NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

JOANA BAPTISTA PARREIRA
BSc in Science and Computer Engineering

FROM AN ONTOLOGY FOR PROGRAMMING TO A TYPE-SAFE TEMPLATE LANGUAGE

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
July, 2022

FROM AN ONTOLOGY FOR PROGRAMMING TO A TYPE-SAFE TEMPLATE LANGUAGE

JOANA BAPTISTA PARREIRA

BSc in Science and Computer Engineering

Adviser: João Costa Seco
Associate Professor, NOVA University Lisbon

Co-adviser: Carla Ferreira
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
July, 2022

From an Ontology for Programming to a Type-Safe Template Language

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.9.6) [34].

To my beloved mother.

Acknowledgements

I would first like to thank my thesis advisors, João Costa Seco and Carla Ferreira, for all the guidance and support. I want to thank João for constantly pushing me to be my best self and for all the incredible opportunities he provided me. And I would like to thank Carla for being so genuinely kind and compassionate and for seeing her students as the vulnerable human beings that we are. I also want to express my gratitude to Bernardo Toninho for always being available to answer any questions and provide feedback throughout this work; and to all the other teachers that impacted me and that helped and encouraged me throughout these years.

I want to thank the GOLEM project for providing me with a grant, allowing me to work full-time on this thesis, and for the opportunity to work on such an incredible project with such talented persons. I would also like to acknowledge everyone involved in the GOLEM project for all the advice and input.

I want to thank my friends for always being there for me during my victories but also in my bad moments. I thank my study buddies, gym buddies, church buddies, dog-walk buddies, archery buddies, confidants and long-distance friends. You know who you are!

I must express my very profound gratitude to my family, especially to my mother, my warrior, who provided me with unfailing support and taught me firsthand to never give up. Without her, I could not have achieved all of this. I also thank my boyfriend for his encouragement and continuous help throughout this dissertation and for being a true teammate and partner. This man was always there to cheer me up on hard days. I cannot forget to mention all the pets in my life and my dog, Estrela, for keeping me company during this pandemic and for all the affection and joy she provides me.

Words are not enough to express my gratitude to all of you!

Last but foremost, I am grateful to God for everything He has done, for He is my strength and my shield.

“And whatever you do, do it heartily, as to the Lord and not to men ” (Colossians 3:23)

Abstract

The demand to develop more applications in a faster way has been increasing over the years. Even non-experienced developers are jumping into the market thanks to low-code platforms such as OutSystems.

The main goal of the GOLEM project is the development of the next generation of low-code software development, aiming to automate programming and make the OutSystems platform easier to use. This work is integrated into the GOLEM project and focuses 1) on designing an ontology that will be used to capture concepts from a user dialogue; 2) on formalizing a template language; and 3) producing a reference implementation for OSTRICH.

The ontology concepts must be representative enough to allow the generation of an application. A domain-specific language (DSL) produced in the scope of the GOLEM project will analyse the captured concepts, generating a set of operations that incrementally build and modify the target application.

Because some of those application components are common patterns, they can be pre-assembled into templates to be later re-used. OSTRICH, a type-safe template language for the OutSystems platform, allows for the definition and instantiation of type-safe templates while ensuring a clear separation between compile-time and runtime computations. We formalize this two-stage language, defining its syntax, type system and operational semantics. We also produce a reference implementation and introduce new features: parametric polymorphism and a simplified form of type dependency. Such features enable instantiating the top ten most-commonly used OutSystems templates, which are more than half of all template instantiations in the platform. These templates ease and fasten the development process, reducing the knowledge required to build OutSystems' applications.

Keywords: Metaprogramming, Ontology, Low-Code, Template Language, Type-Safety, Staged Computation, Parametric Polymorphism, Dependencies Between Types

Resumo

A necessidade de desenvolver aplicações a um ritmo cada vez mais acelerado tem aumentado ao longo dos anos. Mesmo programadores sem experiência têm vindo a integrar o mercado de trabalho nesta área, graças a plataformas *low-code* como a OutSystems.

O projeto GOLEM tem como objetivo o desenvolvimento da próxima geração de *low-code*, visando automatizar a programação e facilitar o uso da plataforma OutSystems. O objetivo desta dissertação, parte do projeto GOLEM, é 1) desenvolver uma ontologia para captar conceitos de um diálogo com o utilizador; 2) formalizar uma linguagem de templates; e 3) desenvolver uma implementação referência para o OSTRICH.

Os conceitos da ontologia devem ser suficientemente representativos de forma a permitir a criação de uma aplicação. Uma linguagem de domínio específico (DSL) criada no escopo do projeto GOLEM irá analisar os conceitos captados e gera um conjunto de operações que constroem e modificam a aplicação-alvo incrementalmente.

Visto alguns desses componentes da aplicação corresponderem a padrões comuns, estes podem ser previamente agregados num *template* para que possam ser reutilizados posteriormente. OSTRICH, uma linguagem de templates com segurança de tipos da plataforma OutSystems, permite definir e instanciar *templates* que respeitam as restrições de tipos, garantindo uma separação clara entre computações que ocorram em tempo de compilação e de execução.

Nós formalizamos esta linguagem de duas etapas, definindo a sua sintaxe, sistema de tipos, e semântica operacional. Também desenvolvemos uma implementação de referência e introduzimos novas funcionalidades: polimorfismo paramétrico, e uma forma simplificada de dependência entre tipos. Estas funcionalidades permitem instanciar os dez *templates* OutSystems mais usados, correspondendo a mais do que metade das instanciações de *templates* na plataforma. Estes *templates* facilitam e aceleram o processo de desenvolvimento, reduzindo o conhecimento necessário exigido ao programador para a construção de aplicações Outsystems.

Palavras-chave: Metaprogramação, Ontologia, Low-Code, Linguagem de Templates, Segurança de Tipos, Computação por Etapas, Polimorfismo Paramétrico, Dependências entre Tipos

Contents

List of Figures

List of Listings

1 Introduction

Our world is becoming more and more digital. In our day-to-day lives, we rely on technology more than we might notice. From searching for the best route for a new restaurant to booking a seat in the theatres or checking the balance in our bank account, the internet is present in a variety of our daily tasks. This increases the demand in the application development market. However, the shortage of software developers is bringing non-experienced programmers into the market. These circumstances promote the expansion of meta-programming tools and low-code development platforms, which ease and automate the development process. However, efforts to simplify the way we program are not a novelty.

Since the development of the first programming languages, there has been a continuous attempt to evolve toward more intuitive languages. The assembly languages used in the 1940s, when the first electronic computer was created, were difficult to understand and lacked any modular programming features, giving rise to the need for higher-level programming languages with a syntax that is easier to write and comprehend. Fortran was invented to answer those needs, together with Cobol and other subsequent languages emerged, always trying to fill the holes left by the previous languages, allowing programmers to handle more complex tasks, and expanding their use beyond scientific and numeric computing [39].

This evolution in programming languages led to the appearance of some technologies in the 1980s and 1990s, such as fourth-generation programming languages (4GL) or early rapid application development tools (RAD), that intended to aid programmers in their tasks [40]. These technologies laid the foundations for the appearance of low-code development and were followed by low-code development platforms and technologies (such as OutSystems, which was founded in 2001). However, the concept itself was not born until 2014, when it was used for the first time [35].

Low-code development platforms allow the development of applications with a minimum of coding, resorting to visual interfaces like drag-and-drop. These platforms are on the rise [35, 40, 32] because the growth in customized application demand has far exceeded the supply of software engineers [32]. The use of low-code platforms allows users to develop applications much faster. Additionally, by enabling non-technical users to develop applications, they unburden programmers and bring innovation and different perspectives into projects [35].

This thesis seeks to assist in the advancement of low-code platforms and automated programming.

1.1 Problem Statement

The OutSystems platform is a low-code development platform that automates the development and eases the deployment process. This platform resorts to a visual interface with drag-and-drop mechanisms. It also contains pre-built screens that can be reused in different applications. However, the developer must manually adjust them to their own data. This task may be cumbersome and require some programming skills and an understanding of the system architecture.

In its resolution to further simplify the development of applications, OutSystems launched the GOLEM project. The GOLEM project is a large-scale research project from the Carnegie Mellon Portugal program and led by OutSystems. Its main goal is the development of the next generation of low-code, aiming to revolutionize the application development experience, automate programming, and make OutSystems' low-code technology easier to use. This project aims to determine the most natural way for a user with no development experience to create applications without writing any code and how to implement such a system.

The GOLEM project contains four research threads related to language-based approaches. Information flows between the threads' components, forming a pipeline. First, a component for natural-language processing analyses the user requests written in natural language. This component gathers key concepts and maps them in an ontology (a formal representation of a domain). A second component, a domain-specific language (DSL), translates the previously obtained ontology into a set of operations. These operations may create, update or replace OutSystems' elements, incrementally building an application according to the developer's requests. Other DSL operations target a data-manipulation component. This component creates and modifies the application's data layer accordingly. Some DSL operations might target the generation of template screens. A templating language component, the OSTRICH language, provides a way of reusing pre-built screens and applying them to different data by supporting the definition of templates with input parameters. OSTRICH ensures the automatic adjustment of templates to the desired data, turning this hour-long process into an immediate event. However, in [20], it still has some limitations regarding the type of restrictions between template parameters.

This dissertation describes our work within the project, travelling through a portion of the GOLEM's pipeline process. We start with the design of an ontology that better bridges the gap between the user dialogue and the DSL. The concepts present in the ontology must be carefully chosen. Mapping from a natural language discourse to such concepts should be easy. Additionally, they should carry enough information for the generation of DSL operations that shape the intended OutSystems' application. Then, we continue with the formalization and extension of the template language, OSTRICH. We formalize it as a two-stage computation language, separating its compile and runtime computations. We add parametric polymorphism to template definitions, and dependencies between its parameters' types, allowing us to further restrict template instantiations.

1.2 Contributions

Our contributions include:

- the design of an ontology [45, 46] whose concepts represent the core components of an application. The user dialogue must be mapped to concepts in the ontology, and the result should be expressive enough to be able to translate it to a set of operations that build the intended application;
- the formalization of the OSTRICH language with staged computation, and the defintion of its syntax, type system, and operational semantics; and
- a reference implementation of the OSTRICH language, and an extension with parametric polymorphism and dependencies between template parameters [41–44].

The work developed in the context of this thesis also gave place to six papers:

- a published paper to the 13th International Conference on Knowledge Engineering and Ontology Development (KEOD/IC3K), entitled: *An Ontology based Task Oriented Dialogue* [45], which won a Best Paper Award, and whose main author is João Silva. We contributed by helping with the ontology development;
- a subsequent paper submission to the SN Computer Science journal, entitled: *An Ontology based Task Oriented Dialogue to Create OutSystems Applications* [46], which is still waiting for feedback, and whose main author is João Silva. We contributed with an extension to the previous paper by explaining the application building process after receiving information from the ontology;
- an accepted (under revision) paper to the International Journal on Software and Systems Modeling (SoSyM), entitled: *OSTRICH - A Rich Template Language for Low-code Development (Extended version)* [41], whose main author is Hugo Lourenço. We contributed with a typechecker extension for expressions with dependencies, and evaluation;
- a published paper as single author, presented at the PhD Symposium at the 17th International Conference on integrated Formal Methods (PhD-iFM'22), later published to the Lecture Notes in Computer Science, entitled: *Simple Dependent Types*

for OSTRICH [43]. We presented features of our implementation, namely staged computation and dependencies between parameters;

- an accepted paper to the 25th Internation Conference on Model Driven Engineering Languages and Systems (MODELS), entitled: *Nested OSTRICH - Hatching Compositions of Low-code Templates* [44], whose main author is João Costa Seco. We contributed with the typing algorithm, and the reference implementation[1] and evaluation;

- an accepted paper to the INForum 2022 - Simpósio de Informática, entitled *Type-Safe Customization of Low-code Templates* [42], whose main author is Constança Manteigas. Our work served as basis for the extension presented in that paper, a template customization mechanism.

1.3 Document Structure

The structure of this document is the following:

In Chapter 2, we explain some basic background concepts that are crucial to understanding our work, focusing on ontologies and description logics.

In Chapter 3, we go over some related work about using concepts for application development, ontology-based development, model-driven development, metaprogramming and staged computation.

In Chapter 4, we introduce and describe the GOLEM project, its goals and components. We also introduce the OSTRICH language.

We then describe the first part of our approach, in Chapter 5, where we present the ontologies we developed.

In Chapter 6, we present the formalization of the template language (syntax, type system and operational semantics), and some of the features we extend it with.

In Chapter 7, we present the evaluation of our contribution, and a representative case study of our reference implementation with a template definition, its typing and compile-time and runtime execution.

Finally, in Chapter 8, we express the concluding remarks.

[1]Please refer to the link `https://github.com/jbp182/OSTRICH-OCaml`

2 Background

In this chapter, we present some contextual information and essential definitions to understand our thesis. We explain what description logics are, their characteristics, and how they can be helpful for the representation of our domain.

2.1 Description Logics

An ontology is a set of concepts and relationships between those concepts and is used to formally represent knowledge within a domain of interest [14]. Since the interface of this work encloses ontology-like representations, we adopt ontologies to represent features of OutSystems applications. We resort to Description Logics and Web Ontology Language (OWL2) because of their expressiveness and decidability.

2.1.1 Definition

Description Logics (DL) are a family of formal knowledge representation languages [23]. They describe classes of objects and their relationships under a specific domain. DLs are a type of first-order logic (FOL) [12], meaning that any DL expression can be translated into FOL formulas without losing its meaning. However, translating FOL formulas to DL expressions is not possible in the general case without losing information, as certain FOL constructs (such as variables) cannot be encoded in DL. The motivation for the absence of certain FOL constructs in DL is to allow DL solvers to be decidable and more efficient, while trying to preserve as much expressiveness from FOLs as possible [12]. Comparing the two, constants in FOL are known as individuals in DL, unary predicates as concepts, and binary predicates are known as roles.

ALC (Attributive Concept Language with Complements) logic is one of the main description logics, and its central feature is the ability to describe complex classes [12], with the help of some concept constructors:

– Bottom ($\bot$)

– Top ($\top$)

- Atomic concept (A)
- Atomic negation ($\neg A$)
- Conjunction ($A \sqcap B$)
- Disjunction ($A \sqcup B$)
- Existential restriction of a concept A by a role R ($\exists R.A$)
- Universal restriction of a concept A by a role R ($\forall R.A$)

For instance, using a more illustrative example of existential restriction, we can say that at least one Post (concept) is created (role) by writing $\exists creates.Post$. An example of universal restriction would be $\forall creates.Post$, which means that only Posts are created (assuming that one could also create forms or other components).

There are extensions to ALC logic that add some additional constructors [4, 12], some of which we describe here:

- F: functional roles, allow us to state that an object can only be related through a specific role to exactly one other object. E.g., for all functional roles P, if $(a,b) : P$ and $(a,c) : P$, then b and c are the same individual.
- H: role hierarchy, allows nesting of roles, i.e., roles can have sub roles.
- R: role inclusion axioms; reflexivity and irreflexivity; role disjointness.
- S: transitive roles, e.g., knowing that $(a,b) : P$ and $(b,c) : P$, if P is transitive, then $(a,c) : P$.
- I: inverse roles. If there is a role R, we can define its inverse role R^-. Hence, if R connects the individuals a and b, written as $(a,b) : R$, we can specify its inverse role, $(b,a) : R^-$.
- N: number restrictions of cardinal roles. We can define the minimum and maximum cardinality k of a certain role R, e.g. $(\geq kR)$ and $(\leq kR)$, respectively. A more illustrative example is $(\geq 2hasChild)$, which means having two or more children.
- Q: qualified number restrictions of cardinal roles, defining cardinality limits k of a certain role R, that is connected to a specific concept C. E.g. $(\geq kR.C)$ or $(\leq kR.C)$. A more illustrative example is $(\geq 2hasChild.Doctor)$, meaning that there are two or more children that are doctors.
- O: nominals. It allows us to define concepts (classes) as an enumerate of individuals. E.g. $\forall hasColor.\{red, green, blue\}$.
- $^{(D)}$: use of datatypes for concrete domains. E.g., $hasAge.(\geq 21)$.
- C: full negation. E.g. $\neg(Doctor \sqcup Lawyer)$.

Different DLs, with different extensions, are chosen according to their purposes. For example, OWL Lite is based on $\mathcal{SHIF}^{(D)}$, OWL-DL on $\mathcal{SHOIN}^{(D)}$, OWL1 on $\mathcal{SHOIQ}$ and OWL2, the one we will use to describe our ontology, is based on $\mathcal{SROIQ}^{(D)}$ [12].

ALC has logical formulas known as axioms. A Description Logics ontology is a pair $O = < T, A >$, where T is a *TBox* and A is an *ABox* [12]. A *TBox* (terminological box) specifies knowledge about concepts and roles, describing the domain concepts (equivalent to a class hierarchy), hence a *TBox* is composed by axioms that specify relations of the type "class-subclass"($A \subset B$). An *ABox* (assertional box) is a set of facts and specifies properties of objects. An *ABox* contains axioms of the type "class individual", describing when an object a belongs to a class C ($a : C$); and of the type "relationship individual", describing a relationship P between two objects ($(a, b) : P$).

To clarify these concepts, Figure 2.1 depicts a small ontology written in DL, discriminating $TBox$ and $ABox$ assertions [4]. The use of “$\equiv$” means that the conditions on the right-hand side of the assertion are *necessary* and *sufficient* to describe the concept on the left-hand side. Necessary conditions mean that if an individual is classified as belonging to a concept (class), then it necessarily fulfils these conditions. But we cannot assert that every individual that fulfils these conditions automatically belongs to that class just by using necessary conditions. For that, we need sufficient conditions. The use of “$\sqsubseteq$” means that the conditions are necessary to describe the concept, but not sufficient.

In the TBox from the example, we can see that a father is a male human that has children (expression 2.1). This assertion resorts to the use of “$\equiv$”, therefore, every individual classified as a father is a male human with at least one child, and also every individual that is known to be a male human with at least one child is also a father. An individual that is a happy father (expression 2.2), necessarily needs to to be a father that only has children that are either doctors, lawyers or happy people. A happy ancestor only has descendants that are happy fathers (expression 2.3). And a teacher is neither a doctor nor a lawyer (expression 2.4). In expression 2.5 we can see that hasChild is a sub role of the descendant role. And in expression 2.6 we can see that hasFather is a sub role of the inverse of hasChild, meaning that when some individual A connects to individual B through hasChild (B is child of A), then B can connect to A through hasFather. It is a sub role if we remember that not everyone that has a child is the father, because they can be the mother. In expression 2.7 there is additional information about the properties of the roles. The role hasFather is functional because, in the domain of the example, one can only have one father. The role descendant is transitive, therefore if A is descendant of B, and B of C, then A is also descendant of C. In other words, this can mean that not only children are descendant of their parents, but also that grandchildren are descendant of their grandparents, and so on.

In the ABox from the example, the membership axioms specify when an individual belongs to a class (expression 2.8 and expression 2.10), or when there is a role between two individuals (expression 2.9).

$\boxed{TBox}$

Inclusion assertions on concepts:

$$Father \equiv Human \sqcap Male \sqcap \exists hasChild \tag{2.1}$$
$$HappyFather \sqsubseteq Father \sqcap \forall hasChild.(Doctor \sqcup Lawyer \sqcup HappyPerson) \tag{2.2}$$
$$HappyAnc \sqsubseteq \forall descendant.HappyFather \tag{2.3}$$
$$Teacher \sqsubseteq \neg Doctor \sqcap \neg Lawyer \tag{2.4}$$

Inclusion assertions on roles:

$$hasChild \sqsubseteq descendant \tag{2.5}$$
$$hasFather \sqsubseteq hasChild^{-} \tag{2.6}$$

Property assertions on roles:

$$(\textbf{transitive}\ \text{descendant}),\ (\textbf{functional}\ \text{hasFather}) \tag{2.7}$$

$\boxed{ABox}$

Membership assertions:

$$mary : Teacher \tag{2.8}$$
$$(mary, john) : hasFather \tag{2.9}$$
$$john : HappyAnc \tag{2.10}$$

Figure 2.1: Example of TBox and ABox assertions [4].

2.1.2 From UML Class Diagrams to Description Logics

It is possible to define an ontology in DL from UML class diagrams (or even from entity-relationship diagrams [10]). UML class diagrams may more easily express our initial domain and we can translate it to DL afterwards, since UML class diagrams are in tight correspondence with ontology languages, and can even be viewed as an ontology language outright [4].

Starting with the basics of the translation from UML class diagrams to DL, each class in UML is represented by an atomic concept in DL, each attribute is represented by a role, and each binary association between classes is also represented by a role in DL. Things get a little more complicated when there are non-binary associations, since roles in DL can only connect two concepts. The solution is to reify each non-binary association, i.e., to represent it as a concept and connect it by roles to the other concepts that are present in that association. The same process (reification) must be applied whenever there is a binary association with an association class. We will now go through each of these transformations in more detail using illustrative examples [2, 3].

Starting with a simple class and its attributes, we have an example of a class from a UML diagram in Figure 2.2. We will not consider the encoding of operations because that will not be needed for the understanding of the ontology developed in this work. As

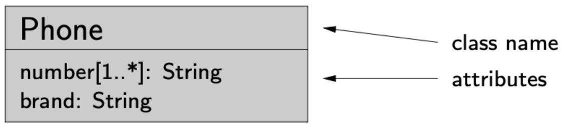


Figure 2.2: Class from a UML class diagram [3].

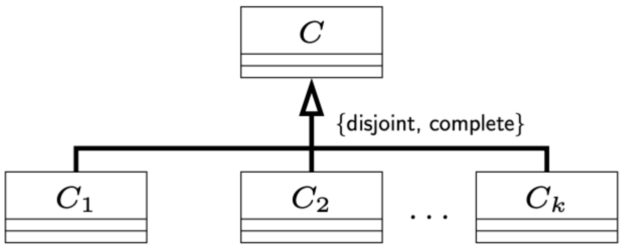


Figure 2.3: ISA relationship from a UML class diagram (adapted [3]).

explained before, since Phone is a class, the concept Phone is introduced, and each of its attributes is a role. For the encoding of its attributes, we need to define their domain and range. The domain of both number and brand is the concept Phone, and their range is the concept String, and it can be represented as:

$$\exists number_P \sqsubseteq Phone \qquad \exists brand_P \sqsubseteq Phone$$
$$\exists number_P^- \sqsubseteq String \qquad \exists brand_P^- \sqsubseteq String$$

This means that, whenever an individual has the role $number_P$, that individual is of type *Phone*, and that role connects a Phone to a String. Same goes for the attribute $brand_P$. We also need to define their multiplicity, by saying that every Phone needs to have these attributes, and it can have several numbers, but only one brand:

$$Phone \sqsubseteq \exists number_P \qquad Phone \sqsubseteq \exists brand_P \sqcap (\leq 1 brand_P)$$

For the encoding of ISA relationships and generalizations such as complete and disjoint, we can look at the UML example in Figure 2.3. Whenever a class is a subclass of another, we use the concept inclusion assertion:

$$C_1 \sqsubseteq C \qquad C_2 \sqsubseteq C \qquad \dots \qquad C_k \sqsubseteq C$$

When there is a *disjoint* generalization, we specify for each pair of concepts that one of the concepts is included in the negation of the other:

$$C_i \sqsubseteq \neg C_j \qquad (1 \leq i < j \leq k)$$

When the generalization is *complete*, we say that every individual of the superclass (C in the example), is a subclass of a disjunction of all the possible concepts, meaning it cannot be anything besides what is specified in the axiom:

$$C \sqsubseteq C_1 \sqcup C_2 \sqcup \dots \sqcup C_k$$

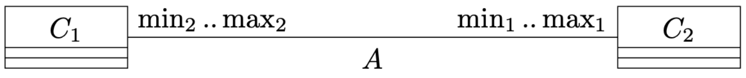


Figure 2.4: Binary association from a UML class diagram [3].

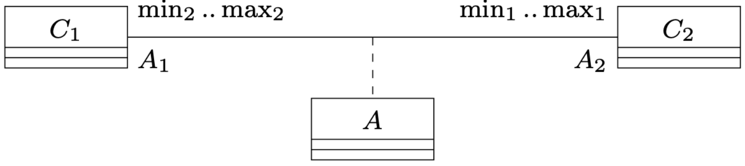


Figure 2.5: Binary association with association class from a UML class diagram [3].

For the encoding of binary associations without association class, consider the UML example in Figure 2.4. Role A connects C_1 to C_2, i.e., role A has domain C_1 and range C_2:

$$\exists A \sqsubseteq C_1 \qquad\qquad \exists A^- \sqsubseteq C_2$$

Each individual from C_1 is connected to at least min_1 and at most max_1 C_2 individuals, and each individual from C_2 is connected through role A^- to at least min_2 and at most max_2 C_1 individuals:

$$C_1 \sqsubseteq (\geq min_1 A) \sqcap (\leq max_1 A)$$
$$C_2 \sqsubseteq (\geq min_2 A^-) \sqcap (\leq max_2 A^-)$$

Figure 2.5 shows an example of a binary association with an association class. Similar to the previous example, here C_1 is connected to C_2, but there is an association class A. This association class is represented as concept A. Next, instead of encoding only one role, we need to encode two roles, connected from concept A to each of the other concepts (A_1 connects to C_1 and A_2 to C_2):

$$\exists A_1 \sqsubseteq A \qquad\qquad A_2 \sqsubseteq A$$
$$\exists A_1^- \sqsubseteq C_1 \qquad\qquad A_2^- \sqsubseteq C_2$$

About the multiplicity of the roles, since A is unique to each relation, then each A can only be connected to exactly one individual of each concept:

$$A \sqsubseteq \exists A_1 \sqcap (\leq 1 A_1) \sqcap \exists A_2 \sqcap (\leq 1 A_2)$$

And since each UML instance from C_1 is connected to at least min_1 and at most max_1 C_2 instances, then in DL that is the multiplicity of the role that connects C_1 to A (role A_1^-). Same goes for C_2 and role A_2^-, with the corresponding changes:

$$C_1 \sqsubseteq (\geq min_1 A_1^-) \sqcap (\leq max_1 A_1^-)$$
$$C_2 \sqsubseteq (\geq min_2 A_2^-) \sqcap (\leq max_2 A_2^-)$$

There are other examples on how to translate UML class diagrams to DL [2, 3], however we will not go through them since it concerns more complex examples that are not necessary to understand this work.

2.1.3 Reasoning

A reasoner offers services that allow reasoning over an ontology in order to infer new knowledge and check for consistency. One of the main services offered by a reasoner is *subsumption testing*, which consists in checking if a concept is a subset of another concept, e.g. $A \sqsubseteq B$ [14, 16, 23]. In other words, it checks if all possible instances belonging to A always belong to B, which would mean that B (the subsumer) is considered more general than A (the subsumee) [23]. Another service is *consistency checking* which checks whether the ontology is satisfiable (consistent), and is also applicable to concepts and roles. An ontology O is consistent/satisfiable if there exists at least one model of O, which would prove that it has no contradictions [16]. Consistency checking is similar for concepts, checking if it is possible for the concepts to have any instances [14], and it is a special case of subsumption in which the subsumer is the empty concept [23].

These are only some of the main services offered by a reasoner, but the fundamental reasoning service from which all the others can be derived is *logical implication* [4], or *entailment* [16]:

An ontology O logically implies an assertion α, i.e. $O \models \alpha$,
if α is satisfied by all models of O.

In other words:

An assertion α is entailed by an ontology O, i.e. $O \models \alpha$,
if it happens for all models I of O, i.e., if $I \models \alpha$.

From the previously mentioned fundamental service, we can enumerate in more detail other services present in *TBox* reasoning [4] (some of which we already talked about):

- **TBox Satisfiability**: *TBox* T is satisfiable if it admits at least one model.
- **Concept Satisfiability**: C is satisfiable *wrt.* T, i.e., $T \not\models C \equiv \bot$, if there is a model I such that C^I is not empty.
- **Subsumption**: C_1 is subsumed by C_2 *wrt.* T if the same happens for every model I of T, i.e., $T \models C_1 \sqsubseteq C_2$ if $C_1^I \subseteq C_2^I$.
- **Equivalence**: C_1 and C_2 are equivalent *wrt.* T if the same happens for every model I of T, i.e., $T \models C_1 \equiv C_2$ if $C_1^I = C_2^I$.
- **Disjointness**: C_1 and C_2 are disjoint *wrt.* T if the same happens for every model I of T, i.e., $T \models C_1 \sqcap C_2 \equiv \bot$ if $C_1^I \cap C_2^I = \emptyset$.
- **Functionality implication**: A functionality assertion is entailed by T, i.e., $T \models$ (*funct* R) if, for every model I of T, we have that $(o, o_1) \in R^I$ and $(o, o_2) \in R^I$ implies $o_1 = o_2$.

Definitions such as satisfiability, subsumption, equivalence and disjointness also hold for roles.

Ultimately, reasoning over an ontology is not only about checking its satisfiability and inferences through the services mentioned above, but also about *instance checking*, for both concept and role instances (*ABox*). If an individual c, or a pair of individuals (c_1, c_2), is an instance of a concept C, or an instance of a role R (respectively), in every model of O, then it is consistent [4].

There is a tradeoff between the expressiveness of a representation language and its reasoning complexity [23]. Regarding time complexity, reasoning over DL ontologies is ExpTime-Hard, even for simpler DLs. However, it still has a lot of expressiveness, with all the constructors mentioned in this section, that allow DL to stay within the ExpTime upper bound [4].

2.1.3.1 Closed World Assumption vs. Open World Assumption

Closed World Assumption (CWA) is the assumption that if something is not known to be true then it is false. When there is a knowledge base (set of facts) it is assumed that only the facts that are stated there are true and if something is not there nor can be inferred from it, then it is false. An example of a common system with CWA is a train timetable, where we can see the times at which every train arrives at a certain train station. If someone wants to know if there is a train at, for example, 6 p.m. and that is not on the table, then it is known that there is no train at that time (assuming that everything is working perfectly fine and there are no delays). Another example of CWA is the list of students that belong to a certain course. The knowledge base is known to be complete, hence only the students in that list are part of the course, and students that are not in the list are known to not belong to that course.

On the other hand, Open World Assumption (OWA) does not discard anything from being true or false if it is not stated, the system is considered to have incomplete information. So, if something is not in the knowledge base, it can either be true or false and we cannot assert that it is true neither that it is false, we can only say that it is unknown with the currently known information. For instance, if someone says that they like tea, we cannot assume that that is the only thing they like, for instance, we do not know if they like coffee or not, it is completely unknown to us and we cannot make any assumptions about it.

Description Logics work with the OWA, because we want to represent knowledge, e.g., ontologies, and want to discover new information. Its knowledge base is not assumed to be complete nor to be possible to infer everything that is true from its incomplete information. In the next section we can see how reasoning in DL works and the implications of OWA.

2.1.3.2 Open World Reasoning

Reasoning in Description Logics is based on the Open World Assumption, and therefore referred to as Open World Reasoning (OWR) [14]. Hence, if there is some missing information, we cannot assume it to be true nor false, it is unknown. Thus, the reasoner can only conclude that something is true in a certain ontology O if it happens in all models of O.

Borrowing an example from the pizza ontology in [14], there is a VegetarianPizza concept, and its complement concept, NonVegetarianPizza. The concept VegetarianPizza refers to Pizzas that can only have toppings that are CheeseTopping or VegetableTopping. The concept NonVegetarianPizza refers to the Pizzas that are not VegetarianPizzas. If we define a RandomPizza as a Pizza that has some CheeseTopping, by using an existential restriction, since we are working under OWA, this pizza might have additional toppings of any kind, including MeatToppings. RandomPizza can contain individuals that are indeed VegetarianPizzas (e.g., an individual with just CheeseToppings), and others that are not (e.g., an individual that, besides CheeseTopping, also has a MeatTopping). Saying that RandomPizza is a subset of VegetarianPizza would mean that every individual of the former is also an individual of the latter, which we just showed that it does not necessarily happen. This means that the reasoner does not know if RandomPizza is a VegetarianPizza or not, hence it will not classify it as a subset of VegetarianPizza. For the same reason, it will not classify it as NonVegetarianPizza. This explains why reasoners use entailment, they can only assert something is true when it is true for all the models.

2.1.4 OWL2 and the Protégé Framework

The Protégé framework is an open-source ontology editor that fully supports the OWL2 language (Web Ontology Language) [37]. We used this framework to reason over our ontology to check for its consistency. This tool allows us to define an ontology and has features beyond the basics we saw in the previous sections. In this section, we will not explain how to use this tool. Instead, we will go through the previous information with more detail, explaining some of the possibilities that this framework brings.

The nomenclature used in OWL is slightly different from DL. Concepts are known as classes in OWL, roles are properties, and individuals (instances of classes) are also known as individuals.

2.1.4.1 Datatype Properties and Object Properties

Properties in OWL can refer to datatype properties or object properties.

Datatype properties connect individuals from a class to data literals. They correspond to the attributes of a class in a UML diagram. As an example, individuals from the class Pizza can have a *price* data property, that associates individuals from that class to float numbers, e.g.:

$$pepperoni\_pizza : Pizza \qquad (pepperoni\_pizza, 12.0) : price$$

Object properties connect individuals from one class to individuals from another class. They correspond to the relations between two classes in a UML diagram. As an example, individuals from the class Pizza can be connected to individuals from the class PizzaTopping through the *hasTopping* object property, e.g.:

$$pepperoni\_pizza : Pizza$$
$$pepperoni : PizzaTopping \qquad (pepperoni\_pizza, pepperoni) : hasTopping$$

All properties have a domain and a range, i.e., they connect individuals from the domain to individuals from the range. Object properties have classes as both domain and range, and data properties have classes as domain, but data literals as range. From the examples above, the *price* property has the Pizza class as domain, and float numbers as range, and the *hasTopping* property has the Pizza class as domain and the PizzaTopping class as range. If multiple classes are defined as range of a property, the range becomes the intersection of those classes. Once again, since OWL works with OWA, domains and ranges do not work as restrictions or constraints, but as axioms in reasoning [14]. E.g., if the *hasTopping* property has Pizza as domain, and we apply it to the IceCream class, it just infers that IceCream is a subclass of Pizza, not giving any error [14].

Object properties can be enriched using object property characteristics [14, 38], some of which are:

- **Functional:** asserts that the individual that has this property can be connected to at most one other individual through it. E.g., an individual can only have one age - (functional *hasAge*). If an object property is defined as being functional but is connecting an individual A to two other individuals, B and C, then B and C are inferred to be the same object, because of OWA.

- **Transitive:** if a property connects an individual A to another, B, and it also connects B to another individual C, then it also connects A to C. A good example of this is relationships between siblings, if A is sibling of B, and B is sibling of C, then A is also sibling of C - (transitive *isSiblingOf*).

- **Symmetric:** asserts that if a property connects A to B, then it also connects B to A. The previous sibling example can be applied here too. If A is sibling of B, obviously B is also sibling of A - (symmetric *isSiblingOf*).

- **Reflexive:** asserts that, if a property is reflexive, then every individual is related to itself via that property. E.g., anyone can know different people, but they always know themselves. A can know B and C, but A also knows A, B knows B, and C knows C - (reflexive *knows*).

OWL also allows us to add numerical, existential, and universal restrictions to properties, just as in the example previously given, where we discussed defining a RandomPizza class as a subset of Pizza class that has some CheeseTopping, by using an existential restriction. As mentioned, due to the Open World Assumption, a RandomPizza can have more toppings besides CheeseTopping, which results in the RandomPizza class not being classified as VegetarianPizza nor as NonVegetarianPizza. But, if we want to ensure that pizzas from this class can only have CheeseToppings, we need to add a universal restriction, which is known as *closure axiom* [14]. Using just the existential restriction, we could only infer that RandomPizza has cheese. With both the existential and universal restrictions, RandomPizzas have at least one CheeseTopping, and cannot have more toppings apart from the CheeseTopping type, hence they can be classified as VegetarianPizza. If one were to only add the closure axiom, it would mean that RandomPizzas are pizzas with only CheeseToppings, or with no toppings at all [14].

2.1.4.2 Classes

The hierarchy of classes can also be named as a taxonomy, and it should be constructed as a tree in which each node (class) only has one parent (superclass), keeping the ontology in a maintainable and modular state. Computing and maintaining multiple inheritance should be left as a task for the reasoner only [14]. In Protégé, OWL classes are assumed to overlap, because it works under OWA, hence it is necessary to specify which classes are disjoint from each other [14].

Classes are described by their conditions, which include subclass-superclass relations and restrictions over object and data properties, like RandomPizza being a subclass of Pizza and having some CheeseTopping. These conditions say what is *necessary* for the individual to have if it belongs to that class. If an individual belongs to the RandomPizza class, it is necessarily a Pizza, and it is necessary to have at least one CheeseTopping. A class that only has necessary conditions is known as a primitive class [14].

If these conditions are *necessary* and *sufficient*, it means that not only those conditions are necessary to belong to that class (every individual from that class needs to satisfy them), but also that they are sufficient to infer membership (having those characteristics is enough for an individual to be a member of that class) [14]. A class that is described with both necessary and sufficient conditions is known as a defined class [14]. This is used by reasoners to infer a hierarchy, but they can only automatically classify under defined classes [14].

3
Related Work

Since our work combines knowledge from different areas, we will cover different topics, from conceptual design of applications to ontology-based development to phase distinction and staged computation, and summarise the literature related to these topics.

Section 3.1 describes how to correctly choose concepts of a software system, and gives an example of the development of an application that applied concepts in a distinct way from ours. Section 3.2 mentions some works where ontologies were used as part of application or language development. In Section 3.3 we compare between model-driven engineering and low-code development platforms. We introduce metaprogramming in Section 3.4. Finally, in Section 3.5 we present some multi-stage programming languages and phase distinction techniques.

3.1 Application Development Using Concepts

Jackson [15] addresses the origin of concepts embodied in software systems and shares some guidelines on how to choose them. In this section, we summarise his research.

3.1.1 Origin of Concepts

According to Jackson, a concept is the building block of a system and to correctly use it one needs to understand its meaning. Choosing the correct concepts facilitates not only the programmer's work but also the use of the application by the user. Most concepts were defined somewhere in time, and in software systems we often use preexisting concepts, embracing their current meaning and sometimes even adding new connotations to them. For example, when deleting files on a computer, we often send them to the computer trash, which works similarly to a trashcan. Then we can recover some files we put in there, or we can empty it, as we do with our trash, and permanently get rid of unimportant files.

However, software features are often abstract, allowing users to perform actions in an abstract world, which can or cannot have visible consequences. This happens because software systems are not tangible, their features are not evident, i.e., it is difficult to understand the effects of a certain action, like shaking a phone, unless it is explained. Also, many of the actions performed in software systems have no immediate tangible effect. That can happen as a consequence of delay or failure of services. But mostly it is due to the fact that many actions only produce internal state changes that are not immediately visible to the user, e.g., saving a webpage as favorite has no visible change until we later open the favorites list so that we can select and revisit that webpage. Hence, creating this parallel with day-to-day objects and concepts by adopting well-known concepts allows users to better understand what the effects of their use will be. This makes it easier and more natural to understand its underlying functionality (its purpose and the immediate consequences of its actions).

3.1.2 Choosing Concepts

It is important to evaluate the choice of concepts. In addition to formal and informal analysis, one can also use some general criteria suggested by Jackson, and adopted by others [9]. There are four criteria: motivation, no redundancy, no overloading, and uniformity.

Motivation. Each concept should have a purpose, a reason for its existence. Motivation of a concept should be expressible in a short sentence. It should complement something needed, something that it would be at fault without it.

No redundancy. Two different concepts must have two different purposes. Hence, a concept should only be added if what it represents is not possible to obtain from other concepts, either from a single one or a combination of them. This criterion prevents an unnecessary increase in complexity.

No overloading. A concept should not have more than one purpose, maintaining modularity. This prevents conflicts and is not difficult to accomplish since software systems are abstract and easier to decouple as we deem appropriate.

Uniformity. Variants of the same abstract concept should have similar functionalities, avoiding misuse of concepts because of unexpected behaviour.

3.1.3 Other Advantages

Concepts can be used as more than just means to an end. Their use is also advantageous to guide a new software design [15].

First, by helping to keep the focus on what is important at each step of development, preventing an unnecessary increase in complexity. Secondly, by naturally simplifying a large problem into a smaller one. This can be explained by the fact that using carefully chosen concepts drives development by purpose, leading to sound increments. If instead, one were to add functionalities as needed, it could cause fragmentation and poor modularity between components. It could even originate an overly specific design that would only work for a concrete application (the one used as model) and would not generalise to other applications.

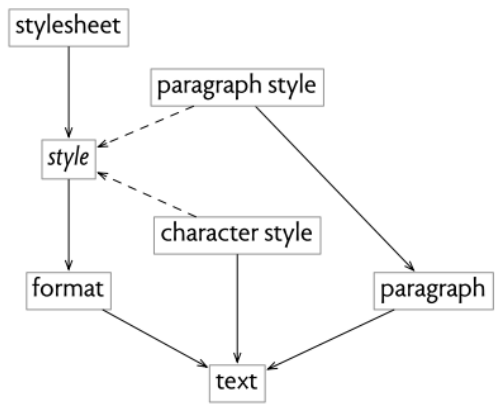


Figure 3.1: Dependence graph example - Partial concept graph for a word processor (in [15]).

Finally, when the goal is to produce an initial minimum version of a product, with just enough features, the use of concept dependence graphs can be useful. This graph allows us to see if the concepts of a domain need to be simplified. A dependence graph is a directed graph which has concepts as vertices, and dependencies between them are represented by edges. An edge going from a concept (C1) to another (C2) means that the existence of the first concept makes no sense without the other, i.e., 'C1 depends on C2'. Sometimes two concepts can be variants of the same abstract concept. In this case, the abstract concept is represented in a vertex too, but with its name in italics. The variant concepts are vertices that have dashed arrows connecting them to the abstract one. If another concept depends on the abstract concept it means that at least one of the variant concepts must be present. In Figure 3.1 we can see an example of a dependence graph. One of the dependencies we can see is that the paragraph style concept would make no sense without a paragraph, neither the paragraph without a text. Both paragraph style and character style are variants of the abstract style concept.

3.1.4 Application Example

An approach to web application development using a catalog of concepts is described in [9]. The catalog they used has concepts like *Authentication*, *Property*, *Event*, *Follow* and *Rating*. However, our goal is not to gather a set of concepts but instead see how the user intent can be categorised in different ways by using concepts.

Given the fact that many applications have a lot of implementation similarities, in order to avoid repeating code, Rosso et al. [9] decided to join similar functionalities inside a concept. Each concept is a self-contained reusable increment of functionality, and allows slightly different implementations (for example, ratings of different entities such as a post or a user), merely corresponding to instantiations of the same generic concept. Different concepts can also be combined in many different ways to generate new functionalities.

Their approach is implemented in the Déjà Vu platform [9] that features a catalog of full-stack functionality modules (concepts), which can be glued together by declarative bindings. This binding allows concepts to synchronise their actions and exchange data between them. To assess the viability of this approach, they built a variety of non-trivial applications using Déjà Vu platform and checked for deviations from their expected behaviour. From that assessment, they concluded that it was possible to replicate most of the desired functionalities by just combining generic concepts from that catalogue. The functionalities that deviated more from the norm could be implemented by creating a new concept module.

This evaluation phase showed them that some concepts lacked some functionalities and that they were even missing some concepts that would be helpful in different situations. Despite using a different approach, we can benefit from making a similar evaluation phase to our ontology to check if we can satisfactorily describe applications with it.

In our work, the choice of concepts is crucial to the creation of an ontology that incorporates them and allows their translation into application modifier operations. Next, we present works whose development was supported by ontologies.

3.2 Ontology-Based Development

There are several works where ontologies were used to help during the development of languages [1, 11, 31, 33] or the development of applications [19, 24].

Some works tried to incorporate domain analysis into the DSL design phase. Bačíková et al. [1] introduce the DEAL (Domain Extraction ALgorithm) method for extracting domain information from GUIs (Graphical User Interfaces) and derive DSL grammars from it. Although they do not use ontologies, it is similar because it extracts information about a specific domain. Walter et al. [31] report an approach where they use ontologies to define DSLs, with the OntoDSL framework. This framework allows checking the consistency of the domain model. There are two other works where the authors developed tools that transform an ontology into DSL grammars, the Ontology2DSL framework [33], and the Onto2Gra [11]. Both tools use an ontology describing a knowledge domain and automatically create a DSL for the same domain.

There are other works that try to use ontologies during application development. The ODESeW framework [19] allows the development of ontology-based Web applications. In [24], O'Connor et al. developed mapping tools to integrate information contained in a variety of formats, among which are ontologies, and then used these tools to develop a prototype Web-based application.

3.3 Model-driven Engineering and Low-code

Both model-driven engineering (MDE) and low-code development platforms (LCDP) share the goal of reducing the efforts required for developing and maintaining applications, as well as enabling people with limited programming skills to participate in software development [13, 27]. Besides their common goal, both approaches share some similarities. One, for instance, is the reliance on domain-specific languages (DSLs). MDE uses specially tailored DSLs to define models clearly, avoiding unnecessary information that increases entropy. LCDPs, on the other hand, make use of DSLs to make some common programming concepts more intuitive, simplifying the development task. Despite their common points, these approaches have some key differences. For instance, MDE does not necessarily entails code generation and not always leads to less code. LCDPs often make use of specialized techniques such as natural language programming, programming by example, programming by demonstration, and visual programming languages, which may not make much sense when coupled with MDE.

3.4 Metaprogramming

Metaprogramming languages, as defined in [18], are programming languages that treat programs as data, allowing for their manipulation and generation at runtime, thus, offering some advantages like increased performance, code reusability, and easier code analysis and inspection.

Work on metaprogramming exists on many fronts. There is, for instance, aspect-oriented programming, multi-stage programming, and generative programming [18]. Multi-stage programming is quite useful in the context of low code development platforms and especially in our work. This technique allows for the division of program evaluation in multiple, distinct, non-interfering phases. For instance, let us consider a C++ compiler. The compiler's execution involves two stages. The first is where it performs template instantiation, i.e., for each type used in a template, the compiler writes a matching class, and posteriorly it performs the rest of the compilation using the generated classes. Multi-stage programming, and metaprogramming in general, comes in quite handy in the context of low code development platforms, where it is often necessary for code manipulation and generation in a multistaged fashion to compile and execute the desired program.

In the context of our work, we employ two-staged programming; the first stage, the compile stage, ensures the correctness of the template definition and performs compile-time operations such as expression simplifications and the execution of compile-time expressions whose value is required at runtime. The second stage, the evaluation stage, executes the program generated in the previous stage.

3.5 Staged Computation

Our work includes the design of a type-safe template language that enables metaprogramming. Metaprogramming is a technique in which programs are able to manipulate other programs or themselves. The language we develop manipulates code fragments and must ensure the well-formedness of the generated program. Staging computation is a popular optimization technique used in high-level program generation [8, 22]. Multistage programming is highly effective as it avoids runtime interpretive overheads [30]. Hence, it is pertinent to mention some approaches that support the semantic separation of computation stages.

We mention some multi-stage programming languages and phase distinction techniques that ensure the well-formedness of the generated runtime computations (code).

3.5.1 Phase Distinctions

A type system can have values, types of those values, and sometimes kinds (a kind is a type of a type, or of a type operator). We can describe the execution of a program as a two-phase process, containing the typechecking phase (which occurs at compile-time) and the execution phase (occurring at runtime). We can distinguish between phases by representing runtime values as just values and compile-time values as types. However, kind-free type systems that allow using $Type\colon Type$ may hinder phase distinction and be classified as phase-free systems. Also, whenever we come across dependent types[1], phase distinction is lost.

A small example of a kind and phase-free system is a program that has $A\colon Type$ and $B\colon A$. If $A = Int$ and $B = 3$, then B is a runtime value. However, if $A = Type$ (using $Type\colon Type$) and $B = Int$, then B is a compile-time computation [5].

Cardelli [5] presents the modification process from a phase-free type system based on dependent types into a phased type system. He tries to enforce the following phase distinction requirement: "If A is a compile-time term and B is a subterm of A, then B must also be a compile-time term.". However, dependent types do not abide by this requirement. Cardelli introduces kinds to the system to meet this requirement and rejects $Type\colon Type$, resulting in a three-level hierarchy of values, types and kinds. Hence, the main judgement ($\Gamma \vdash a\colon A$) turns into three:

$$\begin{aligned}&\Gamma \vdash K\ kind\\ &\Gamma \vdash A :: K\\ &\Gamma \vdash a\colon A\end{aligned}$$

This means that all kind-free operators are replaced by several *kinded* operators. So, for instance, the following conversion rule:

[1]A dependent type is a type whose definition depends on a value (not just the type), e.g., the type of a list where the length is part of the type.

$$\frac{\Gamma \vdash a\colon A \qquad \Gamma \vdash B\colon \mathit{Type} \qquad A =_{\beta\eta\mu} B}{\Gamma \vdash a\colon B}\,[\![\textsc{Conversion}]\!]$$

states that if a has the type A, B is a type, and A is equal to B, then one can say that a has type B. The judgement $\Gamma \vdash B\colon \mathit{Type}$ can either mean that B can be a type, e.g. Int, or a kind. Hence, the conversion rule now turns into two rules, one at the value level (with B as a type, $B :: K$) and another at the type level (with B as a kind, $K\ \mathit{kind}$):

$$\frac{\Gamma \vdash a\colon A \qquad \Gamma \vdash B :: K \qquad A =_{\beta\eta\mu} B}{\Gamma \vdash a\colon B}\,[\![\textsc{Conversion-T}]\!]$$

$$\frac{\Gamma \vdash A :: L \qquad \Gamma \vdash K\ \mathit{kind} \qquad L =_{\beta\eta\mu} K}{\Gamma \vdash A :: K}\,[\![\textsc{Conversion-K}]\!]$$

Similarly, rules for functions will turn into four rules, one that goes from types to types, another from types to kinds, from kinds to types, and from kinds to kinds. This way, the author associates values with runtime computations, and everything else (types and kinds) with compile-time computations, i.e., judgements of the form $\Gamma \vdash a\colon A$ represent runtime terms, while judgements of the form $\Gamma \vdash A :: K$ or $\Gamma \vdash K\ \mathit{kind}$ represent compile-time terms. Next, kinded operators where phase mixing occurs (e.g., when a type - compile-time term - depends on a value - runtime term) are excluded since they do not respect the phase distinction requirement.

In summary, in [5], Cardelli explains how to obtain a system with phase distinction by using kinds to layer the language and rejecting operators that cause phase mixing.

3.5.2 Template Haskell

Template Haskell is an extension to the Haskell language and supports compile-time metaprogramming. A template language (high-level language) is easier to maintain and reason about because the compiler abstracts low-level details [28]. Additionally, more knowledgeable users can manipulate their programs, interlacing them with the compiler's manipulations.

With Template Haskell, functions are written in the same language, regardless of their execution stage (compile-time or runtime). Therefore, explicit annotations that specify when each code should execute are necessary [36, 28]:

1. **Quotations**: quotations delimit code and allow its inspection by taking an expression at compile-time and building an abstract syntax tree that represents it. Quotations for expressions are represented as `[e|...|]`, or simply `[|...|]`. Quotations for declarations, types and patterns have the same notation, replacing `e` with `d`, `t` or `p`, respectively;

2. **Quasi-quotations**: allows one to extend the language with more quotations;
3. **Splices**: `$expr` is a splice. The body (`expr`) of the splice is evaluated at compile-time.

Template Haskell has cross-stage persistence, allowing the usage of compile-time variables inside generated code.

3.5.3 MetaML

MetaML is a multi-stage functional programming language that allows programmers to express staging naturally and concisely using explicit program annotations [22]. The program annotations are the four programming constructs of the language [22, 30]:

1. **Brackets** `<_>`: surrounding an expression `e` with brackets, `<e>`, defers the computation of `e`, converting any syntactic expression into a piece of code;
2. **Escape** `~_`: escape can only be applied to bracketed expressions, e.g., `~<e>`, and must be enclosed by brackets. It inserts `e` into the surrounding bracketed expression. For instance, `<a - ~<e>>` becomes `<a - e>`;
3. **Run** `run _`: run is applied to deferred expressions, forcing their computation. For instance, `run <4 + 1>` evaluates as `5`.
4. **Lift** `lift _`: lift evaluates the expression to which it is applied (such expression must not contain variables nor functions) and converts it into code. For example, `lift 2+3` evaluates to `lift 5` and finally to `<5>`.

To understand the use of these constructs, we analyse a small example:

```
<3 + ~(lift 5-4)>
```

First, we evaluate the expression inside `lift`, `5-4`, and get

```
<3 + ~(lift 1)>
```

Next, the `lift` constructor converts the result of that computation into code resulting in

```
<3 + ~<1>>
```

`Escape` removes the brackets from `<1>`, splicing it into the surrounding code expression:

```
<3 + 1>
```

This is the final expression, since it is bracketed and cannot be further evaluated. To execute this code, we can use

```
run <3+1>
```

which computes the expression 3+1, resulting in 4.

Notice that both the brackets and the lift constructors build code from an expression, but only the latter evaluates the expression beforehand.

An example of the usage of MetaML to generate code is the following [22, 30]:

```
-| fun mult x n = if n=0
                  then <1>
                  else < ~x * ~(mult x (n-1)) >;
val mult = fn : <int> -> int -> <int>

-| val cube = <fn y => ~(mult <y> 3)>;
val cube = <fn a => a * (a * (a * 1))> : <int -> int>

-| fun exponent n = <fn y => ~(mult <y> n)>;
val exponent = fn : int -> <int -> int>
```

Function mult receives an integer as code x and returns code corresponding to the multiplication of x, n times. This function is then used to generate code of the cube function, or of a broader exponentiation function given an exponent n.

These annotations can be used to create more than two stages (or levels). The level of an expression is determined by the number of surrounding Brackets, minus the number of surrounding escapes. However, caution is necessary when using variables at different levels. For the correct usage of this language, one must consider two principles: cross-stage persistence and cross-stage safety [30].

Cross-Stage Persistence. Cross-stage persistence allows the user to stage expressions with free variables (non-closed expressions). However, bracketed expressions with free variables must resolve their free occurrences in the static environment where the expression occurs, which results in the variable being bound to a constant that was resolved in a previous stage. This principle can also be described as a variable i bound in stage n is available in all future stages, n+1. Take as example the expression:

```
let val a=1+4 in <72 + a> end
```

The free variable a is bound, in the first stage, to the constant 5. Even if this piece of code is used inside another context with another value bound to a, it will still return 5, as it is bound in the value's local environment.

Cross-Stage Safety. Conversely, if an input is first available at stage m, it cannot be used at stage n, if m > n, i.e., m occurs at a later stage. An incorrectly staged example is:

```
fn a => <fn b => ~(a+b)>
```

Because of the escape constructor, the computation of a+b should occur in the first stage. However, b is only available in the second stage.

3.5.4 Logic-Based Type Systems

Type systems that support staged computation can also be motivated logically. For instance, Davies and Pfenning, in [8], present a type system that allows specifying and analysing computation stages based on intuitionistic modal logic S4.

In classical logic, each proposition P is either `true` or `false`. Modal logic extends classical logic by allowing other truth values, such as "P is known" or "P is necessarily true". Additionally, the truth value of a proposition can diverge under different circumstances, e.g., the proposition "it is autumn" can have different values at different times or places. Such variations are modalities and indicate the mode in which the statement is said to be true [21].

There are several varieties of modal logic, but we will not dive into them. Instead, we will briefly explain two modal operators introduced by modal logic: box and diamond ($\Box$ and $\Diamond$, respectively). The $\Box$ operator represents necessity, and $\Diamond$ represents possibility. Hence, $\Box P$ means that P *is necessarily true*, i.e., P is true in all possible worlds[2] [21]. $\Diamond P$ means that P is possible, i.e., P is true somewhere.

The definition of necessity is related to the **definition of validity** [25], which states that whenever A is true, no matter what hypothesis we consider (A is true in all worlds), then A is valid. Therefore, if A is valid, then it must be true under some hypothesis (A is true in some world). This can be written as:

1. If $\cdot \vdash A\ \mathit{true}$ then $A\ \mathit{valid}$.
2. If $A\ \mathit{valid}$ then $\Gamma \vdash A\ \mathit{true}$.

Here, "$\cdot$" indicates an empty set of hypotheses.

To allow hypothesis of the form $A\ \mathit{valid}$, one can separate hypothesis about truth and validity and consider the following judgement [8, 25]:

$$B_1\ \mathit{valid}, \ldots, B_m\ \mathit{valid}; A_1\ \mathit{true}, \ldots, A_n\ \mathit{true} \vdash A\ \mathit{true}$$

One can represent the set of validity assumptions as Δ, and the set of truth propositions as Γ, resulting in a judgement of the form $\Delta;\Gamma \vdash A$.

Another way to write that A is valid, is the proposition $\Box A$. The introduction rule for the $\Box$ operator goes from the validity of A to the truth of $\Box A$, according to the definition of validity [25]:

$$\frac{\Delta; \cdot \vdash A\ \mathit{true}}{\Delta;\Gamma \vdash \Box A\ \mathit{true}}\ [\![\Box I]\!]$$

The elimination rule for the box operator cannot be the inverse of the introduction. The rule

$$\frac{\Delta;\Gamma \vdash \Box A\ \mathit{true}}{\Delta; \cdot \vdash A\ \mathit{true}}\ [\![\Box E]\!]$$

[2]In this context, world means anything that can be said to believe a proposition, e.g., people, ideologies.)

is unsound because we drop the assumptions in Γ. This means that the elimination rules are too strong and allow us to derive more information than we should [8, 25].

To solve this, one could try to use the rule

$$\frac{\Delta;\Gamma \vdash \Box A\ \mathit{true}}{\Delta;\Gamma \vdash A\ \mathit{true}}\ [\![\Box E]\!]$$

This rule is sound but not locally complete. This means that the elimination rule is too weak and does not allow us to conclude everything we should be able to because we do not have sufficient information to reconstitute the operator by an introduction rule [8, 25].

Instead, we can use the **substitution principle for validity**:

$$\text{If } \Delta;\cdot \vdash B\ \mathit{true} \text{ and } \Delta, B\ \mathit{valid};\Gamma \vdash J \text{ then } \Delta;\Gamma \vdash J$$

as an inspiration for a locally sound and locally complete elimination rule [25]:

$$\frac{\Delta;\Gamma \vdash \Box A\ \mathit{true} \qquad \Delta, A\ \mathit{valid};\Gamma \vdash C\ \mathit{true}}{\Delta;\Gamma \vdash C\ \mathit{true}}\ [\![\Box E]\!]$$

It can be read as: if $\Box A$ is true under some hypothesis, then any judgment we make under the additional hypothesis that A is valid, must in fact be evident [8]. Notice that validity is only used in assumptions, and never in conclusions.

Davies and Pfenning [8] show that extending the Curry-Howard isomorphism between proofs and programs to the intuitionistic modal logic S4 results in a logical explanation of computation stages. Each world in modal logic corresponds to a different stage, and terms like $\Box A$ correspond to code to be executed in a future stage.

Based on the operators and principles we presented here, modal λ-calculus, $\lambda_e^{\to\Box}$, appears. The syntax [8] is similar to that of the λ-calculus, extended with modal contexts Δ, modal variables u, and the modal constructor (**box**) and destructor (**let box**):

Types	$A ::=$	$a \mid A_1 \to A_2 \mid \Box A$
Terms	$E ::=$	$x \mid \lambda x{:}\ A.E \mid E_1 E_2 \mid$
		$u \mid \mathbf{box}\ E \mid \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ E_2$
Ordinary Contexts	$\Gamma ::=$	$\cdot \mid \Gamma, x{:}\ A$
Modal Contexts	$\Delta ::=$	$\cdot \mid \Delta, u{:}\ A$

The modal constructor **box** is the same as the modal operator with its name, $\Box$. Hence, the introduction and elimination rules are similar to those of the modal logic, and are defined as follows [8]:

$$\frac{u\colon A \text{ in } \Delta}{\Delta;\Gamma \vdash u\colon A}\ [\![\textsc{modal-var}]\!]$$

$$\frac{\Delta;\cdot \vdash E\colon A}{\Delta;\Gamma \vdash \mathbf{box}\, E\colon \Box A}\ [\![\Box I]\!]$$

$$\frac{\Delta;\Gamma \vdash E_1\colon \Box A \qquad (\Delta, u\colon A);\Gamma \vdash E_2\colon B}{\Delta;\Gamma \vdash \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ E_2\colon B}\ [\![\Box E]\!]$$

This forms the basis for extending other languages, enabling to express and check the staging of computation [8].

3.5.5 Remarks

The first approach [5] mentioned in this section explains how to distinguish between compile-time values and runtime values by defining a system containing values, types and kinds. This approach is useful for systems with dependent types. Since we do not have dependent types, we are able to keep it simple and avoid introducing kinds to our solution.

The next three approaches (Template Haskell [36, 28], MetaML [22, 30], and Logic-Based Systems [8, 25]) have some similarities between them. All of them use explicit annotations to delimit runtime computations (code). Template Haskell uses quotations, MetaML uses brackets, and modal languages use the box constructor. They also include other annotations to compute code, and others. Additionally, all of them have cross-stage persistence, i.e., stage-`m` computations may contain free variables defined in a previous stage `n`; and cross-stage safety, which prevents the opposite from happening.

These approaches, specifically modal languages, inspired our solution. Akin to them, our solution also features cross-stage persistence and cross-stage safety.

Furthermore, there are other multi-staged approaches. For example, MetaDepth [17] is a framework that enables the development of multi-level (multi-staged) programs, encompassing a code generation procedure at each stage of the process. Following this approach, entities (such as classes) are associated with a potency, a natural number that signals how many stages are left for the entity to be irreducible. In our approach, since we only have two stages, it is as if our entities implicitly have a potency of one, unless they are boxed, in which case they have the potency of zero. The major difference between MetaDepth and our approach lies in model conformance verification. In MetaDepth, this verification occurs after parameter substitution, while we rely on a static type system to perform correctness verifications at compile time.

4 The GOLEM Project

In this chapter, we present the GOLEM project, led by OutSystems. We start by briefly explaining what is the OutSystems platform in Section 4.1. In Section 4.2, we present the goal of the GOLEM project, and its research threads in Section 4.3. In Section 4.4, we present the OSTRICH language, which is part of one of the research threads we are working on.

4.1 The OutSystems Platform

The OutSystems platform is a low-code development platform that allows non-expert developers to build applications. This platform contains an intuitive visual interface that eases the development of applications and automates deployment. One can assemble simple applications by dragging widgets into screens and define the application behaviour by composing intuitive visual diagrams. Figure 4.1 reveals the platform interface: a preview of the application being built appears in the centre, the visual elements (widgets) that one can add are on the left, and the application elements are on the right. Widgets include containers, tables, forms, and buttons, among others. Application elements are represented as trees of the objects that constitute the application. Element trees describe the application user interface, the database composition, logic behaviour, or processes.

The OutSystems platform also contains pre-built screens that are common patterns in application development. Figure 4.2 shows part of a set of pre-built screens, namely lists, available in the OutSystems platform.

4.2 Goal of the GOLEM Project

Despite the visual interface and the drag-and-drop mechanism, having a solid grasp of software architecture and programming is still necessary for creating applications. OutSystems wishes to ease the development of such digital systems by automating their creation. This led OutSystems to partner with leading research institutions with expertise in automated programming, namely program synthesis, programming languages and

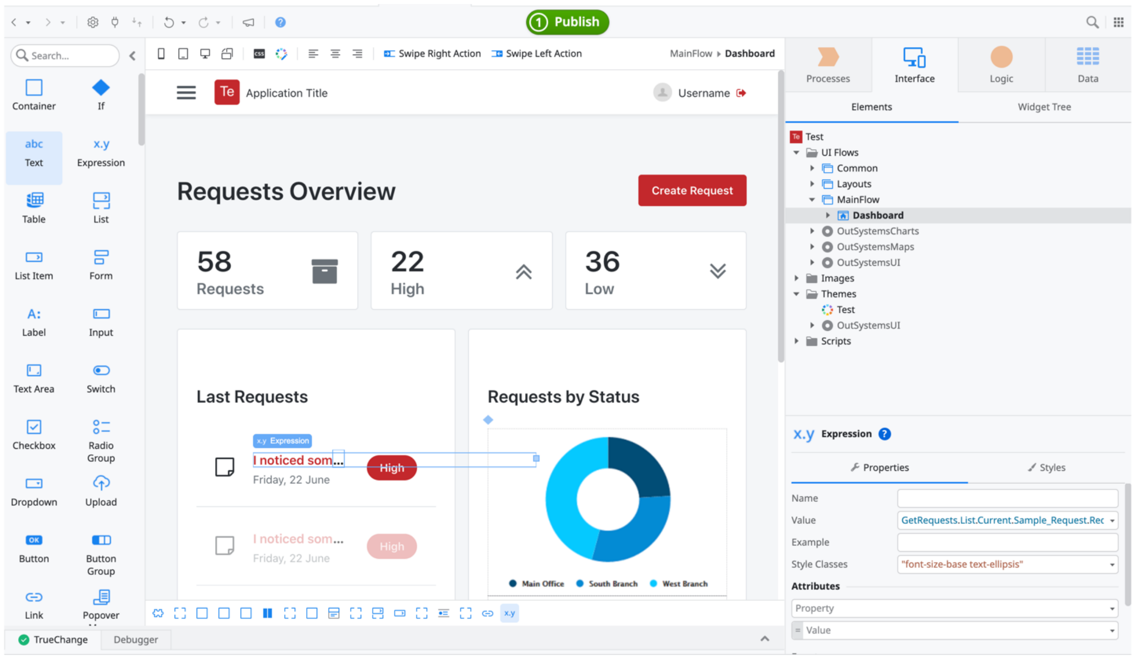


Figure 4.1: OutSystems platform interface, when creating an application from scratch.

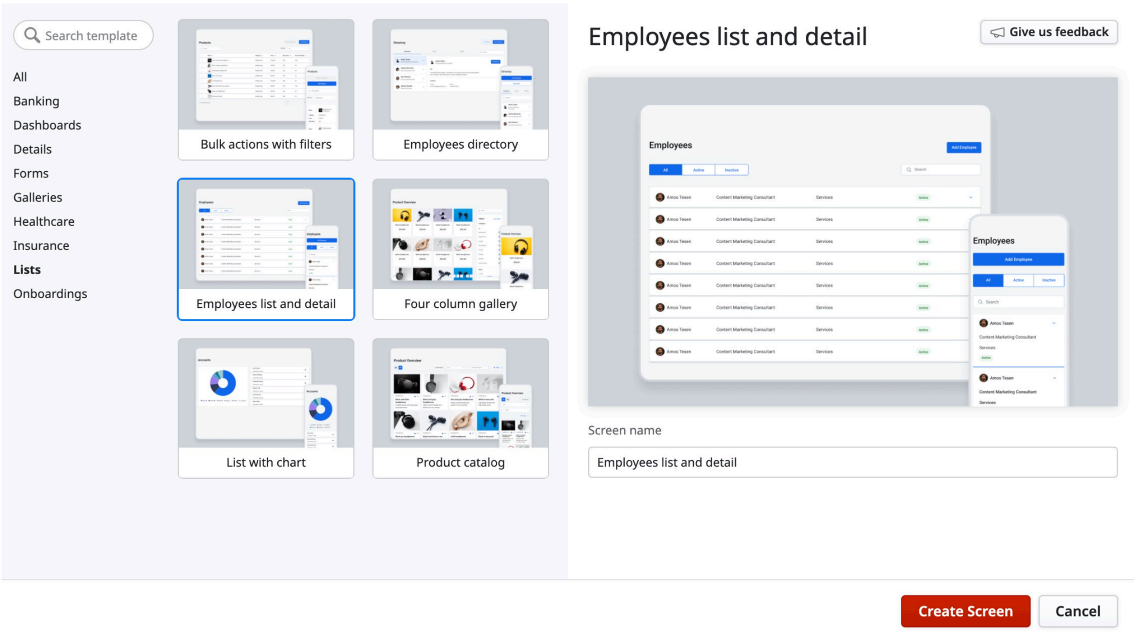


Figure 4.2: OutSystems pre-built options for screens with lists.

models, and human-computer interaction through natural language processing: INESC-ID, NOVA-LINCS, and Carnegie Mellon University (CMU).

From this partnership emerges the GOLEM Project, a large-scale research project whose central goal is the development of the next generation of low-code. It aims to revolutionize the application development experience and automate programming, making OutSystems' low-code technology easier to use. This project aims to determine the most natural way for a user with no development experience to create applications without writing any code and how to implement such a system.

The easiest way for someone to express themselves is through natural language, but its translation to accurate determinist systems bears some challenges. In GOLEM, we explore how to derive meaning from the user dialogue through the selection of some concepts to incrementally build applications that work according to the user's intent. OutSystems made an initial study to determine the needs and expectations of their platform's target users. They identified that most of the problems are related to page design changes, which allowed to reduce the project's scope.

4.3 Abstraction Layers and Research Threads

There are currently four research threads related to language-based approaches, scattered in three levels of abstraction, as depicted in Figure 4.3, where research threads are delimited within vertical lanes. The information flows from high-level components, starting with the user dialogue in natural language, to lower-level components, culminating in the intended OutSystems application.

There is a natural language processing component that analyses the user dialogue [45, 46]. This component recognizes verbs and nouns in the user dialogue, mapping them to the concepts and relationships of a predefined ontology[1]. The chosen ontology must represent applications and their components, i.e., its concepts need to capture the main ingredients that allow application development with OutSystems. We further describe our work and contributions to this thread in Chapter 5.

Next, an intermediate layer contains a domain-specific language (DSL) of system changing operations [46]. By comparing the latest user requests with the current state of the application, this layer generates a sequence of operations that represent the delta, i.e., the changes that need to occur in the application development process. Such a sequence of operations may include the creation of new elements and also the update and substitution of existing ones. The application of that sequence must ensure the preservation of the underlying system's soundness.

Finally, there is a layer in a lower level of abstraction that directly manipulates the OutSystems' applications according to the information received from previous layers. This layer contains an application programming interface for the OutSystems model (model API) with operations that manipulate the OutSystems applications. Two other research threads complement this API: a thread on data manipulation and another on template definition and instantiation. The data manipulation component receives information from other high-level components, creating and modifying the application's data layer according to that information. The template thread consists of a rich type-safe template language for OutSystems, the OSTRICH language [20]. In the OutSystems development

[1]An ontology is a formal representation of a set of concepts within a domain and the relationships between those concepts [14].

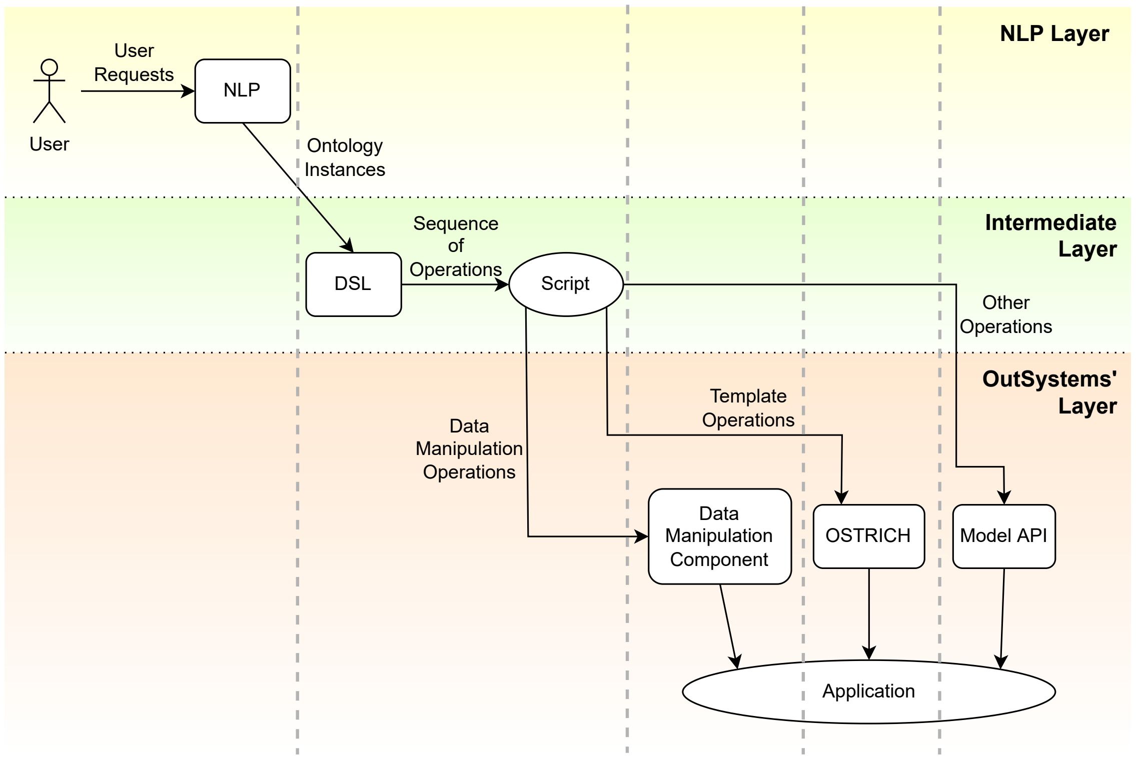


Figure 4.3: Architecture of the GOLEM project.

platform, there are pre-built screens, as shown in Figure 4.2. Such screens require error-prone, time-consuming, manual adjustments to apply the screen to the correct data of the current application. This language supports the definition and instantiation of templates in the OutSystems' applications, automating such adjustments, easing and hastening the development process while ensuring the resulting screen is valid. We formalize and propose extensions to the OSTRICH language [41, 43, 44], and present our contribution to this thread in Chapter 6. We introduce OSTRICH in Section 4.4.

4.4 The OSTRICH Language

Less-experienced developers can use low-code development platforms to develop applications through an intuitive visual interface. An example of such platforms is the OutSystems platform, which automatically manages several details about deployment, streamlining the development process. This platform contains pre-built screens, such as lists and dashboards, that aid and speed up the development of an application. So, for instance, if one were to build an application that lists elements on a screen, one could select a pre-built screen containing a list. However, such pre-built screens pose a problem: they contain temporary dummy data, meaning that developers must manually adjust them to their data to ensure that their application works as expected. Often, such adjustments require the developer to have a good understanding of programming basics. Furthermore,

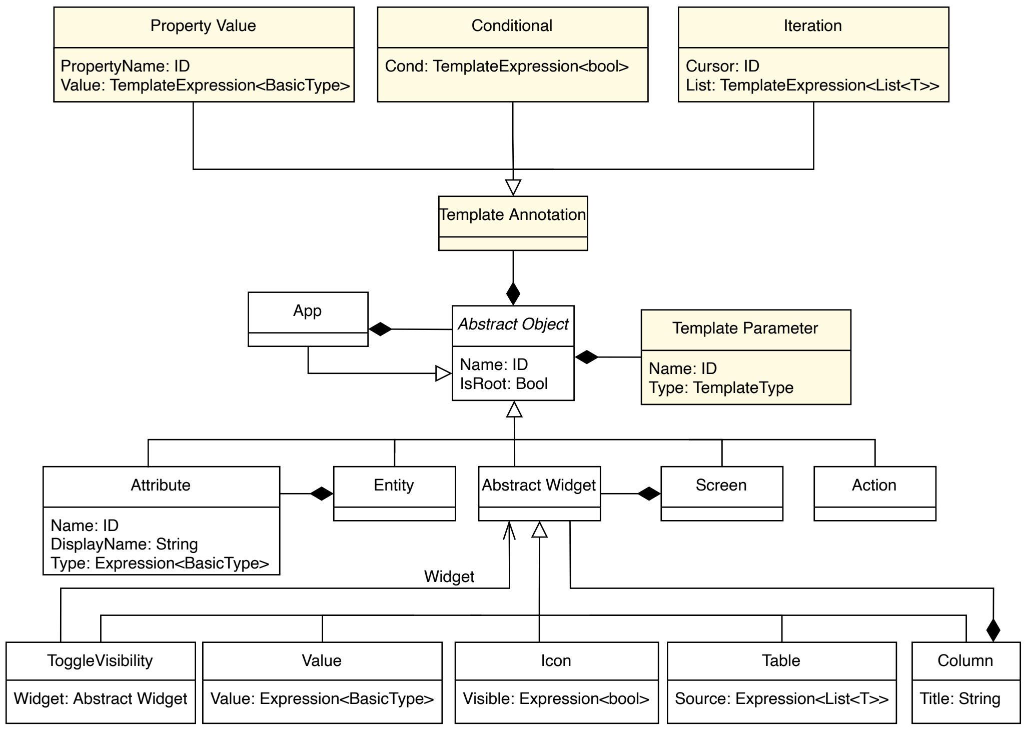


Figure 4.4: OSTRICH's template metamodel (adapted from [20, 41]).

these adjustments may prompt errors that polute the gracefulness of the low-code environment. This contributes to an onerous learning curve, hindering the use and adoption of the platform.

That is the motivation behind the development of the OSTRICH language. OSTRICH is a rich type-safe template language for OutSystems [20]. This language supports the definition and instantiation of templates with input parameters. Templates, in this setting, are analogous to the previously mentioned pre-built screens, but their input parameters are the data to which the template is applied. This language eases the developer's work by automatically adapting the template to the received arguments, thus avoiding time-consuming error-prone manual adjustments. The automated adjustments are defined by annotation nodes present in the language. These nodes contain expressions built during instantiation according to the template input parameters.

The OSTRICH's underlying metamodel [20, 41] incorporates additional elements into the metamodel of OutSystems applications, maintaining the backward compatibility with previous versions. These new elements support the definition of templates and their components. Figure 4.4 depicts a simplified version of this metamodel, where yellow nodes represent elements introduced by OSTRICH.

This metamodel shows that OutSystems applications comprise components to define data, user interface (UI), and behaviour. User interface components are what the application's end-user sees and with which they interact. Data definition components include entities and their attributes. Entities correspond to database tables in a traditional data

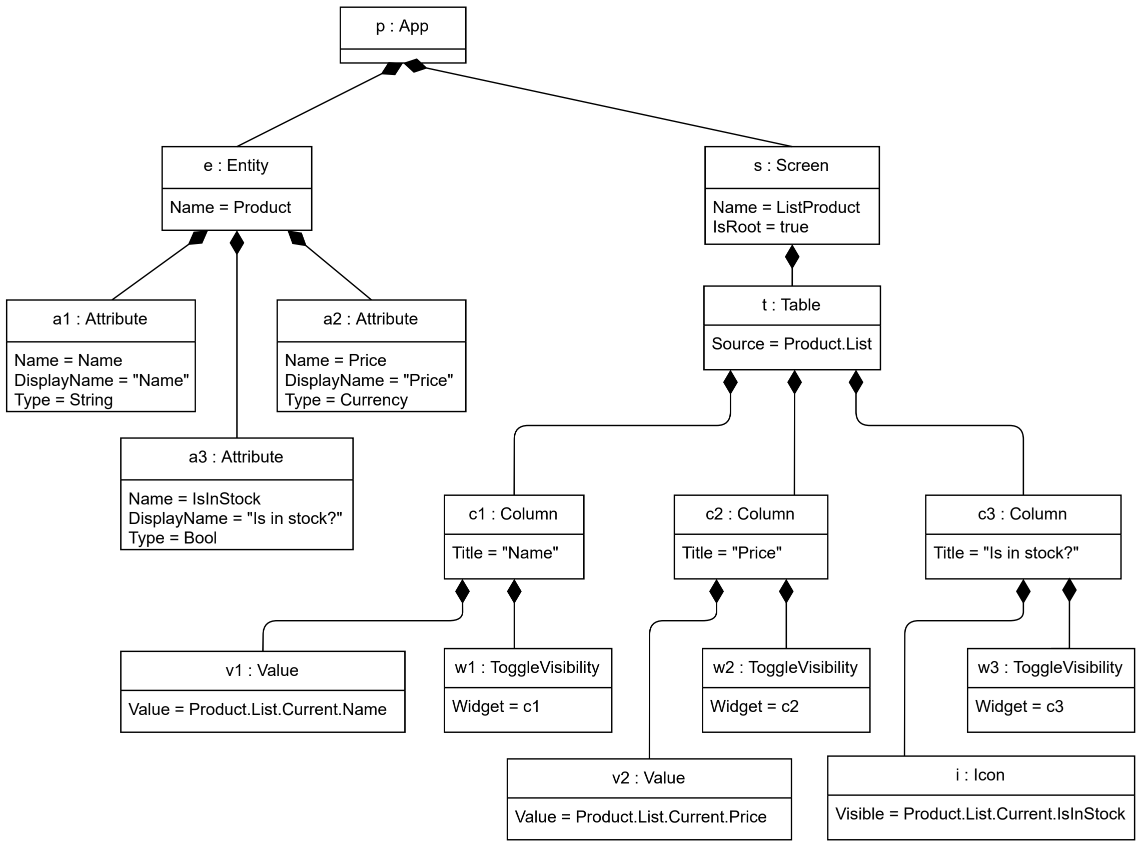


Figure 4.5: Application model of a list of products.

layer. And attributes are their fields or columns. Therefore, one entity may have several attributes, as a database table has several columns. The top-level UI components comprise screens and widgets, as depicted in the diagram. Widgets can be UI tables that display data, with columns that contain other widgets, such as simple values or icons. Columns have a title in the table header, and value and icon widgets have expressions that evaluate the information that must emerge in those widgets. Tables have a source expression that matches a record list with the data to be displayed.

Annotation elements introduce the declaration of template parameters, declaration of property values, and iteration and conditional instructions. Model annotations allow the verification of the validity of template expressions when applied to such parameters. Hence, OSTRICH guarantees that all template instantiations result in a valid program with valid runtime expressions [20, 41].

To better understand the use of these model annotations, take as an example the application model depicted in Figure 4.5. This model describes a screen with a table that lists products and their attribute's information. The table contains as many columns as the number of attributes of the Product entity. Additionally, a table cell is a `Value` widget, unless the attribute value is a boolean, in which case the widget in use is the `Icon`.

This application can be abstracted as a generic model that produces similar screens for different entities. Its corresponding template can be defined as in Figure 4.6, adapted

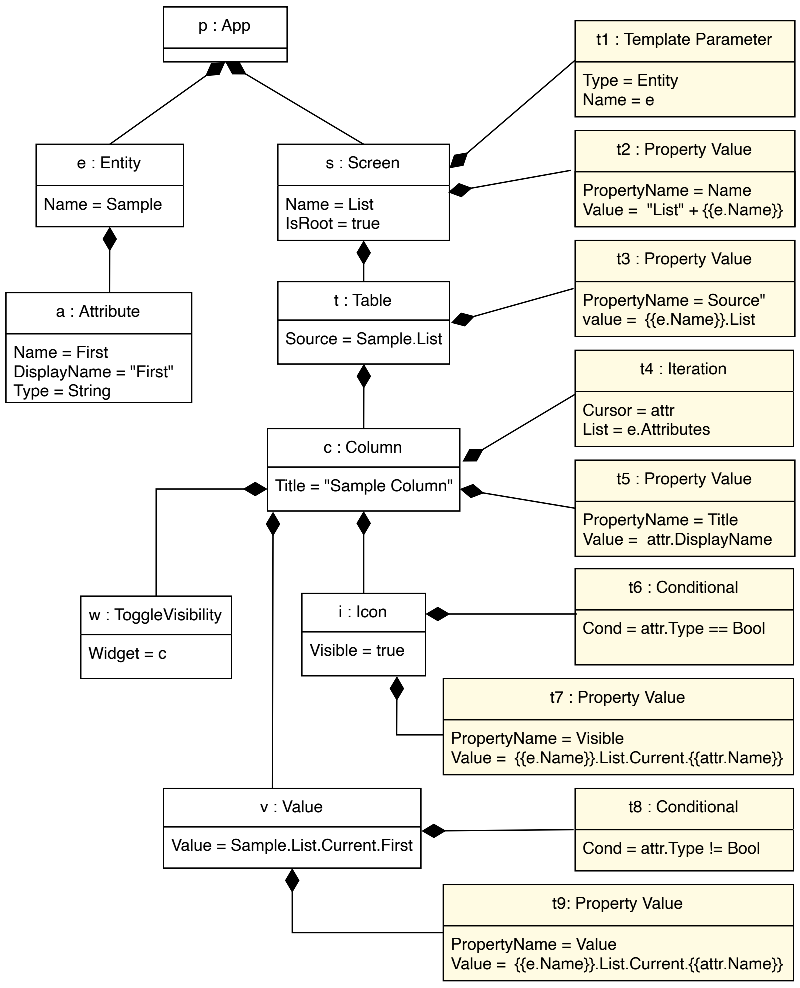


Figure 4.6: List template model (adapted from [20, 41]).

from [20]. First, the `Template Parameter` annotation specifies the template input parameters. In this example, the parameter is the entity that we wish to display, the entity `Product`. Since each column displays values of an attribute and is named according to it, columns are defined with an `Iteration` annotation. This annotation iterates a compile-time list and repeats a model element and its children for each list item. The cursor name allows future annotations to reference the list item being iterated. Next, the choice between a `Value` or an `Icon` widget is accomplished through the `Conditional` annotation. During compile time, this annotation includes, or disregards, an element when its condition value is *true*, or *false*, respectively. Finally, the `Property Value` annotation provides expressions that set the element property's value, which are customized according to the template parameters.

The OSTRICH language still has some limitations that currently prevent the implementation of some of the OutSystems templates. In [20], it is not yet possible to account for template customization, i.e., a user cannot reapply an instantiated template that suffered customization changes to another argument. It is also not possible to specify richer constraints such as dependencies between parameters. Besides formalizing the OSTRICH language, we provide a reference implementation that addresses the latter limitation [44]. Our approach is described in Chapter 6 and Chapter 7.

5 An Ontology For Programming

5.1 Overview

The pipeline developed in the GOLEM project must collect the user intent and carry it through its pipeline elements, culminating in an OutSystems application. Part of the project requires the existence of a domain-specific language that receives information about the user intent in the form of ontology concepts. These concepts work as a specification to generate and combine OutSystems-like components that satisfy the user intent. These components are then used to build the application described by the user.

This domain-specific language captures the intent of web application developers through concepts and relationships. Hence, our first step is to decide which concepts are necessary and sufficient to build a broad and complete model that captures most applications' design and behaviour. We started by writing a descriptive text about a specific web application as an example and then deduced some general concepts. We chose concepts that apply not only to the described application but to other applications in general.

Because we want to describe concepts and relations within a specific domain (application development) and then reason over them, we use description logics to represent our model. Description logics are a family of formal knowledge representation languages designed to represent and reason on structured knowledge [4]. Because OWL2 is compatible with the description logic SROIQ, it proved to be appropriate in the construction of the ontology and the verification of its consistency. We use the Protégé platform, an open-source ontology editor and knowledge-base framework that supports OWL2. This framework is also suitable for building queries and testing their results on the developed ontology. Such queries allow us to filter between the existing individuals (instances) based on their characteristics and relationships.

In this chapter, we present the developed ontologies and iterate over them while explaining the reasons for their creation.

5.2 Ontologies

We present a total of three ontologies that served as inspiration for the final ontology, presented in João et al. [45, 46]. We use the notation of DLs, presented in Chapter 2, to describe the ontologies.

5.2.1 First Ontology

Our ontology needs to comply with the OutSystems concept structure, which contains higher-level concepts common to every application regardless of its domain. Therefore, we designed part of a base ontology containing typical OutSystems concepts. This base ontology is a portion of the minimal ontology that is required such that other ontologies, with user-defined concepts, can be mapped into it. This ontology has generic concepts that can be combined in different ways according to the desired functionalities of the application. Each user may want to further specify their application by extending the base ontology with new concepts and roles. For example, the base ontology can have concepts like *Post* and *PostItem* and users may want to specify new concepts that they understand as being *PostItem*s, such as *MapLocation* or *Picture*.

The overall ontology is theoretically split into four layers, going from more generic to more specific concepts. In Figure 5.1 we can see an example of part of the ontology we are pursuing.

OUTSYSTEMS LAYER	DEFAULT PATTERNS LAYER	GOLEM LIBRARY LAYER
$Entity \sqsubseteq Thing$	$User \sqsubseteq Entity$	$Editor \sqsubseteq Thing$
$Attributes \sqsubseteq Thing$	$LoginAction \sqsubseteq Action$	$List \sqsubseteq Thing$
$Aggregate \sqsubseteq Thing$	$Menu \sqsubseteq Entity$	$PostTemplate \sqsubseteq Template$
$Application \sqsubseteq Thing$	$MenuItem \sqsubseteq Entity$	$Post \sqsubseteq Entity$
$Template \sqsubseteq Thing$		$PostItem \sqsubseteq Entity$
$Action \sqsubseteq Thing$		$PostEditor \sqsubseteq Editor$
$SaveAction \sqsubseteq Action$		$PostList \sqsubseteq List$
$DeleteAction \sqsubseteq Action$		

GOLEM USER-DEFINED LAYER

$Announcement \sqsubseteq Post$	$Picture \sqsubseteq PostItem$
$AdoptionOffer \sqsubseteq Announcement$	$Species \sqsubseteq PostItem$
$LostAnnounc \sqsubseteq Announcement$	$Breed \sqsubseteq PostItem$
$FoundAnnounc \sqsubseteq Announcement$	$MapLocation \sqsubseteq PostItem$

Figure 5.1: Part of the ontology and its four layers.

OUTSYSTEMS LAYER

$$
\begin{aligned}
\exists hasAttr &\sqsubseteq Entity\\
\exists hasAttr^{-} &\sqsubseteq Attribute\\
Entity &\sqsubseteq \exists hasAttr\\
Attribute &\sqsubseteq \exists hasAttr^{-} \sqcap (\leq 1 hasAttr^{-})\\
\exists label &\sqsubseteq Attribute\\
\exists default\_value &\sqsubseteq Attribute\\
\exists values &\sqsubseteq Attribute
\end{aligned}
$$

GOLEM LIBRARY LAYER

$$
\begin{aligned}
PostAttribute &\sqsubseteq Attribute\\
PostStatus &\sqsubseteq PostAttribute\\
PostDescription &\sqsubseteq PostAttribute\\
\exists creates &\sqsubseteq User\\
\exists creates^{-} &\sqsubseteq Post\\
Post &\sqsubseteq \exists creates^{-} \sqcap (\leq 1 creates^{-})\\
Post &\sqsubseteq \exists hasAttr.PostStatus \sqcap (\leq 1 hasAttr.PostStatus)\\
Post &\sqsubseteq \exists hasAttr.PostDescription \sqcap (\leq 1 hasAttr.PostDescription)\\
PostStatus &\sqsubseteq \exists label.\{\text{"}postStatus\text{"}\} \sqcap (\leq 1 label.\{\text{"}postStatus\text{"}\})\\
PostStatus &\sqsubseteq \exists values.\{\text{"}censored\text{"},\text{"}open\text{"},\text{"}solved\text{"}\}\\
&\quad \sqcap (\leq 1 values.\{\text{"}censored\text{"},\text{"}open\text{"},\text{"}solved\text{"}\})\\
PostStatus &\sqsubseteq \exists default\_value.\{\text{"}open\text{"}\} \sqcap (\leq 1 default\_value.\{\text{"}open\text{"}\})\\
PostDescription &\sqsubseteq \exists label.\{\text{"}postDescription\text{"}\} \sqcap (\leq 1 label.\{\text{"}postDescription\text{"}\})\\
PostDescription &\sqsubseteq \exists values.\{string[pattern\text{"}.\{1,100\}\text{"}]\}\\
&\quad \sqcap (\leq 1 values.\{string[pattern\text{"}.\{1,100\}\text{"}]\})
\end{aligned}
$$

Figure 5.2: Part of the small example ontology with restricted domain.

The first layer is the aforementioned **OutSystems built-in concepts layer**. It is the higher-level layer and it contains high-level concepts common to most applications, like *Entity*, *Attribute* of an entity, *Template*, or *Action*. Next, we have the **default patterns layer** which contains other entities that are important and common to most applications, such as the concepts of *login*, *user* and *page menu*, and respective *menu item*. Next comes the

Golem library layer which contains concepts that are still common to most applications but not mandatory. The concepts of this layer intend to help build any application and can be combined in different ways to achieve the desired functionality. Some examples of concepts from this layer are *posts*, *comments*, and the *ranking* present in the commonly known 'likes' system and others. Finally, we have the **Golem user-defined layer**. This last layer contains concepts corresponding to instances of concepts from higher layers, like an *announcement* being a *post*, and a *picture* being a *post item*. The user defines concepts that reflect the intended purpose of its target application.

This base ontology enables building in more detail a small example ontology with a restricted domain. Figure 5.2 depicts part of this ontology, which is an extension of the ontology shown in Figure 5.1. In this excerpt, we defined that *Users* can *create Posts*, and each *Post* needs to be *created* by one and only one *User*. We added the attributes *Post-Description* and *PostStatus* as subclasses of *PostAttributes* and defined their *labels*, *default values*, and the domains of their *values*. The full example ontology is available in Annex I.

The ontology concepts (classes) represent a model of the application the user intends to build. And the instances (individuals) of these classes represent the actual data once the application is up and running. For instance, an application containing *Posts* as its building block will have a *Post* class, and the instances of that class are concrete posts written by the end-users of that application. To test the consistency of this ontology, we create individuals of these concepts and assign values to their data properties. Additionally, such individuals allow testing queries to obtain a subset of these classes according to their characteristics.

Unfortunately, this first ontology has some limitations. Since both the pre-defined and the user-defined concepts are ontology classes, there is no way to distinguish between them, i.e., it is hard to specify which of the possible embeddings the user wishes to apply. Additionally, we are still missing a way to determine which OutSystems components we need to build.

5.2.2 Second Ontology

Our second trial emerged from a smaller portion of our first ontology. From the OutSystems layer, we kept both *Entity* and *Attribute*. As entities, we still have the *User* and the *Post*, and add the *Comment*. We define a set of attributes for some of these entities, and this is where this ontology starts to diverge from the previous one: there are additional attributes named *Extendable* (see Figure 5.3) that allow the user to customize their own attributes with different types of data.

Another difference between these first two ontologies is the meaning of classes and individuals. In the first ontology, both the pre-defined and user-defined layers are classes, and the individuals are runtime values, i.e., concrete values for each concept corresponding to the data displayed to the end-user. In this second ontology, classes define the features that can embed the application. OutSystems developers express their intent

$$\begin{aligned}
Extendable &\sqsubseteq PostAttribute\\
ExtendableString &\sqsubseteq Extendable\\
ExtendableBlob &\sqsubseteq Extendable\\
ExtendableEnumerate &\sqsubseteq Extendable\\
ExtendableRef &\sqsubseteq Extendable\\
ExtendableInt &\sqsubseteq Extendable\\
ExtendableCurrency &\sqsubseteq Extendable
\end{aligned}$$

Figure 5.3: Architecture of the GOLEM project.

through natural language, which, in turn, is translated into a set of ontology individuals. Such individuals belong to the classes that correspond to the desired features. This way, both classes and individuals are part of the application design phase. The runtime values are not part of the ontology since they occur in later stages during execution. Since we only need the ontology for design purposes, using it solely for that purpose allows us to make the most of it.

Finally, this ontology also contains some classes representing OutSystems components that need to be built, namely database entities, a template of a searchable list, and a template of a screen with details of an entity and an edit form. We define such classes using subclass axioms (subsumptions) with class expressions as subclasses. These axioms are known as general class axioms and state that every instance that meets the requirements is an instance of that class. For example, if an individual of a class is related to a user with a read and a search property, i.e., if the user can read and search that individual, the ontology infers that such an individual is an element of a searchable list template. This subsumption relation is represented as follows:

$$\exists read^{-}.User \sqcap \exists search^{-}.User \sqsubseteq ElemSearchableList$$

This means that if the ontology states, for instance, that a user can read and search through posts (due to the presence of the *read* and *search* relations between a user instance and a post instance), we know that a post is an element of a searchable list. Hence, we will need to create a searchable list of posts in the application, resorting to a template.

Another axiom represents the need to create a screen with details of an entity and an edit form. We define that an entity might be an element of such a form if a user can create an object of that entity:

$$\exists create^{-}.User \sqsubseteq ElemDetail$$

Finally, we desire to infer which individuals are database tables (database entities) in the target application. Note that a database entity is not the same as an entity in the ontology. We define three axioms that state that a) any ontology entity with attributes, or

b) any ontology entity that can be created by a user, or c) any attribute with various value options (enumerate), is a database entity:

$$
\begin{aligned}
Entity \sqcap \exists hasAttr.Attribute &\sqsubseteq ElemEntityDB \\
Entity \sqcap \exists create^{-}.User &\sqsubseteq ElemEntityDB \\
Attribute \sqcap \exists values.\{string[pattern''\backslash\backslash[.^{*}(,.)^{*}\backslash\backslash]'']\} &\sqsubseteq ElemEntityDB
\end{aligned}
$$

Using this ontology to describe different applications reveals an adequate level of abstraction and allows to test the inferences about OutSystems components. We define three different applications using the same ontology but different individuals. We remind the reader that the ontology classes represent the possible features of an application, and the set of individuals represents the application and the features to implement.

The entire base ontology is available in Annex I.

Application #1 - Newspaper. We populate the ontology in a way that describes an application for a newspaper. There are three types of users: basic, premium, and administrator. Only an administrator can create and publish news. By creating three individuals of the class *User*, we were able to declare the three desired types of users. Since our ontology is still a prototype, some user permissions, are not yet implemented. We also define *news* as an individual of the *Post* class, and define its attributes *text* and *media*. *Text* is an individual of the already existing *Description* class, and *media* is an *ExtendableBlob*. A relation *create* connects the administrator user to the news. This relation helps infer that we need a detail-screen OutSystems component for the *administrator* to create and edit the *news*. Additionally, the reasoner infers that *news* are database entities because they contain attributes. We are missing some inferences that should occur on the pre-defined OutSystems components. However, we were not exhaustive in defining the axioms in the prototype stage.

Application #2 - Inventory. In this example, we have two types of users: a simple user and a manager, both declared similarly to the previous example. Here, the *Post* is a product entry in the inventory, with several attributes like name, quantity, goal quantity, and cost. The relations *read* and *search* connect the simple user to the product entry, stating that a simple user can read and search through a catalogue of products. These relations allow inferring we need a searchable list whose element is a product entry. There is also a *create* relation between the manager and the product entry, i.e., managers can introduce new products into the catalogue. Thus, the reasoner infers that we need a detail-screen to create and edit product entries.

Application #3 - Pet Adoptions. We now describe a blog where users can create posts for animal adoptions. Thus we need several post statuses that describe the status of the announcement. The status *open* means it is still looking for a family, *closed* means it is solved and *censored* means it is inappropriate for the blog. An administrator user can also create ordinary informational posts. Similarly to the previous examples, we declare more than one type of user: regular user and administrator. Additionally, we declare two different types of posts, announcement posts and informational posts, by creating two individuals of the class *Post*. Regular users can read and search both types of posts but can only create announcements, and administrators can only create informational posts. We declare this through relations connecting the different users with the different posts. Such relations allow inferring that both posts are database entities and elements of detail screens and searchable lists. The announcement post has some attributes, including extendable enumerates to define the different possible species and the statuses (*open*, *closed*, *censored*). Since one of the subsumption axioms states that all lists (enumerates) are database entities, then we know post status and post species attributes are, in fact, database entities in the OutSystems application.

Despite being a good representation of general applications, this ontology still bears some limitations. Because the information extracted from natural language must be directly applied to populate the ontology[1], there is a need to close the gap between natural language and the produced ontology. This need was addressed by the natural language processing component of the GOLEM project, resulting in the final ontology, which we present next. Additionally, the axioms that allow for inferring the need for some OutSystems components are not expressive enough. Let us imagine there are two classes with the same axioms, but one of them has an extra axiom that defines additional conditions. If an individual complies with all the axioms, including the extra, then it will infer that the individual is an instance of both classes. For example, if those two classes represent an element of a list of values and an element of a searchable list, we do not wish to create two similar lists. Instead, we want the one that gives us more information, the searchable list. This kind of selection is not possible with description logics. Hence, we decided to delegate this type of reasoning to the domain-specific language that will receive information from the ontology and create and update the OutSystems application.

5.2.3 Final Ontology

The final ontology results from combined efforts between our work with the previous trials here described and the natural language team's work. The choice of concepts in our first two ontologies is in accordance with the type of information needed for the domain-specific language. However, a gap between the user utterances and these ontologies was still present, leading to the development of a final ontology [45]. Such ontology is outside of the scope of our thesis work.

[1]Populating an ontology is the process of creating individuals (instances) for that ontology.

6 Template Language

6.1 Motivation

The ontology allows the natural language processing component to gather information from the user utterances, as presented in Chapter 4 and Chapter 5, and delivers it to a DSL developed under the scope of the GOLEM project. The DSL produces a sequence of operations that reshapes the final application according to the user's intent. Since some requests are popular patterns, we can benefit from the use of pre-built screens and widgets. These pre-built application's fragments can be reused and allow for the assembly of a safe and nicely designed application.

Currently, OutSystems contains pre-built screens that contain temporary dummy data. This means that the developer must adjust them according to the user's change requests to ensure the application works as expected. These changes proved to be time-consuming, error-prone and esthetically challenging.

The instantiation of such pre-built screens with arguments would fasten this process. This is where OSTRICH comes into the picture. OSTRICH is a strongly-typed rich templating language for the OutSystems platform that allows the correct instantiation of templates [20]. In this chapter, we present a formalization of the OSTRICH language, and include some extensions to it. First, we describe some features of the language (Section 6.2). Next, we define its syntax (Section 6.3), type system (Section 6.4) and operational semantics (Section 6.5). The complete syntax, type system and semantics are present in Annex II.

The formalization presented here echoes our prototype implementation.

6.2 Language Features

We implement a two-stage language that derives from the λ-calculus, and extends the OSTRICH language, presented in [20], with: 1) terms that represent nodes and expressions, instead of a metamodel; 2) parametric polymorphism; 3) dependencies between parameters; 4) template declaration; 5) template instantiation inside another template declaration.

Next, we explain how we guarantee each of these features, resorting to the example depicted in Figure 6.1.

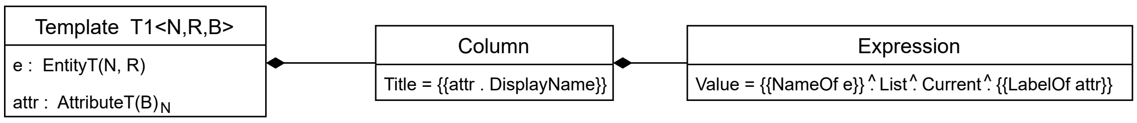

Template T1<N,R,B>
e : EntityT(N, R)
attr : AttributeT(B)N
Column
Title = {{attr . DisplayName}}
Expression
Value = {{NameOf e}} ^ List ^ Current ^ {{LabelOf attr}}

Figure 6.1: Model example of a template definition and its expressions.

6.2.1 Staged Computation

We implement a language that represents a template extension for the OutSystems language. Therefore, it is necessary to represent and separate the corresponding compile-time and runtime stages.

Compile time comprises the typechecking of template definitions, followed by template instantiations. In the OutSystems platform, compile-time computations occur at the design phase, and correspond to the information we can visualize in the platform. The design phase comprises the construction of the application with widgets and expressions and the instantiation of pre-built screens, now templates. In this phase, some information is available, such as the name of entities and their attributes' properties. We may visualize the general database composition and see the presence of, for example, an entity named `Product` and its attribute `IsInStock`. This attribute's property values are available at runtime, such as the way its name should appear in the application, the `DisplayName`: "Is in stock?".

Runtime computations occur during the execution of the OutSystems application. The concrete instances of entities and attributes, the rows of the database tables, are only available during this stage. All expressions whose result depends on such instances must be runtime expressions.

During template instantiation, node property's values can be compile-time expressions, like the title of a column that depends on the name of an attribute (available at compile time), or runtime expressions, like an attribute's value displayed in a table cell (only available at runtime). Figure 6.1 depicts an example of a template definition containing various nodes and both compile-time and runtime expressions. The syntax used in Figure 6.1, adapted from [20], represents compile-time computations surrounded by double curly braces.

The language we implement guarantees that all template instantiations are valid and produce valid runtime expressions by ensuring phase distinction through staged computation [5, 8]. That means it is a multi-stage language with a typechecking algorithm that reports both type and phase errors, thus ensuring that compile-time and runtime expressions are well-formed even before execution. The algorithm detects phase errors using a supplementary environment, Δ, mapping runtime variables to their types. We restrict the typing of runtime expressions so that they only enclose other runtime expressions and variables from Δ. We can delve into Figure 6.1, specifically the runtime expression:

$$\{\{\texttt{NameOf}\ e\}\}\,\hat{.}\,\mathit{List}\,\hat{.}\,\mathit{Current}\,\hat{.}\,\{\{\texttt{LabelOf}\ \mathit{attr}\}\} \tag{6.1}$$

Notice that the variables *e* and *attr* are variables that map to an entity and an attribute, respectively, which are available at compile time. This means that *e* and *attr* are compile-time variables. Both `NameOf` and `LabelOf` are compile-time built-in operations, because of the surrounding double curly braces. However, the overall expression is a runtime expression.

In our implementation, the distinction between compile time and runtime is achieved through a specialized constructor, `Box`. Therefore, expression 6.1 is written as follows:

$$\begin{array}{l}\texttt{letbox}\ u_{name} = \texttt{NameOf}\ e\ \texttt{in}\\ \quad\texttt{letbox}\ u_{label} = \texttt{LabelOf}\ \mathit{attr}\ \texttt{in}\\ \quad\quad\texttt{Box(}\ u_{name}\,\hat{.}\,\mathit{List}\,\hat{.}\,\mathit{Current}\,\hat{.}\,u_{label}\ \texttt{)}\end{array} \tag{6.2}$$

The expression inside the construct `Box` is a runtime expression that can only contain runtime variables, as mentioned. Hence, we need the `letbox` sentence to insert runtime expressions inside the box. This means that both `NameOf` and `LabelOf` are compile-time built-in operations that receive compile-time arguments (*e* and *attr*) and evaluate as runtime expressions, thus securing the well-formedness of the overall expression, which is a runtime expression. These two functions are declared with the following type signatures:

$$\begin{array}{l}\texttt{NameOf}\ e\colon \texttt{EntityT}(N,\tau) \rightarrow \texttt{BoxT}(\{\mathit{List}\colon \{\mathit{Current}\colon \texttt{RecordAttr}_N\}\})\\ \texttt{LabelOf}\ \mathit{attr}\colon \texttt{AttributeT}(B)_N \rightarrow \texttt{BoxT}(\texttt{LabelAttr}(B)_N)\end{array}$$

When the template in Figure 6.1 is instantiated (during compile time) with an entity `Product` and its attribute `Description`, for example, the expression 6.2 evaluates during compile time as:

$$\begin{array}{l}\texttt{letbox}\ u_{name} = \texttt{NameOf}\ e\ \texttt{in}\\ \quad\texttt{letbox}\ u_{label} = \texttt{LabelOf}\ \mathit{attr}\ \texttt{in}\\ \quad\quad\texttt{Box(}\ u_{name}\,\hat{.}\,\mathit{List}\,\hat{.}\,\mathit{Current}\,\hat{.}\,u_{label}\ \texttt{)}\\ \hookrightarrow \texttt{Box(Product.}\mathit{List}.\mathit{Current}.\texttt{Description)}\end{array}$$

The result is a runtime expression that may be later evaluated.

We delimit the end of compile-time stage and the beginning of runtime with the `letbox` sentence. In a sentence with the form `letbox` $u = \texttt{Box}(M_1)$ `in` M_2, M_1 replaces all occurrences of u in M_2. When M_2 is not a runtime term (boxed term), M_2 is then evaluated. Since M_2 contains M_1, which is a runtime expression, M_1 is also evaluated.

This marks the beginning of the runtime stage. If we take the previous example's compile-time result, and want to proceed to its runtime computation, we can write and evaluate it as:

$$
\begin{aligned}
&\texttt{letbox}\ u = \texttt{Box}(\texttt{Product}\hat{.}\mathit{List}\hat{.}\mathit{Current}\hat{.}\texttt{Description})\ \texttt{in}\ u\\
\underset{R}{\hookrightarrow}\ &\texttt{Product}\hat{.}\mathit{List}\hat{.}\mathit{Current}\hat{.}\texttt{Description}\\
\underset{R}{\hookrightarrow}\ &\{\mathit{List} = \{\mathit{Current} = \{\\
&\qquad\qquad \texttt{Description} = \text{“}\texttt{Time turner}\text{”};\\
&\qquad\qquad \texttt{IsInStock} = \mathit{false}\}\}\\
&\}\hat{.}\mathit{List}\hat{.}\mathit{Current}\hat{.}\texttt{Description}\\
\underset{R}{\hookrightarrow}\ &\{\mathit{Current} = \{\\
&\qquad\qquad \texttt{Description} = \text{“}\texttt{Time turner}\text{”};\\
&\qquad\qquad \texttt{IsInStock} = \mathit{false}\}\\
&\}\hat{.}\mathit{Current}\hat{.}\texttt{Description}\\
\underset{R}{\hookrightarrow}\ &\{\texttt{Description} = \text{“}\texttt{Time turner}\text{”};\\
&\ \texttt{IsInStock} = \mathit{false}\\
&\}\hat{.}\texttt{Description}\\
\underset{R}{\hookrightarrow}\ &\text{“}\texttt{Time turner}\text{”}
\end{aligned}
$$

Note that the name `Product` is an identifier to the records of the entity `Product`, which contain its attributes. There may be other instances of `Product`. However, the label `Current` works as a runtime iterator, displaying only the current one, as depicted here.

Because our approach was inspired by Davies and Pfenning [8], presented in Chapter 3, and mimics its type rules related to multi-stage terms (concerning the `Box` and `letbox` constructors), we expect it to have cross-stage persistence and cross-stage safety and guarantee local soundness and local completeness. Local soundness guarantees that the elimination rules are not too strong. It ensures that we do not gain additional information after introducing a new connective and then eliminating it. The same information should be available without taking this detour. Local completeness guarantees that the elimination rules are not too weak. It ensures that we can recover all information after applying the elimination rule followed by the introduction rule.

6.2.2 Nested Templates and Parametric Polymorphism

One of the extensions brought by our implementation is the ability to instantiate templates inside the definition of another template (under submission [44]). For example, we can have a template for a screen containing some widgets, including a table that displays information about an entity and its attributes. The columns of the table can be defined using another template, such as the one in Figure 6.1. This means that our typechecking

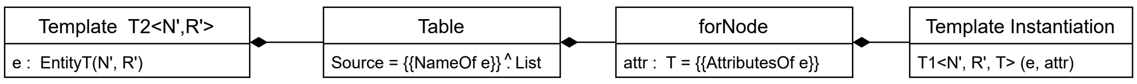


Figure 6.2: Model example of a template definition instantiating another template.

algorithm must also check for the validity of these instantiations. On one side, it needs to check the compatibility of the arguments used on the instantiation against the corresponding interface. On the other side, it needs to check the inner template definition against the specification of each parameter, i.e., verify each node and expression inside the template according to its parameters.

The example in Figure 6.2 shows a template `T2` that instantiates a template `T1` inside its definition. To verify the compatibility of the arguments, it needs to check that e and $attr$ have the types $\texttt{EntityT}(N,R)$ and $\texttt{AttributeT}(B)_N$, respectively, as the inner template expects, in Figure 6.1. N represents a generic name identifier of an entity, R a generic record of an entity, and B a generic type of an attribute. We can see in Figure 6.2 that e has type $\texttt{EntityT}(N',R')$, as defined by the `Template T2` node. Additionally, we know that $attr$ is an element of a list of attributes from entity e. Since typechecking occurs during compile time and before instantiation, we are typing an entity that is not yet instantiated, meaning that we do not know the concrete types of its attributes. Therefore, $attr$ will have type $\texttt{AttributeT}(\top)_{N'}$. This means that it is an attribute of some entity N', and its values have type $\top$. The top type, $\top$, behaves as a wildcard, representing any arbitrary type. This approach is sound due to the immutability of the attributes' list.

To verify if the types of the arguments and the parameters match, we use parametric polymorphism. By delivering N', R' and T $(T = \top)$ during the instantiation of `T1`, $\texttt{T1}\langle N',R',T\rangle(e,attr)$, we state that it must receive an argument of type $\texttt{EntityT}(N',R')$ and an argument of type $\texttt{AttributeT}(\top)_{N'}$. Hence, the expected types match the ones we instantiate `T1` with.

We implement parametric polymorphism for name, type and rows (records) variables.

6.2.3 Dependencies Between Types

Often, some templates require the verification of dependencies between types of parameters to ensure some relation between values, as in Figure 6.1, namely in the aforementioned runtime expression 6.1. Within an entity, we can only access its attributes, and therefore $attr$ must be an attribute of entity e for the whole expression to be well-typed. We ensure it through the entity and attribute types, which contain a common name N, the selection operation type represented as "**.**", and the resulting types of the functions `NameOf` and `LabelOf`.

The function `NameOf` e returns a name that maps to the entity's record. Here, we simplify and merely display that record. Such a record encloses other records that ultimately culminate in another containing the entity's attributes. The result of this function will have type $\{List\colon \{Current\colon \texttt{RecordAttr}_N\}\}$, where N is the name of the entity to which the attributes belong.

The function `LabelOf` *attr* represents the label of an attribute, thus conveying information about the type of the attribute's values, B, and the entity N' to which it belongs, $\texttt{LabelAttr}(B)_{N'}$.

We can use the same example as before, and instantiate e with the entity `Product`, and *attr* with its attribute `Description`. The intermediate selection operations, which have left associativity, when applied over the result of `NameOf` e:

$$\texttt{Product:}\quad \{List\colon \{Current\colon \texttt{RecordAttr}_{\texttt{Product}}\}\}$$

will type as follows:

$$\begin{aligned} \texttt{Product}\hat{.}List\colon &\quad \{Current\colon \texttt{RecordAttr}_{\texttt{Product}}\} \\ \texttt{Product}\hat{.}List\hat{.}Current\colon &\quad \texttt{RecordAttr}_{\texttt{Product}} \end{aligned}$$

The typing of the selection over the resulting record of attributes is done by comparing the two entities' names.

Selecting an attribute from a record of attributes with type $\texttt{RecordAttr}_N$, recurring to a label with type $\texttt{LabelAttr}(N')_B$, expresses the selection of an attribute that belongs to entity N' and whose values have type B, from a record of attributes from the entity N. If the names of the entities, N and N', match, the selection of the attribute is valid and safe. Thus, we prevent the selection of attributes that do not belong to that particular entity. The result of this operation is a value of the attribute, whose type is B.

In our example, the `LabelOf` function is applied to the attribute `Description`, from the entity `Product`. Therefore, this function's result has type $\texttt{LabelAttr(Product)}_{String}$. Hence, the final selection is typed as:

$$\texttt{Product}\hat{.}List\hat{.}Current\hat{.}\texttt{Description:}\quad String$$

These dependencies between types of parameters allow for the definition of more diverse templates, by introducing some restrictions to their usage and guaranteeing their appropriate instantiation and the consequent production of valid templates.

6.3 Syntax

Figure 6.5, on page 53, depicts the syntax of terms. In Figure 6.3 and Figure 6.4, we describe the categories of a node and the properties used inside some terms. Figure 6.6, on page 54, shows a subset of the terms of the language. Those terms are compile-time values and runtime terms. We obtain those values after compile-time evaluation, and their further evaluation proceeds at runtime. See Section 6.5 for more information on term evaluation. We present the syntax of types and the corresponding typing rules in Section 6.4.

$$
\begin{array}{lll}
\alpha ::= & & (\textbf{node categories}) \\
 & \varepsilon & \text{(empty)} \\
 | & \mathit{Top} & \text{(top)} \\
 | & \mathit{Screen} & \text{(screen)} \\
 | & \mathit{Table} & \text{(table)} \\
 | & \mathit{Column} & \text{(column)} \\
 | & \mathit{Icon} & \text{(icon)} \\
 | & \mathit{Expression} & \text{(expression)} \\
 | & \mathit{Input} & \text{(input)} \\
 | & \mathit{CheckBox} & \text{(check box)} \\
 | & \mathit{Calendar} & \text{(calendar)} \\
 | & \mathit{Container} & \text{(container)} \\
 | & \mathit{List} & \text{(list)} \\
 | & \mathit{ListItem} & \text{(list item)} \\
 | & \mathit{Search} & \text{(search)} \\
 | & \mathit{Chart} & \text{(chart)} \\
 | & \mathit{Counter} & \text{(counter)} \\
 | & \mathit{Pagination} & \text{(pagination)}
\end{array}
$$

Figure 6.3: Node categories.

$$
\begin{array}{lll}
p ::= & & (\textbf{properties}) \\
 & \mathit{Name} & \text{(name property)} \\
 | & \mathit{Title} & \text{(title property)} \\
 | & \mathit{Description} & \text{(description property)} \\
 | & \mathit{Type} & \text{(type property)} \\
 | & \mathit{DisplayName} & \text{(display name property)} \\
 | & \mathit{Source} & \text{(source property)} \\
 | & \mathit{Visible} & \text{(visible property)} \\
 | & \mathit{Value} & \text{(value property)} \\
 | & \mathit{InputType} & \text{(input type property)} \\
 | & \mathit{Variable} & \text{(variable property)} \\
 | & \mathit{Attributes} & \text{(attributes field)} \\
 | & \mathit{FilterBy} & \text{(filter property)} \\
 | & \mathit{AttrGroup} & \text{(attribute to group by property)}
\end{array}
$$

Figure 6.4: Properties.

Records, Lists and Projections. Records of the form $\{L_i = M_i{}^{i \in 1..p}\}$ represent a collection of pairs containing a label L_i and a term M_i. To project any term from a record, given a label L, we use the overloaded selection operation $M_1 \,\textbf{.}\, M_2$, where M_1 represents the record, and M_2 represents the label. The notation $[M_i{}^{i \in 1..p}]$ represents list collections of terms M_i. To refer to a specific element of the list, we use the indexing operation $M[\mathit{num}]$, where num is the index of the element we want to retrieve from list M.

Selection Operation. We define the selection operation, written as $M_1 \boldsymbol{.} M_2$, to represent projections of elements from a record M_1. However, the term M_1 can also represent nodes or attributes, which contain a record in their representations. The runtime selection operation is represented as $M_1 \hat{.} M_2$ and can also be applied to records, nodes or attributes.

Model Elements. Model elements include entities and attributes (database tables and rows, respectively). We define attributes with: the name of the entity N they belong to; a label L specific to the attribute; a type B which corresponds to the type of its values; and a record that maps properties to their expressions. Entities have a name identifier N, a record mapping labels L to their corresponding attributes, and a record mapping the same labels to a list of values of those attributes. For example, if we have an entity with name Product, which has attributes Description and IsInStock, we can represent it as:

```
Entity⟨ Product,
        { Description =
            Attribute⟨Product,Description,String,{DisplayName = “Description”}⟩;
         IsInStock =
            Attribute⟨Product,IsInStock,Bool,{DisplayName = “Is In Stock?”}⟩ }
        { Description = [“Chocolate frog”; “Bertie Bott’s Every Flavour Beans”];
         IsInStock = [true ; false] }
        ⟩
```

Model Nodes. Template nodes are the building blocks of the template model. They have a category α that defines what node they represent (e.g., screen, table, or expression). They also contain a record that maps their properties and respective expressions. Finally, they have a list of their children nodes. For example, a column node whose title is the display name of an attribute *attr* can be represented as:

$$\texttt{Node}\langle\texttt{Column}, \{\texttt{Title} = attr \boldsymbol{.} \texttt{DisplayName}\}, [\ldots]\rangle$$

We use ellipsis to simplify the example. Here, it omits nodes that are children of the column node, which may include Expression or Icon nodes, for instance. A Node is an element that is evaluated at compile time. A runtime node contains the same elements (category, record of properties, and children nodes), but is represented as a NodeValue: $\texttt{NodeValue}\langle\alpha, \{p_i = v_{1_i}{}^{i\in 1..p}\}, [v_{2_j}{}^{j\in 1..m}]\rangle$.

Node Categories and Element Properties. In Figure 6.3 we define the possible categories of a node, which corresponds to the template model node category. In Figure 6.4 we present the set of properties that may belong to model elements and model nodes. We integrate them in the syntax of a node as: $\texttt{Node}\langle\alpha, \{p_i = M_{1_i}{}^{i\in 1..p}\}, [M_{2_j}{}^{j\in 1..m}]\rangle$.

Built-in Operations. We define built-in operations to retrieve specific information from model elements and template nodes, namely `NameOf` M, `LabelOf` M, `AttributesOf` M, and M `isOfType` τ. Each receive a term M as argument, appart from the last one, that also receives a type τ.

Template Definitions, Template Instantiations and Let Binders. The template definition expression is represented as `Template`$\langle x,\ \tau,\ M\rangle$, where x is a typed parameter with type τ, and M is the body of the template. Standing for template instantiation we have $M_1(M_2)$ where M_1 is the template and M_2 is the argument. Without loss of generality, we only have one parameter in templates. For multiple parameters, we can define one template after another, one for each parameter. Let binders follow the usual construct of `let` $x = M_1$ `in` M_2 where M_1 denoted by x may appear in M_2. The definition and instantiation of a simple template that receives two arguments, can be represented as:

$$\begin{array}{l} \texttt{let}\ \ t = \texttt{Template}\langle x_1,\ \mathit{Num}, \\ \qquad\qquad \texttt{Template}\langle x_2,\ \mathit{Num}, \\ \qquad\qquad\quad \texttt{Node}\langle \texttt{Expression}, \{\texttt{Value} = x_1\}, []\rangle\ \rangle\rangle \\ \qquad \texttt{in}\ \ (t(3))(5) \end{array}$$

Conditionals and Loops. We represent conditional statements for handling decisions between nodes with the construct `ifNode`$(M,\ M_T,\ M_F)$, where M is a boolean expression, and M_T and M_F are its `then` and `else` branches, respectively. To define loops, we have the `forNode`$(x\colon \tau = M_1$ `in` $M_2)$ construct. Each element of the list M_1 denoted by x with type τ, may appear in the body M_2. Iterating over a collection of attributes will have the form `forNode`$(x\colon$ `AttributeT`$(t)_n = M_1$ `in` $M_2)$. Each attribute of the list is denoted by x, and may appear in the body of M_2. The values of each attribute will have type t, and n is the name of the entity they belong to. A usage example of these constructs is the creation of a node `Icon` or node `Expression` (depending on the attribute's type) for each attribute of an entity e:

$$\begin{array}{l} \texttt{forNode(}\ \mathit{attr}\colon \texttt{AttributeT}(t)_n = \texttt{AttributesOf}\ e\ \ \texttt{in} \\ \qquad\qquad \texttt{ifNode(}\mathit{attr}\ \texttt{isOfType}\ \mathit{Bool}\,, \\ \qquad\qquad\qquad \texttt{Node}\langle \texttt{Icon}, \{\}, []\rangle\,, \\ \qquad\qquad\qquad \texttt{Node}\langle \texttt{Expression}, \{\}, []\rangle\,)) \end{array}$$

Compile-time and Runtime Computation Stages. We represent runtime terms and expressions with the constructor `Box`(M), with M being the term. The destructor for runtime terms has the form `letbox` $u = M_1$ `in` M_2, where M_1 is the runtime term to unbox, which is denoted by u that may appear in M_2. Compile-time terms and expressions have no special notation. Hence, all terms that are not inside a `Box` are compile-time terms.

Polymorphic Abstraction and Application. To abstract over types, names and rows used in the language terms, we provide a polymorphic abstraction for each of the polymorphic variables (name, type and rows). For instance, we represent an abstraction on a name variable n on term M as $\Lambda_n n.M$. We define a way to replace each of the abstracted polymorphic variables. For instance, $M'[N]_n$ defines the name application, where the name N replaces the bound variable n in the abstraction term M'.

$$
\begin{array}{lcll}
vl & ::= & & (\textbf{value literals}) \\
 & & num & \text{(number literal)} \\
 & | & string & \text{(string literal)} \\
 & | & bool & \text{(boolean literal)} \\
 & & & \\
N & ::= & N_1 \mid N_2 \mid \ldots & (\textbf{name identifiers}) \\
 & & & \\
V & ::= & & (\textbf{model elements}) \\
 & & \texttt{Entity}\langle N, \{L_i = V_i{}^{i\in 1..p}\}, \{L_i = [vl_j{}^{j\in 1..m}]_i{}^{i\in 1..p}\}\rangle & \text{(entity element)} \\
 & | & \texttt{Attribute}\langle N, L, B, \{p_i = M_i{}^{i\in 1..p}\}\rangle & \text{(attribute element)} \\
 & & & \\
M & ::= & & (\textbf{template terms}) \\
 & & vl & \text{(value literal)} \\
 & | & x & \text{(compile-time variable)} \\
 & | & u & \text{(runtime variable)} \\
 & | & L & \text{(label)} \\
 & | & V & \text{(model element)} \\
 & | & \{L_i = M_i{}^{i\in 1..p}\} & \text{(record)} \\
 & | & [M_i{}^{i\in 1..p}] & \text{(list)} \\
 & | & M_1 \mathbin{\mathbf{.}} M_2 & \text{(selection operation)} \\
 & | & M_1 \mathbin{\hat{\mathbf{.}}} M_2 & \text{(runtime selection operation)} \\
 & | & \texttt{NameOf}\ M & \text{(name property)} \\
 & | & \texttt{LabelOf}\ M & \text{(label property)} \\
 & | & \texttt{AttributesOf}\ M & \text{(attributes)} \\
 & | & M\ \texttt{isOfType}\ \tau & \text{(type verification)} \\
 & | & \texttt{let}\ x = M_1\ \texttt{in}\ M_2 & \text{(let expression)} \\
 & | & \texttt{Template}\langle x,\ \tau,\ M\rangle & \text{(template declaration)} \\
 & | & M_1(M_2) & \text{(template instantiation)} \\
 & | & \texttt{Node}\langle \alpha, \{p_i = M_{1_i}{}^{i\in 1..p}\}, [M_{2_j}{}^{j\in 1..m}]\rangle & \text{(node element)} \\
 & | & \texttt{NodeValue}\langle \alpha, \{p_i = v_{1_i}{}^{i\in 1..p}\}, [v_{2_j}{}^{j\in 1..m}]\rangle & \text{(runtime node)} \\
 & | & \texttt{forNode}(x\colon t = M_1\ \texttt{in}\ M_2) & \text{(loop instruction node)} \\
 & | & \texttt{forNode}(x\colon \texttt{AttributeT}(t)_n = M_1\ \texttt{in}\ M_2) & \text{(loop for attributes node)} \\
 & | & \texttt{ifNode}(M,\ M_T,\ M_F) & \text{(conditional branching node)} \\
 & | & \texttt{Box}(M) & \text{(runtime term constructor)} \\
 & | & \texttt{letbox}\ u = M_1\ \texttt{in}\ M_2 & \text{(runtime term destructor)} \\
 & | & M_1[M_2] & \text{(indexing)} \\
 & | & \Lambda_n n.M & \text{(name abstraction)} \\
 & | & \Lambda_t t.M & \text{(type abstraction)} \\
 & | & \Lambda_r r.M & \text{(row abstraction)} \\
 & | & M[name]_n & \text{(name application)} \\
 & | & M[type]_t & \text{(type application)} \\
 & | & M[row]_r & \text{(row application)}
\end{array}
$$

Figure 6.5: Syntax of the template language.

$$
\begin{array}{lcll}
v & ::= & & (\textbf{values}) \\
 & & vl & \\
 & | & \mathtt{Entity}\langle N, \{L_i = v_i{}^{i \in 1..p}\}, \{L_i = [vl_j{}^{j \in 1..m}]_i{}^{i \in 1..p}\}\rangle & \\
 & | & \mathtt{Attribute}\langle N, L, B, \{p_i = v_i{}^{i \in 1..p}\}\rangle & \\
 & | & L & \\
 & | & \{L_i = v_i{}^{i \in 1..p}\} & \\
 & | & [v_i{}^{i \in 1..p}] & \\
 & | & v_1 \mathbin{\hat{:}} v_2 & \\
 & | & \mathtt{let}\ x = v_1\ \mathtt{in}\ v_2 & \\
 & | & \mathtt{NodeValue}\langle \alpha, \{p_i = v_{1_i}{}^{i \in 1..p}\}, [v_{2_j}{}^{j \in 1..m}]\rangle & \\
 & | & \mathtt{Box}(M) &
\end{array}
$$

Figure 6.6: Syntax of compile-time values (runtime terms).

6.4 Type System

6.4.1 Syntax of Types

$$
\begin{array}{lll l}
B & ::= & & (\textbf{basic types}) \\
 & & \mathit{Num} & (\text{number}) \\
 & | & \mathit{String} & (\text{string}) \\
 & | & \mathit{Bool} & (\text{bool}) \\
 & & & \\
\tau & ::= & & (\textbf{types}) \\
 & & B & (\text{basic types}) \\
 & | & \mathtt{Name}(N) & (\text{name}) \\
 & | & \mathtt{Label}(L) & (\text{label}) \\
 & | & \mathtt{LabelAttr}(B)_n & (\text{attribute label}) \\
 & | & \{L_i : \tau_i{}^{i\in 1..p}\} & (\text{record}) \\
 & | & \mathtt{RecordAttr}_n & (\text{record of entity's attributes}) \\
 & | & [\tau] & (\text{list}) \\
 & | & \mathtt{ListAttr}_n & (\text{list of entity's attributes}) \\
 & | & \mathtt{EntityT}(n, \{L_i : \tau_i{}^{i\in 1..p}\}) & (\text{entity}) \\
 & | & \mathtt{AttributeT}(\tau)_n & (\text{attribute}) \\
 & | & \mathtt{NodeT}([\alpha_i{}^{i\in 1..p}], \{p_j : \tau_j{}^{j\in 1..m}\}) & (\text{node}) \\
 & | & \mathtt{BoxT}(\tau) & (\text{delayed type}) \\
 & | & \mathtt{TemplateT}(\tau_1 \to \tau_2) & (\text{template}) \\
 & | & n & (\text{name variable}) \\
 & | & t & (\text{type variable}) \\
 & | & r & (\text{row variable}) \\
 & | & \forall_n n.\tau & (\text{forall name}) \\
 & | & \forall_t t.\tau & (\text{forall type}) \\
 & | & \forall_r r.\tau & (\text{forall rows}) \\
 & | & \top & (\text{top})
\end{array}
$$

Figure 6.7: Syntax of types.

In Figure 6.7, we define the grammar for the syntax of types.

Basic Types, Names, Labels and Collections. The basic types comprise *Num*, *String*, and *Bool*. A name identifier N has type $\mathtt{Name}(N)$. A label L can either have type $\mathtt{Label}(L)$ or $\mathtt{LabelAttr}(B)_N$. In the first one, L is the description of the label. The latter represents a label of a generic attribute with type B from entity N. A record $\{L_i : \tau_i{}^{i\in 1..p}\}$ is a collection of pairs containing a label L_i and a type τ_i. We represent a generic record of attributes from entity N by $\mathtt{RecordAttr}_N$. A list $[\tau]$ represents a collection of elements of type τ. Finally, we use *Top* as a supertype that can represent any possible type.

Model Elements, Template Nodes and Template Definition. We represent the type of an entity as $\mathtt{EntityT}(N, \{L_i : \tau_i{}^{i\in 1..p}\})$, with a name N, and a record that maps labels to the

$$
\begin{array}{lcll}
\Gamma & ::= & & (\textbf{compile-time contexts}) \\
 & & \varnothing & (\text{empty context}) \\
 & | & \Gamma, x : \tau & (\text{term variable binding}) \\
\\
\Delta & ::= & & (\textbf{runtime contexts}) \\
 & & \varnothing & (\text{empty context}) \\
 & | & \Delta, u : \tau & (\text{term variable binding}) \\
\\
\Omega & ::= & & (\textbf{name variable contexts}) \\
 & & \varnothing & (\text{empty context}) \\
 & | & \Omega, n & (\text{name variable binding}) \\
\\
\Phi & ::= & & (\textbf{rows variable contexts}) \\
 & & \varnothing & (\text{empty context}) \\
 & | & \Phi, r & (\text{rows variable binding}) \\
\\
\Upsilon & ::= & & (\textbf{type variable contexts}) \\
 & & \varnothing & (\text{empty context}) \\
 & | & \Upsilon, t & (\text{type variable binding})
\end{array}
$$

Figure 6.8: Syntax of contexts.

type of the entity's attributes. Attributes are represented as $\texttt{AttributeT}(B)_N$, with B being the type of its values, and N the name of the entity it belongs to. Template nodes have type $\texttt{NodeT}([\alpha_i{}^{i\in 1..p}], \{p_i : \tau_i{}^{i\in 1..p}\})$, where the list of α's represents the possible categories of the node, and the record contains the type τ of its properties p. Template definition type works as a function from τ_1 to τ_2, and has the form $\texttt{TemplateT}(\tau_1 \to \tau_2)$.

Runtime Type (Delayed Type). We represent the type of a runtime term as $\texttt{BoxT}(\tau)$, where τ is its type during the runtime stage.

Parametric Polymorphism. We define three different generic types, one for each of the polymorphic variables (n, t and r). These variables are bound by universal quantifiers. For instance, $\forall_n n.M$ binds the name-type polymorphic variable n in term M.

6.4.2 Typechecking

We define typing rules for assigning types to terms. Judgments, which are part of such rules, have the form $\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M : \tau$, where the term M has type τ in the contexts, or environments, defined to the left of the judgment. Environments, or contexts, are a set of assumptions about the types of the free variables in M. We define five environments, where Γ is the ordinary context that stores the types of the compile-time variables, Δ is the modal context or runtime context and stores the types of the runtime variables, and Ω, Υ and Φ is where we store the polymorphic variables name, type and rows, respectively. A context is defined solely by its greek letter, e.g. Γ, or extended with an additional mapping of a variable to a type, $(\Gamma, x\colon \tau)$, or by an empty context instead, denoted by ∅ .

Note that Figure 6.7 defines the syntax of types, and Figure 6.8 defines the syntax of contexts.

Literals. Literal values, name identifiers and labels have their own type:

$$\frac{}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \mathit{num}\colon \mathit{Num}}\ [\![\text{T-Num}]\!] \qquad \frac{}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \mathit{string}\colon \mathit{String}}\ [\![\text{T-Str}]\!]$$

$$\frac{}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \mathit{bool}\colon \mathit{Bool}}\ [\![\text{T-Bool}]\!]$$

$$\frac{}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash N\colon \texttt{Name}(N)}\ [\![\text{T-Name}]\!] \qquad \frac{}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash L\colon \texttt{Label}(L)}\ [\![\text{T-Lab}]\!]$$

Collections. Records and lists are typed according to the type of each of their elements. For example, the record {Description = *"Time-turner"*; IsInStock = *true*; Quantity = 3}, has type {Description: *String*; IsInStock: *Bool*; Quantity: *Num*}. Lists are uniform and all their elements have the same type. Hence, the list type contains a single type. The typing rules for records and lists are as follows:

$$\frac{\text{for each } i \qquad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_i\colon \tau_i}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \{L_i = M_i{}^{i\in 1..p}\}\colon \{L_i\colon \tau_i{}^{i\in 1..p}\}}\ [\![\text{T-Rec}]\!]$$

$$\frac{\text{for each } i \qquad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_i\colon \tau}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash [M_i{}^{i\in 1..p}]\colon [\tau]}\ [\![\text{T-List}]\!]$$

The next typing rule (T-Idx) defines the type of an element of a list. If M_1 has the type list with all of its elements having a type τ, and if M_2 is a number, then $M_1[M_2]$ has the type corresponding to its elements' types, which is τ:

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1\colon [\tau] \qquad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2\colon \mathit{Num}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1[M_2]\colon \tau}\ [\![\text{T-Idx}]\!]$$

The projection, or selection, of an element from a record has several typing rules. The compile-time selection term is represented as $M_1 \,\textbf{.}\, M_2$ (there is a runtime variant of this operation). For this term to be well-typed, M_1 must have the type record, or the type of something containing a record (namely, node or attribute), and M_2 the type label. Because this operation occurs and is evaluated only at compile time, and its result will be part of a runtime element, then its type is always a boxed type. If the record M_1 is a collection of pairs (a record), and one of its pairs contains the label from M_2, then its resulting type is the boxed type of the type paired with that label:

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : \{L_i : \tau_i \ ^{i\in 1..p}\} \\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2 : \mathtt{Label}(L_j) \qquad j \in 1..p\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 \,\mathbf{.}\, M_2 : \mathtt{BoxT}(\tau_j)} \; [\![\text{T-Sel1}]\!]$$

Using the same example as presented above, if M_1 is a record of type {Description: *String*; IsInStock: *Bool*; Quantity: *Num*}, the term $M_1 \,\mathbf{.}\, \mathit{Description}$ will have a box of the type that pairs with that label, i.e., BoxT(*String*).

The second rule is similar to the first one. But instead of M_1 being a record, it is an element that contains a record, namely a node:

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : \mathtt{NodeT}([\alpha_i \ ^{i\in 1..p}], \{p_j : \tau_j \ ^{j\in 1..m}\}) \\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2 : \mathtt{Label}(L_k) \qquad k \in 1..m\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 \,\mathbf{.}\, M_2 : \mathtt{BoxT}(\tau_k)} \; [\![\text{T-Sel2}]\!]$$

The last typing rule defines the type of selecting a property from an attribute. All attributes have the same pre-defined properties in the OutSystems environment. Therefore, the type of each property is pre-defined and does not change. Here, we define the rule for typing only one of its properties (`DisplayName`) as an example, which is the one we need for our evaluation:

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : \mathtt{AttributeT}(B)_N \\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2 : \mathtt{Label}(\mathtt{DisplayName})\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 \,\mathbf{.}\, M_2 : \mathtt{BoxT}(\mathit{String})} \; [\![\text{T-Sel3}]\!]$$

Additionally, there is a selection operation, represented as $M_1 \hat{:} M_2$, that always occurs within a box. Hence, this operation only evaluates at runtime, and its typing is similar to the previous operation but does not result in a boxed type. There is a typing rule in case M_1 is a record (T-SelRT1), and another in case M_1 is a node (T-SelRT2).

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : \{L_i : \tau_i \ ^{i\in 1..p}\} \\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2 : \mathtt{Label}(L_j) \qquad j \in 1..p\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 \hat{:} M_2 : \tau_j} \; [\![\text{T-SelRT1}]\!]$$

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : \mathtt{NodeT}([\alpha_i \ ^{i\in 1..p}], \{p_j : \tau_j \ ^{j\in 1..m}\}) \\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2 : \mathtt{Label}(L_k) \qquad k \in 1..m\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 \hat{:} M_2 : \tau_k} \; [\![\text{T-SelRT2}]\!]$$

Since the values of an attribute's properties are available at compile time, the selection of an attribute's property (previously presented in T-Sel3) does not occur with this runtime selection operation. Hence, a corresponding rule does not exist.

If the record has the type of a more generic record of attributes from an entity N, and the label of the attribute to select belongs to the same entity N, then the selection operation results in the type B of the selected attribute's values. This selection intends to retrieve the value of an entity's attributes, which is only available at runtime. Therefore, this rule only applies to this runtime operation:

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1\colon \texttt{RecordAttr}_N \qquad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2\colon \texttt{LabelAttr}(B)_N}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 \mathbin{\dot{:}} M_2\colon B}\ [\![\text{T-SelRT3}]\!]$$

Variables. We define two typing rules for variables, one for compile-time variables (T-CVar), and another for runtime variables (T-RVar). The premise $x\colon \tau \in \Gamma$ reads as "The type assumed for x in Γ is τ.". Hence, a variable has any type we are currently assuming it to have.

$$\frac{x\colon \tau \in \Gamma}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash x\colon \tau}\ [\![\text{T-CVar}]\!] \qquad \frac{u\colon \tau \in \Delta}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash u\colon \tau}\ [\![\text{T-RVar}]\!]$$

Model Elements. An entity type is defined according to its name N, and the type of each of its attributes. The entity is well-formed if both of its records are related, i.e., the first record contains pairs of labels L_i and attributes, and the second pairs the same labels to lists of values of the corresponding attributes. The typing rule for entities is as follows:

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash N\colon \texttt{Name}(N)\\ \text{for each } i \qquad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_i\colon \texttt{AttributeT}(B_i)_N\\ \text{for each } i,j \qquad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash v_{ji}\colon B_i\end{array}}{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{Entity}\langle N, \{L_i = M_i{}^{i\in 1..p}\}, \{L_i = [v_j{}^{j\in 1..m}]_i{}^{i\in 1..p}\}\rangle\colon\\ \texttt{EntityT}(N, \{L_i\colon \texttt{AttributeT}(B_i)_N{}^{i\in 1..p}\})\end{array}}\ [\![\text{T-Ent}]\!]$$

An attribute type is defined according to its basic type B and the name N of its entity. For an attribute to be well-formed, its properties' values must be well-formed as well. The typing rule for attributes is:

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash N\colon \texttt{Name}(N)\\ \text{for each } i \qquad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_i\colon \tau_i\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{Attribute}\langle N, L, B, \{p_i = M_i{}^{i\in 1..p}\}\rangle\colon \texttt{AttributeT}(B)_N}\ [\![\text{T-Attr}]\!]$$

Built-in Operations. For the term `NameOf` M to be well-formed, its argument M must be of type entity. The resulting type of the term is a boxed type (delayed / runtime type) of nested records that ultimately culminate in a record of attributes of that entity:

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M\colon \texttt{EntityT}(N, \{L_i\colon \texttt{AttributeT}(B_i)_N{}^{i\in 1..p}\})}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{NameOf}\ M\colon \texttt{BoxT}(\{\mathit{List}\colon \{\mathit{Current}\colon \texttt{RecordAttr}_N\}\})}\ [\![\text{T-NOf}]\!]$$

The term `LabelOf` M is well-formed if M has the type attribute. The resulting type is a boxed type of a label of that attribute that contains its values' type B and its entity name N:

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M\colon \texttt{AttributeT}(B)_N}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{LabelOf}\ M\colon \texttt{BoxT}(\texttt{LabelAttr}(B)_N)}\ [\![\text{T-LO}\textsc{f}]\!]$$

The term `AttributesOf` M has the type list if M is an entity. Because we are typing the definition of templates, M is not a concrete entity yet. Therefore, the resulting list cannot have the types of each of its elements, but instead will have the generic type of a list of attributes of an entity N, $\texttt{ListAttr}_N$:

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M\colon \texttt{EntityT}(N,R)}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{AttributesOf}\ M\colon \texttt{ListAttr}_N}\ [\![\text{T-AO}\textsc{f}]\!]$$

For the type verification term M `isOfType` τ, which has type *Bool*, to be well-typed, M must also be well-typed:

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M\colon \tau'}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M\ \texttt{isOfType}\ \tau\colon \mathit{Bool}}\ [\![\text{T-O}\textsc{f}\text{T}\textsc{y}]\!]$$

Let Sentences and Templates. The rule for typing let sentences tells us that if M_1 evaluates to a result in τ_1, and if M_2 has type τ_2 under the assumption that x has type τ_1, then the evaluation result of the let sentence will have the type τ_2:

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1\colon \tau_1 \qquad (\Gamma, x\colon \tau_1);\Delta;\Omega;\Phi;\Upsilon \vdash M_2\colon \tau_2}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{let}\ x = M_1\ \texttt{in}\ M_2\colon \tau_2}\ [\![\text{T-L}\textsc{et}]\!]$$

Template definition is typed similarly to abstractions in λ-calculus. In the template definition term, x is already typed as τ_1. The premise states that M evaluates to a result in τ_2 when x is assumed to be of type τ_1. Hence, the template definition type maps τ_1 arguments to τ_2 results:

$$\frac{(\Gamma, x\colon \tau_1);\Delta;\Omega;\Phi;\Upsilon \vdash M\colon \tau_2}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{Template}\langle x,\ \tau_1,\ M\rangle\colon \texttt{TemplateT}(\tau_1 \to \tau_2)}\ [\![\text{T-T}\textsc{emp}]\!]$$

The typing rule for template instantiation, $M_1(M_2)$, is similar to that of function applications. If M_1 evaluates to a template mapping arguments in τ_1 to results in τ_2, and if M_2 evaluates to a result in τ_1, then the result of applying M_1 to M_2 is a value of type τ_2:

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1\colon \texttt{TemplateT}(\tau_1 \to \tau_2)\\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2\colon \tau_1\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1(M_2)\colon \tau_2}\ [\![\text{T-I}\textsc{nst}]\!]$$

Nodes. Nodes are well-typed if the terms M_i of their properties' values are well-typed, $\mathtt{BoxT}(\tau_i)$, and if the list of terms M_j's they contain are well-typed runtime nodes (boxed nodes). The type of a node is a boxed type represented by a list with its single α category, and a record mapping its properties to their respective runtime types τ_i:

$$\frac{\begin{array}{ll}\text{for each } i & \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_i : \mathtt{BoxT}(\tau_i) \\ \text{for each } j & \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_j : \mathtt{BoxT}(\mathtt{NodeT}([\alpha_h{}^{h\in 1..f}],\{p_k : \tau_k{}^{k\in 1..l}\})_j)\end{array}}{\begin{array}{l}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \mathtt{Node}\langle\alpha,\{p_i = M_i{}^{i\in 1..p}\},[M_j{}^{j\in 1..m}]\rangle : \\ \qquad \mathtt{BoxT}(\mathtt{NodeT}([\alpha],\{p_i : \tau_i{}^{i\in 1..p}\}))\end{array}} \text{ [[T-NODE]]}$$

Loops. There are two terms that define loop instructions. When the loop has the form $\mathtt{forNode}(x\colon t = M_1 \mathtt{~in~} M_2)$ (T-FOR), M_1 must be a list of elements with type τ. Under the assumption of x being the element with type τ, and t having x's corresponding type, M_2 must have the type of a runtime node (boxed node). The resulting type of the loop node must be the same as the type of M_2 under such assumptions:

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : [\tau_1] \\ (\Gamma, x\colon t);\Delta;\Omega;\Phi;(\Upsilon, t = \tau_1) \vdash M_2 : \mathtt{BoxT}(\tau_2) \\ \tau_2 = \mathtt{NodeT}([\alpha_j{}^{j\in 1..m}],\{p_k : \tau_k{}^{k\in 1..l}\})\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \mathtt{forNode}(x\colon t = M_1 \mathtt{~in~} M_2) : \mathtt{BoxT}(\tau_2)} \text{ [[T-FOR]]}$$

Because we are typing template definitions, some elements were not instantiated yet. Therefore, M_1 might be a generic collection (T-FOR3). If M_1 is a generic list of attributes from some entity N. And if M_2 types to a value with the type of a runtime node, under the assumption of x being a generic attribute from entity n with type t, and $t = \top$ and $n = N$. Then, the loop will have the type of a runtime node obtained from the typing of M_2.

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : \mathtt{ListAttr}_N \\ (\Gamma, x\colon \mathtt{AttributeT}(t)_n);\Delta;(\Omega, n = N);\Phi;(\Upsilon, t = \top) \vdash M_2 : \mathtt{BoxT}(\tau_2) \\ \tau_2 = \mathtt{NodeT}([\alpha_i{}^{i\in 1..p}],\{p_j : \tau_j{}^{j\in 1..m}\})\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \mathtt{forNode}(x\colon \mathtt{AttributeT}(t)_n = M_1 \mathtt{~in~} M_2) : \mathtt{BoxT}(\tau_2)} \text{ [[T-FORATTR]]}$$

Conditionals. The conditional node is well-typed if its first term has type *Bool*, and the terms M_T and M_F have the type of a boxed node. The result of the if-node is a value with the type of a boxed node. This node type is defined by the union of the node categories of both branches and the intersection of their property record types.

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M : \mathit{Bool} \\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_T : \mathtt{BoxT}(\mathtt{NodeT}([\alpha_i{}^{i\in 1..p}],\{p_j : \tau_j{}^{j\in 1..m}\})) \\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_F : \mathtt{BoxT}(\mathtt{NodeT}([\alpha_h{}^{h\in 1..f}],\{p_k : \tau_k{}^{k\in 1..l}\}))\end{array}}{\begin{array}{l}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \mathtt{ifNode}(M, M_T, M_F) : \\ \qquad \mathtt{BoxT}(\mathtt{NodeT}([\alpha_i{}^{i\in 1..p}] \cup [\alpha_h{}^{h\in 1..f}],\{p_j : \tau_j{}^{j\in 1..m}\} \cap \{p_k : \tau_k{}^{k\in 1..l}\}))\end{array}} \text{ [[T-IF]]}$$

To understand the reason behind the type of the if-node, we present a small example, where attr is an attribute:

$$\begin{array}{l} \texttt{ifNode(}\ \texttt{attr isOfType}\ \mathit{Bool} \\ \qquad \texttt{Node}\langle \mathit{Icon}, \{\texttt{Value} = 3;\ \texttt{Visible} = \mathit{true}\}, []\rangle \\ \qquad \texttt{Node}\langle \mathit{Expression}, \{\texttt{Value} = 10\}, []\rangle) \end{array}$$

The type of this term is: $\texttt{BoxT}(\texttt{NodeT}([\mathit{Icon};\ \mathit{Expression}], \{\texttt{Value}\colon \mathit{Num}\}))$. Note that this type shows the possible nodes we can obtain from this term, which correspond to the union of the α categories, Icon and Expression. It also shows the properties that are guaranteed to occur, obtained through their intersection. In this case, only the Value property is guaranteed to be present. This allows us to know we can only safely apply operations over that property.

Runtime Type (Delayed Type). We draw the reader's attention to the next rule (T-Box). If the term M has type τ, then the term $\texttt{Box}(M)$ has type $\texttt{BoxT}(\tau)$. Notice, however, that the premise enforces that τ is valid by requiring the ordinary context to be empty. This means that only runtime variables from Δ can occur free in M (apart from type polymorphic variables).

$$\frac{\emptyset;\Delta;\Omega;\Phi;\Upsilon \vdash M\colon \tau}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{Box}(M)\colon \texttt{BoxT}(\tau)}\ [\![\text{T-Box}]\!]$$

The next typing rule defines the corresponding elimination rule. If M_1 has type $\texttt{BoxT}(\tau_1)$, and if M_2 has type τ_2 under the assumption that u has type τ_1, then the result of evaluating the letbox sentence, i.e., the result of M_2 after unboxing M_1, is of type τ_2.

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1\colon \texttt{BoxT}(\tau_1) \qquad \Gamma;(\Delta,u\colon \tau_1);\Omega;\Phi;\Upsilon \vdash M_2\colon \tau_2}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{letbox}\ u = M_1\ \texttt{in}\ M_2\colon \tau_2}\ [\![\text{T-LetB}]\!]$$

Parametric Polymorphism. Finally, after adding the type polymorphic variable n to the corresponding environment Ω, if the term M has type τ, then the name abstraction term, $\Lambda_n n.M$, has type $\forall_n n.\tau$ (T-FAN). We have similar rules for each of the other polymorphic variables, but we add those type variables to their corresponding environments (Ω for name variables, Υ for type variables, and Φ for record or rows variables):

$$\frac{\Gamma;\Delta;(\Omega,n);\Phi;\Upsilon \vdash M\colon \tau}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \Lambda_n n.M\colon \forall_n n.\tau}\ [\![\text{T-FAN}]\!] \quad \frac{\Gamma;\Delta;\Omega;\Phi;(\Upsilon,t) \vdash M\colon \tau}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \Lambda_t t.M\colon \forall_t t.\tau}\ [\![\text{T-FAT}]\!]$$

$$\frac{\Gamma;\Delta;\Omega;(\Phi,r);\Upsilon \vdash M\colon \tau}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \Lambda_r r.M\colon \forall_r r.\tau}\ [\![\text{T-FAR}]\!]$$

We define rules for polymorphic type applications for each variable, presented in T-NApp, T-TApp and T-RApp. By performing the corresponding type application to the abstraction term M, the resulting type is τ, where the name N, the rows R, or the type T replaces the bound variable in that term:

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M\colon \forall_n n.\tau}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M[N]_n\colon [N/n]\tau}\ [\![\text{T-NA}\textsc{pp}]\!]$$

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M\colon \forall_t t.\tau}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M[T]_t\colon [T/t]\tau}\ [\![\text{T-TA}\textsc{pp}]\!]$$

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M\colon \forall_r r.\tau}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M[R]_r\colon [R/r]\tau}\ [\![\text{T-RA}\textsc{pp}]\!]$$

6.5 Operational Semantics

In this section, we define the semantics of the language, i.e., how terms are evaluated. The evaluation of this language comprises both the compile-time and the runtime computations. Terms are evaluated into other terms, since the result of their compile-time execution must represent an OutSystems' template model, represented with a fragment of the same syntax.

Next, we show the small-step operational semantics for this language. The evaluation relation on terms has the format $M_1 \hookrightarrow M_2$, which reads as "M_1 evaluates to M_2 in one step".

We distinguish the evaluation that occurs at compile time and runtime.

6.5.1 Compile-time Semantics

Built-in Operations. For the built-in operations `NameOf` M, `LabelOf` M, `AttributesOf` M and M `isOfType` τ, we first evaluate their subterm M. Once their subterm is evaluated, the operation is applied.

The `NameOf` operation, when applied to an entity, evaluates to its name, which points to a box with nested records that contains runtime values for each of its attributes, as presented in E-NOfV. These records include the built-in labels *List* and *Current*, used to access the list of records of an entity and the current record of a list being iterated, respectively. In the rule, we select an arbitrary runtime value, vl_x, as an example of the cursor of the entity. This structure results directly from the structure of iterators in the

OutSystems language.

$$\frac{M \hookrightarrow M'}{\texttt{NameOf}\ M \hookrightarrow \texttt{NameOf}\ M'}\ [\![\text{E-NOf}]\!]$$

$$\frac{}{\begin{array}{c}\texttt{NameOf}\ (\texttt{Entity}\langle n, \{L_i = V_i\ ^{i\in 1..p}\}, \{L_i = [vl_j\ ^{j\in 1..m}]_i\ ^{i\in 1..p}\}\rangle) \\ \hookrightarrow \texttt{Box}(\{List = \{Current = \{L_i = vl_{x_i}\ ^{i\in 1..p}\}\}\})\end{array}}\ [\![\text{E-NOfV}]\!]$$

The `LabelOf` operation, when applied to an attribute, evaluates to a runtime (boxed) label of that attribute (E-LOfV).

$$\frac{M \hookrightarrow M'}{\texttt{LabelOf}\ M \hookrightarrow \texttt{LabelOf}\ M'}\ [\![\text{E-LOf}]\!]$$

$$\frac{}{\texttt{LabelOf}\ (\texttt{Attribute}\langle N, L, B, \{p_i = M_i\ ^{i\in 1..p}\}\rangle) \hookrightarrow \texttt{Box}(L)}\ [\![\text{E-LOfV}]\!]$$

We remind the reader that boxed terms correspond to terms that appear in compile time but whose content must only be executed at runtime. We can see it as code that will execute at runtime.

The `AttributesOf` operation is applied to an entity and, as the name implies, evaluates to a list containing its attributes.

$$\frac{M \hookrightarrow M'}{\texttt{AttributesOf}\ M \hookrightarrow \texttt{AttributesOf}\ M'}\ [\![\text{E-AOf}]\!]$$

$$\frac{}{\begin{array}{c}\texttt{AttributesOf}\ (\texttt{Entity}\langle n, \{L_i = V_i\ ^{i\in 1..p}\}, \{L_i = [vl_j\ ^{j\in 1..m}]_i\ ^{i\in 1..p}\}\rangle) \\ \hookrightarrow [V_i\ ^{i\in 1..p}]\end{array}}\ [\![\text{E-AOfV}]\!]$$

The term with the form M `isOfType` τ, when applied to an attribute term, evaluates to a boolean that is the direct comparison between τ and the type B of the attribute (E-OfTyV). We use $(\!|B = \tau|\!)$ to represent the standard interpretation of comparing both types.

$$\frac{M \hookrightarrow M'}{M\ \texttt{isOfType}\ \tau \hookrightarrow M'\ \texttt{isOfType}\ \tau}\ [\![\text{E-OfTy}]\!]$$

$$\frac{}{(\texttt{Attribute}\langle n, L, B, \{p_i : M_i\ ^{i\in 1..p}\}\rangle)\ \texttt{isOfType}\ \tau \hookrightarrow (\!|B = \tau|\!)}\ [\![\text{E-OfTyA}]\!]$$

Let Sentence. In the let sentence, we first evaluate its first term M_1 (E-Let). Once this first evaluation finishes and we obtain its final term, represented as v_1 in E-LetV (which should not be mistaken for the representation of a literal value), then the sentence evaluates to M_2 under an environment where x maps to term v_1.

$$\frac{M_1 \hookrightarrow M_1'}{\texttt{let}\ x = M_1\ \texttt{in}\ M_2 \hookrightarrow \texttt{let}\ x = M_1'\ \texttt{in}\ M_2}\ [\![\text{E-Let}]\!]$$

$$\frac{}{\texttt{let}\ x = v_1\ \texttt{in}\ M_2 \hookrightarrow [x \mapsto v_1]M_2}\ [\![\text{E-LetV}]\!]$$

Template Instantiation. We define four rules for template instantiation. Starting with the term $M_1(M_2)$, we first evaluate M_1 (E-Inst1). Once we obtain a template definition from that evaluation, we evaluate M_2 (E-Inst2). Once that evaluation reaches an end (v_2, which again should not be mistaken for a literal value), the evaluation of applying the template to that argument results in the term M_3, under an environment where x maps to v_2 (E-Inst3). Note that we can define a template with more than one argument by nesting templates.

$$\frac{M_1 \hookrightarrow M_1'}{M_1(M_2) \hookrightarrow M_1'(M_2)}\ [\![\text{E-Inst1}]\!]$$

$$\frac{M_2 \hookrightarrow M_2'}{(\texttt{Template}\langle x,\ \tau,\ M_3\rangle)(M_2) \hookrightarrow (\texttt{Template}\langle x,\ \tau,\ M_3\rangle)(M_2')}\ [\![\text{E-Inst2}]\!]$$

$$\frac{}{(\texttt{Template}\langle x,\ \tau,\ M_3\rangle)(v_2) \hookrightarrow [x \mapsto v_2]M_3}\ [\![\text{E-Inst3}]\!]$$

Nodes. The evaluation of nodes occurs by evaluating their property records (E-Node1) and their children nodes (E-Node2). After the evaluation of all its subcomponents, the node comprises boxed terms as property values and boxed nodes as children. This means that its compile-time evaluation came to an end, and it must now be turned into a runtime computation. The third rule (E-Node3) shows how such a node is evaluated to a `NodeValue` (corresponding runtime node). To avoid nested boxed terms and maintain two-stage computation, the boxes of its subcomponents are removed, and the whole nodevalue is wrapped inside an outer box. This works similarly to the `letbox` destructor: a node is a runtime value (boxed value), and we are introducing other runtime values inside it (its properties' values).

$$\frac{M_1 \hookrightarrow M_1'}{\texttt{Node}\langle \alpha, M_1, M_2 \rangle \hookrightarrow \texttt{Node}\langle \alpha, M_1', M_2 \rangle} \; [\![\text{E-Node1}]\!]$$

$$\frac{M_2 \hookrightarrow M_2'}{\texttt{Node}\langle \alpha, v_1, M_2 \rangle \hookrightarrow \texttt{Node}\langle \alpha, v_1, M_2' \rangle} \; [\![\text{E-Node2}]\!]$$

$$\frac{}{\begin{array}{c}\texttt{Node}\langle \alpha, \{p_i : \texttt{Box}(v_{1_i})\ ^{i \in 1..p}\}, [\texttt{Box}(v_{2_j})\ ^{j \in 1..m}] \rangle \\ \hookrightarrow \texttt{Box}(\texttt{NodeValue}\langle \alpha, \{p_i : v_{1_i}\ ^{i \in 1..p}\}, [v_{2_j}\ ^{j \in 1..m}] \rangle)\end{array}} \; [\![\text{E-Node3}]\!]$$

Take as example the following node, obtained after applying the first two rules:

```
Node⟨ Column,
      {Title = Box("Price")},
      [Box(Node⟨Icon, {}, []⟩)] ⟩
```

According to the rule E-Node3, it evaluates to:

```
Box( NodeValue⟨ Column,
                {Title = "Price"},
                [Node⟨Icon, {}, []⟩] ⟩ )
```

Collections. We define the evaluation of records and lists in the following rules. When one of their elements can be reduced to another, the collection evaluates to a similar record or list replacing that element with its reduced form. This reduction occurs from left to right.

$$\frac{M_j \hookrightarrow M_j'}{\begin{array}{c}\{L_i = v_i\ ^{i \in 1..j-1}, L_j = M_j, L_k = M_k\ ^{k \in j+1..n}\} \\ \hookrightarrow \{L_i = v_i\ ^{i \in 1..j-1}, L_j = M_j', L_k = M_k\ ^{k \in j+1..n}\}\end{array}} \; [\![\text{E-Rec}]\!]$$

$$\frac{M_j \hookrightarrow M_j'}{[v_i\ ^{i \in 1..j-1}, M_j, M_k\ ^{k \in j+1..n}] \hookrightarrow [v_i\ ^{i \in 1..j-1}, M_j', M_k\ ^{k \in j+1..n}]} \; [\![\text{E-List}]\!]$$

The operation of indexing first evaluates the collection M from which we want to retrieve the element. Then, it retrieves the element located at index *num*.

$$\frac{M \hookrightarrow M'}{M[num] \hookrightarrow M'[num]} \; [\![\text{E-Idx}]\!] \qquad \frac{}{[v_i\ ^{i \in 1..p}][num] \hookrightarrow v_{num}} \; [\![\text{E-IdxV}]\!]$$

To select a label M_2 from a term M_1, we write $M_1.M_2$. M_1 either is or contains a record. First, we evaluate both its terms M_1 and M_2 (E-Sel1, E-Sel2). Then we evaluate the selection by retrieving the term that pairs with the label M_2. Remember that nodes are evaluated at compile time and turned into runtime nodes to be later evaluated. Similarly, the selection is a compile-time operation whose evaluation result will be part of runtime elements, such as nodes. Therefore, its result is a runtime value, i.e., a boxed value.

$$\frac{M_1 \hookrightarrow M_1'}{M_1 \,.\, M_2 \hookrightarrow M_1' \,.\, M_2} \ [\![\text{E-SEL1}]\!]$$

$$\frac{M_2 \hookrightarrow M_2'}{v_1 \,.\, M_2 \hookrightarrow v_1 \,.\, M_2'} \ [\![\text{E-SEL2}]\!]$$

$$\frac{}{\{L_i = v_i{}^{i \in 1..p}\} \,.\, L_j \hookrightarrow \mathsf{Box}(v_j)} \ [\![\text{E-SELREC}]\!]$$

$$\frac{}{\mathtt{Attribute}\langle N, L, B, \{L_i = v_i{}^{i \in 1..p}\}\rangle \,.\, L_j \hookrightarrow \mathsf{Box}(v_j)} \ [\![\text{E-SELATTR}]\!]$$

$$\frac{}{\mathsf{Node}\langle \alpha, \{L_i = v_{1_i}{}^{i \in 1..p}\}, [v_{2_j}{}^{j \in 1..m}]\rangle \,.\, L_k \hookrightarrow \mathsf{Box}(v_{1k})} \ [\![\text{E-SELNODE}]\!]$$

Conditionals. The evaluation of a conditional node, represented as $\mathtt{ifNode}(M,\ M_T,\ M_F)$, starts with the evaluation of its guard M (E-IF). When the guard is the boolean literal *true*, or *false*, then only the first branch M_T, or the second branch M_F, remains after evaluation, respectively (E-IFT, E-IFF). Notice that the evaluation of each branch can only occur after evaluating the guard.

$$\frac{M \hookrightarrow M'}{\mathtt{ifNode}(M,\ M_T,\ M_F) \hookrightarrow \mathtt{ifNode}(M',\ M_T,\ M_F)} \ [\![\text{E-IF}]\!]$$

$$\frac{}{\mathtt{ifNode}(\mathit{true},\ M_T,\ M_F) \hookrightarrow M_T} \ [\![\text{E-IFT}]\!]$$

$$\frac{}{\mathtt{ifNode}(\mathit{false},\ M_T,\ M_F) \hookrightarrow M_F} \ [\![\text{E-IFF}]\!]$$

Loops. To evaluate forNodes, with the form $\mathtt{forNode}(x\colon\ t = M_1\ \mathtt{in}\ M_2)$, we first evaluate their first term M_1 (E-FOR). Once we have a loop whose first term M_1 is a list of terms, we evaluate the term M_2 under the environment where x maps to the first element of the list. Then, we append that result to the beginning of a list obtained from evaluating the forNode with the remaining terms of the list (E-ForL).

$$\frac{M_1 \hookrightarrow M_1'}{\texttt{forNode}(x\colon \tau = M_1 \ \texttt{in}\ M_2) \hookrightarrow \texttt{forNode}(x\colon \tau = M_1' \ \texttt{in}\ M_2)} \ [\![\text{E-For}]\!]$$

$$\frac{}{\begin{array}{c}\texttt{forNode}(x\colon \tau = [v_i{}^{i\in j..k}]\ \texttt{in}\ M_2)\\ \hookrightarrow ([x \mapsto v_j]M_2) :: (\texttt{forNode}(x\colon \tau = [v_i{}^{i\in j+1..k}]\ \texttt{in}\ M_2))\end{array}} \ [\![\text{E-ForV}]\!]$$

with:

$$v_0 :: [u_1, \ldots, u_n] \triangleq [v_0, u_1, \ldots, u_n]$$

The forNode for attributes will have similar rules:

$$\frac{M_1 \hookrightarrow M_1'}{\begin{array}{c}\texttt{forNode}(x\colon \texttt{AttributeT}(t)_n = M_1\ \texttt{in}\ M_2)\\ \hookrightarrow \texttt{forNode}(x\colon \texttt{AttributeT}(t)_n = M_1'\ \texttt{in}\ M_2)\end{array}} \ [\![\text{E-ForA}]\!]$$

$$\frac{}{\begin{array}{c}\texttt{forNode}(x\colon \texttt{AttributeT}(t)_n = [v_i{}^{i\in j..k}]\ \texttt{in}\ M_2)\\ \hookrightarrow ([x \mapsto v_j]M_2) :: (\texttt{forNode}(x\colon \texttt{AttributeT}(t)_n = [v_i{}^{i\in j+1..k}]\ \texttt{in}\ M_2))\end{array}} \ [\![\text{E-ForVA}]\!]$$

Runtime Computation (Delayed Computation). A boxed term, Box(M), is a runtime term present at compile time. Therefore, during compile time, it remains the same (E-Box).

$$\frac{}{\texttt{Box}(M) \hookrightarrow \texttt{Box}(M)} \ [\![\text{E-Box}]\!]$$

The letbox destructor performs the unboxing of these terms. We first evaluate its first term M_1 (E-LetB), that is ultimately reduced to a boxed term, Box(M). The letbox sentence is then removed and only the second term, M_2, remains, under an environment where u maps to the term M that was inside the Box (E-LetBV).

$$\frac{M_1 \hookrightarrow M_1'}{\texttt{letbox}\ \ u = M_1\ \ \texttt{in}\ \ M_2 \hookrightarrow \texttt{letbox}\ \ u = M_1'\ \ \texttt{in}\ \ M_2} \ [\![\text{E-LetB}]\!]$$

$$\frac{}{\texttt{letbox}\ \ u = \texttt{Box}(M)\ \ \texttt{in}\ \ M_2 \hookrightarrow [u \mapsto M]M_2} \ [\![\text{E-LetBV}]\!]$$

We now present some simple examples of how this destructor works. M_1 is a boxed term, Box(M), containing a runtime computation. The letbox sentence allows us to promote such computation to a previous stage, i.e., compile time, and maps u to it. All appearances of u in M_2 are replaced by the term M. In case M_2 is a boxed term, its evaluation ends there. In that case, the destructor allowed us to insert a runtime term inside another runtime term, avoiding nested boxes, thus maintaining a two-stage computation:

$$\begin{aligned} &\texttt{letbox}\ \ u = \texttt{Box}(\{\texttt{Value} = 3, \texttt{Price} = 10\} \hat{:} \texttt{Value})\ \ \texttt{in}\ \ \texttt{Box}(u) \\ \hookrightarrow\ &\texttt{Box}(\{\texttt{Value} = 3, \texttt{Price} = 10\} \hat{:} \texttt{Value}) \end{aligned}$$

In case M_2 is not a boxed term, the `letbox` is promoting the runtime computation to the current moment. After replacing all occurrences of u in M_2 with M, M_2 can be evaluated according to the corresponding rule. When the `letbox` is used this way, it delimits the beginning of the runtime stage:

$$\begin{aligned} &\texttt{letbox}\ \ u = \texttt{Box}(\{\texttt{Value} = 3, \texttt{Price} = 10\} \hat{:} \texttt{Value})\ \ \texttt{in}\ \ u \\ \hookrightarrow\ &\{\texttt{Value} = 3, \texttt{Price} = 10\} \hat{:} \texttt{Value} \\ \hookrightarrow\ &3 \end{aligned}$$

Polymorphism. We evaluate the application of a type variable by first evaluating their term M (E-NApp, E-TApp, E-RApp). From evaluating M, we obtain an abstraction term and evaluate its application to a variable by retrieving the abstract term (E-NAppV, E-TAppV, E-RAppV). We do not add these variables to the environment since these are type variables and are only important to the typechecking phase.

$$\frac{M \hookrightarrow M'}{M[name]_n \hookrightarrow M'[name]_n}\ [\![\text{E-NApp}]\!] \qquad \frac{}{(\Lambda_n n.M)[name]_n \hookrightarrow [name/n]M}\ [\![\text{E-NAppV}]\!]$$

$$\frac{M \hookrightarrow M'}{M[type]_t \hookrightarrow M'[type]_t}\ [\![\text{E-TApp}]\!] \qquad \frac{}{(\Lambda_t t.M)[type]_t \hookrightarrow [type/t]M}\ [\![\text{E-TAppV}]\!]$$

$$\frac{M \hookrightarrow M'}{M[row]_r \hookrightarrow M'[row]_r}\ [\![\text{E-RApp}]\!] \qquad \frac{}{(\Lambda_r r.M)[row]_r \hookrightarrow [row/r]M}\ [\![\text{E-RAppV}]\!]$$

6.5.2 Runtime Semantics

Let Sentence. The let sentence can occur both in compile time and runtime. The rules for its evaluation are the same and are defined above, in Section 6.5.1.

Nodes. All the nodevalue's subcomponents (properties v_1 and children nodes v_2) are evaluated at runtime:

$$\frac{v_1 \underset{R}{\hookrightarrow} v_1'}{\texttt{NodeValue}\langle \alpha, v_1, v_2\rangle \underset{R}{\hookrightarrow} \texttt{NodeValue}\langle \alpha, v_1', v_2\rangle} \; \llbracket\text{E-NodeV1}\rrbracket$$

$$\frac{v_2 \underset{R}{\hookrightarrow} v_2'}{\texttt{NodeValue}\langle \alpha, v_1, v_2\rangle \underset{R}{\hookrightarrow} \texttt{NodeValue}\langle \alpha, v_1, v_2'\rangle} \; \llbracket\text{E-NodeV2}\rrbracket$$

Collections. Records and lists can occur in both compile time and runtime. Their evaluation rules (E-Rec and E-List) were already defined in Section 6.5.1, and are the same in runtime. Similarly, the evaluation rules for the indexing of a list (E-Idx and E-IdxV) are the same in both computation stages.

The selection operation presented in the compile-time stage only occurs at compile time. However, selection can also happen in runtime. Hence, there is a similar operation, the runtime selection. The rules for this operation are similar to the ones in compile time. However, since this operation is only evaluated at runtime, its results are not boxed values:

$$\frac{v_1 \underset{R}{\hookrightarrow} v_1'}{v_1 \mathbin{\hat{:}} v_2 \underset{R}{\hookrightarrow} v_1' \mathbin{\hat{:}} v_2} \; \llbracket\text{E-SelRT1}\rrbracket$$

$$\frac{v_2 \underset{R}{\hookrightarrow} v_2'}{v_1 \mathbin{\hat{:}} v_2 \underset{R}{\hookrightarrow} v_1 \mathbin{\hat{:}} v_2'} \; \llbracket\text{E-SelRT2}\rrbracket$$

$$\frac{}{\{L_i = v_i{}^{i\in 1..p}\} \mathbin{\hat{:}} L_j \underset{R}{\hookrightarrow} v_j} \; \llbracket\text{E-SelRecRT}\rrbracket$$

$$\frac{}{\texttt{Attribute}\langle N, L, B, \{L_i = v_i{}^{i\in 1..p}\}\rangle \mathbin{\hat{:}} L_j \underset{R}{\hookrightarrow} v_j} \; \llbracket\text{E-SelAttrRT}\rrbracket$$

$$\frac{}{\texttt{NodeValue}\langle \alpha, \{L_i = v_{1_i}{}^{i\in 1..p}\}, [v_{2_j}{}^{j\in 1..m}]\rangle \mathbin{\hat{:}} L_k \underset{R}{\hookrightarrow} v_{1k}} \; \llbracket\text{E-SelNodeRT}\rrbracket$$

6.6 Implementation

The core language presented in this chapter, which we call Core-OSTRICH, was implemented in OCaml. Both its implementation and evaluation are available on `https://github.com/jbp182/OSTRICH-OCaml`.

To help the reader better understand the connection between the rules introduced in this chapter and the implementation, we present a portion of its typechecking algorithm implementation (Listing 6.1). It is a recursive function with six parameters:

- `e`: is the expression to be typed;
- `env`: is the compile-time typing environment, which maps compile-time variables to their types;
- `mod_env`: is the runtime typing environment, which maps compile-time variables to their types;
- `tenv`: is an environment, a set, that stores type variables;
- `nenv`: is an environment, a set, that stores name variables;
- `renv`: is an environment, a set, that stores record variables.

This algorithm matches the expression `e` with one of the patterns specified and returns its type. For example, if the expression `e` is some `String s`, then it has type `StringT`. This corresponds to the T-Str rule:

$$\frac{}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \mathit{string}\colon \mathit{String}}\;[\![\text{T-Str}]\!]$$

When presented with a let sentence with the form $\texttt{let}\ x = e1\ \texttt{in}\ e2$, the algorithm adds the type of the expression `e1` to the compile-time environment, then typing the expression `e2`. This final result will be the resulting type of the let sentence, as stated in the T-Let rule:

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1\colon \tau_1 \qquad (\Gamma, x\colon \tau_1);\Delta;\Omega;\Phi;\Upsilon \vdash M_2\colon \tau_2}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{let}\ x = M_1\ \texttt{in}\ M_2\colon \tau_2}\;[\![\text{T-Let}]\!]$$

The selection operation is a binary operation between two expressions, which can be written as $M_1 \mathbf{.} M_2$. By matching the expressions' types with three options, we get different types accordingly, which correspond to the rules T-Sel1, T-Sel2 and T-Sel3:

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1\colon \{L_i\colon \tau_i{}^{i\in 1..p}\}\\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2\colon \mathsf{Label}(L_j) \qquad j \in 1..p\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 \mathbf{.} M_2\colon \mathsf{BoxT}(\tau_j)}\;[\![\text{T-Sel1}]\!]$$

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : \mathtt{NodeT}([\alpha_i{}^{i\in 1..p}],\{p_j : \tau_j{}^{j\in 1..m}\}) \qquad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2 : \mathtt{Label}(L_k) \quad k \in 1..m}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 \,.\, M_2 : \mathtt{BoxT}(\tau_k)} \; [\![\text{T-SEL2}]\!]$$

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : \mathtt{AttributeT}(B)_N \qquad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2 : \mathtt{Label}(\mathtt{DisplayName})}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 \,.\, M_2 : \mathtt{BoxT}(String)} \; [\![\text{T-SEL3}]\!]$$

Listing 6.1: Part of the implementation of the typechecking algorithm.

```
let rec typecheck e env mod_env tenv nenv renv =
    match e with
    | String s -> StringT
    | Let(x,e1,e2) ->
        let t1 = typecheck e1 env mod_env tenv nenv renv in
        let env' = define x t1 env in
        let t2 = typecheck e2 env' mod_env tenv nenv renv in
        t2
    | Select(e1, e2) ->
        let t1 = typecheck e1 env mod_env tenv nenv renv in
        let t2 = typecheck e2 env mod_env tenv nenv renv in
        begin
        match t1, t2 with
        | RecordT r , LabelT l -> BoxT(find l r)
        | NodeT (al, RecordT pr) , LabelT l -> BoxT(find l pr)
        | AttributeT(n, t) , LabelT "DisplayName" -> BoxT(StringT)
        | _ -> error (BadSelectOp(t1, t2))
        end
    | (...)
```

We also implement an algorithm for semantic evaluation, which follows the rules presented in Section 6.5. That algorithm is implemented by means of a recursive function. But this time, it only receives two parameters: the expression e and an environment env, and returns a simplified expression. The correspondence between the evaluation rules and their implementation is similar to the one presented with the typechecking algorithm. Therefore, for simplicity, we opted not to list its implementation here.

7 Evaluation

Currently, the OutSystems platform contains 70 pre-built screens. The OSTRICH language allows for the definition, verification and instantiation OutSystems' templates, having chosen to implement their top 10 [20]. Such templates correspond to more than half of the screen template instantiations happening on the platform. By extending the OSTRICH language with nested templates, i.e. the instantiation of a template inside the definition of another, we can create additional templates that embody common patterns to the original 10 templates. Seven shared patterns were identified, spawning seven additional templates [44], as shown in Figure 7.1.

We offer a reference implementation of the OSTRICH language, which aims to help in the implementation of its extensions. To prove our solution's ability to represent OSTRICH's behaviour, we verify if it covers all cases covered by OSTRICH, i.e., the 10+7 templates. These templates require a way of defining dependencies between the parameters' types. For example, if a template receives an entity and an attribute, we might want to further restrict those parameters and ensure that the attribute belongs to the specified entity. Additionally, all the main templates contain inner templates, and we need to ensure that their instantiation is safe. The code for all the 10+7 templates used in the evaluation of our solution is presented in Annex III.

Next, we present an example of one of these templates. Since some templates have similar features, we chose one that allows for clearer examples and still covers all the features. We start with the syntax of its definition (Section 7.1), followed by the result of typechecking it (Section 7.2). Then, we present its expected compile-time and runtime results (Section 7.3). We do not go into much detail in this chapter, since all the typing rules and evaluation rules were already presented in Chapter 6.

Since we cover all the important features with this example, it is easy to understand that our reference implementation already implements, verifies and executes all of the templates. Since our implementation is only a prototype, some runtime behaviour might not be implemented (such as filtering list entries according to the user's input), as we directed our focus to the template structure and the dependencies between parameters. We believe these behaviours are easy to add and are mainly a matter of increasing the

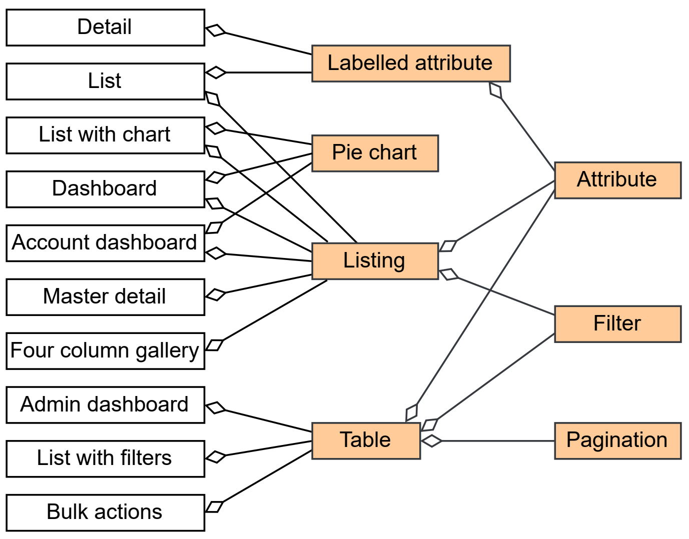


Figure 7.1: Screen templates and their inner templates (in [44]).

syntax for these instances.

We use diagrams to better illustrate the typing of each element. However, we must warn the reader that our diagram notation has no metamodel and is based on the language's syntax defined in Chapter 6. We are merely trying to present the abstract syntax tree neatly.

7.1 Template Definition

We illustrate the typechecking and execution of a template, the `Detail` template, that displays detailed information about an entity's instance. For example, consider a screen that lists products sold by some store. If the application user selects one of the rows of that list, a product named `chocolate frog`, for instance, then another screen shows up. The new screen displays detailed information about that specific instance, such as stock quantity or description - such a screen is the `Detail` screen.

We define a template with two columns. One of the columns emphasizes the main information of the selected instance, the primary attributes. The other one displays less-important information, the secondary attributes. We present the definition of this template in Figure 7.2. The developer of the application must define both the primary and secondary attributes. Therefore, the template receives the entity of the selected instance and two lists of attributes. In each column, the template iterates through one of the lists. Then, for each element iterated, it displays a label with the name of the attribute it is presenting and the value of that attribute.

This template contains shared patterns with other templates. Hence, we defined two other inner templates that can be reused and are used and defined here.

We present the first inner template, `T1`, in Figure 7.3. Its parameters are an entity and an attribute of that entity. It defines the type of widget to be displayed according to the

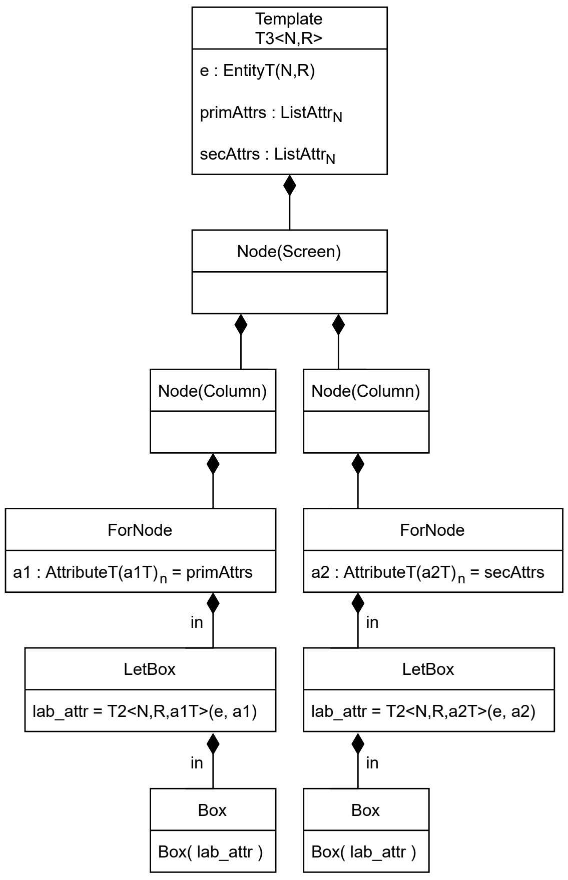


Figure 7.2: Schematic diagram of the definition of the template `T3 - Template Detail`.

attribute's type. If an attribute has boolean values, then it should appear as a `CheckBox`, which is ticked when its value is *true*. Otherwise, we employ an `Expression` widget that displays the actual value of the attribute.

We depict the definition of the second template, `T2`, in Figure 7.4. It receives an entity and an attribute of that entity as parameters. Then, it instantiates the template `T1` to display the attribute's value and adds a label with the attribute's name.

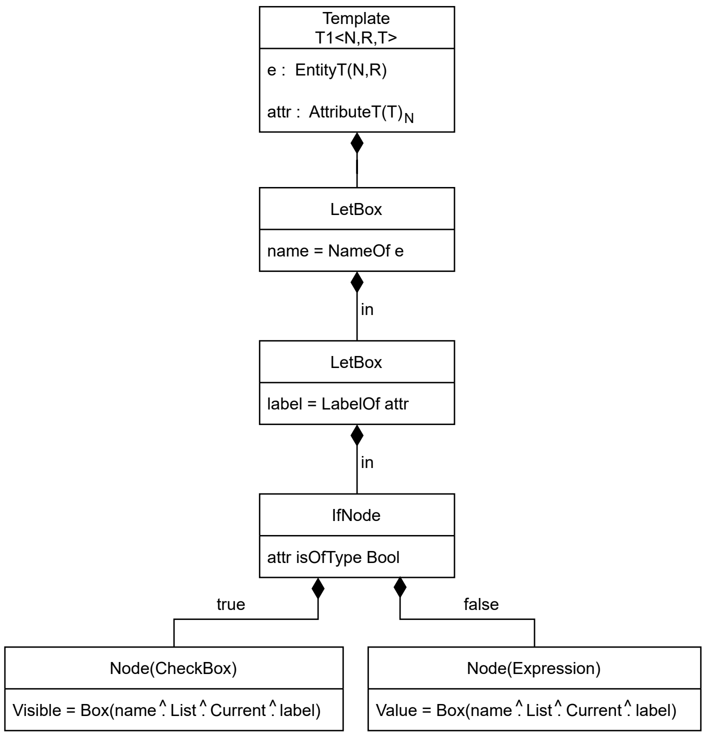


Figure 7.3: Schematic diagram of the definition of the template `T1 - Template Attribute`.

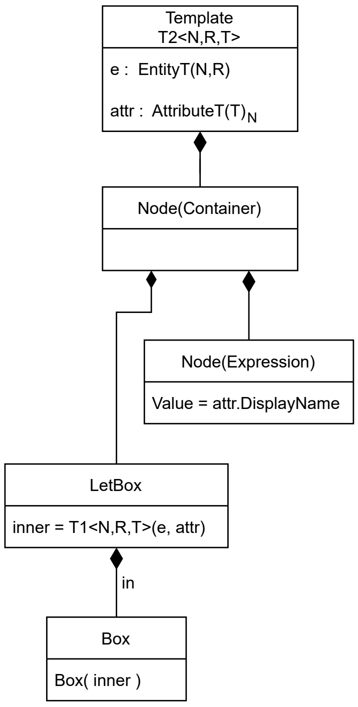


Figure 7.4: Schematic diagram of the definition of the template `T2 - Template Labelled Attribute`.

7.2 Template Typechecking

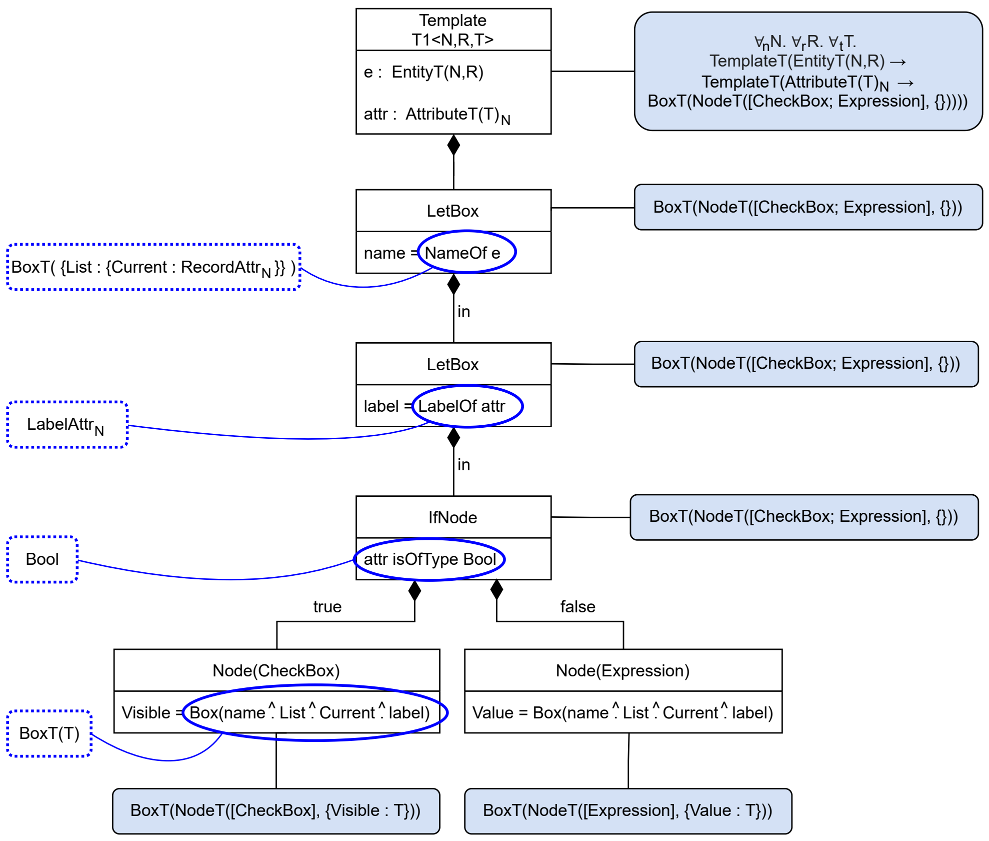

Template
T1<N,R,T>
e : EntityT(N,R)
attr : AttributeT(T)N
∀nN. ∀rR. ∀tT. TemplateT(EntityT(N,R) → TemplateT(AttributeT(T)N → BoxT(NodeT([CheckBox; Expression], {}))))
LetBox
name = NameOf e
BoxT(NodeT([CheckBox; Expression], {}))
BoxT({List : {Current : RecordAttrN }})
in
LetBox
label = LabelOf attr
BoxT(NodeT([CheckBox; Expression], {}))
LabelAttrN
in
IfNode
attr isOfType Bool
BoxT(NodeT([CheckBox; Expression], {}))
Bool
true
false
Node(CheckBox)
Visible = Box(name ^. List ^. Current ^. label)
Node(Expression)
Value = Box(name ^. List ^. Current ^. label)
BoxT(T)
BoxT(NodeT([CheckBox], {Visible : T}))
BoxT(NodeT([Expression], {Value : T}))

Figure 7.5: Schematic diagram of the typechecking of the template T1.

The template language is a two-stage language. During compile time, the typechecking algorithm executes, assigning types to the template's elements. We represent the type of each element inside a blue box, and the type of expressions inside dashed boxes.

In Figure 7.5, we present the result of typing the first inner template. The boxed expressions will have type `Box(T)`, with `T` being the type of the attribute's values, as defined in the first node by the statement `attr: AttributeT`$(T)_N$. This comes from rules `T-SelRT1` and `T-SelRT3`. The `CheckBox` node will have the boxed type of a node with a list containing its category, and a record with the types of its properties. Since the resulting node is a runtime node, any inner boxed types are disposed of their boxes, and the whole node is surrounded with an outer box (`T-Node`). The same happens with the `Expression` node. The type of the `IfNode` will depend on the type of its children nodes (`T-If`). Its type is a boxed node, containing the categories of its children, `CheckBox` and `Expression`, and a record with common properties' types, which in this case do not exist. `Letbox` expressions will have the type of their body (`T-LetB`), which is the boxed node type.

The definition of the quantifiers and parameters of each template is condensed into a single element, the first one. In Figure 7.6, we expand the condensed element in `T1` and present the types of each original term. For simplicity, we preserve the condensed

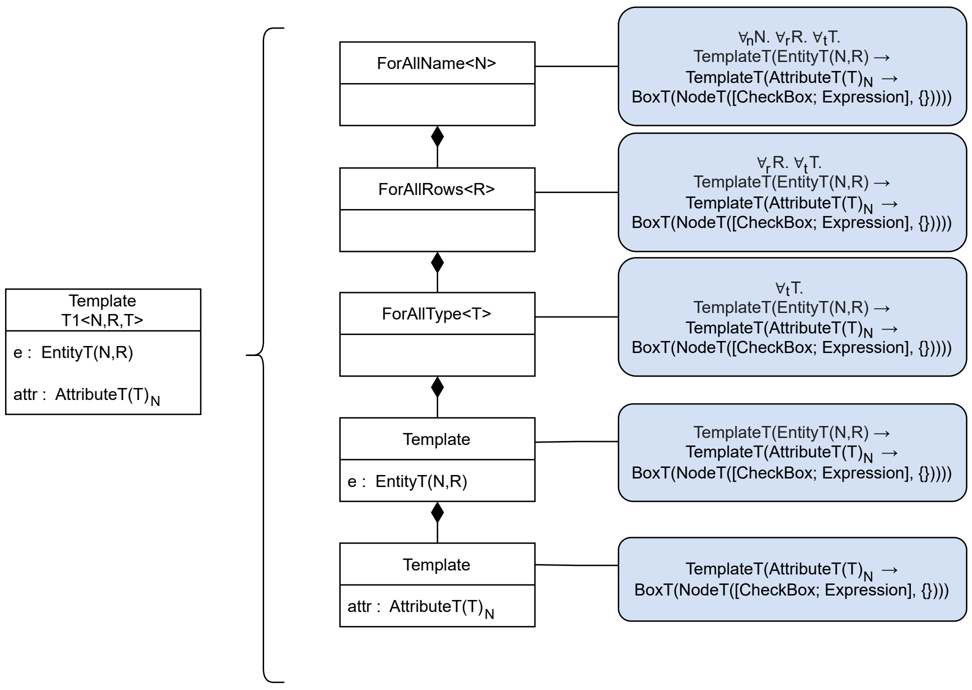

Template
T1<N,R,T>
e : EntityT(N,R)
attr : AttributeT(T)_N
ForAllName<N>
ForAllRows<R>
ForAllType<T>
Template
e : EntityT(N,R)
Template
attr : AttributeT(T)_N
∀nN. ∀rR. ∀tT. TemplateT(EntityT(N,R) → TemplateT(AttributeT(T)_N → BoxT(NodeT([CheckBox; Expression], {}))))
∀rR. ∀tT. TemplateT(EntityT(N,R) → TemplateT(AttributeT(T)_N → BoxT(NodeT([CheckBox; Expression], {}))))
∀tT. TemplateT(EntityT(N,R) → TemplateT(AttributeT(T)_N → BoxT(NodeT([CheckBox; Expression], {}))))
TemplateT(EntityT(N,R) → TemplateT(AttributeT(T)_N → BoxT(NodeT([CheckBox; Expression], {}))))
TemplateT(AttributeT(T)_N → BoxT(NodeT([CheckBox; Expression], {})))

Figure 7.6: Typechecking of the template T1's signature, expanded.

format in the next templates. Since our implementation only accepts one parameter at a time, we need to define successive templates to allow multiple parameters. Each template definition will have type TemplateT($\tau_1 \to \tau_2$), where τ_1 is the parameter's type, and τ_2 is the type of the template's body (T-Temp). In this case, the last node in Figure 7.6 will have the type of a template, with τ_1 being the type of attr, and τ_2 the boxed node type explained above. The next parameter definition will have τ_1 as the type of e, which is an entity, and τ_2 as the type of the term below. Type variable definitions work similarly. The definition of a variable for types has the type $\forall_t T.\tau$, where T is the chosen variable, and τ is the type of its body (T-FAT).

In Figure 7.7, we present the types of the second template's elements. The compile-time expression in the Expression node has type BoxT(*String*) (T-Sel3). The letbox expression assigns to inner the result of instantiating T1. The instantiation with all type variables and arguments will have the type BoxT(NodeT([CheckBox;Expression],{})) (T-Inst).

Finally, in Figure 7.8, we present the types of the outer template's elements. The instantiation of T2 will have the type BoxT(NodeT(Container,{})) (T-Inst). The type of the ForNode will be the type of its body (T-For3).

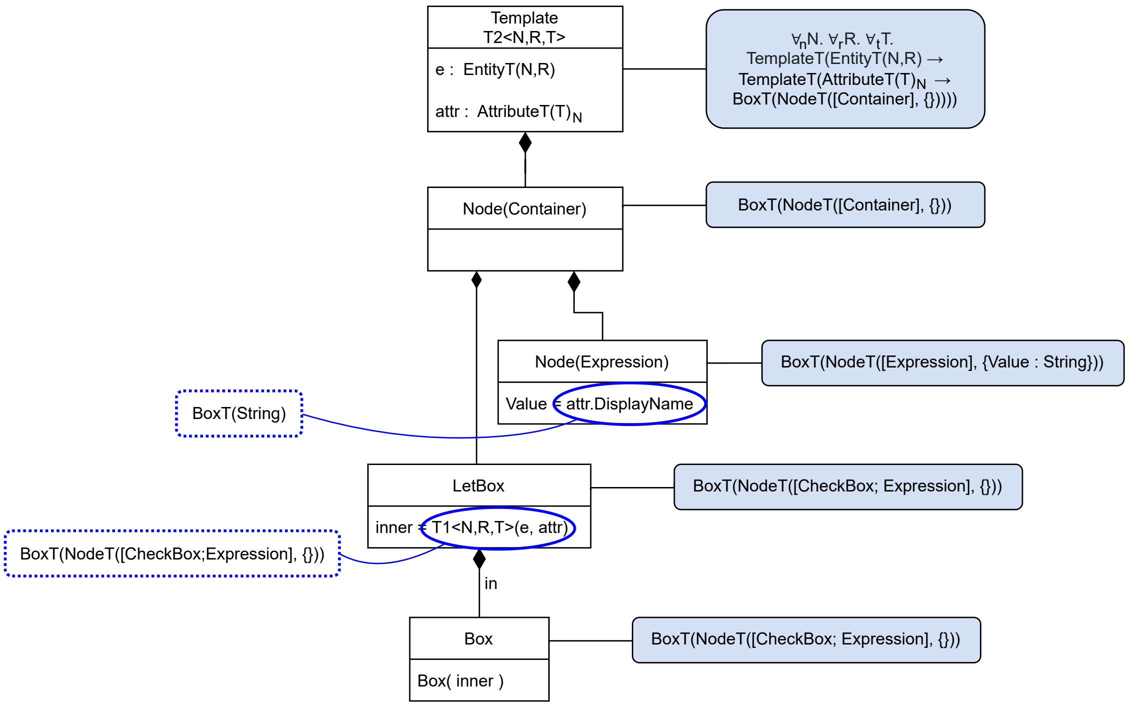

Template
T2<N,R,T>
e : EntityT(N,R)
attr : AttributeT(T)N
∀nN. ∀rR. ∀tT.
TemplateT(EntityT(N,R) →
TemplateT(AttributeT(T)N →
BoxT(NodeT([Container], {}))))
Node(Container)
BoxT(NodeT([Container], {}))
Node(Expression)
Value = attr.DisplayName
BoxT(NodeT([Expression], {Value : String}))
BoxT(String)
LetBox
inner = T1<N,R,T>(e, attr)
BoxT(NodeT([CheckBox; Expression], {}))
BoxT(NodeT([CheckBox;Expression], {}))
in
Box
Box(inner)
BoxT(NodeT([CheckBox; Expression], {}))

Figure 7.7: Schematic diagram of the typechecking of the template T2.

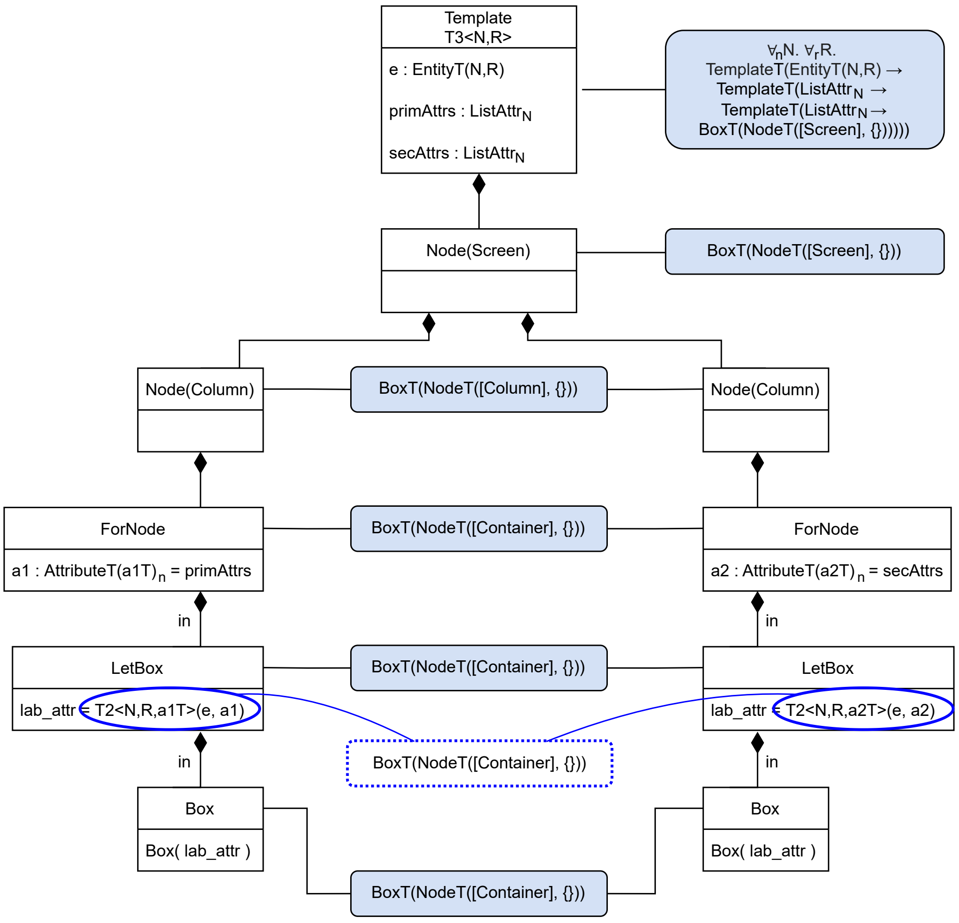

Template
T3<N,R>
e : EntityT(N,R)
primAttrs : ListAttrN
secAttrs : ListAttrN
∀nN. ∀rR.
TemplateT(EntityT(N,R) →
TemplateT(ListAttrN →
TemplateT(ListAttrN →
BoxT(NodeT([Screen], {}))))
Node(Screen)
BoxT(NodeT([Screen], {}))
Node(Column)
BoxT(NodeT([Column], {}))
Node(Column)
ForNode
a1 : AttributeT(a1T)n = primAttrs
BoxT(NodeT([Container], {}))
ForNode
a2 : AttributeT(a2T)n = secAttrs
in
LetBox
lab_attr = T2<N,R,a1T>(e, a1)
BoxT(NodeT([Container], {}))
LetBox
lab_attr = T2<N,R,a2T>(e, a2)
in
BoxT(NodeT([Container], {}))
Box
Box(lab_attr)
BoxT(NodeT([Container], {}))
Box
Box(lab_attr)

Figure 7.8: Schematic diagram of the typechecking of the template T3.

7.3 Template Execution

7.3.1 Compile Time Execution

During compile time and after typechecking, the template is instantiated with the arguments provided by the developer of the application. Some compile-time computation occurs, replacing expressions for their instantiated executable versions, and creating nodes according to the conditions inside `ifNodes` and the iteration of `forNodes`.

We present the result of instantiating template `T3` in Figure 7.9, with an entity `Product`. To simplify our example and the corresponding diagram, we instantiate it with a list with two primary attributes: `Name` and `IsInStock`, and a list with a single secondary attribute: `Price`. The result of the instantiation starts with a `Box` due to being ready to execute once the runtime computations start. Loops and conditionals are evaluated and replaced by their corresponding nodes. Inner templates are instantiated. As seen in Figure 7.9, the result is a screen with two columns. One displays information about two main attributes, and the other displays just one. The expressions that fetch the actual value to be displayed remain unevaluated. However, such expressions now refer to the arguments applied in instantiation.

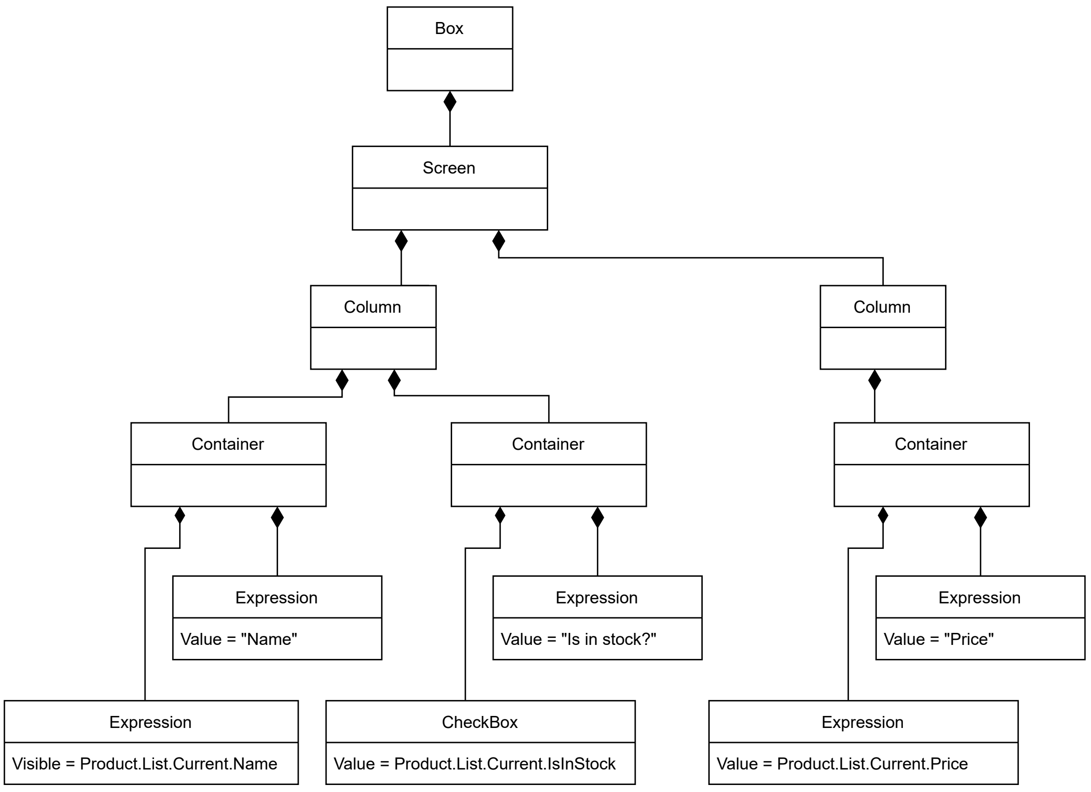
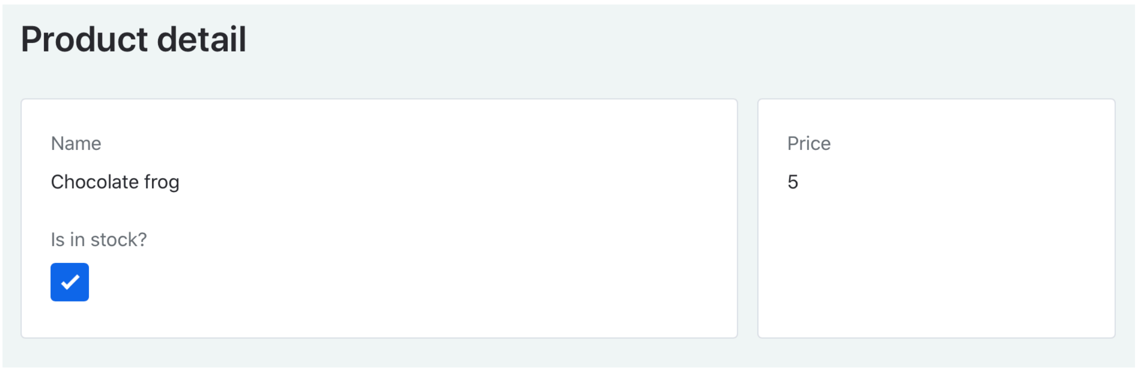


Figure 7.9: Schematic diagram of the template `T3`'s compile-time execution. In this example, the template is instantiated with 1) an entity `Product`, 2) a list containing two of its attributes: `Name` and `IsInStock`, and 3) a list containing another of its attributes: `Quantity`.

7.3.2 Runtime Execution

The result presented in Figure 7.9 is now executed during runtime, and its expressions are evaluated to their actual values. The resulting application in OutSystems would appear as presented in Figure 7.10. The displayed attributes' values were arbitrarily chosen.


Product detail

Name

Chocolate frog

Is in stock?

✓

Price

5

Figure 7.10: Corresponding resulting screen of the OutSystems' application after runtime execution of the template.

8 Conclusion

Through the course of this thesis, we worked in several stages of the GOLEM Project. This dissertation presents several contributions to the project.

First, we designed an ontology [45, 46] to help the natural-language processing-component capture concepts that can be later translated into application elements. The choice of appropriate concepts was a crucial step because such concepts need to resemble the ones usually adopted by non-expert developers to facilitate mapping. Additionally, the concepts need to be representative enough so we can understand them and generate an application based on the intent expressed by the user.

Since some user requests are common patterns, using reusable pre-assembled templates would be beneficial and result in a cleaner and more appealing user interface. Unfortunately, the OutSystems templates use pre-defined data, which the user must adapt to his own data source. To address this issue, we enriched the templates' behaviour [41, 43, 44] to ease the development process. For this, we rely on the OSTRICH language, a type-safe template language for the OutSystems platform that allows the definition and instantiation of templates that can be applied and reused by the DSL developed under the scope of the GOLEM project. We formalized this multi-stage language, OSTRICH, paired with some extensions, such as name and type abstractions and dependencies between type declarations. These extensions help create more OutSystems templates with increasing variety and possibilities, easing the application development process in this low-code platform. We also produced a reference implementation of OSTRICH, which captures all of the main features present in the instances covered by OSTRICH. This makes it a viable option to study possible extensions for OSTRICH.

There is still plenty of room for improvement in future work by extending the restrictions and dependencies between parameters to allow for the instantiation of more complex templates. Additionally, the developer might want to customise a screen after instantiating its template. However, if the original template suffers an update, reapplying the new one might cause some conflicts to emerge. Keeping a log of the customisation progress allows instantiating the newly updated template and reapplying such customisations on the new instantiation, unless they do not produce type-safe results [42].

This broad work allowed us to interact and learn with great professionals, from MSc to PhD students to seniors of natural language processing and programming languages.

Bibliography

[1] M. Bačíková, J. Porubän, and D. Lakatos. “Defining Domain Language of Graphical User Interfaces”. In: *OpenAccess Series in Informatics* 29 (2013-01), pp. 187–202. DOI: 10.4230/OASIcs.SLATE.2013.187 (cit. on p. 19).

[2] D. Berardi, D. Calvanese, and G. De Giacomo. “Reasoning on UML class diagrams”. In: *Artificial Intelligence* 168.1 (2005), pp. 70–118. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2005.05.003. URL: https://www.sciencedirect.com/science/article/pii/S0004370205000792 (cit. on pp. 8, 10).

[3] D. Calvanese. *Description Logics for Conceptual Modeling*. 2012. URL: http://www.epcl-study.eu/content/downloads/slides/calvanese-DLs-conceptual-modeling.pdf (cit. on pp. 8–10).

[4] D. Calvanese. *Ontology and Database Systems: Knowledge Representation and Ontologies. Part 2: Description Logics*. 2017. URL: https://www.inf.unibz.it/~calvanese/teaching/17-18-odbs/lecture-notes/KRO-2-dls.pdf (cit. on pp. 6–8, 11, 12, 36).

[5] L. Cardelli. “Phase Distinctions in Type Theory”. https://www.microsoft.com/en-us/research/publication/phase-distinctions-in-type-theory/. 1988 (cit. on pp. 21, 22, 27, 44).

[6] L. Cardelli. “Type Systems”. In: *ACM Comput. Surv.* 28.1 (1996), 263–264. ISSN: 0360-0300. DOI: 10.1145/234313.234418. URL: https://doi.org/10.1145/234313.234418.

[7] L. Cardelli and P. Wegner. “On Understanding Types, Data Abstraction, and Polymorphism”. In: *ACM Comput. Surv.* 17.4 (1985), 471–523. ISSN: 0360-0300. DOI: 10.1145/6041.6042. URL: https://doi.org/10.1145/6041.6042.

[8] R. Davies and F. Pfenning. “A Modal Analysis of Staged Computation”. In: *J. ACM* 48.3 (2001-05), pp. 555–604. DOI: 10.1145/382780.382785. URL: https://doi.org/10.1145/382780.382785 (cit. on pp. 21, 25–27, 44, 46).

[9] S. P. De Rosso, D. Jackson, M. Archie, C. Lao, and B. A. McNamara. "Declarative assembly of web applications from predefined concepts". In: *Onward! 2019 - Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, co-located with SPLASH 2019* (2019), pp. 79–93. DOI: 10.1145/3359591.3359728 (cit. on pp. 17–19).

[10] M. Fahad. "ER2OWL: Generating OWL ontology from ER diagram". In: *IFIP International Federation for Information Processing* 288 (2008), pp. 28–37. ISSN: 1571-5736. DOI: 10.1007/978-0-387-87685-6_6 (cit. on p. 8).

[11] J. Fonseca, M. Pereira, and P. Rangel Henriques. "Converting Ontologies into DSLs". In: *OpenAccess Series in Informatics* 38 (2014-01), pp. 85–92. DOI: 10.4230/OASIcs.SLATE.2014.85 (cit. on p. 19).

[12] L. Globa, R. Novogrudska, A. Koval, and V. Senchenko. "Ontology for Application Development". In: *Ontology in Information Science*. 2018-03. ISBN: 978-953-51-3887-7. DOI: 10.5772/intechopen.74042 (cit. on pp. 5–7).

[13] M. Hirzel. *Low-Code Programming Models*. 2022-05 (cit. on p. 20).

[14] M. Horridge, H. Knublauch, A. Rector, R. Stevens, and C. Wroe. *A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools Edition 1.3*. 2011-03 (cit. on pp. 5, 11, 13–15, 30).

[15] D. Jackson. "Towards a Theory of Conceptual Design for Software". In: *Onward! 2015 - Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Part of SPLASH 2015* (2015), pp. 282–296. DOI: 10.1145/2814228.2814248 (cit. on pp. 16–18).

[16] R. Kontchakov and M. Zakharyaschev. "An Introduction to Description Logics and Query Rewriting". In: *Reasoning Web. Reasoning on the Web in the Big Data Era*. Springer, Cham, 2014-09, pp. 195–244. DOI: 10.1007/978-3-319-10587-1_5 (cit. on p. 11).

[17] J. de Lara and E. Guerra. "From Types to Type Requirements: Genericity for Model-Driven Engineering". In: *Softw. Syst. Model.* 12.3 (2013-07), pp. 453–474. DOI: 10.1007/s10270-011-0221-0. URL: https://doi.org/10.1007/s10270-011-0221-0 (cit. on p. 27).

[18] Y. Lilis and A. Savidis. "A Survey of Metaprogramming Languages". In: *ACM Comput. Surv.* 52.6 (2019). ISSN: 0360-0300. DOI: 10.1145/3354584. URL: https://doi.org/10.1145/3354584 (cit. on p. 20).

[19] A. López-Cima, Ó. Corcho, and A. Gómez-Pérez. "Rapid Ontology-based Web Application Development with JSTL". In: *Proceedings of the ESWC'07 Workshop on Scripting for the Semantic Web, SFSW 2007, Innsbruck, Austria, May 30, 2007*. Ed. by S. Auer, C. Bizer, T. Heath, and G. A. Grimnes. Vol. 248. CEUR Workshop

Proceedings. CEUR-WS.org, 2007. URL: http://ceur-ws.org/Vol-248/paper5.pdf (cit. on p. 19).

[20] H. Lourenço, C. Ferreira, and J. C. Seco. "OSTRICH - A Type-Safe Template Language for Low-Code Development". In: *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2021, pp. 216–226. DOI: 10.1109/MODELS50736.2021.00030 (cit. on pp. 2, 30, 32–34, 43, 44, 73).

[21] J. A. McCance. "A Brief Introduction to Modal Logic". In: 2008 (cit. on p. 25).

[22] E. Moggi, W. Taha, Z. E. Benaissa, and T. Sheard. *An Idealized MetaML: Simpler, and More Expressive (Includes Proofs)*. Tech. rep. 1998 (cit. on pp. 21, 23, 24, 27).

[23] D. Nardi and R. J. Brachman. "An Introduction to Description Logics". In: *The Description Logic Handbook: Theory, Implementation, and Applications*. Ed. by F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider. Cambridge University Press, 2003, pp. 1–40 (cit. on pp. 5, 11, 12).

[24] M. J. O'Connor, R. D. Shankar, C. Nyulas, S. W. Tu, and A. K. Das. "Developing a Web-Based Application using OWL and SWRL". In: *AI Meets Business Rules and Process Management, Papers from the 2008 AAAI Spring Symposium, Technical Report SS-08-01, Stanford, California, USA, March 26-28, 2008*. AAAI, 2008, pp. 93–98. URL: http://www.aaai.org/Library/Symposia/Spring/2008/ss08-01-012.php (cit. on p. 19).

[25] F. Pfenning and R. Davies. "A Judgmental Reconstruction of Modal Logic". In: *Mathematical Structures in Computer Science* 11.4 (2001), pp. 511–540. DOI: 10.1017/S0960129501003322. URL: https://doi.org/10.1017/S0960129501003322 (cit. on pp. 25–27).

[26] B. C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091.

[27] D. D. Ruscio, D. S. Kolovos, J. de Lara, A. Pierantonio, M. Tisi, and M. Wimmer. "Low-code development and model-driven engineering: Two sides of the same coin?" In: *Softw. Syst. Model.* 21.2 (2022), pp. 437–446. DOI: 10.1007/s10270-021-00970-2 (cit. on p. 20).

[28] T. Sheard and S. P. Jones. "Template Meta-Programming for Haskell". In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell '02. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2002, 1–16. ISBN: 1581136056. DOI: 10.1145/581690.581691. URL: https://doi.org/10.1145/581690.581691 (cit. on pp. 22, 27).

[29] W. Taha and T. Sheard. "Multi-Stage Programming with Explicit Annotations". In: *SIGPLAN Not.* 32.12 (1997), 203–217. ISSN: 0362-1340. DOI: 10.1145/258994.259019. URL: https://doi.org/10.1145/258994.259019.

[30] W. Taha and T. Sheard. *MetaML and Multi-Stage Programming with Explicit Annotations*. Tech. rep. 1999 (cit. on pp. 21, 23, 24, 27).

[31] T. Walter, F. S. Parreiras, and S. Staab. "*OntoDSL*: An Ontology-Based Framework for Domain-Specific Languages". In: *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings*. Ed. by A. Schürr and B. Selic. Vol. 5795. Lecture Notes in Computer Science. Springer, 2009, pp. 408–422. DOI: 10.1007/978-3-642-04425-0_32. URL: https://doi.org/10.1007/978-3-642-04425-0_32 (cit. on p. 19).

[32] M. Woo. "The Rise of No/Low Code Software Development—No Experience Needed?" In: *Engineering* 6 (2020-07). DOI: 10.1016/j.eng.2020.07.007 (cit. on p. 1).

[33] I. Čeh, M. Črepinšek, T. Kosar, and M. Mernik. "Ontology Driven Development of Domain-Specific Languages". In: *Computer Science and Information Systems* 8 (2011-05), pp. 317–342. DOI: 10.2298/CSIS101231019C (cit. on p. 19).

Webography

[34] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[35] D. Midura. *The Rise of Low Code: 2020 and Beyond*. 2019-12-02. URL: https://www.techadv.com/blog/rise-low-code-2020-and-beyond (cit. on pp. 1, 2).

[36] H. T. L. de Paula. *A Brief Introduction to Template Haskell*. 2021-07-06. URL: https://serokell.io/blog/introduction-to-template-haskell (cit. on pp. 22, 27).

[37] *Protégé*. URL: https://protege.stanford.edu (cit. on p. 13).

[38] *Protégé 5 Documentation*. URL: http://protegeproject.github.io/protege/ (cit. on p. 14).

[39] *The History of Low-Code Platforms: How Development Changed Forever*. 2018-05-02. URL: https://kissflow.com/rad/low-code/history-of-low-code-development-platforms/ (cit. on p. 1).

[40] D. Wilfrid. *A Brief History of Low-Code Development Platforms*. 2020-02-10. URL: https://www.quickbase.com/blog/a-brief-history-of-low-code-development-platforms (cit. on p. 1).

Contributed Papers

[41] H. Lourenço, C. Ferreira, J. C. Seco, and J. Parreira. "OSTRICH - A Rich Template Language for Low-Code Development (Extended version)". In: *Software and Systems Modeling (SoSyM)* (2022). (accepted, under revision) (cit. on pp. 3, 31–34, 82).

[42] C. Manteigas, J. Parreira, J. C. Seco, and C. Ferreira. "Type-Safe Customization of Low-code Templates". In: *INForum 2022 - Simpósio de Informática*. 2022. (to appear) (cit. on pp. 3, 4, 82).

[43] J. Parreira. "Simple Dependent Types for OSTRICH". In: *Integrated Formal Methods, PhD Symposium at the 17th International Conference on integrated Formal Methods (PhD-iFM'22)*. Vol. 13274. Lecture Notes in Computer Science. Lugano, Switzerland: Springer, 2022 (cit. on pp. 3, 4, 31, 82).

[44] J. C. Seco, H. Lourenço, J. Parreira, and C. Ferreira. "Nested OSTRICH - Hatching Compositions of Low-code Templates". In: *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2022. (to appear) (cit. on pp. 3, 4, 31, 35, 46, 73, 74, 82).

[45] J. Silva, D. Melo, I. Rodrigues, J. Seco, C. Ferreira, and J. Parreira. "An Ontology based Task Oriented Dialogue". In: *Proceedings of the 13th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management - KEOD*, INSTICC. SciTePress, 2021, pp. 96–107. ISBN: 978-989-758-533-3. DOI: 10.5220/0010711900003064 (cit. on pp. 3, 30, 37, 42, 82).

[46] J. Silva, D. Melo, I. Rodrigues, J. Seco, C. Ferreira, and J. Parreira. "An Ontology based Task Oriented Dialogue to Create OutSystems Applications". In: *SN Computer Science* (2022). (under submission) (cit. on pp. 3, 30, 37, 82).

I

Annex I - Ontology

I.1 Ontology Description

This ontology, written in description logic $SROIQ^{(D)}$, represents part of a website with lost, found, and adoption announcements for pets. It has a page *menu* and information about *menu location* on the page. It also has *menu items* which are not specified in this example. Each menu item will have a name, like for example *Home*, *Account*, *Lost*, among others. And each menu item will have a *position* attribute corresponding to its relative position on the menu. It also has *users*, *posts*, *post items*, *comments*, and *reports*.

In Section I.2 we show portions of the UML class diagram used to build the first ontology. Section I.3 describes the first ontology with Description Logics ($SROIQ^{(D)}$). In Section I.3.1 we can see all classes and their subclasses. In Section I.3.2 we can see the object properties (relations or roles) between those classes and corresponding cardinalities. Data properties (attributes) and information about their domains are specified in Section I.3.3. Section I.3.4 contains some examples of queries in order to retrieve specific individuals (instances) according to what is specified on said query. We present the second ontology in Section I.4.

Because we describe the ontology in $SROIQ^{(D)}$, we assumed that atoms of type string and datatype, and patterns exist.

I.2 UML Class Diagram

In this Section we will be presenting several UML class diagrams. All of them form a larger diagram, which was not possible to present in just one figure. These UML class diagrams represent a portion of our metamodel (the ontology), and the ontology written in Description Logics was based on it.

First, we present the high-level classes, *Entity*, *Attribute*, and *Aggregate*, in Figure I.1.

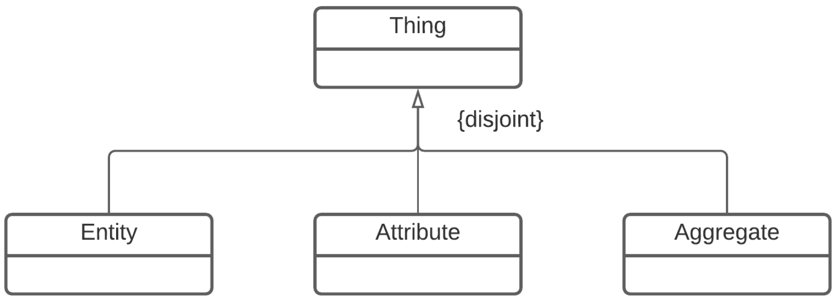


Figure I.1: UML class diagram with three of the main classes (high-level) of our metamodel.

Figure I.2 shows all the subclasses of the *Entity* class.

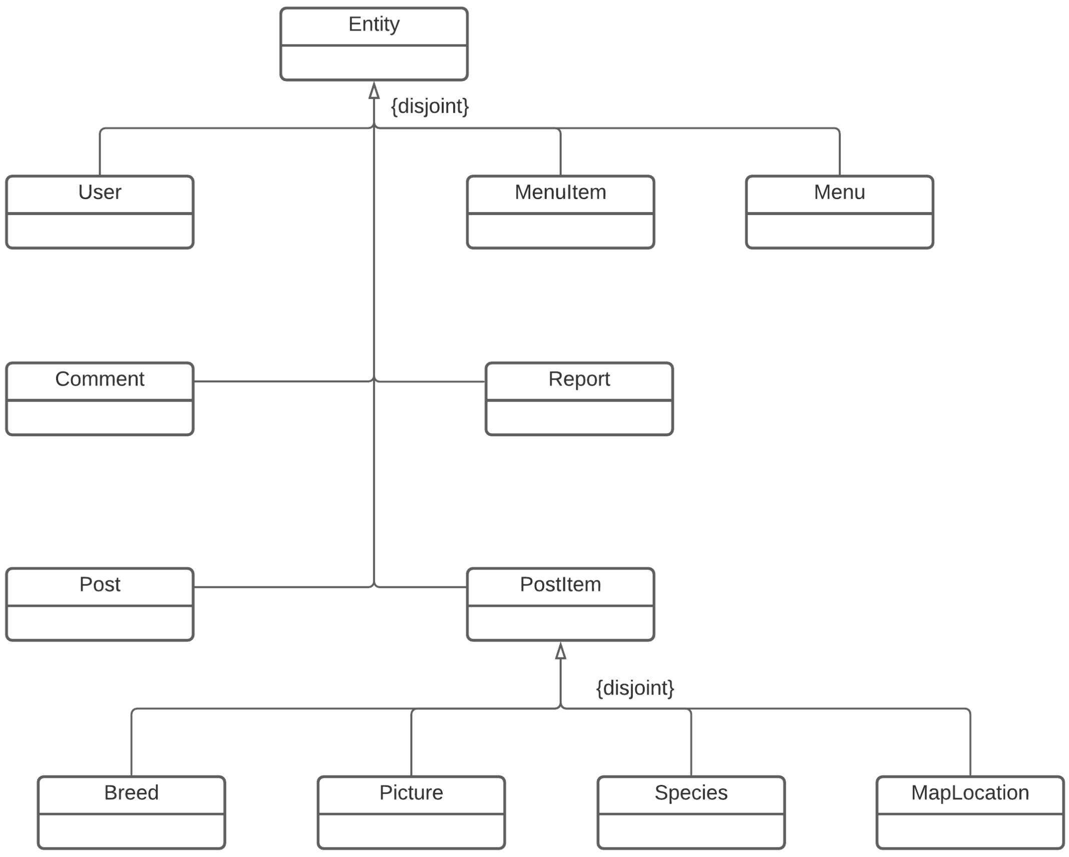


Figure I.2: UML class diagram showing the subclasses of the Entity class.

Figure I.3 shows the attributes of the class *Attribute* (characteristics, such as *label*, *name*, *default_value*, and *values*), and all its subclasses.

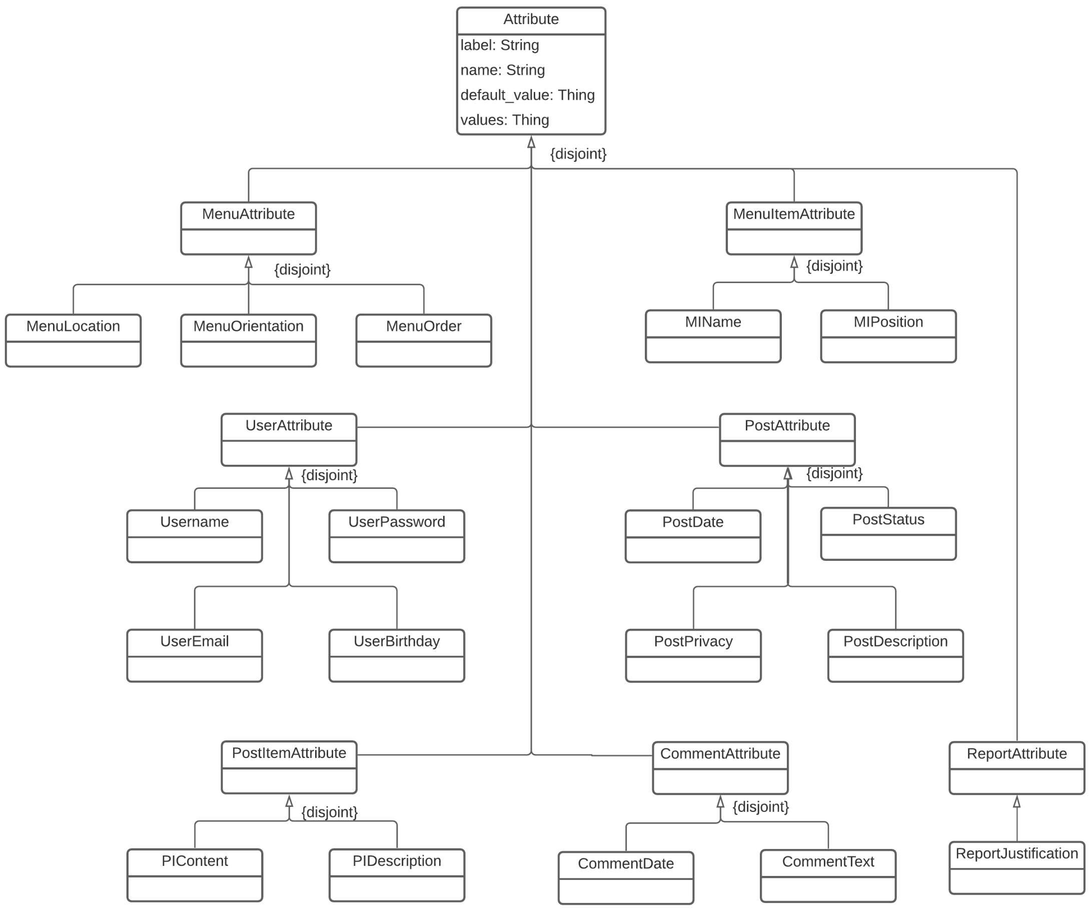


Figure I.3: UML class diagram showing the subclasses of the Attribute class.

Figure I.4 shows some of the associations between subclasses of *Entity*. Two of these associations, *comments* and *reports*, have an association class, hence reification was applied while translating them into Description Logics.

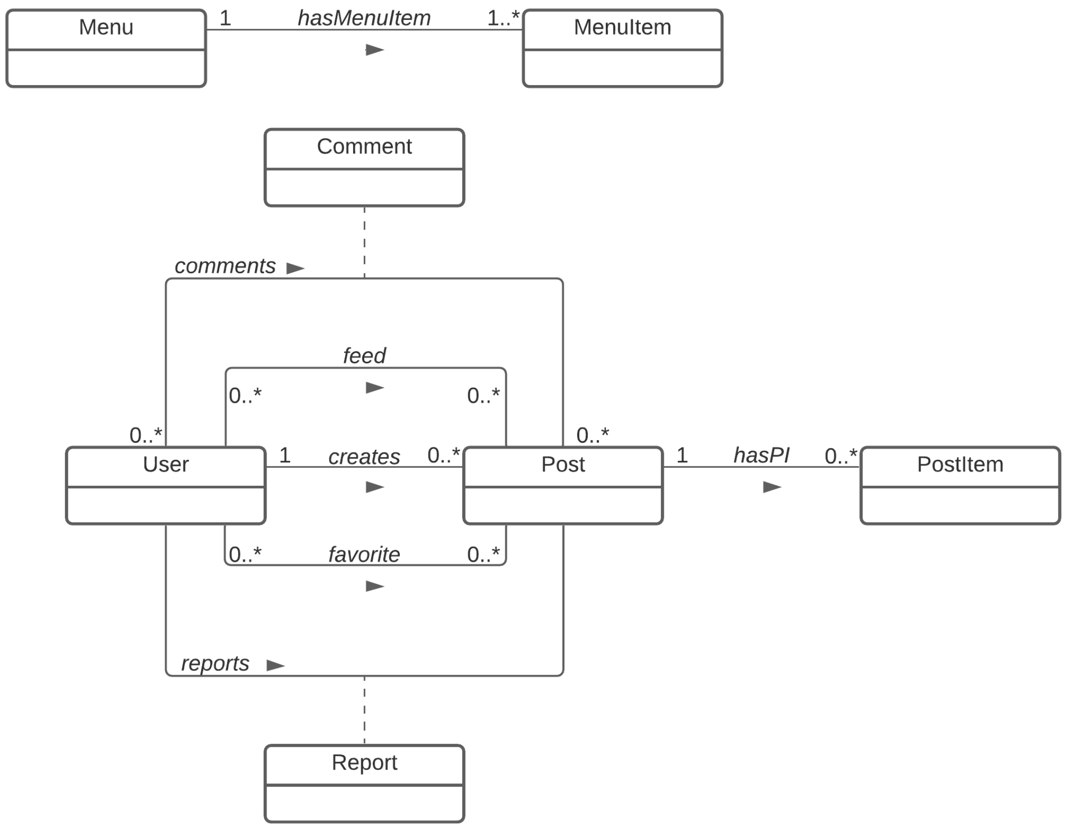


Figure I.4: UML class diagram showing the associations between some of the Entities.

Figure I.5 shows some of the associations between *Entities* and their corresponding *Attributes*.

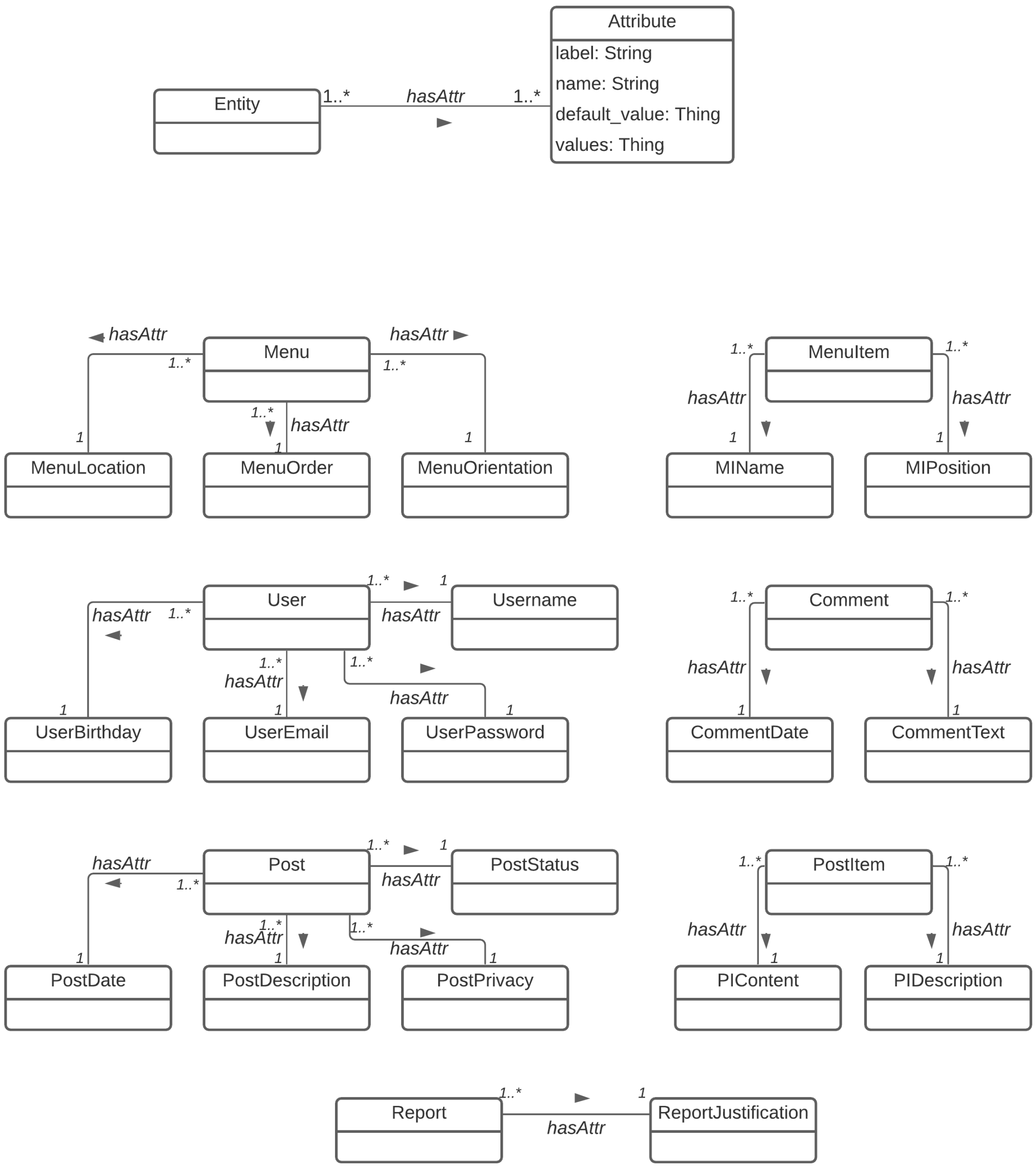


Figure I.5: UML class diagram showing the relationships between some Entities and Attributes.

I.3 Description Logics - First Ontology

I.3.1 Classes

$$\begin{aligned}
Entity &\sqsubseteq Thing\\
Attribute &\sqsubseteq Thing\\
Aggregate &\sqsubseteq Thing
\end{aligned}$$

$$\begin{aligned}
Menu &\sqsubseteq Entity\\
MenuAttribute &\sqsubseteq Attribute\\
MenuLocation &\sqsubseteq MenuAttribute\\
MenuOrder &\sqsubseteq MenuAttribute\\
MenuOrientation &\sqsubseteq MenuAttribute
\end{aligned}$$

$$\begin{aligned}
MenuItem &\sqsubseteq Entity\\
MenuItemAttribute &\sqsubseteq Attribute\\
MIName &\sqsubseteq MenuItemAttribute\\
MIPosition &\sqsubseteq MenuItemAttribute
\end{aligned}$$

$$\begin{aligned}
PostItem &\sqsubseteq Entity\\
Breed &\sqsubseteq PostItem\\
Picture &\sqsubseteq PostItem\\
Species &\sqsubseteq PostItem\\
MapLocation &\sqsubseteq PostItem\\
PostItemAttribute &\sqsubseteq Attribute\\
PIDescription &\sqsubseteq PostItemAttribute\\
PIContent &\sqsubseteq PostItemAttribute
\end{aligned}$$

$$\begin{aligned}
User &\sqsubseteq Entity\\
UserAttribute &\sqsubseteq Attribute\\
Username &\sqsubseteq UserAttribute\\
UserEmail &\sqsubseteq UserAttribute\\
UserBirthday &\sqsubseteq UserAttribute\\
UserPassword &\sqsubseteq UserAttribute
\end{aligned}$$

$$\begin{aligned}
Post &\sqsubseteq Entity\\
PostAttribute &\sqsubseteq Attribute\\
PostDate &\sqsubseteq PostAttribute\\
PostStatus &\sqsubseteq PostAttribute\\
PostPrivacy &\sqsubseteq PostAttribute\\
PostDescription &\sqsubseteq PostAttribute
\end{aligned}$$

$$\begin{aligned}
Comment &\sqsubseteq Entity\\
CommentAttribute &\sqsubseteq Attribute\\
CommentDate &\sqsubseteq CommentAttribute\\
CommentText &\sqsubseteq CommentAttribute
\end{aligned}$$

$$\begin{aligned}
Report &\sqsubseteq Entity\\
ReportAttribute &\sqsubseteq Attribute\\
ReportJustification &\sqsubseteq ReportAttribute
\end{aligned}$$

$$
\begin{aligned}
Entity &\sqsubseteq \neg Attribute\\
MenuAttribute &\sqsubseteq \neg MenuItemAttribute \sqcap \neg UserAttribute \sqcap \neg PostAttribute\\
&\sqcap \neg PostItemAttribute \sqcap \neg CommentAttribute \sqcap \neg ReportAttribute\\
MenuItemAttribute &\sqsubseteq \neg UserAttribute \sqcap \neg PostAttribute \sqcap \neg PostItemAttribute\\
&\sqcap \neg CommentAttribute \sqcap \neg ReportAttribute\\
UserAttribute &\sqsubseteq \neg PostAttribute \sqcap \neg PostItemAttribute \sqcap \neg CommentAttribute\\
&\sqcap \neg ReportAttribute\\
PostAttribute &\sqsubseteq \neg PostItemAttribute \sqcap \neg CommentAttribute \sqcap \neg ReportAttribute\\
PostItemAttribute &\sqsubseteq \neg CommentAttribute \sqcap \neg ReportAttribute\\
CommentAttribute &\sqsubseteq \neg ReportAttribute\\
CommentDate &\sqsubseteq \neg CommentText\\
MenuLocation &\sqsubseteq \neg MenuOrder \sqcap \neg MenuOrientation\\
MenuOrder &\sqsubseteq \neg MenuOrientation\\
PostDate &\sqsubseteq \neg PostDescription \sqcap \neg PostPrivacy \sqcap \neg PostStatus\\
PostDescription &\sqsubseteq \neg PostPrivacy \sqcap \neg PostStatus\\
PostPrivacy &\sqsubseteq \neg PostStatus\\
PIContent &\sqsubseteq \neg PIDescription\\
PIContent &\sqsubseteq \neg PIDescription\\
UserBirthday &\sqsubseteq \neg UserEmail \sqcap \neg Username \sqcap \neg UserPassword\\
UserEmail &\sqsubseteq \neg Username \sqcap \neg UserPassword\\
Username &\sqsubseteq \neg UserPassword\\
Breed &\sqsubseteq \neg MapLocation \sqcap \neg Picture \sqcap \neg Species\\
MapLocation &\sqsubseteq \neg Picture \sqcap \neg Species\\
Picture &\sqsubseteq \neg Species
\end{aligned}
$$

I.3.2 Object Properties

$$\begin{aligned}
\exists hasMenuItem &\sqsubseteq Menu\\
\exists hasMenuItem^{-} &\sqsubseteq MenuItem\\
Menu &\sqsubseteq \exists hasMenuItem\\
MenuItem &\sqsubseteq \exists hasMenuItem^{-}\\
&\sqcap (\leq 1 hasMenuItem^{-})
\end{aligned}$$

$$\begin{aligned}
\exists hasPI &\sqsubseteq Post\\
\exists hasPI^{-} &\sqsubseteq PostItem\\
PostItem &\sqsubseteq \exists hasPI^{-} \sqcap (\leq 1 hasPI^{-})
\end{aligned}$$

$$\begin{aligned}
\exists creates &\sqsubseteq User\\
\exists creates^{-} &\sqsubseteq Post\\
Post &\sqsubseteq \exists creates^{-} \sqcap (\leq 1 creates^{-})\\
\exists favorite &\sqsubseteq User\\
\exists favorite^{-} &\sqsubseteq Post\\
\exists feed &\sqsubseteq User\\
\exists feed^{-} &\sqsubseteq Post
\end{aligned}$$

$$\begin{aligned}
\exists commentAbout &\sqsubseteq Comment\\
\exists commentAbout^{-} &\sqsubseteq Post\\
Comment &\sqsubseteq \exists commentAbout\\
&\sqcap (\leq 1 commentAbout)\\
\exists commentBy &\sqsubseteq Comment\\
\exists commentBy^{-} &\sqsubseteq User\\
Comment &\sqsubseteq \exists commentBy\\
&\sqcap (\leq 1 commentBy)
\end{aligned}$$

$$\begin{aligned}
\exists reportedAbout &\sqsubseteq Report\\
\exists reportedAbout^{-} &\sqsubseteq Post\\
Report &\sqsubseteq \exists reportedAbout\\
&\sqcap (\leq 1 reportedAbout)\\
\exists reportedBy &\sqsubseteq Report\\
\exists reportedBy^{-} &\sqsubseteq User\\
Report &\sqsubseteq \exists reportedBy\\
&\sqcap (\leq 1 reportedBy)
\end{aligned}$$

$$
\begin{aligned}
\exists hasAttr &\sqsubseteq Entity\\
\exists hasAttr^{-} &\sqsubseteq Attribute\\
Entity &\sqsubseteq \exists hasAttr\\
Attribute &\sqsubseteq \exists hasAttr^{-}
\end{aligned}
$$

$$
\begin{aligned}
Menu &\sqsubseteq \exists hasAttr.MenuLocation \sqcap (\leq 1 hasAttr.MenuLocation)\\
Menu &\sqsubseteq \exists hasAttr.MenuOrder \sqcap (\leq 1 hasAttr.MenuOrder)\\
Menu &\sqsubseteq \exists hasAttr.MenuOrientation \sqcap (\leq 1 hasAttr.MenuOrientation)
\end{aligned}
$$

$$
\begin{aligned}
MenuItem &\sqsubseteq \exists hasAttr.MIName \sqcap (\leq 1 hasAttr.MIName)\\
MenuItem &\sqsubseteq \exists hasAttr.MIPosition \sqcap (\leq 1 hasAttr.MIPosition)
\end{aligned}
$$

$$
\begin{aligned}
User &\sqsubseteq \exists hasAttr.UserBirthday \sqcap (\leq 1 hasAttr.UserBirthday)\\
User &\sqsubseteq \exists hasAttr.UserEmail \sqcap (\leq 1 hasAttr.UserEmail)\\
User &\sqsubseteq \exists hasAttr.UserPassword \sqcap (\leq 1 hasAttr.UserPassword)\\
User &\sqsubseteq \exists hasAttr.Username \sqcap (\leq 1 hasAttr.Username)
\end{aligned}
$$

$$
\begin{aligned}
Comment &\sqsubseteq \exists hasAttr.CommentDate \sqcap (\leq 1 hasAttr.CommentDate)\\
Comment &\sqsubseteq \exists hasAttr.CommentText \sqcap (\leq 1 hasAttr.CommentText)
\end{aligned}
$$

$$
\begin{aligned}
Post &\sqsubseteq \exists hasAttr.PostDate \sqcap (\leq 1 hasAttr.PostDate)\\
Post &\sqsubseteq \exists hasAttr.PostDescription \sqcap (\leq 1 hasAttr.PostDescription)\\
Post &\sqsubseteq \exists hasAttr.PostPrivacy \sqcap (\leq 1 hasAttr.PostPrivacy)\\
Post &\sqsubseteq \exists hasAttr.PostStatus \sqcap (\leq 1 hasAttr.PostStatus)
\end{aligned}
$$

$$
\begin{aligned}
PostItem &\sqsubseteq \exists hasAttr.PIContent \sqcap (\leq 1 hasAttr.PIContent)\\
PostItem &\sqsubseteq \exists hasAttr.PIDescription \sqcap (\leq 1 hasAttr.PIDescription)
\end{aligned}
$$

$$
Report \sqsubseteq \exists hasAttr.ReportJustification \sqcap (\leq 1 hasAttr.ReportJustification)
$$

I.3.3 Data Properties

$$
\begin{aligned}
\exists label &\sqsubseteq Attribute\\
\exists name &\sqsubseteq Attribute\\
\exists default\_value &\sqsubseteq Attribute\\
\exists values &\sqsubseteq Attribute
\end{aligned}
$$

$$
\begin{aligned}
UserBirthday &\sqsubseteq \exists label.\{"birthday"\} \sqcap (\leq 1 label.\{"birthday"\})\\
UserBirthday &\sqsubseteq \exists name.\{"birthday"\} \sqcap (\leq 1 name.\{"birthday"\})\\
UserBirthday &\sqsubseteq \exists values.\{dateTime[\geq "1990-01-01T00:00:00"]\}\\
&\sqcap (\leq 1 values.\{dateTime[\geq "1990-01-01T00:00:00"]\})\\
UserEmail &\sqsubseteq \exists label.\{"email"\} \sqcap (\leq 1 label.\{"email"\})\\
UserEmail &\sqsubseteq \exists name.\{"email"\} \sqcap (\leq 1 name.\{"email"\})\\
UserEmail &\sqsubseteq \exists values.\{string[minLength 10, maxLength 50,\\
&pattern"([a-zA-Z0-9]+([.-\_]?[a-zA-Z0-9]+)*)\\
&@[a-zA-Z0-9]+([.-]?[a-zA-Z0-9]+)*.([a-zA-Z]2,5)"]\}\\
&\sqcap (\leq 1 values.\{string[minLength 10, maxLength 50,\\
&pattern"([a-zA-Z0-9]+([.-\_]?[a-zA-Z0-9]+)*)\\
&@[a-zA-Z0-9]+([.-]?[a-zA-Z0-9]+)*.([a-zA-Z]2,5)"]\}\\
Username &\sqsubseteq \exists label.\{"username"\} \sqcap (\leq 1 label.\{"username"\})\\
Username &\sqsubseteq \exists name.\{"username"\} \sqcap (\leq 1 name.\{"username"\})\\
Username &\sqsubseteq \exists values.\{string[pattern"[a-zA-Z0-9.-\_]4,20"]\}\\
&\sqcap (\leq 1 values.\{string[pattern"[a-zA-Z0-9.-\_]4,20"]\})\\
UserPassword &\sqsubseteq \exists label.\{"password"\} \sqcap (\leq 1 label.\{"password"\})\\
UserPassword &\sqsubseteq \exists name.\{"password"\} \sqcap (\leq 1 name.\{"password"\})\\
UserPassword &\sqsubseteq \exists values.\{string[pattern".6,50"]\}\\
&\sqcap (\leq 1 values.\{string[pattern".6,50"]\})
\end{aligned}
$$

$$\begin{aligned}
PostDate &\sqsubseteq \exists label.\{"postDate"\} \sqcap (\leq 1 label.\{"postDate"\})\\
PostDate &\sqsubseteq \exists name.\{"postDate"\} \sqcap (\leq 1 name.\{"postDate"\})\\
PostDate &\sqsubseteq \exists values.\{dateTime[\geq "1990-01-01T00:00:00"]\}\\
&\sqcap (\leq 1 values.\{dateTime[\geq "1990-01-01T00:00:00"]\})\\
PostDescription &\sqsubseteq \exists label.\{"postDescription"\} \sqcap (\leq 1 label.\{"postDescription"\})\\
PostDescription &\sqsubseteq \exists name.\{"postDescription"\} \sqcap (\leq 1 name.\{"postDescription"\})\\
PostDescription &\sqsubseteq \exists values.\{string[pattern".1,100"]\}\\
&\sqcap (\leq 1 values.\{string[pattern".1,100"]\})\\
PostPrivacy &\sqsubseteq \exists label.\{"postPrivacy"\} \sqcap (\leq 1 label.\{"postPrivacy"\})\\
PostPrivacy &\sqsubseteq \exists name.\{"postPrivacy"\} \sqcap (\leq 1 name.\{"postPrivacy"\})\\
PostPrivacy &\sqsubseteq \exists values.\{"friends","private","public"\}\\
&\sqcap (\leq 1 values.\{"friends","private","public"\})\\
PostPrivacy &\sqsubseteq \exists default\_value.\{"friends"\} \sqcap (\leq 1 default\_value.\{"friends"\})\\
PostStatus &\sqsubseteq \exists label.\{"postStatus"\} \sqcap (\leq 1 label.\{"postStatus"\})\\
PostStatus &\sqsubseteq \exists name.\{"postStatus"\} \sqcap (\leq 1 name.\{"postStatus"\})\\
PostStatus &\sqsubseteq \exists values.\{"censored","open","solved"\}\\
&\sqcap (\leq 1 values.\{"censored","open","solved"\})\\
PostStatus &\sqsubseteq \exists default\_value.\{"open"\} \sqcap (\leq 1 default\_value.\{"open"\})\\
CommentDate &\sqsubseteq \exists label.\{"commentDate"\} \sqcap (\leq 1 label.\{"commentDate"\})\\
CommentDate &\sqsubseteq \exists name.\{"commentDate"\} \sqcap (\leq 1 name.\{"commentDate"\})\\
CommentDate &\sqsubseteq \exists values.\{dateTime[>= "1990-01-01T00:00:00"]\}\\
&\sqcap (\leq 1 values.\{dateTime[>= "1990-01-01T00:00:00"]\})\\
CommentText &\sqsubseteq \exists label.\{"commentText"\} \sqcap (\leq 1 label.\{"commentText"\})\\
CommentText &\sqsubseteq \exists name.\{"commentText"\} \sqcap (\leq 1 name.\{"commentText"\})\\
CommentText &\sqsubseteq \exists values.\{string[minLength 1, maxLength 300]\}\\
&\sqcap (\leq 1 values.\{string[minLength 1, maxLength 300]\})
\end{aligned}$$

I.3.4 Queries

This query returns all posts with the status "solved".

$$PostsSolved \sqsubseteq Aggregate$$

$$PostsSolved \equiv Post \sqcap \exists hasAttr.PostStatus.values.\{"solved"\}$$

This query returns all users that created at least one post on the specified date.

$$UserCommentedDay \sqsubseteq Aggregate$$

$$UserCommentedDay \equiv User \sqcap \exists commentBy^{-}.$$
$$Comment.hasAttr.CommentDate.\{dateTime["2021-01-19T00:00:00"]\}$$

This query returns all users that have at least 3 posts that were created this year and that got censored.

$$UsersReportedPosts \sqsubseteq Aggregate$$

$$UsersReportedPosts \equiv User \sqcap (\geq 3 creates.Post \sqcap$$
$$(hasAttr.PostDate.values\{dateTime[\geq "2021-01-01T00:00:00"]\}$$
$$\sqcap\, hasAttr.PostStatus.values\{"censored"\}))$$

I.4 Description Logics - Second Ontology

I.4.1 Classes

$$\begin{aligned}
Entity &\sqsubseteq Thing & CommentAttribute &\sqsubseteq Attribute\\
Attribute &\sqsubseteq Thing & CommentDate &\sqsubseteq CommentAttribute\\
OutSystems &\sqsubseteq Thing & CommentDescription &\sqsubseteq CommentAttribute
\end{aligned}$$

$$\begin{aligned}
User &\sqsubseteq Entity & ElementOfComponent &\sqsubseteq OutSystems\\
Post &\sqsubseteq Entity & ElemDetail &\sqsubseteq ElementOfComponent\\
Comment &\sqsubseteq Entity & ElemEntityDB &\sqsubseteq ElementOfComponent\\
Post &\sqsubseteq Entity & ElemSearchableList &\sqsubseteq ElementOfComponent
\end{aligned}$$

$$\begin{aligned}
PostAttribute &\sqsubseteq Attribute & &\\
PostDate &\sqsubseteq PostAttribute & &\\
PostStatus &\sqsubseteq PostAttribute & Extendable &\sqsubseteq PostAttribute\\
PostAttribute &\sqsubseteq Attribute & ExtendableString &\sqsubseteq Extendable\\
Date &\sqsubseteq PostAttribute & ExtendableBlob &\sqsubseteq Extendable\\
Description &\sqsubseteq PostAttribute & ExtendableEnumerate &\sqsubseteq Extendable\\
Privacy &\sqsubseteq PostAttribute & ExtendableInt &\sqsubseteq Extendable\\
Title &\sqsubseteq PostAttribute & ExtendableRef &\sqsubseteq Extendable\\
Status &\sqsubseteq PostAttribute & &\\
StatusDefault &\sqsubseteq Status & &
\end{aligned}$$

$$\begin{aligned}
Entity &\sqsubseteq \neg Attribute\\
User &\sqsubseteq \neg Post\\
Date &\sqsubseteq \neg Description \sqcap \neg Privacy \sqcap \neg Status \sqcap \neg Title\\
Description &\sqsubseteq \neg Privacy \sqcap \neg Status \sqcap \neg Title\\
Privacy &\sqsubseteq \neg Status \sqcap \neg Title\\
Status &\sqsubseteq \neg Title
\end{aligned}$$

I.4.2 Object Properties

$$\begin{aligned}
\exists create &\sqsubseteq User\\
\exists hasAttribute &\sqsubseteq Entity\\
\exists hasAttribute^{-} &\sqsubseteq Attribute\\
\exists read &\sqsubseteq User\\
\exists search &\sqsubseteq User
\end{aligned}$$

$$Thing \sqsubseteq \exists create^{-} \sqcap (\leq 1 creates^{-})$$

$$\begin{aligned}
Comment &\sqsubseteq \exists create^{-}.User\\
CommentAttribute &\sqsubseteq \forall hasAttr^{-}.Comment
\end{aligned}$$

$$\begin{aligned}
Post &\sqsubseteq \exists create^{-}.User\\
Post &\sqsubseteq \exists hasAttr.Date \sqcap \exists hasAttr.Description \sqcap \exists hasAttr.Privacy\\
&\sqcap \exists hasAttr.Status \sqcap \exists hasAttr.Title
\end{aligned}$$

I.4.3 Data Properties

$$\begin{aligned}
StatusDefault &\sqsupseteq \exists values.``active", ``censored", ``archived"\\
&\sqcap (\leq values.``active", ``censored", ``archived")
\end{aligned}$$

$$\begin{aligned}
\exists default\_value &\sqsubseteq Attribute\\
\exists format &\sqsubseteq Attribute\\
\exists listColumns &\sqsubseteq ElemSearchableList\\
\exists listFilter &\sqsubseteq ElemSearchableList\\
\exists listSort &\sqsubseteq ElemSearchableList\\
\exists mandatory &\sqsubseteq Attribute\\
\exists restriction &\sqsubseteq Thing\\
\exists values &\sqsubseteq Attribute
\end{aligned}$$

I.4.4 General Class Axioms

$$\begin{aligned}
Entity \sqcap \exists hasAttr.Attribute &\sqsubseteq ElemEntityDB \\
Entity \sqcap \exists create^{-}.User &\sqsubseteq ElemEntityDB \\
Attribute \sqcap \exists values.\{string[pattern{''}\backslash\backslash[.^{*}(,.)^{*}\backslash\backslash]{''}]\} &\sqsubseteq ElemEntityDB
\end{aligned}$$

$$\exists create^{-}.User \sqsubseteq ElemDetail$$

$$\exists read^{-}.User \sqcap \exists search^{-}.User \sqsubseteq ElemSearchableList$$

II

Annex II - Template Language Formalization

II.1 Syntax

$$
\begin{array}{lcll}
\alpha & ::= & & (\textbf{node categories}) \\
 & & \varepsilon & (\text{empty}) \\
 & | & \mathit{Top} & (\text{top}) \\
 & | & \mathit{Screen} & (\text{screen}) \\
 & | & \mathit{Table} & (\text{table}) \\
 & | & \mathit{Column} & (\text{column}) \\
 & | & \mathit{Icon} & (\text{icon}) \\
 & | & \mathit{Expression} & (\text{expression}) \\
 & | & \mathit{Input} & (\text{input}) \\
 & | & \mathit{CheckBox} & (\text{check box}) \\
 & | & \mathit{Calendar} & (\text{calendar}) \\
 & | & \mathit{Container} & (\text{container}) \\
 & | & \mathit{List} & (\text{list}) \\
 & | & \mathit{ListItem} & (\text{list item}) \\
 & | & \mathit{Search} & (\text{search}) \\
 & | & \mathit{Chart} & (\text{chart}) \\
 & | & \mathit{Counter} & (\text{counter}) \\
 & | & \mathit{Pagination} & (\text{pagination})
\end{array}
$$

Figure II.1: Node categories.

$$\begin{array}{llll}
p & ::= & & (\textbf{properties}) \\
 & & \textit{Name} & \text{(name property)} \\
 & | & \textit{Title} & \text{(title property)} \\
 & | & \textit{Description} & \text{(description property)} \\
 & | & \textit{Type} & \text{(type property)} \\
 & | & \textit{DisplayName} & \text{(display name property)} \\
 & | & \textit{Source} & \text{(source property)} \\
 & | & \textit{Visible} & \text{(visible property)} \\
 & | & \textit{Value} & \text{(value property)} \\
 & | & \textit{InputType} & \text{(input type property)} \\
 & | & \textit{Variable} & \text{(variable property)} \\
 & | & \textit{Attributes} & \text{(attributes field)} \\
 & | & \textit{FilterBy} & \text{(filter property)} \\
 & | & \textit{AttrGroup} & \text{(attribute to group by property)}
\end{array}$$

Figure II.2: Properties.

vl	$::=$		(**value literals**)
		num	(number literal)
	$\mid$	$string$	(string literal)
	$\mid$	$bool$	(boolean literal)
N	$::=$	$N_1 \mid N_2 \mid \ldots$	**(name identifiers)**
V	$::=$		**(model elements)**
		$\texttt{Entity}\langle N, \{L_i = V_i{}^{i\in 1..p}\}, \{L_i = [vl_j{}^{j\in 1..m}]_i{}^{i\in 1..p}\}\rangle$	(entity element)
	$\mid$	$\texttt{Attribute}\langle N, L, B, \{p_i = M_i{}^{i\in 1..p}\}\rangle$	(attribute element)
M	$::=$		(**template terms**)
		vl	(value literal)
	$\mid$	x	(compile-time variable)
	$\mid$	u	(runtime variable)
	$\mid$	L	(label)
	$\mid$	V	(model element)
	$\mid$	$\{L_i = M_i{}^{i\in 1..p}\}$	(record)
	$\mid$	$[M_i{}^{i\in 1..p}]$	(list)
	$\mid$	$M_1 \boldsymbol{.} M_2$	(selection operation)
	$\mid$	$M_1 \hat{\boldsymbol{.}} M_2$	(runtime selection operation)
	$\mid$	$\texttt{NameOf}\ M$	(name property)
	$\mid$	$\texttt{LabelOf}\ M$	(label property)
	$\mid$	$\texttt{AttributesOf}\ M$	(attributes)
	$\mid$	$M\ \texttt{isOfType}\ \tau$	(type verification)
	$\mid$	$\texttt{let}\ x = M_1\ \texttt{in}\ M_2$	(let expression)
	$\mid$	$\texttt{Template}\langle x, \tau, M\rangle$	(template declaration)
	$\mid$	$M_1(M_2)$	(template instantiation)
	$\mid$	$\texttt{Node}\langle \alpha, \{p_i = M_{1_i}{}^{i\in 1..p}\}, [M_{2_j}{}^{j\in 1..m}]\rangle$	(node element)
	$\mid$	$\texttt{NodeValue}\langle \alpha, \{p_i = v_{1_i}{}^{i\in 1..p}\}, [v_{2_j}{}^{j\in 1..m}]\rangle$	(runtime node)
	$\mid$	$\texttt{forNode}(x\colon t = M_1\ \texttt{in}\ M_2)$	(loop instruction node)
	$\mid$	$\texttt{forNode}(x\colon \texttt{AttributeT}(t)_n = M_1\ \texttt{in}\ M_2)$	(loop for attributes node)
	$\mid$	$\texttt{ifNode}(M, M_T, M_F)$	(conditional branching node)
	$\mid$	$\texttt{Box}(M)$	(runtime term constructor)
	$\mid$	$\texttt{letbox}\ u = M_1\ \texttt{in}\ M_2$	(runtime term destructor)
	$\mid$	$M_1[M_2]$	(indexing)
	$\mid$	$\Lambda_n n.M$	(name abstraction)
	$\mid$	$\Lambda_t t.M$	(type abstraction)
	$\mid$	$\Lambda_r r.M$	(row abstraction)
	$\mid$	$M[name]_n$	(name application)
	$\mid$	$M[type]_t$	(type application)
	$\mid$	$M[row]_r$	(row application)

Figure II.3: Syntax of the template language.

$$
\begin{array}{lcll}
v & ::= & & (\textbf{values}) \\
 & & vl & \\
 & | & \texttt{Entity}\langle N, \{L_i = v_i{}^{i\in 1..p}\}, \{L_i = [vl_j{}^{j\in 1..m}]_i{}^{i\in 1..p}\}\rangle & \\
 & | & \texttt{Attribute}\langle N, L, B, \{p_i = v_i{}^{i\in 1..p}\}\rangle & \\
 & | & L & \\
 & | & \{L_i = v_i{}^{i\in 1..p}\} & \\
 & | & [v_i{}^{i\in 1..p}] & \\
 & | & v_1 \hat{:} v_2 & \\
 & | & \texttt{NodeValue}\langle \alpha, \{p_i = v_{1_i}{}^{i\in 1..p}\}, [v_{2_j}{}^{j\in 1..m}]\rangle & \\
 & | & \texttt{Box}(M) &
\end{array}
$$

Figure II.4: Syntax of compile time values (runtime terms).

$$
\begin{array}{lcll}
B & ::= & & (\textbf{basic types}) \\
 & & \mathit{Num} & (\text{number}) \\
 & | & \mathit{String} & (\text{string}) \\
 & | & \mathit{Bool} & (\text{bool}) \\
 & & & \\
\tau & ::= & & (\textbf{types}) \\
 & & B & (\text{basic types}) \\
 & | & \texttt{Name}(N) & (\text{name}) \\
 & | & \texttt{Label}(L) & (\text{label}) \\
 & | & \texttt{LabelAttr}(B)_n & (\text{attribute label}) \\
 & | & \{L_i : \tau_i{}^{i\in 1..p}\} & (\text{record}) \\
 & | & \texttt{RecordAttr}_n & (\text{record of entity's attributes}) \\
 & | & [\tau] & (\text{list}) \\
 & | & \texttt{ListAttr}_n & (\text{list of entity's attributes}) \\
 & | & \texttt{EntityT}(n, \{L_i : \tau_i{}^{i\in 1..p}\}) & (\text{entity}) \\
 & | & \texttt{AttributeT}(\tau)_n & (\text{attribute}) \\
 & | & \texttt{NodeT}([\alpha_i{}^{i\in 1..p}], \{p_j : \tau_j{}^{j\in 1..m}\}) & (\text{node}) \\
 & | & \texttt{BoxT}(\tau) & (\text{delayed type}) \\
 & | & \texttt{TemplateT}(\tau_1 \to \tau_2) & (\text{template}) \\
 & | & n & (\text{name variable}) \\
 & | & t & (\text{type variable}) \\
 & | & r & (\text{row variable}) \\
 & | & \forall_n n.\tau & (\text{forall name}) \\
 & | & \forall_t t.\tau & (\text{forall type}) \\
 & | & \forall_r r.\tau & (\text{forall rows}) \\
 & | & \top & (\text{top})
\end{array}
$$

Figure II.5: Syntax of types.

$$
\begin{array}{rcll}
\Gamma & ::= & & (\textbf{compile-time contexts}) \\
& & \varnothing & (\text{empty context}) \\
& | & \Gamma, x \colon \tau & (\text{term variable binding}) \\
\\
\Delta & ::= & & (\textbf{runtime contexts}) \\
& & \varnothing & (\text{empty context}) \\
& | & \Delta, u \colon \tau & (\text{term variable binding}) \\
\\
\Omega & ::= & & (\textbf{name variable contexts}) \\
& & \varnothing & (\text{empty context}) \\
& | & \Omega, n & (\text{name variable binding}) \\
\\
\Phi & ::= & & (\textbf{rows variable contexts}) \\
& & \varnothing & (\text{empty context}) \\
& | & \Phi, r & (\text{rows variable binding}) \\
\\
\Upsilon & ::= & & (\textbf{type variable contexts}) \\
& & \varnothing & (\text{empty context}) \\
& | & \Upsilon, t & (\text{type variable binding})
\end{array}
$$

Figure II.6: Syntax of contexts.

II.2 Type System

$$\frac{}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \mathit{num}\colon \mathit{Num}}\;[\![\text{T-Num}]\!] \qquad \frac{}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \mathit{string}\colon \mathit{String}}\;[\![\text{T-Str}]\!]$$

$$\frac{}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \mathit{bool}\colon \mathit{Bool}}\;[\![\text{T-Bool}]\!]$$

$$\frac{}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash N\colon \texttt{Name}(N)}\;[\![\text{T-Name}]\!] \quad \frac{}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash L\colon \texttt{Label}(L)}\;[\![\text{T-Lab}]\!]$$

$$\frac{\text{for each } i \qquad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_i\colon \tau_i}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \{L_i = M_i{}^{i\in 1..p}\}\colon \{L_i\colon \tau_i{}^{i\in 1..p}\}}\;[\![\text{T-Rec}]\!]$$

$$\frac{\text{for each } i \qquad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_i\colon \tau}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash [M_i{}^{i\in 1..p}]\colon [\tau]}\;[\![\text{T-List}]\!]$$

$$\frac{x\colon \tau \in \Gamma}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash x\colon \tau}\;[\![\text{T-CVar}]\!] \qquad \frac{u\colon \tau \in \Delta}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash u\colon \tau}\;[\![\text{T-RVar}]\!]$$

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash N\colon \texttt{Name}(N)\\ \text{for each } i \qquad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_i\colon \texttt{AttributeT}(B_i)_N\\ \text{for each } i, j \qquad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash v_{ji}\colon B_i\end{array}}{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{Entity}\langle N, \{L_i = M_i{}^{i\in 1..p}\}, \{L_i = [v_j{}^{j\in 1..m}]_i{}^{i\in 1..p}\}\rangle\colon\\ \texttt{EntityT}(N, \{L_i\colon \texttt{AttributeT}(B_i)_N{}^{i\in 1..p}\})\end{array}}\;[\![\text{T-Ent}]\!]$$

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash N\colon \texttt{Name}(N)\\ \text{for each } i \qquad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_i\colon \tau_i\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{Attribute}\langle N, L, B, \{p_i = M_i{}^{i\in 1..p}\}\rangle\colon \texttt{AttributeT}(B)_N}\;[\![\text{T-Attr}]\!]$$

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M\colon \texttt{EntityT}(N, \{L_i\colon \texttt{AttributeT}(B_i)_N{}^{i\in 1..p}\})}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{NameOf}\ M\colon \texttt{BoxT}(\{\mathit{List}\colon \{\mathit{Current}\colon \texttt{RecordAttr}_N\}\})}\;[\![\text{T-NOf}]\!]$$

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M\colon \texttt{AttributeT}(B)_N}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{LabelOf}\ M\colon \texttt{BoxT}(\texttt{LabelAttr}(B)_N)}\;[\![\text{T-LOf}]\!]$$

Figure II.7: Type system rules.

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M\colon \mathtt{EntityT}(N,R)}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \mathtt{AttributesOf}\ M\colon \mathtt{ListAttr}_N}\ [\![\text{T-AOF}]\!]$$

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M\colon \tau'}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M\ \mathtt{isOfType}\ \tau\colon \mathit{Bool}}\ [\![\text{T-OFTY}]\!]$$

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1\colon \tau_1 \qquad (\Gamma,x\colon \tau_1);\Delta;\Omega;\Phi;\Upsilon \vdash M_2\colon \tau_2}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2\colon \tau_2}\ [\![\text{T-LET}]\!]$$

$$\frac{\begin{array}{c}\text{for each } i \qquad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_i\colon \mathtt{BoxT}(\tau_i)\\ \text{for each } j \qquad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_j\colon \mathtt{BoxT}(\mathtt{NodeT}([{\alpha_h}^{h\in 1..f}],\{p_k\colon {\tau_k}^{k\in 1..l}\})_j)\end{array}}{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \mathtt{Node}\langle \alpha,\{p_i = {M_i}^{i\in 1..p}\},[{M_j}^{j\in 1..m}]\rangle\colon\\ \mathtt{BoxT}(\mathtt{NodeT}([\alpha],\{p_i\colon {\tau_i}^{i\in 1..p}\}))\end{array}}\ [\![\text{T-NODE}]\!]$$

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1\colon [\tau_1]\\ (\Gamma,x\colon t);\Delta;\Omega;\Phi;(\Upsilon,t=\tau_1) \vdash M_2\colon \mathtt{BoxT}(\tau_2)\\ \tau_2 = \mathtt{NodeT}([{\alpha_j}^{j\in 1..m}],\{p_k\colon {\tau_k}^{k\in 1..l}\})\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \mathtt{forNode}(x\colon t = M_1\ \mathtt{in}\ M_2)\colon \mathtt{BoxT}(\tau_2)}\ [\![\text{T-FOR}]\!]$$

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1\colon \mathtt{ListAttr}_N\\ (\Gamma,x\colon \mathtt{AttributeT}(t)_n);\Delta;(\Omega,n=N);\Phi;(\Upsilon,t=\top) \vdash M_2\colon \mathtt{BoxT}(\tau_2)\\ \tau_2 = \mathtt{NodeT}([{\alpha_i}^{i\in 1..p}],\{p_j\colon {\tau_j}^{j\in 1..m}\})\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \mathtt{forNode}(x\colon \mathtt{AttributeT}(t)_n = M_1\ \mathtt{in}\ M_2)\colon \mathtt{BoxT}(\tau_2)}\ [\![\text{T-FORATTR}]\!]$$

with:

$$\begin{array}{l}\textsc{realType}(\tau) \triangleq \text{match } \tau \text{ with}\\ \qquad\qquad \mid \mathtt{AttributeT}(B)_n \to B\\ \qquad\qquad \mid \_ \to \tau\end{array}$$

Figure II.8: Type system rules (part 2).

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M : \mathit{Bool}\\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_T : \texttt{BoxT}(\texttt{NodeT}([\alpha_i{}^{i\in 1..p}],\{p_j : \tau_j{}^{j\in 1..m}\}))\\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_F : \texttt{BoxT}(\texttt{NodeT}([\alpha_h{}^{h\in 1..f}],\{p_k : \tau_k{}^{k\in 1..l}\}))\end{array}}{\begin{array}{l}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{ifNode}(M,\, M_T,\, M_F) :\\ \qquad \texttt{BoxT}(\texttt{NodeT}([\alpha_i{}^{i\in 1..p}] \cup [\alpha_h{}^{h\in 1..f}],\{p_j : \tau_j{}^{j\in 1..m}\} \cap \{p_k : \tau_k{}^{k\in 1..l}\}))\end{array}}\ [\![\text{T-IF}]\!]$$

$$\frac{(\Gamma, x : \tau_1);\Delta;\Omega;\Phi;\Upsilon \vdash M : \tau_2}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{Template}\langle x,\, \tau_1,\, M\rangle : \texttt{TemplateT}(\tau_1 \to \tau_2)}\ [\![\text{T-TEMP}]\!]$$

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : \texttt{TemplateT}(\tau_1 \to \tau_2)\\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2 : \tau_1\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1(M_2) : \tau_2}\ [\![\text{T-INST}]\!]$$

$$\frac{\varnothing;\Delta;\Omega;\Phi;\Upsilon \vdash M : \tau}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{Box}(M) : \texttt{BoxT}(\tau)}\ [\![\text{T-BOX}]\!]$$

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : \texttt{BoxT}(\tau_1) \qquad \Gamma;(\Delta, u : \tau_1);\Omega;\Phi;\Upsilon \vdash M_2 : \tau_2}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \texttt{letbox}\ u = M_1\ \texttt{in}\ M_2 : \tau_2}\ [\![\text{T-LETB}]\!]$$

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : [\tau] \qquad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2 : \mathit{Num}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1[M_2] : \tau}\ [\![\text{T-IDX}]\!]$$

$$\frac{\Gamma;\Delta;(\Omega, n);\Phi;\Upsilon \vdash M : \tau}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \Lambda_n n.M : \forall_n n.\tau}\ [\![\text{T-FAN}]\!] \quad \frac{\Gamma;\Delta;\Omega;\Phi;(\Upsilon, t) \vdash M : \tau}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \Lambda_t t.M : \forall_t t.\tau}\ [\![\text{T-FAT}]\!]$$

$$\frac{\Gamma;\Delta;\Omega;(\Phi, r);\Upsilon \vdash M : \tau}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \Lambda_r r.M : \forall_r r.\tau}\ [\![\text{T-FAR}]\!]$$

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M : \forall_n n.\tau}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M[N]_n : [N/n]\tau}\ [\![\text{T-NAPP}]\!]$$

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M : \forall_t t.\tau}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M[T]_t : [T/t]\tau}\ [\![\text{T-TAPP}]\!]$$

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M : \forall_r r.\tau}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M[R]_r : [R/r]\tau}\ [\![\text{T-RAPP}]\!]$$

Figure II.9: Type system rules (part 3).

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : \{L_i : \tau_i \ ^{i\in 1..p}\} \\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2 : \mathsf{Label}(L_j) \qquad j \in 1..p\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 \,\mathbf{.}\, M_2 : \mathsf{BoxT}(\tau_j)} \ [\![\text{T-Sel1}]\!]$$

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : \mathsf{NodeT}([\alpha_i \ ^{i\in 1..p}], \{p_j : \tau_j \ ^{j\in 1..m}\}) \\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2 : \mathsf{Label}(L_k) \qquad k \in 1..m\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 \,\mathbf{.}\, M_2 : \mathsf{BoxT}(\tau_k)} \ [\![\text{T-Sel2}]\!]$$

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : \mathsf{AttributeT}(B)_N \\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2 : \mathsf{Label}(\mathsf{DisplayName})\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 \,\mathbf{.}\, M_2 : \mathsf{BoxT}(String)} \ [\![\text{T-Sel3}]\!]$$

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : \{L_i : \tau_i \ ^{i\in 1..p}\} \\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2 : \mathsf{Label}(L_j) \qquad j \in 1..p\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 \,\hat{\mathbf{.}}\, M_2 : \tau_j} \ [\![\text{T-SelRT1}]\!]$$

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : \mathsf{NodeT}([\alpha_i \ ^{i\in 1..p}], \{p_j : \tau_j \ ^{j\in 1..m}\}) \\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2 : \mathsf{Label}(L_k) \qquad k \in 1..m\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 \,\hat{\mathbf{.}}\, M_2 : \tau_k} \ [\![\text{T-SelRT2}]\!]$$

$$\frac{\begin{array}{c}\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 : \mathsf{RecordAttr}_N \\ \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_2 : \mathsf{LabelAttr}(B)_N\end{array}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M_1 \,\hat{\mathbf{.}}\, M_2 : B} \ [\![\text{T-SelRT3}]\!]$$

Figure II.10: Type system rules (part 4).

II.3 Operational Semantics

$$\frac{M \hookrightarrow M'}{\texttt{NameOf}\ M \hookrightarrow \texttt{NameOf}\ M'}\ [\![\text{E-NOf}]\!]$$

$$\frac{}{\begin{array}{c}\texttt{NameOf}\ (\texttt{Entity}\langle n, \{L_i = V_i\ ^{i\in 1..p}\}, \{L_i = [vl_j\ ^{j\in 1..m}]_i\ ^{i\in 1..p}\}\rangle) \\ \hookrightarrow \texttt{Box}(\{List = \{Current = \{L_i = vl_{x_i}\ ^{i\in 1..p}\}\}\})\end{array}}\ [\![\text{E-NOfV}]\!]$$

$$\frac{M \hookrightarrow M'}{\texttt{LabelOf}\ M \hookrightarrow \texttt{LabelOf}\ M'}\ [\![\text{E-LOf}]\!]$$

$$\frac{}{\texttt{LabelOf}\ (\texttt{Attribute}\langle N, L, B, \{p_i = M_i\ ^{i\in 1..p}\}\rangle) \hookrightarrow \texttt{Box}(L)}\ [\![\text{E-LOfV}]\!]$$

$$\frac{M \hookrightarrow M'}{\texttt{AttributesOf}\ M \hookrightarrow \texttt{AttributesOf}\ M'}\ [\![\text{E-AOf}]\!]$$

$$\frac{}{\texttt{AttributesOf}\ (\texttt{Entity}\langle n, \{L_i = V_i\ ^{i\in 1..p}\}, \{L_i = [vl_j\ ^{j\in 1..m}]_i\ ^{i\in 1..p}\}\rangle) \hookrightarrow [V_i\ ^{i\in 1..p}]}\ [\![\text{E-AOfV}]\!]$$

$$\frac{M \hookrightarrow M'}{M\ \texttt{isOfType}\ \tau \hookrightarrow M'\ \texttt{isOfType}\ \tau}\ [\![\text{E-OfTy}]\!]$$

$$\frac{}{(\texttt{Attribute}\langle n, L, B, \{p_i : M_i\ ^{i\in 1..p}\}\rangle)\ \texttt{isOfType}\ \tau \hookrightarrow (\!|B = \tau|\!)}\ [\![\text{E-OfTyV}]\!]$$

$$\frac{M_1 \hookrightarrow M_1'}{\texttt{let}\ x = M_1\ \texttt{in}\ M_2 \hookrightarrow \texttt{let}\ x = M_1'\ \texttt{in}\ M_2}\ [\![\text{E-Let}]\!]$$

$$\frac{}{\texttt{let}\ x = v_1\ \texttt{in}\ M_2 \hookrightarrow [x \mapsto v_1]M_2}\ [\![\text{E-LetV}]\!]$$

$$\frac{M_1 \hookrightarrow M_1'}{M_1(M_2) \hookrightarrow M_1'(M_2)}\ [\![\text{E-Inst1}]\!]$$

Figure II.11: Evaluation rules.

$$\frac{M_2 \hookrightarrow M_2'}{(\texttt{Template}\langle x,\ \tau,\ M_3\rangle)(M_2) \hookrightarrow (\texttt{Template}\langle x,\ \tau,\ M_3\rangle)(M_2')}\ [\![\text{E-Inst2}]\!]$$

$$\frac{}{(\texttt{Template}\langle x,\ \tau,\ M_3\rangle)(v_2) \hookrightarrow [x \mapsto v_2]M_3}\ [\![\text{E-Inst3}]\!]$$

$$\frac{M_j \hookrightarrow M_j'}{\begin{array}{l}\{L_i = v_i{}^{i\in 1..j-1}, L_j = M_j, L_k = M_k{}^{k\in j+1..n}\} \\ \hookrightarrow \{L_i = v_i{}^{i\in 1..j-1}, L_j = M_j', L_k = M_k{}^{k\in j+1..n}\}\end{array}}\ [\![\text{E-Rec}]\!]$$

$$\frac{M_j \hookrightarrow M_j'}{[v_i{}^{i\in 1..j-1}, M_j, M_k{}^{k\in j+1..n}] \hookrightarrow [v_i{}^{i\in 1..j-1}, M_j', M_k{}^{k\in j+1..n}]}\ [\![\text{E-List}]\!]$$

$$\frac{M_1 \hookrightarrow M_1'}{\texttt{Node}\langle \alpha, M_1, M_2\rangle \hookrightarrow \texttt{Node}\langle \alpha, M_1', M_2\rangle}\ [\![\text{E-Node1}]\!]$$

$$\frac{M_2 \hookrightarrow M_2'}{\texttt{Node}\langle \alpha, v_1, M_2\rangle \hookrightarrow \texttt{Node}\langle \alpha, v_1, M_2'\rangle}\ [\![\text{E-Node2}]\!]$$

$$\frac{}{\begin{array}{l}\texttt{Node}\langle \alpha, \{p_i \colon \texttt{Box}(v_{1i})\ {}^{i\in 1..p}\}, [\texttt{Box}(v_{2j})\ {}^{j\in 1..m}]\rangle \\ \hookrightarrow \texttt{Box}(\texttt{NodeValue}\langle \alpha, \{p_i \colon v_{1i}{}^{i\in 1..p}\}, [v_{2j}{}^{j\in 1..m}]\rangle)\end{array}}\ [\![\text{E-Node3}]\!]$$

$$\frac{v_1 \underset{R}{\hookrightarrow} v_1'}{\texttt{NodeValue}\langle \alpha, v_1, v_2\rangle \underset{R}{\hookrightarrow} \texttt{NodeValue}\langle \alpha, v_1', v_2\rangle}\ [\![\text{E-NodeV1}]\!]$$

$$\frac{v_2 \underset{R}{\hookrightarrow} v_2'}{\texttt{NodeValue}\langle \alpha, v_1, v_2\rangle \underset{R}{\hookrightarrow} \texttt{NodeValue}\langle \alpha, v_1, v_2'\rangle}\ [\![\text{E-NodeV2}]\!]$$

$$\frac{M_1 \hookrightarrow M_1'}{\texttt{forNode}(x \colon \tau = M_1\ \texttt{in}\ M_2) \hookrightarrow \texttt{forNode}(x \colon \tau = M_1'\ \texttt{in}\ M_2)}\ [\![\text{E-For}]\!]$$

Figure II.12: Evaluation rules (part 2).

$$\frac{}{\begin{array}{c}\texttt{forNode}(x\colon \tau = [v_i{}^{i\in j..k}]\ \texttt{in}\ M_2)\\ \hookrightarrow ([x \mapsto v_j]M_2) :: (\texttt{forNode}(x\colon \tau = [v_i{}^{i\in j+1..k}]\ \texttt{in}\ M_2))\end{array}}\ [\![\text{E-ForV}]\!]$$

with:

$$v_0 :: [u_1, \dots, u_n] \triangleq [v_0, u_1, \dots, u_n]$$

$$\frac{M_1 \hookrightarrow M_1'}{\begin{array}{c}\texttt{forNode}(x\colon \texttt{AttributeT}(t)_n = M_1\ \texttt{in}\ M_2)\\ \hookrightarrow \texttt{forNode}(x\colon \texttt{AttributeT}(t)_n = M_1'\ \texttt{in}\ M_2)\end{array}}\ [\![\text{E-ForA}]\!]$$

$$\frac{}{\begin{array}{c}\texttt{forNode}(x\colon \texttt{AttributeT}(t)_n = [v_i{}^{i\in j..k}]\ \texttt{in}\ M_2)\\ \hookrightarrow ([x \mapsto v_j]M_2) :: (\texttt{forNode}(x\colon \texttt{AttributeT}(t)_n = [v_i{}^{i\in j+1..k}]\ \texttt{in}\ M_2))\end{array}}\ [\![\text{E-ForVA}]\!]$$

$$\frac{M \hookrightarrow M'}{\texttt{ifNode}(M,\ M_T,\ M_F) \hookrightarrow \texttt{ifNode}(M',\ M_T,\ M_F)}\ [\![\text{E-If}]\!]$$

$$\frac{}{\texttt{ifNode}(\mathit{true},\ M_T,\ M_F) \hookrightarrow M_T}\ [\![\text{E-IfT}]\!]$$

$$\frac{}{\texttt{ifNode}(\mathit{false},\ M_T,\ M_F) \hookrightarrow M_F}\ [\![\text{E-IfF}]\!]$$

$$\frac{}{\texttt{Box}(M) \hookrightarrow \texttt{Box}(M)}\ [\![\text{E-Box}]\!]$$

$$\frac{M_1 \hookrightarrow M_1'}{\texttt{letbox}\ \ u = M_1\ \ \texttt{in}\ \ M_2 \hookrightarrow \texttt{letbox}\ \ u = M_1'\ \ \texttt{in}\ \ M_2}\ [\![\text{E-LetB}]\!]$$

$$\frac{}{\texttt{letbox}\ \ u = \texttt{Box}(M)\ \ \texttt{in}\ \ M_2 \hookrightarrow [u \mapsto M]M_2}\ [\![\text{E-LetBV}]\!]$$

$$\frac{M \hookrightarrow M'}{M[\mathit{num}] \hookrightarrow M'[\mathit{num}]}\ [\![\text{E-Idx}]\!] \qquad \frac{}{[v_i{}^{i\in 1..p}][\mathit{num}] \hookrightarrow v_{\mathit{num}}}\ [\![\text{E-IdxV}]\!]$$

$$\frac{M \hookrightarrow M'}{M[\mathit{name}]_n \hookrightarrow M'[\mathit{name}]_n}\ [\![\text{E-NApp}]\!] \frac{}{(\Lambda_n n.M)[\mathit{name}]_n \hookrightarrow [\mathit{name}/n]M}\ [\![\text{E-NAppV}]\!]$$

Figure II.13: Evaluation rules (part 3).

$$\frac{M \hookrightarrow M'}{M[type]_t \hookrightarrow M'[type]_t}\ [\![\text{E-TA}\textsc{pp}]\!] \qquad \frac{}{(\Lambda_t t.M)[type]_t \hookrightarrow [type/t]M}\ [\![\text{E-TA}\textsc{pp}\text{V}]\!]$$

$$\frac{M \hookrightarrow M'}{M[row]_r \hookrightarrow M'[row]_r}\ [\![\text{E-RA}\textsc{pp}]\!] \qquad \frac{}{(\Lambda_r r.M)[row]_r \hookrightarrow [row/r]M}\ [\![\text{E-RA}\textsc{pp}\text{V}]\!]$$

$$\frac{M_1 \hookrightarrow M_1'}{M_1 \mathbin{\boldsymbol{.}} M_2 \hookrightarrow M_1' \mathbin{\boldsymbol{.}} M_2}\ [\![\text{E-S}\textsc{el}\text{1}]\!]$$

$$\frac{M_2 \hookrightarrow M_2'}{v_1 \mathbin{\boldsymbol{.}} M_2 \hookrightarrow v_1 \mathbin{\boldsymbol{.}} M_2'}\ [\![\text{E-S}\textsc{el}\text{2}]\!]$$

$$\frac{}{\{L_i = v_i{}^{i \in 1..p}\} \mathbin{\boldsymbol{.}} L_j \hookrightarrow \mathsf{Box}(v_j)}\ [\![\text{E-S}\textsc{el}\text{R}\textsc{ec}]\!]$$

$$\frac{}{\mathtt{Attribute}\langle N, L, B, \{L_i = v_i{}^{i \in 1..p}\}\rangle \mathbin{\boldsymbol{.}} L_j \hookrightarrow \mathsf{Box}(v_j)}\ [\![\text{E-S}\textsc{el}\text{A}\textsc{ttr}]\!]$$

$$\frac{}{\mathsf{Node}\langle \alpha, \{L_i = v_{1_i}{}^{i \in 1..p}\}, [v_{2_j}{}^{j \in 1..m}]\rangle \mathbin{\boldsymbol{.}} L_k \hookrightarrow \mathsf{Box}(v_{1k})}\ [\![\text{E-S}\textsc{el}\text{N}\textsc{ode}]\!]$$

$$\frac{v_1 \underset{R}{\hookrightarrow} v_1'}{v_1 \mathbin{\hat{:}} v_2 \underset{R}{\hookrightarrow} v_1' \mathbin{\hat{:}} v_2}\ [\![\text{E-S}\textsc{el}\text{RT1}]\!]$$

$$\frac{v_2 \underset{R}{\hookrightarrow} v_2'}{v_1 \mathbin{\hat{:}} v_2 \underset{R}{\hookrightarrow} v_1 \mathbin{\hat{:}} v_2'}\ [\![\text{E-S}\textsc{el}\text{RT2}]\!]$$

$$\frac{}{\{L_i = v_i{}^{i \in 1..p}\} \mathbin{\hat{:}} L_j \underset{R}{\hookrightarrow} v_j}\ [\![\text{E-S}\textsc{el}\text{R}\textsc{ec}\text{RT}]\!]$$

$$\frac{}{\mathtt{Attribute}\langle N, L, B, \{L_i = v_i{}^{i \in 1..p}\}\rangle \mathbin{\hat{:}} L_j \underset{R}{\hookrightarrow} v_j}\ [\![\text{E-S}\textsc{el}\text{A}\textsc{ttr}\text{RT}]\!]$$

$$\frac{}{\mathtt{NodeValue}\langle \alpha, \{L_i = v_{1_i}{}^{i \in 1..p}\}, [v_{2_j}{}^{j \in 1..m}]\rangle \mathbin{\hat{:}} L_k \underset{R}{\hookrightarrow} v_{1k}}\ [\![\text{E-S}\textsc{el}\text{N}\textsc{ode}\text{RT}]\!]$$

Figure II.14: Evaluation rules (part 3).

III
Annex III - OSTRICH Benchmark

In this annexe, we introduce all the 10+7 templates used in the evaluation of Core-OSTRICH. These templates belong to the OSTRICH benchmark used to evaluate said language. Each template is stored in a local variable so that each template can be reused inside other templates.

Note that we present the definitions used in our evaluation, which mainly focus on template structure, and some runtime behaviour is abstracted or even absent.

III.1 Attribute Template

```
let attribute =
ForAllName(
    "N",
ForAllRows(
    "R",
ForAllType(
    "T",
Template(
    "e",
    EntityT(VarNT "N", VarR "R"),
Template(
    "attr",
    AttributeT(VarNT "N", VarT "T"),

        LetBox("name",
            NameOf (Var "e"),
            LetBox("label",
                LabelOf (Var "attr"),
                IfNode(
                    IsOfType(Var "attr", BoolT),
                  Node(CheckBox,
                      Record [("Visible", Box( mk_select_list
                                                [VarRT "name"; Label "list";
                                                Label "current"; VarRT "label"] ))
                                                    ],
                      List []
                  ),
                  Node(Expression,
                      Record [("Value", Box( mk_select_list
                                                [VarRT "name"; Label "list";
                                                Label "current"; VarRT "label"] ))
                                                    ],
                      List []
                  )
              )
          )
      )
)
)
)
)
)
```

III.2 Labelled Attribute Template

```
let labelled_attribute =
Let( "attr_templ", attribute,
ForAllName(
    "N",
ForAllRows(
    "R",
ForAllType(
    "T",
Template(
    "e",
    EntityT(VarNT "N", VarR "R"),
Template(
    "attr",
    AttributeT(VarNT "N", VarT "T"),
       Node(Container,
           Record [],
           List [
               Node(Expression,
                   Record [("Value", Select (Var "attr", Label "DisplayName"))
                       ],
                   List[]
               );
               LetBox(
                   "inner_templ",
                   Instantiate( Instantiate( CallType(CallRows(CallName(
                       Var "attr_templ", VarNT "N"), VarR "R"), VarT "T"),
                       Var "e"), Var "attr"),
                   Box(VarRT "inner_templ")
               )
           ]

       )
)
)
)
)
)
)
```

III.3 Filter Template

```
let filter =
Template(
    "attrsInFilter",
    ListAttrT(VarNT "N"),
        Node(Search, Record [("filterBy", Var "attrsInFilter")],
        List [])
)
```

III.4 Listing Template

```
let listing =
Let("filter_templ", filter,
Let("attr_templ", attribute,
ForAllName(
    "N",
ForAllRows(
    "R",
Template(
    "ent",
    EntityT(VarNT "N", VarR "R"),
Template(
    "attrs",
    ListAttrT(VarNT "N"),
Template(
    "showFilter",
    BoolT,
        Node(Container,
            Record [],
            List [
                IfNode(
                    Var "showFilter",
                    LetBox("inner",
                        Instantiate(Var "filter_templ", Var "attrs"),
                        Box(VarRT "inner")
                    ),
                    Node(Empty, Record [], List [])
                );
                Node(List,
                    Record [],
                    List [
                        Node(ListItem,
                            Record [],
                            List [
                                ForNode(
                                    "a",
                                    "aT",
                                    Var "attrs",
                                    LetBox("inner",
                                        Instantiate( Instantiate( CallType(
                                            CallRows( CallName(Var "attr_templ",
                                            VarNT "N"), VarR "R"), VarT "aT" ) ,
                                            Var "ent" ) , Var "a" ),
                                        Box(VarRT "inner")
                                    )
                                )
                            ]
                        )
                    ]
                )
            ]
        )

)
)
)
)
)
)
)
```

III.5 Chart Template

```
let chart =
ForAllName(
    "N",
ForAllRows(
    "R",
ForAllType(
    "T",
Template(
    "e",
    EntityT(VarNT "N", VarR "R"),
Template(
    "categoryAttr",
    AttributeT(VarNT "N", VarT "T"),
        Node(Chart,
            Record [("AttrGroup", Var "categoryAttr")],
            List []
        )
)
)
)
)
)
```

III.6 Pagination Template

```
let pagination =
Box(Node(Pagination, Record [], List []))
```

III.7 Table Template

```
let table =
Let("filter_templ", filter,
Let("attr_templ", attribute,
ForAllName(
    "N",
ForAllRows(
    "R",
Template(
    "e",
    EntityT(VarNT "N", VarR "R"),
Template(
    "attrs",
    ListAttrT(VarNT "N"),
Template(
    "showFilter",
    BoolT,
Template(
    "attrsInFilter",
    ListAttrT(VarNT "N"),
Template(
    "showPagination",
    BoolT,
Template(
    "allowBulk",
    BoolT,
        Node(Container,
            Record [],
            List [
                IfNode(
    Var "showFilter",
    LetBox("inner",
        Instantiate(Var "filter_templ", Var "attrsInFilter"),
        Box(VarRT "inner")
    ),
    Node(Empty, Record [], List [])
);
LetBox("name",
    NameOf (Var "e"),
    Node(Table,
        Record [("Source", Box(Select(VarRT "name", Label "list"
            )))],
        List [
            IfNode(Var "allowBulk",
                Node(Column,
                    Record [("Title", String "Select")],
                    List [
                        Node(CheckBox, Record [], List[])
                    ]
                ),
                Node(Empty, Record [], List [])
            );
            ForNode(
                "a",
                "aT",
                Var "attrs",
                Node(Column,
                    Record [("Title",
                        Select(Var "a", Label "DisplayName"))],
                    List [
                        LetBox("inner",
                            Instantiate(Instantiate( CallType(
                                CallRows( CallName(
                                Var "attr_templ", VarNT "N"),
                                VarR "R"), VarT "aT"), Var "e"),
                                Var "a"),
                            Box(VarRT "inner")
                        )
                    ]
                )
```

```
                            )
                        ]
                    )
                );
                IfNode(Var "showPagination",
                    pagination,
                    Node(Empty, Record [], List [])
                )

            ]
        )
)
)
)
)
)
)
)
)
)
)
```

III.8 Detail Template

```
let detail =
Let("f", labelled_attribute,
ForAllName(
    "N",
ForAllRows(
    "R",
Template(
    "e",
    EntityT(VarNT "N", VarR "R"),
Template(
    "primAttrs",
    ListAttrT(VarNT "N"),
Template(
    "secAttrs",
    ListAttrT(VarNT "N"),
        Node(Container,
          Record [],
          List [
              Node(Column,
                  Record [],
                  List [
                      ForNode(
                          "a1",
                          "a1T",
                          Var "primAttrs",
                          LetBox("lab_attr",
                              Instantiate( Instantiate( CallType( CallRows(
                                  CallName(Var "f", VarNT "N") , VarR "R") ,
                                  VarT "a1T" ) , Var "e" ) , Var "a1" ),
                              Box(VarRT "lab_attr")
                          )
                      )
                  ]
              );
              Node(Column,
                  Record [],
                  List [
                      ForNode(
                          "a2",
                          "a2T",
                          Var "secAttrs",
                          LetBox("lab_attr",
                              Instantiate( Instantiate( CallType( CallRows(
                                  CallName(Var "f", VarNT "N") , VarR "R") ,
                                  VarT "a2T" ) , Var "e" ) , Var "a2" ),
                              Box(VarRT "lab_attr")
                          )
                      )
                  ]
              )
          ]
      )
)
)
)
)
```

```
)
)
```

III.9 List Template

```
let list =
Let(
    "la_templ", labelled_attribute,
Let(
    "a_templ", attribute,
Let(
    "f_templ", filter,
ForAllName(
    "N",
ForAllRows(
    "R",
Template(
    "e",
    EntityT(VarNT "N", VarR "R"),
Template(
    "primAttrs",
    ListAttrT(VarNT "N"),
Template(
    "secAttrs",
    ListAttrT(VarNT "N"),
Template(
    "showFilter",
    BoolT,
        Node(Container,
            Record [],
            List [
                IfNode(
                    Var "showFilter",
                    LetBox("inner",
                        Instantiate(Var "f_templ", Var "primAttrs"),
                        Box(VarRT "inner")
                    ),
                    Node(Empty, Record [], List [])
                );
                Node(List,
Record [],
List [
    Node(ListItem,
        Record [],
        List [
            Node(Container,
                Record [],
                List [
                    ForNode(
                        "a1",
                        "a1T",
                        Var "primAttrs",
                        LetBox("inner",
                            Instantiate(Instantiate(CallType(
                                CallRows(CallName(
                                Var "a_templ", VarNT "N") ,
                                VarR "R") , VarT "a1T" ) ,
                                Var "e" ) , Var "a1" ),
                            Box(VarRT "inner")
                        )
                    )
                ]
            );
            Node(Container,
                Record [],
                List [
                    ForNode(
                        "a2",
                        "a2T",
                        Var "secAttrs",
                        LetBox("inner",
                            Instantiate(Instantiate(CallType(
                                CallRows(CallName(
                                Var "la_templ", VarNT "N") ,
                                VarR "R") , VarT "a2T" ) ,
                                Var "e" ) , Var "a2" ),
                            Box(VarRT "inner")
                        )
                    )
                ]
```

```
                                    )
                                ]
                            )

                        ]
                    )
                ]
            )
)
)
)
)
)
)
)
)
)
```

III.10 List with Chart Template

```
let list_with_chart =
Let("chart_templ", chart,
Let("listing_templ", listing,
ForAllName(
    "N",
ForAllRows(
    "R",
ForAllType(
    "T",
Template(
    "e",
    EntityT(VarNT "N", VarR "R"),
Template(
    "attrs",
    ListAttrT(VarNT "N"),
Template(
    "categoryAttr",
    AttributeT(VarNT "N", VarT "T"),
Template(
    "showFilter",
   BoolT,
      Node(Container,
         Record [],
         List [
            Node(Column,
               Record [],
               List [
                  LetBox("inner",
                     Instantiate( Instantiate( CallType( CallRows(
                        CallName(Var "chart_templ", VarNT "N") ,
                        VarR "R") , VarT "T" ) , Var "e" ) ,
                        Var "categoryAttr" ),
                     Box(VarRT "inner")
                  )
               ]
            );
            Node(Column,
               Record [],
               List [
                  LetBox("inner",
                     Instantiate (Instantiate( Instantiate( CallRows(
                        CallName(Var "listing_templ", VarNT "N") ,
                        VarR "R") , Var "e" ) , Var "attrs" ),
                        Var "showFilter"),
                     Box(VarRT "inner")
                  )
               ]
            )
         ]
      )
)
)
)
)
)
)
)
)
)
```

III.11 Dashboard Template

```
let dashboard =
Let("listing_templ", listing,
Let("chart_templ", chart,
ForAllName(
    "N",
ForAllRows(
    "R",
ForAllType(
    "statusT",
ForAllType(
    "categT",
Template(
    "e",
    EntityT(VarNT "N", VarR "R"),
Template(
    "attrs",
    ListAttrT(VarNT "N"),
Template(
    "statusAttr",
    AttributeT(VarNT "N", VarT "statusT"),
Template(
    "categoryAttr",
    AttributeT(VarNT "N", VarT "categT"),
        Node(Container,
            Record [],
            List [
                Node(Container,
                    Record [],
                    List [
                      Node(Counter,
                          Record [("Source", Var "statusAttr")],
                          List []
                      )
                    ]
                );
                Node(Container,
                    Record [],
                    List [
                        Node(Column,
                            Record [],
                            List [
                                LetBox("inner",
                                    Instantiate( Instantiate( Instantiate(
                                        CallRows( CallName( Var "listing_templ",
                                            VarNT "N") , VarR "R") , Var "e" ) ,
                                        Var "attrs" ) , Bool false ),
                                    Box(VarRT "inner")
                                )
                            ]
                        );
                        Node(Column,
                            Record [],
                            List [
                                LetBox("inner",
                                    Instantiate( Instantiate( CallType( CallRows(
                                        CallName( Var "chart_templ", VarNT "N") ,
                                        VarR "R") , VarT "categT" ) , Var "e" ) ,
                                        Var "categoryAttr" ),
                                    Box(VarRT "inner")
                                )
                            ]
                        )
                    ]
                )
            ]
        )
)
)
)
)
)
)
)
)
)
)
```

III.12 Account Dashboard Template

```
let account_dashboard =
Let("la_templ", labelled_attribute,
Let("chart_templ", chart,
Let("listing_templ", listing,
ForAllName(
    "masterN",
ForAllName(
    "detailN",
ForAllRows(
    "masterR",
ForAllRows(
    "detailR",
ForAllType(
    "categT",
Template(
    "masterEnt",
    EntityT(VarNT "masterN", VarR "masterR"),
Template(
    "masterAttrs",
    ListAttrT(VarNT "masterN"),
Template(
    "detailEnt",
    EntityT(VarNT "detailN", VarR "detailR"),
Template(
    "detailAttrs",
    ListAttrT(VarNT "detailN"),
Template(
    "rollingSumAttr",
    AttributeT(VarNT "masterN", NumT),
Template(
    "categoryAttr",
    AttributeT(VarNT "detailN", VarT "categT"),
        Node(Container,
            Record [],
            List [
                Node(Container,
                    Record [],
                    List [
                        ForNode(
                            "ma",
            "maT",
            Var "masterAttrs",
            Node(Column,
                Record [],
                List [
                    LetBox("inner",
                        Instantiate(Instantiate(CallType(CallRows
                            (
                            CallName(Var "la_templ",
                            VarNT "masterN"), VarR "masterR"),
                            VarT "maT"), Var "masterEnt"),
                            Var "ma"),
                        Box(VarRT "inner")
                    )
                ]
            )
        )
    ]
);
Node(Container,
    Record [],
    List [
        Node(Column,
            Record [],
            List [
                LetBox("inner",
                    Instantiate(Instantiate(Instantiate(CallRows(
                        CallName(Var "listing_templ",
                        VarNT "detailN"), VarR "detailR"),
                        Var "detailEnt"), Var "detailAttrs"),
                        Bool false),
                    Box(VarRT "inner")
                )
            ]
        );
        Node(Column,
            Record [("Title",
                Select(Var "categoryAttr", Label "DisplayName"))
                    ],
            List [
                                LetBox("inner",
                                    Instantiate(Instantiate(CallType(
                                        CallRows(CallName(Var "chart_templ",
                                        VarNT "detailN"), VarR "detailR"),
                                        VarT "categT"), Var "detailEnt"),
                                        Var "categoryAttr"),
                                    Box(VarRT "inner")
                                )
                            ]
                        )
                    ]
                )
            ]
        )
)
)
)
)
)
)
)
)
)
)
)
)
)
)
```

III.13 Master Detail Template

```
let master_detail =
Let("listing", listing,
Let("detail", detail,
ForAllName(
    "N",
ForAllRows(
    "R",
Template(
    "entity",
    EntityT(VarNT "N", VarR "R"),
Template(
    "masterAttrs",
    ListAttrT(VarNT "N"),
Template(
    "detailAttrs",
    ListAttrT(VarNT "N"),
Template(
    "showFilter",
    BoolT,
        Node(Container,
            Record [],
            List [
                Node(Column,
                    Record [],
                    List [
                        LetBox("inner",
                            Instantiate(Instantiate(Instantiate(CallRows(
                                CallName(Var "listing", VarNT "N"), VarR "R"),
                                Var "entity"), Var "masterAttrs"),
                                Var "showFilter"),
                            Box (VarRT "inner")
                        )
                    ]
                );
                Node(Column,
                    Record [],
                    List [
                        LetBox("inner",
                            Instantiate(Instantiate(Instantiate(CallRows(
                                CallName(Var "detail", VarNT "N"), VarR "R"),
                                Var "entity"), Var "masterAttrs"),
                                Var "detailAttrs"),
                            Box (VarRT "inner")
                        )
                    ]
                )
            ]
        )
)
)
)
)
)
)
)
)
```

III.14 Four-Column Gallery Template

```
let four_col =
Let("listing", listing,
Let("filter", filter,
ForAllName(
    "N",
ForAllRows(
    "R",
Template(
    "e",
    EntityT(VarNT "N", VarR "R"),
Template(
    "attrs",
    ListAttrT(VarNT "N"),
Template(
    "showFilter",
    BoolT,
Template(
    "attrsInFilter",
    ListAttrT(VarNT "N"),
Template(
    "showPagination",
    BoolT,
        Node(Container,
            Record [],
            List [
                Node(Column,
                    Record [],
                    List [
                        LetBox("inner",
                            Instantiate(Instantiate(Instantiate(CallRows(
                                CallName(Var "listing", VarNT "N"), VarR "R"),
                                    Var "e"), Var "attrs"), Bool false),
                            Box(VarRT "inner")
                        );
                        IfNode(
                            Var "showPagination",
                            pagination,
                            Node(Empty, Record [], List [])
                        )
                    ]
                );
                Node(Column,
                    Record [],
                    List [
                        IfNode(
                            Var "showFilter",
                            LetBox("inner",
                                Instantiate(Var "filter", Var "attrsInFilter"),
                                Box(VarRT "inner")
                            ),
                            Node(Empty, Record [], List [])
                        )
                    ]
                )
            ]
        )
)
)
)
)
)
)
)
)
)
```

III.15 Admin Dashboard Template

```
let admin_dashboard =
Let("table_templ", table,
ForAllName(
    "N",
ForAllRows(
    "R",
ForAllType(
    "T",
Template(
    "e",
    EntityT(VarNT "N", VarR "R"),
Template(
    "attrs",
    ListAttrT(VarNT "N"),
Template(
    "statusAttr",
    AttributeT(VarNT "N", VarT "T"),
        Node(Container,
            Record [],
            List [
                Node(Counter,
                    Record [("Source", Var "statusAttr")],
                    List []
                );
                LetBox("inner",
                    Instantiate(Instantiate(Instantiate(Instantiate(
                        Instantiate(Instantiate(CallRows(CallName(
                        Var "table_templ", VarNT "N"), VarR "R"), Var "e"),
                        Var "attrs"), Bool false), Var "attrs"), Bool false),
                        Bool false),
                    Box(VarRT "inner")
                )
            ]
        )
)
)
)
)
)
)
)
```

III.16 List with Filters Template

```
let list_with_filters =
Let("table_templ", table,
ForAllName(
    "N",
ForAllRows(
    "R",
Template(
    "e",
    EntityT(VarNT "N", VarR "R"),
Template(
    "attrs",
    ListAttrT(VarNT "N"),
Template(
    "showFilter",
    BoolT,
Template(
    "attrsInFilter",
    ListAttrT(VarNT "N"),
Template(
    "showPagination",
    BoolT,
        Node(Container,
            Record [],
            List [
                LetBox("inner",
                    Instantiate(Instantiate(Instantiate(Instantiate(
                        Instantiate(Instantiate(CallRows(CallName(
                        Var "table_templ", VarNT "N"), VarR "R"), Var "e"),
                        Var "attrs"), Var "showFilter"), Var "attrsInFilter"),
                        Var "showPagination"), Bool false),
                    Box(VarRT "inner")
                )
            ]
        )
)
)
)
)
)
)
)
)
```

III.17 Bulk Actions with Filters Template

```
let bulk_actions =
Let("table_templ", table,
ForAllName(
    "N",
ForAllRows(
    "R",
Template(
    "e",
    EntityT(VarNT "N", VarR "R"),
Template(
    "attrs",
    ListAttrT(VarNT "N"),
Template(
    "showFilter",
    BoolT,
Template(
    "attrsInFilter",
    ListAttrT(VarNT "N"),
Template(
    "showPagination",
    BoolT,
        Node(Container,
            Record [],
            List [
                LetBox("inner",
                    Instantiate(Instantiate(Instantiate(Instantiate(
                        Instantiate(Instantiate(CallRows(CallName(
                        Var "table_templ", VarNT "N"), VarR "R"), Var "e"),
                        Var "attrs"), Var "showFilter"), Var "attrsInFilter"),
                        Var "showPagination"), Bool true),
                    Box(VarRT "inner")
                )
            ]
        )
)
)
)
)
)
)
)
)
```