

Derivations with Holes for Concept-Based Program Synthesis*

João Costa Seco
NOVA University Lisbon,
NOVA LINC
Caparica, Portugal
joao.seco@fct.unl.pt

Jonathan Aldrich
Carnegie Mellon University
Pittsburgh, United States of America
jonathan.aldrich@cs.cmu.edu

Luís Carvalho
NOVA University Lisbon,
NOVA LINC
Caparica, Portugal
la.carvalho@campus.fct.unl.pt

Bernardo Toninho
NOVA University Lisbon,
NOVA LINC
Caparica, Portugal
bernardo.toninho@fct.unl.pt

Carla Ferreira
NOVA University Lisbon,
NOVA LINC
Caparica, Portugal
carla.ferreira@fct.unl.pt

Abstract

Program synthesis has the potential to democratize programming by enabling non-programmers to write software. But conventional approaches to synthesis may fail if given insufficient information—a common occurrence when asking non-experts to describe the application they want to write. This paper introduces a new concept-based program synthesis mechanism that can cope with incomplete knowledge, targeting low-code model-driven languages. Concepts are modelled in an ontology that represents user intent, including basic actions (e.g. show, filter, and create) along with their associated data as well as basic user interface structures like screens or pages. Our synthesis framework consists of a system of derivation rules that supports deferred premises, which need not be immediately satisfied during synthesis. A derivation in which some deferred premises are missing will thus contain holes; semantically, it represents a proof that is conditional on the developer filling the holes with additional facts from the ontology. We translate derivations with holes to standard first-order logic derivations, where the holes are transformed into assumptions. We illustrate the feasibility and effectiveness of our framework with a proof-of-concept implementation and a set of illustrative examples.

*This work is supported by FCT/MCTES under Grant NOVA LINC - UIDB/04516/2020 and GOLEM Lisboa-01-0247-Feder-045917.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Onward! '22, December 8–10, 2022, Auckland, New Zealand

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-9909-8/22/12.

<https://doi.org/10.1145/3563835.3567658>

CCS Concepts: • Theory of computation → Constraint and logic programming; • Software and its engineering → Automatic programming.

Keywords: low-code, program synthesis, automated programming, user intent, incomplete knowledge

ACM Reference Format:

João Costa Seco, Jonathan Aldrich, Luís Carvalho, Bernardo Toninho, and Carla Ferreira. 2022. Derivations with Holes for Concept-Based Program Synthesis. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '22)*, December 8–10, 2022, Auckland, New Zealand. ACM, New York, NY, USA, 17 pages. <https://doi.org/10.1145/3563835.3567658>

1 Introduction

Software has revolutionized productivity in numerous fields, yet there remain many tasks for which software support is missing. A key barrier is that only programmers can program: if an application does not have enough users to justify a programmer's time, it will not be written.

Program synthesis is a potential solution to this problem: the user describes what they want the software to do and the program is synthesized from the description. However, in the case of users who are not programmers, it is likely that the user's description will be highly incomplete. While synthesis in the presence of limited underspecification is possible, for instance by showing the user multiple programs or predicting the most likely one, these techniques become impractical as incompleteness grows.

A key insight is that non-programmers often do not need to synthesize complex algorithms: they are typically interested in web and mobile applications that allow them to enter, search, and visualize data. In our work we are pursuing an approach to automatically assemble such applications by configuring and combining coarse-grained, highly parametrizable, library components. This problem domain suggests an approach for dealing with missing information:


we can synthesize the partial structure of an application from the information given, leaving *holes* where there is insufficient information to select a component or to specify component parameter(s). Importantly, in this domain, missing information is not arbitrary: it is likely about “details” that do not affect the large-scale program structure. A designer familiar with the domain can enable synthesis in the presence of incomplete specification by distinguishing these details—which can be deferred until later—from the core information that drives the synthesis procedure.

Our approach forms a pipeline, starting with an ontology of programming concepts that captures the intent of a user in a high-level way. We proceed by selecting components that implement the desired functionalities and whose parameters and preconditions (specification) are met by the surrounding context. For instance, a user may start with an utterance

“I want to show a list of products on a products page”

which, in a conventional three-tier application, may lead to the definition of a data type called “Products,” a database table called “Product,” an app screen called “Products Page,” and the instantiation of a code template that produces some sort of pre-prepared (library) UI element that implements the several logic layers of displaying the products.

Our approach to synthesis leverages inference rules, designed by domain experts, that describe how requests by the user should translate into computational elements. In the example above, the inference rules might specify that showing a list of products can be done by showing a table with a column for each field in the “Products” datatype. This information is enough if the system already knows about products, but if it does not, synthesis will not be complete, because the fields of “Products” are not known. We would like synthesis to produce a preliminary result—even going so far as to show the user a page with an empty table—and then ask the user for the missing “Product” field information.

The novel technical solution proposed in this paper is to allow the designer of the inference rules used in synthesis to explicitly say what information can be deferred during the initial synthesis and specified later. Such information is specified in *deferred premises*, marked with a . When forming a derivation, typically through backwards search (i.e. from a goal to the facts needed to establish it), we allow an inference rule to be triggered even if deferred premises are not derivable. This forms a derivation with *holes*: locations in the derivation tree where a branch is missing. Logically, the conclusion of the derivation is conditional on filling in the missing holes. We justify the soundness of this approach via a (provability-preserving) translation of our system of inference to first-order logic formulas. In our practical context, holes represent missing information from the ontology and can be filled by asking the user for that information. Alternatively, machine learning techniques could be used to

predict the most likely ways to fill in the missing information and present the user with a suite of choices for each hole.

A key design choice is that we do not make all premises deferred; instead, deferral is a choice made by the designer of the inference system. Typically, deferral is used for branches of the derivation that do not determine the derivation’s overall structure, yet must eventually be present for the derivation to be complete and for all component parameters to be fully determined. Premises that are fundamental to the derivation’s structure should not be deferred; in our setting, we expect the user to say something that indicates the overall shape or direction of the derivation, triggering the choice of a rule, but where some details may not be readily available. These are accounted for with deferred premises, which have to be subsequently filled in to complete the derivation. Deferred premises, therefore, express the synthesis system designer’s intent for how the system will interact with the user, identifying which questions are productive to defer and later ask the user, and which questions are not.

In this paper, we flesh out the approach sketched above to ontology-driven application assembly. The target language is OSTRICH [13, 14, 24], which provides an abstraction layer over the model-driven development model by OutSystems, a company providing a popular low-code programming platform. Our work uses a library of OSTRICH component templates and the native capabilities of the OutSystems platform to inspect and create persistent data types and UI elements.

The paper structure is described next. We start with an overview of related work in Section 2. Section 3 describes the motivating context for our work: a system that synthesizes applications out of components based on natural language specifications that are parsed into an ontology. Section 4 describes the baseline approach to synthesis that we are building on, which assumes complete specifications in the ontology. Section 5 outlines the primary contribution of this paper: an approach to specifying inference rules with deferred premises, to accommodate missing information in the ontology. Examples illustrating the expressiveness of the approach are given in Section 6, and then Section 7 gives deferred premises a semantics based on a translation to first-order logic formulas. Section 8 discusses a prototype implementation, and Section 9 evaluates our approach on a set of sample queries that contain missing information. We conclude with a discussion of future work in Section 10.

2 Related Work

The idea of semantically meaningful holes in programs has been studied formally by Omar et al. [18, 19]. This line of work expands on the idea of typed holes, found in functional languages such as Haskell or Agda, going beyond the idea of (typed) holes as an ad-hoc quality-of-life compiler artifact and providing a semantic treatment of holes such that incomplete programs are statically meaningful objects,

which is related to our idea of incomplete, yet semantically meaningful, derivations.

Incomplete programs (or *sketches* [27]) are also found in the program synthesis literature, where a program with holes is treated as a synthesis problem where the synthesizer uses a specification combined with the contextual information provided by the (partial) program fragment in order to meaningfully fill the hole. The specification can take many forms such as logical assertions [10, 27], polymorphic refinement types [22] and input-output examples [1, 15, 20]. While program synthesis fundamentally deals with under or partially specified program fragments, the point of view of synthesis is typically to produce the most suitable result (including no result) with the given inputs. If the user deems the result inappropriate, the synthesis framework expects the user to provide additional information (e.g., a refined specification, more examples, etc.) to further prune or guide the next synthesis attempt, which is independent of previous runs.

While sketching takes a program with holes as input and uses partial specifications to fill the holes, our approach takes partial specifications and uses them to synthesize a program with holes. Those holes will later be filled with other techniques, one of which could be the kind of synthesis used in sketching. Our approach allows the system designer to explicitly characterize the places where the developer can initially leave out details, which are later asked of the developer via direct or indirect means. We are not aware of any other approach to synthesis that aims to characterize what parts of a specification can be incomplete, and thereby internalize this kind of user interaction.

Moreover, the representation of incomplete or deferred premises enables our system to explicitly capture design intent related to which component parameters are accessories to the overall component assembly, in that they are needed to ultimately complete the goal but are not essential in determining the fundamental structure of the synthesis target. This aspect is both unique and essential to the applicability of our approach to our chosen target domain, where we aim to synthesize (executable) component assemblies for data-driven applications from high-level and intrinsically imprecise or ambiguous user intents (e.g. natural language).

The work by Le et al. [11], SmartSynth, has similar goals to ours, starting from natural language and producing a script that orchestrates a sequence of operations in a mobile device. It follows a rather different approach. The authors propose a discovery method using the components of a natural language utterance to come up with a set of base components and a variety of data flow relations between them. These relations are subsequently validated and ranked using the signatures of the operations and the data being exchanged. SmartSynth adds extra information to avoid ambiguity through pre-prepared questions assigned to each operation, whereas we achieve this via our suggestion engine and user questions. Unlike SmartSynth, we provide a

constructive approach to the introduction of new components and their dependencies. Our approach seems more adequate for the more declarative constructs in the low-code domain. This work is a good starting point to automatically define the effects of user interface actions in our future work.

In existing related work, the conceptual distance between the specification and the synthesis output is generally much smaller. Specifications take the form of rich types, assertions and examples and are used to synthesize functional code [22], imperative heap-manipulating code [23] or even SQL queries [28]. In this kind of setting, small increments to the specification as a result of inspecting the synthesis output are generally feasible as independent user-synthesizer interactions, not internalized in the framework itself. Even when the synthesis output is closer in spirit to that of our work (e.g. web layouts [16]), the specification is still conceptually close to the synthesized outputs (i.e., examples). In our context, specifications are both less precise and are significantly distant from the synthesized component assembly such that small changes to the specification may result in dramatic, and often undesired changes to the output. Thus, making explicit the way in which the synthesizer addresses incompleteness is key to producing an effective tool that can take natural language specifications and produce Web or mobile applications.

Our setting is similar to the work on staged compositions by Döder et al. [4, 5] that, on the one hand, uses a well-typed two-stage template language like ours to specify and instantiate components, and on the other hand combines it with the synthesis of compositions using logic, in their case using intersection types and combinatorial logic. Our work offers the possibility of reasoning with incomplete knowledge and using low-code components, as opposed to a more complex, but language agnostic, template mechanism.

Our approach is also inspired on gradual typing [25]—which can be considered types with holes—and more recently gradual verification [2, 29], in which specifications can have holes. Most closely related is work [12] gradualizing the Calculus of Inductive Constructions, the logic underlying proof assistants such as Coq and Agda. These systems also try to construct a derivation showing that a program can be typed or verified when some information is missing. Our work differs in the setting: inference rules intended to be used in forward reasoning, with entire premises that are deferred—as opposed to one part of a formula, type, or program.

3 System Architecture

In this section, we give a bird’s-eye view of the concept-based program synthesis system that forms the context for our approach. Our system, depicted in Figure 1, is a low-code development environment that provides both visual and natural language interfaces. In the middle of the diagram is an inner loop composed of a concept-based ontology

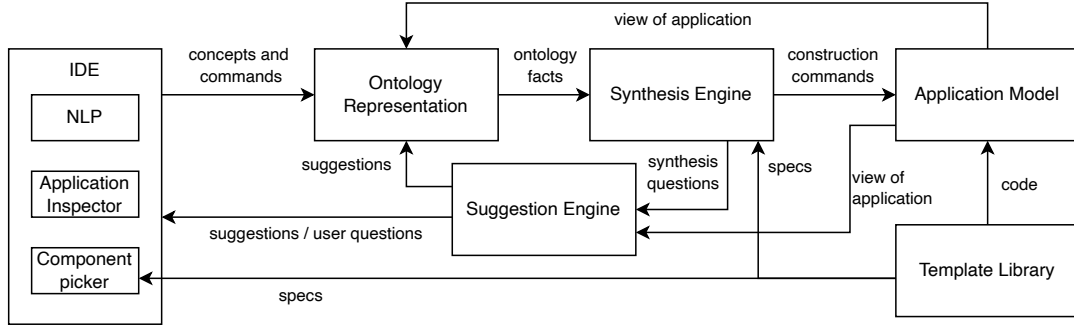


Figure 1. Architecture of the complete synthesis and suggestion system.

representing of the developer’s intent, a synthesis engine, and a suggestion engine. The context of this synthesis loop includes the IDE that gathers concepts and commands from the user while posing questions generated by synthesis, a library of component templates, and an Application Model that assembles an application based on commands from the synthesis engine as well as code from the template library. This paper focuses mainly on one component of the inner loop of this architecture, the concept-based synthesis engine.

The IDE captures the user intent based on a combination of user inputs. One important modality is natural language input combined with a simplified visual interface that shows all application components and components available for composition. Another is a set of high-level operations captured by gestures in a power user interface (e.g., dragging a widget from a toolbar or an entity from the data section to the canvas). All in all, these different user interfaces capture the user’s intent along with extra information and context.

Our approach lies in the domain of concept-based programming [8, 21] where high-level concepts are represented in an ontology [26] and then transformed into assemblies of pre-prepared components, synthesized data types, and glue code. We make crucial use of a task-oriented ontology [26] in our pipeline. The syntactic interpretation of the user’s utterances, using the Stanza library tools,¹ identifies the different sentences’ components and matches them with commands and datatypes in the ontology. This process tries to minimize the alternative meanings of each component in the interaction, following [26], based on distances between terms, and a constraint-solving approach that takes the constraints built into the task-oriented ontology as input. If more than one alternative is still available after this minimization process, an agenda-based dialogue is triggered that queries the user with additional ontology-driven questions to address the ambiguity. Once the number of alternatives reaches one, the resulting ontology instance is passed on to the next step.

The ontology codifies the direct operations on data that are typical of mobile and web applications (e.g., show, create,

search, filter), data representations of the domain (i.e., definitions for persistent data and its attributes), and computed values using a series of known transformations (i.e., built-in and library functions). Other concepts that may be supported in the foreseeable future include declarative definitions of access control conditions and the effects of user actions.

From the concepts expressed by the user, our approach produces code using the template language OSTRICH [13, 24] to represent assemblies of pre-prepared components. Synthesis results range from complete applications to parametrized screens and widgets. One of the key elements of our system is the component library containing a dynamic collection of components and their specifications. The synthesis engine matches concept facts from the ontology to specifications from the library, producing proposed component instantiations and connections with other synthesized elements. The synthesis engine interacts with a suggestion engine to provide the necessary completions of deferred premises used by the search process. The suggestion engine may use default values, built-in (expert or common sense) rules, or AI-based mechanisms to provide the missing information.

The last piece in our architecture is the Application Model, which manages the concrete application being synthesized. On every evolution step, it computes the differences between what is being defined by the synthesis engine and what is present in the current code base. It then uses the differences to create or update elements in a resource-aware fashion that avoids duplication of functionality.

4 Programming with Concepts

The pipeline introduced in this paper is triggered by commands that the user provides in the form of a natural language utterance (e.g. in a text prompt or speech recognition engine). Recall that such utterance is represented using ontology facts [26] denoting the user’s intent.

Consider the utterance “I want to show a list of products on a products page,” provided by the user in natural language. Assuming that the natural language layer eliminates

¹<https://stanfordnlp.github.io/stanza/>

all ambiguities from this sentence and reaches a single ontology instance representation. The concepts referred to in this utterance are represented by the following ontology facts:²

```
Show(show1)
Page(ProductsPage)
Where(show1, ProductsPage)
What(show1, Products)
Record(Products)
```

(1)

where *show1* identifies an event in the user interaction whose main action is *Show*(_⊔), defining the intent of displaying data, that is related to a UI location (a page) named *ProductsPage*, and a data type (record) named *Products*. We also assume that the following fact depicting the structure of a database entity named *Products* is also represented in the ontology, having flowed along the connection from the Application Model to the Ontology Representation component in Figure 1,

```
Entity(Products, {name : String,
                  description : String,
                  price : Int, stock : Int}).
```

(2)

The synthesis engine then proposes the (conditional) instantiation of a library template that creates a table in the UI, on the page named *ProductsPage*. This result is represented by a fact of the form

```
Query(ProductsQuery, show1, Products, {})
Template('List', show1, ProductsPage, ProductsQuery, ?A)
Attributes(show1, ProductsQuery, ?A)
```

(3)

where the name 'List' is a reference to a template in the library, *ProductsPage* is a reference to the target location, *ProductsQuery* is the name of a new query that is also derived in this process, and *?A* is a placeholder for a list of attributes to display. The lack of explicit information about the list of attributes to display in the user utterance triggers a feedback loop in the pipeline, in the form of an interrogation. The questions in the interrogation are processed by the suggestion engine, which may use default values, built-in, or AI-based mechanisms to provide the missing information. If the suggestion engine does not provide all the information, a question may be directed to the developer in the IDE. In the present example, a list with a (pre-checked) checkbox for each possible attribute can be shown to the developer to confirm. Once the information is provided and added to the ontology, the derivation is complete, all the arguments of the template are fully determined, and the application components can be instantiated and previewed.

The target of our synthesis engine is the domain of mobile and web applications, which are constructed via the instantiation and assembly of pre-prepared components (cf. [5, 21]).

²For the sake of clarity we modified the predicate names wrt [26].

```
C ::= Field(f)
    | Record(e)
    | Type(f, T)
    | PartOf(a, e, f)
    | Page(p)
    | Show(a)
    | Create(a)
    | Filter(a)
    | What(a, e)
    | Where(a, p)
    | Operator(f, F)

P ::= C
    | Location(p)
    | Entity(e, {f : T})
    | Query(q, a, e, f)
    | Attributes(a, e, (f = e.f, f = F(e)))
    | Template(TemplateName, a, p, d, f)
```

Figure 2. Syntax of ontology and system-level facts

A low-code application model is a collection of components with well-defined links that represent establish dependencies and containment. Hence, each produced predicate denotes one (or more) components in the target application model. The predicate's arguments help instantiate and link the components together in a low-code application model. Since each component in the template library is associated with one or more actions represented in the ontology, several alternatives may be derivable at any given time, in which case they must be ranked and presented to the developer.

4.1 Representing Concepts

As illustrated by the examples above, we capture programming concepts [8] with predicates parameterized with values from the domain as well as names that are used to connect components. Our system can express concepts for direct actions like showing data, searching rows using simple criteria, creating new records in the database, or computing simple functions over existing data. In the future, we plan to capture UI effects and access control conditions in the ontology, but for now we focus on the direct actions over data.

Recall that the programming concepts we consider in this work represent a limited range of functionalities often used in simple mobile and web applications. The basic concepts are represented in an ontology (cf. [26]), whose instances in our approach are provided by the developer. The facts of the ontology are presented in the syntax in Figure 2. Consider a set of names ranging over *p* for pages (and other UI locations), *f* for field labels, *e* for entities such as records, *a* for actions, *q* for names of queries, and *d* for data sources that can be queries or entities.

Names are used in predicates that instantiate concepts. The concepts present in the ontology and specified directly

by the user are ranged over by C . Derived concepts, ranged over by P , are designed to represent the elements that will be created or updated in the system. Our representation assumes a system that is always running and where new elements can be created and existing elements can be updated.

The ontology facts $\text{Field}(f)$, $\text{Record}(e)$, $\text{Type}(f, T)$, and $\text{PartOf}(a, e, f)$ capture the different aspects of the structure of record data types e and their attributes f related to the user interaction named a . These structures are used to define database entities. The predicate $\text{Operator}(f, F)$ describes a field f that is computed using the formula F . The name of the operator is used to identify the operation to be performed on the fields of an entity. For instance, the operator name “price with VAT” can be used as a label for a computed field; we assume that background knowledge or machine learning techniques can be used when constructing the ontology facts to identify the corresponding formula F that needs to be applied to a product record (a row in a database) to provide the intended result. The language of types (T) for fields includes base types in the style of a relational model: String, Int, and Date.

Fact $\text{Page}(p)$ names locations (application pages) in the user interface. For the sake of simplicity, we used only one kind of possible UI location. The facts $\text{Show}(a)$, $\text{Create}(a)$, and $\text{Filter}(a)$ define what direct action should be described by and implemented for identifier a . These actions are complemented by facts $\text{Where}(a, p)$ and $\text{What}(a, e)$ that link to locations and entities in the user interface.

From the ontology facts (C) our system will derive the system-level concepts that represent an abstract composition of application components. In Figure 1 we can see that this assembly is transported from the synthesis engine to the actual Application Model. Concept $\text{Location}(p)$ declares a valid target location to instantiate new elements. Notice that the ontology concept $\text{Page}(p)$ above refers to one concrete UI element where the effect can take place. Any number of elements can be potentially added (e.g. through a point-and-click interface). The concept $\text{Location}(p)$ is abstract and generalizes across Pages and other types of UI elements. Concept $\text{Entity}(e, \{f : \bar{T}\})$ describes a database table that is either already existing or is being defined by the present action. The concept $\text{Query}(q, a, \bar{e}, \bar{f})$ declares a query to be used in action a . The query uses entities \bar{e} and filters data by fields \bar{f} . In this work, we consider a single entity in queries, even though the concept is more general. To build on predicates describing queries and their connection to the database, we define predicate $\text{Attributes}(A, q, \bar{f} = e.f, \bar{f} = F(\bar{e}))$ that selects a subset of fields $e.f$ from the entities in a query q and specifies an additional set of computed fields $\bar{f} = F(\bar{e})$, given a set of operators \bar{F} . Finally, a fact of the form $\text{Template}(N, a, p, d, A)$ represents the instantiation of a template identified in the library by N and relating to action a , placed at p , using data source d (i.e., a query or an entity) and attributes A .

Notice that information about the application being built is also expressed using these system-level concepts. For instance, if the database already contains an entity called *Products*, the fact in equation 2 above is present in the context.

4.2 Derivation System Based on Concepts

In order to synthesize application code, we have developed a derivation system that takes ontology facts specified by the user and infers system-level concepts that can be interpreted computationally to update or create elements in a running system. The lower-level rules in the derivation system reason about data types and UI concepts. Building on these, a set of higher-level, library-specific rules reason about how to carry out actions using components in the library. A component is instantiated through a template that specifies what action it implements, e.g. show, as well as various parameters that provide the context of the action. When a rule’s conclusion instantiates a component, the rule’s premises show the preconditions for using that component.

Consider the following example as an illustration of our system. We start with a set of known ontology concepts:

```

Show(show1)
Page(ProductsPage)
Where(show1, ProductsPage)
What(show1, Products)
Record(Products)
Field(name)
Field(price)
Field(stock)
PartOf(show1, Products, name)
PartOf(show1, Products, price)
PartOf(show1, Products, stock)
Type(name, String)
Type(price, Int)
Type(stock, Int)

```

(4)

This list of facts denotes the existence of the intent, of building an application component that shows some data, identified by name *show1*. Fresh names, like *show1* above, identify the interaction that took place to issue their related facts. In the example, we define the (data) concept of *Product* and its attributes *name*, *price*, and *stock*. We use $\text{Record}(_)$ to denote a record-like data abstraction and $\text{Field}(_)$ to denote its properties or fields. The relation between the two is denoted by facts with the form $\text{PartOf}(_, _, _)$. The expressed location for this event is an application page called *ProductsPage*. In this case we define the action *show1*, as we want to show data on a page called *ProductsPage*.

We now show the rules that progressively transform the facts above into programming abstractions that realize the user’s intent. First, we need to specify $\text{Location}(_)$, an abstraction for a part of the UI where an application component can be located. The following rule in our system specifies that a $\text{Page}(_)$ is a kind of $\text{Location}(_)$:

$$\frac{\text{Page}(p)}{\text{Location}(p)} \quad (5)$$

Other rules could specify other kinds of locations, such as screens, sidebars, footers, and headers.

Next, we give a set of rules that “typecheck” abstract information in the ontology and capture it in a new, intermediate predicate that is used in later stages of synthesis. For example, if the predicate $\text{Where}(a, p)$ represents the user’s intent to show action a at p , we need to validate that p is actually a location in the UI (as opposed to being a field, or something irrelevant). Checking that some predicate $\text{Location}(p)$ exists is sufficient to do this. The validated location information about action a is then represented in a new predicate $\text{WhatLocation}(a, p)$. All of this reasoning is captured by the first rule below; the other rules work similarly for records and fields associated with an action:

$$\frac{\text{Where}(a, p) \quad \text{Location}(p)}{\text{WhatLocation}(a, p)} \quad (6)$$

$$\frac{\text{What}(a, e) \quad \text{Record}(e)}{\text{WhatRecord}(a, e)} \quad (7)$$

$$\frac{\text{What}(a, f) \quad \text{Field}(f)}{\text{WhatField}(a, f)} \quad (8)$$

Next, we need rules that inductively derive system-level data representations (i.e. database tables) from ontology facts. While the ontology represents information about each field with a separate predicate, we would like to group these into a single predicate representing a database entity and all of its fields. For this we use an Entity predicate that captures a set of fields. The predicate is defined inductively, with a base case to create the Entity and an inductive case that adds a field to an existing entity. The rules are applied until nothing more can be added to the set:

$$\frac{\text{Record}(e)}{\text{Entity}(e, \{\})} \quad (9)$$

$$\frac{\text{Entity}(e, \{\overline{f : T}\}) \quad \text{Field}(f') \quad \text{PartOf}(a, e, f') \quad \text{?Type}(f', T')}{\text{Entity}(e, \{\overline{f : T}, f' : T'\})} \quad (10)$$

The entity concept is defined from the ontology concepts of $\text{Record}(e)$, $\text{Field}(f)$, and $\text{PartOf}(a, e, f)$, which ultimately come from the user’s utterances. In this case, a identifies an action where the field should appear, e is the name for a new datatype and \overline{f}, f' are the labels of the fields or attributes of that datatype. To create a database table, we need types for the columns; this data is extracted from the premise $\text{?Type}(f', T')$, stating that field f' has type T' . The $?$ indicates that this premise is deferred, meaning it represents a part of the proof that does not need to be derived immediately for the synthesis engine to trigger this particular rule.

Its derivation is deferred to a later stage and the result becomes conditional on it. Facts to support deferred premises can be later instantiated by the suggestion system or taken back to the developer via an appropriate user interface. We delve into the details of deferred premises in [Section 5](#).

To represent database access, we derive a description of a query, represented with the judgment $\text{Query}(q, a, e, \overline{f})$, with q freshly generated. The judgment ties together the database entity being accessed, e , with the action a , and also selects the subset of the fields \overline{f} that are used in this action as filters.

$$\frac{\text{Show}(a) \quad \text{WhatRecord}(a, e) \quad \text{Entity}(e, A)}{\text{Query}(q, a, e, \{\})} \quad (11)$$

$$\frac{\text{Filter}(a) \quad \text{WhatRecord}(a, e) \quad \text{Entity}(e, A) \quad \frac{\text{WhatField}(a, f) \quad \overline{f} \subseteq A}{\text{Query}(q, a, e, \overline{f})}}{\text{Query}(q, a, e, \overline{f})} \quad (12)$$

These rules derive queries from existing entities in our rule system. We define how to retrieve and also how to filter data from the database and show it somewhere in the UI. Notice that the transformation of the predicate generates queries with parameters to match the fields being used to filter data. Parameters in queries are common in any database abstraction from object relational mappings to low-code platforms. Future developments of this work will include extensions with more elaborate conditions and multi-entity queries.

An algorithmic interpretation for the derivation rules above is to obtain all derivable system-level concepts from an input consisting of concepts of the ontology and system-level concepts depicting the current existing application. Some of the facts will be mutually exclusive in terms of resources consumed and space occupied in the UI of the target application. The choice between alternatives is performed in a later stage of the process where template instantiation is resource-aware and does not instantiate widgets “repeatedly”.

From the ontology facts above, rules 9 and 10 derive the following database entity facts:

$$\begin{aligned} &\text{Entity}(\text{Products}, \{\}) \\ &\text{Entity}(\text{Products}, \{\text{name} : \text{String}\}) \\ &\text{Entity}(\text{Products}, \{\text{name} : \text{String}, \text{price} : \text{Int}\}) \\ &\text{Entity}(\text{Products}, \{\text{name} : \text{String}, \text{price} : \text{Int}, \text{stock} : \text{Int}\}) \end{aligned} \quad (13)$$

Rules 11 and 12 derive a query related to entity *Products*:

$$\text{Query}(\text{ProductsQuery}, \text{show1}, \text{Products}, \{\}) \quad (14)$$

At this point, all datatypes and queries referred by the developer’s utterances are captured by the rules and ready for the next stage where code and database schemas are generated or updated. Observing the architecture depicted in [Figure 1](#), these facts have to be linearly interpreted to create or update existing resources in the Application Model (the target application code) and corresponding database

instance. Care is needed to avoid situations where all derivable intermediate steps are used to create or update artifacts. For instance, the predicate $\text{Entity}(_, _)$ always holds for all subsets of attributes of a given entity. Our execution strategy adds elements until no more can be added; only then is the judgment that contains all elements used in future derivations. One way of interpreting this is to treat $\text{Entity}(_, _)$ as a linear resource; we do not formalize this notion here.

Notice that the initial set of facts used in the example above allows the entity and query definitions to be derived using a standard interpretation for the derivation rules. Notice also that our initial user's utterance did not refer to any datatypes for fields of concept *Products*.

5 Reasoning with Deferred Premises

The knowledge encoded in the initial ontology (i.e. the input for our synthesis mechanism) is not always sufficient to produce the desired components. To deal with this we allow for premises to be annotated as *deferred*. Any deferred premise, if undetermined in the derivation, is not sufficient to stop the search procedure but is kept as a side condition along to the overall derivation.

Given the initial information in equation 4, which satisfies the deferred premise $\text{?Type}(f, T)$, rule 10 derives the facts in equations 13 and 14. However, if we consider data in equation 4 without facts $\text{Type}(\text{name}, \text{String})$, $\text{Type}(\text{price}, \text{Int})$, and $\text{Type}(\text{stock}, \text{Int})$, the search for the facts in equations 13 and 14 would fail. By stating that premise $\text{?Type}(f, T)$ in rule 10 is deferred, we allow rule 10 to be triggered without needing the information about the type of the attributes. Instead, in each step, the optional premise that was skipped is kept as a side condition and attached to the conclusions it supports. Thus, the conclusions of the derivation are similar to those of equation 13 but mentioning universally quantified variables and with side conditions attached.

$$\begin{aligned}
 &\text{Entity}(\text{Products}, \{\}) \\
 &\text{Entity}(\text{Products}, \{\text{name} : T\}) \\
 &\text{?Type}(\text{name}, T) \\
 &\text{Entity}(\text{Products}, \{\text{name} : T, \text{price} : T'\}) \\
 &\text{?Type}(\text{name}, T) \text{ ?Type}(\text{price}, T') \\
 &\text{Entity}(\text{Products}, \{\text{name} : T, \text{price} : T', \text{stock} : T''\}) \\
 &\text{?Type}(\text{name}, T) \text{ ?Type}(\text{price}, T') \text{ ?Type}(\text{stock}, T'')
 \end{aligned} \tag{15}$$

The derivation of equation 15 indicates that once the types of the attributes with the labels *name*, *price*, and *stock* are determined, the derivation is complete. Again, it is conceivable that this kind of information can be provided by an automated suggestion engine, or be brought to the attention of the developer and answered in a way that is supported by the ontology.

Stepping up to richer concepts, we define rules to determine the attributes to be used in a particular action and the code template or component we need to instantiate to achieve the developer's intent. We begin with the predicate $\text{Attributes}(_, _)$. The rule is defined as follows:

$$\frac{\text{Query}(q, a, e, \bar{g}) \quad \text{?Field}(f) \quad \text{?PartOf}(a, e, f)}{\text{Attributes}(a, q, \bar{f} = e.f)} \tag{16}$$

$$\frac{\text{Query}(q, a, e, \bar{g}) \quad \text{Attributes}(a, q, \bar{f} = e.f) \quad \text{?Operator}(f', F) \quad \text{?PartOf}(a, e, f')}{\text{Attributes}(a, q, (\bar{f} = e.f, \bar{f}' = F(e)))} \tag{17}$$

Rule 16 has one main premise (an existing query), deferred (multiple) premises that specify the set of fields used in the action under consideration, and a side condition that relates the fields to the entities used in the query. If premises $\text{?Field}(f)$ and $\text{?PartOf}(a, e, f)$ are not immediately derivable, the developer may be prompted to declare the fields that need to be displayed. We use sets of premises instead of induction to allow for easier conversion to a question at the ontology level. Rule 17 adds computed attributes to the list based on $\text{Operator}(_, _)$ predicates in the input ontology.

From the facts depicted above, we conclude by rule 16 that action *show1* will display the fields *name*, *price*, and *stock* to the user. Notice that these attributes may be a *subset* of the attributes in the designated entity, depending on the initial facts in the ontology and the facts drawn from the actual database schema:

$$\begin{aligned}
 &\text{Attributes}(\text{show1}, \text{ProductsQuery}, \\
 &\quad \{\text{name} = \text{Products.name}, \\
 &\quad \text{price} = \text{Products.price}, \\
 &\quad \text{stock} = \text{Products.stock}\})
 \end{aligned} \tag{18}$$

This rule is also triggered in situations where no information is given to determine the attributes to be shown. In the case where no $\text{Field}(_)$ or $\text{PartOf}(\text{show1}, _, _)$ are present, the conclusion would be

$$\begin{aligned}
 &\text{Attributes}(\text{show1}, \text{ProductsQuery}, \bar{f}) \\
 &\text{?Field}(f) \quad \text{?PartOf}(\text{show1}, \text{Products}, f)
 \end{aligned} \tag{19}$$

the deferred premises account now for the whole list of attributes to be used in action *show1*.

Finally, to determine the library components or code templates to be instantiated to satisfy the developer's intent, we define predicates of the form

Template('Name', action, location, datasource, attributes)

that denote the instantiation of a template identified by 'Name' with the corresponding arguments. Each template added to the library is accompanied by a specification that

describes the preconditions to the template's instantiation and the set of concepts, intents, and functionalities implemented. This directly corresponds to a derivation rule whose conclusion is the template predicate and the premises correspond to the preconditions. Here are rules for instantiating components that display lists, create a new record, and allow the user to filter a list by a given field:

$$\frac{\text{Show}(a) \quad \text{?WhatLocation}(a, p) \quad \text{?Query}(q, a, e, \{\}) \quad \text{?Attributes}(a, q, A)}{\text{Template('List', } a, p, q, A)} \quad (20)$$

$$\frac{\text{Create}(a) \quad \text{?WhatRecord}(a, e) \quad \text{?WhatLocation}(a, p)}{\text{Template('RecordShowEdit', } a, p, e, \{\})} \quad (21)$$

$$\frac{\text{Filter}(a) \quad \text{?WhatLocation}(a, p) \quad \text{?Query}(q, a, e, \bar{f}) \quad \text{?Attributes}(a, q, A)}{\text{Template('FilteredList', } a, p, q, A)} \quad (22)$$

These rules are triggered by actions $\text{Show}(_)$, $\text{Create}(_)$, and $\text{Filter}(_)$, with all other premises identified as deferred.

A typical situation is to have an already existing page in the application, a predefined database schema, and not determine the attributes to be shown. Consider now the following set of facts, denoting an existing entity and the intent to show such an entity.

$$\begin{aligned} &\text{Entity}(\text{Products}, \{\text{name} : \text{String}, \text{description} : \text{String}, \\ &\quad \text{price} : \text{Int}, \text{stock} : \text{Int}\}) \\ &\text{Record}(\text{Products}) \\ &\text{Page}(\text{ProductsPage}) \\ &\text{Show}(\text{show1}) \\ &\text{What}(\text{show1}, \text{Products}) \end{aligned} \quad (23)$$

Note that these facts do not help to determine where in the user interface the products will be presented, nor the list of attributes to show. The rules above derive the following:

$$\begin{aligned} &\text{Query}(\text{ProductsQuery}, \text{show1}, \text{Products}, \{\}) \\ &\text{Location}(\text{ProductsPage}) \\ &\text{WhatRecord}(\text{show1}, \text{Products}) \\ &\text{Attributes}(\text{show1}, \text{ProductsQuery}, (\bar{f}, \bar{c})) \\ &\text{?Field}(f) \quad \text{?PartOf}(s, e, f) \\ &\text{?Operator}(c, F) \quad \text{?PartOf}(s, e, c) \\ &\text{Template('List', show1, } p, \text{ProductsQuery, } (\bar{f}, \bar{c})) \\ &\text{?Where}(\text{show1}, p) \quad \text{?Page}(p) \end{aligned} \quad (24)$$

where we highlight two universally quantified variables (\bar{f} and p) that are used in the resulting predicates.

Having discussed the mechanics of deferred premises, let us now take stock of the development. First, let us consider

how rules with deferred premises can be treated algorithmically. An algorithmic interpretation for our derivation rules consists of using the concepts of the ontology and the facts about the current application as axioms. Deferred premises that cannot be derived are kept as side conditions. Since such premises may themselves be the conclusion of rules in our system, when the initial proof search concludes and results in a set of deferred premises, we subsequently run a search for such facts in order to trace deferred premises back to their underlying ontology concepts. We do this since these initial concepts are those that can potentially be transformed into questions that the developer can answer directly.

Choosing which premises are deferred. In our approach, any premise can be marked as deferred. On the other hand, marking every premise as deferred is probably a bad idea in any non-trivial synthesis setting: there would be a combinatorial explosion in the number of possible alternatives and the questions produced.

So, which premises should be marked deferred? It is up to the designer of the formal system to decide where deferred premises will be useful in the particular setting of that system. Factors the designer may consider include:

- It is a good practice to create a path through the rules from the user-defined concepts to the predicates that describe the synthesized components. There should be a non-deferred premise on every step of such paths. This ensures that the approach can determine what components should be synthesized based on required predicates.
- It is helpful to use non-deferred premises to reduce branching. If required predicates determine a unique path to the synthesized component, or at least reduce the number of possible paths, the synthesis system will not overwhelm the user by asking about the possible components to synthesize.

The factors above capture the intuition expressed in the introduction that deferred premises should express the details of a derivation and mandatory premises should be used to capture the “essential” aspects of each rule. For instance, when selecting a template, the essential aspect is the intent of the user (or the required action/functionality), and auxiliary aspects include the list of attributes to be shown to the user. When defining an entity, the list of attributes is essential whereas the types are not as essential to proceed and can be inferred or asked for in a later stage.

From holes to questions. When a rule is triggered with deferred premises, its derivation accumulates the deferred premises as side conditions. To complete the derivation, the system must provide answers to such conditions. As such, we need a mechanism for providing answers or generating questions from deferred premises that allows the user to provide answers.

We start by tracing back each deferred premise to the ontology facts that are leaves in the proof to ensure a good level of abstraction for non-technical users. By doing so, we only generate questions that are at the user level thus expected to be understood. For instance, if we consider the example of equation 24, the template instantiation fact is generated by rule 20, with the deferred premise $?WhatLocation(a, p)$. This fact is not from the original ontology but can be traced back to the ontology facts $?Page(p)$ and $?Where(a, p)$ by inverting rule 6 and rule 5.

From questions to suggestions. Recall that the pipeline proposed in this work includes a suggestion engine that can provide possible values for deferred premises that may arise in the conclusion of a derivation. The suggestion engine generates several possible alternatives for deferred premises which the user can then edit, confirm, or discard. We propose the following alternatives to seeking answers to questions.

Existing components In the running example referred above, of equation 24, the information regarding where to place the list and which attributes to show is missing. In this case, we can use the names of existing entities, with lists of attributes, and pages in the system to generate closed alternatives for the deferred premises. Initially, the engine suggests $Page(ProductsPage)$ and $Where(show1, ProductsPage)$ to denote the location using the existing *ProductsPage*. Next, for the attributes, the engine suggests that the list of attributes of entity *Products* is used in event *show1*. A list of facts like $PartOf(show1, Products, name)$ is produced from the original list of attributes and presented to the user. They are then able to select which of the suggestions are applicable if any.

Built-in rules In cases where there is no available information in the context from which to provide alternatives from, a system of built-in rules can be used to provide the user with plausible suggestions. Consider the example of equation 15, where information regarding the types of attributes is missing from the user utterance. The system designer may create a rule to map common attribute names to their respective types. In this case, the suggestion engine may provide the alternatives $Type(name, String)$, $Type(stock, Int)$, and $Type(price, Int)$. These alternatives can, of course, be approved by the user.

Other program synthesis methods Another method of providing alternatives is to resort to other kinds of synthesis methods. Many deferred premises can be answered automatically by following patterns. One way to obtain such patterns is to use machine learning on top of existing applications. For instance, it is common to display attributes *name* and *description* as well as all calculated attributes, but uncommon to

show attributes with internal information like foreign keys or timestamps. Hence, the list of attributes to be displayed can be suggested by learning from existing applications.

Another possibility is to use well-established synthesis methods that use examples as input data [7]. We envision that some of the questions can be translated into different synthesis problems, alternative questions asking for examples, or sketches can be sent to the user and solved.

Accepting a suggestion or alternatives. The user can select from the alternatives provided by the suggestion engine. For example, if the page on which to put the list is missing (as in the case above) the suggestion engine may suggest existing pages from the application to use. In case the user wants to create a new page to place the list, then no alternative from the suggestion engine is suitable. As such, we need to generate a question so that the user can provide the page name. The ontology fact $Page(p)$ can be posed as a question with the prompt “Please provide a name for the page”. Once the user provides an answer with the intended name (e.g. *ProductsListPage*), the suggestion engine is able to provide the alternative $Where(show1, ProductsListPage)$ to complete the derivation.

This mechanism allows for the necessary interaction with the user to complete derivations with deferred premises by either asking for confirmation of a possible alternative or prompting the user for the missing information in the form of questions.

Impact of question order. A remaining challenge is defining the order in which we ask the questions. Since our rules are expressed in first-order logic and the encoding of the problem has no notion of order, we expect the order of questions not to affect the final result. In our current implementation no special technique is used, and as such the result will be the same no matter what path the derivation takes.

However, we expect the order of questions to affect the performance of our pipeline. Consider the deferred premises $?Record(e)$ and $?PartOf(a_1, e, f)$, in the example of equation 25, that denote missing references to a database entity and attributes in the user utterance. In a system with multiple database entities, if we ask first for the $Record(e)$ premise, then we reduce the search space when searching for alternatives for $PartOf(a_1, e, f)$, since e is already known. Conversely, if we were to ask first for the $PartOf(a_1, e, f)$ premise, then we would have many more possible alternatives, which may not be that helpful for the user. Since the search space for suggestions can reduce or increase depending on the order in which the information is provided, we consider this to be a critical challenge. We leave the details of this problem to future iterations of this work.

6 Examples

We now showcase the expressiveness of our framework in three ways: an extremely underspecified user utterance, a filtered data request, and a request to add an entry to an existing entity. We end this section with a consolidated example showcasing the user interaction with the system.

Dealing with extreme underspecification. Consider the case where the user utterance is simply

“I want to see a list.”

The ontology will contain only the fact $\text{Show}(a_1)$ depicting the intent to show something in the UI

Although the utterance is lacking details about what the user wants to see or where to place the component that shows it, our synthesis engine is still able to suggest the instantiation of a template of type ‘List’. Rules 20, 11, and 12 produce the following output:

$$\begin{aligned}
 &\text{Template}('List', a_1, p, q, \{\bar{f}, \bar{c}\}) \\
 &\text{Query}(q, a_1, e, \{\}) \\
 &\quad ?\text{Record}(e) \quad ?\text{What}(a_1, e) \\
 &\quad ?\text{Page}(p) \quad ?\text{Where}(a_1, p) \quad ?\text{Field}(f) \\
 &\quad ?\text{PartOf}(a_1, e, f) \quad ?\text{Operator}(c, F)
 \end{aligned} \tag{25}$$

In the conclusion, there are a number of universally quantified variables (p , e , f , and \bar{c}) that are central to the deferred premises in the conclusion. Variable e denotes an entity the user would want to see in the list, variables \bar{f} denote fields of entity e and variables \bar{c} are calculated attributes to be displayed. The variable p denotes a location in the UI where to place the list, and variable q is an internal identifier for the query to be used.

The intermediate deferred premises in this derivation for the instantiation of the template of type ‘List’ are

$$\begin{aligned}
 &?\text{Attributes}(a_1, q, \{\bar{f}, \bar{c}\}) \\
 &?\text{WhatRecord}(a_1, e) \\
 &?\text{WhatLocation}(a_1, p)
 \end{aligned} \tag{26}$$

However, these are not predicates at the level of ontology. Thus, as mentioned Section 5, we subsequently expand the search backwards from the deferred premise until the leafs of the derivation, which are base facts of the ontology. The resulting ontology facts are

$$\begin{aligned}
 &?\text{Field}(f) \quad ?\text{PartOf}(a_1, e, f) \quad ?\text{Operator}(c, F) \\
 &?\text{Record}(e) \quad ?\text{What}(a_1, e) \\
 &?\text{Page}(p) \quad ?\text{Where}(a_1, p)
 \end{aligned} \tag{27}$$

These facts can then be appropriately used by the suggestion engine. Once the system can assign values to all the

open variables, the derivation becomes complete and the template can be instantiated.

Filtering. Consider an utterance such as

“I want to see a list of products filtered by name.”

represented by ontology facts:

$\text{Show}(a_2)$
 $\text{Filter}(a_2)$
 $\text{What}(a_2, \text{Products})$
 $\text{Record}(\text{Products})$
 $\text{What}(a_2, \text{name})$
 $\text{Field}(\text{name})$
 $\text{PartOf}(a_2, \text{Products}, \text{name})$

From these facts we can derive the instantiation of two different templates from rules 20 and 22:

$$\begin{aligned}
 &\text{Template}('List', a_2, p, q, \{\text{name} = \text{Products.name}\}) \\
 &\text{Template}('FilteredList', a_2, p, q, \{\text{name} = \text{Products.name}\}) \\
 &\text{Entity}(\text{Products}, \{\text{name} : t\}) \\
 &\text{Query}(q, a_2, \text{Products}, \{\text{name}\}) \\
 &\quad ?\text{Type}(\text{name}, t) \quad ?\text{Page}(p) \quad ?\text{Where}(a_2, p)
 \end{aligned} \tag{28}$$

In this case, variable t denotes the type of attribute *name* in the database, which is missing from the initial utterance and is not given in the context of the conversation. As such, the premise $?\text{Type}(\text{name}, t)$ is deferred in the derivation of entity *Products*. This would not happen if, for instance, equation 2 was present in the context before. Variable p is the name of the page on which to place the list. Once the user answers the questions for the missing pieces, they can choose between which of the two alternatives to instantiate.

Entity creation. Consider the utterance

“I want a page to create products.”

which results in the ontology facts:

$\text{Create}(a_3)$
 $\text{Record}(\text{Products})$
 $\text{What}(a_3, \text{Products})$

in a context where equation 2 is also present in the ontology as a result of a projection from the application model, where the entity is defined. We derive by rule 21,

$$\begin{aligned}
 &\text{Template}('RecordShowEdit', a_3, p, \text{Products}, \{\}) \\
 &\quad ?\text{Page}(p) \quad ?\text{Where}(a_3, p)
 \end{aligned} \tag{29}$$

Variable p is the name of the page in which to place the (instantiated) template. Once the user provides a value for p the derivation is complete and the template can be instantiated.

The examples above illustrate how each template may trigger distinct system behaviours that are dependent on the amount of information available from the ontology. This information can be both extracted from the user’s utterance and projected from the application model into ontology facts.

Consolidated example. Finally, we showcase how the user interaction takes place from end-to-end using the running example of past sections. Consider the application with an existing database entity named *Products* and a page *ProductsPage*, denoted by the following facts:

Entity(*Products*, {*name* : String, *description* : String,
 price : Int, *stock* : Int}) (30)
Page(*ProductsPage*)

The user interaction starts with a command in the form of a natural language utterance, which can be provided via a text prompt or provided as speech in the IDE. Consider the user utterance

“I want to see a list of products.”

which is parsed and interpreted by the natural language layer into the following ontology facts:

Record(*Products*)
Show(*show1*) (31)
What(*show1*, *Products*)

These facts, together with the existing context in equation 30, by rule 20 derive the following facts:

Query(*ProductsQuery*, *show1*, *Products*, {})
WhatRecord(*show1*, *Products*)
Attributes(*show1*, *ProductsQuery*, {*f*, *c*})
?Field(*f*) ?PartOf(*show1*, *Products*, *f*) (32)
?Operator(*c*, *F*) ?PartOf(*show1*, *Products*, *c*)
Template('List', *show1*, *p*, *ProductsQuery*, {*f*, *c*})
?Where(*show1*, *p*) ?Page(*p*)

The deferred premises in equation 32 denote the details missing from the user utterance, namely which attributes to display in the list and where to place the instantiation list. Since these premises are needed to conclude the derivation, we need to provide answers for them.

The system may start by addressing the question about the attributes to display in the list. The suggestion engine, which receives and processes the questions, does so by generating possible alternatives, as described in Section 5, and asking the user to approve or edit its suggestion. For the sake of the example, let us assume the first suggestion is to display all the attributes of the entity. This is represented by the ontology facts

Field(*name*) PartOf(*show1*, *Products*, *name*)
Field(*description*) PartOf(*show1*, *Products*, *description*)
Field(*price*) PartOf(*show1*, *Products*, *price*)
Field(*stock*) PartOf(*show1*, *Products*, *stock*) (33)

This suggestion is proposed to the user in the form of a list of attributes and their names. For instance

Suggestion: Use attributes name, description, price, and stock.

The user then has the choice to accept, reject, or edit the suggestion. Let us suppose they reject the suggestion. In this case, the suggestion engine will provide an alternative to display the name, description, and a computed attribute displaying the price with VAT. Assuming that a library function *vat* exists to perform such a computation, this is represented by the ontology facts

Field(*name*) PartOf(*show1*, *Products*, *name*)
Field(*description*) PartOf(*show1*, *Products*, *description*)
Operator(*price*, *vat*) PartOf(*show1*, *Products*, *price*) (34)

Again, the suggestion is proposed to the user in natural language. This is possible because the facts are in the domain of the same ontology used to represent the utterances. In this case, let's consider that the user accepts the suggestion.

The remaining questions that the suggestion engine or the user must fulfil are related to the location in which to place the list. Generating alternatives to these premises involves producing alternatives like using the existing page *ProductsPage*. The suggestion engine first proposes this to the user

“Suggestion: Use page *ProductsPage*.”

and they can accept or reject the suggestion. Let us assume the user rejects the suggestion and no other suggestion is provided. As such, the user is prompted to input the name for a new page

“Please provide a name for the page.”

and they provide the name *Online store*. This results in the ontology facts

Page(*Online store*) Where(*show1*, *Online store*) (35)

With the facts from equations 34 and 35 added to the context, the derivation from rule 20 may now conclude. The complete result of the derivation is

Query(*ProductsQuery*, *show1*, *Products*, {})
Location(*Online store*)
WhatRecord(*show1*, *Products*)
Attributes(*show1*, *ProductsQuery*,
 {*name*, *description*}, {*price*}) (36)
Template('List', *show1*, *Online store*, *ProductsQuery*,
 {*price*}, {*name*, *description*})

which produces a page in the application with an instance of a list template with attributes name, description, and price with VAT.

7 On the Semantics of the System

The system of rules presented in the previous sections incorporates a notion of deferred premises, which are accounted for in one of two ways during proof search. When a rule with a deferred premise is selected to be (potentially) applied during search, the system can proceed by either deriving the premise outright or by effectively assuming the premise to be true, marking it as a fact that needs to be derived at a later stage to complete the derivation.

Semantically, deferred premises act as global assumptions of a given derivation, where the synthesis procedure must ultimately be provided with appropriate witnesses to fill all the holes and produce a complete derivation. Thus, complete, valid derivations in our system are actually independent of deferred premises. We can make this notion precise via a mostly straightforward encoding of our rules as first-order logic formulas. For instance, consider a rule of the form:

$$\frac{A \quad ?B}{P} \quad (37)$$

where A is a non-deferred premise, B is a deferred premise and P is the conclusion. Our goal is to encode the rule as a single logical formula, such that we can then state that a complete derivation is valid in our system if and only if its conclusion (i.e., the goal) is implied by the encoding of all the rules. Since A , B and P are predicates, we must codify the appropriate variable scoping and quantification. Given a predicate A , let $fv(A)$ denote the free variables of A . Moreover let $\forall fv(A). \phi$ denote the formula $\forall x_1, \dots, x_n. \phi$ with $fv(A) = \{x_1, \dots, x_n\}$. We thus encode the rule above as $\forall fv(A) \cup fv(B) \cup fv(P). (A \wedge B) \Rightarrow P$, where \wedge denotes conjunction and \Rightarrow denotes implication. The encoding captures the fact that deferred premises play no special role in terms of provability of complete (i.e., hole-free) proofs by encoding both non-deferred and deferred premises in the same way, simply as antecedents that imply the conclusion of the rule.

The general pattern of the encoding is therefore to produce a formula of the form $\forall fv(A_1) \cup \dots \cup fv(A_n) \cup fv(P). (A_1 \wedge \dots \wedge A_n) \Rightarrow P$ per rule in the system, where the A_i are the premises of the rule (deferred or otherwise) and P is the conclusion. The encoding of the system of rules is simply the conjunction of the encoding of all the rules. Facts (which are necessarily ground) are encoded directly as a conjunction of facts. Therefore if S is the formula encoding of the rule system and F is the encoding of the facts, we have the following:

Lemma 7.1. *Let G be a derived fact in our system, with deferred premises $?A_1, \dots, ?A_n$. Then*

$$(S \wedge F \wedge A_1 \wedge \dots \wedge A_n) \Rightarrow G$$

is derivable in first-order logic.

The lemma above captures the fact that, from the point of view of provability, deferred premises can simply be seen

as extra assumptions to the derivation of the overall goal G . This shows that our treatment of deferred rules does not affect the overall logical soundness of our approach, but rather suggests some intensional aspects of proof search.

The reader may then wonder if we can characterize the more operational aspects of deferred premise via a known logic. While such an encoding would lead us too far astray from the main focus of this paper, we conjecture that deferral acts in a fashion akin to the lax operator from lax logic [6]. In lax logic, the proposition $\bigcirc A$ has a flavour of both possibility and necessity, denoting a kind of *local truth* modality. Propositions marked as locally true may only be used to prove other locally true propositions. In our setting, this is related to the aspect where if a derivation relies on a deferred premise, then the entire derivation must itself be seen as deferred – to be completed when the deferred premise is satisfied. This modality has been given a categorical interpretation in terms of strong monads [17] through Moggi’s computational λ -calculus. We plan to investigate this connection in future work.

8 Implementation

To illustrate the use of our synthesis technique we have implemented a proof of concept prototype that includes a library of utility functions that assist with writing inference rules in our system. The prototype is written in Haskell and makes use of the Logic backtracking, logic-programming monad [9]. The goal is to have a generic library that allows us to define the language of predicates as a Haskell algebraic data type, inference rules in monadic style (via the Logic monad), and using combinators as modifiers for deferred premises. Using the Logic Monad library functions we visit all possible conclusions in the system while accounting for all deferred subproofs.

We define types `Sequent`, `StateType` and `Derivation` below to express for the result of a derivation based on the `LogicT` and `StateT` Monad transformers.

```

1 type Sequent a = ([Int], [a], a)
2 type StateType a = (Int, Map String (Sequent a), Map Int Term)
3 type Derivation a =
4   LogicT (StateT (StateType a) Identity) (Sequent a)
```

A value of type `Sequent a` is the result of a derivation, parameterized in the actual language used to represent the domain. It is represented by a triple. The first element is a list of integers (de Bruijn indexes) that represent variables in the terms used in the derivation. The second element is a list of terms representing deferred (ontology) facts in the derivation. The third element of the triple is the conclusion of the derivation.

We combine the `Logic Monad` with a `State Monad`. The state, depicted by type `StateType` contains a counter to allow the generation of fresh variables (the first element of type `Int`) a map (the second element of the triple) to implement a simple form of memoization, and a unification table (the

```

1 data Fact =
2   Field FieldName
3   | Operator FieldName OpName
4   | Record EntName
5   | Page LocName
6   | Show ActionName
7   | Create ActionName
8   | Filter ActionName
9   | What ActionName EntName
10  | Where ActionName LocName
11  | PartOf ActionName EntName FieldName
12  | Type FieldName Type
13  | Location LocName
14  | Entity EntName (Map FieldName Type)
15  | Query QueryName ActionName EntityList FieldList
16  | Attributes ActionName QueryName FieldList
17  | Template TemplateType ActionName LocName Name FieldList
18  | WhatLocation ActionName LocName
19  | WhatRecord ActionName EntName
20  | MultiAnd IndexList ValueList Fact

```

Figure 3. Type definition for the language of facts.

third element of the triple) to track the dependencies between variables introduced in the derivation. The state is threaded through the entire backtracking, proof search procedure. Type Derivation is a combination of state and logic monads returning sequents.

We define a new datatype `Fact` in Figure 3 to represent all possible predicates in a system. In the present case, the concrete definition of `Fact` closely follows the syntax of Figure 2.

We omit the definitions of types `FieldName`, `OpName`, `LocName`, `ActionName`, `EntName`, `QueryName`, `EntityList`, `FieldList`, `ValueList`, `TemplateType`, and `IndexList`. These are intermediate types that abstract basic value types and data structures and whose names are self-explanatory. With relation to the syntax in Figure 2, we added `MultiAnd` to more conveniently capture (conjunctive) sets of premises.

This construction captures the multiple premises in their deferred form. It is not possible to enumerate a set of premises to create questions if they are deferred. In this way, we can universally quantify a set of values and create a question that iterates such set. For instance, in the example of Section 6, the deferred premises of the form `?PartOf(a_1, e, f)`

are captured by `MultiAnd($\{v\}, \bar{f}, \text{PartOf}(a_1, e, v)$)`. The user provides one or more values to variable v . This yields a set of premises of the form `PartOf(a_1, e, v_i)` for each v_i in v . The resulting set of premises is then bound to variable f , and can be referred to in the derivation.

Rule 10 from Section 4.2 is expressed in our implementation as function `ruleEntity` in Figure 4. Lines 3 to 5 of the function definition consist of defining three globally *fresh* variables (v_0 , v_1 , and v_2) that are used in the subsequent steps. The argument of the function is a list of ontology facts. All

```

1 ruleTemplate :: [Fact] -> Derivation
2 ruleTemplate facts = do
3   v0 <- fresh_variable
4   v1 <- fresh_variable
5   v2 <- fresh_variable
6   Show a0 <- ruleShow facts
7   (i1, h1, WhatRecord a1 e0) <- deferred (ruleWhatRecord facts v0)
8   (i2, h2, WhatLocation a2 l) <- deferred (ruleWhatLocation facts v1)
9   (i3, h3, Query q a3 [e1] _) <- deferred (ruleQuery facts v2)
10  (i4, h4, Attributes a4 e2 fs) <- deferred (ruleAttributes facts a0 e1)
11  unify [a0, a1, a2, a3, a4]
12  unify [e0, e1, e2]
13  return (union [i1, i2, i3, i4],
14          union [h1, h2, h3, h4],
15          Template List a0 l q fs)

```

Figure 4. Rule implementation for template instantiation.

rules in the system follow this pattern, and are parametrized by such a list of facts. Some rules, such as rules `ruleWhatRecord`, `ruleWhatLocation`, `ruleQuery`, and `ruleAttributes`, have additional parameters as a pragmatic way to constrain the search. For instance, the call to `ruleAttributes` relies on the list of ontology facts but is further constrained by taking as argument a_0 , the parameter of `Show`, and e_1 , the entity over which the `Query` is derived. Lines 6 to 10 define the premises of this rule, cf. rule 20. Line 6 specifies a mandatory premise (`ruleShow`). Lines 7 to 10 define deferred premises `ruleWhatRecord`, `ruleWhatLocation`, `ruleQuery`, and `ruleAttributes`. Each of these calls produces a triple of fresh variables, hypotheses, and a conclusion with its corresponding arguments. All variables must then be unified as prescribed by the rule (lines 11 and 12). The conclusion of the rule (i.e., the value returned by the function) is then constructed by unifying variables, hypotheses, and a constructor for the template predicate.

Our system is syntax-driven, allowing explicit calls to rules yielding terminal results to obtain all possible derivations.

9 Evaluation

To evaluate our approach we used a set of examples as input to our prototype and assess that the expected questions were generated properly. For the examples, we considered an existing application with database tables *Products*, *Customers*, and *Orders*, and pages *homepage*, *best-selling*, and *admin*.

We evaluated both the soundness and the size of the resulting alternatives in different scenarios. In particular, we experimented with different application contexts to observe how it affects the proposed alternatives. We evaluated the soundness of the approach by implementing the “human in the loop” to answer the questions. We verified that the system was indeed able to generate all the expected alternatives.

Table 1 summarizes the results of the example detailed in Section 6, synthesized in an empty context. Table 2 summarizes the results of the same examples but considering an

Table 1. Alternatives proposed for the examples of Section 6 without application context.

Utterance	Ontology facts	Questions	Result
"I want a page to create products."	Create(a_0), Record($Products$), What(a_0 , $Products$)	Page(x_1), Where(a_0 , x_1)	Entity($Products$, $\{\}$), Template('RecordShowEdit', a_0 , x_1 , $Products$, $\{\}$)
"I want to see a list."	Show(a_1)	Record(x_1), What(a_1 , x_1), Page(x_2), Where(a_1 , x_2), MultiAnd($\{x_5\}$, x_6 , PartOf(a_1 , x_5)), MultiAnd($\{x_9, x_{10}\}$, x_{11} , Operator(x_9 , x_{10}))	Template('List', a_1 , x_2 , q_1 , x_6)
"I want to see a list of products filtered by name."	Show(a_2), Filter(a_2), Record($Products$), What(a_2 , $Products$), What(a_2 , $name$), PartOf(a_2 , $Products$, $name$), Field($name$)	Type($name$, x_0), Page(x_4), Where(a_2 , x_4)	Entity($Products$, $\{name : x_0\}$), Query(q_1 , a_2 , $\{Products\}$, $\{\}$), Template('List', a_2 , x_4 , q_1 , $\{name :$ $Products.name\}$), Template('FilteredList', a_2 , x_4 , q_1 , $\{name :$ $Products.name\}$, $\{name\}$)

Table 2. Number of alternatives proposed for examples of Section 6 with application context as input.

Utterance	Ontology facts	Alternatives	Context
"I want a page to create products."	Create(a_0), Record($Products$), What(a_0 , $Products$)	4	Entity($Products$, $\{price : \text{Int}, stock : \text{Int}, name :$ String $\}$), Entity($customers$, $\{email : \text{String}\}$),
"I want to see a list."	Show(a_1)	12	Entity($orders$, $\{email : \text{String}, product :$ String $\}$), Location($homepage$),
"I want to see a list of products filtered by name."	Show(a_2), Filter(a_2), Record($Products$), What(a_2 , $Products$), What(a_2 , $name$), PartOf(a_2 , $Products$, $name$), Field($name$)	8	Location($admin$), Location($best-selling$)

ontology depicting the relevant entities. Table 3 presents the results for utterances that close in on the missing details to guide the system. Each step in this table depicts the same intent but augments the available information. In closing, Table 4 presents the results for a single utterance in different application contexts.

Discussion of results. Table 1 demonstrates that in the absence of context, the system is effective at generating questions when the user’s utterance does not have all the necessary details. While there may be many such questions, the aim is that they be relatively simple to answer, by the suggestion engine or the user. Notice, for instance, the second row, where almost no information is given, and yet the system can select one adequate template.

If context is provided, Table 2 shows that our system can generate several alternatives to complete an utterance that ambiguous. Alternatives include all combinations of variable assignments that are possible given the context and also account for new elements. For instance, the example in row 1 of Table 2, generates four alternatives:

```
Template('RecordShowEdit',  $a_0$ ,  $homepage$ ,  $Products$ ,  $\{\}$ )
Template('RecordShowEdit',  $a_0$ ,  $admin$ ,  $Products$ ,  $\{\}$ )
Template('RecordShowEdit',  $a_0$ ,  $best-selling$ ,  $Products$ ,  $\{\}$ )
Template('RecordShowEdit',  $a_0$ ,  $p$ ,  $Products$ ,  $\{\}$ )
```

the first three for all the existing pages and the last one for a new page (p) to which the user has to provide a name. These values can be assigned to the corresponding variable x_1 in row 1 of Table 1, containing the same example, to complete a derivation of the template.

But there is a tradeoff: more context often leads to more alternatives and more questions to be answered or alternatives to be confirmed. Consider row 3 of Table 1. We can avoid asking the type of attribute $name$ by having the entity definition in context, projected from the current application model. However, this approach yields a higher number of alternatives (row 3 of Table 2) that need to be discharged. A more reasonable approach would be to just place a selection of components, e.g. entity $Products$, in the context and omit others, for instance, the pages of the application. Simple heuristics and contextual information from the IDE can be used to frame the reasoning process.

Table 3 shows that context can be particularly useful in supplying missing information when the programmer gives more details. In row 1, very little detail is provided by the programmer, and 12 alternatives are generated in a given context. But saying only a little bit more to describe which fields are to be shown reduces the alternatives to only 4, and mentioning the page narrows it further to just one possibility. Note that even on row 3 our approach is aiding the user, who

Table 3. Number of alternatives proposed for each utterance with the application context as input.

Utterance	Ontology facts	Alternatives	Context
"I want to see a list."	Show(a_0)	12	
"I want to see a list of products with price and name."	Show(a_1), Record(<i>Products</i>), What(a_1 , <i>Products</i>), PartOf(a_1 , <i>Products</i> , <i>name</i>), PartOf(a_1 , <i>Products</i> , <i>price</i>), Field(<i>name</i>), Field(<i>price</i>)	4	Entity(<i>Products</i> , { <i>price</i> : Int, <i>stock</i> : Int, <i>name</i> : String}), Entity(<i>customers</i> , { <i>email</i> : String}), Entity(<i>orders</i> , { <i>email</i> : String, <i>product</i> : String}), Location(<i>homepage</i>), Location(<i>admin</i>), Location(<i>best-selling</i>)
"I want to see a list of products with price and name in page products."	Show(a_2), Record(<i>Products</i>), Page(<i>Products</i>), Where(a_2 , <i>Products</i>), What(a_2 , <i>Products</i>), PartOf(a_2 , <i>Products</i> , <i>name</i>), PartOf(a_2 , <i>Products</i> , <i>price</i>), Field(<i>name</i>), Field(<i>price</i>)	1	

Table 4. Number of alternatives proposed for the same utterance in different contexts.

Utterance	Ontology facts	Alternatives	Context
"I want to see a list."	Show(a_0)	2	Entity(<i>Products</i> , { <i>price</i> : Int, <i>stock</i> : Int, <i>name</i> : String}), Location(<i>homepage</i>)
		3	Entity(<i>Products</i> , { <i>price</i> : Int, <i>stock</i> : Int, <i>name</i> : String}), Location(<i>homepage</i>), Location(<i>best-selling</i>)
		18	Entity(<i>Products</i> , { <i>price</i> : Int, <i>stock</i> : Int, <i>name</i> : String}), Entity(<i>customers</i> , { <i>email</i> : String}), Entity(<i>orders</i> , { <i>email</i> : String, <i>product</i> : String}), Location(<i>homepage</i>), Location(<i>admin</i>), Location(<i>best-selling</i>), Location(<i>products</i>), Location(<i>profile-page</i>)

would nevertheless have to specify the types of attributes *price* and *name* if no context were provided.

Finally, Table 4 shows the orthogonal perspective. If the utterance is unclear, the number of alternatives suggested increases exponentially as more context is provided. A possible approach to this problem is to provide less information and let the suggestion engine *rank* the possible alternatives and only show the user a selected few.

Future work will focus on studying the integration of this synthesis method with other methods in the suggestion engine. For instance, if the user's intent suggests that a built-in function or query may be needed, but the query or function is not defined, the engine could trigger an example-based synthesis technique (cf. [3]) that would ask the user for examples of the computation or sample results of the query.

10 Conclusions and Future Work

Application synthesis for non-programmers is challenging, in part because users' descriptions of what they want are often incomplete. This paper presented an approach to managing that incompleteness in the context of logic-based synthesis by supporting a notion of deferred premises in inference rules. Our approach leads naturally to derivations of an

application specification that contains holes—missing logical predicates that must be supplied by a suggestion facility or by asking the user for input. While preliminary, our evaluation suggests that this approach can be used to effectively synthesize a variety of examples in the domain of information processing mobile and web apps—even when a lot of required information is missing from the initial specification.

The concepts of deferred premises and derivations with holes were devised to deal with missing information in our domain of synthesis, but they can potentially have wider applications. Deferred premises can be used to support hypothetical reasoning when some information is unknown. As mentioned, holes in proofs are already being used in proof assistants such as Agda, in the gradual calculus of inductive constructions, as well as in various proof IDEs; a more systematic study that relates these approaches could be useful.

On the formal side, the study of the relationship between holes in derivations and lax logic may be fruitful. Algorithmically, it may be interesting to study both backwards and forward proof search in the presence of deferred premises. Overall, we believe deferred premises and derivation holes are a general idea with many avenues of future work.

References

- [1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification*. Springer, 934–950. https://doi.org/10.1007/978-3-642-39799-8_67
- [2] Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual program verification. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 25–46. https://doi.org/10.1007/978-3-319-73721-8_2
- [3] Ricardo Brancas, Miguel Terra-Neves, Miguel Ventura, Vasco M. Manquinho, and Ruben Martins. 2022. CUBES: A Parallel Synthesizer for SQL Using Examples. *CoRR* abs/2203.04995 (2022). <https://doi.org/10.48550/arXiv.2203.04995> arXiv:2203.04995
- [4] Boris Döder, Moritz Martens, and Jakob Rehof. 2014. Staged Composition Synthesis. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Vol. 8410. Springer, 67–86. https://doi.org/10.1007/978-3-642-54833-8_5
- [5] Boris Döder, Moritz Martens, and Jakob Rehof. 2015. Staged Composition Synthesis. In *Software Engineering & Management (LNI)*, Vol. P-239. GI, 89–90.
- [6] Matt Fairtlough and Michael Mendler. 1997. Propositional Lax Logic. *Inf. Comp.* 137, 1 (aug 1997), 33. <https://doi.org/10.1006/inco.1997.2627>
- [7] Margarida Ferreira, Miguel Terra-Neves, Miguel Ventura, Inês Lynce, and Ruben Martins. 2021. FOREST: An Interactive Multi-tree Synthesizer for Regular Expressions. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Vol. 12651. 152–169. https://doi.org/10.1007/978-3-030-72016-2_9
- [8] Daniel Jackson. 2015. Towards a Theory of Conceptual Design for Software. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (Onward! 2015). ACM, 282–296. <https://doi.org/10.1145/2814228.2814248>
- [9] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). In *Proc. of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*. ACM, 192–203. <https://doi.org/10.1145/1086365.1086390>
- [10] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo Recursive Functions. In *Proc. of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, 407–426. <https://doi.org/10.1145/2509136.2509555>
- [11] Vu Le, Sumit Gulwani, and Zhendong Su. 2013. SmartSynth: Synthesizing Smartphone Automation Scripts from Natural Language. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '13)*. ACM, 193–206. <https://doi.org/10.1145/2462456.2464443>
- [12] Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. 2022. Gradualizing the Calculus of Inductive Constructions. *ACM Trans. Program. Lang. Syst.* 44, 2, Article 7 (apr 2022), 82 pages. <https://doi.org/10.1145/3495528>
- [13] Hugo Lourenço, Carla Ferreira, and João Costa Seco. 2021. OSTRICH - A Type-Safe Template Language for Low-Code Development. In *24th International Conference on Model Driven Engineering Languages and Systems, MODELS 2021, October 10-15, 2021*. IEEE, 216–226. <https://doi.org/10.1109/MODELS50736.2021.00030>
- [14] Hugo Lourenço, Carla Ferreira, and João Costa Seco. 2022. OSTRICH - A Rich Template Language for Low-code Development (Extended version). *Softw. Syst. Model.* (2022). To appear.
- [15] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. *Proc. ACM Program. Lang.* 4, ICFP, Article 109 (aug 2020), 29 pages. <https://doi.org/10.1145/3408991>
- [16] Dylan Lukes, John Sarracino, Cora Coleman, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. Synthesis of Web Layouts from Examples. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. ACM, 651–663. <https://doi.org/10.1145/3468264.3468533>
- [17] Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comp.* 93, 1 (jul 1991), 55–92. [https://doi.org/10.1016/0890-5401\(91\)90052-4](https://doi.org/10.1016/0890-5401(91)90052-4)
- [18] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. 3, Proc. of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (2019). <https://doi.org/10.1145/3290327>
- [19] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *Proc. of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. ACM, 86–99. <https://doi.org/10.1145/3009837.3009900>
- [20] Peter-Michael Osera and Steve Zdancewicz. 2015. Type-and-Example-Directed Program Synthesis. In *Proc. of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, 619–630. <https://doi.org/10.1145/2737924.2738007>
- [21] Santiago Perez De Rosso, Daniel Jackson, Maryam Archie, Czarina Lao, and Barry A. McNamara III. 2019. Declarative Assembly of Web Applications from Predefined Concepts. In *Proc. of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2019)*. ACM, 79–93. <https://doi.org/10.1145/3359591.3359728>
- [22] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proc. of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, 522–538. <https://doi.org/10.1145/2908080.2908093>
- [23] Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 72 (jan 2019), 30 pages. <https://doi.org/10.1145/3290385>
- [24] João Costa Seco, Hugo Lourenço, Joana Parreira, and Carla Ferreira. 2022. Nested OSTRICH: Hatching Compositions of Low-Code Templates. In *Proc. of the 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22)*. ACM, 210–220. <https://doi.org/10.1145/3550355.3552442>
- [25] Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *Proc. of the 21st European conference on Object-Oriented Programming*. Springer-Verlag, 2–27. https://doi.org/10.1007/978-3-540-73589-2_2
- [26] João Quirino Silva, Dora Melo, Irene Pimenta Rodrigues, João Costa Seco, Carla Ferreira, and Joana Parreira. 2021. An Ontology based Task Oriented Dialogue. In *Proc. of the 13th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, IC3K 2021, Volume 2: KEOD*. SCITEPRESS, 96–107. <https://doi.org/10.5220/0010711900003064>
- [27] Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Proc. of the 7th Asian Symposium on Programming Languages and Systems (APLAS '09)*. Springer-Verlag, 4–13. https://doi.org/10.1007/978-3-642-10672-9_3
- [28] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *Proc. of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, 452–466. <https://doi.org/10.1145/3062341.3062365>
- [29] Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual Verification of Recursive Heap Data Structures. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 228 (nov 2020), 28 pages. <https://doi.org/10.1145/3428296>