NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

CONSTANÇA MARIA MANSO RIBEIRO AMADOR MANTEIGAS

BSc in Computer Science

CUSTOMIZABLE TEMPLATES FOR OUTSYSTEMS APPLICATIONS

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
December, 2022

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

CUSTOMIZABLE TEMPLATES FOR OUTSYSTEMS APPLICATIONS

CONSTANÇA MARIA MANSO RIBEIRO AMADOR MANTEIGAS

BSc in Computer Science

Adviser: João Costa Seco
Associate Professor, NOVA University Lisbon

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

Examination Committee

Chair: Pedro Abílio Duarte de Medeiros
Associate Professor, NOVA University Lisbon

Members: Francisco Cipriano da Cunha Martins
Associate Professor, Açores School of Science and Technology

João Costa Seco
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
December, 2022

Customizable Templates for OutSystems Applications

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.2) [14].

Abstract

This dissertation addresses an extension of a metaprogramming mechanism in low-code platforms, specifically in the OutSystems platform. As proposed in the template language OSTRICH, model templates allow developers to reuse existing and thoroughly tested code fragments in a more productive and sound development process. Code templates help to overcome developer difficulties and lack of training. For instance, the development of professionally designed user interfaces is not a skill that is common amongst developers. Scenarios like specific synchronization algorithms are the other end of the spectrum for templates in OutSystems. The GOLEM project has the chief objective of providing mechanisms for automated programming. Such automation will make programming more accessible to a larger community of developers. This work is part of those efforts.

To this end, our focus will be on developing and improving the template language OSTRICH that targets the OutSystems platform. It enables the creation of code fragments like screen templates, saving the user from the cumbersome task of repeatedly constructing code complex patterns. In OSTRICH, templates are instantiated by expanding their definition in the caller context to enable further customization of the resulting code. OSTRICH preserves the structure of the original model by expanding templates in place. However, expanding the template definition in place breaks the possibility of reapplying the template in the case of an update to a newer version or changing parameters. It requires the user to repeat all customization operations on top of a new instantiation of the template. Our purpose is to solve this issue by supporting customization operations that extend to all future template updates and new parameters. This way, the user will be saved from some erroneous trials, which could, eventually, push them to give up. We plan to evaluate our work using the benchmark template examples used previously to evaluate OSTRICH.

Keywords: Functional Programming, Low Code Development, Templating Language, Staged Computation, Modal Logic

Resumo

Esta dissertação aborda uma extensão de um mecanismo de metaprogramação em plataformas *low-code*, especificamente na plataforma da OutSystems. Como proposto na linguagem de *template* OSTRICH, os *templates* permitem que os utilizadores reutilizem fragmentos de código existentes e completamente testados num processo de desenvolvimento produtivo e seguro. Estes fragmentos ajudam os utilizadores a ultrapassar as dificuldades ou falta de conhecimento em linguagens de programação. Por exemplo, a sua utilização permite criar interfaces de forma profissional, o que nem sempre é uma habilidade comum entre desenvolvedores de *software*. Do outro lado do espetro, estão os algoritmos de sincronização específicos para modelos em OutSystems. O projeto GOLEM tem como principal objetivo fornecer mecanismos para automatizar a programação. Esta automatização irá tornar a programação mais acessível a um maior número de pessoas. Este trabalho faz parte destes esforços.

Para alcançar este objetivo, estaremos focados no desenvolvimento e melhoria da linguagem de *template* OSTRICH para a plataforma da OutSystems. Esta permite a criação de fragmentos de código como *screen templates*, poupando os utilizadores da tarefa complicada de repetir padrões complexos de código. Nesta linguagem, os *templates* são instanciados através da expansão da sua definição para permitir a customização do código resultante. Contudo, expandir esta definição no local quebra a possibilidade de reaplicar o *template* em caso de atualização para uma versão mais recente ou mudança de parâmetros. Isto acaba por obrigrar o utilizador a repetir todas as operações aplicadas numa nova instanciação de *template*. O nosso propósito passa por resolver este problema, permitindo que as operações do utilizador possam ser reaplicadas em todas as novas atualizações do *template* e novos parâmetros. Desta forma, o utilizador será poupado de seguir uma aprendizagem tentativa e erro, que poderia eventualmente levá-lo a desistir. Planeamos avaliar o nosso trabalho utilizando como referência exemplos de *templates* usados anteriormente para avaliar a linguagem OSTRICH.

Palavras-chave: Programação Funcional, Desenvolvivemto *Low Code*, Linguagens de *Template*, Computação por Etapas, Lógica Modal

Contents

List of Figures

List of Tables

1 Introduction

1.1 Context

Now, more than ever before, the Internet plays a role in people's life. Not only do we use it to read the news or research some article, but also for trivial tasks, such as looking for a restaurant menu or booking a reservation in a gym space. This growing use of websites and applications, which only kept rising with the pandemic and the limitations it brought to everyone's routine, strengthens a new reality: it is increasingly necessary that companies and businesses have an online presence. But despite this being a simple concept, its implementation is not within reach of everyone, with the lack of programming knowledge arising with the growing demand for them. This problem is the main incentive for low-code platforms: how can we work around the long and tedious process of creating applications and make them with a few programming skills?

Customers adopt the OutSystems' low-code platform for several purposes, such as creating applications and Modules on the server, creating user interfaces for Traditional Web, Reactive Web, and Mobile Applications, or defining data models [7]. For each of these, there are distinct domain-specific languages (DSLs). DSLs are used instead of General Purpose Languages (GPLs) to address the specific needs of the domain it applies. Its use has some advantages. To name a few, it is less time-consuming than GPLs since the developer does not have to study concepts unrelated to the domain, and DSLs provide more appropriate modeling abstractions and primitives closer to the ones used in it.

OSTRICH is a safe templating language intended to be used in the OutSystems' low-code platform [13]. This platform is a visual model-driven development (MDD), a methodology that uses models as the primary artifact of the development process, and the code is automatically generated from them [1]. In MDD, model-driven techniques are consistently applied to automate as much as possible of the software lifecycle from the requirements down to the deployed application.

Adopting a model-driven approach is essential. Not only does it allow users the possibility of building complex and faultless software through an easily readable interface where pre-built components are available, but it also facilitates communication among all

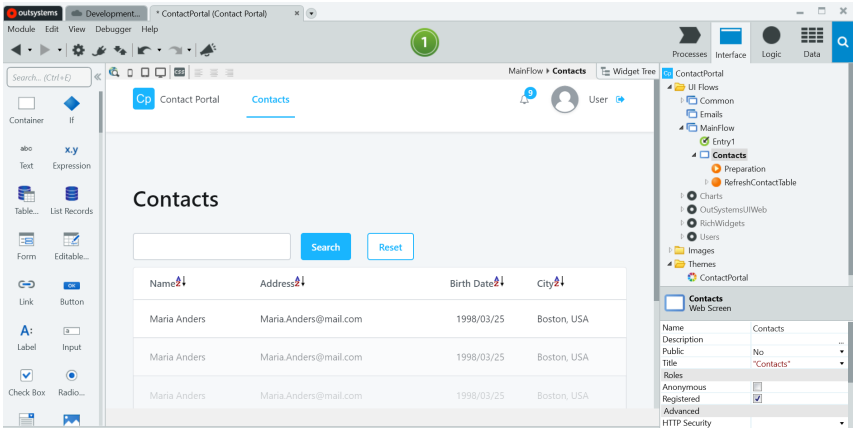


Figure 1.1: A glimpse of the OutSystems' platform.

stakeholders over its understandable structure.

MDD is the most crucial principle of low-code development, an approach that requires minimal to no coding to build applications and processes. To this end, drag-and-drop capabilities are used as well as visual logical interfaces, as previously mentioned, instead of complex programming languages.

The topic of this work is integrated within the GOLEM project, which is a project led by OutSystems and done in partnership with Carnegie Mellon University, NOVA LINCS, and INESC-ID [20]. It is estimated to last three years and has a goal of answering the following three questions: (1) What is the next generation of low-code?; (2) What is the most natural way for a citizen developer to create fit-to-purpose apps without having to write code?; (3) How can such a visionary development experience be implemented?

1.2 Motivation and Contributions

The OutSystems platform has several screen templates available. These allow for the creation of screens with a predefined layout. The layout comes with sample data, which helps the user understand how the screen is designed. These templates can be one of many, such as a gallery or a list of products, and are the direct motivation in the design of OSTRICH [13].

Its use will change some aspects of the platform as it is known today. As an example, and as we can verify in Figure 1.1, the template instantiation comes with dummy data, forcing the user to adapt it to their database entries. When using OSTRICH, this will no longer be required since the developer will have the possibility to manipulate the entries from the beginning.

Apart from that, the changes performed in the application after the template instantiation are not tracked. This way, if the application on the template evolves in a divergent manner, after reaching some point, the user would have to repeat the same changes on the updated template to reach that point. OSTRICH will also solve this issue by keeping

a record of the manual customizations applied to the template.

1.3 Document Structure

This document is composed of four more chapters:

1. Chapter 2, in which the focus is on the core concepts regarding this subject of study, starting with some theory notions and the foundation of functional programming, λ-calculus, and ending with the work already developed in OSTRICH language;
2. Chapter 3 presents related work that encompasses MetaML, Template Haskell, and type systems motivated logically, namely modal λ-calculus and modal Mini-ML;
3. Chapter 4 explains the different customizations we can apply to a template. We carefully studied how they are evaluated and how we make sure they are only applied if it is possible;
4. Chapter 5 describes the type system;
5. Chapter 6 exposes the code implementation and some testing examples;
6. Chapter 7 briefly covers what we have been studying throughout this thesis and what could be done to improve this work.

2

Background

This chapter will cover three main topics divided into three sections: key concepts (Section 2.1), functional programming and λ-calculus (Section 2.2), and OSTRICH (Section 2.3).

The first one will explain five main concepts relevant to OSTRICH: the meaning of a template language and what defines it; type safety; stages and phase distinction. It will also include a section of partial evaluation, as it is necessary to have a firm understanding of the related work studied in Chapter 3.

The second one will explore functional programming, λ-calculus, and its properties. λ-calculus is important as OSTRICH is an extension of it by adding new abstractions to λ-calculus and keeping its code patterns.

The final and third section exposes the work already done in OSTRICH.

2.1 Key Concepts

OSTRICH is a template language built on metamodel annotations. Therefore, it is necessary to get a good grasp of template languages and metamodels. Furthermore, some other properties, such as type safety and phase distinction, both guaranteed by OSTRICH, will be studied.

The notion of partial evaluation will be introduced, as it is strongly related to the work presented in Chapter 3.

2.1.1 Template Languages

A metamodel is an abstraction of a model [1]. A model is a partial representation of some reality to make predictions or inferences. By applying abstraction, the details are hidden, and only the core concepts are covered. Thus, for a model to exist, three aspects have to happen: (1) there is an original object or phenomenon that is mapped to the model; (2) the model is a representation of this original; (3) the model fulfills some purpose. This purpose can either be descriptive or prescriptive [6]. The former is used to describe a reality of a system, while the latter is used to determine the scope and details of the problem.

Software development is increasingly relying on models. This increase is happening since software artifacts are becoming more complex and, consequently, need to be discussed at different levels of abstraction. As this discussion is not exclusive to software developers, the model needs to be made explicit and communicated in a language that can be understood by all stakeholders. This caution is taken into account with descriptive models as discussion and analysis are their principal purpose [27]. However, this communication is not the only intent of models, as they are also used to generate code and tests, which is the essence of model-driven development. For this, prescriptive models are required, which need to be more rigorous, formal, complete, and consistent than descriptive models.

The languages that define a set of models used for a specific purpose are called template languages. The definition of these languages comprises three aspects: (1) the syntax, which describes how its models appear; (2) the semantics, which describes the meaning of each model; (3) its pragmatics, which describes how to use the models according to their purpose.

The syntax of programming languages has two levels of structure: the concrete syntax and the abstract syntax [19]. The former refers to the strings of characters the programmer writes or whatever the user interacts with to create programs - it may be textual, graphical as in template languages, tabular, or any combination of these. The latter is a simple representation of programs as labeled trees, called abstract syntax trees, that hold the core information.

An advantage of having the concrete syntax as graphical over textual is that it allows the user to overview the different relationships between the model elements. However, it takes more space to convey the same content.

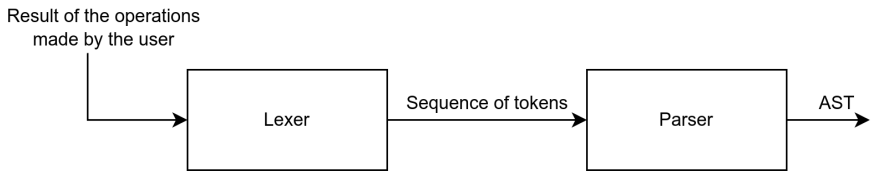
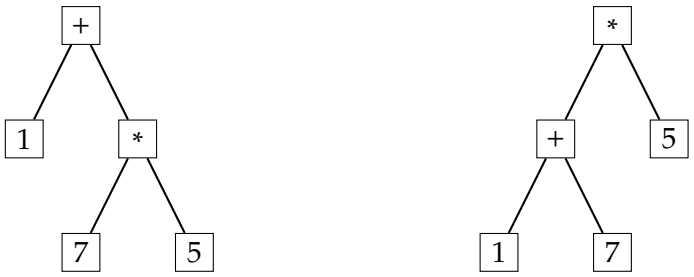


Figure 2.1: Transformation from concrete to abstract syntax.

Figure 2.1 represents the transformation from concrete to abstract syntax. Briefly, a lexical analyzer receives the string of characters written by the programmer (in textual concrete syntax) or the result of the operations available in the user interface (in graphical concrete syntax) as input and outputs a sequence of tokens as a result, such as identifiers or keywords. During this process, the lexer removes comments and resolves other issues, such as whitespaces. Afterward, a parser receives this output and transforms it into an abstract syntax tree.

During parsing, several conventions take place. For example, associativity and precedence reduce the need to clutter programs with parentheses to explicitly indicate the structure of an expression, and if one writes $1 + 7 * 5$ the resulted tree would be the one

on the left rather than the one on the right:



2.1.2 Type Safety

Languages can be one of two kinds: typed or untyped [3]. Typed languages are the ones where variables can assume a range of values. For instance, the variable `x` can be a `Boolean`, and the variable `y` can be an `Integer`. On the other hand, untyped languages do not have this kind of distinction, and all variables lack a type, or they all have the same one. Examples of this are sets in set theory or bit strings in computer memory.

The move from untyped to typed comes with the need to categorize objects according to their usage and behavior and prevent interaction between some of them. Types ensure this since they impose constraints which, in its turn, imposes correctness [4].

There are still a lot of discussions about whether typed languages are better or not than untyped languages. However, there is little space for doubt that the maintenance of untyped languages is a challenging task. Some advantages of having types, and therefore, reasons that make maintenance of untyped languages difficult, are: (1) economy of execution - the information about types at compile time leads to the appropriate operations at run time without the need of expensive tests; (2) economy of development and maintenance in security areas - the casting of some variable, for example, from type `Integer` to a value of pointer type can compromise the system and allow the access of an attacker to the whole system. The most cost-effective way to solve these issues but not eliminate them is to employ typed languages.

A type system is the component of type languages that keeps track of the variables and expressions types presented in a program. Directly, the purpose of it is to prevent the occurrence of execution errors upon the running of the program [3].

These execution errors can vary. There are the so-called obvious software errors that type systems prevent, such as an illegal memory reference fault, but there are also faults that usually can not be. Dividing a number by 0 is an example of this.

When looking more carefully into the execution errors, we can find two distinct classes: the ones that cause a program to crash immediately and the ones that can go unnoticed and sometimes cause different behavior from the one expected. The first ones are called trapped errors, and the last ones are untrapped errors.

In the absence of untrapped errors, a program fragment is called safe. Languages that have all program fragments safe are named safe languages.

A type system has some expected proprieties: (1) it should be verifiable - the typechecking algorithm should be able to ensure the good behavior of the program by capturing execution errors before they can occur; (2) it should be transparent - it should be easy to predict if a program will typecheck, and if not, the reason should be clear to spot; (3) it should be enforceable - type declarations should be either statically or dynamically checked.

With these expected properties, there are a few advantages when using type systems: (1) the errors can be early detected, which also means they can be fixed immediately rather than when the program is being, for example, deployed; (2) the program becomes more easily maintained, as all it takes to switch a type is change the declaration of that datatype, run the compiler and examine the points where typechecking fails; (3) it enforces disciplined programming through the definition of modules with clear interfaces, which by its turn lead to a more abstract style of design, where interfaces can be discussed apart from their implementation; (4) types turn the reading of a program more understandable and can be seen as a form of documentation; (5) more efficient programs, as first mentioned.

When static program analysis can determine the type of every expression and variable, the programming language is said to be statically typed. Static typing not only enforces a more structured and easy-to-read program but also discovers errors in an early stage and allows greater execution-time efficiency. Although this is useful, it is also too restrictive since it requires all variables to be associated with a type at compile-time.

To work around this, ensuring that all variables and expressions are type-consistent, even if the type is statically unknown, lifts some of these restrictions. This type-consistency can be arranged by introducing some run-time type checking. Languages in which all expressions are typed consistent are strongly typed languages [4].

2.1.3 Stages and Phase Distinction

As the name implies, staging is a program transformation that involves restructuring the program execution into stages. In compiled languages, there are two separated stages upon the execution of executing a program: compile-time and run-time. In program generation, there are three different stages: generation, compilation, and execution.

A multi-stage program involves the three phases mentioned in program generation, with all of them occurring inside the same process. Multi-stage languages express multi-stage programs [26].

Staged programs aim to produce efficient programs through the control over resources, both in terms of space and time [23]. To control the resources, programmers use program annotations to control the order of evaluation, which influences the number of steps in a reduction. By its turn, the number of steps relates to the resources consumed, so controlling it also translates to the control of resources.

According to this order of evaluation, most functional programming languages can be classified as either eager (also known as strict) or lazy. Section 2.2 will discuss both of these concepts.

The execution of a program is achieved in two distinct phases: a typechecking phase (compile-time) and an execution phase (run-time). In some languages, there is no clear separation between these phases, as in Pebble, while the difference is clear [2] in others, as in Pascal.

When using dependent types, both phases can get mixed. Dependent types are types that depend on the evaluation of some expression. Carefulness is required to distinguish between compile-time and run-time values to avoid this mix. OSTRICH respects phase distinction as it avoids dependencies between compile-time and run-time expressions.

2.1.4 Partial Evaluation

Partial evaluation is a technique to evaluate a program, as the name indicates, partially when some of its inputs are available. The goal of it is to generate efficient programs from a more general one by completely automating methods [12]. The control flow of partial evaluation is presented in Figure 2.2.

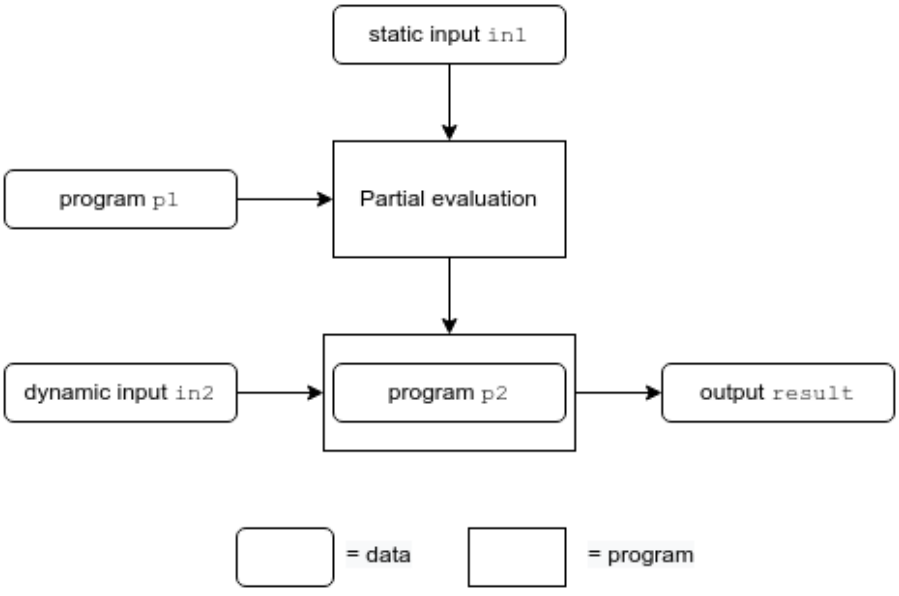


Figure 2.2: A partial evaluation.

Considering a program `power` that receives two arguments dynamically, `x` and `n` at the same time, the program can only be executed at run-time. The program would be as follows:

```
power(int n, int x) {
  int result = 1;
  while (n > 0) {
    if (n % 2 == 0) {
```

```
      x = x * x;
      n = n / 2;
    } else {
      result = result * x;
      n = n -1;
    }
  return result;
}
```

If, on the contrary, one input is available from the beginning, `n`, the program can be partially evaluated, producing a different version of the same program. For example, if we apply `n = 5`, the newly executed program would be:

```
power5(int x) {
  int result = 1;
  result = result * x; // n = 5
  x = x * x; // n = 4
  x = x * x; // n = 2
  result = result * x; // n = 1
  return result;
}
```

Partial evaluation is used to speed up the evaluation process and can be applied to a wide range of problems by first solving them indirectly using static information and then running the generated program on various data supplied dynamically.

2.2 Functional programming and λ-calculus

This section will present the main properties of functional programming languages, such as how the programs are evaluated, and the fundamental aspects of λ-calculus.

Since OSTRICH is a functional programming language, it is essential to understand the primary concepts of the paradigm. We will introduce λ-calculus based on its simplicity and expressivity. These two characteristics make it possible to express all functional programs.

2.2.1 Functional Programming

Functional programming has two core concepts: (1) functions are pure, and (2) there is support for first-class functions.

The first one means that the output only depends on the input, and a function upon receiving the same input as before has to output the same result. Apart from that, functions have no side effects, such as changing global variables or writing to disk. An example of a non-functional function, `incCounter1`, and a functional one, `incCounter2`, are displayed below:

```
var counter = 1;
function incCounter1() {
  counter = counter + 1;
  return counter;
}
function incCounter2(counter) {
  return counter + 1;
}
```

As it can be verified, the function `incCounter1` changes the global variable `counter`, and will never output the same result. On the other hand, the function `incCounter2` does not have this side effect.

The second concept of functional programming means that functions behave like any other data structure. That is, they can be assigned to variables or passed as arguments to other functions, among others, as shown in the following example:

```
var inc = function(counter) { return counter + 1; }
```

The principal advantage of functional programming languages, such as Haskell or OCaml, compared to imperative programming ones, such as Java or C, is having no side effects. Thus it is not necessary to establish an execution order between functions, which allows them to be safely executed in parallel, as they can not change the state of the other. Moreover, keeping track of these states make imperative programs harder to read and compile, as one needs to know their context.

Functional programming has seen an increase in terms of popularity and influence. It has changed the way people construct, verify and think about programs [11]. However, its adoption still faces a few obstacles. Most of these obstacles are social and commercial, rather than technical: they usually fall short in terms of the richness of library packages or debugging tools, and most programmers are trained in an imperative style [21].

Functional programming languages use either lazy evaluation or eager evaluation. Examples of languages that use lazy evaluation are Haskell and Miranda. OCaml and Standard ML use eager evaluation. The prime difference between these two types of evaluation is the delay of some computations. For instance, a function can have an unused parameter, and, in that case, it will not be necessary to evaluate that expression.

```
function add(x, y) {
  return x + 1;
}
add(1,10+13)
```

In the example above, since the value of the parameter `y` is not used, from the perspective of lazy evaluation, there is no need to evaluate `10+13`. Evaluations will only occur when they are needed. On the contrary, eager evaluation will compute these expressions for all cases.

Lazy evaluation can have a few advantages over eager evaluation: it is more efficient since it only evaluates once it is required - for this, it will use memoization, which can be

seen as a dictionary that binds variables to their values; there is no reason to hard-code some behaviors into compilers, such as short-circuiting - if there is a statement with a logical AND between two sides, the right-hand side will not be looked at, if the left-hand is evaluated to `false`; lazy evaluation can work with infinite lists, unlike eager evaluation that would run out of memory.

2.2.2 λ-calculus

In λ-calculus, everything is a function, and everything can be expressed and evaluated as one [15][19]. The core concept around this language is λ, and there are three kinds of λ expressions:

1. Individual variables: they can be represented by any sequence of non-blank characters, such as x, y,...;
2. Applications, $(e_1 e_2)$: this represents the application of e_1 to e_2;
3. Abstractions, $(\lambda x.e)$: this represents the function which returns the value e when given formal parameter x. The "." separates the name from the expression in which the abstraction with that name takes place. It is important to notice that the expression can be any λ expression, including another function.

The syntax of λ-calculus can thus be summarized as it follows (t represents a term and we can see it as being the same as the mentioned expression e, and v represents a value):

$$t ::= x \mid \lambda x.t \mid t\ t$$
$$v ::= \lambda x.t$$

The operational semantics of λ-calculus are the following:

$$(\lambda x.t)v \rightarrow \{x \mapsto v\}t \quad E\_Beta \qquad \frac{t_1 \rightarrow t_1'}{t_1 t_2 \rightarrow t_1' t_2}\ E\_App1 \qquad \frac{t_2 \rightarrow t_2'}{v_1 t_2 \rightarrow v_1 t_2'}\ E\_App2$$

The rules are very self-explanatory and focus on substituting arguments with variables in expressions. However, there are a few things one needs to be aware of when making substitutions, which we will show right away. Before that, there are syntactic conventions worth displaying to make a program easier to read:

1. $\lambda x.xy$ means $\lambda x.(xy)$ and not $(\lambda x.x)\ y$;
2. $\lambda x_1.\lambda x_2....\lambda x_n.e$ is equivalent to $\lambda x_1.(\lambda x_2....(\lambda x_n.e))$;
3. Application associates to the left, implying that xyz represents $(xy)z$ and not $x(yz)$.

Moreover, it is also worth explaining the scope of a variable. A variable x is said to be bound if its occurrence in the body of the definition is preceded by λx. For example, in the function:

$\lambda x.xy$

The variable x is bound and y is free. Considering a more complex function:

$(\lambda x.x)\ (\lambda y.yx)$

The first x is bound to the first λ, but the second x is free.

Considering the abstractions represented in λ-calculus, the most simple one is the identity function. It is usually found in the following format:

$$f(x) = x$$

In λ-calculus, it would appear as:

$$\lambda x.x$$

This function has the bound variable x and has the expression body x. When it is used as the function expression in a function application, the argument expression replaces the variable x. For example, applying the identity function to itself would result in:

$$(\lambda x.x\ \lambda x.x)$$

$$= (\lambda x.x\ \lambda x.x)$$

$$= \lambda x.x$$

As verified, the argument expression replaced the bound variable x in the body expression x, giving the result presented above.

When, in an expression, there are free and bound variables, carefulness is required to avoid mixing both, as already mentioned. The following example illustrates this issue more clearly:

$$(\lambda x.\lambda y.xy)\ y$$

$$= (\lambda x.(\lambda y.xy))\ y$$

$$= (\lambda x.(\lambda y.x\ y))\ y$$

$$= \lambda y.yy$$

At first, the variable y at the right side was free, while the first y was bound. However, the replacement in the expression mixed both identifiers, giving an erroneous result. Renaming the argument y to z would solve that issue:

$$(\lambda x.x\lambda y.xy)\ z$$

$$= \lambda y.zy$$

Another essential aspect to keep in mind is how, in λ-calculus, each function takes exactly one parameter at a time, as it was already shown. The concept of receiving only one argument at the original function and returning nested functions that receive the other arguments is called currying and is rigorously used by λ-calculus. The table below shows an example comparing currying with no currying:

No currying	Currying

`function addBasic(x,y) {`
`  return x + y;`
`}`
`addBasic(1,2);`
	`function addCurry(x) {`
	`  return function(y) {`
	`  return x+ y;`
	`  }`
	`}`
	`addCurry(1)(2);`

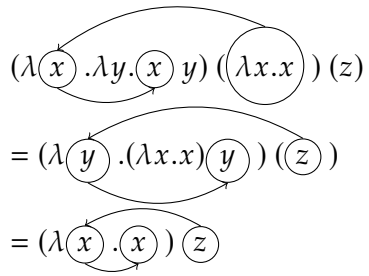
Before introducing types, we show a more complex example below:

$(\lambda x.\lambda y.xy)\ (\lambda x.x)\ (z)$

$= (\lambda y.(\lambda x.x)y)\ (z)$

$= (\lambda x.x)\ z$

$= z$



$= z$

So far, the λ-calculus presented does not have types, but they can also be included in the language to form the most elementary member of the family of type systems.

If we want to introduce them, it is necessary to establish a fixed set of ground types. For simplicity, we can assume the ground types are `boolean`. Therefore we can extend the previous syntax with the following grammar for types:

$$T ::= Bool \mid T \rightarrow T$$
$$t ::= \ldots \mid true \mid false$$

The terms would be the previous ones plus the new `boolean` constants. $T \rightarrow T$ represents the type of functions, in which the first T represents the arguments of type T and the second returns the result of type T.

Assigning types in an abstraction can be represented as $\lambda x : T.t$, which means that variable x has type T. Consequently, we would have to refactor the untyped λ-calculus syntax to include this abstraction instead of the untyped one.

Now we need to think about the typing rules of our simply typed λ-calculus. The typing rules for $true$, $false$ and variables would be simple:

$$\frac{}{\Gamma \vdash true : Bool}\ T-True \qquad \frac{}{\Gamma \vdash false : Bool}\ T-False \qquad \frac{(x:T) \in \Gamma}{\Gamma \vdash x : T}\ T-Var$$

For the other rules, we need to think about what is going to happen if some abstraction is applied to an argument. We will use the connective $\rightarrow$ for this representation. The rules for its introduction and elimination is the following:

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \rightarrow T_2}\ T-Abs \qquad \frac{\Gamma \vdash t1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2}\ T-App$$

On T-Abs we know the type of the argument and the type of the body t_2. From that we can assume the type of the result. The rule T-App would have a similar train of thought.

2.3 OSTRICH

This section presents OSTRICH [13] [22], a template language that is going to be the foundation for this dissertation work. Just like the structure proposed in [13] [22], this

work will also have as a starting point a simple example, which will focus solely on the screen and the table that can be viewed in Figure 2.3.

The list was chosen as an example since it is a usual pattern worth abstracting, considering it can be used in several applications.

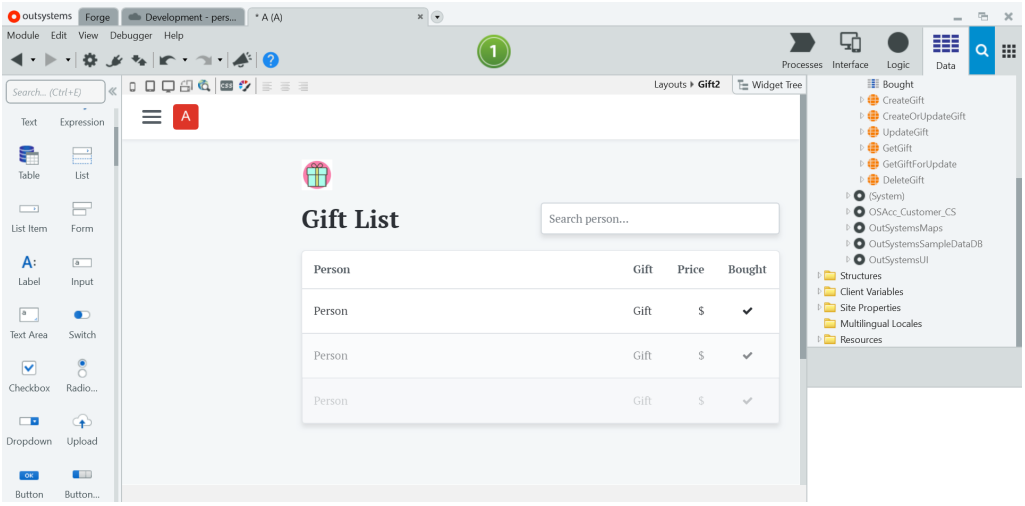


Figure 2.3: Gift list application in the OutSystems platform.

The screen we present in Figure 2.3 has a table with several entries of the same entity, which has four attributes, each corresponding to a column: (1) the person's name, (2) the gift, (3) its price, and (4) whether the person bought it or not. We sum up this information of the application model in Figure 2.4.

There are a few things we can observe when studying the model: an entity has attributes; the table is part of the screen; the column can have a value or an icon, depending on whether the attribute type is a boolean or not; the table can have several columns, with this last amount being the same as the number of attributes; the column has a title that can be configured; the column visibility status can be switched, through the visibility widget toggle.

When keeping these last things in mind, there are OSTRICH annotations that make the instantiation of elements easier: (1) template parameter that replaces the source of the table with the one provided by the developer; (2) the property value that can be used to determine the name of the screen or the title of a column if it was specified, or keeping the value defined in the base model otherwise; (3) iteration, which is necessary for creating multiple columns of the list; (4) conditional, that can either exclude or include an element - for instance, in the gift list example, the column can have an icon or a value, depending on the type of the attribute being represented; (5) template application, which instantiates a template with given arguments and replaces the annotated node.

Figure 2.5 presents the metamodel of the OSTRICH language, omitting nodes that were not relevant to the example we are studying. The yellow elements are exclusive to OSTRICH, while the others are already part of the metamodel of OutSystems applications.

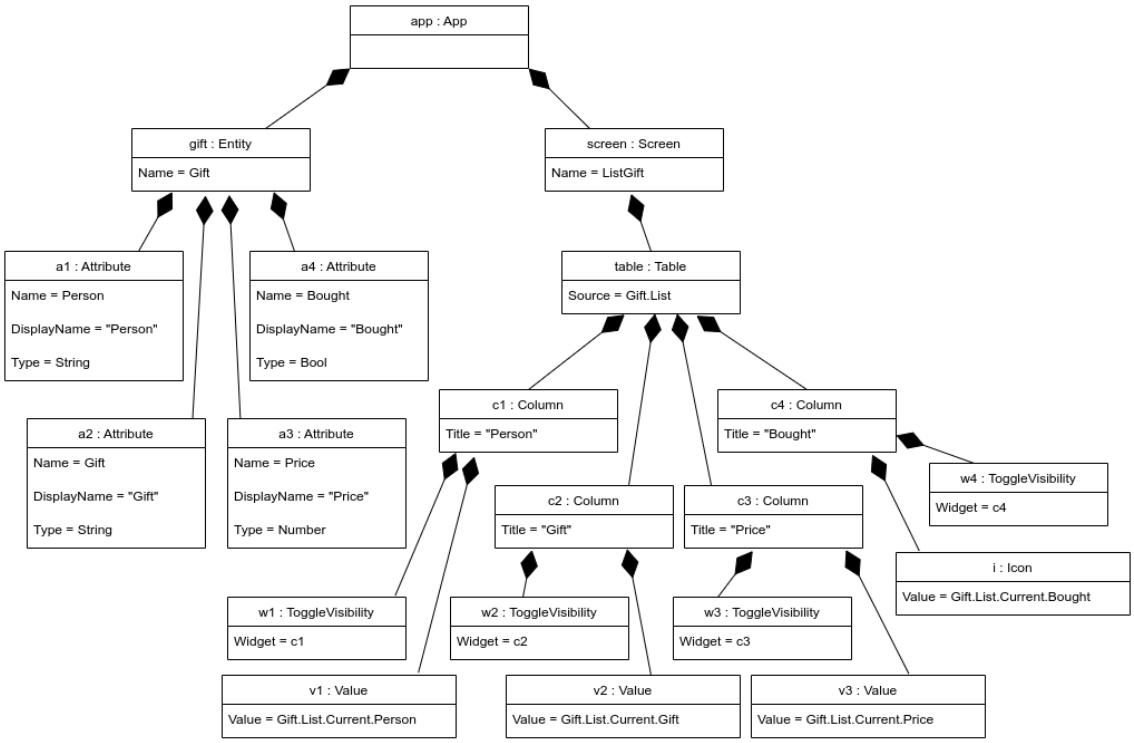


Figure 2.4: Application model of the gift list.

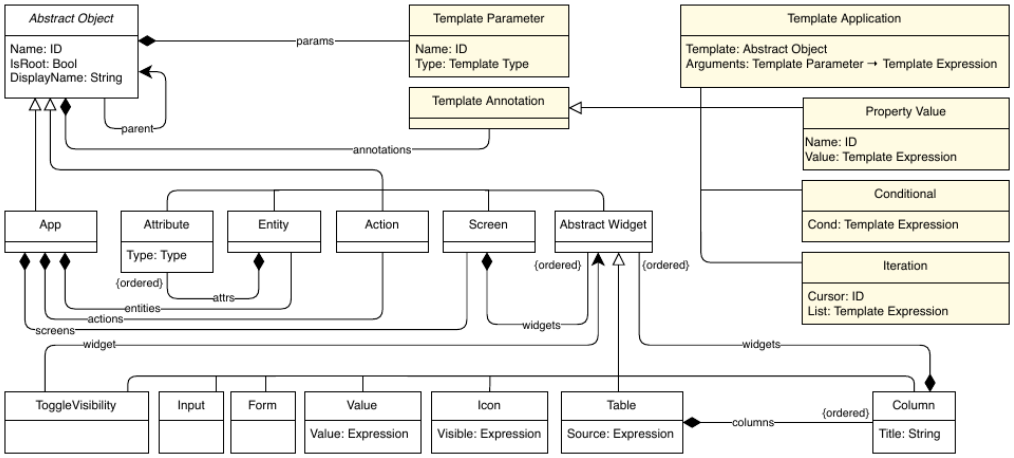


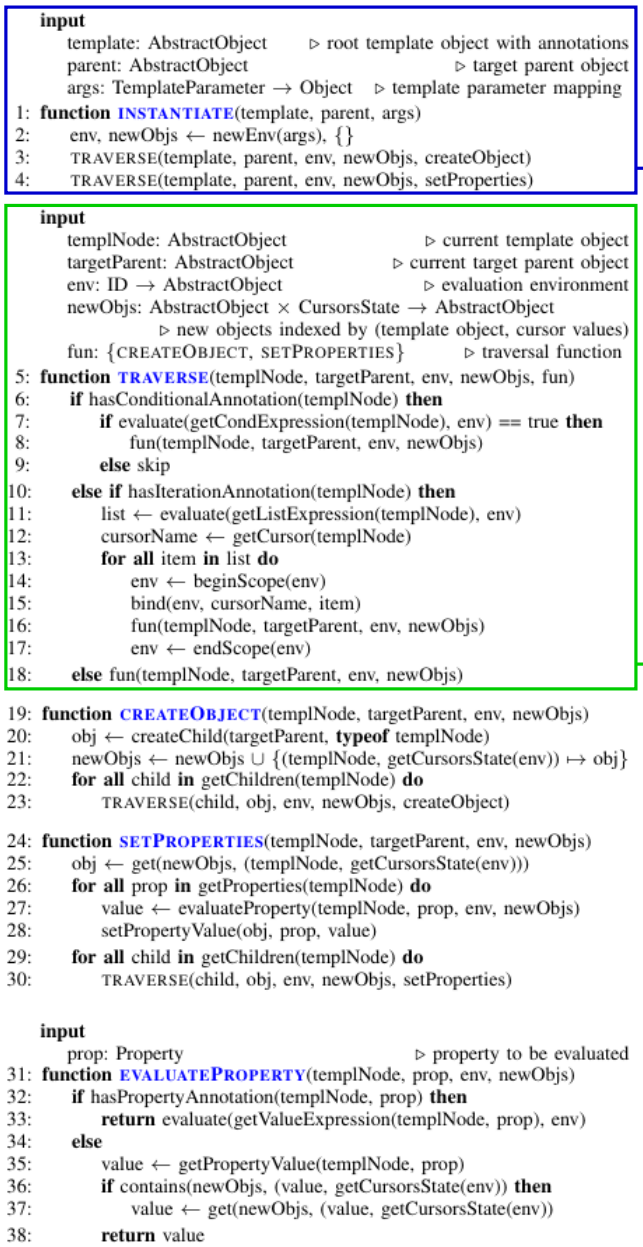
Figure 2.5: Template metamodel.

It is worth noting that the colored ones correspond to the annotations described in the previous paragraph.

We can make a few assumptions when presented with the metamodel. For instance, the application is composed of different types of abstract objects, such as entities with attributes - like the gift example we have seen before - or the screen that can contain widgets - as the value or icon previously mentioned.

The IDE offers an operation that allows the instantiation of the template. For that, the algorithm needs three inputs: (1) the template root object; (2) the target parent object in which the template is going to be instantiated; (3) arguments for each parameter of the

template. The instantiation algorithm can be better studied in Figure 2.6 which presents a few notes to some of its core functions.



```
   input
      template: AbstractObject      ▷ root template object with annotations
      parent: AbstractObject                          ▷ target parent object
      args: TemplateParameter → Object   ▷ template parameter mapping
1: function INSTANTIATE(template, parent, args)
2:     env, newObjs ← newEnv(args), {}
3:     TRAVERSE(template, parent, env, newObjs, createObject)
4:     TRAVERSE(template, parent, env, newObjs, setProperties)

   input
      templNode: AbstractObject                  ▷ current template object
      targetParent: AbstractObject             ▷ current target parent object
      env: ID → AbstractObject                    ▷ evaluation environment
      newObjs: AbstractObject × CursorsState → AbstractObject
                ▷ new objects indexed by (template object, cursor values)
      fun: {CREATEOBJECT, SETPROPERTIES}          ▷ traversal function
5: function TRAVERSE(templNode, targetParent, env, newObjs, fun)
6:     if hasConditionalAnnotation(templNode) then
7:         if evaluate(getCondExpression(templNode), env) == true then
8:             fun(templNode, targetParent, env, newObjs)
9:         else skip
10:    else if hasIterationAnnotation(templNode) then
11:        list ← evaluate(getListExpression(templNode), env)
12:        cursorName ← getCursor(templNode)
13:        for all item in list do
14:            env ← beginScope(env)
15:            bind(env, cursorName, item)
16:            fun(templNode, targetParent, env, newObjs)
17:            env ← endScope(env)
18:    else fun(templNode, targetParent, env, newObjs)

19: function CREATEOBJECT(templNode, targetParent, env, newObjs)
20:    obj ← createChild(targetParent, typeof templNode)
21:    newObjs ← newObjs ∪ {(templNode, getCursorsState(env)) ↦ obj}
22:    for all child in getChildren(templNode) do
23:        TRAVERSE(child, obj, env, newObjs, createObject)

24: function SETPROPERTIES(templNode, targetParent, env, newObjs)
25:    obj ← get(newObjs, (templNode, getCursorsState(env)))
26:    for all prop in getProperties(templNode) do
27:        value ← evaluateProperty(templNode, prop, env, newObjs)
28:        setPropertyValue(obj, prop, value)
29:    for all child in getChildren(templNode) do
30:        TRAVERSE(child, obj, env, newObjs, setProperties)

   input
      prop: Property                                ▷ property to be evaluated
31: function EVALUATEPROPERTY(templNode, prop, env, newObjs)
32:    if hasPropertyAnnotation(templNode, prop) then
33:        return evaluate(getValueExpression(templNode, prop), env)
34:    else
35:        value ← getPropertyValue(templNode, prop)
36:        if contains(newObjs, (value, getCursorsState(env)) then
37:            value ← get(newObjs, (value, getCursorsState(env))
38:        return value
```

A new environment and an empty map, required to store the future generated objects, are created with the received arguments. Then the function `Traverse` is called twice: first, to create the mentioned objects, and then to set their properties.

For each template node, a function `fun` is called with the environment varying. Before that, the algorithm will check and evaluate if there are annotations. If there is a conditional annotation, the function will be called if the result is `true`. If there is an iteration annotation, the resulting list elements will be iterated, and a new environment created for each of these iterations. In this environment, the item becomes available to any template expression contained in the template node and its children through the computed cursor name. The function `fun` is called with this new environment.

Figure 2.6: Template instantiation algorithm, retrieved from [13], and notes.

2.3.1 Syntax

OSTRICH is a multi-stage language. In the first stage, template expressions are evaluated in an environment containing, among others, references to elements in the actual model or parameters introduced in the template declarations. In the second stage evaluation, performed in a closed setting, the components mentioned are populated by model elements, and all the expressions associated with the component properties are set to run-time

values.

The syntax of OSTRICH is as presented in Figure 2.7. We now present each of these elements separately.

V	::=		(Basic Values)
	|	Integer	(Intenger literal)
	|	Boolean	(Boolean literal)
	|	String	(String literal)
	|	Unit	(Unit literal)
M	::=		(Template Expressions)
	|	V	(Basic Values)
	|	x	(Compile-time Variable)
	|	u	(Runtime Variable)
	|	L	(Label)
	|	$\mathsf{let}\ x\ =\ M\ \mathsf{in}\ M$	(Let)
	|	$\{L = M\}$	(Records)
	|	$[\overline{M}]$	(Lists)
	|	$M \oplus M$	(Binary Operations)
	|	$\mathsf{Node}(\alpha, \{\overline{L = M}\}, \overline{M})$	(Node Element)
	|	$\mathsf{ForNode}(x : T\ =\ M\ \mathsf{in}\ M)$	(Iteration Node)
	|	$\mathsf{IfNode}(M, \overline{M}, \overline{M})$	(Conditional Node)
	|	$\mathsf{Entity}(N, \{\overline{L = M}\}, \{\overline{L = M}\})$	(Entity)
	|	$\mathsf{Attribute}(N, L, T, \overline{L = M})$	(Attribute)
	|	$\mathsf{Template}(x, T, M)$	(Template)
	|	$M(M)$	(Template Instantiation)
	|	$\mathsf{box}(M)$	(Box)
	|	$\mathsf{letbox}\ x\ =\ M\ \mathsf{in}\ M$	(Let Box)
	|	$M\ \mathsf{isOfType}\ T$	(Type Validation)
	|	$\mathsf{NameOf}\ M$	(Name Property)
	|	$\mathsf{LabelOf}\ M$	(Label Property)
	|	$\mathsf{PropsOf}\ M$	(Properties)
	|	$\mathsf{AttributesOf}\ M$	(Attributes)
	|	$M \otimes O$	(Customization)
O	::=		(Customization Operation)
	|	$\mathsf{insert}(p, BV, M)$	(Insert Node)
	|	$\mathsf{remove}(p)$	(Remove Node)
	|	$\mathsf{replace}(p, M)$	(Replace Node)
	|	$\mathsf{move}(p, BV)$	(Move Node)
	|	$\mathsf{insertProp}(p, \{\overline{L = M}\})$	(Insert Property)
	|	$\mathsf{replaceProp}(p, \{\overline{L = M}\})$	(Replace Property)
	|	$\mathsf{removeProp}(p, \{\overline{L}\})$	(Remove Property)

Figure 2.7: Syntax of OSTRICH.

Basic values. Basic values compromise four different literals. These are integers, strings, booleans, and unit. This last one was added to the syntax as it was necessary to represent the different customizations uniformly.

Records and lists. Literals for records and lists are defined as usual.

Let binder. The construct let $x = M_1$ in M_2 represent all the elements M_1 designated by x that appear in M_2.

Binary operations. All operations that compromise basic values, records, and lists are represented by binary operations $M \oplus M$.

Nodes. We define nodes based on a type, their properties, and children nodes. The type of a node defines what kind of programming element it represents. The different types of nodes are presented in Figure 2.8a, and can range from, for instance, *Column* to *Screen*, two categories we have seen on our previously example. We have three kinds of nodes: Node, ForNode, and IfNode. Node represents a simple node in the tree, with a list of children nodes and a list of properties. The possibilities for this last one are displayed in Figure 2.8b, and just like the categories, their values can vary, with *Title* being one of the properties present in our previous example. Node IfNode represents a conditional term in the template language. Based on a boolean term, it expands one of two given branches containing a list of other nodes. Node ForNode($x : T = M_1$ in M_2) iterates over a list of values, given by term M_1, and produces an instance of term M_1 for each one of the elements of the list x;

α	::=	(Node Categories)	α	::=	(Properties)
	| *Top*	(Top)		| *Name*	(Name)
	| *Screen*	(Screen)		| *Visible*	(Visible)
	| *Table*	(Table)		| *Description*	(Description)
	| *Column*	(Column)		| *Type*	(Type)
	| *Calendar*	(Calendar)		| *DisplayName*	(Display name)
	| *Icon*	(Icon)		| *Source*	(Source)
	| *Expression*	(Expression)		| *Title*	(Title)
	| *Checkbox*	(Checkbox)		| *Value*	(Value)
	| *Input*	(Input)		| *Variable*	(Variable)
	| *Pages*	(Pages counter)		| *Attributes*	(Attributes)

(a) Possible node categories.

(b) Possible node properties.

Figure 2.8: Node categories and properties.

Entities and attributes. We define an Entity($N, \{\overline{L = M}\}, \{\overline{L = M}\}$) with a name N, a record mapping labels L to the entity's attributes M (columns), and a record mapping labels to properties of the entities themselves. An Attribute($N, L, T, \overline{L = M}$) has the name of the entity N they belong to, a label L, a type T, and a record mapping properties to their expressions.

Templates. Templates are abstracted terms that can be instantiated to create and replace nodes in a target tree. Term $\mathsf{Template}(x, T, M)$ declares a new parameter, with identifiers x, its type T, and the template body, M, that is also the scope of x. Template instantiation, $M_1(M_2)$, denotes a tree node produced from a template, denoted by M_1, and its argument M_2. Arguments can be entities, attributes, or other values.

Box and let box. Term $\mathsf{box}(M)$ represents a runtime expression to be evaluated in a later stage of computation. The destructor for these expressions is represented using $\mathsf{letbox}\ x\ =\ M_1\ \mathsf{in}\ M_2$. M_1 is the runtime term attributed to x, which may appear in the term M_2. When M_2 is also a boxed expression, the result is the combination of different delayed runtime elements to compose a new runtime expression based on compile-time decisions.

Built-in operations. $\mathsf{NameOf}\ M$, $\mathsf{LabelOf}\ M$, $\mathsf{PropsOf}\ M$ and $\mathsf{AttributesOf}\ M$ receive a term and output the information requested. $M\ \mathsf{isOfType}\ T$ has similar logic, with the exception that receives a second parameter, which is the type, T.

Customizations. Finally, we describe customization operations $M \otimes O$. These operations change trees are denoted by the term given as the first operand in a customization term. The operation O includes the type of operation to be applied, the path in the tree to identify the target node of the operation, and the term used as the argument to the operation. This last term can vary, depending on the operation: to insert or replace a node, we need it to be of type node; to remove and move a node, the term corresponds to unit, as it is not required; to insert and update properties on the node, we need it to be a record; lastly, to remove properties, we need it to be a list of the identifiers of those same properties. In case of node insertion and node movement, an integer defining the index is also required. Customizations are more carefully studied in Chapter 4.

Now that we are aware of the syntax, we can consider an example to better understand it: the property value of the Screen in Figure 2.4 can be computed as `Value = "List" + {{e.Name}}`, where `e.Name` refers to the name of the entity used to instatiate the template. We use the syntax with double curly braces to identify compile-time expressions embedded into runtime expressions (`e` is a compile-time variable). The previous concrete syntax is then transformed into abstract syntax, with the parser outputting the value `"List" + (NameOf e)`, in which the + concatenates a string literal with an identifier, resulting in a new identifier. For instance, when using the entity previously shown `Gift`, the result would be `ListGift`.

2.3.2 Type System

OSTRICH is a strongly typed template expression language in which the evaluation of template expressions always results in a valid run-time expression that matches the

expected output. This is a concern that sometimes is overlooked when working with templating languages. Besides that, our language preserves phase distinction. Moreover, the separation between type-level, compile-time, and run-time computations is well drawn.

Figure 2.9 presents the syntax of types. Below we study them separately, just as we did before in Subsection 2.3.1.

B	::=		(Basic Types)
	|	Integer	(Intenger)
	|	Boolean	(Boolean)
	|	String	(String)
	|	Unit	(Unit)
T	::=		(Types)
	|	B	(Basic Types)
	|	$\mathsf{Name}(N)$	(Name)
	|	$\mathsf{Label}(L)$	(Label)
	|	$\mathsf{LabelAttr}(N, T)$	(Attribute label)
	|	$\{L = T\}$	(Record)
	|	$\mathsf{RecordAttr}(N)$	(Records of entity's attributes)
	|	$[\overline{T}]$	(Lists)
	|	$\mathsf{NodeT}([\overline{\alpha}], [\overline{(pt, [\overline{p}])}])$	(Node)
	|	$\mathsf{EntityT}(N, \{\overline{L = T}\})$	(Entity)
	|	AttributeT(N,T)	(Attribute)
	|	$\mathsf{Template}(T_1 \rightarrow T_2)$	(Template)
	|	$\mathsf{BoxT}(T)$	(Box)
	|	Top	(Top)

Figure 2.9: Syntax of types.

Top and basic types. All of our value literals have their respective types: integers, strings, booleans, and unit. The type Top denotes a supertype that can represent all possible types.

Records and lists. A record $\{L = T\}$ denotes a collection of pairs in which label L has type T. A list $[\overline{T}]$ represents a list of types. Moreover, the type $\mathsf{RecordAttr}(N)$ conveys the record of attributes of the entity designated by N.

Names and labels. A name has the type of Name(N), whereas a label can have one of two types: $\mathsf{Label}(L)$ or $\mathsf{LabelAttr}(N, T)$. The first one stands for the description of a label, while the latter stands for an attribute label from entity N with type T.

Nodes. Template nodes have the type $\mathsf{NodeT}([\overline{\alpha}], [\overline{(pt, [\overline{p}])}])$, in which the $[\overline{\alpha}]$ represents the categories of the node, and $[\overline{(pt, [\overline{p}])}]$ represents the list of pairs of path and properties presented in respective path.

Entities and attributes. Entity types are represented as EntityT(N, $\{\overline{L = T}\}$), in which N stands for the the name and the list for the properties and respective types. Attribute types are denoted as AttributeT(N,T), where N is the name of the entity it belongs to, and T being the type of the attribute itself.

Template. We can look at the type Template($T_1 \rightarrow T_2$) as the function from T_1 to T_2.

Box. The type BoxT(T) represents the type of a delayed expression, in which T is the type of that expression during the runtime stage.

Furthermore, and from what we have seen before in Figure 2.5, there are a few template rules that need to be valid: an *App* node can only contain one of three nodes with them being either a *Screen* node, an *Action* node or an *Entity* node (we have seen two of these three in Figure 2.4 when presenting the application model of the gift example); by its turn, an *Entity* node can only contain *Attribute* nodes and these only exist inside entities; the *Table* node can only contain *Column* nodes, and *Column* nodes can include any abstract widget, such as an *Icon* or a *Value* node; similarly, the *Screen* node only contains abstract widgets as well.

Lastly, we present a simple example of types using our example gift once again illustrated in Figure 2.4. This entity is represented by the type Entity(Gift) and has four different attributes. If we look more closely at the attribute that indicates wether the gift was bought or not, we can see that it has the Boolean type. This way, we would represent the attribute type as Attribute(Gift,Boolean). The other three would be Attribute(Gift,String) for two of them - the ones that represent what the gift is about and for whom - and one for Attribute(Gift,Integer) to represent the price.

2.3.3 Evaluation

OutSystems provides a set of sample code fragments that developers can use. However, these still need some adjustments to the developer data, which requires effort and programming skills. This way, it is a process that is still prone to errors. Moreover, the operations applied to a template cannot be reapplied in the case of template updates or new parameters. OSTRICH will try to solve both these issues, and so far, one evaluation has been done to understand how OSTRICH is performing.

This evaluation targets the ten most used screen templates, which account for more than half of all template usage. In total, 70 screen templates exist. Out of the ones tested, only one, "Account Dashboard", was not successfully converted. This failure happened because OSTRICH still has some limitations. In this case, it was not possible to identify the minimum number of attributes an entity should have to be an argument for a template.

Screen template	% of total instantiations	Successfully converted
List with charts	19.1	✓
Admin dashboard	4.7	✓
Detail	4.6	✓
Dashboard	4.2	✓
List	4.2	✓
List with filters	4.0	✓
Bulk actions	3.7	✓
Four column gallery	3.2	✓
Account dashboard	3.2	✗
Master detail	3.0	✓

Table 2.1: Ten most used template screens, and result of conversion [13].

3

Related work

One of the indispensable properties of OSTRICH is code generation. This chapter will start by focusing on three different multi-stage programming languages that support program generation. The first two languages, MetaML and Template Haskell, are motivated algorithmically, while the third one, modal MiniML is motivated logically. The former means that MetaML and Template Haskell were designed to support specialization of a function to its early arguments, and the latter means that the statements of modal MiniML express facts and rules about problems within a system of formal logic [9].

This part of the chapter will end with a discussion in which we establish comparisons between these three languages.

To finish the chapter, we will explore how to create a bidirectional type system, and an example of a type systems with that technique.

3.1 MetaML

MetaML is a functional programming language for multi-stage programming, where programs are represented as abstract syntax trees, making their semantic properties verifiable [26] [23].

The purpose of MetaML is to write programs that build and manipulate other programs. The first ones are named meta-programs, and the second ones object-programs.

The most important aspects of MetaML are enumerated next.

1. Static type-checking: the typechecking occurs before any execution and only happens once, securing meta-programs and all its object-programs from run-time errors. This typechecking is crucial to ensure type-safety;
2. Cross-stage persistence and safety: variables bound in a particular stage are available in future stages but not in earlier stages.
3. Static scoping of variables in code fragments: this means that the bindings between names and objects can be determined at compile-time by examining the text of the program, without consideration of the flow of control at run-time;
4. Staging annotations, some of them analogous to LISPS annotations [26]: these staging

annotations allow the construction, combination, and execution of object-programs. These annotations are:

- Brackets ⟨_⟩ for delaying a computation. Such delayed expression is called a piece of code;
- Escape ~_, used inside Brackets and intended to relax the restriction that no reduction can occur. For a clearer idea, the expression ⟨ `...` ~⟨ `e` ⟩ `...` ⟩ is reduced to ⟨ `... e ...` ⟩;
- Run `run` _ to remove the outermost brackets, forcing a piece of code to be evaluated. The expression `run` ⟨ `e` ⟩ would be reduced to `e`;
- Lift `lift` _ transforms an expression into a piece of code, but reduces its input before delaying it.

The Escape operator has the highest precedence, and Lift and Run operators have the lowest.

3.1.1 Brackets

Brackets can be inserted around any expression to delay its execution, and the type inside brackets is not the same as outside brackets. For instance, ⟨`int`⟩ (read code of int) is not the same as `int`, therefore they cannot be combined.

A simple example of the usage of brackets is presented bellow:

```
-| val a = ⟨1+2⟩;
val a = ⟨1%+2⟩ : ⟨int⟩.
```

As it is verifiable, the addition was not performed. The symbol `%` indicates that + is not a free variable. In fact, in MetaML the operators are identifiers. We will explain this symbol later on.

3.1.2 Escape

As previously written, Escape allows the evaluation of a bracketed expression. As an example:

```
-| val pair = ⟨(~a, ~a)⟩;
val pair = ⟨(1%+2, 1%+2)⟩ : ⟨int * int⟩.
```

The declaration presented binds `pair` to a delayed expression, representing a tuple of the arithmetic expression presented in the previous subsection.

3.1.3 Run

Run executes a code fragment, allowing a delayed computation to be activated. Its use is illustrated in the following example:

```
-| val b = run a;
val b = 3 : int.
```

3.1.4 Lift

Both Brackets and Lift construct a piece of code, but unlike Brackets, Lift does not delay its computation. A simple example:

```
-| val c = lift 1 + 2;
val c = ⟨3⟩ : ⟨int⟩.
```

Now that all four annotations were introduced, a new example combining all of them is shown above:

```
run ⟨1 + ~((fn x => x) ⟨2+3⟩) + ~(lift (1+2))⟩
run ⟨1 + ~(⟨2+3⟩) + ~(lift (1+2))⟩
run ⟨1 + ~⟨2+3⟩ + ~(lift (1+2))⟩
run ⟨1 + 2 + 3 + ~(lift (1+2))⟩
run ⟨1 + 2 + 3 + ~(lift 3)⟩
run ⟨1 + 2 + 3 + ~(⟨3⟩)⟩
run ⟨1 + 2 + 3 + ~⟨3⟩⟩
run ⟨1 + 2 + 3 + 3⟩
9
```

3.1.5 Levels

To further explore the concepts of cross-stage safety and cross-stage persistence, it is essential to understand the notion of levels in MetaML. The level of a piece of code is equal to the number of surrounding brackets minus the number of surrounding escapes.

The expressions 5 and `(fn a => a+5)` are level 0 code. In a more complex example, ⟨`fn a =>` ⟨`a+5`⟩⟩, there are three different stages present: the first one is the piece of code ⟨`fn a =>` ⟨`a+5`⟩⟩; the second one is the result of running that program, a second stage function `fn a =>` ⟨`a+5`⟩; and the third one is ⟨`%a %+ 5`⟩, result of applying an integer to the second stage function.

Now that it is clear what levels are and how they are determined, cross-stage persistence and safety can be better understood. When a variable is bound at some level and used in another higher one, we call it cross-stage persistent. The following program shows an example:

```
let val a = 1 in ⟨a+1⟩ end
```

The example computes the code ⟨`a %+ %1`⟩, where `%` indicates that the variables `a` and `+` are bound in the code's local environment. When `%a` is evaluated in later stages, it will always return 1.

Allowing cross-stage persistence translates into a lack of cross-stage portability. This lack of cross-stage portability happens as most compilers do not maintain a high-level representation for values at run-time. Specifically, if we perform different stages in different computers, the local environment must be port from one machine to the other.

This manipulation can cause some portability issues. For now, MetaML always assumes that the computing environment does not change between stages.

A variable is cross-stage safety if it is not used in a level that comes first then the level it was bounded. This example shows how the concept can be violated:

```
fn a => ⟨fn b => ~(a+b)⟩
```

The variable `a` is available at the first stage, `b` at the second stage, and the expression is being computed at the first stage. Therefore, when `b` is first necessary, it was not yet bounded, causing an error in the code.

3.1.6 Two Stage Example

Consider the following function to search a value in a list:

```
(* search1 : 'a → 'a list → bool *)
fun search1 v l =
  if (null l)
    then false
  else if v = (hd l)
    then true
    else search v (tl l)
```

As we present it, the function does not have partial evaluation. However, if the list is available at the first stage and the value at the second stage, we can make the following assumptions: the type of value would be between brackets since it would only be later evaluated; the list type would still be the same; the result would also be inside brackets since the final result is only computed after the two inputs are available.

```
(* search2 : ⟨'a⟩ → 'a list → ⟨bool⟩ *)
fun search2 v l =
  if (null l)
    then ⟨false⟩
  else if ⟨ ~v = ~(lift (hd l))
    then true
    else ~(search v (tl l))⟩
```

The main difference between the two functions is the `⟨ ~v = ~(lift (hd l))` that ensures the `hd l` is performed during the first stage.

Moving to a more concrete example, evaluating `⟨fn x => ~(search2 x [1,2,3])⟩` yields:

```
⟨fun d1 =>
  if d1 %= 1 then true
    else if d1 %=2 then true
      else if d1 %=3 then true
        else false⟩
```

3.2 Template Haskell

Template Haskell is a language extension that adds features supporting compile-time meta-programming to the Haskell language, a strongly typed and functional language. Its purpose is to manipulate Haskell's code programmatically by allowing the programmer to compute parts of their program instead of writing them, automating aspects of the compilation [18] [24].

This extension is advantageous for, among others: (1) conditional compilation, beneficial for compiling a program for different platforms or with a different configuration; (2) program reification, as programs can inspect their structure; (3) optimizations since the compiler can learn new things; (4) algorithmic program construction where the algorithm that describes the program is simpler than the program itself - some examples are the functions `map` and `show`, now exemplified:

```
-| show 12
"12"
-| map (3*) [1,2,3,4]
[3,6,9,12]
```

It is required to have a few features available to generate a program. Some examples of these are the ability to retrieve information about something through its name and the ability to put and get some custom state that all Template Haskell code can share in the same module.

To achieve this, Template Haskell makes use of something called monads. All functions provided by this templating language are hosted by a special monad called Q, short for "quotation". There are a few types of quotations enabled by this extension:

Produces	**Quotation syntax**	**Type**
Declaration	`[d| ... |]`	`Q [Dec]`
Expression	`[e| ... |]`	`Q Exp`
Typed Expression	`[|| ... ||]`	`Q (TExp a)`
Type	`[t| ... |]`	`Q Type`
Pattern	`[p| ... |]`	`Q Pat`

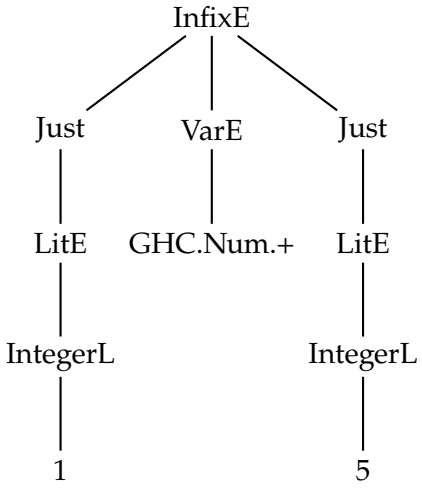
Since most of the time, the code being operated is about expressions, `[| ... |]` is the same as writing `[e| ... |]`.

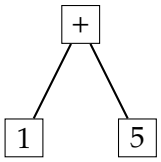
3.2.1 Quoting and Splicing

There are two principal forms in Template Haskell: quotes and splices. The first one takes some type of quotation and makes it a value of type code, resulting in an Abstract Syntax Tree. The second one takes an already quoted definition and evaluates it at compile-time. The code below shows an example of the usage of the two forms. It also shows the resulting Abstract Syntax Trees upon the evaluation of expressions.

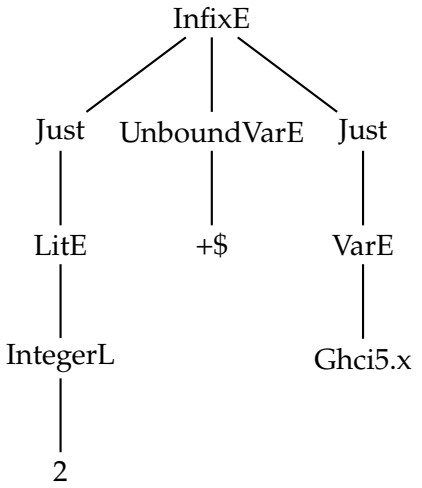
It is interesting to note how there is a parallelism between the annotations of Template Haskell and MetaML: Brackets and Quotes create a piece of code, and Splices and Escape evaluate some kind of definition in a piece of code. The similarities do not end there, as both languages also have the notion of level. Specifically, in both of them, it is not allowed to use a variable before they are bound, but it is possible to use it at some future stage.

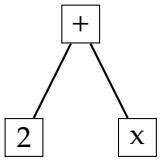
```
x :: Q Exp
x = [| 1 + 5 |]
```






```
y :: Q Exp
y = [| 2 + $(x) |]
```





An example of a forced error is presented below:

```
-| stageError = [| \x -> $(x) |]
Stage error: 'x' is bound at stage 2 but used at stage 1
```

The result is very self-explanatory: the `x` is being evaluated at a lower level than the one it was bounded. We can define a stage as the number of quotations minus the number of splices. Keeping that in mind, if quotations surrounded `x`, the stage error would no longer be present:

```
-| [| \x -> $[| x |] |]
```

This way, both `x` would be at the same stage.

If we now present an example of a variable being used at a future stage, we can verify that no errors occur:

```
-| double :: Int -> Q Exp
-| double x = [| x * 2 |]
-| $(double 10)
20
```

Besides the level restriction, Template Haskell has another limitation: a splice can only be used on already compiled functions. Specifically, functions need to be defined in a different module than where the splice is being applied. As an example, the following file `Main.hs` is presented:

```
import Language.Haskell.TH
decx :: Q [Dec]
decx = [d|x = 0|]
$decx
```

If the file is loaded, this is what will happen:

```
-| :load Main
GHC stage restriction:
'decx' is used in a top-level splice, quasi-quote, or annotation and
must be imported, not defined locally
Failed, no modules loaded.
```

To prevent that from happening, developers create a separate module usually named `TH.hs`, like it is being done below:

Main.hs	TH.hs

`import Language.Haskell.TH`
`import TH`
`$decx`
	`module TH where`
	`import Language.Haskell.TH`
	`decx :: Q [Dec]`
	`decx = [d|x = 0|]`

3.3 Logic Type Systems

This section will expose the programming language modal MiniML. As this language is an extension of modal λ-calculus, it is necessary to disclose that first. However, before diving into these two languages, it is required to understand a few concepts that both languages use. Namely, what is modal logic, and what is Curry-Howard isomorphism.

3.3.1 Modal Logic

Modal logic is part of the family of modal logics, which is also composed, among others, by temporal logic or deontic logic.

Modal logic is just a standard proposition logic extension with two new symbols: the box, $\Box$, and the diamond, $\diamond$. The former expresses necessity, while the latter expresses possibility. It is not required to use both symbols, as one can express both:

$$\Box A \equiv \neg \diamond \neg A$$
$$\diamond A \equiv \neg \Box \neg A$$

An example of usage of both symbols could be:

$\Box(1 + 1 = 2)$ = "It is necessary that the addition of two 1s equals 2"

$\diamond$ (Dog is six years old) = "It is possible that the dog is six years old"

The simplest system of modal logic is K, named after Saul Kripke, and it constitutes the basis for every other system of modal logic. It has three main components:

1. Necessitation rule: if A is an axiom of K, then $\Box$A is as well;
2. Distribution axiom: $\Box(A \rightarrow B) \rightarrow (\Box A \rightarrow \Box B)$. This means that if an implication statement is necessary, then both participants of that implication statement are also necessary;
3. Definition of possibility: expressed by the two rules first presented, relating the $\Box$ with the $\diamond$.

Another system is system T, also known as M, that is, the composition of system K plus axiom M, that says the following:

$\Box A \rightarrow A$: if it is necessary A, then it implies A.

Lastly, system T plus the axiom 4 compose the system S4:

$\Box A \rightarrow \Box\Box A$: necessary A implies that is necessarily necessary A.

There are other modal logic systems, such as S5, but they are not relevant to this work. Recapping the three studied systems:

1. $K = K$;
2. $T = K\&M$;
3. $S4 = K\&M\&4 = T\&4$.

3.3.2 Curry-Howard Isomorphism

Curry-Howard isomorphism, also known as propositions as types or simply Curry-Howard correspondence, establishes a relation between a given logic and a given programming language [28]. It starts by stating that each proposition in logic has a corresponding type in the programming language and vice-versa. Moreover, for each proof of a given proposition, there is a program of the corresponding type and vice-versa. It ends by noting that for each way to simplify a proof, there is a way to evaluate a program and vice-versa. We thus have:

propositions = types

proofs = programs

simplification of proofs = evaluation of programs

These conclusions first started when Curry observed that every type of function $(A \rightarrow B)$ could be read as a proposition $(A \supset B)$, and hence every type of any given

function would always correspond to a provable proposition, and vice-versa. Later on, Howard noted that there is a similar correspondence between natural deduction and simply-typed λ-calculus, and we can describe it as follows:

1. Conjunction $A \wedge B$ corresponds to Cartesian Product $A \times B$. The proof for the conjunction consists of a proof of A and a proof of B. Likewise, a value of type $A \times B$ consists of a value of type A and a value of type B;
2. Disjunction $A \vee B$ corresponds to a disjoint sum $A + B$. A proof of the disjunction would include proof of either A or B, including which one yields. Similarly, a value of type $A + B$ consists of the value of type A or the value of type B, including an indication of whether this is a left or right summand;
3. As mentioned before, the implication $A \supset B$ corresponds to function $A \rightarrow B$. A proof of the implication would imply that given A, then B yields. Likewise, a value of type $A \rightarrow B$ consists of a function that, when applied to a value of type A, returns a value of type B.

Furthermore, the predicate quantifiers $\forall$ and $\exists$ correspond to the already studied dependent types.

Before ending this subsection, we now present a connection between natural deduction (on the right) and simply-typed λ-calculus (on the left):

$$\frac{A \quad B}{A \wedge B}\ \wedge I \qquad\qquad \frac{M : A \quad N : B}{(M, N) : A \times B}\ \times I$$

Briefly, in the first case, the premises A and B yield, so we can conclude that $A \wedge B$. In the second case, term M has type A and term N has type B, then it is possible to form the pairing term (M, N) of product type $A \times B$.

3.3.3 Modal λ-calculus

Modal λ-calculus, $\lambda_e^{\rightarrow \Box}$, is just an extension of simply-typed λ-calculus plus the addition of the new symbol $\Box$ [9]. Any expression of the form $\Box A$ corresponds to code to be executed in a future computation stage, where $\Box A$ represents code of type A.

To decide which operations should be supported in code is necessary a constructor, **box**, where **box** $E : \Box A$, if $E : A$ in the empty context. It is essential to note that this is the modal rule of necessitation already presented.

Code can only be substituted into code and not arbitrary expressions, which is reflected in the necessitation rule. When code of type A is executed, this produces values of type A. It is crucial to make the following judgment to have a system that satisfies subject reduction and other desirable properties for a natural deduction system: A is both true and valid. Thus, there are two different types of hypotheses: (1) the ones that express validity; (2) the ones that convey truth. This types are written as:

$$(A_1, \ldots, A_m); (B_1, \ldots, B_n) \vdash C$$

And it means that under the hypothesis that $A_1, \ldots, A_m$ are valid and $B_1, \ldots, B_n$ are true, then C is true.

As the goal of the paper [9] is the analysis of the Curry-Howard isomorphism between proofs and programs, the hypotheses are labeled, and C is annotated with a proof term E:

$$\overbrace{(u_1 : A_1, ..., u_m : A_m)}^{modal\ variables,\ \triangle}; \overbrace{(x_1 : B_1, ..., x_n : B_n)}^{ordinary\ variables,\ \Gamma} \vdash E : C$$

To conclude that A is valid, its validity must not depend on the value of truth of any hypothesis. In different words, A is valid under the presumption that all hypotheses are true, which could be expressed as:

$$(A_1, ..., A_m); \cdot \vdash C$$

Or, using proof terms:

$$(u_1 : A_1, ..., u_m : A_m); \cdot \vdash E : C$$

The following expression means that E contains free code variables but not free value variables.

From the judgment presented, it is possible to infer two different rules:

1. A is true from the hypothesis that A is true:

$$\frac{x : A\ in\ \Gamma}{\triangle; \Gamma \vdash x : A}\ ovar$$

2. If A is true, then A is also valid.

$$\frac{u : A\ in\ \triangle}{\triangle; \Gamma \vdash x : A}\ mvar$$

Apart from that, it is also possible to make a derivation about the truth and validity of A for all use cases that A is, respectively, true and valid. That is, if C is true from the hypothesis that A is true, then it is possible to substitute a derivation establishing the truth of A for all use cases of the hypothesis, which ultimately results in a derivation of truth C which no longer depends on the hypothesis:

$$\text{If } \triangle; \Gamma \vdash E_1 : A \text{ and } \triangle; (\Gamma, x : A\Gamma') \vdash E_2 : B \text{ then}$$
$$\triangle; (\Gamma, \Gamma') \vdash [E_1/x]E_2 : B$$

Likewise, it is possible to substitute a derivation demonstrating the validity for A for all use cases that A is valid:

$$\text{If } \triangle; \cdot \vdash E_1 : A \text{ and } (\triangle, u : A, \triangle'); \Gamma \vdash E_2 : B \text{ then}$$
$$(\triangle, \triangle'); \Gamma \vdash [E_1/u]E_2 : B$$

Now that we have established the basis, we can introduce logical connectives and operators. Starting on the implication rules, both introduction and elimination, the following would apply:

1. Introduction - $A \to B$ should be true if B is true under the hypothesis that A is true:

$$\frac{\triangle; (\Gamma, x : A) \vdash E : B}{\triangle; \Gamma \vdash \lambda x : A.E : A \to B} \to I$$

2. Elimination - in an implication, if the right side is true, then the left side must also be true:

$$\frac{\triangle; \Gamma \vdash E_2 : A \to B \quad \triangle; \Gamma \vdash E_1 : A}{\triangle; \Gamma \vdash E_2 E_1 : B} \to E$$

For the new modal operator, $\Box$, that, once again, defines computation to be evaluated at a later stage, the rules would be as follows:

1. Introduction - if A is valid, then $\Box A$ is also true. It is important to note that only modal variables can occur free in E:

$$\frac{\triangle; \cdot \vdash E : A}{\triangle; \Gamma \vdash \textbf{box}\, E : \Box A}\ \Box I$$

2. Elimination - since a connective is only meaningful if local soundness and local completeness are valid, there are a few incorrect judgments that seem straightforward: it is not possible to conclude A from $\Box A$, as it is not complete; similarly, it is not possible to assume that A is valid from $\Box A$ is true, as that would be unsound. Instead, we could make the following conclusion: if $\Box A$ is true under some hypothesis, then any judgment made under the additional hypothesis that A is valid must be evident:

$$\frac{\triangle; \Gamma \vdash E_1 : \Box A \quad (\triangle, u : A); \Gamma; \Gamma \vdash E_2 : B}{\triangle; \Gamma \vdash \textbf{let box}\, u = E_1\, \textbf{in}\, E_2 : B}\ \Box E$$

   The term construct **let** is new and is required to express the following reasoning: E_1 represents a value of type $\Box A$, as it can be verified above, and is accessible in E_2 through the name u.

The other connectives, such as conjunction or disjunction, would have the same logic for the rules of introduction and elimination previously presented.

The summary of the properties for the system of natural deduction described is as shown:

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

A, B express types, a represents base types, x is used for ordinary variables in Γ and u is used for modal variables in $\triangle$. Furthermore, each variable can only be declared at most once in a context.

Now that we have put out the basis, the relationship between $\lambda_e^{\rightarrow\Box}$ and staged computation can be better studied. Like it was already noted, every term enclosed by a **box** constructor will have its evaluation delayed. The following judgment is applied:

$$E \mapsto E' \qquad E \text{ reduces to } E'$$

Which, by its turn, is defined by the following rules:

$$\frac{}{(\lambda x : A.E_2)E_1 \mapsto [E_1/x]E_2}\ \rightarrow\beta \qquad \frac{}{\textbf{let box}\, u = \textbf{box}\, E_1\, \textbf{in}\, E_2 \mapsto [E_1/u]E_2}\ \Box\beta$$

$$\frac{E \mapsto E'}{(\lambda x : A.E) \mapsto (\lambda x : A.E')}\ cong\_lam \qquad \frac{E_1 \mapsto E_1'}{E_1 E_2 \mapsto E_1' E_2}\ cong\_app1$$

$$\frac{E_2 \mapsto E_2'}{E_1 E_2 \mapsto E_1 E_2'}\ cong\_app2$$

$$\frac{E_1 \mapsto E_1'}{\textbf{let box } u = E_1 \textbf{ in } E_2 \mapsto \textbf{let box } u = E_1' \textbf{ in } E_2}\ cong\_letbox1$$

$$\frac{E_2 \mapsto E_2'}{\textbf{let box } u = E_1 \textbf{ in } E_2 \mapsto \textbf{let box } u = E_1 \textbf{ in } E_2'}\ cong\_letbox2$$

From this, the following theorem applies: $\triangle; \Gamma \vdash E : A$ and $E \mapsto^* E'$ then $\triangle; \Gamma \vdash E' : A$, where $\mapsto^*$ defines the reflexive and transitive closure of $\mapsto$.

The criteria for defining correctness in binding-time is followed [16] to demonstrate the relationship between this reduction relation and computation staging. Briefly, this correctness has two essential aspects to take into account. The first one is consistency and states that a binding-time analysis is correct if it always produces consistent binding-time information. This consistency is what prevents partial evaluators from going wrong, which happens if a partial evaluator trusts a part of the program to be of a particular form, of which it is not - since $\lambda_e^{\rightarrow\Box}$ is well-typed, this can never happen. The second aspect is that once a stage terminates, all partial evaluators will have eliminated all eliminable-marked parts of an input. In $\lambda_e^{\rightarrow\Box}$, all terms inside of $\Box$ are to be computed in a later stage and are, therefore, persistent. All other term occurrences are considered eliminable.

The following theorem expresses the second aspect: if $\cdot; \cdot \vdash E : \Box A$ and $E \mapsto^* E'$ and E' is irreducible, that is, there are no terms that be further reduced, then E' does not contain any eliminable term occurrences. This can be proven as $\cdot; \cdot \vdash E : \Box A$ reduces to $\cdot; \cdot \vdash E' : \Box A$ - by the former presented theorem -, and, since E' is irreducible, E' must have the form **box** E_0' for some E_0'. This way, all term occurrences in E' are in the scope of a **box** constructor and therefore persistent.

There is still one more property, not included in the correctness criteria, that the authors added to $\lambda_e^{\rightarrow\Box}$, as it needs to be shown explicitly: eliminable subterms cannot be reduced to persistent terms. For this matter, $\lambda_e^{\rightarrow\Box}$ is extended with labels, with the terms possible having the addition of E^l. The rules to apply labels (on the left), and to unlabel (on the right), would be the following:

$$\frac{\triangle; \Gamma \vdash E : A}{\triangle; \Gamma \vdash E^l : A}\ LB \qquad\qquad \frac{}{E^l \mapsto E}\ unlabel$$

If E_1 and E_2 differ only in a way that subterms of E_1 are labeled, then typing and reduction correspond between E_1 and E_2. This correspondence allows tracing the "images under reduction" as the authors called them, of eliminable parts of an unlabeled term by labeling all persistent subterm occurrences. From this, the following theorem applies: if $\triangle; \Gamma \vdash E : A$ and all persistent subterm occurrences of E are labeled with l, and $E \mapsto E'$, then all persistent subterm occurrences of E' are labeled with l.

3.3.4 Modal Mini-ML

Mini-ML$^{\Box}_{e}$ is a language that combines Mini-ML [5] with the already presented $\lambda^{\rightarrow\Box}_{e}$. Mini-ML is the restricted part of ML, and we can define it as a simple typed λ-calculus with constants, products, conditionals, and recursive function definitions. Unlike ML, which is implicitly typed, Mini-ML$^{\Box}_{e}$ is explicitly typed, and both exhibit polymorphism. That is, functions assume the same behavior for arguments of different types.

The syntax of Mini-ML$^{\Box}_{e}$ is as follows:

Types	$A ::= \text{nat} \mid A_1 \rightarrow A_2 \mid \Box A \mid A_1 \times A_2 \mid 1$
Modal Contexts	$\triangle ::= \cdot \mid \triangle, u : A$
Ordinary Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$
Terms	$E ::= x \mid \lambda x : A.E \mid E_1 E_2 \mid u \mid \textbf{box}\ u \mid \textbf{let box}\ u = E_1\ \textbf{in} E_2$ $\mid \langle E_1, E_2 \rangle \mid \textbf{fst}\ E \mid \textbf{snd}\ E \mid \langle\rangle \mid \textbf{z} \mid \textbf{s}\ E$ $\mid (\textbf{case}\ E_1\ \textbf{of z} \Rightarrow E_2 \mid \textbf{s}\ x\ \Rightarrow E_3) \mid \textbf{fix}\ x : A.E$

There are a few new keywords in this syntax compared to the one studied before. Their definition can be found in [8], but briefly, their meaning is:

1. **fst**: Expects a term $\langle s, t \rangle$ and returns s;
2. **snd**: Expects a term $\langle s, t \rangle$ and returns t;
3. **z**: Refers to 0;
4. **s**: Refers the successor function;
5. **case**: It is equivalent to **if** E_1 **then** E_2 **else** E_3;
6. **fix**: Represents a recursive function.

All the rules seen in $\lambda^{\rightarrow\Box}_{e}$ still apply to Mini-ML$^{\Box}_{e}$, with a few additions:

$$\frac{\triangle;\Gamma \vdash E_1 : A_1 \quad \Gamma \vdash E_2 : A_2}{\triangle;\Gamma \vdash \langle E_1, E_2 \rangle : A_1 \times A_2}\ tpe\_pair \qquad \frac{\triangle;\Gamma \vdash E : A_1 \times A_2}{\triangle;\Gamma \vdash \textbf{fst}\ E : A_1}\ tpe\_fst$$

$$\frac{\triangle;\Gamma \vdash E : A_1 \times A_2}{\triangle;\Gamma \vdash \textbf{snd}\ E : A_2}\ tpe\_snd \qquad \frac{}{\triangle;\Gamma \vdash \langle\rangle : 1}\ tpe\_unit$$

$$\frac{}{\triangle;\Gamma \vdash \textbf{z} : \text{nat}}\ tpe\_z \qquad \frac{\triangle;\Gamma \vdash E : \text{nat}}{\triangle;\Gamma \vdash \textbf{s}\ E : \text{nat}}\ tpe\_s$$

$$\frac{\triangle;\Gamma \vdash E_1 : \text{nat} \quad \triangle;\Gamma \vdash E_2 : A \quad \triangle;(\Gamma, x : \text{nat}) \vdash E_3 : A}{\triangle;\Gamma \vdash (\textbf{case}\ E_1\ \textbf{of z} \Rightarrow E_2 \mid \textbf{s}\ x\ \Rightarrow E_3)}\ tpe\_case$$

$$\frac{\triangle;(\Gamma, x : A) \vdash E : A}{\triangle;\Gamma \vdash \textbf{fix}\ x : A.E : A}\ tpe\_fix$$

When an expression is evaluated, it could be expressed as:

$$E \hookrightarrow V \qquad E \text{ evaluates to value } V$$

The written V could be one of the following:

$$\text{Values } V ::= \lambda x : A.E \mid \langle V_1, V_2 \rangle \mid \langle\rangle \mid \textbf{z} \mid \textbf{s}\ V \mid \textbf{box}\ E$$

The semantics for evaluating functions, code, products, natural numbers and recursion act in accordance with the rules shown below:

$$\frac{}{\lambda x : A.E \hookrightarrow \lambda x : A.E}\ ev\_lamb \qquad \frac{E_1 \hookrightarrow \lambda x.E_1' \quad E_2 \hookrightarrow V_2 \quad [V_2/x]E_1' \hookrightarrow V}{E_1 E_2 \hookrightarrow V}\ ev\_app$$

$$\frac{}{\textbf{box}\ E \hookrightarrow \textbf{box}\ E}\ ev\_box \qquad \frac{E_1 \hookrightarrow \textbf{box}\ E_1' \quad [E_1'/u]E_2 \hookrightarrow V_2}{\textbf{let box}\ u = E_1\ \textbf{in}\ E_2 \hookrightarrow V_2}\ ev\_let\_box$$

$$\frac{E_1 \hookrightarrow V_1 \quad E_2 \hookrightarrow V_2}{\langle E_1, E_2\rangle \hookrightarrow \langle V_1, V_2\rangle}\ ev\_pair \qquad \frac{E \hookrightarrow \langle V_1, V_2\rangle}{\textbf{fst}\ E \hookrightarrow V_1}\ ev\_fst$$

$$\frac{E \hookrightarrow \langle V_1, V_2\rangle}{\textbf{snd}\ E \hookrightarrow V_2}\ ev\_snd \qquad \frac{}{\langle\rangle \hookrightarrow \langle\rangle}\ ev\_unit$$

$$\frac{}{\textbf{z} \hookrightarrow \textbf{z}}\ ev\_z \qquad \frac{E \hookrightarrow V}{\textbf{s}\ E \hookrightarrow \textbf{s}\ V}\ ev\_s$$

$$\frac{E_1 \hookrightarrow \textbf{z} \quad E_2 \hookrightarrow V}{(\textbf{case}\ E_1\ \textbf{of z} \Rightarrow E_2 \mid \textbf{s}\ x \Rightarrow E_3) \hookrightarrow V}\ ev\_case\_z$$

$$\frac{E_1 \hookrightarrow \textbf{s}\ V_1' \quad [V_1'/x]E_3 \hookrightarrow V}{(\textbf{case}\ E_1\ \textbf{of z} \Rightarrow E_2 \mid \textbf{s}\ x \Rightarrow E_3) \hookrightarrow V}\ ev\_case\_s$$

$$\frac{[\textbf{fix}\ x.E/x]E \hookrightarrow V}{\textbf{fix}\ x.E \hookrightarrow V}\ ev\_fix$$

Most of the evaluations inferred are simple to understand at first sight. In detail, this is what is happening in three of those evaluations:

1. ev_app: expression E_1 evaluates to an expression E_1' with parameter x and E_2 evaluates to $V2$. Then replacing V_2 for x in E_1' would result in V. From this, it is easy to infer that the evaluation of E_1E_2 would evaluate to V;
2. ev_let_box: Similarly to ev_app, E_1 evaluates to $box\ E_1'$ and replacing E_1' for u in E_2 would result in V_2. This way, evaluating E_2 by accessing u computed from evaluated E_1 would evaluate to V_2;
3. ev_case_s: The first thing required is to evaluate E_1. Then, it is needed to replace V_1', the result of E_1, for x in expression E_3. The result would be V, which is also the final result of the whole expression. env_case_z is similar.

From these rules of evaluations, there are a few statements that can be made:

1. If $E \hookrightarrow V$ then V is a Value;
2. If $E \hookrightarrow V$ and $E \hookrightarrow V'$ then V and V' are the same;
3. If $E \hookrightarrow V$ and $\cdot;\cdot \vdash E : A$ then $\cdot;\cdot \vdash V : A$, which is to say that both E and its result V would have the same type.

Moving on to the relation between products and $\Box$, we show two examples. It is important to remind that since products and conjunction are related as seen in Subsection 3.3.2, these examples correspond to proofs in modal logic:

1. $\text{-} \mid \lambda x : (\Box A) \times (\Box B).\textbf{let box}\ u_1 = \textbf{fst}\ x\ \textbf{in let box}\ u_2 = \textbf{snd}\ x\ \textbf{in box}\langle u_1, u_2\rangle$
   $(\Box A) \times (\Box B) \rightarrow \Box(A \times B)$
   It is easy to conclude that the returning type has the form of $\Box(A \times B)$ by the last expression $box\langle u_1, u_2\rangle$, composed by the $\Box$ and a pair, or product.
2. $\text{-} \mid \lambda x : \Box(A \times B).\textbf{let box}\ u = x\ \textbf{in}\ \langle \textbf{box fst}\ u, \textbf{box snd}\ u\rangle$
   $\Box(A \times B) \rightarrow (\Box A) \times (\Box B)$

Likewise, the returning type is $(\Box A) \times (\Box B)$, as the expression $\langle \textbf{box fst } u, \textbf{box snd } u \rangle$ expresses a product of two values in the $\Box$ form.

Focusing now on recursive functions, the *power* example shows the **fix** and **case** keywords in action. This function has type nat $\rightarrow \Box$(nat $\rightarrow$ nat), and makes use of a function *times* that receives two natural numbers and returns another natural number (nat $\rightarrow$ nat $\rightarrow$ nat).

$$
\begin{aligned}
power \equiv\ & \textbf{fix } p : \text{nat} \rightarrow \Box(\text{nat} \rightarrow \text{nat}). \\
& \lambda n : nat.\textbf{case } n \\
& \ \textbf{of z} \Rightarrow \textbf{box } (\lambda x : \text{nat}.\textbf{s z}) \\
& \ | \ \textbf{s } m \Rightarrow \textbf{let box } q = p\ m \textbf{ in} \\
& \qquad\qquad \textbf{box } (\lambda x : \text{nat}.times\ x\ (q\ x)))
\end{aligned}
$$

This example is simple. It states the returned number will be the successor of 0, that is, 1, if the exponent received equals 0. Otherwise, it will compute the multiplication between the base, x, and the result of calling the recursive function for $n - 1$. In this function, everything that depends on the exponent is computed. In action, this function would behave as it follows:

$$
\begin{aligned}
power\ z &\hookrightarrow \textbf{box } (\lambda x : \text{nat}.\textbf{s z}) && \textit{(from case n of z)} \\
power\ (s\ z) &\hookrightarrow \textbf{box } (\lambda x : \text{nat}.times\ x && \textit{(from case n of s m) (1)} \\
&\qquad ((\lambda x : \text{nat}.\textbf{s z}) && \textit{(from case n of z)} \\
&\qquad x)) && \textit{(1)} \\
power\ (s\ (s\ z)) &\hookrightarrow \textbf{box } (\lambda x : \text{nat}.times\ x && \textit{(from case n of s m) (1)} \\
&\qquad (\lambda x : \text{nat}.times\ x && \textit{(from case n of s m) (2)} \\
&\qquad ((\lambda x : \text{nat}.\textbf{s z}) && \textit{(from case n of z)} \\
&\qquad x)) && \textit{(2)} \\
&\qquad x)) && \textit{(1)}
\end{aligned}
$$

3.4 Discussion

All languages presented have strong staging properties. In MetaML we express the delayed code using Brackets; in Template Haskell using Quotes; and finally, in Mini-ML$_e^{\Box}$ using the constructor **box**. Similarly, all three of them allow the execution of code by having, respectively, Escape, Slices, and **let box**. While using stages, it is possible to use a variable declared in a previous stage but never on a future one. That is, all three languages ensure cross-stage persistence and safety.

Despite these similarities, these languages also have their differences [24]. For instance, in Template Haskell, typechecking is delayed until the last possible moment, which allows for more powerful meta-programs without resorting to dependent types. In MetaML this carefulness is not taken into account, as the typechecking for all stages occurs once and for all before the execution of the first stage. Template Haskell also differentiates from the others in three crucial aspects: (1) it allows manipulation of the data, as its

structure representation can be obtained and inspected by compile-time functions; (2) all meta-computations happen at compile-time; (3) its data structures are represented using algebraic datatypes - these represent, among others, expressions and declarations -, which makes it compliant to pattern matching and case analysis (this last one also present in modal MiniML). On the other hand, this last approach also implies that the construction of a representation of a program may require more effort than the one necessary to construct the program itself.

4

Customizations

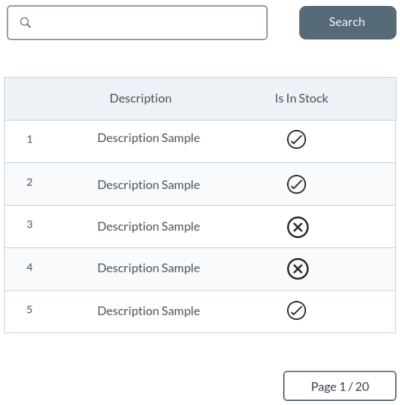
In Section 2.3, we have seen that behind each application, there is a tree of nodes representing the widgets and the general structure of a template. Thus, if anything on the application changes, it will be translated into this same tree. From this knowledge, a few questions may surge regarding this tree. Is it possible to insert nodes between others? Can we remove some nodes and keep the rest of the tree as it is? What functions should be supported? And many others. This chapter will start by outlining the elements composing an application and how to reach each of them. After that, we will explore the seven different customizations that can be applied to a template and the considerations we need to take into account when applying them. The presentation of such customizations will be the purpose of this chapter. Throughout the presentation, we will be guided by an example application to give the reader a better understanding of this matter. We will assume the same approach in every section, with a few modifications to our example.

4.1 Overview

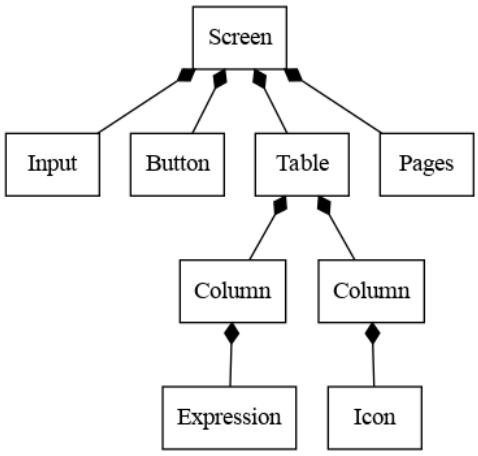
There are seven main operations involving a tree of nodes. The first four regard the structure of the tree, while the last three regard the structure of the node itself. These operations are, respectively: (1) insertion of a node; (2) replacement of a node; (3) removal of a node; (4) move node; (5) addition of new properties to some node; (6) update of the node properties; and (7) removal of node properties.

All these customization functions require mostly different arguments to perform successfully, apart from two common requirements, which are the path and the original tree. The path indicates how to reach some tree node from the root node and can be easily produced by a user interaction that sees the changes on the user interface in an IDE.

To better visualize the path, let's consider an application and its tree of nodes. This application is represented in Figure 4.1a. Our example application has four different widgets and aims to show the properties of several products, with the possibility of searching for them separately. The widgets available for this purpose are an input bar (*Input* node), a search button (*Button* node), a table (*Table* node), and, lastly, the page



(a) Application example.



(b) Tree of widgets on application.

$$
\begin{aligned}
&\text{Node}\langle \textit{Screen}, [], [\\
&\quad \text{Node}(\textit{Input}, [], []),\\
&\quad \text{Node}(\textit{Button}, [], []),\\
&\quad \text{Node}\langle \textit{Table}, [], [\\
&\qquad \text{Node}(\textit{Column}, [], [\text{Node}(\textit{Icon}, [], [])]),\\
&\qquad \text{Node}(\textit{Column}, [], [\text{Node}(\textit{Expression}, [], [])])\\
&\quad]\rangle,\\
&\quad \text{Node}(\textit{Pages}, [], [])\\
&]\rangle
\end{aligned}
$$

(c) Syntax of the example tree.

Figure 4.1: Application with widgets.

counter (*Pages* node). On Figure 4.1a side, in Figure 4.1b, there is a tree of the widgets, without their properties for clarity, present in the application. We can check that the screen has the four mentioned elements, with the table having two columns, each having either an icon (for the stock property) or an expression (for the description property). Using the syntax represented in Section 2.3, the application would have the code illustrated in Figure 4.1c. From it, we can check that *Input*, *Button*, *Table* and *Pages* are all children of the *Screen* node, the latter being the application root node. Moreover, the *Table* includes two distinct columns in its list of children, and each *Column* node contains, respectively, an *Icon* and an *Expression*. Notice that the list of properties of every node is empty, as we decided to remove them from this example in favor of simplicity.

As it was stated, to reach some node from the root node, we always need to know the path. This path consists of a list of numbers. Each number at some index of the list corresponds to the node order in the list of widgets at that depth and with the same parent. For instance, the *Screen* node could be found in path [0]. On the other hand, the *Table* would be at [0 2] - the first number 0 indicates the *Screen* (we always append the path that allows us to reach the parent node), while the 2 indicates the third child of the node *Screen* (we always start counting from 0 and not 1); and the *Icon* would be at [0 2 1 0],

since it is the first child of the *Column* present at [0 2 1], which by its turn is the second child of the *Table* node present in [0 2], and so forth. The complete list of the path to reach each node of the tree is represented in Figure 4.2. Before we go on, it is worth pointing out that these examples will not take into consideration any IfNode or ForNode as they are evaluated before any customization is performed. Therefore, we will be solely focused on grounded node trees. Moreover, the IfNode and ForNode require a little more care and will be studied in Section 5.1 as we encounter them in the typechecking phase.

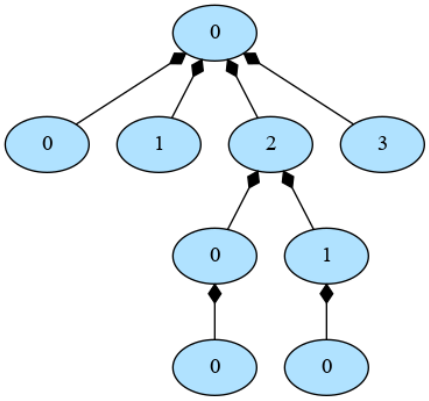


Figure 4.2: Path to nodes.

Apart from the path and the original tree, the other necessary parameters required to apply the different operations to the tree of nodes vary. If our goal is to insert or replace a node, we will have to provide one. In case of node insertion, or node movement, we will also have to supply the index intended for the target element - that is, the position it is going to assume in its siblings' list (this will become clearer in the upcoming examples of this matter). It is worth noting that the index will not be part of the path, as we found it to be more explicit having both separated. If the operations in question regard properties, we have to either provide a record of these in case of property addition or replacement - the record maps the identifiers to the values they hold - or a list of labels corresponding to the identifiers in the case of property removal. These requirements were already seen in the syntax presented previously in Section 2.3.

Now that we are acquainted with the operations, we can study them more carefully one by one, always taking into account our example application as the starting point of each applied customization.

4.2 Customization Operation Description

Insert node Apart from the original tree, which along with the path, is always required, we would need to provide the index, the path, and the node to insert.

As an example, let's consider our previous application represented in Figure 4.1. To add a *Checkbox* between the *Button* and the *Table* node, we would have the following parameters: the path [0], since the new node would be a child of *Screen*; the index 2 -

the first two elements of that level, index 0 and 1, would still be the *Input* and the *Button*, while the *Table* and *Pages* node assume the positions following the new node place; and the *Checkbox* node to insert. The resulting graph of applying this operation is illustrated in Figure 4.3. On the left, we see the initial template. On the right, we see the template after the customization has been applied.

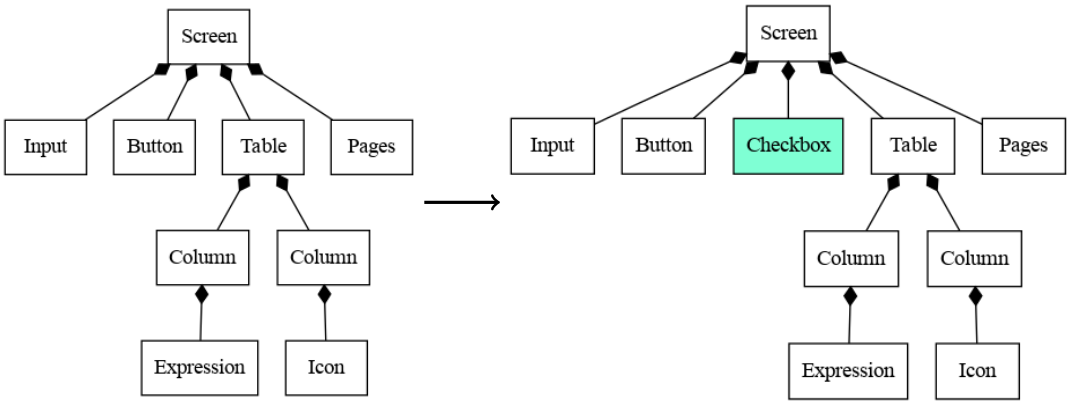


Figure 4.3: Example of insert node operation.

If we consider t to be the original tree represented in Figure 4.3 (the one on the left), we would have the syntax of this operation as:

$$t \otimes \mathsf{insert}([0], 2, \mathsf{Node}(\mathit{Checkbox}, [], []))$$

Once again, we are inserting a node without properties as all the others were, hence the first empty list inside it. The second empty list indicates that the node does not have any children.

Replace Node To replace a node on the tree, we would need the path for replacement and the node that is going to replace the one we reached from the given path. We would no longer need the index as that is only required for node insertion and node movement.

Grabbing the same example application, if we wanted to switch the *Icon* for the *Checkbox* node, we would have the following parameters: the path [0 2 1 0], the original tree (the same as before) and the *Checkbox* node. Regarding the path provided, we would start on *Screen*, placed at path [0]. Then we would move to the third child of *Screen*. That is, the *Table*. The *Table* has one of two *Column* nodes, and we will move to the second one as it is the parent of the *Icon* node. After that, we will simply choose the *Icon*, which would finally be substituted.

The result of applying such customization is presented in Figure 4.4.

Using our syntax, the customization would be written as:

$$t \otimes \mathsf{replace}([0\ 2\ 1\ 0], \mathsf{Node}(\mathit{Checkbox}, [], []))$$

where t is the initial tree of Figure 4.4.

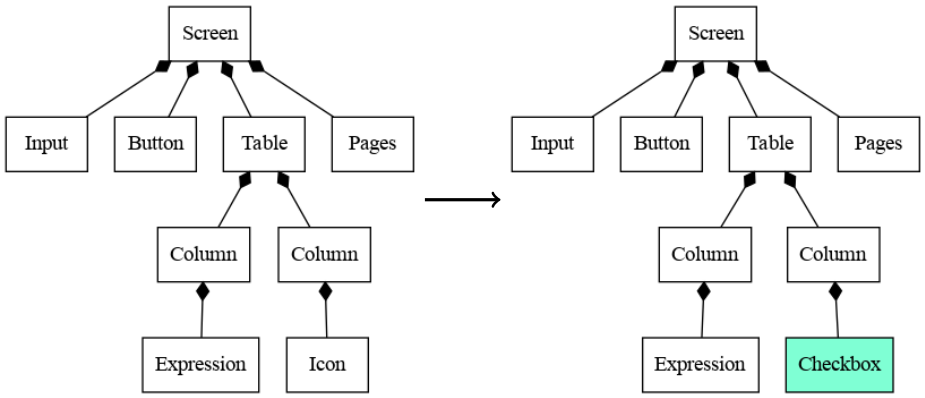


Figure 4.4: Example of the replace node operation.

Remove Node To remove a node on the tree, the only two arguments we need to provide are the path and the original tree.

If we use the same structure as in the previous operations, and we wanted to remove the *Table* node, we would have to provide the path [0 2] - it is a child of the *Screen* node and corresponds to the third element of its children. Using our syntax, we would express such customization as:

$$t \otimes \mathsf{remove}([0\ 2])$$

where t is the initial tree in Figure 4.5. This example is illustrated in Figure 4.5. We can check that the *Table* node, as well as all its children, are no longer presented in the final tree while all the other nodes remain in the same position.

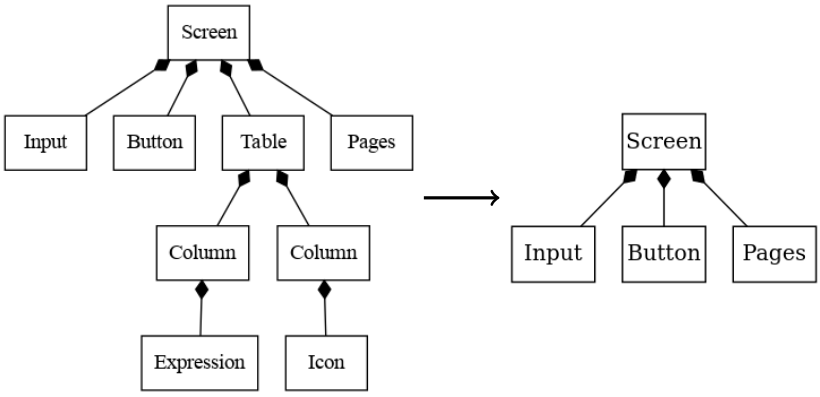


Figure 4.5: Example of remove node operation.

Move Node To move a node, we need to provide the original tree, the path for that node, and an index indicating where the node is going to be moved on that list. This list is the list of its siblings (the nodes that share the same parent).

Using our example once again, if we wanted to move the *Table* node to the first position of the list, we would have the path [0 2] and the index 0. The path [0 2] indicates where the node is on the tree concerning the root node (the *Screen*).

In our syntax, this example would look like the following:

$$t \otimes \mathsf{move}([0\ 2], 0)$$

where t is the initial tree in Figure 4.6, in which our full example is illustrated. We can check that the *Table* node replaced the *Input* as the first element. All the other elements remained the same.

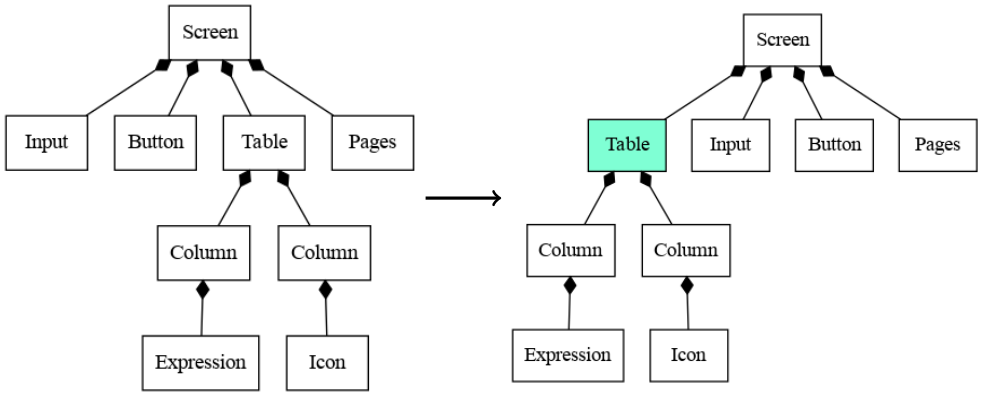


Figure 4.6: Example of move node operation.

Add Property All the three customizations left to present regard properties instead of nodes.

To add a property to some node, we need to have the original tree, the path to the node that is going to be modified, and a record of properties. The length of this record can vary according to the user's needs.

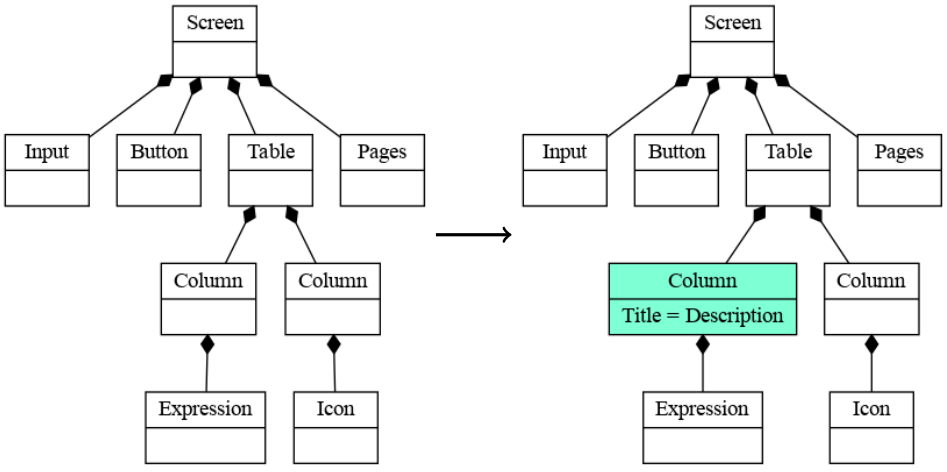


Figure 4.7: Example of add property operation.

If, for instance, we wanted the first *Column* to have a property named *Title* with the value *"Description"*, then we could simply provide the path to that node and a record containing this one property. This way, the expression in our syntax would be as below:

$$t \otimes \mathsf{insertProp}([0\ 2\ 0], \{\mathit{Title} = \text{“}\mathit{Description}\text{”}\})$$

where t is the initial tree of Figure 4.7. This operation is illustrated in Figure 4.7. The box has the category of the node above the line and its properties below. From now on, we will use them to represent nodes with properties.

For this operation to work, we have to make sure that the new properties to be added do not exist in the original tree.

The other two customizations that affect the node properties - remove a property or replace one - have similar behavior to what we have already seen. Both operations require the path to reach the target node and a record of properties, mapping their identifier to their value. To avoid repetition, we will not be demonstrating them in an identical structure as we did with the others.

Given all the seven customizations, we now focus on how the type system must work to avoid execution errors. The most obvious check is to confirm if the path is valid. That is, if we can reach some node from the root node, taking it into consideration. At first, it seems that all paths provided are, as the changes are obtained through the drag-and-drop capabilities of the low-code platform, thus using existent paths. However, we cannot disregard that some operations are applied through a log of customizations. Therefore, even if some paths were valid on a template, they may not be valid on a new template. Apart from these, we have to check different arguments depending on the operation. For the node insertion and node movement, we have to check if the index is valid for the same reason we have to check the path. For the property addition and replacement, the type system must check if the properties we are targeting exist. Lastly, and following the same reasoning, the labels given on property removal must correspond to actual properties. We will study more carefully these types of situations in the next chapter.

5

Type System

Following the presentation of customizations in Chapter 4, we will take a closer look at what the types involved in these operations hold. First, we will present a more general outline in Section 5.1 that focuses solely on nodes. Then, we go deeper into how our typechecker behaves in Section 5.2 and the reason for this behavior. We separate these two as it is essential to understand the type produced by a tree without involving customizations, which make the type of the resulting node more complex. Moreover, the changes produced by the path on the node type are the last ones left to study since Section 2.3 presented the syntax of types and seems fitted to learn about them first.

We will finish this chapter by exposing the typing rules involved in these operations in Section 5.3 and the steps to carry out their execution in **??**. We leave these rules for the end, as it will be easier to get a grasp on them having the knowledge shared up to their start in Chapter 4 and this one.

5.1 Path Type

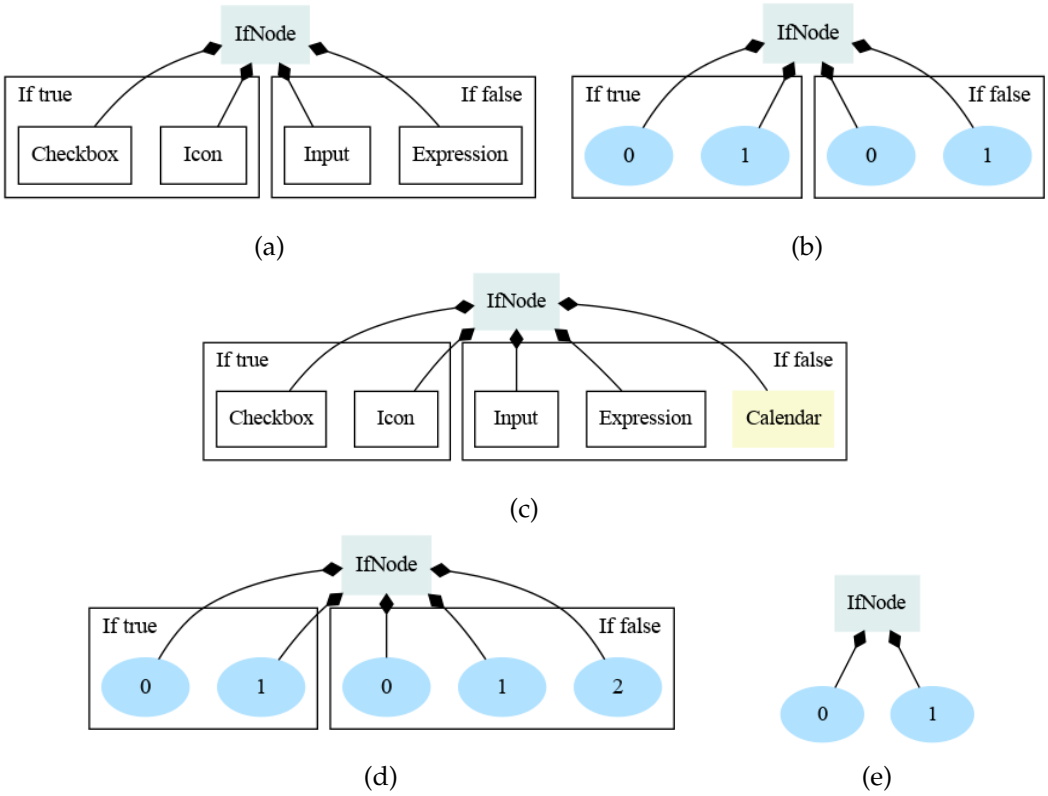
When we are evaluating expressions, we never find an IfNode nor a ForNode because those terms have to be first evaluated to produce grounded node trees (Node). However, when the moment of typechecking occurs, we can encounter these type of nodes, so it is necessary to think about what information is included in the specification. Namely, should the IfNode and ForNode be ignored or should the path of these compile-time nodes be kept?

The types of IfNode andForNode comprise the following: (1) the set of node types they may produce - the categories of an IfNode account for the disjunction of all nodes, independently of the branch they are in; (2) the set of paths available; (3) the record of properties in each node of the tree. To better understand this, we will study the path of each customization represented by an IfNode and a ForNode.

So far, the path to reach a node has been very straightforward: we always start from 0 and go from there. However, not always does this correspond to reality. If we happen to have an IfNode present in the original tree, then we have two possible branches: the one

that evaluates to true and the one that results to false.

To better visualize a case like this, we disregard our previous application and consider a more simple one that either displays a *Checkbox* and an *Icon* or displays an *Input* and an *Expression*. This example is illustrated in Figure 5.1a. We can see that the cluster on the left corresponds to the value true on the evaluation, while the right corresponds to the value false. It is also important to note that each branch can contain a list and not just a single node. Although this last one is also possible, corresponding to a list with a length of one. The path of each of these nodes is represented in Figure 5.1b.



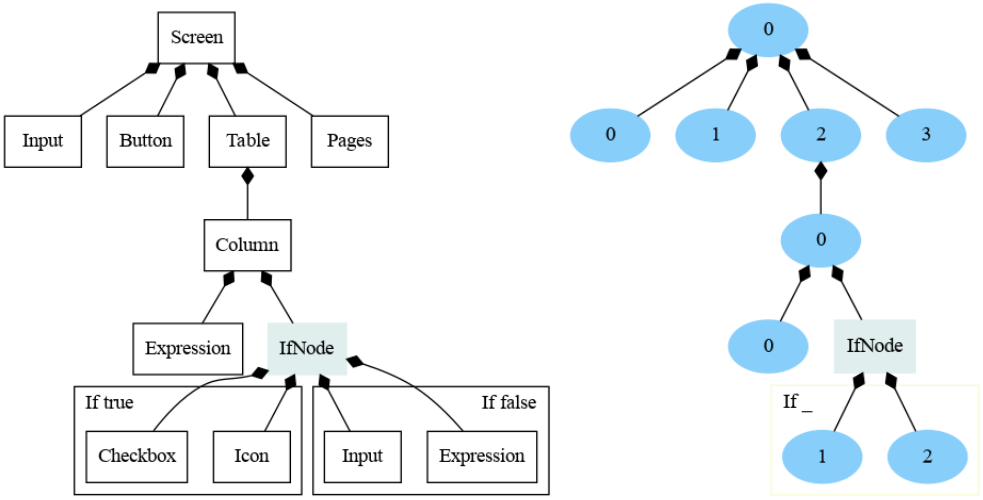
NodeT⟨[*Checkbox*, *Icon*, *Input*, *Expression*, *Calendar*], [
([0], []);
([1], [])]⟩

(f)

Figure 5.1: Example of IfNode, path and type.

If we now consider the example in Figure 5.1c, our intuition tells us that we should add a 2 to the right side. Although this is correct (we can check this from Figure 5.1d, where the *Calendar* node added one more possible path to the false cluster), this number 2 will not matter when it comes to the moment of typechecking. This unimportant attribution happens because when typechecking occurs, it is used the maximum information we can get from a node. When it comes to an IfNode, this information must be generalized between the left and the right side. Using this reasoning, the most we can know from the

two clusters is as much as they have in common. Thus, the paths that we can assume the IfNode to have will be the intersection of paths from the two sets. This way, Figure 5.1e shows the path for each of our two examples. We can see that even though our trees were different, the type produced ended up being the same. It should also be pointed out that the same logic goes for the properties of the nodes and not just for the path. For simplicity, we only considered the path in these examples. Moreover, the result of typechecking this last example is shown in Figure 5.1f. Notice that the possible categories for the node type are obtained from the disjunction of the ones of all nodes in each branch - the one that evaluates to true and the one that evaluates to false - as we have previously stated. The path (and respective properties) remain the same.



(a) Node tree.

(b) Resulting path tree.

NodeT⟨[*Screen*], [
([0], []);
([0 0], []);
([0 1], []);
([0 2], []);
([0 2 0], []);
([0 2 0 0], []);
([0 2 0 1], []);
([0 2 0 2], []);
([0 3], [])]⟩

(c) Node type.

Figure 5.2: A more complex example of IfNode usage.

In Figure 5.2 we illustrate a more complex example. It is the application example from Figure 4.1 with a few modifications. Instead of just having two *Column* nodes, we only have one that displays an *Expression* at all times, followed by either a *Checkbox* and an *Icon* or an *Input* and another *Expression*. If we look at the path, we can see that the IfNode

produced two node paths: 1 and 2, obtained from the intersection of the two clusters. It starts at 1 because we have, at the same depth and with the same parent, the node *Expression* that sits at 0. All the other paths are pretty straightforward and have already been studied. The result of typechecking such tree is also illustrated. We can notice that the kind of category is *Screen*, which corresponds to the root node, and all paths and corresponding properties are present on the type.

Now that the IfNode has been presented, we have one type of node left to study: the ForNode. This node has similar reasoning to what we have already analyzed. Its purpose is to range over the attributes of some entity.

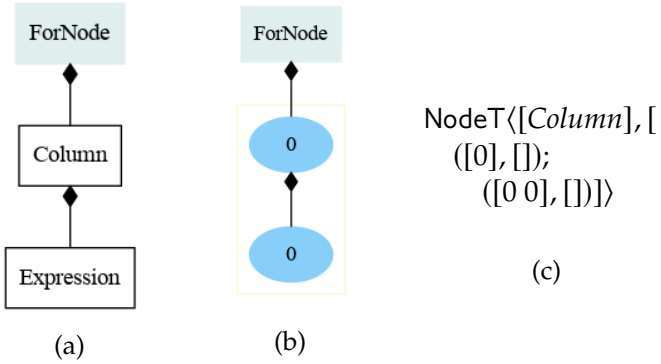
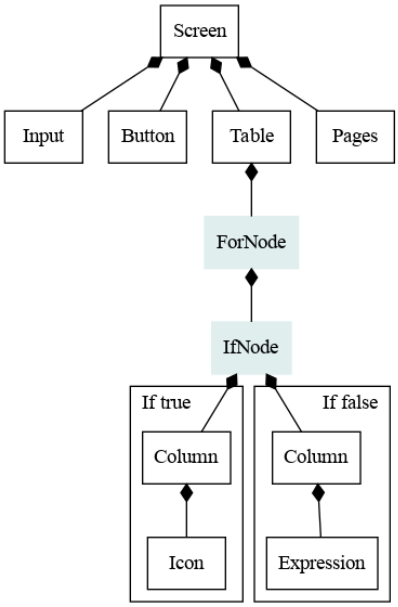


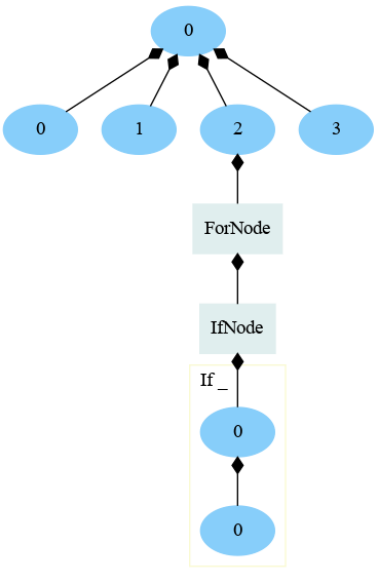
Figure 5.3: Simple example of ForNode, respective path and type.

The ForNode produces just one child, independently of the number of attributes it is ranging over. This consistency happens because, at the moment of typechecking, we do not know anything about those attributes. If we consider an example of a ForNode over an arbitrary number of attributes - just like it is illustrated in Figure 5.3a -, we get a path that corresponds to the one produced by the child node - shown in Figure 5.3b. The type of such tree includes the category and the pair that matches the path to the respective properties of the child node (presented in Figure 5.3c).

To end this subsection, let's consider, once again, our example application. This time, we will use it just as it was presented in see Figure 4.1. In that application, we have two attributes. One for the product description and another to display an icon indicating whether there is stock left of that product or not. From this, we can see a conditional situation: if we are on the description, then we will be presented with an *Expression*; if we are on the indication on whether there is stock, then we are presented with an *Icon*. We can now build our IfNode from this information, just like it is illustrated in Figure 5.4a. We assume that the IfNode is evaluating the respective attribute as either being a boolean or not. Since we have two attributes, we can include this IfNode on a ForNode. All the rest of our node tree is the same as we have seen before. Finally, the path follows the previously approach: the IfNode produces an intersection of the sets of the true and false side, which is just the path [0 0] (that is, one *Column* and either an *Expression* or an *Icon*); the ForNode produces what is originated by its child node. This is illustrated in Figure 5.4b. Moreover, and since we have not yet presented an example of our syntax using an IfNode



(a) Widgets tree.



(b) Path tree.

```
Node⟨Screen, [], [
  Node(Input, [], []),
  Node(Button, [], []),
  Node⟨Table, [], [
    ForNode⟨attr in Entity.attrs, [
      IfNode⟨attr isOfType Boolean, [
        Node(Column, [], [Node(Icon, [], [])])
      ], [
        Node(Column, [], [Node(Expression, [], [])])
      ]⟩
    ]⟩
  ]⟩,
  Node(Pages, [], [])
]⟩
```

(c) Syntax of the tree.

```
NodeT⟨[Screen], [
  ([0], []);
    ([0 0], []);
    ([0 1], []);
    ([0 2], []);
    ([0 2 0], []);
      ([0 2 0 0], []);
    ([0 3], [])]⟩
```

(d) Node type.

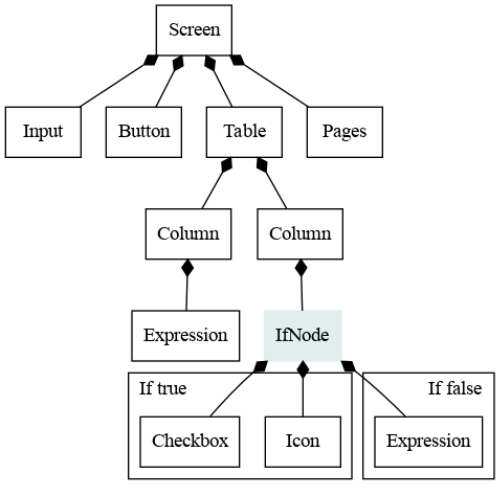
Figure 5.4: A more complex example of IfNode and ForNode usage.

nor a ForNode, we could represent this tree as it is displayed in Figure 5.4c. In it, we are ranging over the attributes (*attr*) of the entity (*Entity.attr*) product of our example. Lastly, Figure 5.4d presents the type of such tree. The category of the node type is the one that matches the root node, *Screen*, and the list of paths and properties corresponds to the information of every node on the tree.

5.2 Hybrid Typechecking

When we presented the possible customizations, we briefly discussed what our type system must check to avoid execution errors. In our language, typechecking can either be static or dynamic. These have different strengths and weaknesses [25] [10]. Static type checking provides better documentation, the errors are detected early, and the program execution is more efficient, whereas dynamic typing enables rapid development and fast adaptation to changing requirements.

To further introduce hybrid type systems - that is, type systems that combine static and dynamic checking - we will consider a simple application. Our application is presented in Figure 5.5a and similar to the one we have show in Figure 4.1. We can see that one of the columns can either display a *Checkbox* and an *Icon*, or simply an *Expression*. Furthermore, we saw from Section 5.1 that when we encounter this type of conditional situation, what our type system knows is as much as the two clusters have in common. That is the intersection between paths and properties. Following this logic, the information retrieved from this application is shown in Figure 5.5b. The type has the node categories, which in this case is just *Screen*, and a list of paths and respective properties. For instance, the pair ([0],[]) means that at path 0 we have an empty list of properties - we decided to disregard them as it makes the example simpler. More interestingly, we can check that below [2 1 0], we only encounter [0 2 1 0] and not [0 2 1 1], which would correspond to the possible path of our *Icon*.



(a) Example application.

$$\mathsf{NodeT}\langle[\mathit{Screen}], [$$
$$([0], []);$$
$$([0;0], []);$$
$$([0;1], []);$$
$$([0;2], []);$$
$$([0;2;0], []);$$
$$([0;2;0;0], []);$$
$$([0;2;1], []);$$
$$([0;2;1;0], []);$$
$$([0;3], [])]\rangle$$

(b) Result of the application typechecking.

Figure 5.5: Example application and respective typechecking.

If we now consider the situation in which we want to add the property *Title* with the value "Icon" to the *Icon* node, we would do the following, using our syntax:

$$t \otimes \mathsf{addProp}([0\ 2\ 1\ 1], \{\mathit{Title} = \text{“}\mathit{Icon}\text{”}\})$$

where t is the initial tree of Figure 5.5. From now on, and until we say otherwise, name t

will always represent that tree.

If we only check type information statically, we can do one of two things: accept or reject the execution of such a program. However, we do not have sufficient information to make a decision. If we decide to accept it, we may be allowing the execution of programs that cause errors, as specifications are later violated at run-time, but if we reject it, we may be defecting programs that end up being well-typed. For example, if we check statically if the given path is valid, then we would be rejecting the previous customization at all times since the typechecker does not know about the possible existence of the path [0 2 1 1].

On the other hand, if we only check the programs dynamically, our specifications will only be checked on data values of actual executions. Moreover, dynamic checking results in incomplete and late detection of defects, apart from consuming cycles that could be used to perform useful computation. For instance, it is not worth evaluating a program at run-time that does not have the correct types, which can be determined statically.

To work around these issues, the solution is to combine both static and dynamic checking. We should check the programs statically whenever possible and dynamically whenever necessary. Thus, we can benefit from the advantages of these two techniques: the defects will be detected early, and all well-typed programs are statically accepted.

We will now see which situations can be statically checked and which ones should be dynamically checked.

5.2.1 Static Typechecking

From what we have already studied, we need to check the types and everything related to properties, paths and indexes (in case the customization regards node insertion or node movement). However, we do not have access to the information of these last three aspects at compile-time, as we have seen from the previous example illustrated in Figure 5.5. Therefore, we will leave everything related to that to be checked at run-time.

Let's now consider that we try to do the following operation over our example tree:

$$t \otimes \mathsf{remove}(\mathsf{Node}\langle \mathit{Checkbox}, [], [] \rangle)$$

In this customization, we are trying to remove a *Checkbox* from the tree. However, we have seen that the customization remove requires a path instead of a node. Thus, this program will not be executed at run-time, as it will be rejected at compile-time. We can consider that all operations similar to this example, in which the type of the arguments is incorrect, will be immediately rejected.

Moreover, we have to decide which information our types will hold when we perform customizations over a tree, as we will accept all programs in which the types are correct. Below, we study each customization one by one.

Insert node. The path and properties of the node for insertion will always be added to the type information, except on one occasion: if the index received is a negative number,

as we already know this will never correspond to a valid index within the sequence of siblings' nodes. Apart from that situation, the index will solely be checked dynamically, as already stated. For example, if we wanted to add another *Checkbox* node after the *Icon* one, we would do the following:

$$t \otimes \mathsf{insert}([0\ 2\ 1], 2, \mathsf{Node}\langle \mathit{Checkbox}, [], [] \rangle)$$

This program would be accepted, even if it is an impossible execution in case the IfNode evaluates to false because the index of the last element (the *Expression* node) sits at position 0 and not at 1.

The resulting type is as displayed in Figure 5.6, in which we use the ellipsis to indicate that the path and properties remain the same, except for the one being expanded. So, in this case, the children in the node we reached from the path [0 2 1] will suffer a modification while the others will not.

$$\begin{array}{l} \mathsf{NodeT}\langle [\mathit{Screen}], [\\ \quad \ldots \\ \qquad ([0;2;1], []); \\ \qquad\quad ([0;2;1;0], []); \\ \qquad\quad ([0;2;1;2], []) \rangle \end{array}$$

Figure 5.6: Example of insert typechecking.

Remove node. The node that corresponds to the given path is removed. The program will be statically accepted even if that path does not appear on the type. We do this because, as we have seen, the path can be valid upon execution even if it is not present on the type. For instance, if we have the following customization:

$$t \otimes \mathsf{remove}([0\ 2\ 1\ 1])$$

The final result would be as previously shown in Figure 5.5b. That is, no modifications occur, and the program will later be evaluated at run-time. We will study in the upcoming Subsection 5.2.2 what will happen if we cannot reach any node from the given path.

Replace node. As happened with the node insertion, the path and properties will always be added to the type in case they do not belong to it already. Let's consider we perform the following customization:

$$t \otimes \mathsf{replace}([0\ 2\ 1\ 1], \mathit{Node}\langle \mathit{Input}, [], [] \rangle)$$

We will be replacing the *Icon* node - in case the execution evaluates to true - or no node at all - in case the execution evaluates to false - with the *Input* node. The final type would be exactly like the one presented in Figure 5.7. We can notice that the path for replacement appears now on the type.

$$\begin{array}{l}\mathsf{NodeT}\langle[\mathit{Screen}], [\\ \quad \ldots \\ \quad\quad ([0;2;1], []); \\ \quad\quad\quad ([0;2;1;0], []); \\ \quad\quad\quad ([0;2;1;1], [])\rangle\end{array}$$

Figure 5.7: Example of replace and remove property typechecking.

Move node. The final type will have the target path and respective properties modified, even if the former given does not appear in it. For instance, let's imagine we perform the customization as follows:

$$t \otimes \mathsf{move}([0\ 2\ 1\ 1], 0)$$

In this customization, we are trying to either move the *Checkbox* node (in case branch evaluates to true) or no node at all (in case branch evaluates to false) to appear as the first element of its siblings' node list. Since we do not have the path [0 2 1 1] on the type, we will only modify [0 2 1 0] to appear as the former (since the node is moved to occupy the first place, we need to add one more position to every node on the correspondent list). The final result is presented in Figure 5.8.

$$\begin{array}{l}\mathsf{NodeT}\langle[\mathit{Screen}], [\\ \quad \ldots \\ \quad\quad ([0;2;1], []); \\ \quad\quad\quad ([0;2;1;1], [])\rangle\end{array}$$

Figure 5.8: Example of move typechecking.

Add property. When adding a property, the type will retain the path for insertion together with the properties to be added. However, if these last ones already exist in the given path, then nothing will be changed. For instance, if we perform the following customization:

$$t \otimes \mathsf{addProp}([0\ 2\ 1\ 1], \{\mathit{Title} = \text{“}\mathit{Icon}\text{”}\})$$

Then we will be adding the pair [0 2 1 1] and the property *Title* to our type, as neither of those two appeared on the type specification. The result would be as the one shown in Figure 5.9.

Replace property. Just as it happened when adding a property, the path and properties will be added to the type. If they already exist in it, then, once again, nothing will be changed. For example, in the case of the following customization:

$$t \otimes \mathsf{replaceProp}([0\ 2\ 1\ 1], \{\mathit{Title} = \text{“}\mathit{Icon}\text{”}\})$$

$$\begin{array}{l}\mathsf{NodeT}\langle[\mathit{Screen}],[\\ \quad\ldots\\ \quad\quad([0;2;1],[]);\\ \quad\quad\quad([0;2;1;0],[]);\\ \quad\quad\quad([0;2;1;1],[(\text{“}\mathit{Title}\text{”},\mathit{StringT}])\rangle\end{array}$$

Figure 5.9: Example of add and replace property typechecking.

We would be adding the path [0 2 1 1] and the property *Title*, as neither of these information appeared previously on the type. The result would be the same we have seen before in Figure 5.9. On the other hand, if the original tree for this customization had the type presented in Figure 5.9, the result would still be the same, as we would not need to add anything to the type.

Remove property. If the path and properties received exist, then we will be deleting the properties, and keeping the path (as the node remains present when performing such customization). In contrast, if the path does not exist, then we will be adding it, but not the properties, since the final result will never contain them. As an example, consider the following customization:

$$t \otimes \mathsf{removeProp}([0\ 2\ 1\ 1],[\mathit{Title}])$$

From it, we can see that we are trying to remove from the target node reached from the path [0 2 1 1] the property Title. However, in our type, it does not appear such path. In that case, we will be adding it. The final result is presented in Figure 5.7.

5.2.2 Dynamic Typechecking

In the examples presented in Chapter 4, we have seen what happens if we perform a customization successfully (see Section 4.1). However, the parameters given are not always valid, as we have seen previously. At run-time, we will have to check if the (positive) index is valid in case the customization is to insert or move a node, and the path is valid, for all customizations. Moreover, we also have to study the given properties in case the operation regards them.

Considering the example in Figure 5.5, let's imagine that the lfNode evaluated to true. Then we know that the index under the second *Column* ranges from 0 to 2 in case of insertion, or 0 to 1 in case of node movement, as the node has two children (the *Checkbox* and the *Icon* node, respectively). Thus, if we try to perform any of the following customizations:

$$t \otimes \mathsf{insert}([0\ 2\ 1], 5, \mathit{Node}\langle \mathit{Checkbox}, [], []\rangle)$$

$$t \otimes \mathsf{move}([0\ 2\ 1\ 1], 2)$$

The resulted tree will not be any different than the initial tree. Both the customizations fail because the value provided for the index is too high. This example is illustrated in Figure 5.10.

Similarly, if the path is invalid and we cannot reach any node from it, the final tree would remain the same as the initial one. This transition without modifications is what happens if we try to replace the inexistent second child of the *Checkbox* node for a *Calendar* node. Or using our syntax:

$$t \otimes \mathsf{replace}([0\ 2\ 1\ 0\ 1], Node\langle Calendar, [], [] \rangle)$$

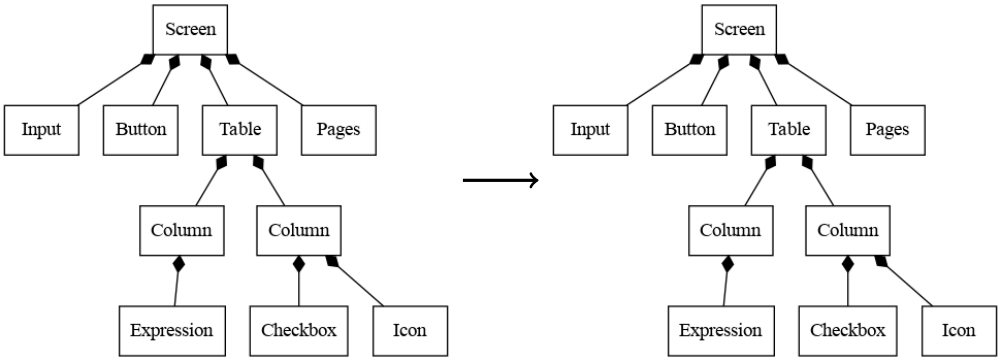


Figure 5.10: Example of typechecking dynamically.

To end this subsection, we give one last example. This example will take as the original tree the one displayed on the left of Figure 5.11, which is similar to the one used in the previous example, except for the node properties. We can see that the second *Column* has two different attributes: one for the *Title* and one for the *Visibility*. Let's consider we try the following customization:

$$t \otimes \mathsf{removeProp}([0\ 2\ 1], [Title, Description])$$

Or using words, we are trying to remove a list of properties containing *Title* and *Description* as the property identifiers from the second *Column* node of our example. Since everything that regards properties is accepted statically, this program will be evaluated at run-time as all the given types match the ones expected. However, and as we can check, our *Column* node does not have associated with it the two properties requested to be removed (the one missing is *Description*), so it will only try to remove the *Title*. Consequently, this last attribute will be deleted, and all the others will remain present. The result of executing such an operation is fully illustrated in Figure 5.11. We follow a similar approach for all customizations regarding properties.

5.3 Typing Rules

In this section, we will focus solely on the types involved in customizations. However, the reader can find a list of the complete typing rules of OSTRICH language in [17]. We will

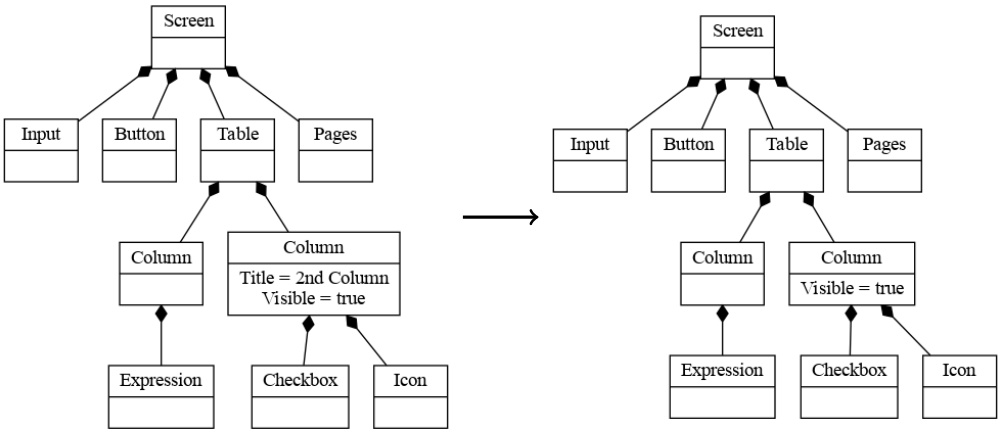


Figure 5.11: Example of typechecking dynamically.

also use the same typing judgment form by reason of consistency. Keeping it implies that a judgment will assume the form $\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash M : T$, in which M represents the term, where free variables occur, with type T. We use five different environments to represent: (1) the ordinary context that stores compile-time variables, Γ; (2) the run-time context, Δ; (3) a context to store polymorphic variables' name, type, and rows, defined respectively by Ω,Φ, and Υ.

Before we move on to the typing rules of customizations, we expose three rules for integers (T_INT), Strings (T_STRING) and path (T_PATH), that consists of a list of integers. These are important since we use these two literals and the list to represent respectively the index, the identifier we provide to remove a target property, and the path to reach some node of the tree.

$$\frac{}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash int : \text{Integer}}\ \text{T\_INT} \qquad \frac{}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash string : \text{String}}\ \text{T\_STRING}$$

$$\frac{\text{for each} \quad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash v : \text{Integer}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash \{\overline{v : \text{Integer}}\}}\ \text{T\_PATH}$$

Both the T_INT and T_STRING rule are straightforward. We consider a path well-typed if every element belonging to it assumes the type Integer, as the T_PATH rule shows.

The typing rule for the different customizations varies according to the arguments required to perform them successfully. We will now present each separately. We will also represent the type of a Node as $\mathsf{NodeT}(A, C)$, instead of $\mathsf{NodeT}([\overline{\alpha}], [\overline{(pt, [\overline{p}])}])$ as we used when studying the syntax of types in Section 2.3, since it is easier to read.

Insert node. The typing rule to insert a node is well-typed if every type of each given argument matches the expected type. This correspondence means that our original tree has the Node type, the path is composed solely of integers, the index is an integer superior to 0 (we have seen before that any index below this value is considered to be invalid), and

the term to insert is of type Node.

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash O : \mathsf{NodeT}(A, C) \quad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash P : \{\overline{v : \text{Integer}}\} \quad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash I : \text{Integer}, I \geq 0 \quad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash N : \mathsf{NodeT}(A, C)}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash O \otimes \mathsf{insert}(P, I, N)} \text{ T\_INSERT}$$

Remove node. This customization adds nothing new to what we have seen previously. We have to simply check if the original tree is of type Node and that the path is composed of Integers.

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash O : \mathsf{NodeT}(A, C) \quad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash P : \{\overline{v : \text{Integer}}\}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash O \otimes \mathsf{remove}(P)} \text{ T\_REMOVE}$$

Replace node. The typing rule to replace a node is very similar to the one to add a node, with the exception that we do not have to check the index, as this argument is not required to perform such customization. Therefore, we have to check if the given path is composed solely of integers, and if our original tree and term to replace are of type Node.

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash O : \mathsf{NodeT}(A, C) \quad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash P : \{\overline{v : \text{Integer}}\} \quad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash N : \mathsf{NodeT}(A, C)}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash O \otimes \mathsf{replace}(P, N)} \text{ T\_REPLACE}$$

Move node. To move a node, we need the node path and the index that the target node is going to assume in its siblings' list. The path is composed of integers, as we have seen previously, and the index follows the same logic as in the typing rule to insert a node: it should be an integer equal to or greater than 0. We also need to check the type of the original tree.

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash O : \mathsf{NodeT}(A, C) \quad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash P : \{\overline{v : \text{Integer}}\} \quad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash I : \text{Integer}, I \geq 0}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash O \otimes \mathsf{move}(P, I)} \text{ T\_MOVE}$$

Add property. Having the typing rule to add a property to be well-typed means that, apart from checking the types for the original tree and the path received, we have to check the type of each value assumed by the new properties. This type can be any of the possible ones.

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash O : \mathsf{NodeT}(A, C) \quad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash P : \{\overline{v : \text{Integer}}\} \quad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash v : T}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash O \otimes \mathsf{insertProp}(P, \{\overline{L = v : L = T}\})} \text{ T\_INSERTPROP}$$

Replace property. The typing judgments to the replace properties typing rule are similar to the previous customization, as the reader can confirm.

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash O : \mathsf{NodeT}(A, C) \quad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash P : \{\overline{v : \text{Integer}}\} \quad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash v : T}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash O \otimes \mathsf{replaceProp}(P, \{\overline{L = v : L = T}\})} \text{ T\_REPLACEPROP}$$

Remove property. To remove a property, we need to check the type of the given labels, which should be of type String. The common typing judgments for the path and the original tree are required once again.

$$\frac{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash O : \mathsf{NodeT}(A, C) \quad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash P : \{\overline{v : \text{Integer}}\} \quad \text{for each} \quad \Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash L : \text{String}}{\Gamma;\Delta;\Omega;\Phi;\Upsilon \vdash O \otimes \mathsf{removeProp}(P, \{\overline{L : \text{String}}\})} \text{ T\_REMOVEPROP}$$

6

Implementation

In this chapter, we will briefly show how we implemented the customizations. We will first take a look at how our typechecker behaves when presented with one, and then we will move on to its respective evaluation. To not overbear the reader, we will just study the insert node customization, as all the others follow a similar reasoning.

```
let typecheck_insert e envs =
    match e with
    | t1 ⊗ insert(p,i,t2) ->
        let t1' = typecheck t1 envs in
        let p' = typecheck p envs in
        let p'' = begin match p' with
            | PathT _ -> extract_path p
            | _-> raise (Exception)
        end in
        let i' = typecheck i envs in
        let t' = typecheck t envs in
        begin match t1',i',t' with
            | NodeTP(n,l), NumT, NodeTP(_,l2) ->
                let idx = match i with
                    | Num(n) -> n
                    | _ -> assert false
                in
                if idx < 0 then
                    raise (Exception)
                else
                    let l = update_indexes l l2 i in
                    NodeTP(n, append l' r )
            | _ -> raise (Exception)
        end
    | _ -> raise (Exception)
```

Listing 1: Insert typechecking.

The code for the insert typechecking is shown in Listing 1. It is important to note that

the code, as it is presented, was slightly simplified in the interest of comprehension.

The typechecking function requires two types of input: one representing the expression to be evaluated (`e`), and the other representing the environments needed to store variables (`envs`). After that, the following will happen:

1. We will see if the expression received matches an insert customization (line 3). If not, we will raise an exception (line 19), and the typechecking will stop here;
2. We will obtain the type of every term as we need to check if they match the expected (lines 4,5,10, 11, and 14-17);
3. As stated, we will compare the returned types to the ones we are expecting (line 13). If they do not match, we will raise an exception (line 23), and the typechecking stops here;
4. We will check if the given index is a number greater or equal to zero, as we know that any negative number is invalid for this operation (line 18);
5. When we reach line 20, we already know that everything works as expected, and we can proceed with this customization evaluation. All that we have to do to return the type (line 22) is update the position of every tree element (line 21).

```
1  let eval_tree_insert  e envs =
2      match e with
3      | t1 ⊗ insert(p,i,t2) ->
4          let t1' = List [eval t1 []] in
5          let t1'' = match t1' with
6              | List xs -> map (fun n -> eval n env) xs
7              | _ -> raise (Exception)
8          in
9          let is_path_valid,node = check_path_validity t1'' p in
10         let is_index_valid = check_index_validity op i p in
11         if not (is_path_valid && is_index_valid) then
12            t1''
13         else begin
14             let t2' = eval t2 [] in
15             let is_t2_valid,t2'' = check_t2_validity op t2' node in
16             if not (is_t2_valid) then
17                 t1''
18             else insert t1'' p i t2''
19     | _ -> raise (Exception)
```

Listing 2: Insert evaluation.

The code for the insert evaluation is shown in Listing 2. Just like we did for the typechecking function, we simplified the code for the evaluation to accommodate only the insert customization.

For the evaluation to work, it needs the same two inputs as before: one for the expression to be evaluated (`e`), and the other for the environments needed to store variables (`envs`).

After that, the following will happen:

1. We will see if the expression received matches an insert customization (line 3). If not, we will raise an exception (line 19), and the evaluation will stop here. This first step is the same as the first one for the typechecking function;
2. After that, we will need to evaluate the original tree, `t1` (line 4). The result will be stored in a list that will have its elements further evaluated (line 6);
3. We will check if the path provided by the user is valid. If it is, we will store the target node in the variable `node` (line 9);
4. Similarly to the previous step, we will check if the given index is valid (line 10);
5. If 3) or 4) fail, then we will return the original tree without applying any modification (lines 11 and 12);
6. If the index and the path are valid, then we will evaluate the node that is going to be inserted into the tree (line 14);
7. We will then check if the term is valid (line 14);
8. If the previous step fails, we will return the original tree (line 17). Otherwise, we will perform the customization (line 18) and return the resulting tree.

6.1 Testing

Now that we have studied the implementation for the typechecking and evaluation of the insert node customization, we can check if both functions work as expected. We will used the original tree we previously studied in Chapter 4 examples. To remember the reader, this tree is illustrated in Figure 6.1. On the other hand, we will have the *Checkbox* node as the one that is going to be inserted into the original tree.

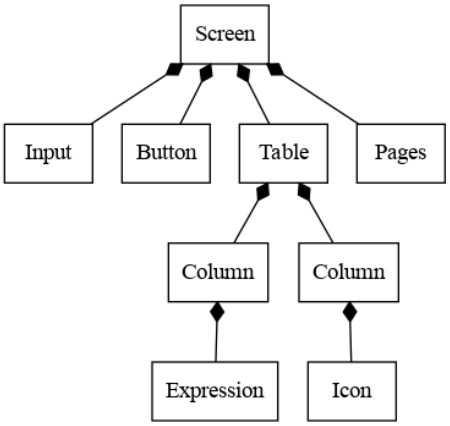


Figure 6.1: Original tree.

The representation of this tree code is shown in Listing 3. The variable `screen` represents the original tree, while the variable `cb` represents the *Checkbox* node to insert. We take this opportunity to remember that a node is represented as `Node`, and receives

three arguments: its category, a list of its attributes, and a list of its children. As an example, in line 4, we create a *Column* node with 0 attributes and one child named `i1`.

```
1    let exp1 = Node(Expression, [], List [])
2    let c1 = Node(Column, [], List [exp1])
3    let i1 = Node(Icon, [], List [])
4    let c2 = Node(Column, [], List[i1])
5    let t = Node(Table, [], List [c1;c2])
6    let inp = Node(Input, [], List [])
7    let pg = Node(Pages, [], List [])
8    let btn = Node(Button, [], List [])
9    let screen = Node(Screen, [], List [inp; btn; t; pg])
10
11   let cb = Node(CheckBox, [], List [])
```

Listing 3: Code for the original tree and node to insert.

In this section, we will study three types of insert node customization:

1. One that works well in both typechecking and evaluation;
2. One that works well in typechecking but does not meet the criteria to perform the actual operation in the original tree upon evaluation;
3. One that fails the typechecking and, therefore, the evaluation.

For all the three situations, we will always use the presented original tree and the *Checkbox* node. Without further due, let's approach these case studies.

1 For this example, we will consider the already studied customization (see Section 4.2 for a more detailed description):

$$t \otimes \mathsf{insert}([0], 2, \mathsf{Node}(\mathit{Checkbox}, [], []))$$

In it, we are trying to insert our *Checkbox* node in between the *Button* and *Table* node. We will use t to represent our original tree.

The result of the insert customization typechecking is shown in Listing 4. We can confirm that the positions of every node were updated, and while the type does not tell us the category of each node for each position, it is safe to say that the *Table* node moved from position 2 to 3. We can say this because this node has 2 children (the *Column* nodes), while the *Table* siblings did not have any. We can also assume that the *Checkbox* assumed the position [0 2], which was what caused the *Table*, and therefore the *Pages* node, to have its positions updated.

The evaluation for this customization is represented in Listing 5. We can see that the resulting type has the *Checkbox* node, positioned in between the *Button* and *Table* node, just as it was expected.

```
let result = typecheck_insert (screen ⊗ insert([0],2,cb)) {}
```

```
val result : ty =
    NodeTP ([Screen], [
        ([0], []);
            ([0; 0], []);
            ([0; 1], []);
            ([0; 2], []);
            ([0; 3], []);
                ([0; 3; 0], []);
                    ([0; 3; 0; 0], []);
                ([0; 3; 1], []);
                    ([0; 3; 1; 0], []);
            ([0; 4], [])
                     ])
```

Listing 4: Insert customization typechecking example, index 2.

```
let result = eval_tree_insert (screen ⊗ insert([0],2,cb)) {}
```

```
val result : term list =
    [Node (Screen, [], List [
        Node (Input, [], List []);
        Node (Button, [], List []);
        Node (CheckBox, [], List []);
        Node (Table, [], List [
            Node (Column, [], List [Node (Expression, [], List [])]);
            Node (Column, [], List [Node (Icon, [], List [])])
                                ]);
        Node (Pages, [], List [])
                               ])]
```

Listing 5: Insert customization evaluation example, index 2.

2 In this example, we will be trying to insert the *Checkbox* node as the fifth child of the *Screen* node. Using our syntax, the customization is represented as follows:

$$t \otimes \mathsf{insert}([0], 5, \mathsf{Node}(\mathit{Checkbox}, [], []))$$

Since we know the structure of our original tree, we already know that this should fail as the index needs to be a number between 0 and 4. However, as we studied in Chapter 5, our typechecking function does not make any assumptions about the index, apart from imposing that it should be a number greater or equal to 0. This way, the path for insertion, [0 5], will always be added to the type. Everything that was previously presented in the type will remain the same. The result of this operation is shown in Listing 6.

```
1    let result = typecheck_insert (screen ⊗ insert([0],5,cb)) {}
```

```
val result : ty =
    NodeTP ([Screen], [
        ([0], []);
            ([0; 0], []);
            ([0; 1], []);
            ([0; 2], []);
                ([0; 2; 0], []);
                    ([0; 2; 0; 0], []);
                ([0; 2; 1], []);
                    ([0; 2; 1; 0], []);
            ([0; 3], []);
            ([0; 5], [])
                    ])
```

Listing 6: Insert customization typechecking example, index 5.

At the moment of evaluation, our function will detect that the index 5 for the given path is invalid. Since this happens, the tree will not suffer any modification, and the returned output will be the original tree. This example is represented in Listing 7.

```
1    let result = eval_tree_insert (screen ⊗ insert([0],5,cb)) {}
```

```
val result : term list =
    [Node (Screen, [], List [
        Node (Input, [], List []); Node (Button, [], List []);
        Node (Table, [], List [
            Node (Column, [], List [Node (Expression, [], List [])]);
            Node (Column, [], List [Node (Icon, [], List [])])
                                ]);
        Node (Pages, [], List [])
                            ])]
```

Listing 7: Insert customization evaluation example, index 5.

3 Our last example will force our customization to be rejected at the moment of typechecking. To accomplish this, we will try to insert the *Checkbox* node in the position -1 for the path $[0]$:

$$t \otimes \mathsf{insert}([0], -1, \mathsf{Node}(\mathit{Checkbox}, [], []))$$

As previously stated, since the index is a negative number, it will never pass the typechecking operation. We can see this exception raising in lines 18 and 19 of the

Listing 1, in which we presented the typechecking function. The result of this operation shows this exact failure in Listing 8.

```
1    let result = typecheck_insert (screen ⊗ insert([0],-1,cb)) {}
```

```
        Exception: Exception "Index needs to be bigger than 0."
```

Listing 8: Insert customization typechecking example, index -1.

We do not show the evaluation for this customization since we know that the typechecking operation failure will prevent it from being evaluated.

The complete code of this work can be found here.

7

Conclusion

Throughout this thesis work, we studied the template language OSTRICH and the concepts relevant to the low-code development field. The goal is to integrate this work into the OutSystems platform, Service Study. Currently, this platform has a few limitations that could be overcome with the usage of our template language. For instance, the available screen templates always come with dummy data. Although this, indeed, helps the user understand how the design of the screen, it is also true that the user will be obligated to adapt the template to their database entries, an activity that could reveal itself as cumbersome and complicated. Furthermore, there is no support to track the changes applied to a template. Thus, if the template suffers a modification, from an update to changing parameters, the developer would have to perform all customization operations on top of a new template instantiation to reach the same point. This dissertation approaches this last issue.

To find a solution that did not obligate the user to repeat all customizations, we covered several steps. First, we decided which operations should be tracked. This process resulted in studying seven different customizations that involved the structure of the whole template as well as particular elements of it. Afterward, we had to decide how to evaluate such customizations and how to ensure the evaluation occurred as expected. For example, some customizations may target an element of the old template that does not correspond to the one that occupies its place on the new template. In such an occurrence, we need to determine if there are still conditions to operate or if we should just ignore the customizations. Moreover, we also have to choose the moment we make such a judgment, with the options being compile-time or run-time. We explored the advantages and disadvantages of each of these, concluding that we should take action as early as possible so as not to waste cycles that could otherwise be performing useful computation and not reject well-typed programs. In the end, these cautions resulted in a hybrid type system. We took the time to study different situations to better understand how our type system approaches distinct programs.

Despite this work being finished, some of its aspects could be worked on and improved. For instance, it would be helpful to compare two different templates and have a way to

obtain the sequence of customizations that could lead one to the other.

Bibliography

[1] M. Brambilla, J. Cabot, and M. Wimmer. *Model-driven software engineering in practice*. Vol. 3. 1. Morgan & Claypool Publishers, 2017, pp. 1–207 (cit. on pp. 1, 4).

[2] L. Cardelli. *Phase distinctions in type theory*. 1988 (cit. on p. 8).

[3] L. Cardelli. "Type systems". In: *ACM Computing Surveys (CSUR)* 28.1 (1996), pp. 263–264 (cit. on p. 6).

[4] L. Cardelli and P. Wegner. "On understanding types, data abstraction, and polymorphism". In: *ACM Computing Surveys (CSUR)* 17.4 (1985), pp. 471–523 (cit. on pp. 6, 7).

[5] D. Clément et al. "A simple applicative language: Mini-ML". In: *Proceedings of the 1986 ACM conference on LISP and functional programming*. 1986, pp. 13–27 (cit. on p. 35).

[6] B. Combemale et al. *Engineering modeling languages: Turning domain knowledge into tools*. CRC Press, 2016 (cit. on p. 4).

[7] O. community. *Service Studio Overview*. [Online; accessed 3-December-2021]. 2020-12. URL: https://success.outsystems.com/Documentation/11/Getting_started/Service_Studio_Overview (visited on 2021-12-03) (cit. on p. 1).

[8] G. Cousineau, P.-L. Curien, and M. Mauny. "The categorical abstract machine". In: *Science of Computer Programming* 8.2 (1987), pp. 173–202. ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(87)90020-7 (cit. on p. 35).

[9] R. Davies and F. Pfenning. "A modal analysis of staged computation". In: *Journal of the ACM (JACM)* 48.3 (2001), pp. 555–604 (cit. on pp. 23, 31, 32).

[10] C. Flanagan. "Hybrid type checking". In: *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2006, pp. 245–256 (cit. on p. 51).

[11] Z. Hu, J. Hughes, and M. Wang. "How functional programming mattered". In: *National Science Review* 2.3 (2015), pp. 349–370 (cit. on p. 10).

[12] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993 (cit. on p. 8).

[13] 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)*. IEEE. 2021, pp. 216–226 (cit. on pp. 1, 2, 13, 16, 22).

[14] 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).

[15] G. Michaelson. *An Introduction to Functional Programming Through Lambda Calculus*. Dover Publications, 1989 (cit. on p. 11).

[16] J. Palsberg. "Correctness of binding-time analysis". In: *Journal of functional programming* 3.3 (1993), pp. 347–363 (cit. on p. 34).

[17] J. Parreira. "From an ontology for programming to a type-safe template language". MA thesis. Nova SST, 2022 (cit. on p. 56).

[18] H. T. L. de Paula. *A Brief Introduction to Template Haskell*. [Online; accessed 9-December-2021]. 2021. URL: https://serokell.io/blog/introduction-to-template-haskell (visited on 2021-12-09) (cit. on p. 27).

[19] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002 (cit. on pp. 5, 11).

[20] C. M. Portugal. *GOLEM*. [Online; accessed 6-December-2021]. 2020. URL: https://www.cmuportugal.org/large-scale-collaborative-research-projects/golem/ (visited on 2021-12-06) (cit. on p. 2).

[21] M. L. Scott. *Programming language pragmatics*. Morgan Kaufmann, 2000 (cit. on p. 10).

[22] J. C. Seco et al. "Nested OSTRICH - Hatching Compositions of Low-code Templates". In: *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE. 2022 (cit. on p. 13).

[23] T. Sheard. "Using MetaML: A staged programming language". In: *International School on Advanced Functional Programming*. Springer. 1998, pp. 207–239 (cit. on pp. 7, 23).

[24] T. Sheard and S. P. Jones. "Template meta-programming for Haskell". In: *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. 2002, pp. 1–16 (cit. on pp. 27, 37).

[25] J. Siek and W. Taha. "Gradual typing for objects". In: *European Conference on Object-Oriented Programming*. Springer. 2007, pp. 2–27 (cit. on p. 51).

[26] W. Taha and T. Sheard. "MetaML and multi-stage programming with explicit annotations". In: *Theoretical computer science* 248.1-2 (2000), pp. 211–242 (cit. on pp. 7, 23).

[27] M. Voelter et al. “DSL engineering-designing, implementing and using domain-specific languages”. In: *Voelter DSL Book* (2013) (cit. on p. 5).

[28] P. Wadler. “Propositions as types”. In: *Communications of the ACM* 58.12 (2015), pp. 75–84 (cit. on p. 30).