic schema. For example, consider the following simple specification:

\[
\begin{align*}
\text{Report} & := \text{OK} \mid \text{Not\_Exists} \\
S_1 & \equiv \{a? \in N, b! \in N, r! \in \text{Report} \mid a? > 0, b! < a? \land r! = \text{OK}\} \\
S_2 & \equiv \{a? \in N, r! \in \text{Report} \mid a? = 0, r! = \text{Not\_Exists}\} \\
TS & \equiv \exists b! \in N \cdot (S_1 \lor c \land_S S_2)
\end{align*}
\]

Using the definition of $\exists c$ and $\lor c$, we can show the following equality:

\[
\begin{align*}
TS & \equiv \{a? \in N, r! \in \text{Report} \mid \text{true}, \exists b! \in N \\
& \quad (a? > 0 \land b! < a? \land r! = \text{OK}) \lor (a? = 0 \land r! = \text{Not\_Exists})\}
\end{align*}
\]

While the schema $S_1$ is nondeterministic, the schema $TS$ which includes $S_1$ is itself deterministic. Informally, we can say a specification is nondeterministic iff after applying all operations of the schema calculus existing in the specification, it still involves at least one nondeterministic schema.

The existence of nondeterministic schemas may cause specifications to involve nondeterminism implicitly. As it has been shown in example 6, after deriving a
program from a correctness proof of an implicitly nondeterministic specification, only one of the possible outputs is guaranteed for each input. We investigate this problem more precisely using the general form of operation schemas in CZ, defined in subsection 2.2 as follows:

\[ \text{OP}_\text{Schema} \cong [x_1 \in A_1, x_2 \in A_2, \ldots, x_m \in A_m, y_1 \in B_1, y_2 \in B_2, \ldots, y_n \in B_n \mid \phi, \psi], \]

where \( \phi \) and \( \psi \) denote the pre- and postconditions of the specified operation, respectively. Also, \( x_i (i : 1..m) \) are its input (or before state) variables, and \( y_j (j : 1..n) \) are its output (or after state) variables. In subsection 2.4, we have defined the assignment function \( \xi \) which interprets formulas of the language of CZ in type theory. We have also extended this function to translate schemas of CZ in type theory. By the extended version of \( \xi \), the following equality holds:

\[ [\text{OP}_\text{Schema}]_\xi = \Pi \alpha_1 \in (\xi(A_1))^-, \alpha_2 \in (\xi(A_2))^-, \ldots, \alpha_m \in (\xi(A_m))^-, \]

\[ ([\phi]_\xi \Rightarrow \Sigma \beta_1 \in (\xi(B_1))^-, \beta_2 \in (\xi(B_2))^-, \ldots, \beta_n \in (\xi(B_n))^-[\psi]_\xi)]([\xi(A_1)\alpha_1/x_1][\xi(B_1)\beta_1/y_1]), \]

where \([\text{OP}_\text{Schema}]_\xi\) is the type theoretical equivalent of \( \text{OP}_\text{Schema} \). We use the following conventions:

\[ A_i' = (\xi(A_i))^- \text{ for } i : 1..m \]
\[ B_j' = (\xi(B_j))^- \text{ for } j : 1..n \]
\[ \phi' = [\phi]_\xi[\xi(A_1)\alpha_1/x_1][\xi(B_1)\beta_1/y_1] \]
\[ \psi' = [\psi]_\xi[\xi(A_1)\alpha_1/x_1][\xi(B_1)\beta_1/y_1] \]

Therefore, \([\text{OP}_\text{Schema}]_\xi\) is equal to the following type in type theory:

\[ \Pi \alpha_1 \in A_1', \ldots, \alpha_m \in A_m', (\phi' \Rightarrow \Sigma \beta_1 \in B_1', \beta_2 \in B_2', \ldots, \beta_n \in B_n' \cdot \psi') \]

\[
\begin{align*}
\Gamma &\equiv \alpha_1 \in A_1', \ldots, \alpha_m \in A_m' \\
\text{prog} &\equiv \lambda (\alpha_1, \ldots, \alpha_m) \cdot t \\
t &\equiv ((v_1, \ldots, v_n), q) \\
\text{Hyp} &
\end{align*}
\]

\[ \Gamma \vdash v_1 \in B_1', \ldots, v_n \in B_n' \quad \Gamma \vdash q \in (\psi'[v_1/\beta_1] \ldots [v_n/\beta_n]) \quad \sum j \]

\[ \Gamma \vdash t \in (\sum \beta_1 \in B_1', \ldots, \beta_n \in B_n', \phi') \quad \Pi i \]

\[ \sum \beta_1 \in B_1', \ldots, \beta_n \in B_n', \phi' \]

\[ \text{prog} \in (\Pi \alpha_1 \in A_1', \ldots, \alpha_m \in A_m', \sum \beta_1 \in B_1', \ldots, \beta_n \in B_n', \phi') \]

\[ \text{Figure 5. Program extraction from the operation schema OP_Schema} \]

We can now derive a program from a correctness proof of the above type theoretical specification. An initial part of such a proof is shown in Fig. 5. The extracted program is as follows:

\[ \text{prog} = \lambda (a_1, a_2, \ldots, a_m) \cdot \lambda t \cdot ((v_1, v_2, \ldots, v_n), q), \]
where \( t_1 \) and \( q \) are the proof objects of \( \phi' \) and \( \psi'[v_1/\beta_1][v_2/\beta_2]...[v_n/\beta_n] \), respectively. For each valuation of \( \alpha_1 \in \mathcal{A}_1', \alpha_2 \in \mathcal{A}_2', ..., \alpha_m \in \mathcal{A}_m' \), this program produces only one output, i.e. the \( n \)-ary \( (v_1,v_2,...,v_n) \), although the schema \( Op\_Schema \) may be itself nondeterministic according to definition 3.1.1. This problem is due to the fact that when we use the introduction rule for dependent sum \( (\Sigma_i) \) in the proof tree (see the circled \( \Sigma_i \) in Fig. 5), we can replace the \( n \)-ary \( (\beta_1,\beta_2,...,\beta_n) \) by only one value.

It seems that if the writer of a nondeterministic specification indicates all possible behaviors explicitly in the initial specification, then a program can be obtained that involves all possible behaviors. In this way, the mentioned problem will be addressed without changing the rules of type theory and/or the translation of CZ into type theory. To evaluate this solution, suppose that the operation schema \( Op\_Schema \), which has been introduced earlier in this section, is nondeterministic (according to definition 3.1.1). We rename and rewrite this schema as follows:

\[
N_{Op\_Schema} \cong [x_1 \in A_1, x_2 \in A_2, ..., x_m \in A_m, \text{aovar} \in P (B_1 \times B_2 \times ... \times B_n) \mid \phi, \forall (y_1, y_2, ..., y_n) \in (B_1 \times B_2 \times ... \times B_n) \cdot (y_1, y_2, ..., y_n) \in \text{aovar} \Leftrightarrow \psi]
\]

In the new schema, we have promoted the combination of the after state and output variables to a maximal set (i.e. \( \text{aovar} \)) of possible combinations of these variables which satisfy the postcondition of the schema. Notice that if we promoted each after state or output variable to a distinct set and placed a universal quantifier in front of the schema postconditions to contain all possible values, the relationship between valuations that make the schema predicates true would disappear. To overcome this problem, we have combined all previous after state and output variables in a new variable using the cartesian product of their types. Theorem 3.1.2 shows that applying our program development method to the schema \( N_{Op\_Schema} \) rather than \( Op\_Schema \), we can extract a program which preserves the initial nondeterminism involved in \( Op\_Schema \).

**Theorem 3.1.2.** Let \( \text{prog} \) be the program that is constructed after applying the function \( \xi \) to the schema \( N_{Op\_Schema} \) and then proving the correctness of the resulting type theoretical specification. For all \( X_1 \in A_1, ..., X_m \in A_m, \) if \( \phi(X_1, ..., X_m) = \text{true} \), then \( \text{prog} \), when applied to \( \xi(X_1), ..., \xi(X_m) \), produces a boolean-valued function \( \text{dv} : \{\xi(B_1) \otimes ... \otimes \xi(B_n)\} \Rightarrow \text{Bool} \) such that for all \( Y_1 \in B_1, ..., Y_n \in B_n, \)

\[
\text{dv}(\{\xi(Y_1), ..., \xi(Y_n)\}) = \text{true} \iff [x_1 \mapsto X_1, ..., x_m \mapsto X_m, y_1 \mapsto Y_1, ..., y_n \mapsto Y_n] \in Op\_Schema.
\]

**Proof.** We first apply the function \( \xi \) to the schema \( N_{Op\_Schema} \) as follows:

\[
[N_{Op\_Schema}]_{\xi} = \Pi \alpha_1 \in (\xi(A_1))^{-}, \alpha_2 \in (\xi(A_2))^{-}, ..., \alpha_m \in (\xi(A_m))^{-}.
\]

\[
(\phi)_{\xi} \Rightarrow \Sigma \gamma \in (\xi(P(B_1 \times B_2 \times ... \times B_n)))^{-}.
\]

\[
(\beta_1, \beta_2, ..., \beta_n) \in (\xi(P(B_1 \times B_2 \times ... \times B_n))) \Rightarrow [\psi]_{\xi}(\xi(\alpha_1), ..., \xi(\alpha_m), \xi(\beta_1), ..., \xi(\beta_n)).
\]

To achieve the above equality, we used the interpretation of the cartesian product constructor in type theory (see subsection 2.4). To continue the translation of the schema \( N_{Op\_Schema} \), we should use the interpretation of the decidable power set constructor in type theory given in subsection 2.4. We present this interpretation again here:

1. \( (\xi(PX))^{-} \cong (\xi(X))^{-} \Rightarrow \text{Bool} \)
2. \( (\xi(PX))I f \cong \sup(\Sigma \beta \in (\xi(X))^{-} \cdot f \beta = \text{true}, \lambda u \cdot (\xi(X))\text{fst}(u)) \)
To make the remainder of the proof easier, we use a simple interpretation of the equality 2 above: in type theory, we can consider $\Sigma x \in A.B$ as a subset $\{x \in A | B(x)\}$, the set of all elements $a$ in $A$ for which $B(a)$ holds. This subset is the same as the set $\Sigma x \in A.B$ except that the second component (the proof object) of each element of $\Sigma x \in A.B$ has been removed [55]. In this way, for $v \in (\xi(X))$, the proposition $v \in (\xi(PX))$ can be replaced by $fv \mapsto true$. Thus, in the current type theoretical interpretation of $\text{N.OP.Schema}$, we can replace

$$(\beta_1, \beta_2, ..., \beta_n) \in (\xi(P(B_1 \times B_2 \times ... \times B_n))) \gamma$$

by

$$\gamma(\beta_1, \beta_2, ..., \beta_n) \mapsto true.$$ 

Now if we use the simplified interpretation, and then replace

$$(\xi((B_1 \times B_2 \times ... \times B_n))) \gamma \mapsto \text{Bool},$$

and finally use the interpretation of the cartesian product constructor, the following equality holds:

$$[N,\text{OP.Schema}]_\xi = \Pi a_1 \in (\xi(A_1))^- \cdot a_2 \in (\xi(A_2))^- \cdot \cdot \cdot a_m \in (\xi(A_m))^- \cdot

(\xi((B_1 \times B_2 \times ... \times B_n)) \gamma \mapsto \text{Bool})$$

We can now derive a program from a correctness proof of the above specification. An initial part of such a proof is shown in Fig. 6: The resulting program is:

We can now derive a program from a correctness proof of the above specification. An initial part of such a proof is shown in Fig. 6: The resulting program is:

We can now derive a program from a correctness proof of the above specification. An initial part of such a proof is shown in Fig. 6: The resulting program is:
of extracting \( \text{prog} \) (see the proof tree in Fig. 6), for each valuation \((b_1, ..., b_n)\) of output variables, we must construct two functions \(\lambda q_4 \cdot q_5\) and \(\lambda q_6 \cdot q_7\). When constructing the function \(\lambda q_4 \cdot q_5\), indeed, we try to prove that \(q_5 \in \psi'(a_1, a_m, b_1, ..., b_n)\) if \(\text{dv}(b_1, ..., b_n) = \text{true}\). By a similar reasoning, when constructing the function \(\lambda q_6 \cdot q_7\), indeed, we try to prove that if \(q_6 \in \psi'(a_1, a_m, b_1, ..., b_n)\), then \(\text{dv}(b_1, ..., b_n) = \text{true}\).

On the other hand, since we have \(\phi(X_1, ..., X_m) = \text{true}\), we can use the membership relation between bindings and schema types, stated in subsection 2.2, and say that for all \(Y_1 \in B_1, ..., Y_n \in B_n\), the predicate \(\psi(X_1, ..., X_m; Y_1, ..., Y_n)\) holds iff \([x_1 \mapsto X_1, ..., x_m \mapsto X_m, y_1 \mapsto Y_1, ..., y_n \mapsto Y_n] \in \text{OP\_Schema}\). In other words, using convention \(\psi' = [\psi][((\xi(A_1)\tilde{x}_1/x_1) ... ((\xi(B_n)\tilde{y}_1/y_1) \in \text{OP\_Schema}\) if there exists some type theoretical term \(t_2\) such that \(t_2 \in \psi'(\xi(X_1), ..., \xi(X_m), \xi(Y_1), ..., \xi(Y_n))\). Now by this result and also the conclusion of the previous paragraph, we can say that for all \(Y_1 \in B_1, ..., Y_n \in B_n\),

\[
[x_1 \mapsto X_1, ..., x_m \mapsto X_m, y_1 \mapsto Y_1, ..., y_n \mapsto Y_n] \in \text{OP\_Schema} \text{ iff } \text{dv}((\xi(Y_1), ..., \xi(Y_n))) = \text{true}.
\]

In this way, the proof of the theorem terminates.
Theorem 3.1.2 shows that the problem of implicitly specified nondeterminism could be addressed by indicating all possible outputs directly in the specification. In this way, we can obtain all possible outputs for a given input. However, it is not hard to see that in comparison to the initial operation schema, namely, $OP\_Schema$, the new one, namely, $N\_OP\_Schema$, has a longer and more complicated structure. Also, when we write specifications such as $N\_OP\_Schema$, we must take care that our method to specify nondeterminism is suitable at the same time that we describe the functional properties of the system. Both of these issues become more serious when we need to specify modalities of nondeterminism.

Having some dedicated notations for specifying nondeterminism explicitly in our specifications not only preserves the effects of the initially specified nondeterminism in finally constructed programs, but also helps the writer of a specification to concentrate on the functional properties of the problem at the same that s/he brings nondeterminism into her/his specification. This specially facilitates specifying modalities of nondeterminism. In subsection 3.2, we introduce a simple nondeterministic construct by which one can write explicitly nondeterministic specifications in CZ. Our initial approach will be extended in subsection 3.3 to cover modalities of nondeterminism. We give a semantics for all of the new nondeterministic constructs in CZ itself; hence to produce nondeterministic programs from specifications which involve new constructs, one can still use the method of program development introduced in section 2.

3.2. Specifying nondeterminism in CZ. We use the notion of multi-schema as a tool to specify nondeterminism explicitly in CZ. A multi-schema is a version of an operation schema that may include nondeterministic, after state (or output) variables. The word may used in the definition of multi-schemas shows that we will consider ordinary operation schemas as special cases of multi-schemas involving no nondeterministic variable. The notion of nondeterministic variables has been adopted from [8] in which a nondeterministic propositional logic has been developed by introducing nondeterministic, propositional variables. In the proposed logic, an ordinary variable can evaluate to only one of the values true and false at each time whereas a nondeterministic variable can denote both values true and false simultaneously. In general, we can consider a nondeterministic variable as a variable of a type which can be evaluated to a set of values of that type.

Therefore, using nondeterministic variables, the writer of a specification can specify situations where some variables have nondeterministic nature and are allowed to be evaluated to more than one value. The following nondeterministic construct shows how we can declare a nondeterministic variable in CZ (The symbol & comes from [8]):

$$\text{ndvar} \in \& \text{Type}$$

The above declaration introduces $\text{ndvar}$ as a nondeterministic variable of the type $\text{Type}$. From the type checking point of view, nondeterministic variables are the same as ordinary ones. To emphasize this similarity, we add the following rule, called Nondeterminism, to the type inference rules of CZ\textsuperscript{17}:

\textsuperscript{17}Since this rule is a type checking rule and not a membership definition, we use : instead of $\in$. 

\[ \text{ndvar} \in \& \text{Type} \]
By the above rule, we consider the type of a nondeterministic variable as same as the type of the corresponding ordinary variable. This allows us to use these variables in specifications for indicating nondeterminism explicitly without changing the structure of the specification predicates. However, since a nondeterministic variable can be evaluated to a set of values, it semantically belongs to the power set of the basic type. Now, it would seem that we can interpret the notion of multi-schema in CZ by only replacing instances of \&Type by instances of PType. However, using nondeterministic variables in the declaration part of a schema may affect other variables and also the predicate part of the schema. Therefore, when replacing nondeterministic variables by set-valued ones, we must propagate required changes to related variables and the predicate part.

Now we are ready to give the interpretation of multi-schemas in CZ. At first, let the following schema be the general form of multi-schemas:

\[
\text{\textit{M\textunderscore Schema}} \equiv [x_1 \in A_1, x_2 \in A_2, ..., x_m \in A_m, y_1 \in B_1, y_2 \in B_2, ..., y_d \in B_d, y_{d+1} \in \&B_{d+1}, y_{d+2} \in \&B_{d+2}, ..., y_n \in \&B_n \mid \phi, \psi]
\]

The structure of \textit{M\textunderscore Schema} is similar to the structure of \textit{OP\textunderscore Schema} given earlier in this section as the general form of operation schemas, but unlike \textit{OP\textunderscore Schema}, the schema \textit{M\textunderscore Schema} involves some nondeterministic, after state (or output) variables, i.e. \(y_{d+1}, y_{d+2}, ..., y_n\). Notice that as a special case, all the variables of \textit{M\textunderscore Schema} may be ordinary. In other words, as it has been stated at the beginning of this subsection, we consider ordinary operation schemas as special cases of multi-schemas involving no nondeterministic variable. Now, let \([\cdot]^D\) be a function that interprets multi-schemas in CZ. The next definition shows how the function \([\cdot]^D\) maps the multi-schema \textit{M\textunderscore Schema} above to an ordinary operation schema of CZ.

**Definition 3.2.1.** The interpretation of the multi-schema \textit{M\textunderscore Schema} in CZ is as follows:

if none of the variables of \textit{M\textunderscore Schema} are nondeterministic, then \([\textit{M\textunderscore Schema}]^D = \textit{M\textunderscore Schema};

otherwise, \([\textit{M\textunderscore Schema}]^D \equiv [x_1 \in A_1, x_2 \in A_2, ..., x_m \in A_m, \textit{dvar} \in P(B_1 \times B_2 \times ... \times B_n) \mid \phi,

\forall(y_1, y_2, ..., y_n) \in (B_1 \times B_2 \times ... \times B_n) \cdot (y_1, y_2, ..., y_n) \in \textit{dvar} \Leftrightarrow \psi]
\]

The function \([\cdot]^D\) behaves as an identity function when applied to a multi-schema involving no nondeterministic variable. Otherwise, it promotes the combination of after state and output variables, either nondeterministic or ordinary, to a set (\textit{dvar}) of all possible combinations of these variables which satisfy the postcondition of the schema. We have combined all previous after state and output variables in a new variable using the cartesian product of their types. The reason is the same as the one given earlier in this section for the schema \textit{N\textunderscore OP\textunderscore Schema}: if we promote each after state and output variable to a separate set-valued variable, the relationship between valuations that make the schema predicates true will disappear. Combining after state and output variables in a new variable preserves the relationship between these variables after the interpretation. In subsection 3.3.3, we will show that this leads to the singular interpretation of nondeterminism. To obtain the plural interpretation of nondeterminism, we will change the current interpretation.
Notice that the interpretation of the multi-schema $M_{\text{Schema}}$ in CZ is equivalent to the operation schema $N_{\text{OP_Schema}}$ given earlier in this section. Therefore, according to theorem 3.1.2, using the notion of multi-schemas and then applying the function $[ ]^D$ to interpret multi-schemas in CZ guarantees that the final program will implement all possible behaviors allowed by the initial specification. Moreover, using the notion of multi-schemas to specify nondeterministic situations instead of specifying all possible behaviors directly (as we did by the schema $N_{\text{OP_Schema}}$) results in shorter and more concrete specifications which are easier to read and write. Also, in this way, nondeterminism can come into our specifications without changing their main structures and without decreasing our focus on describing the functional properties of the system. The above mentioned benefits will seem to be more effective in the following subsections when we will extend our toolkit to specify modalities of nondeterminism and revise the operations of the schema calculus to consider the new nondeterministic constructs.

Using the notion of multi-schemas, we will rewrite our specification of the sorting problem in example 10. We will also back to the specification of the concurrent database management system in example 16 after modelling modalities of nondeterminism in CZ and introducing a new set of the schema calculus operations. Nevertheless, in the next example, we use our toolkit to specify a small nondeterministic problem because its specification results in a simple program allowing us to easily probe the effects of the explicitly specified nondeterminism on the final program. We also construct and compare the programs resulting from the bounded and unbounded interpretations of the nondeterminism involved in the specification of this problem.

**Example 8.** The following multi-schema specifies a program which produces a natural number that is less than or equal to its input. $n?$ and $sc!$ are the input and output of the program, respectively.\footnote{Although this program seems to be very simple and far from applications in the real world, it can simulate many of the situations in which we need to select an alternative from a given set of possible choices. For example, in a multi programming environment, two or more entities (programs, processes, threads, or expressions) may want to use a common resource (e.g., a lock, an address space, a communication network, or a shared variable) at the same time [9, 56, 61]. To resolve this competition, one of the entities must be selected nondeterministically. This situation can be implemented by the current program where $sc$ plays the role of the identifier of the entity that must be selected from $n+1$ given entities.}

\[ M_{\text{GetLE}} \equiv [n? \in N, sc! \in \&N \mid \text{true}, sc! \leq n?] \]

To extract a program from $M_{\text{GetLE}}$, we first use the function $[ ]^D$ to transform the multi-schema $M_{\text{GetLE}}$ into an ordinary schema in CZ:

\[ [M_{\text{GetLE}}]^D \equiv [n? \in N, dsc \in PN \mid \text{true}, \forall sc! \in N \cdot sc! \in dsc \Leftrightarrow sc! \leq n?] \]

We have derived a program from a correctness proof of the above specification in the Appendix A. The resulting program is:

\[
\text{getLE} = \lambda \alpha \cdot \text{natrec}(\alpha, (\lambda c. \text{if } c = 0 \text{ then true else false, p}), \alpha \times \lambda y_1.(\lambda c. \text{if } c = s(x) \text{ then true else if } c < s(x) \text{ then } \text{fst}(y_1)(c) \text{ else false, q}),)
\]

where $\alpha$ is the counterpart of the variable $n?$ in the type theoretical equivalent of $[M_{\text{GetLE}}]^D$ (see the Appendix A). $p$ and $q$ are the proof objects of

\[
\lambda c. \text{if } c = 0 \text{ then true else false and}
\lambda c. \text{if } c = s(x) \text{ then true else if } c < s(x) \text{ then } \text{fst}(y_1)(c) \text{ else false,}
\]

respectively. $\text{natrec}$ is the symbol of recursion on natural numbers defined in subsection 2.4. We present this definition again here:
\[ \text{natrec}(0, e, f) = e \]
\[ \text{natrec}(s(u), e, f) = f(u, \text{natrec}(u, e, f)) \]
where \( s(u) \) is the successor of \( u \).

By the above discussion, we can conclude that for each \( \alpha \in \mathbb{N} \), the program \text{getLE} results in a boolean-valued function:

\[
\text{getLE}(0) = \lambda c. \text{if } c = 0 \text{ then true else false}
\]
\[
\text{getLE}(s(x)) = \lambda c. \text{if } c = s(x) \text{ then true else if } c < s(x) \text{ then getLE}(x)(c) \text{ else false}
\]
\[
= \lambda c. \text{if } c < s(x) \text{ then true else false}
\]

\( \text{getLE}(0) \) is a boolean-valued function whose output is true iff it is applied to 0. Also, for \( \alpha > 0 \), \( \text{getLE}(\alpha) \) is a boolean-valued function whose output is true iff it is applied to a value equal to or less than \( \alpha \). Thus, for each \( \alpha \in \mathbb{N} \), the program \text{getLE} produces a boolean-valued function whose result is true if and only if it is applied to one of the possible outputs for \( \alpha \). In this way, we can obtain all possible outputs for each input. Of course, to gain the outputs, another process is required in which the resulting boolean-valued function should be applied to all elements of its domain.

As it can be realized from example 8 and the proof of theorem 3.1.2, when we interpret a multi-schema \( M_{\text{Schema}} \) by the function \( [\cdot ]^D \), the resulting specification yields a program which produces a boolean-valued function for each input. This is due to the fact that when we translate CZ specifications into type theoretical ones, instances of the power set constructor, used in \( [M_{\text{Schema}}]^D \), are replaced by instances of the type function space with the range \( \text{Bool} \); elements of such instances are boolean-valued functions. By such functions in place, to gain all possible outputs for each input, another process is required in which the boolean-valued functions should be applied to all elements of their domains. In the following paragraph, we show that for some of the multi-schemas that will be introduced later, we can extract simpler programs; for each input, these programs produce a single sequence, consisting of all possible outputs, instead of a boolean-valued function.

In the process of program development, it is recommended to replace instances of the power set constructor by instances of the finite power set constructor, and then refine these new constructs to finite sequences since developers concern only those elements of power sets which are finite [38, 57]. Using this idea, we propose to use the interpretation function \( [\cdot ]^D \) only for those multi-schemas containing at least one nondeterministic variable whose set of possible values is infinite. If all the nondeterministic variables of a schema denote finite sets of values, we can replace instances of the power set constructor by instances of the type seq (see the type system of CZ in subsection 2.2.1). By definition, if the set of possible values of a nondeterministic variable is infinite, then the resulting nondeterminism is called unbounded; otherwise, it is called bounded.

Now, let \( [\cdot ]^{BD} \) be a function that interprets multi-schemas with bounded nondeterminism in CZ. The next definition shows how this function maps the multi-schema \( M_{\text{Schema}} \), defined earlier in this subsection as the general form of multi-schemas, to an ordinary operation schema of CZ, provided that \( M_{\text{Schema}} \) involves bounded nondeterminism.

\textbf{Definition 3.2.2.} Suppose that all the nondeterministic variables of the multi-schema \( M_{\text{Schema}} \) have bounded nondeterminism. The interpretation of \( M_{\text{Schema}} \) in CZ is as follows:
if none of the variables of \(M_{\text{Schema}}\) are nondeterministic, then \([M_{\text{Schema}}]^{BD} = M_{\text{Schema}}\):

- otherwise, \([M_{\text{Schema}}]^{BD} \cong [x_1 \in A_1, x_2 \in A_2, \ldots, x_m \in A_m, \text{dvar} \in \text{seq}(B_1 \times B_2 \times \ldots \times B_n)] \psi\),\n
\[
\forall (y_1, y_2, \ldots, y_n) \in (B_1 \times B_2 \times \ldots \times B_n), (y_1, y_2, \ldots, y_n) \in \text{dvar} \Leftrightarrow \psi
\]

Like the function \([\cdots]^D\), \([\cdots]^{BD}\) behaves as an identity function when applied to a multi-schema having no nondeterministic variable.

Theorem 3.2.3 shows that using the function \([\cdots]^{BD}\) for interpreting the multi-schema \(M_{\text{Schema}}\), which involves bounded nondeterminism, and then applying our program development method, we can extract a program which preserves the initial nondeterminism of \(M_{\text{Schema}}\). This theorem also demonstrates that using \([\cdots]^{BD}\), instead of \([\cdots]^D\), will lead to simpler programs.

**Theorem 3.2.3.** Suppose that all the nondeterministic variables of the multi-schema \(M_{\text{Schema}}\) involve bounded nondeterminism. Let prog be the program that is constructed after applying the function \(\xi\) to the schema \([M_{\text{Schema}}]^{BD}\) and then proving the correctness of the resulting type theoretical specification. For all \(X_1 \in A_1, \ldots, X_m \in A_m\), if \(\phi(X_1, \ldots, X_m) = \text{true}\), then prog, when applied to \(\xi(X_1), \ldots, \xi(X_m)\), produces a sequence \(dv : \text{List}(\xi(B_1) \otimes \ldots \otimes \xi(B_n))\) such that for all \(Y_1 \in B_1, \ldots, Y_n \in B_n\),

\[
(\xi(Y_1), \ldots, \xi(Y_n)) \in dv \iff [x_1 \mapsto X_1, \ldots, x_m \mapsto X_m, y_1 \mapsto Y_1, \ldots, y_n \mapsto Y_n] \in [M_{\text{Schema}}]/[\&],
\]

where \([M_{\text{Schema}}]/[\&]\) denotes the result of removing all occurrences of \(\&\) in \(M_{\text{Schema}}\) or rewriting \(M_{\text{Schema}}\) as an ordinary operation schema.

**Proof.** We can use an approach similar to the one used for proving theorem 3.1.2. □

Now we present the simple example 8 again here. However, for interpreting the multi-schema \(M_{\text{GetLE}}\), we use the function \([\cdots]^{BD}\) instead of \([\cdots]^D\).

**Example 9.** Consider the multi-schema \(M_{\text{GetLE}}\) given in example 8. Since for each \(n\), the set of possible values of \(sc!\) is finite, we can use the function \([\cdots]^{BD}\) to interpret \(M_{\text{GetLE}}\):

\[
[M_{\text{GetLE}}]^{BD} \cong [n \in N, dsc \in \text{seq}N | \text{true}, \forall sc! \in N \cdot \text{sc}\! \in dsc \Leftrightarrow \text{sc}! \leq n?]
\]

In the Appendix B, we have extracted the following program from the above specification:

\[
\text{getLE} = \lambda \alpha \cdot \text{natrec}(\alpha, (\langle 0 \rangle, p), \lambda x.\lambda g_1.\langle s(x) \rangle \sim \text{fst}(g_1), q)),
\]

Recall that \(\sim\) is the concatenation operator on sequences. By the definition of \(\text{natrec}\), we can conclude that for each \(\alpha \in N\), the program \(\text{getLE}\) results in a sequence of natural numbers:

\[
\text{getLE}(0) = \langle 0 \rangle
\]

\[
\text{getLE}(s(x)) = s(x) \sim \text{getLE}(x)
\]

For each \(\alpha \in N\), \(\text{getLE}\) produces the sequence \(\langle 0, 1, \ldots, \alpha \rangle\) containing all possible outputs for \(\alpha\).

Both the programs extracted from \(M_{\text{GetLE}}\) in examples 3.2 and 3.3, provide us with all possible outputs. However, the former produces a boolean-valued function for each input whereas the latter results in a single sequence consisting of all possible outputs.
In example 6, we extracted a program from the CZ specification of the sorting problem. Although the initial specification is nondeterministic, the resulting program has a deterministic behavior and for each input sequence of ordered pairs, produces one and only one permutation in which the elements are in an increasing order in terms of their second components. In the next example, we bring nondeterminism explicitly into the specification of the sorting problem and construct a new program which produces all sorted permutations of the input sequence.

**Example 10.** We replace the schema Sort, given in example 3, by a multi-schema M\_Sort in which the output variable out! is nondeterministic:

\[
M_\text{Sort} \equiv \left[ \text{Increasing}, \text{Perm}, \text{in}! \in \text{seq}(N \times N), \text{out}! \in \text{&seq}(N \times N) \mid \\
\text{true, } \text{out}! \in \text{increasing} \land (\text{in}!, \text{out}!) \in \text{perm} \right]
\]

Notice that the only difference between two schemas Sort and M\_Sort is that the output variable out! is nondeterministic in the latter. Since the set of sorted permutations of a sequence is finite, we can use the function \([BD\,]^{M_\text{Sort}}\) to translate the multi-schema M\_Sort into an ordinary schema in CZ:

\[
[M_\text{Sort}]^{BD} \equiv \left[ \text{Increasing}, \text{Perm}, \text{in}! \in \text{seq}(N \times N), \text{dout} \in \text{seq}(N \times N) \mid \\
\text{true, } \forall \text{out}! \in \text{seq}(N \times N) \cdot \text{out}! \in \text{dout} \Leftrightarrow (\text{out}! \in \text{increasing} \land (\text{in}!, \text{out}!) \in \text{perm}) \right]
\]

In the Appendix C, we have derived the following program from a correctness proof of the above specification:

\[
n\text{sort} = \lambda a \cdot \text{tree}(a, (\langle \rangle, r), \lambda a \cdot \lambda x \cdot \lambda h_1 \cdot (T_\text{Insert}(a, \text{fst}(h_1)), q))
\]

We first give an abstract illustration of the program. The readers interested in details of the program can see them later in this example. This program is a recursive function that for each input sequence of ordered pairs, produces all permutations in which the elements are in the increasing order in terms of their second components:

\[
n\text{sort} : \text{List}(N \otimes N) \Rightarrow \text{List}(\text{List}(N \otimes N))
\]

\[
n\text{sort}(\langle \rangle) = \langle \rangle
\]

\[
n\text{sort}(h :: t) = T_\text{Insert}(h, \text{sort}(t))
\]

When the function T\_Insert applied to an ordered pair n of two natural numbers and a sequence r of sorted sequences, it produces a sequence s that includes all sorted permutations of \(\langle n \rangle \sim t\) for all \(t \in r\) and not anything else. Formally, the following predicate holds:

\[
(\forall u \in s \cdot \exists v \in r \cdot u \text{ is a sorted permutation of } \langle n \rangle \sim t) \land \\
(\forall v \in r \cdot \forall u \in (\text{the set of sorted permutations of } \langle n \rangle \sim t) \cdot \exists u \in s \cdot u = v)
\]

When proving the correctness proof of the multi-schema M\_Sort, T\_Insert is constructed as follows:

\[
T_\text{Insert} : (N \otimes N) \otimes \text{List}(\text{List}(N \otimes N)) \Rightarrow \text{List}(\text{List}(N \otimes N))
\]

\[
T_\text{Insert}(n, (\langle \rangle)) = \langle \rangle
\]

\[
T_\text{Insert}(n, h :: t) = D_\text{Insert}(n, h) \sim T_\text{Insert}(n, t)
\]

Recall the function Insert defined in example 6: for each ordered pair n of two natural numbers and a sorted sequence s, Insert produces one and only one permutation of \(\langle n \rangle \sim s\) whose elements are in the increasing order in terms of their second components. Unlike Insert, the function D\_Insert, used in the definition of T\_Insert above, results in all permutations of \(\langle n \rangle \sim s\) whose elements are in the increasing order in terms of their second components. The definition of D\_Insert is as follows:

\[
D_\text{Insert} : (N \otimes N) \otimes \text{List}(N \otimes N) \Rightarrow \text{List}(\text{List}(N \otimes N))
\]

\[
D_\text{Insert}(n, (\langle \rangle)) = \langle \langle n \rangle \rangle
\]
In the above definition, we have used the symbol of recursion on lists defined in subsection 2.3. We give this definition as follows:

\[ D_{\text{Insert}}(n, h :: t) = \begin{cases} \text{snd}(n) < \text{snd}(h) & \text{then } (\langle n \rangle \sim (h :: t)) \\ \text{else if } \text{snd}(n) > \text{snd}(h) & \text{then } T_{\text{Concate}}(h, D_{\text{Insert}}(n, t)) \\ \text{else } (\langle n \rangle \sim (h :: t)) & \sim T_{\text{Concate}}(h, D_{\text{Insert}}(n, t)) \end{cases} \]

where for an ordered pair \( n \) of two natural numbers and a sequence \( s \) whose elements are themselves sequences, \( T_{\text{Concate}}(n, s) \) is a sequence constructed by concatenating \( n \) and elements of \( s \). The elements of the resulting sequence are themselves sequences:

\[ T_{\text{Concate}} : (N \otimes N) \otimes \text{List}(\text{List}(N \otimes N)) \Rightarrow \text{List}(N \otimes N) \]

\[ T_{\text{Concate}}(n, h :: t) = (\langle n \rangle \sim h) \sim T_{\text{Concate}}(n, t) \]

According to the above discussion, when applied to a sequence \( \alpha \), the program \( nsort \) produces a sequence consisting of all sorted permutations of \( \alpha \) whereas the program \( sort \), developed in example 6, produces one and only one sorted permutation of \( \alpha \). This is due to the fact that when proving the correctness of \([M_{\text{Sort}}]^{BD}\), we are to use functions such as \( D_{\text{Insert}} \) instead of functions such as \( \text{Insert} \). The difference between these two functions has been mentioned earlier.

A detailed description of the program \( nsort \) is as follows: \( r \) and \( q \) are the proof objects of \( \langle \rangle \) and \( T_{\text{Insert}}(a, \text{fst}(h)) \), respectively (see the Appendix C). \( \text{lrec} \) is the symbol of recursion on lists defined in subsection 2.3. We give this definition again here:

\[ \text{lrec}(\langle \rangle, e, f) = e \]
\[ \text{lrec}(h :: t, e, f) = f(h, t, \text{lrec}(t, e, f)) \]

Thus, the obtained program is a recursive function which results in \( \langle \rangle \) for the input \( \alpha = \langle \rangle \). In the recursive step, the program computes the output \( T_{\text{Insert}}(h, nsort(t)) \) for the input \( \alpha = h :: t \). The definition of \( T_{\text{Insert}} \) is as follows:

\[ T_{\text{Insert}} = \lambda \alpha \cdot \lambda s \cdot \text{lrec}(s, \langle \rangle), \lambda h \cdot \lambda t \cdot \lambda r \cdot D_{\text{Insert}}(n, h) \sim r \]

In the above definition, we have used \( D_{\text{Insert}} \) defined as follows:

\[ D_{\text{Insert}} = \lambda \alpha \cdot \lambda s \cdot \text{lrec}(s, (\langle n \rangle), \lambda h \cdot \lambda t \cdot \lambda r \cdot \text{if } \text{snd}(n) < \text{snd}(h) \text{ then } (\langle n \rangle \sim (h :: t)) \\ \text{else if } \text{snd}(n) > \text{snd}(h) \text{ then } T_{\text{Concate}}(h, r) \\ \text{else } (\langle n \rangle \sim (h :: t)) \sim T_{\text{Concate}}(h, r) \]

In the above definition, the function \( T_{\text{Concate}} \) has been used whose definition is as follows:

\[ T_{\text{Concate}} = \lambda \alpha \cdot \lambda s \cdot \text{lrec}(s, \langle \rangle), \lambda h \cdot \lambda t \cdot \lambda r \cdot (\langle n \rangle \sim h) \sim r \]

In this subsection, we have introduced a simple approach to specify nondeterminism explicitly in CZ. We have shown that using this approach, we can preserve the effects of the initially specified nondeterminism in final programs. This approach distinguishes between bounded and unbounded nondeterminism. In the next subsection, we will extend the current approach to cover loose, strict, erratic, angelic, demonic, singular, and plural nondeterminism.

3.3. **Modalities of nondeterminism.** In addition to the bounded and unbounded nondeterminism, some other modalities of nondeterminism have been so far mentioned in the literature (for example, see \([2, 13, 33, 41, 56, 60, 71]\)). We first give a brief overview of these modalities of nondeterminism:

- **Erratic, angelic, and demonic** nondeterminism: an erratic choice is made in a random manner. In other words, an erratic choice corresponds to selecting an alternative regardless of its effect upon establishing a certain postcondition, such as termination; angelic choices are made in a way that the postcondition is established (if possible), while demonic nondeterminism is
resolved so as to avoid establishing the postcondition [11, 13]. For example, a program which implements a telephone exchange examines incoming lines and chooses one that has a defined input (a call coming in) and processes that call. The choice here is angelic that means undefined-avoiding. On the other hand, a flight control system may use demonic choice: in a plane, two computers may calculate the same flight instruction. When both are working fine, one of their outputs is chosen. If one of the computers stops working (making its output undefined), both computers shut themselves down (making their combined output undefined) and the pilot has to take over.

The combination of demonic and angelic nondeterminism can model interactions in game-like situations, where a number of agents try to achieve (potentially conflicting) goals by taking turns in making choices. The agent or coalition whose goal we are focusing on is modelled as the angel and the associated nondeterminism is angelic; the remaining agents are collectively considered as the demon [13, 41]. The two types of nondeterminism are also useful when modelling system-user situations, with the angel modelling the user side and the demon the system side (because the demonic choice reflects our uncertainty as to how the execution will proceed). We (the user) do not know which one of the alternatives will actually be chosen by the system. Thus, we need to guard against either choice if we want to be certain to achieve some specific final state with our program. Of course, system-user situations can be considered as special cases of game-like situations since the execution of a program can be seen as a game between the user (angel) and the system (demon).

- **Strict and loose nondeterminism:** in the strict interpretation of nondeterminism, we are interested in extracting all possible forms of a specified nondeterministic behavior. Such an interpretation is required to compare different implementations of an abstract specification, to preserve the specified nondeterminism in final nondeterministic programs such as concurrent/parallel systems or two player games, or to verify final programs against all possible behaviors. In the loose interpretation, however, the existence of more than one behavior is regarded in the specification stage, but the final implementation is itself deterministic. For example, in the sorting problem, we expect one and only one sorted permutation of the input sequence when the final program executes. In this case, we have a nondeterministic specification whereas the final program is itself deterministic. This coincides with the loose interpretation of nondeterminism. Of course, in some situations, we may be interested in comparing all sorted permutations. In these situations, we must use the strict interpretation of nondeterminism, but to guarantee the deterministic behavior of the final execution, we must also use the singular semantics of nondeterminism (This modality of nondeterminism will be defined later).

- **Bounded and unbounded nondeterminism:** if the set of possible choices is an infinite set, then the resulting nondeterminism is unbounded; otherwise, it is bounded.
Singular and plural nondeterminism: singular and plural nondeterminism coincide with call-by-value and call-by-name semantics in programming languages, respectively [71]. In the plural semantics, decisions on a same nondeterministic construct are made independently. Thus, they may lead to different choices. However, in the singular semantics, same decision is always made in different places with the same nondeterministic construct [70]. Fairness issues can also be discussed in the plural semantics [56]. For example, we would expect the drinks machine (see subsection 1.1) to be fair in the sense that pressing a button for *Either* does not always pour a cup of tea. A lottery machine that chooses who wins the prize should be probabilistically fair and give every ticket the same chance of winning. An important application of the fair nondeterminism is to implement concurrent systems, where the operations of a number of processes are interleaved, and thus must be scheduled by fair scheduling algorithms. For example, consider the CZ specification of a concurrent database management system (see example 4). If the nondeterministic schema *AcquireLock* will not yield a sub-program that makes its choices fairly, then the operations of some transactions may be discarded forever.

It seems appropriate to clarify the difference between the loose nondeterminism and the singular one: in the loose nondeterminism, we are interested in extracting one and only one implementation or behavior among those specified in the initial specification; no alternative has priority over the others; indeed, we are satisfied with any of the alternatives. On the other hand, by the singular semantics, we may need to produce and then compare all possible outputs (the strict nondeterminism instead of the loose one); however, when we select the preferred alternative, we fix it for next selections. The reason for producing all possible outputs is that the choice is made in terms of implementation considerations or the programmer interests; such criteria cannot be evaluated in the specification phase, and thus must be delegated to the implementation phase. Therefore, in contrast to the loose semantics, in the singular one, there may be certain priorities, but they can be evaluated in the implementation phase; in this phase, however, any of the alternatives taking priority over the others is fixed for next selections.

In the previous subsection, to specify nondeterministic operations in CZ, we introduced the notion of multi-schema as an operation schema that may involve nondeterministic variables. We used the operator & to declare nondeterministic variables. To consider modalities of nondeterminism, we replace this operator by a parametric one $\&_{(bu, gp, sl, cad)}$. Like &, the operator $\&_{(bu, gp, sl, cad)}$ promotes the ordinary (single-valued) nature of the declared variable to its nondeterministic (set-valued) nature, but the subscript of $\&_{(bu, gp, sl, cad)}$ is used to distinguish different modalities of nondeterminism.

The first component of $(bu, gp, sl, cad)$, $bu$, denotes bounded or unbounded nondeterminism and is instantiated with either $b$ (bounded) or $u$ (unbounded). The second component, $gp$, indicates singular or plural nondeterminism and is evaluated to either $g$ (singular) or $p$ (plural). The third component, $sl$, corresponds to the strict or the loose semantics of nondeterminism; one of the values $s$ (strict)
or l (loose) can be assigned to sl. Finally, the fourth component, ead, is used to distinguish erratic, angelic, and demonic nondeterminism, and one of the values e (erratic), a (angelic), or d (demonic) is used in place of it. We can consider the bounded interpretation of nondeterminism as a special case of the unbounded one. Nevertheless, as we have shown in the previous subsection, if the set of possible choices is finite, it is more appropriate to specify bounded nondeterminism instead of the unbounded one since it results in simpler programs.

Before interpreting \&_{(bu, gp, sl, ead)}\), we replace the nondeterministic construct
\ndvar \in \text{Type},
which was presented in the previous subsection to declare a nondeterministic variable, by the following one:
\ndvar \in \&_{(bu, gp, sl, ead)}\text{Type},
where bu, gp, sl, and ead can be instantiated with any of the elements of the sets \{b, u\}, \{g, p\}, \{s, l\}, and \{e, a, d\}, respectively. Also, we replace the type inference rule
\ndvar : \&\text{Type}
\ndvar : \text{Type}
\nby the following rule:
\ndvar : \&_{(bu, gp, sl, ead)}\text{Type}
\ndvar : \text{Type}

Now we proceed by giving a semantics for the parametric operator \&_{(bu, gp, sl, ead)}\) in CZ. For convenience, we assume that all the nondeterministic variables, declared in a multi-schema, denote the same interpretation of nondeterminism. For example, if a nondeterministic variable involves angelic nondeterminism, others cannot involve demonic or erratic nondeterminism, or if a nondeterministic variable denotes strict nondeterminism, others cannot denote loose nondeterminism. To interpret the parametric operator \&_{(bu, gp, sl, ead)}\), we first give the semantics for different values of the component ead and replace this component by i that means an interpreted component (Thus, we must add i to all the sets \{b, u\}, \{g, p\}, \{s, l\}, and \{e, a, d\} introduced in the previous paragraph when we wanted to define a type inference rule for \&_{(bu, gp, sl, ead)}\)). Then any of the possible values of sl is interpreted, and thus sl is replaced by i. Finally, we interpret both the components bu and gp simultaneously, using the mapping functions \[D\] and \[BD\], defined in subsection 3.2, and two new functions \[PD\] and \[PBD\] which will be introduced in subsection 3.3.3.

Using the above mentioned, step by step approach, in the following subsections, we interpret the operator \&_{(bu, gp, sl, ead)}\) in CZ.

3.3.1. Interpretation of erratic, angelic, and demonic nondeterminism. As we have stated earlier, if a choice is to be made in favor of establishing a postcondition q, we should have angelic choice semantics. Demonic choice, however, is resolved so as to avoid establishing the postcondition. On the other hand, when we use the combination of angelic and demonic nondeterminism for modelling interactions in game-like situations, the postcondition q is established if the angel can make its choices in a way that q is reached, regardless of how the demon makes its choices [13]. In both the above mentioned applications (either using only one of the angelic and demonic nondeterminism or using the combination of them), we can consider
angelic and demonic choices as the best and the worst choices with respect to a postcondition q, respectively.

To implement the above idea, we assume that some score is associated to each alternative. By an angelic (demonic) choice, we choose one of the alternatives having the highest (lowest) score. By an erratic choice, however, we choose one of the alternatives regardless of the scores. Example 11 describes a two-player game and shows how scores can be assigned to alternatives.

**Example 11.** In the Nim game [13], two players take turns in removing one or two matches from a pile (of x matches). The player to remove the last match loses the game. The first player is considered as the angel and the other one as the demon. Provided that initially \( x \mod 3 \neq 1 \), the angel can win the game by making sure that \( x \mod 3 = 1 \) after each of its turns. Therefore, in each turn of the angel, if \( x \mod 3 = 0 \) (\( x \mod 3 = 2 \)), it must remove two matches (one match) from the pile.

Now we can define a score function \( sco \) over the domain \( \{1, 2\} \), which is the set of possible choices of the angel:

\[
sco(n) = |n - (x \mod 3)| \quad n \in \{1, 2\}
\]

if \( x \mod 3 = 0 \), then \( sco(2) \geq sco(1) \); in this case, the alternative 2 (removing two matches) is the preferred choice. In the other case (\( x \mod 3 = 2 \)), \( sco(1) \geq sco(2) \), and thus it is more appropriate to remove one match from the pile.

Example 11 shows how we can define scores in game-like situations where we use the combination of angelic and demonic nondeterminism. A similar approach can be adopted when we use only one of the angelic and demonic nondeterminism for achieving or avoiding a certain postcondition q (such as termination or non-abortion): we can assign the score one to alternatives satisfying q and the score zero to alternatives that do not satisfy q. For example, in a program implementing a telephone exchange, we assign the score one to lines that have an incoming call and the score zero to other lines. Similarly, in a flight control system, we assign the score zero to the undefined outputs and the score one to the others. This solution can be used for a well known application of angelic and demonic nondeterminism where the non-abortion of programs is considered as the criterion for distinguishing angelic and demonic choices: in such situations, we can treat the abortion of a program as an ordinary output and assign the score zero to this output and the score one to others. For example, consider the following specification:

\[
S_1 \equiv [a? \in N, b! \in N \mid a? > 0 \land b! = a? - 1 ]
\]

\[
S_2 \equiv [a? \in N, b! \in N \mid a? > 1 \land b! = a? - 2 ]
\]

\[
S \equiv S_1 \lor S_2
\]

For the input \( a? = 1 \), the program satisfying \( S_1 \) returns 0 while the program satisfying \( S_2 \) aborts. By the angelic nondeterminism, the program satisfying \( S \) must behave as the program satisfying \( S_1 \). By the demonic nondeterminism, however, this program must behave as the program satisfying \( S_2 \). To implement such a behavior, we must first rewrite the above specification by a new one in which the abortion of programs is considered as an ordinary output:

\[
S_1 \equiv [a? \in N, b! \in N \mid \text{true}, (a? > 0 \land b! = a? - 1) \lor (a? = 0 \land b! = \text{abort})]
\]

\[
S_2 \equiv [a? \in N, b! \in N \mid \text{true}, (a? > 1 \land b! = a? - 2) \lor (a? \leq 1 \land b! = \text{abort})]
\]

\[
S \equiv S_1 \lor S_2 \equiv [a? \in N, b! \in N \mid \text{true}, (a? > 0 \land b! = a? - 1) \lor (a? = 0 \land b! = \text{abort}) \lor (a? > 1 \land b! = a? - 2) \lor (a? \leq 1 \land b! = \text{abort})]
\]

Now we assign the score zero to the value abort and the score one to other values. Consider the behavior of the program that satisfies \( S \). When this program is applied
to the input \( a_2^? = 1 \), it can select one of the alternatives 0 with the score one and abort with the score zero; by the angelic nondeterminism, it selects the former, and by the demonic nondeterminism, it selects the latter. This result is similar to that of many literature (See [18], [33], [53], [56], [60], [64], and [72], for example).

Now we interpret the angelic and demonic nondeterminism according to the above proposed solution. Suppose that the general form of multi-schemas, defined in subsection 3.2, is modified as follows:

\[
M_{\text{Schema}} \equiv [x_1 \in A_1, x_2 \in A_2, \ldots, x_m \in A_m, y_1 \in B_1, y_2 \in B_2, \ldots, y_d \in B_d, y_{d+1} \in \&_{(bu, gp, sl, cad)} B_{d+1}, y_{d+2} \in \&_{(bu, gp, sl, cad)} B_{d+2}, \ldots, y_n \in \&_{(bu, gp, sl, cad)} B_n | \phi, \psi],
\]

where some of the after state and output variables of \( M_{\text{Schema}} \), i.e. \( y_{d+1}, y_{d+2}, \ldots, y_n \), are nondeterministic. As it has been mentioned earlier, we assume that all these nondeterministic variables denote the same interpretation of nondeterminism. We define a function \( [\cdot]^{\text{cad}} \) which maps the multi-schema \( M_{\text{Schema}} \) to a multi-schema involving some variables declared by \( \&_{(bu, gp, sl, i)} \):

**Definition 3.3.1.** If \( M_{\text{Schema}} \) involves no nondeterministic variable, then \( [M_{\text{Schema}}]^{\text{cad}} = M_{\text{Schema}} \); otherwise, if \( M_{\text{Schema}} \) involves some nondeterministic variables declared by:

1. \( \&_{(bu, gp, sl, a)} \), then \( [M_{\text{Schema}}]^{\text{cad}} \) is defined as follows:

\[
[M_{\text{Schema}}]^{\text{cad}} \equiv [x_1 \in A_1, x_2 \in A_2, \ldots, x_m \in A_m,
\]

\[
ay_1 \in \&_{(bu, gp, sl, a)} B_1, ay_2 \in \&_{(bu, gp, sl, a)} B_2, \ldots, ay_n \in \&_{(bu, gp, sl, a)} B_n | \phi,
\]

\[
\psi[ay_1/y_1][ay_2/y_2]\cdots[ay_n/y_n] \land \forall(y_1, y_2, \ldots, y_n) \in (B_1 \times B_2 \times \ldots \times B_n) : \psi \Rightarrow \text{scn}(y_1, y_2, \ldots, y_n) \leq \text{scn}(ay_1, ay_2, \ldots, ay_n)
\]

2. \( \&_{(bu, gp, sl, d)} \), then \( [M_{\text{Schema}}]^{\text{cad}} \) is defined as follows:

\[
[M_{\text{Schema}}]^{\text{cad}} \equiv [x_1 \in A_1, x_2 \in A_2, \ldots, x_m \in A_m,
\]

\[
dy_1 \in \&_{(bu, gp, sl, d)} B_1, dy_2 \in \&_{(bu, gp, sl, d)} B_2, \ldots, dy_n \in \&_{(bu, gp, sl, d)} B_n | \phi,
\]

\[
\psi[dy_1/y_1][dy_2/y_2]\cdots[dy_n/y_n] \land \forall(y_1, y_2, \ldots, y_n) \in (B_1 \times B_2 \times \ldots \times B_n) : \psi \Rightarrow \text{scn}(y_1, y_2, \ldots, y_n) \leq \text{scn}(dy_1, dy_2, \ldots, dy_n)
\]

3. \( \&_{(bu, gp, sl, e)} \), then we can easily map \( M_{\text{Schema}} \) to \( [M_{\text{Schema}}]^{\text{cad}} \) by replacing instances of \( \&_{(bu, gp, sl, e)} \) by instances of \( \&_{(bu, gp, sl, i)} \):

\[
[M_{\text{Schema}}]^{\text{cad}} \equiv [x_1 \in A_1, x_2 \in A_2, \ldots, x_m \in A_m, y_1 \in B_1, y_2 \in B_2, \ldots, y_d \in B_d, y_{d+1} \in \&_{(bu, gp, sl, i)} B_{d+1}, y_{d+2} \in \&_{(bu, gp, sl, i)} B_{d+2}, \ldots, y_n \in \&_{(bu, gp, sl, i)} B_n | \phi, \psi]
\]

The above interpretation is based on the fact that when making an erratic choice, we choose one of the alternatives without regarding any of their particular properties.

As before, in the above definition, we considered ordinary operation schemas as special cases of multi-schemas involving no nondeterministic variable. Therefore, the function \( [\cdot]^{\text{cad}} \) behaves as an identity function when applied to an ordinary operation schema. The function \( \text{scn} \) is assumed to return the score of its input. Thus, two combinations \( (ay_1, ay_2, \ldots, ay_n) \) and \( (dy_1, dy_2, \ldots, dy_n) \) of output variables (see definition 3.3.1) are choices having the highest and the lowest score, respectively.
Since it is possible that more than one choice have the highest or the lowest score, all variables $ay_1$, $ay_2$, ..., $ay_n$, $dy_1$, $dy_2$, ..., $dy_n$ have still the nondeterministic nature and are declared as nondeterministic variables by using the operator $\&_{\{bu, gp, sl, i\}}$.

If we assume that $sco$ has an intensional definition (for example, see the function $sco$ in example 11 that has an intensional definition), we can show the conservativeness of our approach for modelling angelic and demonic nondeterminism. Nevertheless, when using angelic and demonic nondeterminism in our specifications, we want to abstract away from details of how a particular postcondition establishing (or postcondition avoiding) strategy is implemented. In other words, in the specification phase, the writer of a specification is only interested in specifying his/her intention to establish or avoid a certain postcondition $q$ and does not want to know the way in which the postcondition is established or avoided; this issue must be delegated to the implementation phase. Thus, we can treat $sco$ as an abstract function in the specification phase and guarantee that our constructive approach for program development results in programs which select alternatives that have the highest (or the lowest) score; the scores themselves can be determined in the implementation phase. For example, a program that satisfies a specification of the telephone exchange system is guaranteed to select a line having the highest score. In the implementation phase, we assign the score one to lines that have an incoming call and the score zero to other lines. On the other hand, a program that satisfies a specification of the flight control system is guaranteed to select an output with the lowest score. In the implementation phase, we assign the score one to the defined outputs and the score zero to the undefined ones.

For the game-like situations, we offer a solution to determine the function $sco$ in the implementation phase that leads to implementing a winning strategy for the angel (when such a strategy exists). Our approach is similar to that of [13] which deals with angelic nondeterminism in a refinement calculus: by the winning strategy, we are given a precondition $p$ and a postcondition $q$ for the angelic choice (For instance, in the Nim game, $p \equiv x \mod 3 \neq 1$ and $q \equiv x \mod 3 = 1$). The angel must select one alternative from $n$ alternatives $c_1, c_2, ..., c_n$ so that the execution of the program is transformed from a state where $p$ holds to a state where $q$ holds. We show this goal as follows:

$$p[c := c_1, c := c_2, ..., c := c_n] q,$$

where $c$ is the (angelic) choice variable that can be instantiated with any of the elements of the set $\{c_1, c_2, ..., c_n\}$.

Now we introduce the following rule which enables us to derive the function $sco$:

$$\frac{r_1 \land p[c := c_1] q \quad r_2 \land p[c := c_2] q \quad \ldots \quad r_n \land p[c := c_n] q}{p[c := c_1, c := c_2, ..., c := c_n] q}$$

The predicates $r_1, r_2, ..., r_n$ describe the winning strategy for the angel by partitioning the precondition $p$ into $n$ parts depending on which of the assignments $c := c_1, c := c_2, ..., c := c_n$ can establish $q$. It can be shown that the existence of partitioning predicates $r_1, r_2, ..., r_n$ is guaranteed when the angel has a winning strategy. Discovering such predicates can be reduced to calculating weakest
preconditions. We determine \( r_i \) as \( (c := c_1)_i q \) since the other assignments need to be done only when the first one fails. By a similar reasoning, for \( 2 \leq i \leq n \), the following equality holds:

\[
r_i \equiv \neg (c := c_1)_i q \land \ldots \land \neg (c := c_{i-1})_i q \land (c := c_i)_i q
\]

Now we can define \( sco \) as follows:

For \( 1 \leq i \leq n \), if \( r_i = \text{true} \) then \( sco(c_i) = 1 \) else \( sco(c_i) = 0 \).

For instance, in example 11, for two alternatives \( c_1 = 1 \) and \( c_2 = 2 \), we have

\[
r_1 \equiv (x \mod 3 = 2) \quad \text{and} \quad r_2 \equiv (x \mod 3 = 0),
\]

respectively. In this case, \( sco \) is thus defined as follows:

\[
\text{if} \ (x \mod 3 = 2) \ \text{then} \ sco(1) = 1 \ \text{else} \ sco(1) = 0
\]

\[
\text{if} \ (x \mod 3 = 0) \ \text{then} \ sco(2) = 1 \ \text{else} \ sco(2) = 0
\]

In this subsection, we have interpreted the fourth component of the subscript of \( \land \). In the next subsection, we concentrate on the third component, i.e. \( sl \).

3.3.2. Interpretation of strict and loose nondeterminism. In definition 3.3.1, we defined the function \( \llbracket \cdot \rrbracket^{\text{ed}} \) for interpreting the fourth component of the subscript of \( \land \). After using \( \llbracket \cdot \rrbracket^{\text{ed}} \), we have a multi-schema that either involves no nondeterministic variable or has some nondeterministic variables declared by \( \land \). In this subsection, we introduce a mapping function \( \llbracket \cdot \rrbracket^s \) which interprets the third component of \( \land \). For ordinary operation schemas, this function behaves as an identity function. In other words, for a multi-schema \( M_{\text{Schema}} \) that has no nondeterministic variable, \( [M_{\text{Schema}}]^s \) is equal to \( M_{\text{Schema}} \) itself.

Now we explore the result of \( \llbracket \cdot \rrbracket^s \) when applied to a multi-schema that involves some nondeterministic variables declared by \( \land \). Consider first the case of \( sl = s \): both the mapping functions \( \llbracket \cdot \rrbracket^D \) and \( \llbracket \cdot \rrbracket^{BD} \), proposed in subsection 3.2, match the strict interpretation of nondeterminism since they yield programs which produce all possible outputs for each input (see theorems 3.1.2 and 3.2.3). We will use these functions in subsection 3.3.4 to do the last step of the interpretation of multi-schemas. Thus, if a multi-schema \( M_{\text{Schema}} \) involves a nondeterministic variable declared by \( \land \), we are to do nothing; it is sufficient to replace \( \land \) by \( \land \) in order to map \( M_{\text{Schema}} \) to \( [M_{\text{Schema}}]^s \); the last step of the interpretation will itself implement the strict semantics.

In the case of \( sl = l \), we are to do a simple task again: as shown earlier, constructed programs from implicitly nondeterministic specifications produce only a single output for each input. Therefore, the loose interpretation of nondeterminism can automatically be achieved when we do not specify nondeterminism explicitly. Thus, if a multi-schema \( M_{\text{Schema}} \) involves a nondeterministic variable declared by \( \land \), we can map \( M_{\text{Schema}} \) to \( [M_{\text{Schema}}]^s \) by rewriting \( M_{\text{Schema}} \) as an ordinary operation schema or removing all occurrences of the operator \( \land \) from \( M_{\text{Schema}} \).

The next definition formalizes the above discussion.

**Definition 3.3.2.** Let \( M_{\text{Schema}} \) be a multi-schema that involves zero or more nondeterministic variables declared by the nondeterministic construct \( \land \).

\footnote{For a statement \( S \) and a postcondition \( q \), if \( \sigma \) is a state, then the weakest precondition of \( S \), that we show as \( S \sigma \), holds in \( \sigma \) if and only if the execution of \( S \) from the state \( \sigma \) is guaranteed to terminate in a final state where \( q \) holds}
In example 8:

(1) If $M_{\text{Schema}}$ involves no nondeterministic variable, then $[M_{\text{Schema}}]^s = M_{\text{Schema}}$.

(2) If $M_{\text{Schema}}$ involves some nondeterministic variables declared by $\&_{(bu, gp, s, i)}$, then

$$[M_{\text{Schema}}]^s = M_{\text{Schema}}[\&_{(bu, gp, i, i)} / \&_{(bu, gp, s, i)}],$$

(3) If $M_{\text{Schema}}$ involves some nondeterministic variables declared by $\&_{(bu, gp, l, i)}$, then

$$[M_{\text{Schema}}]^s = M_{\text{Schema}}[\&_{(bu, gp, l, i)}],$$

where $M_{\text{Schema}}[\&_{(bu, gp, l, i)}]$ denotes the result of removing all occurrences of $\&_{(bu, gp, l, i)}$ in $M_{\text{Schema}}$.

In the next subsection, we will focus upon the second component of the subscript of $\&_{(bu, gp, s, i, cad)}$, i.e. $gp$.

3.3.3. Interpretation of singular and plural nondeterminism. We have two alternative ways to explore the semantics of the singular and plural choices in CZ specifications. At first, we give the interpretation of these modalities of nondeterminism in terms of various occurrences of a nondeterministic variable in a schema: if all occurrences of a nondeterministic variable are evaluated to a single value, we have the singular semantics; otherwise, we have the plural one. To clarify this statement, we focus on the following simple schema, an extension of the schema $M_{\text{GetLE}}$ given in example 8:

$$M_{\text{GetLE23}} \equiv [n? \in N, \, sc!, \, a!, \, b! \in \&N \mid \text{true, } sc! \leq n? \land a! = 2 * sc! \land b! = 3 * sc!]$$

The above multi-schema specifies a program which produces a natural number $sc!$ that is less than or equal to its input, i.e. $n?$, and also produces two other outputs $a!$ and $b!$ which are equal to $2*sc!$ and $3*sc!$, respectively. After interpreting $M_{\text{GetLE23}}$ by the function $[]^{BD}$ (see definition 3.2.2), the following schema is obtained:

$$[M_{\text{GetLE23}}]^{BD} \equiv [n? \in N, \, dscab \in \text{seq} (N \times N \times N) \mid \text{true, } \forall (sc, a, b) \in (N \times N \times N) : (sc, a, b) \in dscab \iff (sc \leq n? \land a = 2 * sc \land b = 3 * sc)]$$

For $n? = \text{in}$, the resulting program from the above specification produces a sequence consisting of $in + 1$ tuples $(sc, 2sc, 3sc)$, where $sc$ ranges from 0 to $in$. Notice that each tuple is related to one of the possible values of the nondeterministic variable $sc!$. Now we can consider any of these tuples as the final output: if we select the value $\text{Selsc}$ for $sc!$, we will be forced to select the values $2 \ast \text{Selsc}$ and $3 \ast \text{Selsc}$ for $a!$ and $b!$, respectively. This means that all occurrences of $sc!$ are evaluated to a single value, i.e. $\text{Selsc}$. In other words, we have implemented the singular interpretation of nondeterminism. This is due to the fact that when we interpret the notion of multi-schemas by the functions $[\cdot]^{D}$ and $[\cdot]^{BD}$, we combine all previous after state and output variables in a new variable using the cartesian product of their types. In this way, the relationship between these variables is preserved after the interpretation.

Therefore, to achieve the plural interpretation of nondeterminism, it seems sufficient to promote each nondeterministic variable to a distinct set (in unbounded nondeterminism) or sequence (in bounded nondeterminism). By this modification, the interpretation of $M_{\text{GetLE23}}$ in CZ is the following schema:

$$[n? \in N, \, dsc, \, da, \, db \in \text{seq} N \mid \text{true, } \forall sc, a, b \in N : (sc \in dsc \land a \in da \land b \in db) \iff (sc \leq n? \land a = 2 * sc \land b = 3 * sc)]$$
In the above schema, the possible values of \( sc! \), \( a! \), and \( b! \) are specified in three distinct sequences \( dsc \), \( da \), and \( db \), respectively. For example, for \( n? = 2 \), the three distinct sequences \( (0, 1, 2) \), \( (0, 2, 4) \), and \( (0, 3, 6) \) are produced as possible values of \( sc! \), \( a! \), and \( b! \), respectively. In the final program, we may select three values 0, 2, and 6 as the preferred values for \( sc! \), \( a! \), and \( b! \), respectively. In other words, the variable \( sc! \) is not evaluated to a same value in each of its occurrences in the initial schema \( (sc! = 0 \text{ in } \langle sc! \rangle, sc! = 1 \text{ in } \langle a! = 2 * sc! \rangle, \text{ and } sc! = 2 \text{ in } \langle b! = 3 * sc! \rangle) \). Such a behavior corresponds to the plural interpretation of nondeterminism. By this interpretation, the writer of the specification wants to specify \( a! \) as twice a natural number which is less than or equal to \( n? \) and \( b! \) as the multiplication of 3 and a natural number that is less than or equal to \( n? \); there is no need to have a natural number \( sc! \leq n? \) such that \( a! = 2 * sc! \) and \( b! = 3 * sc! \) simultaneously.

It seems that the above mentioned approach to interpret plural nondeterminism leads to strange consequences in some situations. For example, suppose that we have the following multi-schema:

\[
M_{GetSumBelow4} \cong [a!, b!, sum! \in &N | true, a! > 0 \land b! > 0 \land (a! + b!) < 4 \land sum! = (a! + b!)]
\]

using the plural interpretation of nondeterminism, \( M_{GetSumBelow4} \) is interpreted in CZ as follows:

\[
[da, db, dsum \in seq N | true, \forall a, b, sum \in N^* \rightarrow (a \in da \land b \in db \land sum \in dsum) \Leftrightarrow (a > 0 \land b > 0 \land (a + b) < 4 \land sum = (a + b))]
\]

The above schema specifies three distinct sequences \( (1, 2) \), \( (1, 2) \), and \( (2, 3) \) as the possible values of \( a! \), \( b! \), and \( sum! \), respectively. Now, in the final program, we may simultaneously select 2 as the preferred value for both \( a! \) and \( b! \). It seems that, in this way, we have assigned \( 2 + 2 = 4 \) to \( sum! \); whereas in the specification phase, we have emphasized that \( sum! \) cannot be greater than 3. Do we encounter a contradiction? The answer is No. This consequence is not a contradiction; it is only counter-intuitive. The reason for this is that although we can simultaneously select 2 as the preferred value for both \( a! \) and \( b! \), we are only allowed to select a value for \( sum! \) from the sequence \( (2, 3) \); in either case, \( sum! \) shall be less than 4.

Such a scenario does not conflict with the intention of who has written the schema \( M_{GetSumBelow4} \) and has been interested in plural nondeterminism. If the writer of this specification has used plural nondeterminism, s/he requires \( a! \) and \( b! \) as two positive natural numbers such that when each of them is added by another positive natural number the result is a number less than 4. Moreover, s/he requires \( sum! \) as a positive natural number which is less than 4. S/he him/herself has not forced any relationship among various evaluations of \( a! \), \( b! \), and \( sum! \).

Now we formalize our current idea to interpret singular and plural nondeterminism. According to this idea, we can assign the two functions \( [\_]D \) and \( [\_]BD \) defined in subsection 3.2 for interpreting only those multi-schemas involving singular nondeterminism; to consider the plural semantics of nondeterminism, we define two new mapping functions. Recall the general form of multi-schemas given in subsection 3.2:

\[
M_{Schema} \cong [x_1 \in A_1, x_2 \in A_2, \ldots, x_m \in A_m, y_1 \in B_1, y_2 \in B_2, \ldots, y_d \in B_d, y_{d+1} \in &B_{d+1}, y_{d+2} \in &B_{d+2}, \ldots, y_n \in &B_n | \phi, \psi]
\]
Now we define a function $P_D$ that is similar to $D$ and is intended to interpret multi-schemas in CZ; unlike $D$, the new function $P_D$ interprets multi-schemas according to the plural semantics of nondeterminism.

Definition 3.3.3. According to the unbounded and plural semantics of nondeterminism, the interpretation of the multi-schema $M_{\text{Schema}}$ in CZ is as follows:

- if none of the variables of $M_{\text{Schema}}$ are nondeterministic, then $[M_{\text{Schema}}]_{P_D} = M_{\text{Schema}}$;
- otherwise, $[M_{\text{Schema}}]_{P_D} \cong [x_1 \in A_1, x_2 \in A_2, \ldots, x_m \in A_m,\,$
  
  \[dy_1 \in P B_1, dy_2 \in P B_2, \ldots, dy_n \in P B_n, | \phi,\]
  
  \[\forall y_1 \in B_1, y_2 \in B_2, \ldots, y_d \in B_d \cdot (y_1 \in dy_1 \land y_2 \in dy_2 \land \ldots \land y_n \in dy_n) \Leftrightarrow \psi]\]

Like the function $D$, $P_D$ behaves as an identity function when applied to a multi-schema involving no nondeterministic variable. Otherwise, it promotes each after state and output variable to a distinct, maximal set of values which satisfy the postcondition of the schema.

In the next definition, a similar approach is followed to define a function $P_{BD}$ which interprets multi-schemas with bounded and plural semantics.

Definition 3.3.4. According to the bounded and plural semantics of nondeterminism, the interpretation of the multi-schema $M_{\text{Schema}}$ in CZ is as follows:

- if none of the variables of $M_{\text{Schema}}$ are nondeterministic, then $[M_{\text{Schema}}]_{P_{BD}} = M_{\text{Schema}}$;
- otherwise, $[M_{\text{Schema}}]_{P_{BD}} \cong [x_1 \in A_1, x_2 \in A_2, \ldots, x_m \in A_m,\,$
  
  \[dy_1 \in \text{seq } B_1, dy_2 \in \text{seq } B_2, \ldots, dy_n \in \text{seq } B_n, | \phi,\]
  
  \[\forall y_1 \in B_1, y_2 \in B_2, \ldots, y_d \in B_d \cdot (y_1 \in dy_1 \land y_2 \in dy_2 \land \ldots \land y_n \in dy_n) \Leftrightarrow \psi]\]

In the next subsection, using four mapping functions $D$, $BD$, $P_D$, and $P_{BD}$, we will interpret both the first and the second components of $(bu, gp, sl, ead)$ simultaneously, but before ending this subsection, we should say that our approach to interpret singular and plural semantics has been based on various occurrences of a nondeterministic variable in a multi-schema. However, we can investigate the semantics of these modalities of nondeterminism from another point of view: we can imagine that an operation schema leads to a sub-routine in the final program which may be called more than once in an actual execution. This may be due to either the various occurrences of the schema in the specification or the presence of the schema in an iteration construct. For example, consider the schema $\text{AcquireLock}$ given in example 4. The corresponding sub-routine of this schema will frequently be called in the final execution for assigning locks to requesting transactions. Various calls of a sub-routine lead to various evaluations of the nondeterministic variables existing in the sub-routine or indeed the initial schema.

Now we can define new semantics for singular and plural nondeterminism based upon the above discussion: by the plural semantics of nondeterminism, in various calls of a sub-routine, we want a nondeterministic variable to be evaluated to different values. For example, in various calls of the sub-routine extracted from $\text{AcquireLock}$, we expect that it does not select a same transaction among those which ask for a lock and do not conflict with the set of current locks. Moreover, we expect that this sub-routine has a fair behavior and does not discard a transaction forever. On the other hand, by the singular semantics of nondeterminism, we want
various calls of a subroutine to have a fixed behavior. For example, consider our CZ specification of the sorting problem: we specify and produce all possible behaviors of this problem. However, in the implementation phase, according to the implementation considerations, such as performance, we may prefer a specific behavior and want this behavior to occur in all calls of the sub-routine.

The recent interpretation of the singular and plural nondeterminism can be only implemented in the execution phase. In other words, only when the program is executed, we can keep track of previous choices, and thus make decisions the same as the previous ones or different from them. To achieve this goal, the following facilities are required:

• The framework used for the formal program development must provide the final program with all possible outputs for each input.

• The framework used for the formal program development must notify the final program of the specified modality of nondeterminism.

• The target programming language should contain nondeterministic constructs which allow the final program to implement singular and plural (and even fair) nondeterminism among the supplied alternatives.  

Fortunately, in our approach to interpret strict nondeterminism, we develop a functional program that is provided with a collection of all possible outputs for each input. Now, by the above discussion, we can think of a target functional language which contains nondeterministic constructs implementing singular and plural nondeterminism. The introduction of nondeterministic constructs to functional languages and λ-calculus has a long tradition [56]. A survey of attempts to add nondeterminism to λ-calculus and functional languages can be found in [18], [56], and [60]. We give a brief overview of such activities here. In [60], the nondeterministic construct $x ::= \text{exp}_1 \sqcap \text{exp}_2$, which nondeterministically assigns one of the expressions $\text{exp}_1$ and $\text{exp}_2$ to $x$, has been added to a typical functional language. Then, erratic, angelic, and demonic interpretations of this construct have been given in terms of the program termination and based on the Plotkin, Hoare, and Smyth power domains, respectively. Also, both the singular and plural interpretations of the construct $x ::= \text{exp}_1 \sqcap \text{exp}_2$ have been presented in [60]. In [56], the type free λ-calculus was enriched with singular, plural, angelic, and demonic nondeterminism. The convergence of computations has been the criterion for distinguishing between angelic and demonic nondeterminism. In [56], unbounded, erratic, plural, and fair nondeterminism have been studied in λ-calculus.

---

20For example, we can think of a programming language that always preserves a history of alternatives that it has so far chosen. In each time, it can then make its choices according to the current history and the initially specified modality of nondeterminism (singular, plural, or fair).

21The extension of functional languages to cover nondeterministic constructs gives rise to some problems originating from the introduction of non-functional features to the functional framework. To see some problems, refer to [18].

22In many literature, power domains have been used to define the semantics of functional programming languages [59]. For a detailed description of these domains, see [59].
By the above discussion, it seems that we only require in somehow to notify final programs of the specified modality of nondeterminism. This forces us to introduce some new nondeterministic constructs in CZ and type theory which distinguish between singular and plural nondeterminism. Moreover, the existing interpretation of CZ in type theory must be extended to translate the new nondeterministic constructs of CZ into their type theoretical counterparts. Finally, some new rules must be added to type theory which help us to extract programs from nondeterministic type theoretical specifications that contain the initial specified modality of nondeterminism. Introducing such facilities to CZ and type theory and studying the ways in which functional programming languages implement singular, plural, and fair nondeterminism can be considered as a direction for future research.

From this point, we use the first proposed semantics for interpreting singular and plural nondeterminism which is based on various occurrences of a nondeterministic variable in a multi-schema. Therefore, in the next subsection, using four mapping functions $D$, $BD$, $PD$, and $PBD$, we will interpret both the first and the second components of $(bu, gp, sl, ead)$ simultaneously.

3.3.4. Interpretation of bounded and unbounded nondeterminism. After interpreting the various values of the third and the fourth components of $(bu, gp, sl, ead)$, we have now either an ordinary schema or a multi-schema involving some nondeterministic variables declared by $&_{(bu, gp, i, i)}$. The former case occurs when the initial multi-schema either has no nondeterministic variable or has some nondeterministic variables with loose nondeterminism ($sl = l$); the latter case, however, occurs when the initial multi-schema has some nondeterministic variables with strict nondeterminism ($sl = s$). Now, let $bu$ be a function that maps the resulting schema in each of the above mentioned forms to ordinary schemas in CZ. Obviously, if the resulting schema is a multi-schema with no nondeterministic variable, $bu$ maps this schema to itself.

Definition 3.3.5 shows how $bu$ interprets the resulting schema when it involves a nondeterministic variable declared by $&_{(bu, gp, i, i)}$. In this definition, we use four mapping functions $D$, $BD$, $PD$, and $PBD$ that have been introduced earlier.

Definition 3.3.5. If a multi-schema $M_{Schema}$ involves some nondeterministic variables declared by

(1) $&_{(u, g, i, i)}$, then the following equality holds:
$$[M_{Schema}]^{bu} = [M_{Schema}[&/_{(u, g, i, i)}]]^{D}$$

(2) $&_{(b, g, i, i)}$, then the following equality holds:
$$[M_{Schema}]^{bu} = [M_{Schema}[&/_{(b, g, i, i)}]]^{BD}$$

(3) $&_{(u, p, i, i)}$, then the following equality holds:
$$[M_{Schema}]^{bu} = [M_{Schema}[&/_{(u, p, i, i)}]]^{PD}$$

(4) $&_{(b, p, i, i)}$, then the following equality holds:
$$[M_{Schema}]^{bu} = [M_{Schema}[&/_{(b, p, i, i)}]]^{PBD}$$

Recall that $D$, $BD$, $PD$, and $PBD$ have been defined to interpret unbounded and singular, bounded and singular, unbounded and plural, and finally bounded and plural interpretations of nondeterminism, respectively.

Theorem 3.3.6 shows that using the functions $ead$, $sl$, and $bu$ in turn for interpreting a multi-schema and then applying our program development method,
we can extract a program which behaves according to the modalities of nondeterminism involved in the initial schema. In this theorem, we use the multi-schema \( M_{\text{Schema}} \), defined earlier in this subsection as the general form of multi-schemas:

\[
M_{\text{Schema}} \equiv [x_1 \in A_1, x_2 \in A_2, \ldots, x_m \in A_m, y_1 \in B_1, y_2 \in B_2, \ldots, y_d \in B_d, \\
\ldots \in \&_{(b, g, s, ead)} B_{d+1}, y_{d+2} \in \&_{(b, g, s, ead)} B_{d+2}, \ldots, y_n \in \&_{(b, g, s, ead)} B_n | \phi, \psi],
\]

where some of the after state and output variables of \( M_{\text{Schema}} \), i.e. \( y_{d+1}, y_{d+2}, \ldots, y_n \), are nondeterministic.

**Theorem 3.3.6.** Let prog be the program that is constructed after applying the function \( \xi \) to the schema \([[M_{\text{Schema}}]^{ead}]^{ead} \) and then proving the correctness of the resulting type theoretical specification. For all \( X_1 \in A_1, \ldots, X_m \in A_m \), if \( \phi(X_1, \ldots, X_m) = true \), then

1. If \( M_{\text{Schema}} \) has no nondeterministic variable or has some nondeterministic variables declared by \&_{(b, g, s, ead)}, then prog, when applied to \( \xi(X_1), \ldots, \xi(X_m) \), produces one and only one valuation \( \xi(Y_1), \ldots, \xi(Y_n) \) of output variables such that
   a. IsBinding\((Y_1, \ldots, Y_n)\) if \( M_{\text{Schema}} \) has no nondeterministic variable.
   b. IsPosOut\((Y_1, \ldots, Y_n, ead)\) if \( M_{\text{Schema}} \) has some nondeterministic variables.
2. If \( M_{\text{Schema}} \) has some nondeterministic variables declared by \&_{(b, g, s, ead)},
   then prog, when applied to \( \xi(X_1), \ldots, \xi(X_m) \), produces a sequence \( dv_1: \text{List}(\xi(B_1), \ldots, \xi(B_n)) \) such that for all \( Y_1 \in B_1, \ldots, Y_n \in B_n, \xi(Y_1), \ldots, \xi(Y_n) \in dv_1 \) iff IsPosOut\((Y_1, \ldots, Y_n, ead)\).
3. If \( M_{\text{Schema}} \) has some nondeterministic variables declared by \&_{(a, g, s, ead)},
   then prog, when applied to \( \xi(X_1), \ldots, \xi(X_m) \), produces a boolean-valued function \( dv_1: \text{List}(\xi(B_1), \ldots, \xi(B_n)) \) \( \Rightarrow \) Bool such that for all \( Y_1 \in B_1, \ldots, Y_n \in B_n, \xi(Y_1), \ldots, \xi(Y_n) \in dv_1 \) iff IsPosOut\((Y_1, \ldots, Y_n, ead)\).
4. If \( M_{\text{Schema}} \) has some nondeterministic variables declared by \&_{(b, p, s, ead)},
   then prog, when applied to \( \xi(X_1), \ldots, \xi(X_m) \), produces \( n \) sequences \( dv_1: \text{List}(\xi(B_1), \ldots, \xi(B_n)) \) such that for all \( Y_1 \in B_1, \ldots, Y_n \in B_n, \xi(Y_1), \ldots, \xi(Y_n) \in dv_1 \wedge \ldots \wedge (Y_n) \in dv_1 \) iff IsPosOut\((Y_1, \ldots, Y_n, ead)\).
5. If \( M_{\text{Schema}} \) has some nondeterministic variables declared by \&_{(a, p, s, ead)},
   then prog, when applied to \( \xi(X_1), \ldots, \xi(X_m) \), produces \( n \) boolean-valued functions \( dv_1: \xi(B_1) \Rightarrow \text{Bool}, \ldots, dv_n: \xi(B_n) \Rightarrow \text{Bool} \) such that for all \( Y_1 \in B_1, \ldots, Y_n \in B_n, \xi(Y_1), \ldots, \xi(Y_n) \in dv_1 \) \( \wedge \ldots \wedge dv_n \) \( (\xi(Y_n)) = true \) iff IsPosOut\((Y_1, \ldots, Y_n, ead)\).

where

- \( \text{IsBinding}(Y_1, \ldots, Y_n) \equiv [x_1 \mapsto X_1, \ldots, x_m \mapsto X_m, y_1 \mapsto Y_1, \ldots, y_n \mapsto Y_n] \in M_{\text{Schema}}[\&_{(b, g, s, ead)}], \) where \( M_{\text{Schema}}[\&_{(b, g, s, ead)}] \) denotes the result of removing all occurrences of \&_{(b, g, s, ead)} in \( M_{\text{Schema}} \).
- \( \text{IsBest}(Y_1, \ldots, Y_n) \equiv \forall Z_1 \in B_1, \ldots, Z_n \in B_n, \text{IsBinding}(Z_1, \ldots, Z_n) \geq \text{sc}(\xi(Y_1), \ldots, \xi(Y_n)) \) \( \geq \text{sc}(\xi(Z_1), \ldots, \xi(Z_n)) \)
- \( \text{IsWorst}(Y_1, \ldots, Y_n) \equiv \forall Z_1 \in B_1, \ldots, Z_n \in B_n, \text{IsBinding}(Z_1, \ldots, Z_n) \leq \text{sc}(\xi(Y_1), \ldots, \xi(Y_n)) \) \( \leq \text{sc}(\xi(Z_1), \ldots, \xi(Z_n)) \)
- \( \text{IsPosOut}(Y_1, \ldots, Y_n, ead) \equiv \text{IsBinding}(Y_1, \ldots, Y_n) \wedge (ead = e \lor (\text{IsBest}(Y_1, \ldots, Y_n) \wedge ead = a) \lor (\text{IsWorst}(Y_1, \ldots, Y_n) \wedge ead = d)) \)
Proof. Although the proof is long, its general approach is obvious: For each of the cases 1-5 above and in terms of the value of ead, involved in $M_{\text{Schema}}$, we can use the functions $\lbrack \lbrack \lbrack ead \rbrack \rbrack$ (see definition 3.3.1), $\lbrack \lbrack e \rbrack \rbrack$ (see definition 3.3.2), and $\lbrack \lbrack bu \rbrack \rbrack$ (see definition 3.3.5) in turn to transform $M_{\text{Schema}}$ into an ordinary schema in CZ. Then we can apply an approach similar to the one used for proving theorem 3.1.2; we do not give the straightforward but tedious details of the proof here. \hfill \Box

In the following two examples, it will be shown how we can use the interpretation function $\lbrack \lbrack [ead,s] \rbrack \rbrack$ in order to interpret multi-schemas in CZ. In the first example, we specify a typical problem of nondeterminism, the eight queens problem.

Example 12. The following multi-schema specifies the eight queens problem in which we specify a typical problem of nondeterminism, the eight queens problem.

In [27], using angelic and demonic choice semantics, we presented a general template for the specification and development of two player games in type theory. The next example shows how our Z-style formalism, presented in this section, can be used for the formal specification and development of any two player game, such as the Nim game described in example 11.

Example 13. In each step of a two player game, a set of choices of a type $T$ exists from which one of the players should select an element. The following multi-schema specifies our player’s choice:

$$M_{\text{OurPlayer}} \equiv \{ rc? \in \text{seq}T, sc! \in \&_{\{b, g, l, a\}} T \mid rc? \neq \{ \}, sc! \in rc? \}$$

$rc?$ is the set of possible choices, and $sc!$ is the element that will be selected by our player. Using $\&_{\{b, g, l, a\}}$ to declare the nondeterministic variable $sc!$ shows that our player is interested in selecting a single ($sl = l$) alternative having the highest score ($ead = a$) from a finite set ($bu = b$) of possible alternatives. By the reasoning given in the previous example, we use the singular semantics of nondeterminism ($gp = g$) here.

We interpret the multi-schema $M_{\text{OurPlayer}}$ in CZ as follows:
In the Appendix D, we have shown that the program satisfying \([\text{M}_{\text{OurPlayer}}]^{\text{cad}^d}\text{bu}\) selects an element with the highest score from a set of possible choices, provided that this set is non-empty; otherwise, the program aborts.

In this section, we have proposed to use multi-schemas instead of conventional operation schemas in order to specify nondeterminism in CZ. However, the operations of the schema calculus (see subsection 2.2) will no longer work on multi-schemas. Therefore, in the next subsection, the schema calculus operations of CZ will be redefined to consider nondeterministic constructs involved in multi-schemas.

3.4. The nondeterministic schema calculus. The schema calculus operations of CZ will no longer work on multi-schemas. For example, recall example 13 in which we specified our player’s choice in a general two player game as follows:

\[ \text{M}_{\text{OurPlayer}} \equiv [rc! \in \text{seqT}, sc! \in \text{&}_{\{b.g.1.a\}}T | rc! \neq \langle \rangle, sc! \in rc?] \]

where \(rc?\) is the set of possible choices, and \(sc!\) is the element that will be selected by our player. The operation specified by the multi-schema \(\text{M}_{\text{OurPlayer}}\) is not total\(^{23}\): \(\text{M}_{\text{OurPlayer}}\) yields a program that aborts for input \(\langle \rangle\) (see the Appendix D). We rewrite the previous specification to describe a total operation:

\[
\begin{align*}
\text{Report} &::= \text{OK} | \text{No\_Choice\_Exists} \\
\text{M}_{\text{P\_OurPlayer}} &\equiv [rc? \in \text{seqT}, sc! \in \text{&}_{\{b.g.1.a\}}T, r! \in \text{Report} | rc? \neq \langle \rangle, sc! \in rc? \land r! = \text{OK}] \\
\text{Exception} &\equiv [rc? \in \text{seqT}, r! \in \text{Report} | rc? = \langle \rangle, r! = \text{No\_Choice\_Exists}]
\end{align*}
\]

The above specification can describe a total operation by applying a disjunction between two operation schemas \(\text{M}_{\text{P\_OurPlayer}}\) and \(\text{Exception}\), but the existing operator for disjunction, i.e. \(\lor\), is not defined on multi-schemas; hence, it seems that we should first interpret \(\text{M}_{\text{P\_OurPlayer}}\) as an ordinary operation schema and then apply a disjunction. The interpretation of \(\text{M}_{\text{P\_OurPlayer}}\) in CZ is as follows:

\[
\begin{align*}
[[\text{M}_{\text{P\_OurPlayer}}]^{\text{cad}^d}\text{bu}] &\approx [rc? \in \text{seqT}, asc \in T, ar \in \text{Report} | rc? \neq \langle \rangle, \\
&\quad asc \in rc? \land ar = \text{OK} \land \\
&\quad \forall (sc, r) \in (T \times \text{Report}) \cdot (sc \in rc? \land r = \text{OK}) \Rightarrow \text{seo}((sc, r)) \leq \text{seo}((asc, ar))]
\end{align*}
\]

Two schemas \([[[\text{M}_{\text{P\_OurPlayer}}]^{\text{cad}^d}]\text{bu}\] and \(\text{Exception}\) are type compatible, and thus we can apply the operator \(\lor\) to them. However, in the resulting schema, there is no relationship between the variable \(r!\) coming from \(\text{Exception}\) and the variable \(ar\) coming from \([[[\text{M}_{\text{P\_OurPlayer}}]^{\text{cad}^d}]\text{bu}\] whereas both variables denote the same output. This problem is due to the renaming of the variable \(r!\) of \(\text{M}_{\text{P\_OurPlayer}}\) forced by the definition of the function \(\text{cad}\) (see definition 3.3.1). It seems that we can easily solve the problem by applying a slight change to the definition of \(\text{cad}\) such that we avoid renaming of output and after state variables. However, we will encounter another problem if we require all of the possible choices with the highest score. In this case, we must use the strict semantics of nondeterminism rather than the loose one in the schema \(\text{M}_{\text{P\_OurPlayer}}\). This causes the output variable \(r!\)

\(^{23}\)If an operation schema places no constraints upon the state of the system beforehand, then we say that the operation it describes is total. Otherwise, the operation is said to be partial. The effect of a partial operation may be undefined for some combinations of before states and inputs. Here, undefined means that the schema tells us nothing about the result, or the resulting state could be any state of the system [75].
existing in $M_P\_OurPlayer$ to be combined with the other variable, i.e. sc!, and the resulting variable to be promoted into a sequence. In this way, the relationship between two variables r! existing in Exception and $M_P\_OurPlayer$ will be lost.

Similar problems occur when we apply the other operators of the schema calculus to schemas which have been obtained from the interpretation of multi-schemas. Therefore, we redefine the schema calculus operations of CZ to consider nondeterministic constructs involved in multi-schemas. In subsection 2.2, we defined the operators $\land_c, \lor_c, \exists_c, \forall_c$, and $;_c$ for the CZ schema calculus operations Conjunction, Disjunction, Existential Quantifier, Universal Quantifier, and Sequential Composition, respectively. Here, we define a new collection of operators consisting of $\land_{nc}, \lor_{nc}, \exists_{nc}, \forall_{nc}$, and $;_{nc}$ instead of the previous ones and interpret them in CZ. As in CZ, the application of Negation to an operation schema and thus a multi-schema is undefined. Thus, we do not consider any new operator for it. Also, we assume all schemas are in their normalized form (see subsection 2.2). We first present a semantics for the operators $\land_{nc}$ and $\lor_{nc}$.

3.4.1. The conjunction and disjunction over multi-schemas. The following definition plays an important role to give a semantics for the operators $\land_{nc}$ and $\lor_{nc}$:

**Definition 3.4.1.** Two multi-schemas are nondeterminism compatible iff each after state and output variable common to two multi-schemas either is not nondeterministic in at least one of the schemas or is nondeterministic with the same modality of nondeterminism in both of them.

In other words, the above definition states that two multi-schemas are not nondeterminism compatible iff there exists at least one after state or output variable common to two schemas that is nondeterministic in both of them but with the different modalities of nondeterminism.

The next definition gives a semantics for the new operators $\land_{nc}$ and $\lor_{nc}$. These operators can be applied to ordinary operation schemas as special cases of multi-schemas. However, Both operators must be applied to schemas which are type compatible (see subsection 2.2) and nondeterminism compatible. Otherwise, the result will be undefined. Notice that when we use at least one of the operations of the nondeterministic schema calculus, such as $\land_{nc}$ and $\lor_{nc}$, in a specification, we must interpret these operations according to the next definition and the other definitions presented in this subsection before transporting any instances of multi-schemas into ordinary ones in CZ. The importance of such a constraint will be justified later in this subsection.

**Definition 3.4.2.** Let $M\_Schema_1 \cong [d_1 | \phi_1, \psi_1]$ and $M\_Schema_2 \cong [d_2 | \phi_2, \psi_2]$ be two multi-schemas. Then we have:

$M\_Schema_1 \land_{nc} M\_Schema_2 \cong [d_{1,nc} d_{2} | \phi_1 \land \phi_2, \psi_1 \land \psi_2]$,

$M\_Schema_1 \lor_{nc} M\_Schema_2 \cong [d_{1,nc} d_{2} | \phi_1 \lor \phi_2, (\phi_1 \land \psi_1) \lor (\phi_2 \land \psi_2)]$,

where $d_{1,nc}$ is a special version of merging $d_1$ and $d_2$: if there exists a variable that is nondeterministic in one of the declarations $d_1$ and $d_2$ and ordinary in the other one, we use the nondeterministic version of the variable in $d_{1,nc} d_{2}$. Since we have assumed that $M\_Schema_1$ and $M\_Schema_2$ are type compatible and nondeterminism compatible, this union makes sense.

The subscript $nc$ used in the new operators denotes nondeterministic and constructive.
To give a semantics for \( \land_{nc} \) and \( \lor_{nc} \), we have used an approach similar to the one used in [47] to define the semantics of \( \land_c \) and \( \lor_c \) except that we have applied a special version of merging. Thus, in a similar way, we can prove that the given semantics are sound. In the next example, we apply the new operator \( \lor_{nc} \) to schemas \( M \_P \_OurPlayer \) and \( Exception \) defined earlier at the beginning of this subsection.

**Example 14.** By the following specification, we describe the total version of the operation specified in example 13:

\[
\begin{align*}
\text{Report} & ::= OK \mid No \_Choice \_Exists \\
M \_P \_OurPlayer & \equiv [rc? \in seqT, sc! \in &\{b,g,l,a\}T, r! \in \text{Report} \mid rc? \not= \langle \rangle, sc! \in rc? \land r! = OK] \\
Exception & \equiv [rc? \in seqT, r! \in \text{Report} \mid rc? = \langle \rangle, r! = No \_Choice \_Exists] \\
M \_T \_OurPlayer & \equiv M \_P \_OurPlayer \lor_{nc} Exception
\end{align*}
\]

In the above specification, two schemas \( M \_P \_OurPlayer \) and \( Exception \) are type compatible and nondeterminism compatible. By the semantics of \( \lor_{nc} \), we have:

\[
\begin{align*}
M \_T \_OurPlayer & \equiv [rc? \in seqT, sc! \in &\{b,g,l,a\}T, r! \in \text{Report} \mid true, \\
\langle rc? \not= \langle \rangle \land sc! \in rc? \land r! = OK \rangle \lor (rc? = \langle \rangle \land r! = No \_Choice \_Exists) ]
\end{align*}
\]

We interpret the multi-schema \( M \_T \_OurPlayer \) as follows:

\[
\begin{align*}
&[[[M \_T \_OurPlayer]]]^{rc? \in T, sc! \in &\{b,g,l,a\}T} = [rc? \in seqT, asc \in T, ar \in \text{Report} \mid true, \\
&\langle rc? \not= \langle \rangle \land asc \in rc? \land ar = OK \rangle \lor (rc? = \langle \rangle \land ar = No \_Choice \_Exists) ] \\
&\Rightarrow sco(asc, ar) \leq sco(asc, ar)
\end{align*}
\]

The above specification results in a program that for \( rc? = \langle \rangle \) reports \( No \_Choice \_Exists \); however, it returns an element of \( rc? \) with the highest score and reports \( OK \).

As it has been mentioned earlier, when we use at least one of the operations of the nondeterministic schema calculus in a specification, we must first apply these operations and then transport any instances of multi-schemas into ordinary ones in CZ. The importance of such a constraint can easily be explored in the above example: suppose that we interpret the multi-schema \( M \_P \_OurPlayer \) before applying the operator \( \lor_{nc} \) between it and the schema \( Exception \). In this way, the relationship between two instances of the variable \( r! \) is removed. The importance of the above issue can also be seen in the subsequent examples of this subsection when we use other operations of the schema calculus.

After giving a semantics for the operators \( \land_{nc} \) and \( \lor_{nc} \), we now interpret the operators \( \exists_{nc} \) and \( \forall_{nc} \) in CZ.

### 3.4.2. The existential and universal quantifiers over multi-schemas.

To give a semantics for the operators \( \exists_{nc} \) and \( \forall_{nc} \), we distinguish between situations in which the set of hidden variables does not contain nondeterministic variables with angelic or demonic nondeterminism and situations in which there exists at least one nondeterministic variable with angelic or demonic nondeterminism in the set of hidden variables:

1. We first consider situations in which the set of hidden variables contain only either ordinary variables or nondeterministic ones with erratic nondeterminism. Exploring the semantics of \( \exists_{nc} \) and \( \forall_{nc} \) in such situations seems to be more convenient: when we hide an ordinary variable or a variable with erratic nondeterminism by \( \forall_{nc} \), we want to show that for all valuations of
this variable, a postcondition must be established, and when we hide such a variable by \(\exists_{nc}\), we want to show that a postcondition must be established for at least one valuation of this variable. By a simple example, we show that the semantics of the existing operators \(\exists\) and \(\forall\) are sufficient to interpret the new operators \(\exists_{nc}\) and \(\forall_{nc}\): consider the following specification:

\[
\begin{align*}
\text{Report} & := OK | \text{Not}_\exists \\
M\_P\_Get2L & \equiv [x? \in N, y!, z! \in & (b,g,s,e)N, r! \in \text{Report} | x!? > 0, \\
& \quad y! < x!? \land z! = 2 * y! \land r! = OK] \\
\text{Exception} & \equiv [x? \in N, r! \in \text{Report} | x!? = 0, r! = \text{Not}_\exists] \\
M\_T\_Get2L & \equiv M\_P\_Get2L \lor_{nc} \text{Exception}
\end{align*}
\]

The multi-schema \(M\_T\_Get2L\) specifies an operation which reports an error when applied to 0. Otherwise, it reports \(OK\) in addition to producing two outputs \(y!\) and \(z!\) that the former is less than the operation input, and the latter is twice the former. Now, consider three sample scenarios: 1. When we hide \(y!\) by \(\exists_{nc}\), we still want to specify all even numbers \(z!\) such that \(z!/2 < x!?\). To achieve this goal, it is enough to use the semantics of current operator for existential quantifier, i.e. \(\exists\), since in this way, the nondeterministic nature of \(z!\) will not be lost, and we can still gain all possible values of \(z!\). 2. When we hide both \(y!\) and \(z!\) by \(\exists_{nc}\), we only want to report \(OK\) or \(\text{Not}_\exists\) in terms of the value of \(x!?\). The same as the previous case, we can use the semantics of \(\exists\). We will lose nothing; 3. Finally, when we hide \(r!\) by \(\exists_{nc}\), we want to specify all possible values (if possible) of \(y!\) and \(z!\) without reporting anything. Again, the semantics of \(\exists\) is sufficient. It can now be concluded that when we use an existential quantifier to hide ordinary variables or nondeterministic ones with erratic nondeterminism, we can apply the semantics of the existing operator \(\exists\) regardless of the nondeterministic nature of the schema variables.

Applying \(\forall_{nc}\) to the multi-schema \(M\_T\_Get2L\) seems to be artificial. However, we can reason about \(\forall_{nc}\) similar to what we did in the previous paragraph for \(\exists_{nc}\). The next definition formalizes the above discussion.

**Definition 3.4.3.** Let \(M\_\text{Schema} \equiv [d | \phi, \psi]\) be a multi-schema, and \(u_h\) be the set of hidden variables containing only ordinary variables or nondeterministic ones with erratic nondeterminism. Then we have:

\[
\begin{align*}
\exists_{nc,u_h}M\_\text{Schema} & \equiv \exists_{u_h}A\_\text{Schema} \setminus \{d - u_h, u_h / \langle (b,g,p,s,e)ad \rangle \} | \phi, \psi \\
\forall_{nc,u_h}M\_\text{Schema} & \equiv \forall_{u_h}A\_\text{Schema} \setminus \{d - u_h, u_h / \langle (b,g,p,s,e)ad \rangle \} | \phi, \psi,
\end{align*}
\]

where \(d - u_h\) is the set of declarations which are in \(d\) and not in \(u_h\), and \(A\_\text{Schema} \setminus \{d - u_h, u_h / \langle (b,g,p,s,e)ad \rangle \} | \phi, \psi\) denotes the result of removing all occurrences of \(\langle (b,g,p,s,e)ad \rangle\) in \(A\).

According to the above definition, before applying the existing semantics of existential quantifier and universal quantifier of CZ, we transform nondeterministic variables, that are being hidden, to their ordinary versions.

(2) Now consider situations in which the set of hidden variables contains at least one nondeterministic variable with angelic or demonic nondeterminism. For this case, we explore the semantics of \(\exists_{nc}\) and \(\forall_{nc}\) by a familiar example. In the following, we have slightly changed the specification of our player’s choice given in example 14:

\[
\begin{align*}
\text{Report} & := OK | \text{No}_\text{Choice}_\exists
\end{align*}
\]
\[ M_{\text{P} \cdot \text{OurPlayer}} \cong [r? \in \text{seq}N, sc_1!, sc_2! \in \&_{(b,g,l,a)}N, r! \in \text{Report} \mid r? \neq (\emptyset), sc_1! \in r? \land sc_2! = 2 \ast sc_1! \land r! = \text{OK}] \]

\[ \text{Exception} \cong [r? \in \text{seq}N, r! \in \text{Report} \mid r? = (\emptyset), r! = \text{No}, \text{ChoiceExists}] \]

\[ M_{\text{T} \cdot \text{OurPlayer}} \cong M_{\text{P} \cdot \text{OurPlayer}} \lor_{\text{nc}} \text{Exception} \]

In the new specification, we have replaced the general type \( T \) by the set of natural numbers. Also, in addition to specifying an alternative \( sc_1! \) with the highest score as an output, we have specified \( sc_2! = 2 \ast sc_1! \) as another output. Now, assume that we apply an existential quantifier to \( M_{\text{T} \cdot \text{OurPlayer}} \). Consider the following scenario: when we hide \( sc_1! \) by the existential quantifier, we still want to specify an even number such that \( sc_2!/2 \) is an alternative with the highest score. If we first hide \( sc_1! \), we cannot then achieve the above mentioned goal since scores have been defined in terms of \( sc_1! \). On the other hand, if we first interpret the angelic nondeterminism, we lose \( sc_1! \) since it is renamed. It would even be combined with other output variables and promoted to a sequence of values if we used the strict interpretation of nondeterminism rather than the loose one; in this way, it is impossible to hide \( sc_1! \).

A similar scenario occurs when we use the universal quantifier. Therefore, it seems that nondeterministic variables must be hidden at the same time that we interpret the angelic and demonic nondeterminism. The next definition formalizes this conclusion for angelic nondeterminism. A similar definition can be given for the demonic one. The form of the next definition is close to the form of the part 1 of definition 3.3.1; however, it also involves the interpretation of quantifiers.

**Definition 3.4.4.** Let \( M_{\text{Schema}} \) be a multi-schema having the following structure:

\[ M_{\text{Schema}} \cong [x_1 \in A_1, ..., x_{bh} \in A_{bh}, x_{bh+1} \in A_{bh+1}, ..., x_{m} \in A_{m}, y_1 \in B_1, ..., y_{ah} \in B_{ah}, y_{ah+1} \in B_{ah+1}, ..., y_{d} \in B_{d}, y_{d+1} \in \&_{(b,g,p,l,i)}B_{d+1}, ..., y_{dah} \in \&_{(b,g,p,l,a)}B_{dah}, y_{dah+1} \in \&_{(b,g,p,s,l,a)}B_{dah+1}, ..., y_{n} \in \&_{(b,g,p,s,l,a)}B_{n} \mid \phi, \psi]\]

where some of the before state and input variables, i.e. \( x_{bh+1}, ..., x_{m} (bh \leq m) \), some of the ordinary, after state and output variables, i.e. \( y_{ah+1}, ..., y_{d} (ah \leq d) \), and some of the nondeterministic, after state and output variables, i.e. \( y_{dah+1}, ..., y_{n} (d \leq dah \leq n) \) are in the set of hidden variables, i.e. \( u_{h} \); \( u_{h} \) contains at least one nondeterministic variable (with angelic nondeterminism). In other words, \( dah < n \). Now we have:

(a) \( \exists_{u_{h}} \cdot M_{\text{Schema}} \cong [x_1 \in A_1, ..., x_{bh} \in A_{bh}, ay_1 \in \&_{(b,g,p,l,i)}B_{1}, ..., ay_{ah} \in \&_{(b,g,p,l,i)}B_{ah}, ay_{d+1} \in \&_{(b,g,p,l,i)}B_{d+1}, ..., ay_{dah} \in \&_{(b,g,p,l,i)}B_{dah} \mid \exists_{u_{h}} \mid \&_{(b,g,p,l,a)}] \cdot \phi, \psi \]

\( \exists_{u_{h}} \mid \&_{(b,g,p,l,a)}] \cdot \phi \land (\psi[ay_1/y_1] ... [ay_{ah}/y_{ah}] [ay_{d+1}/y_{d+1}] ... [ay_{dah}/y_{dah}] \land (\forall(z_1, z_2, ..., z_{n}) : ((B_1 \times B_2 \times ... \times B_n) \mid \psi[z_1/y_1] ... [z_{n}/y_{n}] \Rightarrow \text{sco}((z_1, z_2, ..., z_{n})) \leq \text{sco}((ay_1, ..., ay_{ah}, y_{ah+1}, ..., y_{d}, ay_{d+1}, ..., ay_{dah}, y_{dah+1}, ..., y_{n})))]) \]

(b) \( \forall_{u_{h}} \cdot M_{\text{Schema}} \cong [x_1 \in A_1, ..., x_{bh} \in A_{bh}, \]

\(^{25}\)Notice that the general form of multi-schemas, used in definition 3.3.1, has been modified slightly here: we distinguish between those nondeterministic variables that are being hidden and the others.
exists choice

terministic variable, but calculus operations, i.e. ;

sequential composition between multi-schemas.

By the above discussion, when

We can reason about this fact from another point of view: If we wrote

\( GetLessThan \)

\( ∼ \)

\( \exists \)

\( \exists \)

\( \& \)

\( ∨ \)

\( \phi \).

\( \forall \) \( a_1 \in \&_{(b, y, s, l)} B_1, \ldots, a_y \in \&_{(b, y, s, l)} B_\alpha \),

\( a_y \in \&_{(b, y, s, l)} B_{d+1}, \ldots, a_y \in \&_{(b, y, s, l)} B_\alpha | \forall z_i | \&_{(b, y, s, l)} \)

\( \phi \).

\( \forall a_1 | \&_{(b, y, s, l)} \{ \psi[a_1/y_1] \ldots [a_y/y_\alpha] \[ a_y \in \&_{(b, y, s, l)} y_\alpha \] \land

\( \forall (z_1, z_2, \ldots, z_n) \in (B_1 \times B_2 \times \ldots \times B_\alpha) \cup \psi[z_1/y_1] \ldots [z_n/y_\alpha] \Rightarrow

\( sco((z_1, z_2, \ldots, z_n)) \leq sco((a_1, \ldots, a_y, a_y + 1, \ldots, y_\alpha, a_\alpha + 1, \ldots, a_\beta (y_\beta + 1, \ldots, y_\gamma))) \}

where \( A | \&_{(b, y, s, l)} \) denotes the result of removing all occurrences of 

\( \&_{(b, y, s, l)} \) in \( A \).

The above definition shows how we can interpret the quantification over

multi-schemas at the same time that we replace nondeterministic variables

by their versions declared by \( \&_{(b, y, s, l)} \).

Now we apply \( \exists \) to the multi-schema \( M \cdot T \cdot OurPlayer \) given earlier, and

then interpret it in CZ:

\[ \exists_{nc} sec_1 \in N \cdot M \cdot T \cdot OurPlayer \cong \]

\[ \exists_{nc} sec_1 \in N \cdot [rc \in seq N, \text{asc}_2 \in \&_{(b, g, l, a)} N, r! \in Report \mid true, \]

\[ (rc \neq \emptyset \land sc \in sec_1 \in rc \land sc_2 \neq 2 \ast sc_1 \land r! = OK) \lor (rc \neq \emptyset \land r! = \text{No\_Choice\_Exists})] \cong \]

\[ [rc \in seq N, asc_2 \in N, ar \in Report \mid true, \]

\[ \exists_{nc} sec_1 \in N \cdot \{(rc \neq \emptyset \land sc \in sec_1 \in rc \land asc_2 = 2 \ast sc_1 \land ar = OK) \lor \]

\[ (rc \neq \emptyset \land ar = \text{No\_Choice\_Exists})] \land \]

\[ \forall (z_1, z_2, z_3) \in (N \times N \times Report) \cdot (rc \neq \emptyset \land z_1 \in rc \land z_2 = 2 \ast z_1 \land z_3 = OK) \lor \]

\[ \exists_{nc} sec_1 \in \text{No\_Choice\_Exists}; \text{for rc?} \neq \emptyset, \text{this program reports OK and returns an} \]

even number \( sc_2 \) such that \( sc_2/2 \) is an element of \( rc? \) with the highest

score.

Now we define a semantics for the last operation of the new set of the schema

calculus operations, i.e. \( \cdot nc \).

3.4.3. The sequential composition between multi-schemas. To define a semantics for

\( S_1 \cdot nc \cdot S_2 \), we must first consider a special case in which \( S_1 \) has at least one nondeterministic

variable, but \( S_2 \) is an ordinary schema, i.e., it has no nondeterministic variable. For instance, consider the following simple specification in CZ:

\[ \text{GetLessThan} \cong [x \in N, x \in N | true, x \leq x] \]

\[ \text{GetSquare} \cong [x, x \in N | true, x = x \times x] \]

GetSquare is a deterministic schema, and thus we have written it without using nondeterministic variables; however, when using \( \text{GetLessThan} \cdot nc \text{GetSquare} \), we want to specify an operation that returns a set of all possible natural numbers each of whose members is the square of a number equal to or less than the operation input.

We can reason about this fact from another point of view: If we wrote \( \text{GetLessThan} \)

in the original CZ without using nondeterministic variables, then by definition 3.1.1, the following schema would be a nondeterministic schema.

\[ \text{GetLessThan} \cdot ic \text{GetSquare} \cong [x, x \in N | true, \exists x \in N | x \leq x \land x = x \times x] \]

By the above discussion, when \( S_1 \) is a multi-schema having at least one nondeterministic variable, and \( S_2 \) is an ordinary schema, before interpreting \( S_1 \cdot nc \cdot S_2 \), we must replace the after state and output variables of \( S_2 \) by their nondeterministic
versions using the instance of the nondeterministic operator $\&_{bu, gp, sl, ead}$ involved in $S_1$.

Now we introduce our method to interpret $S_1':inc S_2$ in CZ. We first investigate how the modalities of nondeterminism involved in $S_1$ are propagated through the sequential composition: by $S_1':inc S_2$, we want to state that there exists some values of the after state and output variables of $S_1$ (or the before state and input variables of $S_2$) by which the postconditions of $S_2$ are established, provided that the preconditions of $S_1$ hold (To achieve this goal, we apply an existential quantifier before the predicates of $S_1$ and $S_2$ which quantifies over the after state and output variables of $S_1$ and then hide these variables from the resulting specification). In this way, the only modalities of nondeterminism that seems to affect the sequential composition are angelic and demonic nondeterminism; by using the existential quantifier in the predicate part of the final schema, other modalities of nondeterminism become neutral.

Using the angelic (demonic) modality of nondeterminism in $S_1$, we want to state that there exists some values of the after state and output variables of $S_1$ that have the highest (lowest) score among possible values of these variables and result in establishing the postconditions of $S_2$, provided that the preconditions of $S_1$ hold. The next definition summarizes the above discussion.

**Definition 3.4.5.** Let $S_1 \equiv [d_1^a, d_1^b | \phi_1, \psi_1]$ and $S_2 \equiv [d_2^a, d_2^b | \phi_2, \psi_2]$ be two multi-schemas, where $d^b (d^a)$ denotes the set of declarations of before state and input (after state and output) variables. Now we have:

1. If the modality of the nondeterminism involved in $S_1$ is erratic, then: $S_1:inc S_2 \equiv S_1[\&_{bu, gp, sl, e}] : S_2$, where $A[\&_{bu, gp, sl, e}]$ denotes the result of removing all occurrences of $\&_{bu, gp, sl, e}$ in $A$.

2. If the modality of the nondeterminism involved in $S_1$ is angelic, then:

   $S_1:inc S_2 \equiv [d_1^a, d_1^b | \phi_1, \exists d_2^a[\phi_2'[\phi_2'] \wedge \psi_2]' \wedge \psi_2]'-1, \ldots, 1, 2) \}
   \Rightarrow \text{sco}(d_1'[\phi_2'] \geq \text{sco}(d_2'[\phi_2'])],
   \text{where } A[\phi_2']$ and $A[\psi_2']$ denote the result of substituting all variables of $A$ by their versions ending with $'$ and $''$, respectively. Also, $A[\phi_2']$ and $A[\psi_2']$ correspond to replacing after state and output (before state and input) variables of $A$ by their versions ending with $'$ and $''$, respectively.

3. If the modality of the nondeterminism involved in $S_1$ is demonic, then:

   $S_1:inc S_2 \equiv [d_1^a, d_1^b | \phi_1, \exists d_2^a[\phi_2'[\phi_2'] \wedge \psi_2]' \wedge \psi_2]'-1, \ldots, 1, 2) \}
   \Rightarrow \text{sco}(d_1'[\phi_2'] \leq \text{sco}(d_2'[\phi_2'])].

As before, for the composition to be defined, both schemas must refer to the same state: for any primed component in $S_1$, there must be an unprimed component of the same name in $S_2$. Also, we assume that for any output variable in $S_1$, there is an input variable of the same name in $S_2$. In the above definition, the modality of the nondeterminism involved in $S_1$ determines the form of the interpretation of $S_1:inc S_2$. Notice that, as reasoned earlier, when $S_1$ is a multi-schema having at least one nondeterministic variable, and $S_2$ is an ordinary schema, before using the above defined interpretations, we must replace the after state and output variables
of $S_2$ by their nondeterministic versions using the instance of the nondeterministic operator $\&_{\{b,a,g,p,s,c,a\}}$ involved in $S_1$.

In the next example, we extend the problem specified in example 14 and give a new specification of it using the operator $\circ_{\text{inc}}$.

**Example 15.** In example 14, we have given a specification of our player’s choice in a general two player game. Now we rewrite this specification to specify a sequential composition of our player’s choice and our opponent’s choice:

\[
\text{Report} := \text{OK} \land \text{No\_Choice\_Exists}
\]

\[
\text{Game} \equiv \{rc \in \text{seq}T\}
\]

The state schema $\text{Game}$ is written to specify states of the game, where $rc$ is the set of possible choices in each state. Having the schema $\text{Game}$, we slightly change the previous specification of our player’s choice as follows:

\[
M_{\_P\_OurPlayer} \equiv \{\Delta \text{Game}, s! \in \&_{\{b,a,g,p,s,c,a\}}T, r! \in \text{Report} \mid rc \neq \langle \rangle, \text{sc}! \in rc \land rc' = f(rc, sc!) \land r! = \text{OK}\}
\]

Similarly, we can specify our opponent’s choice as follows:

\[
M_{\_P\_OurOpponent} \equiv \{\Delta \text{Game}, s! \in \&_{\{b,a,g,p,s,c,a\}}T, r! \in \text{Report} \mid rc \neq \langle \rangle, \text{sc}! \in rc \land rc' = f(rc, sc!) \land r! = \text{OK}\}
\]

In the above two schemas, $rc' = f(rc, sc!)$ indicates that the set of possible choices in the after state, $rc'$, is obtained by a mapping $f$ of this set in the before state, $rc$, and the current selected alternative, $sc!$. To specify a total operation, we use the following operation schema:

\[
\text{Exception} \equiv \{\exists \text{Game}, r! \in \text{Report} \mid rc = \langle \rangle, r! = \text{No\_Choice\_Exists}\}
\]

Now we give the final specification as follows:

\[
\text{Game\_Round} \equiv (M_{\_P\_OurPlayer} \lor_{\text{inc}} \text{Exception}) \land_{\text{inc}} (M_{\_P\_OurOpponent} \lor_{\text{nc}} \text{Exception})
\]

Using the semantics of $\lor_{\text{nc}}$, we have:

\[
\text{Game\_Round} \equiv \{\Delta \text{Game}, s! \in \&_{\{b,a,g,p,s,c,a\}}T, r! \in \text{Report} \mid true, \quad (rc \neq \langle \rangle) \land sc! \in rc \land rc' = f(rc, sc!) \land r! = \text{OK} \lor (rc = \langle \rangle) \land r! = \text{No\_Choice\_Exists}\} \land_{\text{inc}} \{\Delta \text{Game}, s! \in \&_{\{b,a,g,p,s,c,a\}}T, r! \in \text{Report} \mid true, \quad (rc \neq \langle \rangle) \land sc! \in rc \land rc' = f(rc, sc!) \land r! = \text{OK} \lor (rc = \langle \rangle) \land r! = \text{No\_Choice\_Exists}\}
\]

Since the modality of the nondeterminism involved in the first operand of $\land_{\text{inc}}$ above is angelic, we use the case 2 of definition 3.4.5 to interpret $\land_{\text{inc}}$.

\[
\text{Game\_Round} \equiv \{rc \in \text{seq}T, rc' \in \text{seq}T, sc! \in \&_{\{b,a,g,p,s,c,a\}}T, r! \in \text{Report} \mid true, \quad \exists r'' \in \text{seq}T, r''' \in \text{Report} \quad (((rc \neq \langle \rangle) \land sc'' \in rc \land rc'' = f(rc, sc'') \land r'' = \text{OK}) \lor (rc = \langle \rangle) \land r'' = \text{No\_Choice\_Exists})) \land (((rc'' \neq \langle \rangle) \land sc'' \in rc'' \land rc' = f(rc'', sc!) \land r! = \text{OK}) \lor (rc'' = \langle \rangle) \land r! = \text{No\_Choice\_Exists})) \land (((rc'' \neq \langle \rangle) \land sc'' \in rc'' \land rc' = f(rc'', sc!) \land r! = \text{OK}) \lor (rc'' = \langle \rangle) \land r! = \text{No\_Choice\_Exists})) \land ((rc \neq \langle \rangle) \land sc'' \in rc'' \land rc'' = f(rc, sc'') \land r'' = \text{OK}) \lor (rc = \langle \rangle) \land r'' = \text{No\_Choice\_Exists})) \land ((rc'' \neq \langle \rangle) \land sc'' \in rc'' \land rc' = f(rc'', sc!) \land r! = \text{OK}) \lor (rc'' = \langle \rangle) \land r! = \text{No\_Choice\_Exists}))
\]

\[
\Rightarrow sc''(r''), (sc'', r''') \geq sc''(r'''), (sc'', r''')
\]

Now we obtain the final specification in CZ by interpreting the nondeterministic construct $\&_{\{b,a,g,p,s,c,a\}}$ used in the above schema. In the following schema, we use $\psi$ as a shorthand for the postcondition of the above schema:

\[
\text{Game\_Round} \equiv \{rc \in \text{seq}T, drc \in \text{seq}T, dsc \in T, dr \in \text{Report} \mid true, \quad \psi[drc/rc'][dsc/sc][dr/r!] \land (\forall (rc, sc, r) \in (\text{seq}T \times T \times \text{Report}))
\]

\[
\psi[rc/rc'][sc/sc][r/r!] \Rightarrow sc((drc, dsc, dr)) \leq sc((rc, sc, r))
\]
The above specification results in a program that first provides an element of \( rc \) with the highest score for our player and then provides an element with the lowest score for our opponent, provided that \( rc \) has at least two elements. If \( rc \) has less than two elements, the program reports \text{No\_Choice\_Exists}.

In the next subsection, we apply the new formalism to a familiar problem, the strict 2PL algorithm whose both informal and formal specifications were given earlier in examples 1.2 and 2.2, respectively. In this way, we demonstrate that our formalism can be considered as a starting point for specifying and developing concurrent systems. We also dedicate few pages to show how we can verify the resulting specification. Notice that we will not intend to give special solutions for specifying and verifying concurrent systems. Indeed, it is out of the main focus of the paper, i.e. nondeterminism. Therefore, we will give a brief illustration of our ideas and delegate most of the details to other work.

3.5. Case study - the strict 2PL method. In example 4, we specified the strict 2PL algorithm for concurrency control in database management systems. Although the problem being specified was nondeterministic, this was not indicated by the specification explicitly. As we have shown in this section, after deriving a program from a correctness proof of such a specification, only one of the possible behaviors will appear in the final program. In the next example, using the notion of multi-schemas and the new operations of the schema calculus, we revise the previous specification of the strict 2PL method to address the above mentioned issue.

Example 16. We replace the operation schema \( \text{AcquireLock} \), written in example 4, by a new multi-schema \( M_{\text{AcquireLock}} \):

\[
M_{\text{AcquireLock}} \cong [\Delta \text{CCDB}, \text{transId}! \in \&_{\{b,p,s,r\}} N | \text{true}, \text{transId}! \in \text{dom activeTransactions}\wedge \{(\{\text{transId}!\} \triangleleft \text{runningTransactions}) \cap \text{conflictLock}(\text{lockBase}) = \emptyset)\wedge (\text{lockBase}.\text{locks} = \text{lockBase}.\text{locks}\}\}

\{[\text{transId}! \mapsto \text{transId}, \text{command} \mapsto \text{FirstCommand}(\text{runningTransactions}.\text{transId}!), \text{dataItem} \mapsto \text{FirstDataItem}(\text{runningTransactions}.\text{transId}!)]\}]
\]

In the new schema, the only thing that is modified is replacing the output variable \( \text{transId!} \) by its nondeterministic version. Notice that we have declared \( \text{transId!} \) as a nondeterministic variable with the plural modality of nondeterminism. The reason was given in subsection 3.3.3 when we were trying to interpret the singular and plural nondeterminism in CZ: according to the second approach for interpreting these modalities of nondeterminism, we expect that in various calls of the subroutine extracted from \( M_{\text{AcquireLock}} \), it does not select a same transaction among those which ask for a lock and do not conflict with the set of current locks. Of course, as we have emphasized in subsection 3.3.3, such an interpretation can only be implemented in the final program.

Using the new operations of the schema calculus, the final specification is as follows:

\[
M_{\text{STwoPL}} \cong M_{\text{AcquireLock}};_{\text{nc}} (\text{Read} \lor_{\text{nc}} \text{Write} \lor_{\text{nc}} \text{Commit} \lor_{\text{nc}} \text{Abort})
\]

Assume that the disjunctions between four multi-schemas \( \text{Read}, \text{Write}, \text{Commit}, \) and \( \text{Abort} \) result in a multi-schema called \( \text{RWCA} \). This schema specifies the task done by one of the operations read, write, commit, or abort in terms of the first command of the input transaction identified by \( \text{transId!} \). As we have discussed earlier, since \( M_{\text{AcquireLock}} \) has a nondeterministic variable, and \( \text{RWCA} \) has no nondeterministic variable, before interpreting \( M_{\text{AcquireLock}};_{\text{nc}} \text{RWCA} \), we must
redeclare the after state and output variables of RWCA using the nondeterministic operator \( &_{(b,p,s,e)} \) involved in \( M_{AcquireLock} \). Now, using the part 1 of definition 3.4.5 to interpret the sequential composition and then applying each of the functions \( \text{read} \), \( \text{write} \), and \( \text{commit} \) in turn yield an operation schema specifying all possible operations that can be done in a state of the system.

The notion of nondeterminism has an important role when we want to state the semantics of concurrent processes: concurrency is usually modelled by the nondeterministic interleaving of atomic operations [23]. Evans in a series of his papers (for example, [21], [22], and [23]) has tried to model such a semantics of concurrency in the Z notation. He describes the concurrent behavior of a system in terms of its allowable computations, i.e. sequences of state changes, that result from the interleaved execution of concurrent operations.

In example 16, we have given a formal specification of the strict 2PL method for controlling concurrency in a database management system. Our approach relies on specifying nondeterminism explicitly whereas Evans has not given a specific method to support nondeterminism; an Evans’s specification of a concurrent system is implicitly nondeterministic and yields a program that is provided with only one of the possible interleaved executions of concurrent processes. On the other hand, in comparison to the work of Evans, the main drawback of our approach is as follows: we explicitly specify all possible choices that can be made in a state. For instance, in example 16, we explicitly specified all possible transactions that can acquire a lock or in other words, all possible operations that can be done in a state. However, we modelled the action of only one operation in a state. More precisely, we did not model the interleaving of atomic operations read, write, commit, and abort that leads to a valid computation.

Capturing the ideas stated by Evans, we can easily extend our approach to model the nondeterministic interleaving of atomic operations. To achieve this goal, we discard the previous specification of the database management system written in example 16 and formalize the behavior of the system as a history of state changes resulting from the repeated application of the schema \( STwoPL \) given in example 4:

\[
CCDBBehavior \equiv [\text{stateChange} \in \text{seqCCDB}] \land \text{true}, \forall n \in \mathbb{N} \cdot n > 0 \land n < \#\text{stateChange} \Rightarrow [CCDB \mapsto \text{stateChange}.n, CCDB' \mapsto \text{stateChange}.(n + 1)] \in STwoPL
\]

Briefly, the above schema specifies a computation as a valid sequence of system states if for every two successive steps, they are related by the schema \( STwoPL \). To solve the problem of Evans’s approach and thus obtain all possible computations in the final program, it is sufficient to rewrite the schema \( CCDBBehavior \) as a multi-schema in which the variable \( \text{stateChange} \) has been replaced by its nondeterministic version having strict nondeterminism. In the above specification, we have not forced fairness for computations. This issue can be addressed easily by adding related constraints to the predicate part of \( CCDBBehavior \). The details of such a modelling can be delegated to another research. Notice that since we have given the semantics of all new nondeterministic constructs and new operations of the schema calculus in CZ itself, we can use the constructive framework, introduced in section 2, to develop a program from the new specification of the concurrent database management system. We do not develop such a program here. Nevertheless, theorem 3.3.6 guarantees that this program is provided with a sequence including
all possible interleaved executions of the concurrent transactions. This is the main advantage of our approach to that of Evans.

Besides specifying concurrent systems, there is a related field to explore approaches for verifying formal specifications of concurrent systems. In [21] and [22], Evans has presented a set of inference rules for formally proving safety and liveness\textsuperscript{26} properties of Z specifications of concurrent systems. The rules are based on assertional verification techniques but are implemented in an emerging deductive calculus for Z, called W\textsuperscript{27}. In the assertional approach, safety and liveness properties are proved by examining the effect of individual operations on the system state.

The logic used by Evans is based on proofs of the form $DOp \vdash P(s) \Rightarrow Q(s')$, where $DOp$ is a deterministic operation schema, and $s$ is the state schema of the system, respectively. $P(s)$ and $Q(s)$ are both predicates over variables of $s$. $Q(s')$ is obtained from $Q(s)$ by replacing variables of $s$ by their primed versions. $DOp \vdash P(s) \Rightarrow Q(s')$ states that if $DOp$ is applied to a state $s$ in which the predicate $P$ holds, then predicate $Q$ will hold after the application of $DOp$. An essential part of the logic is the fairness assumption that any operation whose preconditions are satisfied by the current state of the system will eventually be executed. Safety and liveness properties are proved simply by showing that some or all the operations in the system will result in the desired properties becoming true. As a simple example, suppose that a concurrent system has a deterministic operation $Op$ for which $Op \vdash true \Rightarrow Q(s')$ holds. It can be claimed at any point during the execution of the system that $Q$ will eventually hold. This is because operation $Op$ will be executed eventually (by the fairness assumption) and thereby establish $Q$.

Most properties one may wish to prove of a concurrent system can be expressed using proofs of the above mentioned form. However, it is cumbersome to do so all the time; hence, in [21] and [22], a number of commonly used inference rules have been defined in terms of these proofs. For example, safety properties have been expressed by the inference rule unless interpreted in Z as follows: For any specification $S$, where $Op_1, ..., Op_n$ represent all the possible deterministic operations of $S$, then:

$$
\begin{array}{c}
Op_1 \vdash P(s) \land \neg Q(s) \Rightarrow P(s') \lor Q(s') \\
... \\
Op_n \vdash P(s) \land \neg Q(s) \Rightarrow P(s') \lor Q(s') \\
\end{array}
\quad \vdash P \text{ unless } Q
$$

This states that if the system is in a state that satisfies $P$ and $\neg Q$, then after the application of any of the operations in the system either $P$ will remain true or $Q$ will become true. Besides the above rule, two concepts stable predicate and invariant are also useful here. A stable predicate remains true once it has become true, while an invariant is always true during the lifetime of the system. Both of the concepts are formally defined using unless:

$P \text{ unless false } \Leftrightarrow P \text{ is stable}$

$((\text{initialization schema } \vdash P(s')) \land P \text{ is stable}) \Leftrightarrow P \text{ is invariant}$

To prove that a property always holds during the execution of the system, it is sufficient to use the unless rule and the above definitions to show that the property

\textsuperscript{26}A safety property is expressed in terms of an invariant, i.e. a property that must always hold during the execution of the system, whilst a liveness property asserts that a property will eventually become true.

\textsuperscript{27}See [74] for full details.
is an invariant; there is no need to prove that the property holds in all system states existing in a computation.

Now we apply the above mentioned rule and definitions to our CZ specification of the concurrent database management system given in example 4. When defining the schema \textit{LockBase} to specify the set of existing locks, we considered an invariant for it indicating that it never contains conflicting locks at each time. Suppose that we have not defined such a constraint. Then, the following safety property must be proved:

\[
\text{SafeLockBase}(CCDB) \equiv \forall x, y \in \text{lockBase}.\text{locks} \cdot (x.\text{dataItem} = y.\text{dataItem} \Rightarrow (x.\text{transId} = y.\text{transId} \lor (x.\text{command} = \text{Read} \land y.\text{command} = \text{Read})))
\]

\text{SafeLockBase}(CCDB) is an invariant; there is no need to prove that the property holds in all system states existing in a computation.

Now we apply the above mentioned rule and definitions to our CZ specification of the concurrent database management system given in example 4. When defining the schema \textit{LockBase} to specify the set of existing locks, we considered an invariant for it indicating that it never contains conflicting locks at each time. Suppose that we have not defined such a constraint. Then, the following safety property must be proved:

\[
\text{SafeLockBase}(CCDB) \equiv \forall x, y \in \text{lockBase}.\text{locks} \cdot (x.\text{dataItem} = y.\text{dataItem} \Rightarrow (x.\text{transId} = y.\text{transId} \lor (x.\text{command} = \text{Read} \land y.\text{command} = \text{Read})))
\]

\text{SafeLockBase}(CCDB) is a predicate over the variable \textit{lockBase} of the state schema \textit{CCDB}. To show that the predicate \textit{SafeLockBase} is an invariant, we must prove the following conjunction (Recall that, in this way, there is no need to show \textit{SafeLockBase} holds in all system states existing in the sequence \textit{stateChange}, declared in the schema \textit{CCDBBehavior}):

1. \textit{CCDB_Init} ⊢ \text{SafeLockBase}(CCDB′)
2. \textit{SafeLockBase} is stable

The predicate part of the schema \textit{CCDB_Init} demonstrates that \textit{lockBase′.locks} = ∅; it simply proves the first conjunct of the above conjunction. By the unless rule, the second conjunct can be proved if we show that all the following predicates hold:

3. \textit{AcquireLock} ⊢ \text{SafeLockBase}(CCDB) ⇒ \text{SafeLockBase}(CCDB′)
4. \textit{Read} ⊢ \text{SafeLockBase}(CCDB) ⇒ \text{SafeLockBase}(CCDB′)
5. \textit{Write} ⊢ \text{SafeLockBase}(CCDB) ⇒ \text{SafeLockBase}(CCDB′)
6. \textit{Commit} ⊢ \text{SafeLockBase}(CCDB) ⇒ \text{SafeLockBase}(CCDB′)
7. \textit{Abort} ⊢ \text{SafeLockBase}(CCDB) ⇒ \text{SafeLockBase}(CCDB′)

Fortunately, all the operation schemas \textit{Read}, \textit{Write}, \textit{Commit}, and \textit{Abort} are deterministic. Moreover, there are straightforward proofs to show the correctness of the cases 4-7 above. However, \textit{AcquireLock} is nondeterministic, and thus Evans’s rules cannot be applied to it; suppose that we try to use such rules: We must show that the following predicate holds:

\[
\text{AcquireLock} \vdash (\forall x, y \in \text{lockBase′.locks} \cdot (x.\text{dataItem} = y.\text{dataItem} \Rightarrow (x.\text{transId} = y.\text{transId} \lor (x.\text{command} = \text{Read} \land y.\text{command} = \text{Read})))) \Rightarrow
\]

\[
(\forall x, y \in \text{lockBase′.locks} \cdot (x.\text{dataItem} = y.\text{dataItem} \Rightarrow (x.\text{transId} = y.\text{transId} \lor (x.\text{command} = \text{Read} \land y.\text{command} = \text{Read}))))
\]

By the definition of \textit{AcquireLock} (see example 4), we have:

\[
\text{lockBase′.locks} = \text{lockBase.locks} \cup \\
\{[\text{transId} \mapsto \text{transId}, \text{command} \mapsto \text{FirstCommand}(\text{runningTransactions.\text{transId}}!), \text{dataItem} \mapsto \text{FirstDataItem}(\text{runningTransactions.\text{transId}}!)]\}
\]

\text{transId}! has a nondeterministic nature and does not refer to a single transaction; we only know that it corresponds to an active transaction whose first operation does not conflict with the current set of locks. Indeed, this has been the main reason for ignoring nondeterministic operations (at least in a formal form) in Evans’s rules.

---

\textsuperscript{28} \textit{SafeLockBase}(CCDB′) is obtained from \textit{SafeLockBase}(CCDB) by replacing variables of \textit{CCDB} by their primed versions.
Using the notion of multi-schemas and applying a slight change to the Evans’s basic rule, i.e. \( Op \vdash P(s) \Rightarrow Q(s') \), we can simply extend his work to cover nondeterministic operations: recall the general form of multi-schemas given in section 3:

\[
M_{\text{Schema}} \equiv [x_1 \in A_1, x_2 \in A_2, \ldots, x_m \in A_m, y_1 \in B_1, y_2 \in B_2, \ldots, y_n \in B_n],
\]

where the after state and output variables \( y_{d+1}, y_{d+2}, \ldots, y_n \) are nondeterministic. Now, for the multi-schema \( M_{\text{Schema}} \), we change the Evans’s basic form of proofs, i.e. \( M_{\text{Schema}} \vdash P(s) \Rightarrow Q(s') \), to the new one:

\[
M_{\text{Schema}} \vdash P(s) \Rightarrow (\forall (y_1, y_2, \ldots, y_n) \in (B_1 \times B_2 \times \ldots \times B_n) \cdot (\psi \Rightarrow Q(s')))\]

The above form of proof states that if a multi-schema \( M_{\text{Schema}} \) is applied to a state \( s \) in which the predicate \( P \) holds, then predicate \( Q \) will hold, for all possible values of the after state and output variables of \( M_{\text{Schema}} \), after the application of this schema. As before, we consider ordinary operation schemas as special cases of multi-schemas having no nondeterministic variable; hence, we can apply the new form of proof to ordinary operation schemas.

Similar to the change we have applied to the Evans’s basic rule, we must change all his rules, such as the \textit{unless} rule. Having the new set of rules, we can replace the ordinary schema \texttt{AcquireLock} by its corresponding multi-schema, \texttt{M_AcquireLock}, and show that the predicate

\[
M_{\text{AcquireLock}} \vdash (\forall x, y \in \text{lockBase}.locks \cdot (x.dataItem = y.dataItem \Rightarrow (x.transId = y.transId \lor (x.command = \text{Read} \land y.command = \text{Read})))) \Rightarrow
\]

\[
(\forall (\text{transId}!, \text{DataBase}', \text{lockBase}', \text{activeTransactions}', \text{runningTransactions}', \text{localDBs}') \in (N \times \text{DataBase} \times \text{LockBase} \times \text{seq Transaction} \times \text{seq Transaction} \times \text{seq DataBase}) \cdot (\psi \Rightarrow (\forall x, y \in \text{lockBase}.locks \cdot (x.dataItem = y.dataItem \Rightarrow
\]

\[
(x.transId = y.transId \lor (x.command = \text{Read} \land y.command = \text{Read}))))))
\]

holds (\( \psi \) is the postcondition of \( M_{\text{AcquireLock}} \)) instead of showing the correctness of the following predicate:

\[
\text{AcquireLock} \vdash (\forall x, y \in \text{lockBase}.locks \cdot (x.dataItem = y.dataItem \Rightarrow (x.transId = y.transId \lor (x.command = \text{Read} \land y.command = \text{Read}))))
\]

The correctness of the new predicate can easily be proved using the definition of \( M_{\text{AcquireLock}} \) (see example 16). In the above approach, we considered the operator \& instead of \&\&(\( \text{bn}, \text{gp}, \text{sl}, \text{cad} \)) whereas the latter is the improved version of the former and regards the modalities of nondeterminism. The extension of our approach to cover modalities of nondeterminism is straightforward, however, the details of such an extension is out of the scope of this paper and is considered as a direction for future research.

Evans has provided several rules to prove liveness properties. Like rules related to safety properties, these rules are based on deterministic operations. To cover nondeterministic operations, we can apply changes to these rules, similar to what we did for the rules related to safety properties: it is sufficient to demonstrate that all (or for some cases, a number of) possible values of nondeterministic variables satisfy the required inference rules. Exploring the details of this idea and applying it to a real application can be an interesting topic in continuing this work. For example, as a case study, we can specify a deadlock free algorithm for controlling
concurrency in a database management system (instead of the strict 2PL method that is deadlock prone) and show the deadlock freedom of transactions, a liveness property of the system.

4. Conclusions and future work

The notion of nondeterminism has been studied in computer science since its early days; the use of this notion in algorithms [14, 24], imperative programming languages [19], and abstract data types [32, 64] are but a few examples. In the field of formal program development, many authors have so far investigated nondeterminism in various refinement calculi. In [51], demonic nondeterminism has been introduced into a refinement calculus by constructing a weakest precondition semantics for imperative specifications and programs that holds good even when expressions may be nondeterministic. In [72], Ward defined a nondeterministic refinement calculus. He first added some (set theoretical) specification constructs to a functional language. Then, he defined a number of nondeterministic constructs in the specification language for modelling erratic, angelic, and demonic choices. Finally, a set of refinement rules were proposed to support refining specifications to nondeterministic functional programs. Most of the existing approaches to deal with nondeterminism in refinement calculi refine nondeterministic specifications by decreasing demonic choices or increasing angelic ones [2, 13, 52]. In [13], however, two methods have been proposed to refine nondeterministic specifications by reducing angelic nondeterminism into demonic one or pure determinism. A survey on studying nondeterminism in refinement calculi can be found in [52].

In this paper, we investigated the notion of nondeterminism in a constructive framework for formal program development. The development of programs in this framework begins by writing specifications in the CZ formal language. CZ is a Z-style notation which benefits from a constructive underlying theory allowing us to investigate the notion of nondeterminism from the formal program development point of view. In the paper, we first showed that without specifying nondeterminism explicitly, the effects of the nondeterminism involved in initial specifications will not be preserved in final programs. We then introduced the notions of multi-schemas and nondeterministic variables into CZ to specify bounded, unbounded, loose, strict, erratic, angelic, demonic, singular, and plural modalities of nondeterminism. Our approach to the angelic and demonic nondeterminism covers game-like situations in which the combination of both modalities of nondeterminism are used to model conflicting agents. For distinguishing singular and plural nondeterminism, we used a semantics that can be implemented in the specification phase. We finally modified the schema calculus operations of CZ to be definable on multi-schemas. Using the new constructs, we can specify usual applications of nondeterminism, such as concurrency, by concentrating only on the functional properties of the system being specified. Also, since the semantics of the new nondeterministic constructs have been given in CZ itself, using the existing interpretation of CZ in Martin-Löf’s theory of types [47, 48], we can derive proper programs from our initial nondeterministic specifications. To continue the current research or improve its results, we introduce the following directions for future work:

(1) There are some attempts to translate Z specifications into specifications in refinement calculi (For example, see [36] and [73]). Since our definitions can be modified slightly to be used in the Z notation, a direction for future
research can be to extend an existing interpretation of $Z$ in a refinement calculus in order to map nondeterministic specifications of $Z$ to their counterparts in the refinement calculus.

(2) In the paper, we have given a solution to translate schemas of CZ into some types in type theory. Our solution is rather primitive and does not cover the schema calculus operations of CZ. By extending the current translation to interpret the schema calculus in Martin-Löf’s theory of types, we obtain an alternative semantics for the schema calculus, based on type theory rather than set theory. In a related work [29], Henson and Reeves have introduced a logic for the schema calculus of $Z$. This logic has been constructed within the specification logic $Z_C$ which is a constructive, typed set theory based upon the notion of schema type. The language of $Z_C$ is associated with

1. rules for determining the types of terms,
2. rules for determining that propositions are well-formed, and
3. rules of inference [30].

Henson and Reeves have also interpreted $Z_C$ in a constructive theory of intensional types and operations called $\tau$. $\tau$ is a generalization of Beeson’s $\xi \ ON$ [4].

(3) Another interesting topic in continuing this research could be studying underspecification in the process of formal program development. When developing a software system in a number of refinement steps, we are often interested in specifying the functionality of the system not uniquely but only with respect to some required properties. Later in the development process, we may add more properties whenever we find this appropriate. Then we speak of underspecification [69]. Underspecification corresponds to choices between deterministic models while nondeterminism corresponds to choices within a nondeterministic model. This conceptual difference between underspecification and nondeterminism does not contradict the fact that both arise as natural abstraction mechanisms and sometimes may be supported by similar modelling techniques. Especially, the refinement concepts for underspecification and nondeterminism are rather similar [69].

In the refinement method, underspecification can be modelled by demonic nondeterminism or indeed demon’s arbitrariness: when a specification specifies less, it may yield a program that satisfies the specification but is still not what is wanted. By the refinement steps, we move from more abstract to more concrete specifications in order to achieve a program that is wanted or remove the chance for developing unwanted programs. This corresponds to the well known treatment of demonic nondeterminism in various refinement calculi [13]: decreasing demonic nondeterminism or demon’s arbitrariness makes a specification more refined.

(4) Another direction for future research is to provide nondeterministic constructs with choice probabilities such that one can associate a selection probability to each choice and probe the effects of these probabilities on extracted programs. Nondeterminism and probability are different, although related, phenomena. In various approaches to model probability in concurrent systems, nondeterministic behavior has been substituted by probabilistic one in which the different types of behaviors are quantified with probabilities [68]. Nevertheless, it is sometimes necessary to provide
operators that allow us to deal with nondeterminism, but, at the same time, taking into account the probabilistic aspects: in the process of modelling a concurrent system, we sometimes find nondeterministic behaviors that can be quantified (for instance, the failing probability of a faulty communication channel), whereas on other occasions, nondeterministic behaviors cannot be quantified (for instance, relative speed between two processes).

However, some problems arise when we want to model computations in which both nondeterminism and probability are present [12]. In [49], it has been shown how to build denotational models to support both nondeterministic choice and probabilistic choice simultaneously, so that the laws for nondeterministic choice and for probabilistic choice that one expects to hold actually are valid. Both notions of probability and nondeterminism have also been studied in sequential programming by several publications of the Programming Research Group at the University of Oxford. For instance, in [45], McIver and Morgan have extended probabilistic predicate transformers, introduced in [37] as a logic for imperative probabilistic programming, to include demonic, angelic, and unbounded nondeterminism.

**References**

Appendix A

Program extraction from \([M_{\text{GetLE}}]^{D}\)

We apply the extended version of the function \(\xi\) to the schema \([M_{\text{GetLE}}]^{D}\) step by step; the result will be a type in Martin-Löf’s theory of types:

\[\|M_{\text{GetLE}}\|^{D}_{\xi} = (\Pi \alpha \in (\xi(N))^{-} \cdot \Sigma \gamma \in (\xi(PN))^{-} \cdot \forall \forall c \in N \cdot \sigma c! \in \text{dsc} \Rightarrow \sigma c! \leq \eta_{\xi}(\xi(N)) \eta_{\xi} / \text{dsc}) \]
\[= (\Pi \alpha \in (\xi(N))^{-} \cdot \Sigma \gamma \in (\xi(PN))^{-} \cdot \Pi \beta \in (\xi(N))^{-} \cdot \forall \forall c \in N \cdot \sigma c! \in \text{dsc} \Rightarrow \sigma c! \leq \eta_{\xi}(\xi(N)) \eta_{\xi} / \text{dsc}) \]

According to the interpretation of the set of natural numbers and the decidable power set constructor in \([47]\) and also the fact that \(\alpha\), \(\beta\), and \(\gamma\) are themselves elements of \(V\), the following equality is obtained:

\[\|M_{\text{GetLE}}\|^{D}_{\eta} = \Pi \alpha \in N \cdot \Sigma \gamma \in N \Rightarrow \text{Bool} \cdot \Pi \beta \in N \cdot \gamma(\beta) = \text{true} \Leftrightarrow \beta \leq \alpha,\]

where \(\beta \leq \alpha\) is an abbreviation for \(\Sigma \beta \cdot \beta + p \geq \alpha\). Now we derive a program from a correctness proof of the resulting type theoretical specification. An initial part of such a proof is shown here:

\[\begin{align*}
A &= \Sigma \gamma \in N \Rightarrow \text{Bool} \cdot \Pi \beta \in N \cdot \gamma(\beta) = \text{true} \Leftrightarrow \beta \leq \alpha \\
B &= \Pi \beta \in N \cdot \gamma(\beta) = \text{true} \Leftrightarrow \beta \leq \alpha \\
\Gamma &= \alpha \in N, x \in N, g \in A[x/\alpha] \\
\text{Hyp} \\
\alpha \in N &\vdash z = (\lambda c. \text{if } c \leq x \text{ then } \text{true else false}) \\
\text{Hyp} \\
\alpha \in N &\vdash z \in (N \Rightarrow \text{Bool}) \\
\alpha \in N &\vdash p \in B[0/\alpha][z/\gamma] \\
\alpha \in N &\vdash u \in A[0/\alpha] \\
\text{Hyp} \\
\alpha \in N &\vdash v = (\lambda c. \text{if } c < s(x) \text{ then } \text{true else false}) \\
\text{Hyp} \\
\alpha \in N &\vdash v \in (N \Rightarrow \text{Bool}) \\
\alpha \in N &\vdash g \in A[s(x)/\alpha] \\
\text{Hyp} \\
\alpha \in N, x \in N, g \in A[x/\alpha] &\vdash g \in A[s(x)/\alpha] \\
\text{Hyp} \\
\alpha \in N, x \in N &\vdash g \in A[x/\alpha] \\
\text{Hyp} \\
\alpha \in N &\vdash f \in (\Pi x \in N \cdot A[x/\alpha] \Rightarrow A[s(x)/\alpha]) \\
\text{Hyp} \\
\alpha \in N &\vdash t \in A \\
lGetLE \in (\Pi \alpha \in N \cdot \Sigma \gamma \in N \Rightarrow \text{Bool} \cdot \Pi \beta \in N \cdot \gamma(\beta) = \text{true} \Leftrightarrow \beta \leq \alpha)
\end{align*}\]

Figure 7. Program extraction from the operation schema \([M_{\text{GetLE}}]^{D}\)
Appendix B
Program extraction from $[M_{\text{GetLE}}]^{BD}$

We apply the extended version of $\xi$ to the schema $[M_{\text{GetLE}}]^{BD}$ step by step; the result will be a type in Martin-Löf’s theory of types:

$[[M_{\text{GetLE}}]^{BD}]_{\xi} = (\Pi \alpha \in (\xi(N))^{-} \cdot \Sigma \gamma \in (\xi(seq N))^{-} \cdot \forall sc! \in N \cdot sc! \in dsc \iff sc! \leq n^2_{\xi}(\xi(seq N)(\alpha/\gamma[/dsc])) \cdot \forall \beta \in (\xi(N))^{-} \cdot (\Pi \gamma \in (\xi(seq N))^{-} \cdot \Sigma \gamma \in (\xi(seq N))^{-} \cdot \forall \beta \in (\xi(N))^{-} \cdot \forall sc! \in dsc \iff sc! \leq n^2_{\xi}(\xi(seq N)(\alpha/\gamma[/dsc])))(\xi(seq N)(\beta/\gamma[/dsc]))$

According to the interpretation of the set of natural numbers and sequences in [47] and also the fact that $\alpha$, $\beta$, and $\gamma$ are themselves elements of $V$, the following equality is obtained:

$[[M_{\text{GetLE}}]^{BD}]_{\xi} = \Pi \alpha \in N \cdot \Sigma \gamma \in List(N) \cdot \Pi \beta \in N \cdot \beta \in \gamma \iff \beta \leq \alpha$, where $\beta \leq \alpha$ is an abbreviation for $\Sigma \rho \cdot \beta + \rho = \alpha$. Now we derive a program from a correctness proof of the resulting type theoretical specification. An initial part of such a proof is shown here:

$$
\begin{align*}
A & = \Sigma \gamma \in List(N) \cdot \Pi \beta \in N \cdot \beta \in \gamma \iff \beta \leq \alpha \\
B & = \Pi \beta \in N \cdot \beta \in \gamma \iff \beta \leq \alpha \\
\Gamma & = \alpha \in N, x \in N, g \in A[x/\alpha] \\
\hline
\begin{array}{l}
\text{Hyp} \\
\alpha \in N \vdash z = (0) \\
\alpha \in N \vdash u \in A[0/\alpha] \\
\alpha \in N, x \in N, g \in A[x/\alpha] \vdash \text{getLE} = \lambda \alpha \cdot \text{natrec}(u, u, f) \\
\end{array}
\end{align*}
$$

$\text{Hyp} \\
\alpha \in N \vdash \text{head}(v) = s(x) \\
\alpha \in N \vdash \text{tail}(v) = \text{fst}(g)$

$\sum i \\
\alpha \in N, x \in N, g \in A[x/\alpha] \vdash g \cdot (x, q) \Rightarrow (s(x), \text{fst}(g))$

$\Pi i \\
\alpha \in N \vdash t \in A$

$\text{getLE} \in (\Pi \alpha \in N \cdot \Sigma \gamma \in List(N) \cdot \Pi \beta \in N \cdot \beta \in \gamma \iff \beta \leq \alpha)$

Figure 8. Program extraction from the operation schema $[M_{\text{GetLE}}]^{BD}$
Appendix C
Program extraction from \([\text{M}_\text{Sort}]^{BD}\)

We use the extended version of the function \(\xi\) to translate the operation schema \([\text{M}_\text{Sort}]^{BD}\). As in example 6, for convenience, we do not translate two schemas \textit{Increasing} and \textit{Perm} which are included in the schema \([\text{M}_\text{Sort}]^{BD}\).

\[
[[\text{M}_\text{Sort}]^{BD}]_{\xi} = \Pi \alpha \in (\xi(seq(N \times N)))^{-} \cdot \Sigma \gamma \in (\xi(seq seq(N \times N)))^{-}.
(\forall \text{out!} \in \text{seq}(N \times N) \cdot \text{out!} \in \text{dout} \Leftrightarrow (\text{out!} \in \text{increasing} \land (\text{in?}, \text{out!}) \in \text{perm}))_{\xi}
\]

By the interpretation of the set of natural numbers, the cartesian product constructor, and sequences in [47] and also the fact that \(\alpha, \beta,\) and \(\gamma\) are themselves elements of \(V\), the following equality holds:

\[
[[\text{M}_\text{Sort}]^{BD}]_{\xi} = \Pi \alpha \in \text{List}(N \otimes N) \cdot \Sigma \gamma \in \text{List}(\text{List}(N \otimes N)) \cdot \Pi \beta \in \text{List}(N \otimes N) \cdot 
\beta \in \gamma \Leftrightarrow (\beta \in \xi(\text{increasing}) \otimes (\alpha, \beta) \in \xi(\text{perm}))
\]

We can now derive a program from a correctness proof of the resulting type theoretical specification. An initial part of such a proof is shown in Fig. 9. In the proof tree, we have used two conventions \(IN = \xi(\text{increasing})\) and \(P = \xi(\text{perm})\).
Figure 9. Program extraction from the multi-schema $M_{Sort}$.
Appendix D  
Program extraction from \([[[M_{OurPlayer}]^{cad}]^{st}]^{bu}\)

If we apply the extended version of the function \(\xi\) to the schema \([[[M_{OurPlayer}]^{cad}]^{st}]^{bu}\), the following specification will be obtained in Martin-Löf’s theory of types:

\[
\Pi \alpha \in \text{List}(T') \cdot (I[\text{List}(T'), \alpha, ()] \Rightarrow \Omega) \Rightarrow \Sigma \gamma \in T' \cdot \gamma \hat{\epsilon} \alpha \odot (\Pi \beta \in T' \cdot \beta \hat{\epsilon} \alpha \Rightarrow \text{sco}(\beta) \leq \text{sco}(\gamma)),
\]

where \(T' = (\xi(T))^{-}\), and \(b \leq a\) is an abbreviation for \(\Sigma c \cdot b + c = a\). We can now derive a program from a correctness proof of the resulting specification. An initial part of such a proof is shown in Fig. 10. A complete proof similar to the one required here is given in [27].

A simplified form of the extracted program is:

\[
\text{ourplayer} = \lambda \alpha \cdot \text{lrec}(\alpha, \text{abort}, \\
\lambda a \cdot \lambda x \cdot \lambda h_1 \cdot (\text{if } \text{fst}(h_1) = \text{abort} \text{ then } a \text{ else if } \text{sco}(a) \geq \text{sco}(\text{fst}(h_1)) \text{ then } a \text{ else } \text{fst}(h_1), q))
\]

This program is a recursive function that selects an element with the highest score from a non-empty sequence. For the empty sequence, the program aborts:

\[
\text{ourplayer}(\langle\rangle) = \text{abort}
\]
\[
\text{ourplayer}(h :: t) = \text{if } \text{ourplayer}(t) = \text{abort} \text{ then } h \\
\text{else if } \text{sco}(h) \geq \text{sco}(\text{ourplayer}(t)) \text{ then } h \text{ else } \text{ourplayer}(t)
\]

Software Quality Research Laboratory, Computer Engineering Department, Sharif University of Technology, Tehran, Iran

E-mail address: haghighi@ce.sharif.edu, mirian@sharif.edu
Figure 10. Program extraction from the multi-schema $M_{\text{OurPlayer}}$