KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

# Formalizing a Modelling Language for the Financial Industry

## The Rosetta language

Simon Cockx

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen

**Promotor:**
Prof. dr. ir. Tom Schrijvers

**Assessoren:**
Prof. dr. Raf Vandebril
Ir. Roger Bosman

**Begeleider:**
Dr. ir. Alexander Vandenbroucke

Academiejaar 2021 – 2022

# Preface

Finding a suitable subject for my thesis was a long and stressful task, so it was a relief to actually find a topic that provoked both my passion for mathematics and for computer science. Tom, thank you for finally guiding me to this topic after months of uncertainty.

Thank you to my supervisors Alexander and Tom for giving me the opportunity to work on this interesting topic, and for supporting me when changing the subject of my thesis to the Rosetta language itself. The valuable advice you gave during our numerous meetings always steered me in the right direction.

My gratitude goes to Minesh, Hugo and the rest of the Rosetta team for meeting with me and bringing helpful discussions to the table about the goals and design of Rosetta.

Persevering would have been impossible without my Chiro, providing me with a constant relieve of stress. A big thank you to all the wonderful people there for bringing me joy and purpose.

The pandemic has been going on for two years now. A special thanks to my brother Robbe for keeping me (in)sane, even during the most drastic lockdowns.

Thank you to my father and my brother Jesper for proofreading a preliminary version of this text and giving me valuable feedback.

Finally, thanks to both of my parents and all of my brothers for the love and support you gave while working on this thesis, and before. You are my cornerstone.

*Simon Cockx*

# Contents

# Abstract

The Rosetta language aims to become a successful, new standard to model business data and business logic across the financial industry. Such an ambitious language better be built on solid foundations. However, the lack thereof currently leads to a number of issues.

First, to fulfil its goal of modelling both data and logic, Rosetta employs an atypical combination of features from modelling languages, such as entities and cardinality, and features from programming languages, such as functions and types. The former—that expressions can have plural cardinality—strongly influences the latter: operations must take this into account, and ideally the type system is able to watch over the correct usage of cardinality. Yet, Rosetta lacks any kind of formal specification, leading to ambiguities in the semantics of the language, and the type system does not include cardinality checks, increasing the potential for runtime errors in Rosetta models.

Second, Rosetta attempts to increase its adoptability among financial companies by providing code generators to multiple target languages, such as Java, Scala and TypeScript. This is, however, where its ill-definedness becomes especially problematic: each code generator implements its own interpretation of Rosetta, leading to a divergence between the semantics of generated code in different languages. This undermines Rosetta's goal to consolidate market standards and operational practices within the financial industry.

This thesis provides a full formal specification of Rosetta's core features: entities, functions, expressions and types. Based on a novel interpretation of expressions with cardinality, we address the found issues with a stronger type system that includes cardinality constraints, adjustments to the syntax, and an exact description of the meaning of expressions. Additionally, we describe a systematic approach to translate our formal semantics into an actual code generator using Java as an example, where we focus on preserving well-typedness of generated code.

# Samenvatting

De taal Rosetta streeft ernaar om een nieuwe, succesvolle standaard te worden voor het modelleren van data en operaties binnen de financiële industrie. Een taal met zulke ambities kan maar beter een stevig fundament hebben. Het gebrek daaraan leidt momenteel echter tot problemen.

Ten eerste bestaat Rosetta uit een atypische combinatie van constructies uit modelleertalen (denk aan entiteiten en kardinaliteit) en constructies uit programmeertalen (denk aan functies en types). Het feit dat expressies een meervoudige kardinaliteit kunnen hebben, heeft echter een sterke invloed op de semantiek van functies: elke operatie moet hier rekening mee houden, en het typesysteem moet idealiter in staat zijn om een juist gebruik van kardinaliteit te garanderen. Rosetta heeft echter geen formele specificatie, wat leidt tot ambiguïteiten in de semantiek van de taal, en het typesysteem controleert niet op kardinaliteit, wat het potentieel voor fouten in Rosetta-modellen verhoogt.

Ten tweede probeert Rosetta zich als standaard gemakkelijk te verspreiden door code-generatoren aan te bieden naar verschillende doeltalen, zoals Java, Scala en TypeScript. Dit is echter waar het gebrek aan een formele specificatie nog problematischer wordt: elke code-generator implementeert zijn eigen interpretatie van Rosetta, dus de semantiek van gegenereerde code in verschillende talen kan verschillen. Dit ondermijnt Rosetta's doel om de uiteenlopende marktstandaarden en operationele praktijken binnen de financiële industrie te unificeren.

Deze thesis definieert een volledige formele specificatie van de kern van Rosetta: entiteiten, functies, expressies en types. Met een nieuwe kijk op expressies met kardinaliteit pakken we de gevonden problemen aan met een sterker typesysteem dat kardinaliteitscontroles bevat, aanpassingen aan de syntaxis, en een precieze beschrijving voor de betekenis van expressies. Daarnaast beschrijven we een systematische aanpak om onze formele semantiek te vertalen naar een code-generator, waarbij we focussen op hoe we kunnen garanderen dat gegenereerde expressies goed-getypeerd zijn. Ter voorbeeld beschrijven we een volledige code-generator naar Java.

# Chapter 1

# Introduction

In this age of digitalization, the financial industry relies more and more on machine-readable models for representing data and operational practices. Currently there exists a wide variety of modelling standards across different organizations—standards such as FIX[1], FpML[2] and EFET[3], supplemented by even more dialects of those. Unfortunately, this variety impedes inter-operability and processing across firms, because each firm involved needs to reconcile their model representation with the other variations in use. Moreover, this multitude of data representations hinders financial technology innovation as there is no common foundation for technologies such as distributed ledgers, smart contracts, cloud computing, and artificial intelligence. Lastly, it complicates regulatory oversight, making it difficult for regulators to demand consistent and uniform regulatory reporting from market participants.

To address these problems, the International Swaps and Derivatives Association (ISDA [4]) launched a project to consolidate these market standards into one Common Domain Model (CDM). In order for this new standard to become successful, it would need to conform to three non-functional requirements.

1. **The model needs to be machine-readable.** This requirement is of course necessary for any digital standard.

2. **The model needs to be human-readable.** Domain specialists must be able to understand the model by reading it, even if they have no technical background.

3. **The model needs to be easily adoptable.** A company that wants to adopt it, should not need to completely recreate its digital infrastructure. For example, the model should be integrable both for companies that currently use Java and companies that currently use Haskell.

To fulfil these requirements, work began on a new domain-specific language (DSL) called Rosetta [5]. Rosetta aims to fully support the implementation of the CDM, ranging from its data and workflow representations to regulatory reporting rules. Moreover, it intends to facilitate the adoption of this model into the implementation stack of any organization by providing automatic translation to executable code

in different languages. This way it can meet the need for a common, readable data representation, a machine-executable workflow definition and a standard for regulatory reporting.

## 1.1 Goal of this thesis

Rosetta has several peculiar features that are uncommon among DSLs and programming languages in general. The most significant one might be that each attribute of a data type does not only have an associated type, which describes the possible values that the attribute can have, but also a cardinality constraint, which describes the possible number of values of the attribute. Such a feature strongly influences the semantics and type system of the language: each operation should take into account that parameters can have a plural cardinality, and the type system preferably needs to give additional static guarantees concerning the correct usage of cardinality. However, in the current design of Rosetta it is often not clear how operations should handle plural cardinality. Additionally, the type system does not include cardinality checks, which allows the programmer to write erroneous code that will not be detected until runtime. Other uncommon features include an unusual syntax for defining output of functions and permitting delegating the implementation of a function to a later stage of development, i.e., after code generation. Yet, the semantics of both of these features are not entirely clear.

Despite the fact that Rosetta has features one would consider unusual and for whose semantics we therefore have no clear intuition, it still lacks a precise specification, formal or informal, other than its implementation. This ill-definedness becomes especially problematic during the development of a code generator, in particular because Rosetta aims to support code generators to many different languages. The absence of a standard can lead to a divergence between the semantics of generated code, because each code generator implements its own interpretation. This undermines the goal of Rosetta to consolidate market standards and operational practices across the financial industry. The problem grows with the number of supported generators.

Moreover, the Java code generator of Rosetta contains some problems that cause the generated code to be ill-typed and thus useless. In general, implementing semantics of a language by means of code generation becomes more challenging depending on how much the target language (e.g., Java) differs from the source language (i.e., Rosetta). The challenge here is to make sure that the semantics of a Rosetta model corresponds to the semantics of the code in the target language generated from that model. In some cases, this can be easy, for example when a type in the DSL corresponds to a type in the target language, e.g., `int` in Rosetta might correspond to `Integer` in Java and `number` might correspond to `BigDecimal`. For other language constructs, this becomes more error-prone, e.g., in Rosetta `int` is considered a subtype of `number`, but `Integer` is not considered a subtype of `BigDecimal` in Java. A naive code generator might therefore generate code that is not well-typed and consequently useless.

The main goal of this thesis is to improve the foundation of the Rosetta language. We address the found issues with a stronger type system that includes cardinality constraints, adjustments to the syntax, and an exact description of the meaning of expressions, all unified in a full formal specification for a subset of the language. Here we focus explicitly on the core features of Rosetta—entities, functions, expressions and types. Additionally, we describe a systematic approach to preserve well-typedness while implementing a code generator, which tackles problems caused by a discrepancy between the subtype relation of Rosetta and the subtype relation of a target language.

To the best of our knowledge, a type system that models cardinality constraints has never been described before in scientific literature. Although there exist stronger type systems in which we can embed this behaviour—think of dependently typed systems such as Agda[6] or type systems with advanced tuple types such as TypeScript[7]—the benefit of modelling this explicitly is that we retain a relatively simple type system. Besides its practical contribution, this thesis therefore also makes a contribution of theoretical interest.

## 1.2 Related work

Most competing standards of the CDM such as FIX[1], FpML[2] and EFET[3] are XML based, which make them independent of any specific programming language and easy to integrate in a company's application. They are implemented as an XML schema with additional tools and protocols for processing and communication. Their focus is on representing data and ways of communicating that data with message protocols (e.g., to enforce non-repudiation), but they are short of a representation for operational practices, which Rosetta solves by introducing functions. This is one of features which makes Rosetta stand out. Additionally, XML based standards lack the readability of a custom DSL such as Rosetta.

The methods for formal language design used throughout this thesis are based on the accumulation of research from the last five decades. The theory about type systems discussed in the great book of "Types and Programming Languages"[8] of Benjamin C. Pierce certainly leaves its marks. The formal semantics are largely based on the book "Denotational semantics: a methodology for language development"[9] of David A. Schmidt.

## 1.3 Overview

**Chapter 2** gives a short introduction to the core features of Rosetta—entities, functions, expressions and types—and identifies its current issues. This acts as a motivation for the next chapters.

**Chapter 3** introduces the necessary tools for formal language development. Using a BNF notation for syntax, inference rules for the type system, and denotations for defining the meaning of expressions, it shows how we can build every part of a formal language and applies this to a simple example language.

**Chapter 4** introduces a dialect of Rosetta—or rather of its core features—called Nouga and formally specifies its syntax. It also gives an informal description of the meaning of all operations in Nouga. Lastly, it describes a useful interpretation of the language, which forms the basis on which the following chapters build.

**Chapter 5** explains how we can integrate cardinality constraints into a type system. After explaining Nouga's typing rules, it translates these rules into an *algorithmic* version; one that is suitable for an actual implementation.

**Chapter 6** specifies the meaning of language constructs in Nouga. It does so in a way that is independent of any target language for code generation by mapping expressions and their types into a mathematical domain.

**Chapter 7** describes a systematic approach to convert the formal semantics of the previous chapter into an actual code generator using Java as an example. It especially pays attention to how well-typedness can be preserved in generated code, even if there is a discrepancy between the subtype relation of Nouga and the one of the target language.

**Chapter 8** evaluates our proposed solution by writing a real-life combinatorial model for financial contracts in Nouga. It discusses the issues that Nouga solves, but also gives some examples of its limitations. A possible workaround for these limitations is discussed too.

**Appendix A** shows the full specification of the typing rules of Nouga, both the declarative and the algorithmic ones.

**Appendix B** shows the full specification of the semantics of expressions and types in Nouga.

# Chapter 2

# Rosetta

This chapter explains Rosetta's main features by example, and shows the problems that Rosetta's current implementation suffers from. The first three sections describe the constructs used to model entities, functions and expressions. The last section demonstrates the problems present in Rosetta, which motivates the improvements we develop in the subsequent chapters.

Note that Rosetta includes many more features—such as constructs for regulatory reporting—that are not considered here to keep the focus on the core features of Rosetta. Additional features could be added as extensions in future work without influencing the core that this thesis discusses.

## 2.1 Entities

First of all, Rosetta is designed to be used as a modelling language. One of its main features is to describe real-world concepts, or *entities*. For example, the CDM uses this to model products, contracts and transactions, although it can be used to describe concepts outside of the financial industry as well.

As an example, let us describe a candy factory. A candy factory has a list of candy recipes it can use to produce candy. Each recipe consists of a list of ingredients and an end product. To describe possible types of candy and ingredients, we can *enumerate* them. In Rosetta, such an enumeration starts with a keyword `enum` followed by a name. It then lists all possible values of the enumeration.

```
enum CandyEnum:
  TAFFY
  NOUGAT
  MARZIPAN

enum IngredientEnum:
  SUGAR
  HONEY
  ALMOND
```

The convention is to end the name of an enumeration with Enum.

With these definitions, we can now define a recipe as follows.

```
type Recipe:
  product CandyEnum (1..1)
  ingredients IngredientEnum (2..*)
```

The keyword type is used to start a new entity definition, which in this case we call Recipe. We then describe the attributes of this entity, namely product and ingredients. Each attribute is described by three parts: a name, a type and a cardinality.

Consider the attribute named product. It has a type CandyEnum, which means that the product of a Recipe can be any candy as defined earlier in the enumeration. Furthermore, it has a cardinality constraint (1..1) (read as "from one to one"), meaning that a recipe has *exactly one* product, not less or more.

Likewise, the attribute named ingredients can hold any ingredient as defined in the enumeration of IngredientEnum. Unlike product however, this attribute has a cardinality constraint of (2..∗) (read as "from two to infinity"), meaning that a recipe has *at least* two ingredients, and can have as many ingredients as necessary. We can interpret attributes with plural cardinality as a list of values of its type.

A candy factory then has one or more recipes it can produce.

```
type CandyFactory:
  recipes Recipe (1..*)
```

Note that in this case the type of recipes refers to another entity, namely Recipe, whereas the types in previous snippet referred to enumerations. Besides enumerations and entities, types can also refer to built-in types: among others, `int` for integers (e.g., 42, 0 and −7), `number` for fractional numbers (e.g., 3.14, 4.0 and −2.6), and `boolean` for boolean values (i.e., `True` and `False`).

Rosetta also supports a simple form of *inheritance*. When an entity *extends* another entity, it inherits all of its attributes. We will see an example of this in Chapter 4.

Entities in Rosetta support many more features, such as `conditions` with which we can describe invariant properties about an entity's attributes, but the next chapters in this thesis ignore them to focus on Rosetta's core features.

## 2.2 Functions

A second important feature of Rosetta, and one with a great impact on its complexity, is a way to define operations on entities, called *functions*, which lifts Rosetta from a simple modelling language to an actual programming language. These operations can perform calculations, check conditions, or transform entities into other entities. For example, the CDM uses this to calculate return rates and equity performance, although it can be used to model operations outside of the financial context too.

Let us return to the sweetness of our example model for candy factories. Suppose we have a sweet tooth, and we are not satisfied with the amount of sugar that is

added in the current recipes. We can define an operation that transforms a recipe into one that has more sugar in it. The resulting candy product remains the same, but we add SUGAR to the list of ingredients.

```
func AddSugar:
  inputs: recipe Recipe (1..1)
  output: result Recipe (1..1)

  assign-output result->product:
    recipe->product
  assign-output result->ingredients:
    [recipe->ingredients, IngredientEnum->SUGAR]
```

The keyword `func` followed by the name AddSugar starts the definition of our transformation. We then describe the inputs of the operation; in this case a single Recipe which we call recipe. The output is again a single Recipe which we call result. Lastly, we specify what the result looks like using an `assign-output` statement for each attribute of the output result. The first specifies that the product of the result is the same as the product of the input recipe, while the second specifies that the list of ingredients of the result is the same as that of the input recipe, but with SUGAR added to the end.

In Rosetta, the `assign-output` statements are optional. When none of these statements are given, the function is considered *abstract* and the concrete implementation of the function is delegated to a later stage of development, i.e., after code generation. When such a function is translated to e.g., Java, there will be an abstract class with a method representing this function. A Java developer can then provide an implementation by extending the class and implementing the method.

## 2.3 Expressions

The previous section states that functions have a great impact on the complexity of Rosetta, and this complexity comes largely from the expressions that can be used when defining the output of a function. The example above contains two expressions: recipe -> product returns the value of the product attribute of recipe, and [recipe -> ingredients, IngredientEnum -> SUGAR] adds SUGAR to the list recipe -> ingredients. An example of one of Rosetta's more peculiar operators is `only exists`. This operator checks for a given attribute whether it is the only non-empty attribute of an entity. For example, consider an entity representing a client of one of our candy stores.

```
type Client:
  favoriteCandies CandyEnum (0..*)
  favoriteNumber int (0..1)
  favoriteDay date (0..1)
```

Given a client `c`, the expression `c -> favoriteNumber only exists` checks whether `c` has a favorite number and does not have any favorite candies or a favorite day.

The list of possible expressions is, however, much larger. For example, we can:

- perform calculations on numbers using arithmetic operations (`+`, `-`, `*`, `/`),
- perform numerous checks on lists such as equality and the existence of one or more items (e.g., the `exists` operation checks whether its argument is at least of singular cardinality),
- count the number of items in a list,
- combine checks using boolean operations, and
- call other functions.

In fact, all these constructs can be combined with each other to form even more complex expressions. Chapter 4 gives a full overview of all possible expressions.

This flexibility gives power to the users of Rosetta, but also makes it more difficult not to make a mistake, as it becomes increasingly easy to write expressions that are syntactically correct but are semantic nonsense. Luckily, some mistakes are easy to detect automatically by means of a type system. For example, suppose we have a CandyFactory named factory, and we would wrongfully try to add sugar to it, i.e., AddSugar(factory), then Rosetta will detect this error, as it sees that the AddSugar function expects a recipe, but something of type CandyFactory was given. In general, Rosetta can compute the type of any expression, and immediately give feedback to the user if it does not match the type expected by the context where it is used.

## 2.4 Shortcomings of Rosetta

This section demonstrates the issues that were identified in Rosetta. Note that some of these issues might have been mitigated during the writing of this thesis, although the majority still stands.

### 2.4.1 Problems with the type system

**No cardinality checks.** Most problems in Rosetta are caused by a weak type system or the lack of a precise specification of its semantics. Often, they are actually caused by a combination of both. To demonstrate this, let us try to add two lists of numbers together: $[1, 2] + [3, 4]$. This expression is, in fact, allowed by the type system, but there is no clear meaning. Should this concatenate the lists, resulting in $[1, 2, 3, 4]$? Or should the numbers be added pairwise, resulting in $[4, 6]$? And what about this expression?

```
3 + [5, 8] - [0, 1, 2] * [] / [101, 90]
```

When we evaluate the Java code generated from these expressions, they both result in null, which is both surprising and unuseful. A more sensible possibility is to disallow these expressions altogether by defining typing rules that only allow arithmetic

operations on expressions with a singular cardinality. A similar problem is present for boolean operators and conditional expressions.

As another example, suppose that in our definition of the AddSugar function, we had forgotten to add the ingredients of the input recipe, i.e.,

```
func AddSugar:
  ...
  assign-output result->ingredients:
    [IngredientEnum->SUGAR]
```

Instead of adding sugar to the recipe, we are creating a recipe that only contains sugar! This is not only not what we intended, but also a violation of our model: the ingredients attribute has a cardinality constraint of $(2..*)$, so it should at least have *two* ingredients, whereas the recipe we define here only has one. However, the type system of Rosetta does not report this, as it does not include any cardinality checks. This causes many models to pass the type checker that are actually invalid.

Note that the two examples above contain bugs that are still easy to spot. These are, of course, toy examples. Imagine, however, that we want to add two expressions together that both consist of some complicated combinations of function calls and other operations instead. In such more realistic scenarios detecting a mistake becomes less easy, and it is in those cases that a well-designed type system can shine.

**Weak subtyping.** Rosetta supports a limited form of subtyping, which allows us to use a child entity in a context where a parent entity is expected. For example, consider the following hierarchy of dummy entities, together with abstract functions that return such entities.

```
type Parent:
type Child1 extends Parent:
type Child2 extends Parent:

func CreateParent:
  output: result Parent (1..1)
func CreateChild1:
  output: result Child1 (1..1)
func CreateChild2:
  output: result Child2 (1..1)
```

By taking advantage of Rosetta's subtyping, we may create a list with mixed types, e.g., [CreateChild1(), CreateParent()]. However, for no clear reason, mixing sibling entities is not allowed, e.g., [CreateChild1(), CreateChild2()] results in a type error, although these expressions have a common supertype Parent and could be typed as such. Allowing such expressions would give a user more freedom with only a minimal increase in complexity of the language, since it already supports a form of subtyping.

**Incorrect type for the `empty` literal.** In Rosetta, the literal `empty` represents an empty value, which should be assignable to any attribute with a cardinality

constraint that includes zero, no matter its type. However, Rosetta assigns it the type `any`, which is incompatible with all other types (`any` is the top type in Rosetta: all types are subtypes of `any`, not the other way around). Consider a function that takes two optional integers and returns the one that is not empty. If both or none are empty, the result is also empty.

```
func OneOfTwo:
  inputs:
    a int (0..1)
    b int (0..1)
  output: result int (0..1)
  assign-output result:
    if a exists and b exists then
      empty
    else if a exists then
      a
    else
      b
```

Although this function is semantically correct, Rosetta's type system wrongly rejects this model because the type `any` is not a subtype of `int`.

### 2.4.2 Problems with the syntax

Although the design of a syntax in itself cannot be formally wrong, it can have great consequences for the complexity of the semantics of a language. To see this, we first extend our candy example model with *quantities*, such as "2 kilograms" or "0.5 litres". An obvious way to do this would be to add an enumeration for units (e.g., kilograms and litres), and define an entity Quantity that has two attributes; one number for the scale and another one for the unit. However, for the sake of explaining the problem, we make an (admittedly, over-engineered) combinatorial model.

We split the concept of a quantity over three entities: Zero representing no quantity at all, One representing one unit of a given metric unit (e.g., "1 kilogram" or "1 litre"), and Scaled representing a quantity scaled with some number. A quantity can then be seen as a *tagged union* of these three entities, which we can represent as an entity that can have any of these three entities as attribute.

```
enum UnitEnum:
  KILOGRAMS
  LITRES
```

```
type Zero:
type One:
  unit UnitEnum (1..1)
type Scaled:
  quantity Quantity (1..1)
  scale number (1..1)

type Quantity:
  zero Zero (0..1)
  one One (0..1)
  scaled Scaled (0..1)

  condition: one-of
```

We use a `condition: one-of` to stipulate that one of the attributes of Quantity must be non-empty, and all the others must be empty. In that sense, a quantity can be either a Zero quantity, a One quantity, or a Scaled quantity. Note that Zero has no attributes at all.

Someone familiar with Haskell might see similarities with algebraic data types and their constructors, e.g.,

```
data Unit = Kilograms | Litres
data Quantity = Zero | One Unit | Scaled Quantity Double
```

Entities such as Quantity that have attributes all of which might be empty, are actually quite common in Rosetta models. One interesting operation to perform on an instance of such an entity is the `only exists` operator. Here is a function that checks whether the one attribute of a quantity is the only attribute that exists.

```
func IsOne:
  inputs: q Quantity (1..1)
  output: result boolean (1..1)
  assign-output result:
    q -> one only exists
```

This is a perfectly valid use case of the `only exists` operator.

Here comes the problem: the `only exists` operator may also act on expressions that do not end with a projection `->`. For example, let us rewrite the IsOne function as follows.

```
func IsOne:
  inputs: o One (1..1)
  output: result boolean (1..1)
  assign-output result:
    o only exists
```

Given a quantity q, we can now write IsOne(q -> one) to check that one is the only non-empty attribute of q. While it is possible for Rosetta to support the `only exists` operation in this way, it greatly complicates the semantics of the whole language,

because each runtime value will need to keep track of the entity from which it came. Moreover, not all values have an instance they belong to, creating a possibility for more runtime errors such as passing an instance of One to IsOne that did not belong to any Quantity instance. A more sensible choice is to stipulate that the argument of the `only exists` operator must always end with a projection like in the first implementation of IsOne.

A different kind of problem is caused by Rosetta's peculiar `assign-output` syntax. As demonstrated in Section 2.2, the output of a function is defined by specifying the result for each attribute of the output. When calling such a function, an entity is implicitly instantiated. For example, a constructor for the entity Scaled could look like this.

```
func CreateScaled:
  inputs:
    q Quantity (1..1)
    s number (1..1)
  output: result Scaled (1..1)
  assign-output result->quantity: q
  assign-output result->scale: s
```

Calling the function CreateScaled instantiates Scaled with attributes `q` and `s`. In one particular case however, this implicit instantiation is problematic, namely when the resulting entity has no attributes at all, e.g., for the entity Zero. In this case, there is no way to instantiate Zero in Rosetta itself, as a function without any `assign-output` statements is considered abstract.

### 2.4.3 Problems with the semantics

The main problem with the lack of a semantic specification is that developers working on a code generator need to make their own interpretation of the meaning of language constructs. This is especially problematic because Rosetta aims to support code generators for many different languages. If developers have a different interpretation of the semantics, code generated to Java might behave differently from code generated to Scala. To pick one example: the current Java generator represents the type `number` as a 128 bit decimal number, whereas the current TypeScript generator represents this type as a 64 bit binary number, resulting in different semantics.

### 2.4.4 Problems with the Java code generator

One property we would like to have is that if a Rosetta model is well-typed, then the generated Java code should be well-typed too; otherwise the Java code would not compile and the generated model is useless. However, this is not always the case. One important situation where this problem occurs is when an `int` is used in a context where a `number` is expected. Because `int` is a subtype of `number` in Rosetta, the following model is valid.

```
type SomeEntity:
  r number (1..1)

func CreateSomeEntity:
  inputs: n int (1..1)
  output: result SomeEntity (1..1)
  assign-output result->r: n
```

Note that the attribute r of SomeEntity is of type `number`, but we assign an integer n to it. When translated to Java, the choice was made to represent an `int` with the Java type Integer and a `number` with the Java type BigDecimal. In Java, however, Integer is *not* a subtype of BigDecimal. As such, a naive code generator would try to construct SomeEntity by setting the the r attribute to the integer n, while a BigDecimal is expected, resulting in a type error in Java.

Important to note is that the decision to represent an `int` with Integer and a `number` with BigDecimal is itself *not* the mistake. Failing to generate code that takes this discrepancy between the subtype relations into account, however, is.

# Chapter 3

# Language Design

Formal methods for language design aim to prevent mistakes and improve the quality of modelling languages and programming languages. A formal language is usually defined in the following three distinct parts.

1. **The syntax** refers to the rules that define how symbols can be combined to form correctly structured statements and expressions in the language.
2. **The type system** refers to rules that state when such a syntactical construct is valid. It can statically analyze code written in the language, and inform the user when they write code that does not make sense, e.g., when dividing a number by a string. The more complex a language is, the easier it is to write erroneous code, and it is the type system's purpose to reduce the number of bugs that users of the language can write.
3. **The semantics** is concerned with the meaning of well-typed constructs of the language. It does so by specifying how language constructs can be evaluated in a mathematically precise way.

This chapter introduces the necessary tools to describe each of these parts using a simple expression language as example. It starts by explaining the intuition of the language, after which it formalizes the syntax, type system and semantics in that order.

## 3.1 A simple expression language

Our example language contains four kinds of syntactic constructs: arithmetic expressions (e.g., $3 + 5 * 2$), boolean expressions (e.g., `true or false`, $3 < 5$), conditional expressions (e.g., `if` $3 < 5$ `then` $42$ `else` $0$) and assignments (e.g., $x := 1 + 2;$). A program starts with a sequence of assignments and ends with a single expression representing the end result of the program, e.g.,

```
x := 1 + 2;
y := 3*x;
c := (x + 5) < y;
z := y * (if c then x else y);
y + x*z
```

In this program, `x` evaluates to the integer 3, `y` to the integer 9, `c` to the boolean value *true*, `z` to the integer 27, and the final expression to the integer 80.

We will also allow conditional expression without "else" branch. An expression of the form `if` *condition* `then` *result* is syntactic sugar for `if` *condition* `then` *result* `else` 0.

## 3.2 Syntax and Backus-Naur form

Backus-Naur form (BNF) is a widely-known notation used typically for defining the syntax of formal languages. Many variants exist, so what follows is a brief recapitulation of the main concepts and how we will use them, based on the BNF grammar for our expression language. (see Spec. 3.1)

We define four *non-terminals*, namely ⟨PROG⟩, ⟨ASSIGN⟩, ⟨EXPR⟩ and ⟨LIT⟩. The first non-terminal, which is implicitly assumed to be the start symbol, formalizes that a program consists of any number of assignments (note the star operator $*$) followed by a single expression. The second states that an assignment starts with a valid variable name, indicated with the symbol $x$, followed by the terminal `:=`, an expression and the terminal symbol `;`. The third, ⟨EXPR⟩, lists all possible types of expressions (note the operator ? for the optional "else" branch). The fourth and last shows that literals are either the terminals `true` or `false`, or an integer, indicated with the symbol $i$. We ignore any whitespace: terminal and non-terminal symbols may be separated by any number of spaces, newlines, tabs, and so on.

SPECIFICATION 3.1: formal syntax definition of our expression language.

⟨PROG⟩::= *programs*
- ⟨ASSIGN⟩*
- ⟨EXPR⟩

⟨ASSIGN⟩::= *assignments*
- $x$ `:=` ⟨EXPR⟩ `;`

⟨EXPR⟩::= *expressions:*
- | ⟨LIT⟩
- | $x$
- | ⟨EXPR⟩ (`+` | `*`) ⟨EXPR⟩
- | ⟨EXPR⟩ `<` ⟨EXPR⟩
- | ⟨EXPR⟩ `or` ⟨EXPR⟩
- | `if` ⟨EXPR⟩ `then` ⟨EXPR⟩ (`else` ⟨EXPR⟩)?
- | `(` ⟨EXPR⟩ `)`

⟨LIT⟩::= *literals:*
- | `true` | `false` *booleans*
- | $i$ *integers*

### 3.2.1 Metavariables

We did not give an explicit definition for integers and valid variable names. This would be quite verbose, and would divert the focus to unimportant details. Instead, we use the concept of *metavariables* for such symbols to unburden us from their uninteresting implementation.

A metavariable is simply a symbol that is used to consistently represent a certain set of possible symbols. For example, we use the metavariable $i$ to represent any signed integer (e.g, 42, 0 and $-7$), and the metavariable $x$ for any valid variable name (e.g., x and myVar).

### 3.2.2 Concrete versus abstract syntax

A BNF description defines the *concrete syntax* of a language, which describes exactly how users of the language will write programs, i.e., as a sequence of symbols. However, for type checking or evaluation, it is more convenient to work with an internal tree-like representation of the syntax, called *abstract syntax*. It is the task of a *parser* to convert concrete syntax into an abstract syntax tree.

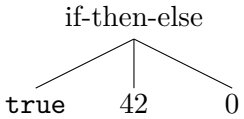
To illustrate the difference, consider the expression $2 * 3 + 4$. Given the BNF grammar in Spec. 3.1, there are two ways to parse this into a tree.


The meaning of both trees is entirely different though: the first is equivalent with $2 * (3 + 4)$ and would evaluate to 14, while the second is equivalent with $(2 * 3) + 4$ and would evaluate to 10. To avoid this ambiguity, we augment our grammar with *operator precedence*. In this case, we say the operator $*$ has higher precedence than the operator $+$, so for the given expression the literals 2 and 3 should be multiplied first, and only then should the literal 4 be added, i.e., the tree on the right.

We list the precedence of all operators from top to bottom. The precedence we use here is considered standard and conforms to the C++ specification [10].

1. Multiplication $*$.
2. Addition $+$.
3. Comparisons $<$.
4. Logical `or`.

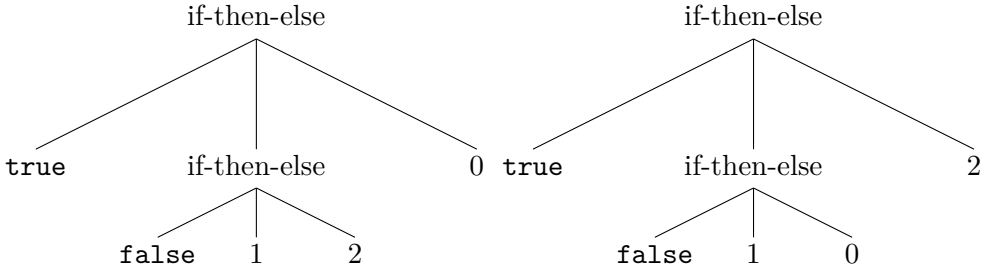
A second task of the parser is to eliminate syntactic sugar; in our case the optional "else" branch. For example, an expression such as `if true then` 42 is parsed into the following tree.



Note the additional integer 0, which represents our implicit "else" branch.
Unfortunately, another ambiguity arises. Consider the following expression.

```
if true then if false then 1 else 2
```

We can again parse this in two ways, i.e., as `if true then (if false then 1 else 2)` or as `if true then (if false then 1) else 2`.




The left tree would evaluate to 2 while the right one would evaluate to 0. This infamous problem is called the *dangling else*. The conventional solution is to attach the "else" branch to the closest conditional expression, so we accept the left tree as the correct one. We use this convention for Rosetta too.

## 3.3 The typing relation and inference rules

Type systems rule out a certain class of runtime errors. Such a type system is often defined using a set of *typing rules*. The goal of these rules is two-fold. On the one hand, they aim to assign a type to every valid expression, and on the other hand, they specify exactly when an expression is well-typed, and thus cannot lead to any runtime error that the type system wants to prevent.

For the most simple type systems, the typing relation ("expression $e$ has type $T$") is binary, and often denoted as $e : T$. However, most type systems need additional context. For example, to type an expression such as `y * (if c then x else y)` we need to know the types of variables `x`, `y` and `c`. Given that `x` and `y` are integers and that `c` is a boolean, we would then like to type the whole expression as an integer. We can write this as follows.

```
x : int , y : int , c : boolean ⊢ y * (if c then x else y) : int
```

The part before the symbol $\vdash$ is called the *typing context*, and is often denoted with $\Gamma$. Formally, this is a sequence of variables and their types. An empty context is written as $\emptyset$.

Note that this makes the typing relation ternary. We will write $\Gamma \vdash e : T$, which can be read as "given typing context $\Gamma$, expression $e$ has type $T$".

### 3.3.1 Typing rules for expressions

Our expression language features two types: `boolean` and `int`. Literals `true` and `false` always have type `boolean`, while integer literals always have type `int`. This

is formalized by the following three *axioms*.

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{boolean}}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{boolean}}$$

$$\frac{}{\Gamma \vdash i : \texttt{int}}$$

An addition always results in an integer, so we write $\Gamma \vdash e_1 + e_2 : \texttt{int}$. However, this expression is only well-typed if $e_1$ and $e_2$ are integers too. We formalize this using the following *inference rule*.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

An inference rule is often easier to understand when read from bottom to top: "we conclude that $e_1 + e_2$ has type `int` if $e_1$ has type `int` and $e_2$ has type `int`, given the same typing context $\Gamma$ everywhere". The two statements above the line are called *premises*, and the statement below the line is called the *conclusion*.

For the other binary operators, the typing rules are similar.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 * e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 < e_2 : \texttt{boolean}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{boolean} \qquad \Gamma \vdash e_2 : \texttt{boolean}}{\Gamma \vdash e_1 \,\texttt{or}\, e_2 : \texttt{boolean}}$$

Conditional expressions of the form $\texttt{if}\, e_1\, \texttt{else}\, e_2\, \texttt{then}\, e_3$ are somewhat different. The result type of such an expression depends on the type of its contents $e_2$ and $e_3$. Furthermore, $e_2$ and $e_3$ may have any type, as long as they have the same type. We therefore introduce a metavariable for types $T$ and demand that $e_2$ and $e_3$ both have the same type $T$, but without specifying which type it is. We can then conclude that the whole expression also has type $T$.

$$\frac{\Gamma \vdash e_1 : \texttt{boolean} \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : T}$$

To type a variable $x$, we look up its type in typing context $\Gamma$. The only premise is that $x$ must be present in the context.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

### 3.3.2 Typing rules for programs containing assignments

To type check a whole program, we start with an empty typing context $\emptyset$. Then, for each assignment $x := e$ we encounter, we type check the expression $e$ using the typing rules from above and add $x$ to the typing environment such that the remaining program knows the type of this variable.

To formalize this, we extend our typing rules to programs that contain assignments, i.e., of the form $x := e; p$ where $x := e$ represents the first assignment of the program and $p$ is the remaining part.

$$\frac{\Gamma \vdash e : T_x \qquad \Gamma, x : T_x \vdash p : T}{\Gamma \vdash x := e; p : T}$$

Note that this is a recursive definition: in the second premise, $\Gamma, x : T_x \vdash p : T$, we type check the remaining part of the program, this time with the knowledge of $x$'s type. If the program still contains assignments, we can apply the same rule another time. If no assignments remain, we type check the final expression with the rules from previous section, and the recursion ends.

Finally, we say a program $p$ is well-typed, written $p$ OK, if we can infer a type for it given an empty typing environment.

$$\frac{\emptyset \vdash p : T}{p \text{ OK}}$$

Using these rules, we can prove that a specific program is well-typed with a *derivation tree*. For example, the following tree proves that the program $\texttt{a} := 3; 5 < \texttt{a}$ is well-typed.

$$\dfrac{\dfrac{}{\emptyset \vdash 3 : \texttt{int}} \qquad \dfrac{\dfrac{}{\texttt{a} : \texttt{int} \vdash 5 : \texttt{int}} \qquad \dfrac{}{\texttt{a} : \texttt{int} \vdash \texttt{a} : \texttt{int}}}{\texttt{a} : \texttt{int} \vdash 5 < \texttt{a} : \texttt{boolean}}}{\emptyset \vdash \texttt{a} := 3; 5 < \texttt{a} : \texttt{boolean}}$$
$$\texttt{a} := 3; 5 < \texttt{a} \text{ OK}$$

We prove that the program is well-typed (the final conclusion) by proving it has the type `boolean` (the premise just above the final conclusion). This premise is true because the expression we assign to `a` (the literal 3) is of type `int`, and using this information, we can prove that $5 <$ `a` is of type `boolean`. Our proof ends with the axioms for literals and variables at the top of the derivation tree.

## 3.4 Denotational semantics

Semantics is concerned with the meaning of language constructs. There are several ways to define this. In general, there are three categories.

- **Operational semantics** directly describes how the language should be executed and is often close to an actual implementation of an interpreter for the language. This kind of semantics is the most detailed.

- **Denotational semantics** describes the evaluation of the language by mapping language constructs to a mathematical representation. It does not specify how the result of an evaluation should be obtained, so in that sense it is more abstract than operational semantics.
- **Axiomatic semantics** defines the meaning of a language by describing how assertions about the program state change. This semantics is the most abstract: only properties of interest need to be described in an axiomatic way, all other details can be left out.

In Rosetta, we want to clearly specify the meaning of any expression, but we are not so much interested in *how* they are evaluated. Remember that the final goal for Rosetta is code generation to a variety of languages, each with its own evaluation strategy; one of which might use strict evaluation (e.g., Java) and another one lazy evaluation (e.g., Haskell), so we want to leave that abstract. We therefore use *denotational semantics*.

As an example, we formally define the semantics of our expression language using denotations.

### 3.4.1 Denotations of types

We start by choosing a representation for evaluated expressions. For integers, an evident choice would be the set of integers $\mathbb{Z}$, and for booleans we can take the logical values *true* and *false*, represented by the set $\mathbb{B}$. We formally write this as follows.

$$\mathcal{T}[\![\texttt{int}]\!] = \mathbb{Z}$$
$$\mathcal{T}[\![\texttt{boolean}]\!] = \mathbb{B} = \{\, \mathit{true}, \mathit{false} \,\}$$

Here, the double brackets $[\![\cdot]\!]$ represent the "meaning of" operator, and the symbol $\mathcal{T}$ clarifies that we are defining the meaning of a type, as opposed to the meaning of an expression, which we denote with $\mathcal{E}$.

### 3.4.2 Denotations of expressions

Similarly to the typing rules, we need additional context if we want to evaluate an expression containing variables such as $\mathsf{x} + \mathsf{y}$. We therefore use a *store* $S$ which maps variables to values. The store can be thought of as a representation of the runtime stack of a program. For example, given a store $[\mathsf{x} \mapsto 3, \mathsf{y} \mapsto 5]$, the expression $\mathsf{x} + \mathsf{y}$ would evaluate to 8, formally written as follows.

$$\mathcal{E}[\![\mathsf{x} + \mathsf{y}]\!]\,[\mathsf{x} \mapsto 3, \mathsf{y} \mapsto 5] = 8$$

This is again a ternary relation $\mathcal{E}[\![e]\!]\,S = v$ between expressions, stores and values, read as "given store $S$, expression $e$ evaluates to the value $v$".

The rules for literals are again the simplest.

$$\begin{aligned}\mathcal{E}\,[\![\texttt{true}]\!]\,S &= \mathit{true}\\ \mathcal{E}\,[\![\texttt{false}]\!]\,S &= \mathit{false}\\ \mathcal{E}\,[\![i]\!]\,S &= i\end{aligned}$$

Note the slight abuse of notation for defining the meaning of integers. Within the brackets, $[\![i]\!]$, $i$ represents a string of symbols in our language, while for the second use of $i$, we actually mean the integer represented by those symbols, which is an element of $\mathbb{Z}$.

For an addition $e_1 + e_2$, we evaluate its arguments $e_1$ and $e_2$, and then add the results together.

$$\begin{aligned}\mathcal{E}\,[\![e_1 + e_2]\!]\,S = \textbf{let}\ & x = \mathcal{E}\,[\![e_1]\!]\,S,\\ & y = \mathcal{E}\,[\![e_2]\!]\,S\\ \textbf{in}\ & x + y\end{aligned}$$

It might seem that this rule does not say much: you evaluate an addition by… well, adding the integers together. However, know that the addition $e_1 + e_2$ is a syntactical construct of the language we want to describe, whereas the addition $x + y$ is part of the *metalanguage* with which we describe it. In the former, + represents a syntactical token, but in the latter + represents the usual mathematical operator for the addition of two integers in $\mathbb{Z}$. Similarly, the *let-in* construct is also part of the metalanguage to describe our sample language.

The evaluation of the other binary operations is similar.

$$\begin{aligned}\mathcal{E}\,[\![e_1 * e_2]\!]\,S = \textbf{let}\ & x = \mathcal{E}\,[\![e_1]\!]\,S,\\ & y = \mathcal{E}\,[\![e_2]\!]\,S\\ \textbf{in}\ & x * y\\ \mathcal{E}\,[\![e_1 < e_2]\!]\,S = \textbf{let}\ & x = \mathcal{E}\,[\![e_1]\!]\,S,\\ & y = \mathcal{E}\,[\![e_2]\!]\,S\\ \textbf{in}\ & \begin{cases}\mathit{true}, & \text{if } x < y\\ \mathit{false}, & \text{otherwise}\end{cases}\\ \mathcal{E}\,[\![e_1\ \texttt{or}\ e_2]\!]\,S = \textbf{let}\ & x = \mathcal{E}\,[\![e_1]\!]\,S,\\ & y = \mathcal{E}\,[\![e_2]\!]\,S\\ \textbf{in}\ & x \vee y\end{aligned}$$

A conditional expression $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ simply evaluates $e_2$ if the condition is true, or $e_3$ otherwise.

$$\mathcal{E}\,[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]\,S = \begin{cases}\mathcal{E}\,[\![e_2]\!]\,S, & \text{if } \mathcal{E}\,[\![e_1]\!]\,S = \mathit{true}\\ \mathcal{E}\,[\![e_3]\!]\,S, & \text{otherwise}\end{cases}$$

Lastly, to evaluate a variable, we simply look up its value in the store.

$$\mathcal{E}\,[\![x]\!]\,S = S(x)$$

### 3.4.3 Denotation of programs

Similarly to type checking, we start the evaluation of a program with an empty store $[\,]$. Then, for each assignment $x := e$ we encounter, we evaluate the expression $e$ and add $x$ to the store such that the remaining program can use it. The result of a program is the evaluation of its last expression. We formalize this with the two following rules.

$$\begin{aligned}\mathcal{P}\,[\![x := e; p]\!]\,S &= \textbf{let } v = \mathcal{E}\,[\![e]\!]\,S \\ &\quad\ \textbf{in } \mathcal{P}\,[\![p]\!]\,S[x \mapsto v] \\ \mathcal{P}\,[\![e]\!]\,S &= \mathcal{E}\,[\![e]\!]\,S\end{aligned}$$

Using these rules, we can derive the meaning of the example program $\mathsf{a} := 3; 5 < \mathsf{a}$.

$$\begin{aligned}\mathcal{P}\,[\![\mathsf{a} := 3; 5 < \mathsf{a}]\!]\,[\,] &= \textbf{let } v = \mathcal{E}\,[\![3]\!]\,[\,] \\ &\quad\ \textbf{in } \mathcal{P}\,[\![5 < \mathsf{a}]\!]\,[\mathsf{a} \mapsto v] \\ &= \mathcal{P}\,[\![5 < \mathsf{a}]\!]\,[\mathsf{a} \mapsto 3] \\ &= \mathcal{E}\,[\![5 < \mathsf{a}]\!]\,[\mathsf{a} \mapsto 3] \\ &= \textbf{let } x = \mathcal{E}\,[\![5]\!]\,[\mathsf{a} \mapsto 3], \\ &\qquad\ y = \mathcal{E}\,[\![\mathsf{a}]\!]\,[\mathsf{a} \mapsto 3] \\ &\quad\ \textbf{in } \begin{cases} \mathit{true}, & \text{if } x < y \\ \mathit{false}, & \text{otherwise} \end{cases} \\ &= \mathit{false}\end{aligned}$$

We have proven that the meaning or evaluation of the program is *false*.

## 3.5 Consistency between type system and semantics: semantic soundness

Now we have a full formal description of the expression language, one might wonder: is this specification correct? There are actually many ways in which we could describe correctness. For example, we would like the formal description of an evaluation to correspond to what we actually had in mind. Unfortunately, such a property is hard to verify, and errors can only be detected through extensive testing of the language.

Luckily, there are other properties that can be verified rigorously, or for which errors can be found in a systematic way; one of which is *semantic soundness*[11]. This property says that if a program $p$ is of type $T$, then its evaluation $\mathcal{P}\,[\![p]\!]\,S$ should be an element of $\mathcal{T}\,[\![T]\!]$. Even stronger: we can say this is true for any store $S$, as long as it is "consistent" with the typing context $\Gamma$ which gives $p$ type $T$. For example, we say the typing context $x : \texttt{boolean}, y : \texttt{int}$ is consistent with the store $[x \mapsto \mathit{true}, y \mapsto 42]$, while it is not consistent with store $[x \mapsto 3, y \mapsto \mathit{false}]$. More formally, we say store $S$ is consistent with typing context $\Gamma$, written $S : \Gamma$, if and only if store $S$ maps every variable in $\Gamma$ to an element in the denotation of its type:

$$S : \Gamma \quad \text{iff} \quad x : T \in \Gamma \Rightarrow S(x) \in \mathcal{T}\,[\![T]\!]$$

Given this definition, we can define semantic soundness as follows.

**Semantic soundness.** Suppose we have a well-typed program $p$ with $\Gamma \vdash p : T$. If a store $S$ is consistent with context $\Gamma$, i.e., $S : \Gamma$, then the following holds: $\mathcal{P}\,[\![p]\!]\,S \in \mathcal{T}\,[\![T]\!]$.

Although it is interesting to actually prove this property, it is outside the scope of this thesis. Still, when designing the semantics of a language, it is good to keep this property in mind.

# Chapter 4

# Towards a Formal Specification: Syntax and Intuition

In Chapter 2, we saw several issues in the current implementation of Rosetta. Using the techniques of previous chapter, we address these issues by creating a dialect of Rosetta's core features, called Nouga, that has a complete formal specification. This chapter starts with an informal description of entity and function declarations, after which it gives the full syntax definition. It also introduces some auxiliary definitions that help extract information from a syntactically well-formed construct, which will prove useful for formalizing the type system and semantics in later chapters. In the last two sections we explain the intuition behind all supported operations, and share a useful insight on how to reason about expressions in Nouga.

## 4.1 Entities

The syntax for declaring entities does not differ from Rosetta, except that Rosetta has additional features that are not considered here. Remember that a new entity declaration starts with the keyword `type`, followed by a name and a colon, after which a list of attribute declarations follow. For example,

```
type Employee:
  age int (1..1)
  salary number (1..1)
  isSeniorMember boolean (1..1)
  mentor Employee (0..1)
```

This entity could be a description of an employee of a company.

Entities can also *extend* another entity. Suppose that in our candy factory we have a special kind of employee that oversees the manufacturing of a specific recipe, say a *candy artisan*.

```
type CandyArtisan extends Employee:
  speciality Recipe (1..1)
  favoriteCandies CandyEnum (0..*)
```

A candy artisan is still an employee, but we need some additional information for these employees, namely which recipe that they oversee, and, of course, a list of their favorite candies. This can be done using the `extends` keyword. In this case, Employee is said to be the *parent entity* of CandyArtisan.

## 4.2 Functions

As Chapter 2 explains, Rosetta wields a peculiar `assign-output` syntax for declaring the result of a function. One disadvantage of this is that instantiation of entities remains implicit and that it is impossible to instantiate an entity with zero attributes. Therefore, Nouga makes instantiation explicit and replaces the multiple `assign-output` statements with a single one. For example, in Nouga the AddSugar function from Section 2.2 would look as follows.

```
func AddSugar:
  inputs: recipe Recipe (1..1)
  output: result Recipe (1..1)

  assign-output:
    Recipe {
      product: recipe->product,
      ingredients:
        [recipe->ingredients, IngredientEnum->SUGAR]
    }
```

Notice the JSON-like syntax to instantiate the resulting recipe. An additional advantage of adding this syntax is that it often removes the need to explicitly define a constructor for entities, resulting in less boilerplate code in Rosetta models.

As another example, consider the following function that checks whether an employee has a senior mentor.

```
func HasSeniorMentor:
  inputs: employee Employee (1..1)
  output: result boolean (1..1)

  assign-output:
    employee->mentor exists
    and employee->mentor->isSeniorMember = True
```

This function features two operators we have not seen before: the `exists` operator to check the existence of at least one item in an expression, and the equality operator `=`. Section 4.5 explains the possible expressions in more detail.

## 4.3 Formal syntax and metavariables

In previous sections, we saw how we can declare entities and functions. A Nouga model formally consists of a sequence of these declarations. The complete formal grammar is shown in Spec. 4.1.

To simplify some of the details of our syntax, we once more rely on metavariables. For example, we again use the metavariable $i$ to represent any signed integer, and we use the metavariable $F$ for any valid function name. A complete list of metavariables used throughout the remainder of this thesis can be found in Table 4.1.

## 4.4 Auxiliary definitions

In this section, we define several look-up functions that will help us to reason about syntactic constructs. A formal definition of these look-ups can be seen in Spec. 4.2. All of these look-ups make use of the *entity table* ET, which maps entity names to their definitions, or the *function table* FT, which maps function names to their definitions.

As an example, for an entity $D$, we define attrs($D$) to be the list of declared attributes of entity $D$. Here, an attribute is represented by a triple, i.e., its name, type and cardinality constraint.

The other auxiliary definitions are allattrs($D$) to look up the attributes of an entity $D$ including all attributes of its ancestors, ancestors($D$) to get a set of all ancestors of an entity $D$, inputs($F$) to look up the list of input attributes of a function $F$, output($F$) to look up the output attribute of a function, and op($F$) to get the expression representing the operation of the function.

Table 4.1: a list of metavariables and examples of what they represent.

| Description | Metavariables | Examples |
|---|---|---|
| Entity names | $D$, $E$ | CandyFactory, Recipe |
| Function names | $F$ | AddSugar |
| Attribute names | $a$, $b$ | product, ingredients, result |
| Signed integers | $i$, $j$ | $42, -5$ |
| Positive integers | $k$, $l$ | 42, 0 |
| Signed decimals | $r$ | $42.0, -3.14$ |
| Expressions | $e$ | `True and False`, $[1, 2]$ `contains` 3 |
| Types | $T$, $S$, $U$ | `boolean`, Recipe |
| Cardinality constraints | $C$ | $(1..1)$, $(0..*)$, $(2..3)$ |

SPECIFICATION 4.1: formal syntax definition of Nouga.

⟨MODEL⟩::= *root model:*
  (⟨ED⟩ | ⟨FD⟩)*

⟨ED⟩::= *entity declarations:*
  `type` *D* (`extends` *E*)? `:`
    ⟨AD⟩*

⟨FD⟩::= *function declarations:*
  `func` *F* `:`
    `inputs :` ⟨AD⟩*
    `output :` ⟨AD⟩
    `assign-output :` ⟨E⟩

⟨AD⟩::= *attribute declarations:*
  *a* ⟨T⟩ ⟨CC⟩

⟨CC⟩::= *cardinality constraints:*
  | `(` *l* `..` *k* `)` *bounded*
  | `(` *l* `..` `*` `)` *unbounded*

⟨E⟩::= *expressions:*
  | ⟨LIT⟩
  | *a*
  | ⟨E⟩ `or` ⟨E⟩
  | ⟨E⟩ `and` ⟨E⟩
  | `not` ⟨E⟩
  | ⟨E⟩ (`+` | `-`) ⟨E⟩
  | ⟨E⟩ (`*` | `/`) ⟨E⟩
  | *D* `{` (*a* `:` ⟨E⟩ (`,` *b* `:` ⟨E⟩)*)? `}`
  | ⟨E⟩ `->` *a*
  | ⟨E⟩ (`single` | `multiple`)? `exists`
  | ⟨E⟩ `is absent`
  | ⟨E⟩ `->` *a* `only exists`
  | ⟨E⟩ `count`
  | ⟨E⟩ `only-element`
  | ⟨E⟩ (`all` | `any`)? (`=` | `<>`) ⟨E⟩
  | ⟨E⟩ `contains` ⟨E⟩
  | ⟨E⟩ `disjoint` ⟨E⟩
  | `[` (⟨E⟩ (`,` ⟨E⟩)*)? `]`
  | `if` ⟨E⟩ `then` ⟨E⟩ (`else` ⟨E⟩)?
  | *F* `(` (⟨E⟩ (`,` ⟨E⟩)*)? `)`
  | `(` ⟨E⟩ `)`

⟨LIT⟩::= *literals:*
  | `True` | `False` *booleans*
  | *i* *signed integers*
  | *r* *signed decimals*
  | `empty` *empty literal*

⟨T⟩::= *types:*
  | *D*
  | `boolean`
  | `int`
  | `number`
  | `nothing`

To clarify the use of these look-ups, here are some examples.

attrs(CandyArtisan) = speciality Recipe (1..1), favoriteCandies CandyEnum (0..∗)
allattrs(CandyArtisan) = age `int` (1..1), salary `number` (1..1),
isSeniorMember `boolean` (1..1), mentor Employee (0..1),
speciality Recipe (1..1), favoriteCandies CandyEnum (0..∗)
inputs(AddSugar) = recipe Recipe (1..1)
output(AddSugar) = result Recipe (1..1)
op(AddSugar) = Recipe {product: recipe -> product, ingredients :
[recipe -> ingredients, CandyEnum -> SUGAR]}

SPECIFICATION 4.2: auxiliary definitions for extracting information from syntactical constructs.

Attribute lookup.

$$\frac{\mathrm{ET}(D) = \texttt{type}\, D \cdots : a_1\ T_1\ C_1 \ldots a_n\ T_n\ C_n}{\mathrm{attrs}(D) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n}$$

$$\frac{\mathrm{ET}(D) = \texttt{type}\, D : \ldots}{\mathrm{allattrs}(D) = \mathrm{attrs}(D)}$$

$$\frac{\mathrm{ET}(D) = \texttt{type}\, D\, \texttt{extends}\, E : \ldots}{\mathrm{allattrs}(D) = \mathrm{allattrs}(E), \mathrm{attrs}(D)}$$

Ancestors.

$$\frac{\mathrm{ET}(D) = \texttt{type}\, D\, \texttt{extends}\, E : \ldots}{E \in \mathrm{ancestors}(D)}$$

$$\frac{A \in \mathrm{ancestors}(D) \qquad \mathrm{ET}(A) = \texttt{type}\, A\, \texttt{extends}\, B : \ldots}{B \in \mathrm{ancestors}(D)}$$

Function lookups.

$$\frac{\mathrm{FT}(F) = \texttt{func}\, F : \texttt{inputs} : a_1\ T_1\ C_1 \ldots a_n\ T_n\ C_n\, \texttt{output} : \ldots}{\mathrm{inputs}(F) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n}$$

$$\frac{\mathrm{FT}(F) = \texttt{func}\, F : \texttt{inputs} : \ldots \texttt{output} : a\ T\ C \ldots}{\mathrm{output}(F) = a\ T\ C}$$

$$\frac{\mathrm{FT}(F) = \texttt{func}\, F : \ldots \texttt{assign-output} : e}{\mathrm{op}(F) = e}$$

## 4.5 Expressions

This section gives a brief and informal overview of all possible expressions in Nouga, and states explicitly when this is a feature not present in Rosetta. Note that, without formalization, this overview would be inconclusive, and many details are left open for interpretation. This is actually exactly the case for Rosetta, hence the motivation for this thesis. Still, for those unfamiliar with the language, starting with an intuitive explanation can be instructive.

**Literals.** Nouga has four kinds of literals: booleans (`True` and `False`), signed integers (represented by metavariable $i$), signed decimals (represented by metavariable $r$), and the literal `empty`, which is syntactic sugar for an empty list `[]`.

**Boolean and arithmetic operators.** Booleans can be combined using `or` and `and`, which represent the conventional boolean disjunction and conjunction respectively. Unlike Rosetta, booleans can also be negated with the `not` operator.

Integers and numbers can be combined using the usual operators `+`, `-`, `*` and `/`.

**Instantiation and projection.** Entities can be instantiated using a JSON-like syntax. The main difference with JSON is that the entity name must be in front of the record and that the attribute names must not be quoted. For example, we can instantiate an employee from Section 4.1 with the following expression.

```
Employee {
  age: 23,
  salary: 2000.00,
  isSeniorMember: False,
  mentor: Employee {
    age: 53,
    salary: 3500.00,
    isSeniorMember: True,
    mentor: empty
  }
}
```

Note that in Rosetta, instantiation is impossible to do explicitly.

We can take the value of an attribute of an entity using the projection operator `->`. If we call the employee above `e`, then `e -> age` would result in 23. This operator can be applied to expressions with plural cardinality too. For example, if `es` is a list of employees, then `es -> age` results in a list of all ages of the given employees. In case the attribute has a plural cardinality too, the result is a flattened list of all values of the projected attribute of given entities.

**Cardinality operators.** There are several operators that perform checks on the length of expressions. For example, given an expression `e`, the expression `e exists` checks whether the length of `e` is not zero, i.e., there exists at least one item.

There are two variants: `e single exists` checks whether there is *exactly* one item, and `e multiple exists` checks whether there are at least *two* items. The expression `e is absent` is syntactic sugar for `not` (`e exists`). Given an attribute `a`, the expression `e -> a only exists` checks that the attribute `a` of `e` exists and that all other attributes of `e` are absent. The expression `e count` | returns the number of items. The expression `e only-element` returns `e` if it consists of exactly one item, otherwise it returns `empty`.

**Equality operators.** There are several operators that check equality in some way between expressions. The equality operator `a = b` checks that both the length of list `a` and list `b` are equal, and that every item in `a` equals the corresponding item in `b`, checked in order. Similarly, the inequality operator `a <> b` checks that either the length of `a` and `b` differs, or that *every* item in `a` differs from the corresponding item in `b`. Note that this is not equivalent to `not` $(a = b)$, as this would check that *any* item in `a` differs from the corresponding item in `b`.

Furthermore, both these operators can be preceded by the keywords `all` and `any`. The expression `a all = b` checks that every item in `a` equals the single item `b`, and `a any = b` checks that any item in `a` equals the single item `b`. This is analogous for the `<>` operator.

Additionally, the expression `a contains b` checks that for every item in `b`, there is an equal item in `a`, and the expression `a disjoint b` checks that `a` and `b` have no items in common.

**Other expressions.** Lists can be constructed using brackets, e.g., $[1, 2, 3]$. Note that this operation will flatten all elements, e.g., $[[11, 12], 2,$ `empty` $, [31, 32, 33]]$ results in $[11, 12, 2, 31, 32, 33]$.

Conditional expressions are written using the well-known if-then-else syntax, e.g., `if` $1 + 1 = 2$ `then` $42.0$ `else` $-1/12$ results in the number 42.0. The else-branch is optional: `if` *condition* `then` *expr* is equivalent with `if` *condition* `then` *expr* `else empty`.

Functions can be called using a syntax familiar from other programming languages. For example, the function AddSugar can be called with AddSugar(recipe), given some recipe. The order of the arguments in parentheses follows the order of input attributes written in the function's declaration.

## 4.6 Everything is a list

Nouga has several operations that do not care about whether their argument has a singular or plural cardinality. For example, these are all valid expression: $[1, 2, 3]$ `count`, $4$ `count`, $[\,]$ `count`. A similar reasoning holds up for operations such as `exists`, `contains` and the projection operator `->`. Therefore, it is more elegant to *interpret all values in* Nouga *as a list.* We even consider a literal such as 4 to be *a list with a single item* 4. We could differentiate between values of singular cardinality (numbers, booleans, entities, etc.) and values of plural cardinality (lists of numbers, booleans, entities, etc.), but this would make the type system and semantics of Nouga unnecessarily complex.

A priori, one might think that this interpretation becomes a liability for the code that is generated from a Nouga model. For example, in Java we do not want single values to be wrapped in lists. However, the fact that Nouga treats everything as a list does not mean that the resulting generated code must do so. Chapter 7, which discusses a code generator to Java, shows an example of this.

Although we interpret everything as lists, we cannot have lists of lists. Operations such as list constructions $[\cdot]$ and projections `->` therefore have *flattening semantics*: if they would produce nested lists, these lists are concatenated one after the other, again resulting in a single flat list.

It is best to keep this interpretation in mind during the following chapters, as the type system and formal semantics are based on this.

# Chapter 5

# Formal Type System

Type systems rule out a certain class of runtime errors. Most type systems are concerned with preventing *validity errors*, e.g., when multiplying two expressions, then we better make sure that those expressions are numbers, and not candies. In Nouga, the type system is also concerned with *cardinality errors*, e.g., when dividing two expressions, then we better make sure that those expressions are *single* numbers, and not lists of them.

To reason about cardinality, this chapter starts by introducing *list types*. Before defining the typing rules of our system, we first discuss subtyping in Nouga, and we define some auxiliary predicates and operations that will prove useful. Lastly, after we have described the typing rules of Nouga, we convert these rules into an algorithmic version, which is more practical to use for an actual implementation.

The full specification of typing rules can be found in Appendix A.

## 5.1 Types and list types

The type system of Nouga includes cardinality constraints, which is an atypical feature. Therefore, to reason about such a system, we first need to agree on notational and naming conventions.

The type system assigns a *list type* to each expression. A list type, denoted as $T\ C$, consists of two parts: a type $T$ (e.g., `boolean`, Recipe), and a cardinality constraint $C$ (e.g., $(0..1)$, $(2..*)$). For example, the expression [1, 2, 3] is typed as `int` $(3..3)$, because it is a list of `int`s with an exactly known length of three. In case a clear contrast between a list type $T\ C$ and its type $T$ is necessary, we will call $T$ the *item type*.

In Nouga, types can either refer to entities (e.g., Recipe), or to one of three built-in types: `int`, `number` and `boolean`. To keep the focus on Rosetta's core features, we do not consider enumerations or other built-in types such as `string`, `date` and `zonedDateTime`.

## 5.2 Subtyping

Following the example of Rosetta, Nouga takes an object-oriented turn by featuring *subtyping* based on inheritance. The main motivation for subtyping is that it makes a type system less rigid. Demanding that argument types exactly match the declared parameter types of a function would lead the type checker to reject expressions that seem well-behaved to the user. The downside is that subtyping introduces a lot of additional complexity. First of all, our typing rules will no longer be *syntax-directed*, which prevents them from being implemented in a straightforward manner. As described in Section 5.5, this will necessitate converting our typing rules to an *algorithmic* version. Secondly, it complicates the code generation, which becomes worse depending on how much the subtype relation of the target language differs from Nouga's. This problem is tackled in Chapter 7.

### 5.2.1 Subtype rules of Nouga

A classic example of subtyping is Java's class hierarchy. If a class $A$ extends a class $B$, then $A$ is considered a subtype of $B$, and every instance of $A$ can be used in a context that expects an instance of type $B$. In general, a subtype relation stipulates when it is safe to allow elements of one type to be used where another is expected.

Similar to Java, in Nouga, an entity $A$ that extends another entity $B$ is considered a subtype of that entity. This is formally written as $A <: B$, where $<:$ represents the subtype relation. Formally, the rule that an entity is a subtype of its parent entity can be written as follows.

$$\frac{\mathrm{ET}(D) = \texttt{type}\ D\ \texttt{extends}\ E : \ldots}{D <: E} \quad \text{S-Extends}$$

In general, any subtype relation must also be reflexive and transitive.

$$\frac{}{T <: T} \quad \text{S-Refl}$$

$$\frac{S <: U \qquad U <: T}{S <: T} \quad \text{S-Trans}$$

These rules follow from the intuition of safe substitution. From transitivity, it follows that entities are also subtypes of their grandparents, great grandparents, etc.

Lastly, much like the Java specification considers `int` to be a subtype of `float` [12], we require `int` to be a subtype of `number`.

$$\frac{}{\texttt{int} <: \texttt{number}} \quad \text{S-Num}$$

### 5.2.2 List subtyping and the rule of subsumption

The rules from the previous section completely define Nouga's subtype relation. However, remember that expressions in Nouga have a *list* type. To know when a

list type can be considered a subtype of another list type, we need to answer the question "given an expression of list type $T_1\ C_1$, when is it safe to assume that it can behave as an expression of list type $T_2\ C_2$?"

Consider our example of employees and candy artisans from previous chapters. Given a list of three employees `es`, we can determine their age with `es -> age` resulting in a list of three integers, and we can append another employee `e` to the list with `[es, e]` resulting in a list of four employees. Note that we could perform these operations on a list of three candy artisans as well, and the result would still be a list of three integers and a list of four employees respectively. Of course, this is not the case when we apply the same operations to a list of three candy factories or a list of three numbers. The first requirement is therefore that the item type $T_1$ should be a subtype of the item type $T_2$.

Note that if a list of employees meets a cardinality constraint of (2..5), it automatically also meets any looser constraint, e.g., (1..7) and (2..∗), thus it can be used in a context where a looser constraint is expected as well. Of course, the reverse is not true, e.g., we cannot use a list of employees that meets a constraint (1..7) if a constraint of (2..5) is expected, as the number of employees might be out of the expected range. We therefore make as second demand that the cardinality constraint $C_1$ of a list subtype may not be looser than the expected constraint $C_2$, which we denote as $C_1 \subseteq C_2$ (see Section 5.3 for a precise definition).

When these two conditions are met, $T_1\ C_1$ is considered a list subtype of $T_2\ C_2$, written $T_1\ C_1 <:^* T_2\ C_2$. This rule is formalized as follows.

$$\frac{T_1 <: T_2 \qquad C_1 \subseteq C_2}{T_1\ C_1 <:^* T_2\ C_2} \quad \text{S-List}$$

The reader can verify that the list subtype relation $<:^*$ is reflexive and transitive as well.

Using this relation, we can precisely state what we mean when we say that any expression of list type $T_1\ C_1$ can be used in a context that expects an expression of a list supertype $T_2\ C_2$.

$$\frac{\Gamma \vdash e : T_1\ C_1 \qquad T_1\ C_1 <:^* T_2\ C_2}{\Gamma \vdash e : T_2\ C_2} \quad \text{T-Sub}$$

This is called the rule of subsumption. As we will see in Section 5.4, the combination of this rule with others has far-reaching consequences for the type system.

## 5.3 Type algebra

Spec. A.1 gives a precise overview of all typing rules. To do so, it uses several predicates and operations which we introduce here.

We begin with the notion of subconstraints, which formalizes when a cardinality constraint $C_1$ is stricter than another constraint $C_2$. This is the case when the

bounds of $C_1$ are equal to or between the bounds of $C_2$, i.e.,

$$\frac{l_2 \leq l_1 \qquad u_1 \leq u_2}{(l_1..u_1) \subseteq (l_2..u_2)}$$

Note that $u_1$ and $u_2$ may be infinite. Unsurprisingly, we consider an infinite bound to be greater than any finite bound, and equal to another infinite bound.

The following rule formalizes when two cardinality constraints overlap.

$$\frac{u_1 \geq l_2 \qquad u_2 \geq l_1}{\text{overlap}((l_1..u_1), (l_2..u_2))}$$

We extend the operators for addition and multiplication to work on cardinality constraints in an obvious way.

$$(l_1..u_1) + (l_2..u_2) \equiv (l_1 + l_2..u_1 + u_2)$$
$$(l_1..u_1) * (l_2..u_2) \equiv (l_1 * l_2..u_1 * u_2)$$

The *union* of two constraints is defined as the strictest constraint that includes both of its arguments.

$$\text{union}((l_1..u_1), (l_2..u_2)) \equiv (\min(l_1, l_2)..\max(u_1, u_2))$$

We say that an entity $D$ may be empty if all of its attributes have cardinality constraints that include zero.

$$\frac{\text{allattrs}(D) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n \qquad \forall i \in 1..n : (0..0) \subseteq C_i}{\text{maybeempty}(D)}$$

An example of such an entity is Quantity from Section 2.4.2. Such entities typically represents "union types" of other entities.

Two types are said to be *comparable* when we cannot derive statically that elements of these types are unequal. For example, employees are certainly unequal to recipes, so Employee is not comparable with Recipe. On the other hand, an expression of type employee might be equal to a candy artisan, so Employee is comparable with CandyArtisan. In general, two types are comparable when one type is a subtype of the other.

$$\frac{T_1 <: T_2 \vee T_2 <: T_1}{\text{comparable}(T_1, T_2)}$$

For list types, elements can only be equal if their item types are comparable and if their cardinality constraints overlap. For example, an element of `int` (1..3) is certainly unequal to an element of `int` (5..∗) because their lengths cannot be equal, whereas an element of `int` (1..3) and an element of `int` (3..5) might be equal.

$$\frac{\text{comparable}(T_1, T_2) \qquad \text{overlap}(C_1, C_2)}{\text{comparable}^*(T_1\ C_1, T_2\ C_2)}$$

## 5.4 Typing rules

We now have the necessary background to begin our endeavour of defining typing rules for every class of expressions. We go over them in the same order as in Chapter 4. At the end, we precisely define when a Nouga model passes the type checker.

**Literals.** The easiest rules are for the literals. For example, for number literals, we simply state that it is of type `number` (1..1), without any conditions.

$$\frac{}{\Gamma \vdash r : \texttt{number}\ (1..1)}\ \text{T-Number}$$

The other literals are similar.

**Variables.** The only case where we explicitly need the typing environment, is for typing variables.

$$\frac{x : T\ C \in \Gamma}{\Gamma \vdash x : T\ C}\ \text{T-Var}$$

To type a variable $x$, we simply look up its type in $\Gamma$.

**Boolean operators.** As discussed in Section 2.4, one problem of Rosetta is that it allowed expressions such as [`True`, `False`] `or` [`False`, `True`, `False`], but without specifying how this expression should be evaluated. There are two solutions for this: come up with a meaningful semantics, or disallow these expressions using the type system. We choose the latter by demanding that arguments of boolean operators must be singular. For instance, the typing rule for the `or` operator is formalized as follows.

$$\frac{\Gamma \vdash e_1 : \texttt{boolean}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{boolean}\ (1..1)}{\Gamma \vdash e_1\ \texttt{or}\ e_2 : \texttt{boolean}\ (1..1)}\ \text{T-Or}$$

Note that the arguments must have a cardinality constraint of (1..1).

**Arithmetic operators.** Analogous to the boolean operators, we demand that arguments of arithmetic operators are singular. One additional complexity is that the operators for addition, subtraction and multiplication are overloaded; they work both on integers and numbers, and the resulting type depends on the type of the arguments. We formalize this using two rules per operator, e.g.,

$$\frac{\Gamma \vdash e_1 : \texttt{int}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{int}\ (1..1)}{\Gamma \vdash e_1 + e_2 : \texttt{int}\ (1..1)}\ \text{T-PlusInt}$$

$$\frac{\Gamma \vdash e_1 : \texttt{number}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{number}\ (1..1)}{\Gamma \vdash e_1 + e_2 : \texttt{number}\ (1..1)}\ \text{T-PlusNumber}$$

Note that it is still possible to add an integer and a number together, e.g., the expression $1 + 2.5$ is well-typed. To see this, remember that `int` is a subtype of `number`. Therefore, by the rule of subsumption T-Sub, the literal `1` also has type `number` (1..1) and we can use rule T-PlusNumber to conclude that the whole expression has type `number` (1..1).

For division, we only allow `number`s as argument.

$$\frac{\Gamma \vdash e_1 : \texttt{number}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{number}\ (1..1)}{\Gamma \vdash e_1 \,\texttt{/}\, e_2 : \texttt{number}\ (1..1)}\ \text{T-Division}$$

Again, using the rule of subsumption, dividing two integers is still permitted, but the resulting type is always `number`.

**Instantiation and projection.** Typing an instantiation expression is easy: an expression of the form $D\,\{\,a_1 : e_1, \ldots, a_n : e_n\,\}$ is typed as $D$ (1..1). To check that it is well-typed however, we need to check that all argument expressions $e_1, \ldots, e_n$ are well-typed and that their type conforms to the type of the attributes $a_1, \ldots, a_n$.

$$\frac{\text{allattrs}(D) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n \qquad \forall i \in 1..n : \Gamma \vdash e_i : T_i\ C_i}{\Gamma \vdash D\,\{\,a_1 : e_1, \ldots, a_n : e_n\,\} : D\ (1..1)}\ \text{T-Instantiate}$$

In order to type an expression $e\,\texttt{->}\,a_k$, we first determine the list type $D\ C$ of expression $e$ and then look-up the type $T_k\ C_k$ of attribute $a_k$ in the declaration of $D$. Remember, though, that both expression $e$ and the attribute $a_k$ are allowed to have a plural (or zero) cardinality, and that the result is a flattened list of projecting every item in $e$ to its attribute $a_k$. For example, when $e$ is a list of length 4 and projecting a single item results in a list of length 3, then the result will be a flattened list of length 12, i.e., the product of the lengths. In general, the cardinality constraint of $e\,\texttt{->}\,a_k$ is the product of the constraints of expression $e$ and attribute $a_k$.

$$\frac{\Gamma \vdash e : D\ C \qquad \text{allattrs}(D) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n}{\Gamma \vdash e\,\texttt{->}\,a_k : T_k\ C * C_k}\ \text{T-Project}$$

**Cardinality operators.** For expressions of the form $e\,$`exists` and its variants with `single` and `multiple`, we technically need no conditions other than that its argument $e$ is well-typed. However, we will go a step further and demand that these operations are only used when sensible, i.e., when their result cannot be derived statically. For example, if $e$ has a constraint of (1..2), then we can say that $e\,$`exists` will certainly be true without having to evaluate $e$, as its length cannot be zero. Similarly, with the constraint (0..1), the expression $e\,$`multiple exists` is certainly false. We disallow these cases by demanding that the cardinality constraint of $e$ is loose enough, e.g.,

$$\frac{\Gamma \vdash e : T\ C \qquad (0..1) \subseteq C}{\Gamma \vdash e\,\texttt{exists} : \texttt{boolean}\ (1..1)}\ \text{T-Exists}$$

Here we stipulate that the constraint $C$ should at least be loose enough to allow a length of zero and one.

Unlike for projections, we impose that the subexpression $e$ in $e$ `->` $a_k$ `only exists` must be singular. If not, the semantics become unclear: for which item are you checking that $a_k$ is the only attribute that exists? Analogous to the `exists` operator, we impose an additional constraint to eliminate useless cases; namely that every attribute at least may have a length of zero and one.

$$\frac{\Gamma \vdash e : D\ (1..1) \qquad \text{allattrs}(D) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n \qquad \text{maybeempty}(D)}{\Gamma \vdash e\ \texttt{->}\ a_k\ \texttt{only exists} : \texttt{boolean}\ (1..1)}\ \text{T-OnlyExists}$$

A perceptive reader might assert that the premise maybeempty($D$) is too strong; that it eliminates cases where the result might not be determined statically. If we assume that no constraint equals (0..0) (attributes with such constraints have no use anyway), then the condition that the result may not be determined statically simplifies to $\forall i \in 1..n : i \neq k \Rightarrow (0..1) \subseteq C_i$, i.e., the condition on the cardinality constraint $C_k$ of the projection attribute itself is lifted. However, the `only exists` operator was added to Rosetta precisely for cases where *every* attribute may be empty, and (mis)using it in other cases hurts the readability of the language.

For the `count` and `only-element` operations, we set no constraints other than that their arguments are well-typed.

**Equality operators.** In the same spirit as for the cardinality operators, we stipulate that the result of equality operations may not be derivable statically. For the operators `=` and `<>`, this means that the list types of their arguments must be comparable, e.g.,

$$\frac{\Gamma \vdash e_1 : T_1\ C_1 \qquad \Gamma \vdash e_2 : T_2\ C_2 \qquad \text{comparable}^*(T_1\ C_1, T_2\ C_2)}{\Gamma \vdash e_1\ \texttt{=}\ e_2 : \texttt{boolean}\ (1..1)}\ \text{T-Equals}$$

When a modifier `all` or `any` is added, we require the second argument to be singular, and stipulate that the item types of both arguments are comparable, e.g.,

$$\frac{\Gamma \vdash e_1 : T_1\ C \qquad \Gamma \vdash e_2 : T_2\ (1..1) \qquad \text{comparable}(T_1, T_2)}{\Gamma \vdash e_1\ \texttt{any =}\ e_2 : \texttt{boolean}\ (1..1)}\ \text{T-AnyEquals}$$

For the operators `contains` and `disjoint` we only require the item types of their arguments to be comparable.

$$\frac{\Gamma \vdash e_1 : T_1\ C_1 \qquad \Gamma \vdash e_2 : T_2\ C_2 \qquad \text{comparable}(T_1, T_2)}{\Gamma \vdash e_1\ \texttt{contains}\ e_2 : \texttt{boolean}\ (1..1)}\ \text{T-Contains}$$

**Other expressions.** For list expressions, the only requirement we have is that every element has the same item type. Remember that elements need not be singular, but may also be plural or empty, and that the result is a flattened list of all the

elements. As such, the length of the result is the sum of the lengths of all the elements. Therefore, the cardinality constraint of a list expression is the sum of all constraints of its elements.

$$\frac{\forall i \in 1..n : \Gamma \vdash e_i : T\ C_i}{\Gamma \vdash [e_1, \ldots, e_n] : T \sum_{i \in 1..n} C_i} \quad \text{T-List}$$

Note that, although we require all elements to have the same item type, it is still possible to mix different types in a single list through the rule of subsumption, as long as they have a common supertype. For example, a list of type Employee may also contain elements of type CandyArtisan.

One special edge case occurs for the empty list $[\,]$. In this case, the premises in T-List vanish and we are left with the conclusion $\Gamma \vdash [\,] : T\ (0..0)$ for *any type* $T$. Combined with the rule of subsumption, this means that the empty list has every possible list type that includes a zero cardinality.

Unlike in Rosetta, the condition of an if-then-else expression $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ must be singular in Nouga. The expressions $e_2$ and $e_3$ must have the same list type.

$$\frac{\Gamma \vdash e_1 : \texttt{boolean}\ (1..1) \qquad \Gamma \vdash e_2 : T\ C \qquad \Gamma \vdash e_3 : T\ C}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T\ C} \quad \text{T-If}$$

Note again that, by the rule of subsumption, expressions $e_2$ and $e_3$ may have different list types, as long as they have a common list supertype. For example, the expressions 3.14 and $[0, 1, 2]$ have a common list supertype $\texttt{number}\,(1..3)$, so the expression $\texttt{if True then}\ 3.14\ \texttt{else}\ [0, 1, 2]$ is well-typed.

The list type of a function call is simply the declared list type of its output. To verify that a function call is well-typed, we check that every argument expression matches the expected type of the declared input attribute.

$$\frac{\begin{array}{c}\text{inputs}(F) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n \\ \text{output}(F) = a\ T\ C \qquad \forall i \in 1..n : \Gamma \vdash e_i : T_i\ C_i\end{array}}{\Gamma \vdash F(e_1, \ldots, e_n) : T\ C} \quad \text{T-Func}$$

**Well-typedness of functions.** Finally, we say a function $F$ is well-typed (written $F$ OK) if the inferred list type of its operation matches the declared list type of its output. We place the input attributes of the function in the typing environment.

$$\frac{\begin{array}{c}\text{inputs}(F) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n \\ \text{output}(F) = a\ T\ C \qquad a_1 : T_1\ C_1, \ldots, a_n : T_n\ C_n \vdash \text{op}(F) : T\ C\end{array}}{F\ \text{OK}}$$

A Nouga model passes the type checker if every function is well-typed.

## 5.5 Algorithmic typing rules

The typing rules in Spec. A.1 give a full and unambiguous definition for Nouga's type system. However, these rules exhibit two unappealing characteristics that make them unsuitable for immediate implementation.

First of all, expressions can have *multiple* (possibly infinitely many) list types. By the rule of subsumption, if an expression has a list type $T\ C$, it automatically also has every list supertype of $T\ C$. For example, the expression $[1, 2]$ has list type `int` $(2..2)$, but also `number` $(2..2)$, `int` $(0..5)$, `number` $(1..*)$ and so on. However, we are only interested in the *minimum list type* of expressions, i.e., the list type that is not a list supertype of any other list type that the expression has, as this list type holds all necessary information. We therefore would like to limit the typing relation to type expressions with their minimum list type, and remove the rule of subsumption T-Sub from the type system.

Note that this forces us to be explicit when a subtype is allowed where another type is expected. For example, when calling a function, we no longer demand that the argument types exactly match the declared list types, but rather that they are list subtypes of the expected list types.

For the special case of the empty list, however, the problem is even worse, as it has *every possible* list type that includes a zero cardinality, and no minimum list type exists. We therefore introduce a new type, called `nothing`, that is a subtype of every other type. Note that this type, often called the bottom type in type theory, cannot contain any elements, as there is no element that behaves both as a `number` and as a `boolean`. With this type, we can assign `nothing` $(0..0)$ as the minimum list type to the empty list.

A second problem is that the rules are not *syntax directed*, mainly caused by rules T-Sub and S-Trans. The reason that these rules are problematic is that their conclusion contains bare metavariables, so we never know when they might be useful for a typing derivation and when they are not. Take T-Sub.

$$\frac{\Gamma \vdash e : T_1\ C_1 \qquad T_1\ C_1 <:^*\ T_2\ C_2}{\Gamma \vdash e : T_2\ C_2} \quad \text{T-Sub}$$

For example, when given an expression $\texttt{if}\ 1 = 2\ \texttt{then}\ 42\ \texttt{else} - 1/12$, then from its syntactical form, we see immediately that rule T-If is possibly applicable, but another possibility would be to apply rule T-Sub. However, a naive type checker wouldn't know what to fill in for $T_1\ C_1$ and $T_2\ C_2$.

The same holds for S-Trans.

$$\frac{S <: U \qquad U <: T}{S <: T} \quad \text{S-Trans}$$

Since $S$ and $T$ are bare metavariables, the conclusion overlaps with every other rule, and it can be applied at any point in a subtype derivation. In addition to T-Sub, we thus would like to get rid of rule S-Trans. This forces us to be explicit in the other rules when the transitivity of subtyping is relevant.

The resulting algorithmic typing rules are depicted in Spec. A.2. Compared to the declarative version, there is one additional rule and there are nine rules that need alteration. We go through them in order.

### 5.5.1 Algorithmic subtyping rules

We state that our new `nothing` type is the bottom type with the following axiom.

$$\frac{}{\texttt{nothing} <: T} \quad \text{SA-Nothing}$$

Removing the rule of transitivity S-Trans only has consequences for derivations that conclude that an entity is the subtype of its ancestor. For example, take the following model.

```
type A:
type B extends A:
type C extends B:
```

To derive that $C <: A$, we would apply S-Extends two times to derive that $C <: B$ and $B <: A$, after which S-Trans gives us the wanted conclusion. We make S-Trans obsolete by generalizing rule S-Extends to the following.

$$\frac{E \in \text{ancestors}(D)}{D <: E} \quad \text{SA-Ancestor}$$

Here we explicitly state that an entity is the subtype of all its ancestors, and not only its direct parent entity.

### 5.5.2 Algorithmic typing rules

Removing rule T-Sub has greater consequences. We start with the arithmetic expressions. Remember that every operator, except for division, has two typing rules: one for `int`s and one for `number`s, e.g.,

$$\frac{\Gamma \vdash e_1 : \texttt{int}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{int}\ (1..1)}{\Gamma \vdash e_1 + e_2 : \texttt{int}\ (1..1)} \quad \text{T-PlusInt}$$

$$\frac{\Gamma \vdash e_1 : \texttt{number}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{number}\ (1..1)}{\Gamma \vdash e_1 + e_2 : \texttt{number}\ (1..1)} \quad \text{T-PlusNumber}$$

We still were able to add an integer and a number by combining rule T-PlusNumber with T-Sub. For the algorithmic version, we need to make this explicit. A first attempt could look like this.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1\ (1..1) \\ \Gamma \vdash e_2 : T_2\ (1..1) \qquad T_1 <: \texttt{number} \qquad T_2 <: \texttt{number}\end{array}}{\Gamma \vdash e_1 + e_2 : \texttt{number}\ (1..1)}$$

The problem is that, because `int` is a subtype of `number`, this rule would also apply to the addition of two integers. We therefore could type the expression $1 + 2$ both as `int` (1..1) *and* `number` (1..1). To preserve the property that expressions are typed with their minimum list type, we add the additional premise that at least one of the arguments must be a number.

$$\frac{\Gamma \vdash e_1 : T_1\ (1..1) \qquad \Gamma \vdash e_2 : T_2\ (1..1) \qquad T_1 <: \texttt{number} \qquad T_2 <: \texttt{number} \qquad T_1 = \texttt{number} \vee T_2 = \texttt{number}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{number}\ (1..1)} \quad \text{TA-PlusNumber}$$

The same holds for the rules of subtraction and multiplication. For division there is no conflicting rule for integers, so the additional premise is not needed.

For instantiation, we must loosen the constraint that the list type of every argument expression should exactly match the declared list type of the corresponding attribute, and explicitly state that the list types of these expressions may also be list subtypes.

$$\frac{\text{allattrs}(D) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n \qquad \forall i \in 1..n : \Gamma \vdash e_i : T_i'\ C_i' \qquad \forall i \in 1..n : T_i'\ C_i' <:^*\ T_i\ C_i}{\Gamma \vdash D\,\{\, a_1 : e_1, \ldots, a_n : e_n \,\} : D\ (1..1)} \quad \text{TA-Instantiate}$$

For a function call the alteration is analogous.

The two remaining rules that need changes, T-List and T-If, are special in the sense that their arguments may have completely different types as long as they have a common supertype. Note that their types do not even have to be comparable. Take the following example.

```
type A:
type B extends A:
type C extends A:
```

Types B and C are not comparable, but an expression such as [B {}, [C {}, C{}]] is still well-typed because rule T-List can assign it the list type A (3..3) (or any list supertype) using the rule of subsumption. Similarly, an expression such as `if True then` [B {}, B {}] `else` C {} is still well-typed, as T-If can assign it the list type A (1..2) (or any list supertype).

When we get rid of T-Sub, we therefore need an algorithm to determine the least common supertype of the arguments of the list and if-then-else operations. The resulting type is called the *join* in type theory. Formally, the join of types $T_1, \ldots, T_n$, written $\text{join}(T_1, \ldots, T_n)$ with $n \geq 0$, is defined as the type $T$ such that the following two properties are satisfied.

1. $T_i <: T$ for every $i \in 1..n$ (that is, $T$ is a supertype of $T_1, \ldots, T_n$).
2. For any type $U$, if $T_i <: U$ for every $i \in 1..n$, then $T <: U$ (that is, $T$ is a subtype of any other supertype of $T_1, \ldots, T_n$).

Analogously, we can define the *list join*, written $\text{join}^*(T_1\ C_1, \ldots, T_n\ C_n)$ as the list type satisfying the same properties, but substituting the subtype relation $<:$ with its list counterpart $<:^*$.

Using these definitions, we replace T-List and T-If with the following rules.

$$\frac{\forall i \in 1..n : \Gamma \vdash e_i : T_i\ C_i \qquad T = \text{join}(T_1, \ldots, T_n)}{\Gamma \vdash [e_1, \ldots, e_n] : T \sum_{i \in 1..n} C_i} \text{ TA-List}$$

$$\frac{\Gamma \vdash e_2 : T_1\ C_1 \qquad \Gamma \vdash e_1 : \texttt{boolean}\ (1..1) \qquad \Gamma \vdash e_3 : T_2\ C_3 \qquad T\ C = \text{join}^*(T_1\ C_1, T_2\ C_2)}{\Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : T\ C} \text{ TA-If}$$

Bear in mind that the join of two types may not exist. For example, `number` and `boolean` do not have a common supertype, so they cannot have a join either. A premise of the form $T = \text{join}(T_1, T_2)$ of an inference rule should therefore be understood as "this rule only applies if $T_1$ and $T_2$ have a join, and if they do, then we call it $T$".

Lastly, we explicitly say that a function is also well-typed if the operation is a subtype of its declared output type.

$$\frac{\text{inputs}(F) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n \qquad \text{output}(F) = a\ T\ C \qquad a_1 : T_1\ C_1, \ldots, a_n : T_n\ C_n \vdash \text{op}(F) : T'\ C' \qquad T'\ C' <: T\ C}{F\ \text{OK}}$$

### 5.5.3 Algorithmic join

Note that the definition for the join is—again—a declarative one, which is unsuitable for an actual implementation. We therefore derive an algorithmic definition from the subtyping rules of Nouga. We start with the join of only two types.

First of all, we can state in general that this operation is *idempotent* and *symmetric*. The former follows from reflexivity of the subtype relation, while the latter reaffirms that the join does not depend on the order of its arguments.

$$\frac{}{\text{join}(T, T) = T}$$

$$\frac{}{\text{join}(T_1, T_2) = \text{join}(T_2, T_1)}$$

Then, SA-Num implies the following rule.

$$\frac{}{\text{join}(\texttt{int}, \texttt{number}) = \texttt{number}}$$

To find the join of two entities $D$ and $E$, there are two cases. If $E$ is an ancestor of $D$, then the join is simply $E$. If not, it is still possible that they have a common ancestor. We therefore walk up the ancestral branch of $E$ and try to join $D$ with the

parent of $E$.

$$\frac{E \in \text{ancestors}(D)}{\text{join}(D, E) = E}$$

$$\frac{E \notin \text{ancestors}(D) \qquad \text{ET}(E) = \texttt{type}\, E\, \texttt{extends}\, E' : \ldots \qquad T = \text{join}(D, E')}{\text{join}(D, E) = T}$$

Lastly, because `nothing` is the subtype of every other type, a join with `nothing` always results in the type that was passed as other argument.

$$\frac{}{\text{join}(\texttt{nothing}, T) = T}$$

We now generalize this definition to $n \geq 0$ arguments. Note that with $n = 0$, the definition of the join simplifies to the definition of the bottom type, thus we take join() = `nothing`. The join of a single type is simply the type itself: $\text{join}(T) = T$. For $n \geq 3$, we fold the list of types $T_1, \ldots, T_n$ using the two-argument join operation from above. For example, $\text{join}(T_1, T_2, T_3, T_4)$ is equivalent with $\text{join}(T_1, \text{join}(T_2, \text{join}(T_3, T_4)))$. This is formalized as follows.

$$\frac{n \geq 3 \qquad T' = \text{join}(T_2, \ldots, T_n) \qquad T = \text{join}(T_1, T')}{\text{join}(T_1, \ldots, T_n) = T}$$

Finally, for the join of two list types, we take the join of their item types and the union of their cardinality constraints.

$$\frac{T = \text{join}(T_1, T_2) \qquad C = \text{union}(C_1, C_2)}{\text{join}^*(T_1\ C_1, T_2\ C_2) = T\ C}$$

This concludes the algorithmic type system of Nouga.

# Chapter 6

# Formal Semantics

Semantics is concerned with the meaning of language constructs. This chapter specifies the meaning of types and expressions in Nouga by mapping them into an independent mathematical domain that represents their evaluation, i.e., via denotational semantics.

Remember that the final goal of modelling in Nouga is code generation. The formal reference that we specify here is mainly meant as an aid for developers of code generators. We therefore augment our abstract reference with specific information about the representation and precision of some types and operations, e.g. for `number` and the divisions operator `/`. This way, it can fulfil its goal of providing a language-independent reference that gives a single source of truth to code generation developers.

## 6.1 Denotations of types and the semantic domain

In next section, we define a mapping from expressions to a domain that represents evaluated expressions. We first specify which sets we use to represent these evaluations by mapping types to their respective sets of possible evaluated expressions of that type. The full mapping of types is summarized in Spec. 6.1.

An important difference with the simple expression language of Chapter 3 is that Nouga supports subtyping, and this needs to be reflected in the denotations of our types if we want to maintain semantic soundness. Throughout this section, we therefore make sure that if one type is the subtype of the other, then the denotation of the former should be a subset of the denotation of the latter. As we will see, this mainly has an influence on the denotations of entities.

### 6.1.1 Representing primitives

Much like in Chapter 3, we map integers and booleans to the sets $\mathbb{Z}$ and $\mathbb{B}$ respectively. Additionally, we map numbers to the set of real numbers $\mathbb{R}$. We call the union of these three sets the set of primitive values, denoted with $\mathbb{P}$.

Note that the sets $\mathbb{Z}$ and $\mathbb{R}$ are infinite, but in practice we will use a finite representation. To avoid differences in precision among code generators, we additionally state that integers should be represented with at least 32 bits, and that numbers should have the same precision as the decimal128 standard of IEEE 754 [13]. The reason for choosing a decimal representation over a binary one is that we work in a financial context, so Rosetta (and Nouga) often needs to perform calculations on monetary values. A detailed discussion on this can be found in [14].

### 6.1.2 Representing lists

We represent a list of length $n$ as an $n$-tuple of its items. For instance, we represent a list $[1, 2, 3]$ with the triple $(1, 2, 3)$. We therefore map its list type `int (3..3)` to the Cartesian product $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$, or $\mathbb{Z}^3$. In general, we can map lists of length $n$ to the set $A^n$, given the set of items $A$. Formally, we have the following.

$$A^n = A \times A^{n-1} \quad \text{with } n \geq 1$$
$$A^0 = Unit = \{\,()\,\}$$

Here we map an empty list to the unit value $()$. Note that, given this definition, the set $\mathbb{Z}^2$ is technically equivalent with $\mathbb{Z} \times \mathbb{Z} \times Unit$, so an element always ends with the unit value $()$, e.g. $(1, 2, ())$. To circumvent this, we introduce a bit of syntactic sugar: we write $[1, 2]$ when we mean $(1, 2, ())$, and we write $[\,]$ when we mean the unit value $()$.

When the cardinality constraint of a list type is looser, we take the union of all possible lengths, e.g. we map `int (1..3)` to $\mathbb{Z}^1 \cup \mathbb{Z}^2 \cup \mathbb{Z}^3$, or $\mathbb{Z}^{1:3}$. We define the following operations.

$$A^{l:k} = \bigcup_{i \in l..k} A^i$$
$$A^{l:\infty} = \bigcup_{i \geq l} A^i$$
$$A^* = A^{0:\infty}$$

With these definitions, we define the meaning of a list type $T$ $(l..k)$, written $\mathcal{T}^*[\![T\ (l..k)]\!]$, as $(\mathcal{T}[\![T]\!])^{l:k}$.

### 6.1.3 Representing entities

For each instance of an entity, the relevant information to remember is *the name of the entity* and *the values of its attributes*. Consider the following example of a client and a child entity called member.

```
type Client:
  age int (1..1)
  favoriteNumbers number (1..*)

type Member extends Client:
  membershipIsPaid boolean (1..1)
```

An instance of Client could look as follows.

```
Client {
  age: 23,
  favoriteNumbers: [42, 3.14]
}
```

We represent such an instance as ( Client , 23, [42, 3.14]). Note that the actual attribute names are redundant; the order in which the values appear is enough to reconstruct their original meaning. In general, each instance of Client can be represented as an element in the set $\{\,\mathsf{Client}\,\} \times \mathbb{Z}^1 \times \mathbb{R}^{1:\infty}$. We therefore might be tempted to define $\mathcal{T}\,[\![\,\mathsf{Client}\,]\!]$ as this set. However, the entity Client has a child entity Member, and as such, an instance of Member might also be typed as Client. Accordingly, we would like instances of Member to be included in the set of clients. We therefore map the meaning of an entity type to the union of its set of instances with the sets of instances of all offspring entities. Formally:

$$\begin{aligned}\mathcal{T}\,[\![D]\!] &= \bigcup_{E<:D} \mathcal{D}\,[\![E]\!] \\ \mathcal{D}\,[\![D]\!] &= \textbf{let } a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n = \text{allattrs}(D) \\ &\quad\ \textbf{in } \{\,D\,\} \times \prod_{i \in 1..n} \mathcal{T}^*\,[\![T_i\ C_i]\!]\end{aligned}$$

The second line, $\mathcal{D}\,[\![D]\!]$, captures our initial intuition of representing an entity as a tuple containing its name and the values of its attributes.

As an example, let us work out the denotation of Client.

$$\begin{aligned}\mathcal{T}\,[\![\,\mathsf{Client}\,]\!] &= \mathcal{D}\,[\![\,\mathsf{Client}\,]\!] \cup \mathcal{D}\,[\![\mathsf{Member}]\!] \\ &= (\{\,\mathsf{Client}\,\} \times \mathcal{T}^*\,[\![\texttt{int}\ \ (1..1)]\!] \times \mathcal{T}^*\,[\![\texttt{number}\ \ (1..*)]\!]) \cup \\ &\quad\ (\{\,\mathsf{Member}\,\} \times \mathcal{T}^*\,[\![\texttt{int}\ \ (1..1)]\!] \times \mathcal{T}^*\,[\![\texttt{number}\ \ (1..*)]\!] \times \mathcal{T}^*\,[\![\texttt{boolean}\ \ (1..1)]\!]) \\ &= \left(\{\,\mathsf{Client}\,\} \times \mathbb{Z}^1 \times \mathbb{R}^{1:\infty}\right) \cup \left(\{\,\mathsf{Member}\,\} \times \mathbb{Z}^1 \times \mathbb{R}^{1:\infty} \times \mathbb{B}^1\right)\end{aligned}$$

We denote the union of all entity denotations with $\mathbb{E}$.

### 6.1.4 Semantic domain

Any item is either a primitive value or an entity. We denote the set of all items with $\mathbb{D}$, which is the union of $\mathbb{P}$ and $\mathbb{E}$.

Remember that we interpret every expression of Nouga as a list. Section 6.2 will make this explicit by mapping expressions into $\mathbb{D}^*$, the set of lists with any length.

### 6.1.5 Recursive types

As another example, let us try to calculate the meaning of the entity Employee from Section 4.1. To simplify the calculation, pretend that we did not define any child entities of Employee.

SPECIFICATION 6.1: denotations of types in Nouga.

Semantics of item types $\mathcal{T}[\![T]\!]$.

$$\mathcal{T}[\![\texttt{boolean}]\!] = \mathbb{B} = \{\, true, false \,\}$$

$$\mathcal{T}[\![\texttt{int}]\!] = \mathbb{Z}$$

$$\mathcal{T}[\![\texttt{number}]\!] = \mathbb{R}$$

$$\mathcal{T}[\![D]\!] = \bigcup_{E <: D} \mathcal{D}[\![E]\!]$$

Semantics of entities $\mathcal{D}[\![D]\!]$.

$$\mathcal{D}[\![D]\!] = \textbf{let } a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n = \text{allattrs}(D) \\ \textbf{in } \{\, D \,\} \times \prod_{i \in 1..n} \mathcal{T}^*[\![T_i\ C_i]\!]$$

Semantics of list types $\mathcal{T}^*[\![T\ C]\!]$.

$$\mathcal{T}^*[\![T\ (l..k)]\!] = (\mathcal{T}[\![T]\!])^{l:k}$$

Remember that an employee has an optional attribute mentor, which is again of type Employee, i.e.,

```
type Employee:
  age int (1..1)
  salary number (1..1)
  isSeniorMember boolean (1..1)
  mentor Employee (0..1)
```

Working out its meaning, we get the following.

$$\begin{aligned} \mathcal{T}[\![\text{Employee}]\!] &= \{\, \text{Employee} \,\} \times \mathcal{T}^*[\![\texttt{int}\ (1..1)]\!] \times \mathcal{T}^*[\![\texttt{number}\ (1..1)]\!] \\ &\quad \times \mathcal{T}^*[\![\texttt{boolean}\ (1..1)]\!] \times \mathcal{T}^*[\![\text{Employee}\ (0..1)]\!] \\ &= \{\, \text{Employee} \,\} \times \mathbb{Z}^1 \times \mathbb{R}^1 \times \mathbb{B}^1 \times (\mathcal{T}[\![\text{Employee}]\!])^{0:1} \\ &= \{\, \text{Employee} \,\} \times \mathbb{Z}^1 \times \mathbb{R}^1 \times \mathbb{B}^1 \times \left(Unit \cup \mathcal{T}[\![\text{Employee}]\!]^1\right) \end{aligned}$$

We have a problem here: the definition of $\mathcal{T}[\![\text{Employee}]\!]$ contains itself!

Although recursive types raise no practical issues when actually generating code, a question we might pose is: is this definition still consistent? Is there no fundamental hole in our semantic definition? This is actually a known and studied problem for any denotational semantics of a language with recursive types. We show here the solution presented by "Denotational semantics: a methodology for language development"[9].

Let's call the desired set of employees $A$. The idea is to define $A$ in a slightly different way, namely by creating a sequence of sets $A_0, A_1, \ldots$ that approximate our desired set better and better, and take the limit of this sequence as definition. We define this sequence by *unfolding* the equation from above. We take $A_0$ to be the set of employees without a mentor, $A_1$ as the set of employees without a mentor *or* with a mentor that doesn't have a mentor him or herself, and so on. More precisely:

$$\begin{aligned} A_0 &= \{\,\mathsf{Employee}\,\} \times \mathbb{Z}^1 \times \mathbb{R}^1 \times \mathbb{B}^1 \times \mathit{Unit} \\ A_1 &= \{\,\mathsf{Employee}\,\} \times \mathbb{Z}^1 \times \mathbb{R}^1 \times \mathbb{B}^1 \times (\mathit{Unit} \cup A_0) \\ A_2 &= \{\,\mathsf{Employee}\,\} \times \mathbb{Z}^1 \times \mathbb{R}^1 \times \mathbb{B}^1 \times (\mathit{Unit} \cup A_1) \\ &\vdots \end{aligned}$$

In general:

$$A_{i+1} = \{\,\mathsf{Employee}\,\} \times \mathbb{Z}^1 \times \mathbb{R}^1 \times \mathbb{B}^1 \times (\mathit{Unit} \cup A_i)$$

Note that this is a monotone sequence: $A_i$ contains all previous sets $A_{i-1}, \ldots, A_0$. We can then define $A$ as its limit: $A \equiv \lim_{n\to\infty} A_n = \cup_{n\in\mathbb{N}} A_n$.

A similar construction can be made for entities that are mutually recursive. The only condition is that, somewhere in the dependency chain, an attribute must be optional (i.e., with a cardinality constraint that includes zero). As a counter example, consider the following entity.

```
type D:
  a D (1..1)
```

Note that it is actually impossible to instantiate D, as Nouga does not support circular references (in fact, it does not support references in general). We therefore refrain from defining its semantics. These kind of circular dependencies must be detected statically and disallowed by the type system.

## 6.2 Semantics of expressions

In this section we start the last part of our endeavour to fully specify the Nouga language. We state in a language-independent way what expressions should evaluate to, and as such, we provide a single source of truth for developers of code generators.

The first part defines several auxiliary functions, called the *semantic algebra*. These functions will prove useful in the second section, which discusses the evaluation of expressions. The full specification is shown in Appendix B.

### 6.2.1 Semantic algebra

In what follows, we informally explain why and how some auxiliary functions are defined. For a formal definition, see Spec. 6.2.

The meaning of several operations, such as `exists`, `count` and `only-element`, relies on the number of items in a list. We therefore define count(_) that takes a list and returns its length.

Remember that everything in Nouga is a list, but that lists cannot be nested, i.e., there are no lists of lists. List literals and projections have flattening semantics: if an expression produces multiple lists, then all of these lists are concatenated, resulting in a single flattened list. Therefore, the function $\text{flatten}_n$, which returns the concatenation of its $n$ input lists, will prove useful.

In Nouga, equality between instances of entities is checked *deeply*, i.e., two instances are only considered equal if they stem from exactly the same entity and the values of all their attributes are equal. To make this explicit, we define three auxiliary equality functions: $\text{equals}_{\mathbb{P}}$ for primitive values, $\text{equals}_{\mathbb{E}}$ for instances of entities, and $\text{equals}^*$ for lists. Primitive values are considered equal if they simply refer to the same mathematical element. Lists are considered equal if their lengths are equal and all corresponding items are equal, checked in order.

Note that instances of different entities are still considered unequal even if they have exactly the same attributes. For example, take the following model.

```
type A:
  n int (1..1)
type B extends A:
```

Entity B has exactly the same attributes as A, but an instance $(\mathsf{B}, [42])$ is considered unequal to $(\mathsf{A}, [42])$.

For the projection operator `->`, a function that projects a single instance to its attribute will prove useful. The function $\text{project}_a(\_)$ takes an instance and returns the value of the attribute $a$.

### 6.2.2 Denotations of expressions

We now have the necessary background to define evaluation rules for every well-typed expression. Again, we go over every expression in the same order as in previous chapters.

**Literals.** We immediately formalize the statement that "everything is a list" by evaluating literals to lists with a single item, e.g.,

$$\mathcal{E}\,[\![\texttt{True}]\!]\,S = [\mathit{true}] \qquad \text{E-True}$$

The rules for the other literals are analogous.

**Variables.** Similar to the typing rule T-Var, which is the only rule that explicitly uses the typing environment $\Gamma$, we only use the store explicitly for evaluating variables.

$$\mathcal{E}\,[\![x]\!]\,S = S(x) \qquad \text{E-Var}$$

Here we simply look up the value of variable $x$ in the store $S$.

SPECIFICATION 6.2: auxiliary functions for the denotations of expressions.

$$\text{count}(\_) : \mathbb{D}^* \to \mathbb{Z} : \text{count}([a_1, \ldots, a_n]) = n$$

$$\text{flatten}_n(\_, \ldots, \_) : (\mathbb{D}^*)^n \to \mathbb{D}^* : \text{flatten}_n([a_{11}, \ldots, a_{1m_1}], \ldots, [a_{n1}, \ldots, a_{nm_n}])$$
$$= [a_{11}, \ldots, a_{1m_1}, \ldots, a_{nm_n}]$$

$$\text{equals}(\_, \_) : \mathbb{D} \times \mathbb{D} \to \mathbb{B} : \text{equals}(x, y)$$
$$= \begin{cases} \text{equals}_{\mathbb{P}}(x, y), & \text{if } x \in \mathbb{P} \wedge y \in \mathbb{P} \\ \text{equals}_{\mathbb{E}}(x, y), & \text{if } x \in \mathbb{E} \wedge y \in \mathbb{E} \\ \mathit{false}, & \text{otherwise} \end{cases}$$

$$\text{equals}_{\mathbb{P}}(\_, \_) : \mathbb{P} \times \mathbb{P} \to \mathbb{B} : \text{equals}_{\mathbb{P}}(x, y)$$
$$= \begin{cases} \mathit{true}, & \text{if } x = y \\ \mathit{false}, & \text{otherwise} \end{cases}$$

$$\text{equals}_{\mathbb{E}}(\_, \_) : \mathbb{E} \times \mathbb{E} \to \mathbb{B} : \text{equals}_{\mathbb{E}}((D, v_1, \ldots, v_m), (E, w_1, \ldots, w_n))$$
$$= \begin{cases} \mathit{true}, & \text{if } D = E \wedge \forall i \in 1..n : \text{equals}^*(v_i, w_i) \\ \mathit{false}, & \text{otherwise} \end{cases}$$

$$\text{equals}^*(\_, \_) : \mathbb{D}^* \times \mathbb{D}^* \to \mathbb{B} : \text{equals}^*([x_1, \ldots, x_m], [y_1, \ldots, y_n])$$
$$= \begin{cases} \mathit{true}, & \text{if } m = n \wedge \forall i \in 1..n : \text{equals}(x_i, y_i) \\ \mathit{false}, & \text{otherwise} \end{cases}$$

$$\text{project}_a(\_) : \mathbb{E} \to \mathbb{D}^* : \text{project}_{a_i}((D, v_1, \ldots, v_n))$$
$$= \textbf{let } a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n = \text{allattrs}(D)$$
$$\quad \textbf{in } v_i$$

**Boolean and arithmetic operators.** For an operation such as $e_1$ `or` $e_2$, we first evaluate its arguments $e_1$ and $e_2$. Given that the expression is well-typed, we know that $e_1$ and $e_2$ must result in a list with a single boolean item, which we call $x$ and $y$ respectively. We then combine these items using a logical or ($x \vee y$) and pack the result in a list again.

$$\begin{aligned}\mathcal{E}\,[\![e_1 \texttt{ or } e_2]\!]\,S = \textbf{let}\ & [x] = \mathcal{E}\,[\![e_1]\!]\,S, \\ & [y] = \mathcal{E}\,[\![e_2]\!]\,S \\ \textbf{in}\ & [x \vee y]\end{aligned} \qquad \text{E-Or}$$

The evaluation for other boolean and arithmetic operations are defined analogously.

Operators acting on numbers should be of the same precision as the decimal128 arithmetic described in the IEEE 754 standard [13].

**Instantiation and projection.** For an instantiation expression of the form $D\,\{\,a_1 : e_1, \ldots, a_n : e_n\,\}$, we evaluate all subexpressions and use the tuple representation from Section 6.1.3. The result is wrapped in a list, i.e., $[(D, \mathcal{E}\,[\![e_1]\!]\,S, \ldots, \mathcal{E}\,[\![e_n]\!]\,S)]$.

The evaluation of a projection operation $e$ `->` $a$ is a bit more intricate.

$$\begin{aligned}\mathcal{E}\,[\![e \texttt{ -> } a]\!]\,S = \textbf{let}\ & [x_1, \ldots, x_n] = \mathcal{E}\,[\![e]\!]\,S \\ \textbf{in}\ & \text{flatten}_n(\text{project}_a(x_1), \ldots, \text{project}_a(x_n))\end{aligned} \qquad \text{E-Project}$$

We first evaluate expression $e$, resulting in a list $[x_1, \ldots, x_n]$. Then, for each of these instances $x_i$, we get the value of attribute $a$, resulting again in a list. Lastly, we flatten these lists using the $\text{flatten}_n$ function.

**Cardinality operators.** For the operation $e$ `exists` we return a list with the single item *true* if the length of the evaluation of $e$ is at least one, i.e., $\text{count}(\mathcal{E}\,[\![e]\!]\,S) \geq 1$. Otherwise we return a list with the single item *false*. Similarly for `single exists` and `multiple exists`, we check that the length of the argument is exactly equal to one or at least two respectively.

An `only exists` operator applied to $e$ `->` $a$ first evaluates $e$ to a single entity $(D, v_1, \ldots, v_n)$. It returns *true* if and only if the count of the value corresponding to attribute $a$ is at least one, and the value of all other attributes is empty. Note that this operator acts on the *runtime type* $D$ of $e$, not on its static type. It is possible that the entity $D$ has more attributes than the static type of $e$, and the values of these additional attributes must also be empty for the operation to evaluate to *true*. In that sense, the `only exists` operator violates the Liskov substitution principle[15].

The expression $e$ `count` evaluates to a list containing the number of items of $\mathcal{E}\,[\![e]\!]\,S$.

The expression $e$ `only-element` evaluates its argument $e$, and returns it if it contains a single item, otherwise it returns the empty list $[\,]$.

**Equality operators.** To check whether two lists $e_1$ and $e_2$ of list type $T_1\ C_1$ and $T_2\ C_2$ respectively are equal, we simply call the auxiliary function equals$^*$:

$$\mathcal{E}\,[\![e_1 \texttt{ = } e_2]\!]\,S = \text{equals}^*(\mathcal{E}\,[\![e_1]\!]\,S, \mathcal{E}\,[\![e_2]\!]\,S) \qquad \text{E-Equals}$$

Remember that this first checks whether the lengths of $\mathcal{E}\,[\![e_1]\!]\,S$ and $\mathcal{E}\,[\![e_2]\!]\,S$ are equal. If they are, then the items are checked in order. If they contain entities, equality is checked deeply on all of their attributes.

The inequality operator $e_1$ `<>` $e_2$ checks that the length of its evaluated arguments differs, or, if they are of equal length, that *every item* in $\mathcal{E}\,[\![e_1]\!]\,S$ is not equal to the corresponding item in $\mathcal{E}\,[\![e_2]\!]\,S$. Note that this is *not* equivalent with `not` $(e_1$ `=` $e_2)$, as this would be true if *any item* differs.

$$\mathcal{E}\,[\![e_1 \texttt{<>} e_2]\!]\,S = \textbf{let}\ [x_1,\ldots,x_m] = \mathcal{E}\,[\![e_1]\!]\,S,\quad [y_1,\ldots,y_n] = \mathcal{E}\,[\![e_2]\!]\,S \quad \textbf{in}\ \begin{cases} [\mathit{true}], & \text{if } m \neq n \vee \\ & \forall i \in 1..n : \neg\,\text{equals}_(\,x_i, y_i) \\ [\mathit{false}], & \text{otherwise} \end{cases} \qquad \text{E-NotEquals}$$

If one of these operators has the modifier `all` (`any`), the second argument evaluates to a list with a single item, and we check that *every* (*any*) item in the first argument is equal/unequal to it.

For an expression $e_1$ `contains` $e_2$, we first evaluate its arguments to $[x_1,\ldots,x_m]$ and $[y_1,\ldots,y_n]$ respectively, and check that for every item $y_j$, there exists an equal item $x_i$.

$$\mathcal{E}\,[\![e_1 \texttt{ contains } e_2]\!]\,S = \textbf{let}\ [x_1,\ldots,x_m] = \mathcal{E}\,[\![e_1]\!]\,S,\quad [y_1,\ldots,y_n] = \mathcal{E}\,[\![e_2]\!]\,S \quad \textbf{in}\ \begin{cases} [\mathit{true}], & \text{if } \forall j \in 1..n : \\ & \exists i \in 1..m : \text{equals}(x_i, y_j) \\ [\mathit{false}], & \text{otherwise} \end{cases} \qquad \text{E-Contains}$$

For a `disjoint` operation, we check that *every* item $x_i$ is unequal to *every* item $y_j$.

**Other expressions.** For a list expression, we simply evaluate all of its elements and then flatten the result.

$$\mathcal{E}\,[\![[e_1,\ldots,e_n]]\!]\,S = \text{flatten}_n(\mathcal{E}\,[\![e_1]\!]\,S,\ldots,\mathcal{E}\,[\![e_n]\!]\,S) \qquad \text{E-List}$$

There are no surprises for an if-then-else expression.

$$\mathcal{E}\,[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]\,S = \textbf{let}\ [x] = \mathcal{E}\,[\![e_1]\!]\,S \quad \textbf{in}\ \begin{cases} \mathcal{E}\,[\![e_1]\!]\,S, & \text{if } x = \mathit{true} \\ \mathcal{E}\,[\![e_2]\!]\,S, & \text{otherwise} \end{cases} \qquad \text{E-If}$$

Lastly, to evaluate a function call $F(e_1,\ldots,e_n)$, we evaluate all of its arguments and put them in a new store. This store is then used to evaluate the operation of $F$, as declared in its `assign-output` statement.

$$\mathcal{E}\,[\![F(e_1,\ldots,e_n)]\!]\,S = \textbf{let}\ a_1\ T_1\ C_1,\ldots,a_n\ T_n\ C_n = \text{inputs}(F) \quad \textbf{in}\ \mathcal{E}\,[\![\text{op}(F)]\!]\,[\ a_1 \mapsto \mathcal{E}\,[\![e_1]\!]\,S,\ldots,a_n \mapsto \mathcal{E}\,[\![e_n]\!]\,S\ ] \qquad \text{E-Func}$$

Unlike our example language from Chapter 3, we do not define semantics for evaluating whole programs. Nouga is a modelling language, and as such it does not define a starting point for execution, but is purely meant as library that can be called from external languages after code generation. We have therefore reached an end to our formal description of Nouga.

### 6.2.3 Recursive functions

Nouga supports recursive functions. The most famous example is probably that of the factorial.

```
func Fac:
  inputs: n int (1..1)
  output: result int (1..1)
  assign-output:
    if n = 0 then 1 else n * Fac(n - 1)
```

With rule E-Func, we can indeed calculate the factorial of any positive integer, e.g.,

$$
\begin{aligned}
\mathcal{E}\,[\![\mathsf{Fac}(2)]\!]\,\emptyset &= \mathcal{E}\,[\![\texttt{if}\ \mathsf{n} = 0\ \texttt{then}\ 1\ \texttt{else}\ \mathsf{n} * \mathsf{Fac}(\mathsf{n}-1)]\!]\,[\mathsf{n} \mapsto [2]] \\
&= \textbf{let}\ [x_0] = \mathcal{E}\,[\![\mathsf{Fac}(\mathsf{n}-1)]\!]\,[\mathsf{n} \mapsto [2]] \\
&\quad\ \textbf{in}\ [2 * x_0] \\
&= \textbf{let}\ [x_0] = \mathcal{E}\,[\![\texttt{if}\ \mathsf{n} = 0\ \texttt{then}\ 1\ \texttt{else}\ \mathsf{n} * \mathsf{Fac}(\mathsf{n}-1)]\!]\,[\mathsf{n} \mapsto [1]] \\
&\quad\ \textbf{in}\ [2 * x_0] \\
&= \textbf{let}\ [x_1] = \mathcal{E}\,[\![\mathsf{Fac}(\mathsf{n}-1)]\!]\,[\mathsf{n} \mapsto [1]], \\
&\qquad\ x_0 = 1 * x_1 \\
&\quad\ \textbf{in}\ [2 * x_0] \\
&= \textbf{let}\ [x_1] = \mathcal{E}\,[\![\texttt{if}\ \mathsf{n} = 0\ \texttt{then}\ 1\ \texttt{else}\ \mathsf{n} * \mathsf{Fac}(\mathsf{n}-1)]\!]\,[\mathsf{n} \mapsto [0]], \\
&\qquad\ x_0 = 1 * x_1 \\
&\quad\ \textbf{in}\ [2 * x_0] \\
&= \textbf{let}\ x_1 = 1, \\
&\qquad\ x_0 = 1 * x_1 \\
&\quad\ \textbf{in}\ [2 * x_0] \\
&= [2]
\end{aligned}
$$

On the other hand, what happens if we pass a negative integer, say $\mathsf{Fac}(-1)$? Similarly to recursive types, expanding its definition with E-Func will never end, so the mapping $\mathcal{E}\,[\![\mathsf{Fac}(-1)]\!]\,S$ is not well-defined. What happens in practice is of course that the function will enter an infinite loop. Still, we can ask ourselves whether our semantic definition is not broken in some way if the meaning of a well-typed expressions is not always defined. As another example, suppose we have the following function.

```
func Loop:
  inputs:
  output: result boolean (1..1)
  assign-output: Loop()
```

When applying E-Func to determine the meaning of Loop(), we get the following.

$$\mathcal{E}\,[\![\mathsf{Loop}()]\!]\,\emptyset = \mathcal{E}\,[\![\mathsf{Loop}()]\!]\,\emptyset$$

Of course, any value in our domain satisfies this equation!

Again, this problem has already been solved over 30 years ago, e.g., in "Denotational semantics: a methodology for language development"[9] using *least fixed point semantics*, and we apply its solution to Nouga here.

To keep our semantic mapping $\mathcal{E}\,[\![\cdot]\!]\,S$ well-defined, we must map non-terminating computations to something. We therefore introduce a new value, $\bot$, and add it to our semantic domain, denoted as $\mathbb{D}^*_\bot$. To detect non-termination, we first replace the evaluation mapping $\mathcal{E}\,[\![\cdot]\!]$ with a version $\mathcal{E}_n\,[\![\cdot]\!]$ that can only "perform computations of depth $n$", and otherwise terminates by returning the value $\bot$, i.e., $\mathcal{E}_0\,[\![e]\!]\,S = \bot$ for any expression $e$. Note that this requires adapting every evaluation rule to decrement this depth counter, e.g., for the `or` operator:

$$\begin{aligned}\mathcal{E}_n\,[\![e_1\ \texttt{or}\ e_2]\!]\,S = \textbf{let}\ & [x] = \mathcal{E}_{n-1}\,[\![e_1]\!]\,S, \\ & [y] = \mathcal{E}_{n-1}\,[\![e_2]\!]\,S \\ \textbf{in}\ & [x \vee y]\end{aligned} \qquad \text{E-Or}$$

We implicitly assume that if a subcomputation such as $\mathcal{E}_{n-1}\,[\![e_1]\!]\,S$ returns the value $\bot$, then the result of the whole computation is also $\bot$.

If the computation of an expression $e$ ever terminates, then there must be an integer $n$ such that $\mathcal{E}_n\,[\![e]\!]$ does not return $\bot$. We therefore define the actual meaning of an expression as follows.

$$\mathcal{E}\,[\![e]\!]\,S \equiv \begin{cases}\mathcal{E}_k\,[\![e]\!]\,S, & \text{if } \exists k \in \mathbb{N} : \mathcal{E}_k\,[\![e]\!]\,S \neq \bot \\ \bot, & \text{otherwise}\end{cases}$$

Using this as definition, the reader can indeed verify that $\mathcal{E}_k\,[\![\mathsf{Loop}()]\!]\,S$ equals $\bot$ for any $k$. We therefore map its meaning to $\bot$, making our mapping whole again.

# Chapter 7

# Code Generation

Given the specification of previous chapter, there is still a gap between the denotational semantics of Nouga and an actual code generator. This chapter describes how to breach this gap in a systematic way, using Java as an example. It is therefore intended as the basis for a guide to implement Rosetta code generators in a disciplined way.

To specify our code generator, we follow the same steps as previous chapter to specify the meaning of list types and expressions, but replacing our mathematical domain with the Java language. We also specify how we represent entities and functions. The first section starts by stating the general goals of a code generator.

## 7.1 Goals of a code generator

When designing the code generator for a specific target language, there are two goals to consider.

1. **The generated code should have a convenient API, following the best practices of the target language.** Ideally, the generated code should look the same as if it was written manually by a specialist in the target language. For example, we should follow the naming conventions of the target language, using camelCase for methods in Java and PascalCase for methods in C#. Moreover, we should not wrap single values in array types or list types of the target language, even though we interpret expression in Nouga that way. Last, the generated code should be readable. Preferably, it should be formatted to conform to a specific style standard. One way to do this is by automatically running an opinionated code formatter such as Prettier Java[16] after generating a java file, which relieves us from the responsibility of generating pretty code ourselves.

2. **The generated code should implement the semantics correctly.** The most important requirement is that the semantics of *expressions* is implemented correctly. Additionally, we should implement the semantics of *types* correctly to preserve type safety in the target language. This is, however, of less importance,

and may be exchanged in favour of convenience for the API. For example, the most safe way to represent the list type `int` (7..7) in Java would be with a class with seven integer fields, but it is more convenient to represent all list types with plural cardinality as a list. Conversely, the list type `boolean` (1..1) is best represented with the primitive Java type boolean, which is both convenient and perfectly preserves type safety.

In the remainder of this chapter, we apply these goals to a Java code generator.

## 7.2 Representing entities

We represent entities in Java using classes. An example of the Client entity is shown in Listing 7.1.

### 7.2.1 Representing attributes

For each attribute, we generate a field with the same name and an appropriate type. Next section specifies which Java types we use. Because Rosetta (and Nouga) use the same casing conventions as Java, we can directly use the attribute names as fields. However, it is possible that a name in Nouga is actually a keyword in Java, in which case we should escape it. For example, an attribute named const should map to a field named const_.

Listing 7.1: Java representation of the Client entity. For the actual representation of the attribute types, see Section 7.3.

```
public class Client extends NougaEntity {
  private final T*J[[int (1..1)]] age;
  private final T*J[[number (1..*)]] favoriteNumbers;

  public Client(T*J[[int (1..1)]] age,
                T*J[[number (1..*)]] favoriteNumbers) { ... }

  public T*J[[int (1..1)]] getAge() { ... }
  public T*J[[number (1..*)]] getFavoriteNumbers() { ... }

  @Override
  public List<String> getAttributeNames() { ... }
  @Override
  public Object getAttributeValue(String attr) { ... }

  @Override
  public boolean equals(Object obj) { ... }
  @Override
  public boolean hashcode() { ... }
}
```

Because instances in Nouga are immutable, we declare all fields final and only provide getter methods. Note that these getter methods correspond to the auxiliary function $\text{project}_a$ from previous chapter.

### 7.2.2 Auxiliary methods

Thinking ahead about the semantics of expressions, we provide four auxiliary methods: a constructor, getAttributeNames, getAttributeValue and equals.

We will use the constructor to represent instantiation expressions, e.g., the expression Client { ... } can then be translated into new Client(...).

The next two methods are necessary for the `only exists` operator. Because this operator acts on the *runtime* type of an expression, we need a way of checking dynamically whether all attributes except for a particular one are empty, even if the static type of the entity does not know about some of the attributes. Therefore, getAttributeNames adds a runtime representation of an entity's attributes using strings. Alternatively, we could have used an enumeration, providing a safer interface. To look up the value of a given attribute, we then provide the method getAttributeValue. Another approach would be to use reflection. To make sure we can call these methods on any kind of entity, we create an abstract class NougaEntity containing these two methods and let every entity class extend from it.

The fourth method, equals, corresponds to the auxiliary function $\text{equals}_{\mathbb{E}}$, which checks equality deeply. We also override the hashcode method in order to be consistent with our deep equality. See the implementation[17] for their exact details.

### 7.2.3 Representing child entities

If an entity extends another one, we generate a class that extends the class of the other. We automatically inherit all attributes of the parent, so we do not need to generate them anymore. An example of the Member entity is shown in Listing 7.2.

An additional benefit of this approach is that we get a hierarchy-based subtype relation for free, i.e., the rule S-Extends (or its algorithmic version SA-Ancestor) has a corresponding subtype rule in Java. This will simplify the expression generator.

The auxiliary methods need to be generated as well, but can often be simplified by depending on the implementation of their superclass.

## 7.3 Representing list types

The types we use in Java act as an interface for the users of our Java model. As such, we want to generate Java types that closely resemble the types that a specialist would have chosen if they would have created the model directly in Java. For example, we do not want to wrap singular attributes in a list.

In general, we want to specify a translation from any list type $T\ C$ to an appropriate Java type $\mathcal{T}^*_{\mathcal{J}}[\![T\ C]\!]$. Based on the cardinality constraint, we distinguish three cases: *optional types* with a constraint (0..1), *singular types* with a constraint

LISTING 7.2: Java representation of the Member entity.

```
public class Member extends Client {
  private final T*J[[boolean (1..1)]] membershipIsPaid;

  public Member(T*J[[int (1..1)]] age,
                T*J[[number (1..*)]] favoriteNumbers,
                T*J[[boolean (1..1)]] membershipIsPaid) {
    super(age, favoriteNumbers);
    ...
  }

  public T*J[[boolean (1..1)]] getMembershipIsPaid() { ... }

  ...
}
```

(1..1), and *plural types* with an upper bound greater than one. We discuss them below in order.

For the edge case of *empty types* with a constraint (0..0), we use the same mapping as if it were an optional type. Note that, other than for empty lists, such a constraint has no use.

### 7.3.1 Representing optional types

In Java, the value null is an element of any reference type, so optional types are actually the most straightforward to represent. We map the list type $T$ (0..1) to an appropriate reference type $\mathcal{T}_{\mathcal{J}}\,[\![T]\!]$ and represent an empty value with null.

The item types `boolean` and `int` are the most easy to represent in Java. We use the types Boolean and Integer respectively. Note that Integer in Java uses 32 bits, which corresponds to what we specified in Chapter 6.

For the item type `number`, we might be tempted to use the Java type BigDecimal, which is able to represent decimal numbers with a precision of 128 bits. There is a catch though. The BigDecimal class keeps track of its precision on a *per instance* basis, and two instances are only considered equal if their precision is equal too[18] (e.g., the decimal 1.50 is considered different from 1.500). Moreover, its API provides no straightforward way of controlling this precision[19]. We therefore provide our own wrapper, called NougaNumber, whose precision conforms to the decimal128 standard of IEEE 754. Note that although Nouga considers `int` to be a subtype of `number` (SA-NUM), Java has no corresponding subtype rule between Integer and NougaNumber; a problem which we tackle in Section 7.5.2.

As Java has no bottom type, there is no obvious mapping for the item type `nothing`. The closest we can get is the Void type, whose only inhabitant is the value null. Unfortunately, Java has no subtype rule corresponding to SA-NOTHING; another problem that Section 7.5.2 addresses.

Lastly, we represent entity types by their corresponding class that we described in Section 7.2, i.e., $\mathcal{T}_{\mathcal{J}}\,[\![D]\!] = D$.

### 7.3.2 Representing singular types

Although null is an inhabitant of any reference type in Java, we can actually do better in case that there is an appropriate primitive type. For example, the list type `boolean` (1..1) maps one-to-one onto the Java primitive type boolean. Likewise, we can find no better representation for `int` (1..1) than the Java primitive type int. For all other types, we use the reference type as specified in previous section, and accept that we cannot eliminate null in those cases.

### 7.3.3 Representing plural types

If the upper bound of the cardinality constraint is greater than one, we represent the list type as a list in Java. However, we need to care if we want to preserve some of the subtype relation between plural types.

As a first attempt, we might map Client (0..∗) to the Java type List<Client> and Member (0..∗) to the Java type List<Member>. Unfortunately, List<Member> is *not* a subtype of List<Client>, even though Member is a subtype of Client. This is because lists in Java are *mutable*, which contradicts covariance (for an extended explanation, see [20]). Fortunately, we can use a wildcard pattern to make them covariant, e.g., List<? extends Member> is a subtype of List<? extends Client>. We therefore represent plural types as a list with such a wildcard pattern as generic parameter.

Note that the Java representation of an empty value depends on its type: if it is plural, we represent it with an empty list, and otherwise we represent it with null.

The complete translation of list types is summarized in Spec. 7.1.

## 7.4 Representing functions

Java does not support a way of defining standalone functions. We therefore wrap functions inside a class. To call a function, we first acquire an instance of its class, and then call its designated evaluate method.

As an example, let us take the following two functions. The first simply returns 42, while the second checks whether a client has this number as favorite number.

SPECIFICATION 7.1: representation of list types in Java.

Java representation of list types $\mathcal{T}^*_{\mathcal{J}}[\![T\ C]\!]$.

$$\mathcal{T}^*_{\mathcal{J}}[\![T\ (0..1)]\!] = \mathcal{T}_{\mathcal{J}}[\![T]\!] \qquad \text{J-OPTIONAL}$$

$$\mathcal{T}^*_{\mathcal{J}}[\![T\ (1..1)]\!] = \begin{cases} \texttt{boolean}, & \text{if } T = \texttt{boolean} \\ \texttt{int}, & \text{if } T = \texttt{int} \\ \mathcal{T}_{\mathcal{J}}[\![T]\!], & \text{otherwise} \end{cases} \qquad \text{J-SINGULAR}$$

$$\mathcal{T}^*_{\mathcal{J}}[\![T\ (k..l)]\!] = \texttt{List<? extends}\ \mathcal{T}_{\mathcal{J}}[\![T]\!]\texttt{>}, \quad \text{if } l > 1 \qquad \text{J-PLURAL}$$

$$\mathcal{T}^*_{\mathcal{J}}[\![T\ (0..0)]\!] = \mathcal{T}_{\mathcal{J}}[\![T]\!] \qquad \text{J-EMPTY}$$

Java reference type of each item type $\mathcal{T}_{\mathcal{J}}[\![T]\!]$.

$$\mathcal{T}_{\mathcal{J}}[\![\texttt{boolean}]\!] = \texttt{Boolean}$$

$$\mathcal{T}_{\mathcal{J}}[\![\texttt{int}]\!] = \texttt{Integer}$$

$$\mathcal{T}_{\mathcal{J}}[\![\texttt{number}]\!] = \texttt{NougaNumber}$$

$$\mathcal{T}_{\mathcal{J}}[\![\texttt{nothing}]\!] = \texttt{Void}$$

$$\mathcal{T}_{\mathcal{J}}[\![D]\!] = D$$

```
func GetAnswerToTheUniverse:
  inputs:
  output: result int (1..1)
  assign-output:
    42

func IsFormidableClient:
  inputs: client Client (1..1)
  output: isFormidable boolean (1..1)
  assign-output:
    client->favoriteNumbers any = GetAnswerToTheUniverse()
```

The function GetAnswerToTheUniverse has no inputs and its output is a singular `int`. Its corresponding evaluate method therefore has no parameters and returns a primitive `int`. (see Listing 7.3)

The function IsFormidableClient needs to be able to call GetAnswerToTheUniverse, so it should have access to an instance of the corresponding class. Following the example of Rosetta, we implement this using the famous *dependency injection* pattern. Note that this works especially well with Rosetta's abstract functions; using this

Listing 7.3: Java representation of the GetAnswerToTheUniverse function.

```
public class GetAnswerToTheUniverse {
  public int evaluate() {
    return genoutput(GetAnswerToTheUniverse);
  }
}
```

pattern, Java developers can easily inject a concrete implementation of such a function into the model. Instead of injecting instances manually through the constructor, we rely on the dependency injection framework Google Guice[21]. Dependencies are annotated with @Inject. (see Listing 7.4)

## 7.5 Representing expressions

We arrive at the last and most challenging part of our Java code generator. Our goal is to define a translation genexpr(_) from Nouga expressions to Java that preserves the semantics as specified in Chapter 6. Unfortunately, semantic correctness of our translation is hard to formally verify, especially because Java itself lacks a formal semantic specification. We will, however, focus on a weaker form of correctness, namely *well-typedness*, and delegate the verification of semantic correctness to extensive testing. In what follows, we assume that the type checker has annotated all expressions $e$ with their minimal list type $T\ C$, written $e : T\ C$.

When generating code, it is often convenient to provide a library that implements most of the language operations directly in the target language, and then generate calls to this library. Such a library is called the *runtime* of a language. We start by providing a Java runtime of Nouga.

### 7.5.1 Java runtime of Nouga

Listing 7.5 shows the signatures of our runtime methods. We often provide overloads for an operation. In some cases this is necessary, e.g., to make sure that add accepts both integers and NougaNumbers, and returns the right type depending

Listing 7.4: Java representation of the IsFormidableClient function.

```
public class IsFormidableClient {
  @Inject
  private GetAnswerToTheUniverse getAnswerToTheUniverse;

  public boolean evaluate(Client client) {
    return genoutput(IsFormidableClient);
  }
}
```

on its arguments. In other cases, this makes the API more convenient, e.g., to make sure that calling `exists` works both on plural types and non-plural types, which circumvents the need of wrapping non-plural types in a list. Overloading such methods simplifies the generation of expressions.

The last three methods provide coercions from one type into another. These will prove useful in next section.

### 7.5.2 Preserving well-typedness

To guarantee well-typed Java code, a naive generator might try to ensure that the translation of an expression $e : T\ C$ always conforms the corresponding Java type $\mathcal{T}^*_{\mathcal{J}}\,[\![T\ C]\!]$. However, that is not enough. Take the expression $1 + 2.5$ as an example. With annotations, we have the following.

$$((1 : \texttt{int}\ \ (1..1)) + (2.5 : \texttt{number}\ \ (1..1))) : \texttt{number}\ \ (1..1)$$

We could translate this expression to Nouga.add(1, new NougaNumber("2.5")), and indeed, the literal 1 has type int and new NougaNumber("2.5") has type NougaNumber, but there is no overload of Nouga.add that takes an int and a NougaNumber. The desired type actually depends on the context in which an expression is used.

Note that in the above expression, the subtype rule $\texttt{int} <: \texttt{number}$ was indirectly used to derive its type (i.e., in TA-AddNumber), while Java has no corresponding subtype rule. It is in such cases that we should be careful not to generate ill-typed code. We can identify three cases in which the list subtype relation $T_1\ C_1 <:^* T_2\ C_2$ is not preserved.

1. When $T_1$ equals `int` and $T_2$ equals `number`. E.g., the list types `int` (0..1) and `number` (0..1) translate to Integer and NougaNumber respectively.

2. When $T_1$ equals `nothing` and $T_2$ is another type. E.g., the list types `nothing` (0..0) and Client (0..1) translate to Void and Client respectively.

3. When $T_1\ C_1$ is not a plural type, while $T_2\ C_2$ is. E.g., the list types `boolean` (0..1) and `boolean` (0..∗) translate to Boolean and List<? extends Boolean> respectively.

In each of these cases, we cannot use one type if the other is expected, which will force us to insert coercions at appropriate locations. We therefore split the expression generation in two steps. (1) From an expression $e : T\ C$, generate Java code that conforms to its corresponding Java type $\mathcal{T}^*_{\mathcal{J}}\,[\![T\ C]\!]$. (2) If the resulting Java type is not a subtype of the expected type depending on its context, we add code that coerces the former type into the latter. Formally:

$$\begin{aligned}\text{genexpr}(e : T\ C, T_{\text{exp}}\ C_{\text{exp}}) = \ &\textbf{let}\ e_{\mathcal{J}} = \mathcal{E}_{\mathcal{J}}\,[\![e : T\ C]\!] \\ &\textbf{in}\ \ \text{addcoercions}(e_{\mathcal{J}}, T\ C, T_{\text{exp}}\ C_{\text{exp}})\end{aligned}$$

If the actual item type is `int`, but we expected `number`, we add a call to coerceIntToNumber. If the actual type is not plural while the expected type is,

Listing 7.5: Java runtime of Nouga.

```
public class Nouga {
    public static <T> List<? extends T> empty() { ... }

    public static boolean or(boolean e1, boolean e2) { ... }
    // analogously for `and` and `not`

    public static int add(int e1, int e2) { ... }
    public static NougaNumber add(NougaNumber e1, NougaNumber e2) { ... }
    // analogously for `-` and `*`
    public static NougaNumber divide(NougaNumber e1, NougaNumber e2) { ... }

    public static <T, Prop> Prop applyOrDefault(
        T e, Function<T, Prop> f, Prop default_) { ... }
    public static <T, Prop> List<? extends Prop> project(
        List<? extends T> e, Function<T, Prop> f) { ... }
    public static <T, Prop> List<? extends Prop> flatProject(
        List<? extends T> e, Function<T, List<? extends Prop>> f) { ... }

    public static <T> boolean exists(T e) { ... }
    public static <T> boolean exists(List<? extends T> e) { ... }
    // analogously for `single exists`
    public static <T> boolean multipleExists(List<? extends T> e) { ... }
    public static boolean onlyExists(NougaEntity e, String attr) { ... }
    public static <T> int count(T e) { ... }
    public static <T> int count(List<? extends T> e) { ... }
    public static <T> T onlyElement(List<? extends T> e) { ... }

    public static <T> boolean equals(
        List<? extends T> e1, List<? extends T> e2) { ... }
    // analogously for `<>`, `contains` and `disjoint`
    public static <T> boolean allEquals(List<? extends T> e1, T e2) { ... }
    // analogously for `all <>`, `any =` and `any <>`

    public static <T> List<? extends T> list(T ... es) { ... }
    public static <T> List<? extends T> list(List<? extends T> ... es) { ... }
    public static <T> T ifThenElse(
        boolean condition, Supplier<T> thenResult, Supplier<T> elseResult) { ... }

    public static NougaNumber coerceIntToNumber(int e) { ... }
    public static List<? extends NougaNumber> coerceIntToNumber(
        List<? extends Integer> e) { ... }
    public static <T> List<? extends T> coerceItemToList(T e) { ... }
}
```

we add coerceItemToList. Note that it is possible that both coercions are necessary, e.g., for an actual type `int` (1..1) and an expected type `number` (0..∗). If the actual item type is `nothing` but the expected one is not, we replace the whole expression with null or Nouga.empty(), depending on whether the type is plural or not.

$$\text{addcoercions}(e_{\mathcal{J}}, T_{\text{act}}\ C_{\text{act}}, T_{\text{exp}}\ C_{\text{exp}}) = \begin{cases} \mathsf{null}, & \text{if } T_{\text{act}} = \texttt{nothing} \wedge T_{\text{exp}} \neq \texttt{nothing} \wedge \\ & C_{\text{exp}} \text{ is not plural} \\ \mathsf{Nouga.empty()}, & \text{if } T_{\text{act}} = \texttt{nothing} \wedge T_{\text{exp}} \neq \texttt{nothing} \wedge \\ & C_{\text{exp}} \text{ is plural} \\ \text{addlistcoercion}( & \text{if } T_{\text{act}} = \texttt{int} \wedge T_{\text{exp}} = \texttt{number} \\ \quad \mathsf{Nouga.coerceIntToNumber}(e_{\mathcal{J}}), & \\ \quad C_{\text{act}}, C_{\text{exp}}), & \\ \text{addlistcoercion}(e_{\mathcal{J}}, C_{\text{act}}, C_{\text{exp}}) & \text{otherwise} \end{cases}$$

Here we used an auxiliary function for coercing non-plural types to plural types, defined as follows.

$$\text{addlistcoercion}(e_{\mathcal{J}}, C_{\text{act}}, C_{\text{exp}}) = \begin{cases} \mathsf{Nouga.coerceItemToList}(e_{\mathcal{J}}), & \text{if } C_{\text{act}} \text{ is not plural} \wedge C_{\text{exp}} \text{ is plural} \\ e_{\mathcal{J}}, & \text{otherwise} \end{cases}$$

Using this method, we can guarantee well-typedness of the generated expressions. To generate the output of a function, we call genexpr and expect the type as declared in its output.

$$\begin{aligned} \text{genoutput}(F) = \textbf{let}\ & a\ T\ C = \text{output}(F) \\ \textbf{in}\ & \text{genexpr}(\text{op}(F), T\ C) \end{aligned}$$

### 7.5.3 Generating expressions

The final task is to define the semantic translation from Nouga expressions $e : T\ C$ to Java expressions $\mathcal{E}_{\mathcal{J}}\ [\![e : T\ C]\!]$, while making sure that their type conforms to $\mathcal{T}^*_{\mathcal{J}}\ [\![T\ C]\!]$. We go over some examples.

**Literals.** We map boolean and integer literals to their corresponding literal in Java. For a number literal $r$, we instantiate our custom number class: new NougaNumber ("$r$").

**Boolean operators.** For a boolean operator such as $e_1$ `or` $e_2$, we generate a call to Nouga.or with as arguments genexpr($e_1$, `boolean` (1..1)) and genexpr($e_2$, `boolean` (1..1)).

**Arithmetic operators.** A call to an arithmetic operation such as Nouga.add is only well-typed if both arguments are either integers or NougaNumbers. We ensure this by using the function genexpr from previous section.

$$\mathcal{E}_{\mathcal{J}}\,[\![((e_1 : T_1\ C_1) + (e_2 : T_2\ C_2)) : T\ C]\!] = \mathsf{Nouga.add}(\,\mathrm{genexpr}(e_1 : T_1\ C_1, T\ C),\ \mathrm{genexpr}(e_2 : T_2\ C_2, T\ C))$$

If one of the arguments is an integer, but we expect a NougaNumber, genexpr will add the appropriate coercion for us.

**Cardinality operators.** Because our runtime method for `exists` is overloaded for both plural and non-plural types, we can safely pass it any argument without providing coercions.

$$\mathcal{E}_{\mathcal{J}}\,[\![(e : T\ C)\,\texttt{exists}]\!] = \mathsf{Nouga.exists}(\mathcal{E}_{\mathcal{J}}\,[\![e : T\ C]\!])$$

**Projection.** For projections $(e : D\ C)\,\texttt{->}\,a$, we distinguish four cases, depending on whether the expression $e$ and the attribute $a$ are plural or not. In the easiest case, we have a non-plural instance and we can call the appropriate getter, as long as the instance is not null. If it is null, we return either null or an empty list, depending on the whether the attribute is plural or not. In the other cases, we either generate a call to Nouga.project if $a$ is non-plural or a call to Nouga.flatProject if $a$ is plural. Formally:

$$\mathcal{E}_{\mathcal{J}}\,[\![(e : D\ C)\,\texttt{->}\,a]\!]$$

$$= \begin{cases} \mathsf{Nouga.applyOrDefault}( & \text{if } C \text{ is not plural} \wedge a \text{ is not plural} \\ \quad \mathcal{E}_{\mathcal{J}}\,[\![e : D\ C]\!], D :: \mathsf{get}A, \mathsf{null}), & \\ \mathsf{Nouga.applyOrDefault}( & \text{if } C \text{ is not plural} \wedge a \text{ is plural} \\ \quad \mathcal{E}_{\mathcal{J}}\,[\![e : D\ C]\!], D :: \mathsf{get}A, \mathsf{Nouga.empty}()), & \\ \mathsf{Nouga.project}( & \text{if } C \text{ is plural} \wedge a \text{ is not plural} \\ \quad \mathrm{genexpr}(e, D\ (0..*)), D :: \mathsf{get}A), & \\ \mathsf{Nouga.flatProject}( & \text{otherwise} \\ \quad \mathrm{genexpr}(e, D\ (0..*)), D :: \mathsf{get}A), & \end{cases}$$

Note that if $D\ C$ is not plural in the last two cases, the function genexpr will insert a coercion that converts $e$ to a list.

We can continue in this fashion. For the full translation from Nouga expressions to Java, see the implementation[17].

# Chapter 8

# Evaluation and Limitations

To evaluate the work of this thesis, we provide an implementation of the formal specification from Chapters 4–7 (see [17]). Following the example of Rosetta, we implemented Nouga using the language workbench *Xtext*[22] and the DSL *Xsemantics*[23] for implementing type systems, following the guidelines described in the paper *Type errors for the IDE with Xtext and Xsemantics*[24] for developing a responsive and user-friendly type system.

Aside from extensive unit testing, we evaluate this implementation by discussing examples of several Nouga models that illustrate the differences with Rosetta. We also discuss an example where our type system is too strong—or rather not smart enough—and rejects a useful model that cannot lead to runtime errors.

In Section 8.2 we evaluate our work by implementing real-life model for financial contracts, based on the paper that originally proposed this model and provides an implementation in Haskell[25]. Aside from Nouga's limited feature set, we conclude that its main limitations are currently caused by too strong cardinality checks and a cumbersome syntax for instantiating entities with many optional attributes. Chapter 9 discusses potential solutions for these problems.

## 8.1 Examples

### 8.1.1 Cardinality checks

In Chapter 2, we saw that Rosetta performs no cardinality checks, which made it accept expressions like $[1, 2] + [3, 4]$. Nouga's type system not only disallows such simple cases, but the compositionality of its typing rules enables it to check more complex expressions as well, such as in the following example.

```
func CC:
  inputs:
  output: result int (3..4)
  assign-output:
    if True then [1, 2] else [1, 2, 3, 4]
```

In this case, the user is notified with the message "`Expected a list subtype of 'int (3..4)', but was 'int (2..4)'.`"

### 8.1.2 Subtyping and empty entities

Rosetta does not allow the creation of heterogeneous lists if the types of its elements are not comparable, even when the types are siblings with a common supertype. Nouga fixes this limitation. For example, the following model is valid.

```
type A:
type B extends A:
type C extends A:
type D:

func HeterogeneousList:
  inputs:
  output: result A (2..2)
  assign-output:
    [B {}, C {}]
```

If, however, we would replace C {} with the unrelated entity D {}, the type checker would provide us with the message "`Elements do not have a common supertype: 'B', 'D'.`" Note that, unlike in Rosetta, we are also able to instantiate entities with no attributes, and we do not need to define constructor functions to do so.

### 8.1.3 Robust empty literal and optional else

In Rosetta, empty literals and conditional expressions without an 'else' branch always result in null in the generated Java code, which either leads to null reference exceptions or to generated code with extensive null checks. Furthermore, Rosetta disallows explicitly writing an empty 'else' branch such as `if True then 42 else empty` because an empty literal is wrongly assigned the top type any instead of the bottom type. Nouga fixes this by introducing a bottom type and considering empty expression as syntactic sugar for an empty list []. Furthermore, in Java such expressions either result in null or an empty list depending on the context in which they are used. This removes the need for null checks, as in the following example that creates a list of all integers between two limits.

```
func Range:
  inputs:
    low int (1..1)
    high int (1..1)
  output: range int (0..*)
  assign-output:
      [low, if low <> high then Range(low + 1, high)]
```

In Rosetta, the conditional without else branch would result in null when low is equal to high. The code generator from Chapter 7 instead generates an empty list.

### 8.1.4 Type coercions in generated code

In some cases, Rosetta generates ill-typed Java code. For example, given an expression `if True then 5 else [1.0, 3.14]`, the literal 5 becomes an `Integer` and the literals 1.0 and 3.14 become `BigDecimal`s, and even though the Java generator takes care of the cardinality mismatch, it fails to convert the integer to a `BigDecimal`. The code generator from Chapter 7, however, would generate two coercions, effectively taking care of the mismatch in type and in plurality.

```
Nouga.ifThenElse(true,
  () -> Nouga.coerceItemToList(Nouga.coerceIntToNumber(5)),
  () -> Nouga.list(new NougaNumber("1.0"),
                   new NougaNumber("3.14")))
```

### 8.1.5 Limitations of cardinality checks

Including cardinality constraints in the type system allows Nouga to reject models that could otherwise result in runtime errors. However, it also rejects some useful models that *cannot* result in runtime errors. For example, consider a function that tries to adds two optional numbers, but returns zero if at least one of them is empty.

```
func AddOrZero:
  inputs:
    a number (0..1)
    b number (0..1)
  output: result number (1..1)
  assign-output:
    if a exists and b exists then
      a + b
    else
      0
```

Unfortunately, even though we know that `a` and `b` cannot be empty when we add them, rule TA-AddNumber forbids this expression and the type checker rejects it with a message "`Expected constraint ‘(1..1)‘, but was ‘(0..1)‘.`"

A quick solution to this problem would be to weaken the type system. For example, instead of demanding that expressions must exactly have a cardinality of one when we add them, we could demand that their cardinality constraints *allow* a cardinality of one, i.e., replace rule TA-PlusInt with the following.

$$\frac{\Gamma \vdash e_1 : \texttt{int}\ C_1 \qquad \Gamma \vdash e_2 : \texttt{int}\ C_2 \qquad (1..1) \subseteq C_1 \qquad (1..1) \subseteq C_2}{\Gamma \vdash e_1 + e_2 : \texttt{int}\ (1..1)} \quad \text{T-PlusInt-Weak}$$

This way, the expression `a+b` from above would be valid, and we still reject expression such as $[1, 2] + [3, 4]$. In case we try to add plural integers, the Java generator could provide an *unsafe* coercion to convert a list to a single item. If the list is empty or contains multiple items, this then results in a runtime error; the disadvantage of this

approach. Fortunately, there are other solutions that do not surrender the strength of our type system. Chapter 9 discusses two of them.

## 8.2 Case study: a combinatorial model for financial contracts

To see how Nouga holds up in a real-world scenario, we try to make a combinatorial model for financial contracts based on the paper by Peyton Jones et al. called "Composing Contracts: An Adventure in Financial Engineering"[25]. We start by briefly describing the model and its fundamental building blocks. After that, we implement the model in Nouga. As we will see, the issue described in Section 8.1.5 forms the main obstacle. Other issues are mostly due to Nouga's limited set of features, which is expected as we only focussed on Rosetta's core features.

### 8.2.1 Motivation of the model

We start with a motivational example contract loosely taken from the "Composing Contracts" paper.

$C$ The right to choose on 30 June 2022 between:

- $C_1$ Both of:
  - $C_{11}$ Receive €100 on 29 Jan 2023.
  - $C_{12}$ Pay €105 on 1 Feb 2024.
- $C_2$ An option exercisable on 15 Dec 2022 to choose one of:
  - $C_{21}$ Both of:
    - $C_{211}$ Receive €100 on 29 Jan 2023.
    - $C_{212}$ Pay €106 on 1 Feb 2024.
  - $C_{22}$ Both of:
    - $C_{221}$ Receive €100 on 29 Jan 2023.
    - $C_{222}$ Pay €112 on 1 Feb 2025.

Although simplified, this contract is a realistic example of the contracts that are traded in financial derivative markets. The key point is that complex contracts, such as $C$, can be defined in terms of simpler contracts, such as $C_1$ and $C_2$, which in their turn are defined in terms of even simpler contracts.

The financial industry wields an extensive vocabulary for describing different kinds of financial contracts (swaps, futures, caps, floors, swaptions, spreads, straddles, captions, European options, American options, ...). The trouble with modelling each of these contracts separately is that, as Peyton Jones et al. argue, "someone will soon want a contract that is not in the catalogue." Using a carefully chosen set of building blocks and combinators instead, describing new and unforeseen contracts becomes much easier. Furthermore, when using a compositional model, downstream functions (e.g. for valuing contracts or generating legal paperwork) do not need to be adjusted every time a new type of contract is added.

### 8.2.2 Brief description of the contract primitives

To describe the semantics of contracts, we use two core concepts: the *acquisition date* and the *expiry date*.

The consequences of a contract depend on the date at which it is acquired, i.e., its acquisition date. For example, a contract "receive €100 on 1 Jan 2022 and receive €100 on 1 Jan 2023" is worth a lot less if acquired *after* 1 Jan 2022, because any rights and obligations from before the acquisition of the contract are dropped.

Some contracts may only be acquired before a given date, after which they expire. A typical example of such a contract is an called an *option*, e.g., "the right to decide on or before 1 Jan 2023 whether to acquire another contract $C$". Its expiry date is 1 Jan 2023, so it may not be acquired after that day, although the consequences of the underlying contract $C$ may extend well beyond 1 Jan 2023. Note that the expiry date is inherent to a contract, whereas the acquisition date is not.

Given these concepts, we briefly describe the primitives for defining contracts.

- **Zero.** This contract has no consequences, and never expires.
- **One of currency** $c$**.** The holder of this contract immediately receives one unit of currency $c$. It never expires.
- **Give contract** $C$**.** Normally, the holder of a contract receives the payments and makes the choices as specified in the contract; as opposed to the counterparty who pays and undergoes the choices. The situation is reversed with this primitive.
- $C_1$ **and** $C_2$**.** Acquiring this contract means that you immediately acquire both $C_1$ (if it has not expired) and $C_2$ (if it has not expired). When both have expired, the composite contract expires.
- $C_1$ **or** $C_2$**.** Acquiring this contract means that you immediately must either acquire $C_1$ (if it has not expired) or $C_2$ (if it has not expired), but not both. When both have expired, the composite contract expires.
- **Truncate contract** $C$ **to date** $t$**.** This contract is exactly like $C$, but expires at a given date $t$. If $C$ expires before $t$, the truncated contract expires then too.
- $C_1$ **then** $C_2$**.** Acquiring this contract means that you acquire $C_1$ if it has not expired, or else acquire $C_2$. When both have expired, the composite contract expires.
- **Scale contract** $C$ **with the value of observable** $o$**.** Acquiring this contract means that you acquire $C$, but with all consequences scaled with the value of an observable $o$ at that time. This observable might be a constant (e.g., scale everything with 100), or it might be a varying observable quantity (e.g., scale everything with the current gold price in euros). The compound contract expires when $C$ does.

- **Get contract $C$.** The holder of this contract acquires $C$ just before it expires. The compound contract expires when $C$ does.
- **Acquire contract $C$ at any time.** The holder of this contract must acquire $C$, but may do so at any time before $C$ expires.

### 8.2.3 Implementation in Nouga

**Constract primitives.** In Rosetta, we could represent each contract primitive as an entity, and define a contract as the *tagged union* of these entities using the condition one–of. Unfortunately, Nouga does not support conditions. We will still represent a contract in the same way, but we loose some safety.

```
type Contract:
  zero          Contract_Zero     (0..1)
  one           Contract_One      (0..1)
  give          Contract_Give     (0..1)
  andContract   Contract_Both     (0..1)
  orContract    Contract_Or       (0..1)
  truncate      Contract_Truncate (0..1)
  thenContract  Contract_Then     (0..1)
  scale         Contract_Scale    (0..1)
  get           Contract_Get      (0..1)
  anytime       Contract_Anytime  (0..1)

type Contract_Zero:
type Contract_One:
  currency Currency (1..1)
type Contract_Give:
  contract Contract (1..1)
...
```

**Auxiliary entities.** There are three primitives that depend on additional types. The first is Contract_One, as defined above, which has an attribute of type Currency. In Rosetta, we could have defined an enum for that, but that feature is not supported in Nouga. We use the following workaround.

```
type Currency:
type EUR extends Currency:
type USD extends Currency:
...
```

The primitive "receive one euro" can then be represented as Contract_One {currency : EUR {}}.

The second is Contract_Truncate, which needs a type for representing *dates*. In Rosetta, there is a designated date type for that, which Nouga ignores. We therefore stub this type with a simple entity.

```
type Contract_Truncate:
  contract   Contract (1..1)
  expiryDate Date     (1..1)

type Date:
  day   int (1..1)
  month int (1..1)
  year  int (1..1)
```

The third notable primitive is Contract_Scale, which relies on an *observable*.

```
type Contract_Scale:
  contract   Contract   (1..1)
  observable Observable (1..1)
```

An observable can be anything from a simple constant to the current gold price. We define two observable quantities to illustrate: constants and the number of days to a given date. We again model this as a tagged union.

```
type Observable_Const:
  val number (1..1)
type Observable_Time:
  time Date (1..1)

type Observable:
  const Observable_Const (0..1)
  time  Observable_Time  (0..1)
```

The actual value of an observable can then be observed with the following function.

```
func Observe:
  inputs:
    observable Observable (1..1)
    time       Date       (1..1)
  output: result number (0..1)
  assign-output:
    if observable->const exists then observable->const->val
    else if observable->time exists then DateDifference(
   observable->time->time, time)
```

Here we used an auxiliary function DateDifference to calculate the days between two dates. In the CDM, this function is abstract which delegates its implementation to the target language. Again, Nouga does not support this, so we provide a stub instead.

```
func DateDifference:
  inputs:
    firstDate  Date (0..1)
    secondDate Date (0..1)
  output: difference int (1..1)
  assign-output:
      42 // dummy value
```

Note that the cardinality constraint of the output of Observe as well as the constraints of the inputs of DateDifference are (0..1), although we know they can actually not be empty. This is due to the limitation discussed in Section 8.1.5.

Because DateDifference returns an `int`, but Observe needs to return a `number`, the current code generator of Rosetta actually generates ill-typed code for the function Observe. Our code generator fixes this problem.

**Smart constructors.** If we actually want to create an observable, we must first either create an Observable_Const or an Observable_Time and then wrap it in the Observable entity. To eliminate this boilerplate, we provide two "smart constructors".

```
func MkConst:
  inputs: const number (1..1)
  output: observable Observable (1..1)
  assign-output:
    Observable {
      const: Observable_Const { val: const },
      time: empty
    }
func MkTime:
  inputs: time Date (1..1)
  output: observable Observable (1..1)
  assign-output:
    Observable {
      const: empty,
      time: Observable_Time { time: time }
    }
```

Note that in Rosetta, using an input named const leads to invalid Java code, because const is a keyword in Java. Our Java generator takes care of this problem by escaping such names (e.g., const_).

Similarly, we can define smart constructors for all contract primitives.

```
func MkZero:
  inputs:
  output: contract Contract (1..1)
  assign-output:
    Contract {
      zero: Contract_Zero {}, one: empty, give: empty,
      andContract: empty, orContract: empty,
      thenContract: empty, truncate: empty, scale: empty,
      get: empty, anytime: empty
    }
func MkOr:
  inputs:
    left  Contract (1..1)
    right Contract (1..1)
  output: contract Contract (1..1)
  assign-output:
    Contract {
      orContract: Contract_Or { left: left, right: right },
      ...
    }
...
```

There are two things to note here. An advantage of Nouga is that it allows us to instantiate an empty entity like Contract_Zero, which is impossible in Rosetta. A disadvantage is that we need to explicitly assign `empty` to every attribute of Contract except for one, resulting in boilerplate code. Rosetta, however, implicitly assigns null to absent attributes.

**Defining new contracts.** The great advantage of the combinatorial model of Peyton Jones et al. is that it is *expandable*. New contracts can be easily described in terms of primitives, and downstream functions such as valuing a contract immediately work for new contracts too, without needing adjustments.

A simple example of another contract is the *zero-coupon discount bond*, such as "receive €100 on 1 Jan 2030". We break this contract down in four primitives.

The contract "immediately receive €100" is described by scaling one euro with a constant observable of 100: MkScale(MkConst(100), MkOne(EUR {})). To postpone receiving €100, we first truncate the contract to 1 Jan 2030 (i.e., "you may only acquire this contract before 1 Jan 2030") and then wrap it in a Contract_Get (i.e., "you acquire this contract exactly at the expiry date; 1 Jan 2030"). In general:

```
func ZeroCouponBond:
  inputs:
    maturesOn Date     (1..1)
    amount    number   (1..1)
    currency  Currency (1..1)
  output:
    contract Contract (1..1)
  assign-output:
    MkGet(MkTruncate(maturesOn, MkScale(MkConst(amount),
                                        MkOne(currency))))
```

In a similar way, we can describe more complex contracts such as European options without a problem.

### 8.2.4 Conclusion

To summarize, the two most important limitations we encountered are that

1. cardinality checks are not smart enough, and that
2. instantiating entities with many empty attributes is cumbersome.

A potential solution to the latter could be to automatically assign empty to absent attributes. Alternatively, to prevent accidentally forgetting an attribute, we could add syntax such as three dots (e.g., Contract {zero: Contract_Zero {}, ...} which would force a programmer to make this intention explicit. Chapter 9 discusses solutions to the other problem.

We also saw several advantages compared to Rosetta. Aside from the benefits of the stronger type checking of our type system, we could

1. instantiate empty entities such as Contract_Zero,
2. use identifiers such as const, which would not be escaped in Rosetta's Java generator, and
3. use an `int` were a `number` was expected, which would cause Rosetta to generate ill-typed Java code.

Other limitations of Nouga we saw come from its limited set of features, such as the absence of conditions, enums, abstract functions and the date type. Note though that it was never the intention to include all of Rosetta's features, so for the purpose of this thesis, we cannot really regard this as limitations. It should be possible to add these missing features to Nouga without altering the framework that we built in a major way.

# Chapter 9

# Conclusion and Future Work

Originally, the goal of this thesis was to investigate the possibility of creating a compositional model for financial contracts using Rosetta, as we did in Chapter 8. During this investigation, however, the issues discussed in Section 2.4 of this thesis became apparent. These issues—a type system without cardinality checks, weak subtyping, the lack of a semantic specification, limiting syntactical constructs, and a faulty code generator—became the new drive for the research of this thesis.

The main goal of this thesis is to improve the foundation of the Rosetta language. We introduced a stronger type system that includes cardinality constraints to prevent additional runtime errors, we modified the syntax to simplify the semantics and the generated code, and we provided an exact description of the meaning of expressions to give solid ground for developers of code generators; all unified in a full formal specification for the core features of the language. Additionally, we described a systematic approach to preserve well-typedness while implementing a code generator, which tackles problems caused by a discrepancy between the subtype relation of Rosetta and the subtype relation of a target language.

This thesis has made its impact. Minesh Patel, the manager of the REGnosys team developing Rosetta, declared that "[...] as requirements grow, the analysis and input Simon has provided will significantly impact future development." Furthermore, "REGnosys intends to implement the recommendation as part of the 2022 roadmap."

To the best of our knowledge, a type system that models cardinality constraints has never been described before in scientific literature. By modelling this explicitly, we are able to maintain a relatively simple type system as compared to languages with stronger type systems such as Agda or TypeScript. Besides its practical contribution, this thesis therefore also makes a contribution of theoretical interest outside the scope of Rosetta.

In Chapter 8, we saw that cardinality constraints are actually too strong in some cases, and reject useful models that do not result in runtime errors. Aside from the workaround proposed there which weakens the safety guarantees of our type system, a potential future extension for the type system could be *type narrowing*. This technique, used in type systems such as TypeScript[7] and mypy[26], makes it possible to convince the type checker that a broader type is actually more specific by

adding appropriate checks. Recall the example we gave in previous chapter.

```
func AddOrZero:
  inputs:
    a number (0..1)
    b number (0..1)
  output: result number (1..1)
  assign-output:
    if a exists and b exists then
      a + b
    else 0
```

Using type narrowing, the `exists` operator could convince the type system that variables `a` and `b` actually have list type `number` (1..1) when we add them, so the premises of rule TA-AddNumber are fulfilled and the function is well-typed. Note that type narrowing is contextual: in the "then" branch the type system knows that `a` and `b` have list type `number` (1..1), while in the "else" branch they have list type `number` (0..0). Outside of the conditional expression, they both have their original list type `number` (0..1).

Another approach to solve this issue is to add a feature similar to *pattern matching*. Functional programming languages often use this feature to perform a case analysis of the form "if the list is empty, do this, or if the list is non-empty, do that". Similarly, we could introduce a `match` operation such as in this example.

```
func ProcessList:
  inputs: a int (0..*)
  output: result int (2..2)
  assign-output:
    match a
      (0..0) [42, 0]
      (1..1) [a, a]
      (2..*) [42, a count]
```

When `a` is empty, we return $[42, 0]$, when `a` is a single item, we return $[\mathsf{a}, \mathsf{a}]$, and otherwise we return $[42, \mathsf{a}\ \texttt{count}]$. An advantage of this approach is that it is more transparent and more general than type narrowing: conditional expressions stay simple and we can perform a case analysis for arbitrary cardinalities.

Other interesting future work includes extending our small set of features with `enum`s, `condition`s, abstract functions, and so on. During the research of this thesis, new features such as mapping and filtering lists have been added to Rosetta, and seeing how they fit in would be intriguing as well.

Now that we have a formal specification that addresses the several issues that we discussed, an interesting question would be "Does this specification solve all issues?" The extensive testing discussed in Chapter 8 gives us some confidence, but does not give us the guarantees of a mathematical proof. It would certainly be interesting to see that correctness properties such as semantic soundness and the well-typedness of our code generator actually hold.

# Appendix A

# Typing Rules of Nouga

## A.1 Declarative typing rules

SPECIFICATION A.1: declarative typing rules of Nouga.

Subtyping $S <: T$.

$$\frac{}{T <: T} \quad \text{S-REFL}$$

$$\frac{S <: U \qquad U <: T}{S <: T} \quad \text{S-TRANS}$$

$$\frac{}{\texttt{int} <: \texttt{number}} \quad \text{S-NUM}$$

$$\frac{\mathrm{ET}(D) = \texttt{type}\ D\ \texttt{extends}\ E : \ldots}{D <: E} \quad \text{S-EXTENDS}$$

List subtyping $T_1\ C_1 <:^* T_2\ C_2$.

$$\frac{T_1 <: T_2 \qquad C_1 \subseteq C_2}{T_1\ C_1 <:^* T_2\ C_2} \quad \text{S-LIST}$$

Typing rules $\Gamma \vdash e : T\ C$.

$$\frac{\Gamma \vdash e : T_1\ C_1 \qquad T_1\ C_1 <:^* T_2\ C_2}{\Gamma \vdash e : T_2\ C_2} \quad \text{T-SUB}$$

$$\frac{}{\Gamma \vdash \texttt{True} : \texttt{boolean}\ (1..1)} \quad \text{T-TRUE}$$

$$\frac{}{\Gamma \vdash \texttt{False} : \texttt{boolean}\ (1..1)} \quad \text{T-FALSE}$$

$$\frac{}{\Gamma \vdash i : \texttt{int}\ (1..1)}\ \textsc{T-Int}$$

$$\frac{}{\Gamma \vdash r : \texttt{number}\ (1..1)}\ \textsc{T-Number}$$

$$\frac{x : T\ C \in \Gamma}{\Gamma \vdash x : T\ C}\ \textsc{T-Var}$$

$$\frac{\Gamma \vdash e_1 : \texttt{boolean}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{boolean}\ (1..1)}{\Gamma \vdash e_1\ \texttt{or}\ e_2 : \texttt{boolean}\ (1..1)}\ \textsc{T-Or}$$

$$\frac{\Gamma \vdash e_1 : \texttt{boolean}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{boolean}\ (1..1)}{\Gamma \vdash e_1\ \texttt{and}\ e_2 : \texttt{boolean}\ (1..1)}\ \textsc{T-And}$$

$$\frac{\Gamma \vdash e : \texttt{boolean}\ (1..1)}{\Gamma \vdash \texttt{not}\ e : \texttt{boolean}\ (1..1)}\ \textsc{T-Not}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{int}\ (1..1)}{\Gamma \vdash e_1 + e_2 : \texttt{int}\ (1..1)}\ \textsc{T-PlusInt}$$

$$\frac{\Gamma \vdash e_1 : \texttt{number}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{number}\ (1..1)}{\Gamma \vdash e_1 + e_2 : \texttt{number}\ (1..1)}\ \textsc{T-PlusNumber}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{int}\ (1..1)}{\Gamma \vdash e_1 * e_2 : \texttt{int}\ (1..1)}\ \textsc{T-MultInt}$$

$$\frac{\Gamma \vdash e_1 : \texttt{number}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{number}\ (1..1)}{\Gamma \vdash e_1 * e_2 : \texttt{number}\ (1..1)}\ \textsc{T-MultNumber}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{int}\ (1..1)}{\Gamma \vdash e_1 - e_2 : \texttt{int}\ (1..1)}\ \textsc{T-SubtInt}$$

$$\frac{\Gamma \vdash e_1 : \texttt{number}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{number}\ (1..1)}{\Gamma \vdash e_1 - e_2 : \texttt{number}\ (1..1)}\ \textsc{T-SubtNumber}$$

$$\frac{\Gamma \vdash e_1 : \texttt{number}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{number}\ (1..1)}{\Gamma \vdash e_1 / e_2 : \texttt{number}\ (1..1)}\ \textsc{T-Division}$$

$$\frac{\text{allattrs}(D) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n \qquad \forall i \in 1..n : \Gamma \vdash e_i : T_i\ C_i}{\Gamma \vdash D\,\{\, a_1 : e_1, \ldots, a_n : e_n \,\} : D\ (1..1)}\ \textsc{T-Instantiate}$$

$$\frac{\Gamma \vdash e : D\ C \qquad \text{allattrs}(D) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n}{\Gamma \vdash e \texttt{->} a_k : T_k\ C * C_k}\ \textsc{T-Project}$$

$$\frac{\Gamma \vdash e : T\ C \qquad (0..1) \subseteq C}{\Gamma \vdash e\ \texttt{exists} : \texttt{boolean}\ (1..1)}$$ T-Exists

$$\frac{\Gamma \vdash e : T\ C \qquad (1..1) \subseteq C \qquad C \neq (1..1)}{\Gamma \vdash e\ \texttt{single exists} : \texttt{boolean}\ (1..1)}$$ T-SingleExists

$$\frac{\Gamma \vdash e : T\ C \qquad (1..2) \subseteq C}{\Gamma \vdash e\ \texttt{multiple exists} : \texttt{boolean}\ (1..1)}$$ T-MultipleExists

$$\frac{\Gamma \vdash e : D\ (1..1) \qquad \text{allattrs}(D) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n \qquad \text{maybeempty}(D)}{\Gamma \vdash e\ \texttt{->}\ a_k\ \texttt{only exists} : \texttt{boolean}\ (1..1)}$$ T-OnlyExists

$$\frac{\Gamma \vdash e : T\ C}{\Gamma \vdash e\ \texttt{count} : \texttt{int}\ (1..1)}$$ T-Count

$$\frac{\Gamma \vdash e : T\ C}{\Gamma \vdash e\ \texttt{only-element} : T\ (0..1)}$$ T-OnlyElement

$$\frac{\Gamma \vdash e_1 : T_1\ C_1 \qquad \Gamma \vdash e_2 : T_2\ C_2 \qquad \text{comparable}^*(T_1\ C_1, T_2\ C_2)}{\Gamma \vdash e_1 = e_2 : \texttt{boolean}\ (1..1)}$$ T-Equals

$$\frac{\Gamma \vdash e_1 : T_1\ C_1 \qquad \Gamma \vdash e_2 : T_2\ C_2 \qquad \text{comparable}^*(T_1\ C_1, T_2\ C_2)}{\Gamma \vdash e_1\ \texttt{<>}\ e_2 : \texttt{boolean}\ (1..1)}$$ T-NotEquals

$$\frac{\Gamma \vdash e_1 : T_1\ C \qquad \Gamma \vdash e_2 : T_2\ (1..1) \qquad \text{comparable}(T_1, T_2)}{\Gamma \vdash e_1\ \texttt{all} = e_2 : \texttt{boolean}\ (1..1)}$$ T-AllEquals

$$\frac{\Gamma \vdash e_1 : T_1\ C \qquad \Gamma \vdash e_2 : T_2\ (1..1) \qquad \text{comparable}(T_1, T_2)}{\Gamma \vdash e_1\ \texttt{all <>}\ e_2 : \texttt{boolean}\ (1..1)}$$ T-AllNotEquals

$$\frac{\Gamma \vdash e_1 : T_1\ C \qquad \Gamma \vdash e_2 : T_2\ (1..1) \qquad \text{comparable}(T_1, T_2)}{\Gamma \vdash e_1\ \texttt{any} = e_2 : \texttt{boolean}\ (1..1)}$$ T-AnyEquals

$$\frac{\Gamma \vdash e_1 : T_1\ C \qquad \Gamma \vdash e_2 : T_2\ (1..1) \qquad \text{comparable}(T_1, T_2)}{\Gamma \vdash e_1\ \texttt{any <>}\ e_2 : \texttt{boolean}\ (1..1)}$$ T-AnyNotEquals

$$\frac{\Gamma \vdash e_1 : T_1\ C_1 \qquad \Gamma \vdash e_2 : T_2\ C_2 \qquad \text{comparable}(T_1, T_2)}{\Gamma \vdash e_1\ \texttt{contains}\ e_2 : \texttt{boolean}\ (1..1)}$$ T-Contains

$$\frac{\Gamma \vdash e_1 : T_1\ C_1 \qquad \Gamma \vdash e_2 : T_2\ C_2 \qquad \text{comparable}(T_1, T_2)}{\Gamma \vdash e_1\ \texttt{disjoint}\ e_2 : \texttt{boolean}\ (1..1)}$$ T-Disjoint

$$\frac{\forall i \in 1..n : \Gamma \vdash e_i : T\ C_i}{\Gamma \vdash [e_1, \ldots, e_n] : T\ \sum_{i \in 1..n} C_i} \quad \text{T-List}$$

$$\frac{\Gamma \vdash e_1 : \texttt{boolean}\ (1..1) \qquad \Gamma \vdash e_2 : T\ C \qquad \Gamma \vdash e_3 : T\ C}{\Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : T\ C} \quad \text{T-If}$$

$$\frac{\begin{array}{c}\text{inputs}(F) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n \\ \text{output}(F) = a\ T\ C \qquad \forall i \in 1..n : \Gamma \vdash e_i : T_i\ C_i\end{array}}{\Gamma \vdash F(e_1, \ldots, e_n) : T\ C} \quad \text{T-Func}$$

Typing function declarations $F$ OK.

$$\frac{\begin{array}{c}\text{inputs}(F) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n \\ \text{output}(F) = a\ T\ C \qquad a_1 : T_1\ C_1, \ldots, a_n : T_n\ C_n \vdash \text{op}(F) : T\ C\end{array}}{F\ \text{OK}}$$

## A.2 Algorithmic typing rules

Specification A.2: algorithmic typing rules of Nouga. The rules that are new or altered compared to the declarative version are in red.

Subtyping $S <: T$.

$$\frac{}{T <: T} \quad \text{SA-Refl}$$

$$\frac{}{\texttt{int} <: \texttt{number}} \quad \text{SA-Num}$$

$$\frac{E \in \text{ancestors}(D)}{D <: E} \quad \text{SA-Ancestor}$$

$$\frac{}{\texttt{nothing} <: T} \quad \text{SA-Nothing}$$

List subtyping $T_1\ C_1 <:^* T_2\ C_2$.

$$\frac{T_1 <: T_2 \qquad C_1 \subseteq C_2}{T_1\ C_1 <:^* T_2\ C_2} \quad \text{SA-List}$$

Typing rules $\Gamma \vdash e : T\ C$.

$$\frac{}{\Gamma \vdash \texttt{True} : \texttt{boolean}\ (1..1)} \quad \text{TA-True}$$

$$\frac{}{\Gamma \vdash \texttt{False} : \texttt{boolean}\ (1..1)}\ \text{TA-False}$$

$$\frac{}{\Gamma \vdash r : \texttt{number}\ (1..1)}\ \text{TA-Number}$$

$$\frac{}{\Gamma \vdash i : \texttt{int}\ (1..1)}\ \text{TA-Int}$$

$$\frac{x : T\ C \in \Gamma}{\Gamma \vdash x : T\ C}\ \text{TA-Var}$$

$$\frac{\Gamma \vdash e_1 : \texttt{boolean}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{boolean}\ (1..1)}{\Gamma \vdash e_1\ \texttt{or}\ e_2 : \texttt{boolean}\ (1..1)}\ \text{TA-Or}$$

$$\frac{\Gamma \vdash e_1 : \texttt{boolean}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{boolean}\ (1..1)}{\Gamma \vdash e_1\ \texttt{and}\ e_2 : \texttt{boolean}\ (1..1)}\ \text{TA-And}$$

$$\frac{\Gamma \vdash e : \texttt{boolean}\ (1..1)}{\Gamma \vdash \texttt{not}\ e : \texttt{boolean}\ (1..1)}\ \text{TA-Not}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{int}\ (1..1)}{\Gamma \vdash e_1 + e_2 : \texttt{int}\ (1..1)}\ \text{TA-PlusInt}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1\ (1..1) \qquad \Gamma \vdash e_2 : T_2\ (1..1) \\ T_1 <: \texttt{number} \qquad T_2 <: \texttt{number} \qquad T_1 = \texttt{number} \vee T_2 = \texttt{number}\end{array}}{\Gamma \vdash e_1 + e_2 : \texttt{number}\ (1..1)}\ \text{TA-PlusNumber}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{int}\ (1..1)}{\Gamma \vdash e_1 * e_2 : \texttt{int}\ (1..1)}\ \text{TA-MultInt}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1\ (1..1) \qquad \Gamma \vdash e_2 : T_2\ (1..1) \\ T_1 <: \texttt{number} \qquad T_2 <: \texttt{number} \qquad T_1 = \texttt{number} \vee T_2 = \texttt{number}\end{array}}{\Gamma \vdash e_1 * e_2 : \texttt{number}\ (1..1)}\ \text{TA-MultNumber}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int}\ (1..1) \qquad \Gamma \vdash e_2 : \texttt{int}\ (1..1)}{\Gamma \vdash e_1 - e_2 : \texttt{int}\ (1..1)}\ \text{TA-SubtInt}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1\ (1..1) \qquad \Gamma \vdash e_2 : T_2\ (1..1) \\ T_1 <: \texttt{number} \qquad T_2 <: \texttt{number} \qquad T_1 = \texttt{number} \vee T_2 = \texttt{number}\end{array}}{\Gamma \vdash e_1 - e_2 : \texttt{number}\ (1..1)}\ \text{TA-SubtNumber}$$

$$\frac{\Gamma \vdash e_1 : T_1\ (1..1) \qquad \Gamma \vdash e_2 : T_2\ (1..1) \qquad T_1 <: \texttt{number} \qquad T_2 <: \texttt{number}}{\Gamma \vdash e_1 / e_2 : \texttt{number}\ (1..1)}\ \text{TA-Division}$$

$$\frac{\text{allattrs}(D) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n \quad \forall i \in 1..n : \Gamma \vdash e_i : T_i'\ C_i' \qquad \forall i \in 1..n : T_i'\ C_i' <:^*\ T_i\ C_i}{\Gamma \vdash D\,\{\,a_1 : e_1, \ldots, a_n : e_n\,\} : D\ (1..1)}$$ TA-Instantiate

$$\frac{\Gamma \vdash e : D\ C \qquad \text{allattrs}(D) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n}{\Gamma \vdash e\ \texttt{->}\ a_k : T_k\ C * C_k}$$ TA-Project

$$\frac{\Gamma \vdash e : T\ C \qquad (0..1) \subseteq C}{\Gamma \vdash e\ \texttt{exists} : \texttt{boolean}\ (1..1)}$$ TA-Exists

$$\frac{\Gamma \vdash e : T\ C \qquad (1..1) \subseteq C \qquad C \neq (1..1)}{\Gamma \vdash e\ \texttt{single exists} : \texttt{boolean}\ (1..1)}$$ TA-SingleExists

$$\frac{\Gamma \vdash e : T\ C \qquad (1..2) \subseteq C}{\Gamma \vdash e\ \texttt{multiple exists} : \texttt{boolean}\ (1..1)}$$ TA-MultipleExists

$$\frac{\Gamma \vdash e : D\ (1..1) \quad \text{allattrs}(D) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n \qquad \text{maybeempty}(D)}{\Gamma \vdash e\ \texttt{->}\ a_k\ \texttt{only exists} : \texttt{boolean}\ (1..1)}$$ TA-OnlyExists

$$\frac{\Gamma \vdash e : T\ C}{\Gamma \vdash e\ \texttt{count} : \texttt{int}\ (1..1)}$$ TA-Count

$$\frac{\Gamma \vdash e : T\ C}{\Gamma \vdash e\ \texttt{only-element} : T\ (0..1)}$$ TA-OnlyElement

$$\frac{\Gamma \vdash e_1 : T_1\ C_1 \qquad \Gamma \vdash e_2 : T_2\ C_2 \qquad \text{comparable}^*(T_1\ C_1, T_2\ C_2)}{\Gamma \vdash e_1\ \texttt{=}\ e_2 : \texttt{boolean}\ (1..1)}$$ TA-Equals

$$\frac{\Gamma \vdash e_1 : T_1\ C_1 \qquad \Gamma \vdash e_2 : T_2\ C_2 \qquad \text{comparable}^*(T_1\ C_1, T_2\ C_2)}{\Gamma \vdash e_1\ \texttt{<>}\ e_2 : \texttt{boolean}\ (1..1)}$$ TA-NotEquals

$$\frac{\Gamma \vdash e_1 : T_1\ C \qquad \Gamma \vdash e_2 : T_2\ (1..1) \qquad \text{comparable}(T_1, T_2)}{\Gamma \vdash e_1\ \texttt{all =}\ e_2 : \texttt{boolean}\ (1..1)}$$ TA-AllEquals

$$\frac{\Gamma \vdash e_1 : T_1\ C \qquad \Gamma \vdash e_2 : T_2\ (1..1) \qquad \text{comparable}(T_1, T_2)}{\Gamma \vdash e_1\ \texttt{all <>}\ e_2 : \texttt{boolean}\ (1..1)}$$ TA-AllNotEquals

$$\frac{\Gamma \vdash e_1 : T_1\ C \qquad \Gamma \vdash e_2 : T_2\ (1..1) \qquad \text{comparable}(T_1, T_2)}{\Gamma \vdash e_1\ \texttt{any =}\ e_2 : \texttt{boolean}\ (1..1)}$$ TA-AnyEquals

$$\frac{\Gamma \vdash e_1 : T_1\ C \qquad \Gamma \vdash e_2 : T_2\ (1..1) \qquad \text{comparable}(T_1, T_2)}{\Gamma \vdash e_1\ \texttt{any <>}\ e_2 : \texttt{boolean}\ (1..1)}$$ TA-AnyNotEquals

$$\frac{\Gamma \vdash e_1 : T_1\ C_1 \qquad \Gamma \vdash e_2 : T_2\ C_2 \qquad \text{comparable}(T_1, T_2)}{\Gamma \vdash e_1\ \texttt{contains}\ e_2 : \texttt{boolean}\ \ (1..1)} \quad \text{TA-Contains}$$

$$\frac{\Gamma \vdash e_1 : T_1\ C_1 \qquad \Gamma \vdash e_2 : T_2\ C_2 \qquad \text{comparable}(T_1, T_2)}{\Gamma \vdash e_1\ \texttt{disjoint}\ e_2 : \texttt{boolean}\ \ (1..1)} \quad \text{TA-Disjoint}$$

$$\frac{\forall i \in 1..n : \Gamma \vdash e_i : T_i\ C_i \qquad T = \text{join}(T_1, \ldots, T_n)}{\Gamma \vdash [e_1, \ldots, e_n] : T\ \sum_{i \in 1..n} C_i} \quad \text{TA-List}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \texttt{boolean}\ \ (1..1) \\ \Gamma \vdash e_2 : T_1\ C_1 \qquad \Gamma \vdash e_3 : T_2\ C_2 \qquad T\ C = \text{join}^*(T_1\ C_1, T_2\ C_2)\end{array}}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T\ C} \quad \text{TA-If}$$

$$\frac{\begin{array}{c}\text{inputs}(F) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n \qquad \text{output}(F) = a\ T\ C \\ \forall i \in 1..n : \Gamma \vdash e_i : T_i'\ C_i' \qquad \forall i \in 1..n : T_i'\ C_i' <: T_n\ C_n\end{array}}{\Gamma \vdash F(e_1, \ldots, e_n) : T\ C} \quad \text{TA-Func}$$

Typing function declarations $F$ OK.

$$\frac{\begin{array}{c}\text{inputs}(F) = a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n \qquad \text{output}(F) = a\ T\ C \\ a_1 : T_1\ C_1, \ldots, a_n : T_n\ C_n \vdash \text{op}(F) : T'\ C' \qquad T'\ C' <: T\ C\end{array}}{F\ \text{OK}}$$

# Appendix B

# Semantics of expressions in Nouga

Specification B.1: denotations of expressions in Nouga.

$$\mathcal{E}\,[\![\texttt{True}]\!]\,S = [\mathit{true}] \qquad \text{E-True}$$

$$\mathcal{E}\,[\![\texttt{False}]\!]\,S = [\mathit{false}] \qquad \text{E-False}$$

$$\mathcal{E}\,[\![i]\!]\,S = [i] \qquad \text{E-Int}$$

$$\mathcal{E}\,[\![r]\!]\,S = [r] \qquad \text{E-Number}$$

$$\mathcal{E}\,[\![x]\!]\,S = S(x) \qquad \text{E-Var}$$

$$\mathcal{E}\,[\![e_1\ \texttt{or}\ e_2]\!]\,S = \textbf{let}\ [x] = \mathcal{E}\,[\![e_1]\!]\,S,\ [y] = \mathcal{E}\,[\![e_2]\!]\,S\ \textbf{in}\ [x \vee y] \qquad \text{E-Or}$$

$$\mathcal{E}\,[\![e_1\ \texttt{and}\ e_2]\!]\,S = \textbf{let}\ [x] = \mathcal{E}\,[\![e_1]\!]\,S,\ [y] = \mathcal{E}\,[\![e_2]\!]\,S\ \textbf{in}\ [x \wedge y] \qquad \text{E-And}$$

$$\mathcal{E}\,[\![\texttt{not}\ e]\!]\,S = \textbf{let}\ [x] = \mathcal{E}\,[\![e]\!]\,S\ \textbf{in}\ [\neg x] \qquad \text{E-Not}$$

$$\mathcal{E}\,[\![e_1\ \texttt{+}\ e_2]\!]\,S = \textbf{let}\ [x] = \mathcal{E}\,[\![e_1]\!]\,S,\ [y] = \mathcal{E}\,[\![e_2]\!]\,S\ \textbf{in}\ [x + y] \qquad \text{E-Plus}$$

$$\mathcal{E}\,[\![e_1\ \texttt{-}\ e_2]\!]\,S = \textbf{let}\ [x] = \mathcal{E}\,[\![e_1]\!]\,S,\ [y] = \mathcal{E}\,[\![e_2]\!]\,S\ \textbf{in}\ [x - y] \qquad \text{E-Subt}$$

$$\mathcal{E}\,[\![e_1 \texttt{ * } e_2]\!]\,S = \textbf{let } [x] = \mathcal{E}\,[\![e_1]\!]\,S, \quad [y] = \mathcal{E}\,[\![e_2]\!]\,S \textbf{ in } [x * y] \qquad \text{E-Mult}$$

$$\mathcal{E}\,[\![e_1 \texttt{ / } e_2]\!]\,S = \textbf{let } [x] = \mathcal{E}\,[\![e_1]\!]\,S, \quad [y] = \mathcal{E}\,[\![e_2]\!]\,S \textbf{ in } [x/y] \qquad \text{E-Div}$$

$$\mathcal{E}\,[\![D\,\{\,a_1 : e_1, \ldots, a_n : e_n\,\}]\!]\,S = [(D, \mathcal{E}\,[\![e_1]\!]\,S, \ldots, \mathcal{E}\,[\![e_n]\!]\,S)] \qquad \text{E-Instantiate}$$

$$\mathcal{E}\,[\![e \texttt{ -> } a]\!]\,S = \textbf{let } [x_1, \ldots, x_n] = \mathcal{E}\,[\![e]\!]\,S \textbf{ in } \text{flatten}_n(\text{project}_a(x_1), \ldots, \text{project}_a(x_n)) \qquad \text{E-Project}$$

$$\mathcal{E}\,[\![e \texttt{ exists}]\!]\,S = \begin{cases} [\mathit{true}], & \text{if } \text{count}(\mathcal{E}\,[\![e]\!]\,S) \geq 1 \\ [\mathit{false}], & \text{otherwise} \end{cases} \qquad \text{E-Exists}$$

$$\mathcal{E}\,[\![e \texttt{ single exists}]\!]\,S = \begin{cases} [\mathit{true}], & \text{if } \text{count}(\mathcal{E}\,[\![e]\!]\,S) = 1 \\ [\mathit{false}], & \text{otherwise} \end{cases} \qquad \text{E-SingleExists}$$

$$\mathcal{E}\,[\![e \texttt{ multiple exists}]\!]\,S = \begin{cases} [\mathit{true}], & \text{if } \text{count}(\mathcal{E}\,[\![e]\!]\,S) \geq 2 \\ [\mathit{false}], & \text{otherwise} \end{cases} \qquad \text{E-MultipleExists}$$

$$\mathcal{E}\,[\![e \texttt{ -> } a_i \texttt{ only exists}]\!]\,S = \textbf{let } [(D, v_1, \ldots, v_n)] = \mathcal{E}\,[\![e]\!]\,S, \quad a_1\ T_1\ C_1, \ldots, a_n\ T_n\ C_n = \text{allattrs}(D) \textbf{ in } \begin{cases} [\mathit{true}], & \text{if } \text{count}(v_i) \geq 1\ \wedge\ \forall j \in 1..n : \\ & \quad i \neq j \Rightarrow \text{count}(v_j) = 0 \\ [\mathit{false}], & \text{otherwise} \end{cases} \qquad \text{E-OnlyExists}$$

$$\mathcal{E}\,[\![e \texttt{ count}]\!]\,S = [\text{count}(\mathcal{E}\,[\![e]\!]\,S)] \qquad \text{E-Count}$$

$$\mathcal{E}\,[\![e \texttt{ only-element}]\!]\,S = \textbf{let } v = \mathcal{E}\,[\![e]\!]\,S \textbf{ in } \begin{cases} v, & \text{if } \text{count}(v) = 1 \\ [\,], & \text{otherwise} \end{cases} \qquad \text{E-OnlyElement}$$

$$\mathcal{E}\,[\![e_1 \texttt{ = } e_2]\!]\,S = \text{equals}^*(\mathcal{E}\,[\![e_1]\!]\,S, \mathcal{E}\,[\![e_2]\!]\,S) \qquad \text{E-Equals}$$

$$\mathcal{E}\,[\![e_1 \texttt{ <> } e_2]\!]\,S = \textbf{let } [x_1, \ldots, x_m] = \mathcal{E}\,[\![e_1]\!]\,S, \quad [y_1, \ldots, y_n] = \mathcal{E}\,[\![e_2]\!]\,S \textbf{ in } \begin{cases} [\mathit{true}], & \text{if } m \neq n\ \vee \\ & \forall i \in 1..n : \neg\,\text{equals}_(\,x_i, y_i) \\ [\mathit{false}], & \text{otherwise} \end{cases} \qquad \text{E-NotEquals}$$

$$\mathcal{E}\,[\![e_1\ \texttt{all =}\ e_2]\!]\,S = \textbf{let}\ [x_1,\ldots,x_n] = \mathcal{E}\,[\![e_1]\!]\,S,\ [y] = \mathcal{E}\,[\![e_2]\!]\,S \ \textbf{in}\ \begin{cases}[\mathit{true}], & \text{if } \forall i \in 1..n : \text{equals}_(\ x_i, y)\\ [\mathit{false}], & \text{otherwise}\end{cases} \qquad \text{E-AllEquals}$$

$$\mathcal{E}\,[\![e_1\ \texttt{all <>}\ e_2]\!]\,S = \textbf{let}\ [x_1,\ldots,x_n] = \mathcal{E}\,[\![e_1]\!]\,S,\ [y] = \mathcal{E}\,[\![e_2]\!]\,S \ \textbf{in}\ \begin{cases}[\mathit{true}], & \text{if } \forall i \in 1..n : \neg\,\text{equals}(x_i, y)\\ [\mathit{false}], & \text{otherwise}\end{cases} \qquad \text{E-AllNotEquals}$$

$$\mathcal{E}\,[\![e_1\ \texttt{any =}\ e_2]\!]\,S = \textbf{let}\ [x_1,\ldots,x_n] = \mathcal{E}\,[\![e_1]\!]\,S,\ [y] = \mathcal{E}\,[\![e_2]\!]\,S \ \textbf{in}\ \begin{cases}[\mathit{true}], & \text{if } \exists i \in 1..n : \text{equals}(x_i, y)\\ [\mathit{false}], & \text{otherwise}\end{cases} \qquad \text{E-AnyEquals}$$

$$\mathcal{E}\,[\![e_1\ \texttt{any <>}\ e_2]\!]\,S = \textbf{let}\ [x_1,\ldots,x_n] = \mathcal{E}\,[\![e_1]\!]\,S,\ [y] = \mathcal{E}\,[\![e_2]\!]\,S \ \textbf{in}\ \begin{cases}[\mathit{true}], & \text{if } \exists i \in 1..n : \neg\,\text{equals}(x_i, y)\\ [\mathit{false}], & \text{otherwise}\end{cases} \qquad \text{E-AnyNotEquals}$$

$$\mathcal{E}\,[\![e_1\ \texttt{contains}\ e_2]\!]\,S = \textbf{let}\ [x_1,\ldots,x_m] = \mathcal{E}\,[\![e_1]\!]\,S,\ [y_1,\ldots,y_n] = \mathcal{E}\,[\![e_2]\!]\,S \ \textbf{in}\ \begin{cases}[\mathit{true}], & \text{if } \forall j \in 1..n :\\ & \quad \exists i \in 1..m : \text{equals}(x_i, y_j)\\ [\mathit{false}], & \text{otherwise}\end{cases} \qquad \text{E-Contains}$$

$$\mathcal{E}\,[\![e_1\ \texttt{disjoint}\ e_2]\!]\,S = \textbf{let}\ [x_1,\ldots,x_m] = \mathcal{E}\,[\![e_1]\!]\,S,\ [y_1,\ldots,y_n] = \mathcal{E}\,[\![e_2]\!]\,S \ \textbf{in}\ \begin{cases}[\mathit{true}], & \text{if } \forall i \in 1..m :\\ & \quad \forall j \in 1..n : \neg\,\text{equals}(x_i, y_j)\\ [\mathit{false}], & \text{otherwise}\end{cases} \qquad \text{E-Disjoint}$$

$$\mathcal{E}\,[\![[e_1,\ldots,e_n]]\!]\,S = \text{flatten}_n(\mathcal{E}\,[\![e_1]\!]\,S,\ldots,\mathcal{E}\,[\![e_n]\!]\,S) \qquad \text{E-List}$$

$$\mathcal{E}\,[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]\,S = \textbf{let}\ [x] = \mathcal{E}\,[\![e_1]\!]\,S \ \textbf{in}\ \begin{cases}\mathcal{E}\,[\![e_1]\!]\,S, & \text{if } x = \mathit{true}\\ \mathcal{E}\,[\![e_2]\!]\,S, & \text{otherwise}\end{cases} \qquad \text{E-If}$$

$$\mathcal{E}\,[\![F(e_1,\ldots,e_n)]\!]\,S = \textbf{let}\ a_1\ T_1\ C_1,\ldots,a_n\ T_n\ C_n = \text{inputs}(F) \ \textbf{in}\ \mathcal{E}\,[\![\text{op}(F)]\!]\,[\ a_1 \mapsto \mathcal{E}\,[\![e_1]\!]\,S,\ldots, a_n \mapsto \mathcal{E}\,[\![e_n]\!]\,S\ ] \qquad \text{E-Func}$$

# Bibliography


[1] FIX Trading Community. *FIX Trading Community*. URL: https://www.fixtrading.org/, last checked on 5 Jan. 2022.

[2] ISDA. *Financial products Markup Language*. URL: https://www.fpml.org/, last checked on 5 Jan. 2022.

[3] European Federation of Energy Traders. *EFET Communication Standard*. URL: https://www.efet.org/home/documents?id=660, last checked on 5 Jan. 2022.

[4] International Swaps and Derivatives Association. *ISDA*. URL: https://www.isda.org/, last checked on 28 Nov. 2021.

[5] REGnosys and ISDA. *Rosetta Documentation*. URL: https://docs.rosetta-technology.io/index.html, last checked on 28 Nov. 2021.

[6] *Agda documentation*. URL: https://agda.readthedocs.io/en/v2.6.2.1/getting-started/what-is-agda.html, last checked on 12 Jan. 2022.

[7] Microsoft. *TypeScript: JavaScript with syntax for types*. URL: https://www.typescriptlang.org/, last checked on 12 Jan. 2022.

[8] Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.

[9] David A Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown Publishers, 1986.

[10] *C Operator Precedence*. URL: https://en.cppreference.com/w/c/language/operator_precedence, last checked on 1 Jan. 2022.

[11] Jeremy G Siek. "Revisiting Elementary Denotational Semantics". In: *arXiv preprint arXiv:1707.03762* (2017).

[12] James Gosling et al. *The Java Language Specification, Java SE 12 Edition*. URL: https://docs.oracle.com/javase/specs/jls/se12/html/index.html, last checked on 1 Jan. 2022.

[13] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[14] zneak (https://stackoverflow.com/users/251153/zneak). *Why not use Double or Float to represent currency?* URL: https://stackoverflow.com/a/3730040/3083982, last checked on 2 Jan. 2022.

[15] *Liskov substitution principle*. URL: https://en.wikipedia.org/wiki/Liskov_substitution_principle, last checked on 13 Jan. 2022.

[16] *Prettier Java*. URL: https://prettier-java.tech/, last checked on 12 Jan. 2022.

[17] Simon Cockx. *Implementation of the Nouga language*. URL: https://github.com/SimonCockx/nouga, last checked on 12 Jan. 2022.

[18] Buhake Sindi (https://stackoverflow.com/users/251173/buhake-sindi). *BigDecimal equals() versus compareTo()*. URL: https://stackoverflow.com/q/6787142/3083982, last checked on 6 Jan. 2022.

[19] David J. (https://stackoverflow.com/users/109618/david-j). *Set specific precision of a BigDecimal*. URL: https://stackoverflow.com/a/20365270/3083982, last checked on 6 Jan. 2022.

[20] Steven Schlansker (https://stackoverflow.com/users/171061/steven-schlansker). *Java collections covariance problem*. URL: https://stackoverflow.com/a/3763220/3083982, last checked on 7 Jan. 2022.

[21] Google. *Google Guice*. 2021. URL: https://github.com/google/guice.

[22] Eclipse. *Xtext - Language engineering made easy*. URL: https://www.eclipse.org/Xtext/, last checked on 9 Jan. 2022.

[23] Lorenzo Bettini. *Xsemantics*. URL: https://github.com/eclipse/xsemantics, last checked on 9 Jan. 2022.

[24] Lorenzo Bettini. "Type errors for the IDE with Xtext and Xsemantics". In: *Open Computer Science* 9.1 (2019), pp. 52–79.

[25] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. "Composing contracts: an adventure in financial engineering". In: *ACM SIG-PLAN Notices* 35.9 (2000), pp. 280–292.

[26] Jukka Lehtosalo et al. *mypy: Optional Static Typing for Python*. URL: http://mypy-lang.org/, last checked on 12 Jan. 2022.