The Choose-Your-Own-Adventure Calculus (Pearl/Brave New Idea)

Tomas Petricek
Charles University, Prague, Czechia

Jan Liam Verter
Charles University, Prague, Czechia

Mikoláš Fromm
Charles University, Prague, Czechia

Abstract

Some of the most remarkable results in mathematics reveal connections between different branches of the discipline. The aim of this paper is to point out a modest, but still remarkable, similarity between a range of different interactive programming systems.

use a simple formal mathematical model that we call *the choose-your-own-adventure calculus* to todo

2012 ACM Subject Classification Software and its engineering → General programming languages

Keywords and phrases Interactive programming systems, Type providers, Proof assistants

1 Introduction

Multiple interactive programming systems, ranging from code editors for object-oriented programming languages to data exploration systems and interactive proof assistants, exhibit a remarkably similar pattern of interaction. They offer the user, who can be a programmer, a data scientist or a proof writer, a range of choices that the user can select from in order to complete their program, script or proof. The user can initiate the interaction iteratively, using it to create and refine a larger part of their program.

There are subtle differences between different implementations of the general pattern. In some systems, the resulting source code will contain a trace of the choices made by the user. For example, when choosing an item from a list of class members, the code will contain the member name. In some systems, the interaction results in a block of code that can be included in the source file, but does not include a trace of the interaction. For example, invoking a proof search or case split in Idris [4] constructs a well-typed program, but leaves no trace of the command used to construct it. The nature of the generated options also varies. The list of choices may include all possible options that are valid at a given location, or it may list only a subset of the valid options. In some cases, it may also include incorrect options as, for example, in auto-completion for dynamic languages [7].

The aim of this paper is paper is to formally capture the recurring interaction pattern:

1. We motivate the formalism by reviewing four different systems that implement a variation on the interaction pattern. These include type providers for data access in F# [25], type providers for data exploration in The Gamma [18, 16], AI assistants for semi-automated data wrangling [21] and tooling for interactive proof assistants [2, 4, 26] (Section 2).
2. We introduce the *choose-your-own-adventure calculus*, which is a small formal structure that models an interactive system where a user constructs a program by repeatedly choosing from a list of options offered by the system (Section 3).
3. The calculus allows us to make the aforementioned subtle differences precise. We define the notions of *correctness* and *completeness* for the choose-your-own-adventure calculus. To distinguish the different ways of embedding the interactions in the edited programs, we also formally define *internal* and *external* mode of system integration.
4. We show that various programmer assistance tools, such as search and AI-based recommendations can be built on top of the primitives offered by the calculus, showing how the choose-your-own-adventure calculus supports of transfer of ideas across different kinds of interactive programming systems.

The main contribution of this paper is conceptual rather than technical. We capture a pattern that is perhaps not surprising in retrospect, but that is easy to overlook until it is given a name. We use formal programming language theory methods to precisely describe interesting aspects of the pattern. Moreover, our work also confirms that programming language theory methods can be extremely effective for studying not just *programming languages*, but also interactive *programming systems* [10].

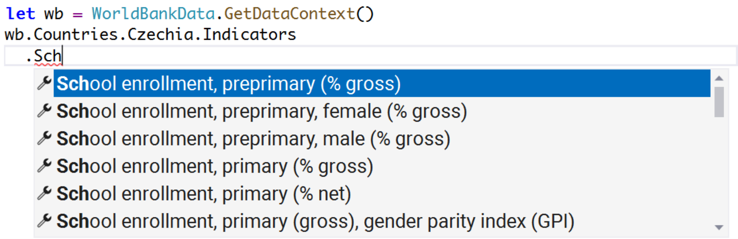

Figure 1 F# code editor showing completions offered by the World Bank type provider.

2 Motivation

Computer scientists studying programming have long focused on programming languages as syntactic entities, sometimes neglecting the interactive environments in which they are inevitably embedded [8]. Notably, in many of the motivating examples that we draw from in this section, the interactive aspect of the system is only described in supplementary materials [4, 25, 2]. Only recently, programming language theory started to be used to study interactive environments [1, 12]. Our work contributes to this research direction.

The following sections review four different instances of the choose-your-own-adventure interaction pattern. In all of those, an interactive editor offers the user some kind of a completion list during working with the system.

Type providers. F# type providers [25] are a mechanism for integrating external data sources into the F# type system. A type provider is a compiler extension, loaded and executed at compile-time and at edit-time. It can run arbitrary code to read the structure of external data and use it to generate a suitable statically-typed representation of the data, typically as objects with members. Type providers can, for example, infer the type from a sample JSON [20] or read a database schema.

The example in Figure 1 shows a simple type provider for accessing information from the World Development Indicators database. The provided wb object allows the programmer to access any indicator of any country in the database by choosing an appropriate [Country] and an [Indicator] in a chain of members wb.Countries.[Country].Indicator.[Indicator]. The result is a time series with values for the given indicator and a country. More generally, the example can be seen as a special case of a type provider for slicing n-dimensional data cube [18] – we choose a fixed value for two of the three dimensions (country, indicator, time).

When using the type provider, the user types the first line of code and triggers auto-completion by typing wb followed by the dot. The rest of the code is constructed by choosing an option from a list and typing another dot.[1]

[1] This interaction pattern has been lightheartedly called *dot-driven development* by Phil Trelford [23].

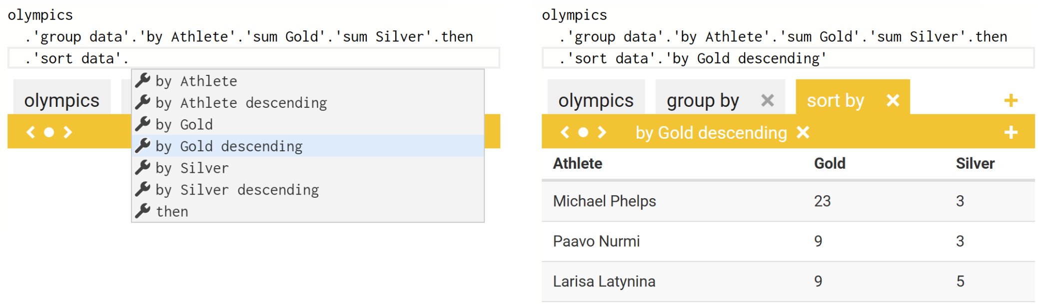

Figure 2 Constructing a query in The Gamma. We count the number of gold and silver medals for each athlete and sort the data by the number of gold medals.

Data exploration. The Gamma [18] is a programmatic data exploration environment for non-programmers. In The Gamma, type providers are the primary programming mechanism. They are used not just for data access, but also for constructing queries.

The type provider shown in Figure 2 lets the user construct an SQL-like query by repeatedly choosing operations and their parameters [16]. It keeps track of the schema and uses it to generate all possible valid parameters. When sortin data, it generates an object with two members for each columns – one for ascending and one for descending sort. Similarly, the grouping operation first offers all columns as possible grouping keys and then lets the user choose from a range of pre-defined aggregations (sum, count, average, concatenate). The system also evaluates the query on the fly, providing a live preview during editing [17].

The interaction pattern is the same as before. After the usertriggers auto-completion, they repeatedly select an operation and its parameters to construct a query. One notable difference is that the structure of the generated types is potentially infinite (the user can keep adding further operations) and so the types are generated lazily.

AI assistants. The third instance of the choose-your-own-adventure interaction pattern comes from the work on semi-automatic data wrangling tools known as AI assistants [21]. An AI assistant guides the analyst through a data wrangling problem such as reconciling mismatched datasets, filling missing values or inferring data format and types. An AI assistant solves the problem automatically and suggests an initial data transformation, but it also generates a number of constraints that the user can choose from to refine the initial solution. If the initial solution is not correct, the user chooses a constraint and the AI assistant runs again, suggesting a new data transformation that respects the constraint.

Figure 3 shows an example. It uses the datadiff [24] AI assistant, running in a Wrattler notebook [19], to merge broadband quality data published by Ofcom for two subsequent years. The format of the CSV files for the two years differs. Columns were added, removed, renamed and their order has changed. In the example, we selected 6 columns from the year 2015 and want to find matching data from 2014.

When the AI assistants runs automatically, it correctly maps the numerical columns, but it incorrectly maps the Urban.rural (2014) column to Nation (2015). This happens because both columns are categorical and have three values with similar distribution. A data analyst can easily spot the mistake. They click the "+" button to add a constraint and choose Don't match Urban.rural and Nation to specify that the two columns should not be matched. Datadiff then runs again and finds the correct matching.

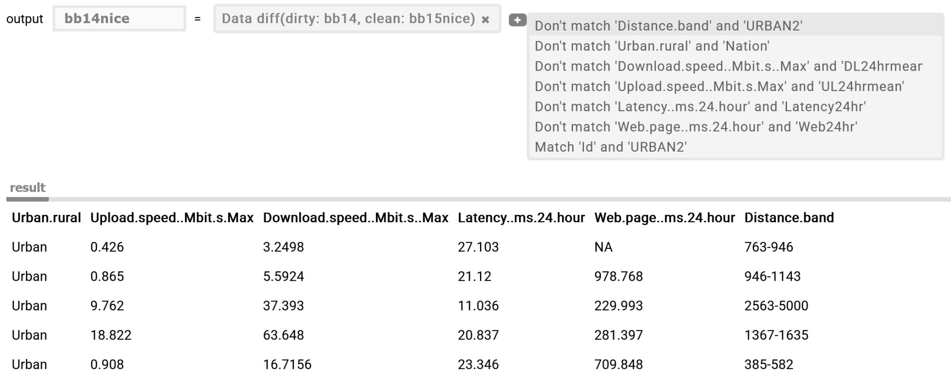

Figure 3 Using the datadiff AI assistant to reconcile the structure of the two datasets. The user is offered a list of constraints to prevent or force matching between specific columns.

The interaction patter is the same as in the previous two cases. The analyst constructs the correct data transformation by repeatedly choosing from a list of options, until they obtain the desired result. However, the way the interaction pattern is implemented differs. First, in the case of type providers, we are gradually constructing a program by adding operations to a method chain. Now, the AI assistant synthesizes a data transformation (program) and we are gradually adding constraints to control the synthesis. Second, in the case of type providers, the completion list offered all possible members of the object. Now, the list offers constraints recommended by the AI assistant which may not be complete.

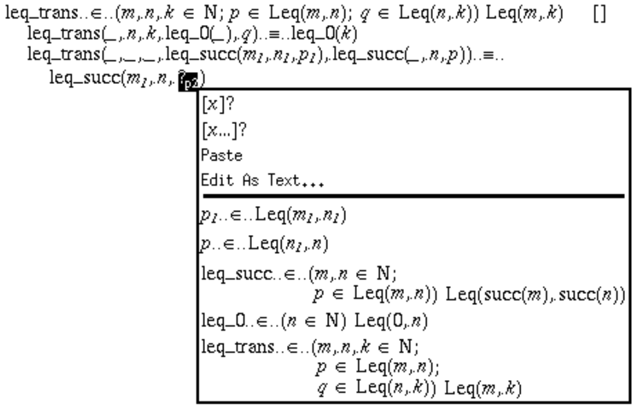

Figure 4 Constructing a proof of the transitivity of the $\leq$ relation in the ALF editor. The user is offered a range of variables and constructors in scope at the current location. [2]

Interactive theorem provers. A fourth example of the choose-your-own-adventure interactive pattern can be found in interactive theorem provers. When writing programs in systems like Idris [5], the user typically works by stating the desired conclusion and filling the implementation with a hole. The system provides a range of interactive editing capabilities to fill the holes [13]. It can, for example, generate a case split or search for a proof [4].

Systems like Idris provide key bindings to invoke the completions, but the functionality could also be offered through a user interface. An example that illustrates this is the interactive editor for the ALF theorem prover [11], which is based on the refinement of an incomplete proof object [2]. This is illustrated in Figure 4. The user is proving the transitivity of the $\leq$ relation for Peano arithmetic natural numbers. They pattern match on the proof argument p and complete the first branch. For the second branch, they need to fill a hole $?_{p2}$ (called a wildcard in ALF). They trigger a completion and a pop-up menu shows the available variables and constructors, including leq_trans that can be used to complete the proof. After choosing leq_trans, two new holes are generated for its arguments. Those can be, again, filled interactively, by choosing p_1 and p from the completion.

The interaction pattern is again the same. The user repeatedly triggers a completion and uses it to refine and complete their proof by filling holes. There are subtle differences too. Unlike with AI assistants, each completion directly refines the proof that the user is editing. Unlike with type providers, a completion may generate multiple new holes, rather than just adding to a chain of operations.

The ALF editor is a historical example, but a similar user interface could be built for systems like Idris or Coq. The two would work differently. As in ALF, Idris source code represents the proof itself and a completion would replace a hole with a suggested term. In Coq, the proof is a series of tactic invocations and so selected completions would be added to this list and would form a trace of the interaction with the user.

3 Formal model

A system that implements the choose-your-own-adventure interaction pattern repeatedly offers the user a range of options to choose from. Each of the options is designated by an identifier. The system also maintains a state during the process which determines subsequent options. The state may not be visible to the user, but the user can always explicitly request the program constructed so far.

We can think of the interaction with the system as navigating through a tree structure, starting from a root and choosing one of the possible branches in each step.[2] In the following definition, the key choices operation can thus be seen as returning branches of a given node.

▶ **Definition 1** (Choose-your-own-adventure system). *Given expressions $e \in \mathbb{E}$ and states $\sigma \in \Sigma$, a choose-your-own-adventure system is a pair of operations* choices, choose *such that:*

- $\mathsf{choices}(\sigma) = \{\iota_1 \mapsto \sigma_1, \ldots, \iota_n \mapsto \sigma_n\}$ *is an operation that takes a state and generates options designated by an identifier ι_i and represented by a state σ_i,*
- $\mathsf{choose}(\sigma) = e$ *is an operation that returns generated program for a given state.*

The definition is not a programming language calculus in the usual sense in that it does not define a concrete syntax with reduction rules. It is an abstract algebraic structure that captures the structure of a system that supports the choose-your-own-adventure interaction pattern. The definition is close to that of an AI assistant [21], which is written using a language specific for the data wrangling domain (such as cleaning scripts or input and output data) but is structurally similar. It is also worth noting that the definition may describe not just trees, but also graphs with cycles – a system can return to an already visited state. This is not practically useful, but it does not pose a theoretical problem.

External mode of embedding. One of the subtle questions about the choose-your-own-adventure pattern raised in the introduction concerns the different ways in which a trace of the interaction is embedded in the interactively constructed program. In the *external mode*, the interaction results in code that becomes a part of the edited program, but it is not possible to reconstruct the steps used to generate the code.

The choose-your-own-adventure interaction pattern is typically used to complete a partial program. To model this, we assume that the host language has a notion of a hole, written as ? and that a user can select a part of program to invoke the completion on. We write $E[e]$ for a completion context, akin to evaluation contexts in operational semantics.

We assume that, for a program containing a hole in a completion context $E[?]$, we can construct an initial choose-your-own-adventure state using an operation $\mathsf{init}(E[?]) = \sigma_0$.

▶ **Definition 2.** *An expression $E[?]$ is completed as $E[e]$ via external embedding of an interaction with a choose-your-own-adventure system consisting of* choices *and* choose *if:*

1. $\mathsf{init}(E[?]) = \sigma_0$ *obtains the initial state of a choose-your-own-interaction system,*
2. σ_n *is a system state such that* $\forall i \in 1 \ldots n. (\iota_i \mapsto \sigma_i) \in \mathsf{choices}(\sigma_{n-1})$*, i.e., the user makes a series of choices resulting in a final state of the system σ_n,*
3. $E[e]$ *where* $e = \mathsf{choose}(\sigma_n)$*, i.e., the final program is constructed by replacing the hole in the completion context with the expression e generated from σ_n.*

[2] Serving as another evidence for the surprising effectiveness of the concept of a tree [14].

As we will see when we revisit the earlier examples formally, the external mode of embedding is used, for example, in the case of interactive theorem provers like ALF or Idris. In those systems, the user triggers the completion on a proof (program) containing a hole. They then fill the hole and, possibly iteratively, further holes in the generated proof. The final expression is embedded in the source code, but it does not indicate what options, identified by $\iota_1, \ldots, \iota_n$, were selected in the process.

Internal mode of embedding. In the *internal mode*, a trace of the interaction with a choose-your-own-adventure system is embedded directly in the constructed program. This is the case with type providers, where a user chooses a sequence of object members to be accessed. The same would be the case in a completion system for Coq that would offer tactics to apply, becuase the resulting proof would contain a record of the selected tactics.

To talk about the internal mode formally, we again need the init operation, but also an operation decode that extracts identifiers of invoked completions from an expression. An internal embedding is the same as external embedding with an additional constraint:

▶ **Definition 3.** *An expression $E[?]$ is completed as $E[e]$ via internal embedding of an interaction with a choose-your-own-adventure system consisting of* $\mathsf{choices}$ *and* choose *if:*

1. *$E[?]$ is completed as $E[e]$ via external embedding using* init*,* $\mathsf{choices}$ *and* choose *through a series of choices designated by identifiers $\iota_1, \ldots, \iota_n$,*
2. *it also holds that $\mathsf{decode}(e) = (\iota_1, \ldots, \iota_n)$.*

If a choose-your-own-adventure system is integrated in a programming language through internal embedding, we can reconstruct the choices through which the user constructed an expression e in a completion context $E[e]$, assuming they used the interactive system rather than entering the code directly. This also means that we can reconstruct the final state σ_n of the system by starting from $\mathsf{init}(E[?])$ and following the choices specified by $\iota_1, \ldots, \iota_n$.

4 Examples

We now revisit the four examples from Section 2 and show how they fit the above formal model. All four examples rely on some domain-specific logic. We describe what information the logic provides, but do not model it formally. This has been done elsewhere, in works describing the individual systems.

To show how the model lets us distinguish subtle details of interactive programming systems, we start with a model of data exploration system that is inspired by The Gamma, but differs in one notable way. We then discuss type providers more generally and show how to correctly model The Gamma. We then revisit the remaining two examples.

4.1 Data exploration

In The Gamma, the choose-your-own-adventure interaction pattern is used to construct a query that transforms the given input data. The query is a sequence of operations with parameters, $op(p_1, \ldots, p_n)$, loosely modelled after relational algebra [6].

In The Gamma, the query is hidden from the data analyst. Behind the scenes, the system generates objects with members and the identifiers designating individual options are the names of those members. The operation is encapsulated in the code of the accessor of the member. In the simplified model in this section we ignore this fact. The model presented here directly generates code that calls the underlying operations. For example, assume that the user makes the following choices:

«group data» . «by Athlete» . «sum Gold» . «count all» . «then»

In The Gamma, the individual identifiers become object members and they are included as a member chain in the generated code. In the following simplified model, the completion instead fills the hole with an expression representing the operation:

`group("Athlete", sum("Gold"), count())`

The two approaches have different human-computer interaction trade-offs. In terms of cognitive dimensions [9, 3], the latter has a greater closeness of mapping, while the former is less cognitively demanding to read for a non-programmer. As discussed in Section 5, the two implementations of the choose-your-own-adventure interaction pattern also differ in terms of their formal properties.

Formal model. The options generated by The Gamma let the user select both the next operation and the parameters of the previously selected operation. The available operations and parameters are generated based on a schema S that is transformed by the operations. The state of the system σ contains the current schema S and the operations applied so far. In the following, we write $op(\boldsymbol{p})$ for an operation with a vector of parameters:

$$\sigma = S, [op_1(\boldsymbol{p_1}), \ldots, op_n(\boldsymbol{p_n})]$$

The behaviour of the choices operation depends on whether the last operation in the sequence expects further parameters or whether it is fully-specified. In the first case, the recommendation engine generates possible additional parameter values $p_{\prime}, p_{\prime\prime}, \ldots$ based on the schema S, the operation op_n and the already known parameters $\boldsymbol{p_n}$. The choices operation then generates options that add the additional parameter. We generated the identifiers $\iota_{\prime}, \iota_{\prime\prime}, \ldots$ based on the state and the parameter value, such as «by Gold descending». Note that adding a parameter may also result in a new schema $S', S'', \ldots$ (which the recommendation engine computes based on the previous schema and the new parameter):

$$\begin{aligned}&\mathsf{choices}(S, [op_1(\boldsymbol{p_1}), \ldots, op_n(\boldsymbol{p_n})]) = \\ &\quad \{\ \iota' \mapsto (S', [op_1(\boldsymbol{p_1}), \ldots, op_n(\boldsymbol{p_n}, p')]), \\ &\quad\ \ \iota'' \mapsto (S'', [op_1(\boldsymbol{p_1}), \ldots, op_n(\boldsymbol{p_n}, p'')]),\ \ldots\ \}\end{aligned}$$

If the last operation takes no further parameters, the system produces a choice of possible next operations $op', op'', \ldots$. Again, we are also given new schemas $S', S'', \ldots$ and we generate identifiers $\iota', \iota'', \ldots$ based on the operation name. The choices operation then returns options that add the additional operation:

$$\begin{aligned}&\mathsf{choices}(S, (op_1(\boldsymbol{p_1}), \ldots, op_n(\boldsymbol{p_n}))) = \\ &\quad \{\ \iota' \mapsto (S', [op_1(\boldsymbol{p_1}), \ldots, op_n(\boldsymbol{p_n}), op'()]), \\ &\quad\ \ \iota'' \mapsto (S'', [op_1(\boldsymbol{p_1}), \ldots, op_n(\boldsymbol{p_n}), op''()]),\ \ldots\ \}\end{aligned}$$

Finally, the choose operation takes the state σ and generates an expression that represents the data transformation. This is only possible if all parameters are fully-specified. For simplicity, assume that k is the index of the last fully-specified operation (either n or $n-1$). If the host language lets us compose functions using $f \circ g$, we can write:

$$\mathsf{choose}(S, (op_1(\boldsymbol{p_1}), \ldots, op_n(\boldsymbol{p_n}))) = op_1(\boldsymbol{p_1}) \circ \ldots \circ op_k(\boldsymbol{p_k})$$

The recommendation engine behind The Gamma provides a domain-specific logic for generating possible operations and their parameters based on the current schema of the data. As the above definition shows, this underlying engine can be easily exposed through the common choose-your-own-adventure interface.

4.2 Type providers

The type provider mechanism in F# operates at the level of the type system. It is not a merely an editor feature. A type provider for a data source, such as the World Development Indicators database, generates a collection of types that model the external data source. In F#, the types are classes with members that implement the logic to retrieve data at runtime.

The auto-completion mechanism in F# code editors, which implements the choose-your-own-adventure interaction pattern, is not specific to type providers. It offers a list of members of an object based on its type. We model the completion as an iterative process, repeatedly adding further members to a chain. The state σ thus consists of an initial expression on which the completion is invoked, a chain of selected members and the type of the last member.

To model the completion mechanism, we also need to model information about types. We loosely follow the Foo calculus model [20] and write $\mathbb{C}$ for a set of class definitions, each consisting of an implicit class constructor and a collection of members M:

$$\begin{aligned}\sigma &= e.\iota_1.[\ldots].\iota_n, C \\ \mathbb{C} &= \{C \mapsto \mathsf{type}\ C(\overline{x:\tau}) = \overline{M},\ \ldots\ \} \\ M &= \mathsf{member}\ \iota\!:\!C = e\end{aligned}$$

Each member in the Foo calculus consists of a of a name ι, return type C and implementation e. For our purposes, we only need the type information and so the operations that define the choose-your-own-adventure are parameterized by the set of classes $\mathbb{C}$.

The $\mathsf{choices}_{\mathbb{C}}$ operation finds the class definition corresponding to the type of the last member in the current chain. It offers choices appending each of the available members to the current chain. The $\mathsf{choose}_{\mathbb{C}}$ operation returns the constructed member chain:

$$
\begin{array}{l}
\mathsf{choices}_{\mathbb{C}}(e.\iota_1.[\ldots].\iota_n, C) = \\
\qquad \{\ \iota' \mapsto (e.\iota_1.[\ldots].\iota_n.\iota', C') \\
\qquad\quad \iota'' \mapsto (e.\iota_1.[\ldots].\iota_n.\iota'', C''),\ \ldots\ \} \\
\quad \text{where } \mathbb{C}(C) \;=\; \mathsf{type}\ C(\overline{x:\tau}) = M', M'', \ldots \\
\qquad\quad \text{and } M' \;=\; \mathsf{member}\ \iota' : C' = e'
\end{array}
$$

$$\mathsf{choose}_{\mathbb{C}}(e.\iota_1.[\ldots].\iota_n, C) = e.\iota_1.\ldots.\iota_n$$

The model does not directly refer to type providers. Those are responsible solely for generating the type definitions in $\mathbb{C}$ as documented in earlier work [20]. It is worth noting that the type provider for data exploration, implemented by The Gamma, additionally needs to generate classes lazily [16]. To model this aspect, the simple lookup $\mathbb{C}(C)$ needs to be replaced with an operation that returns the type definition, alongside with a new context $\mathbb{C}'$ that contains additional generated type definitions (return types for all the members of the class C).

The model follows the internal mode of embedding the interaction in the program. It is easy to define the decode operation that takes the resulting generated expression and returns the sequence of choices, because the choices are items of the member chain. A slight caveat is that the completion is not invoked on an empty hole, but on a hole that contains the initial expression on which the completion is applied. We can model this using filled holes [15] and write $?_e$ for a hole containing the initial expression e. The $\mathsf{init}(?_e)$ operation then returns e alongside with an empty chain and the type of e.

As noted earlier, The Gamma does not embed query expressions directly into the generated code. It uses the same model as type providers and generates choices as members of types behind the scenes. We return to the differences between the two models in Section 5.

4.3 AI assistants

AI assistants guide the analyst through a data wrangling task. They generate a data cleaning script, taking into account constraints selected by the user. Most AI assistants obtain the script by performing statistical optimization with respect to a set of constraints specified by the user. That is, they look for an expression from the set of all possible expressions that optimizes some objective function that assigns score to the expression with respect to the given input data. Note that AI assistants do not iterate over all possible expressions. They use a machine learning method to approximate a solution to the problem.

An optimization-based AI assistant [21] thus provides another, very different, way of implementing the choose-your-own-adventure pattern. The assistant operates with respect to some input data X that does not change during the interaction and so we parameterize the choose-your-own-adventure calculus operations by the data. The input data X can be actual input data or a representative sample and so the AI assistant can be use past data to infer a cleaning script that will be used on new inputs.

The state σ consists of a set of constraints specified by the user. We write c for individual constraints and $\boldsymbol{c}$ for a set of constraints. The initial state is an empty set:

$$
\begin{array}{rcl}
\sigma &=& \{c_1, \ldots, c_n\} \\
\sigma_0 &=& \emptyset
\end{array}
$$

Unlike in the previous examples, the crucial logic of an AI assistants is implemented in the choose operation. The operation runs the optimization algorithm to choose the best cleaning script for given constraints. Formally, this can be written using the arg max operator which finds an argument (an expression) for which the given function (scoring function) is maximized. The user-specified constraints can either restrict the set of possible expressions or influence the scoring function. More formally, we assume that:

- $E_{\boldsymbol{c}} \subseteq E$ is a set of expressions that respect constraints $\boldsymbol{c}$,
- $Q_{\boldsymbol{c}}(X, e)$ is a scoring function with respect to the constraints $\boldsymbol{c}$, which returns the score of an expression e, i.e., how good e is at cleaning the data X.

For a given set of constraints $\boldsymbol{c}$, the choose operation looks for $e \in E_{\boldsymbol{c}}$ with the largest score:

$$\mathsf{choose}_X(\boldsymbol{c}) = \arg\max_{e \in E_{\boldsymbol{c}}} Q_{\boldsymbol{c}}(X, e)$$

The actual implementation of the optimization uses various machine learning techniques to find the optimal expression. In case of datadiff, X is a pair of datasets X_1, X_2 to be reconciled. The AI assistant uses the Hungarian algorithm [24] to construct a matching of columns from X_1 and X_2. The generated expression is a sequence of patches that can be applied to X_2 in order to reconcile its structure with the sturcture of X_1. The constraints specified by the user restrict the space of possible column matchings and so they affect $E_{\boldsymbol{c}}$. The scoring function $Q_{\boldsymbol{c}}$ is independent of the constraints and computes a sum of distances between the statistical distributions of the columns from X_1 and a patched version of X_2.

The choices operation is responsible for generating possible constraints that the user may want to add to guide the inference. AI assistants typically offer the user options to prevent or adapt some aspect of the cleaning logic inferred by the system. For example, if datadiff matches two columns, it will offer a constraint to prevent the matching. It also generates constraints that let the user force a specific matching.

To implement $\mathsf{choices}_X$, optimization-based AI assistants first call $\mathsf{choose}_X(\sigma)$ to get the best expression e. Based on this, they generate possible constraints $c_1, c_2, \ldots$ that the user may want to choose from. The identifiers $\iota_1, \iota_2, \ldots$ provide a human-readable description of the constraints. Note that this operation is specific to the particular AI assistant. The $\mathsf{choices}_X$ operation then offers a list of constraint sets where the additional constraint is added to the previously collected set:

$$\mathsf{choices}_X(\boldsymbol{c}) = \{\iota_1 \mapsto \boldsymbol{c} \cup \{c_1\}, \iota_2 \mapsto \boldsymbol{c} \cup \{c_2\}, \ldots\}$$

The integration of an AI assistant, as described here, has to follow the external mode of embedding. The interaction with the assistant results in a cleaning script (expression), but there is no way of reconstructing the constrains used to guide the optimization. To support internal embedding, the choose operation would need to explicitly include the constraints in the resulting expression. However, rerunning the choose operation with the same constraints may result in a different cleaning script if the machine learning algorithm is probabilistic.

4.4 Interactive theorem proving

The internal mode can also be used when working with theorem provers.

In Coq, where proofs are represented as series of tactics,

However, this would also be case in a completion engine for the Coq proof system that would offer a choice of tactics to apply, because Coq proofs are represented

5 Properties

6 Applications

[22]

7 Limitations

hard to edit - you cannot easily go back

References

1 Michael D. Adams, Eric Griffis, Thomas J. Porter, Sundara Vishnu Satish, Eric Zhao, and Cyrus Omar. Grove: A bidirectionally typed collaborative structure editor calculus. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. `doi:10.1145/3704909`.

2 Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. A user's guide to alf. Technical report, Chalmers University of Technology, Sweden, 1994. Unpublished Draft. URL: `https://people.cs.nott.ac.uk/psztxa/publ/alf94.pdf`.

3 Alan F. Blackwell and Thomas R. G. Green. Notational systems – the cognitive dimensions of notations framework. In John M. Carroll, editor, *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*, pages 103–134. Morgan Kaufmann, San Francisco, 2003.

4 Edwin Brady. *The Idris Programming Language*, pages 115–186. Springer International Publishing, Cham, 2015. `doi:10.1007/978-3-319-15940-9_4`.

5 Edwin Brady. Idris 2: Quantitative type theory in practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:26, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECOOP.2021.9`.

6 E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970. `doi:10.1145/362384.362685`.

7 Damian Frölich and L. Thomas van Binsbergen. On the soundness of auto-completion services for dynamically typed languages. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE '24, page 107–120, NY, USA, 2024. Association for Computing Machinery. `doi:10.1145/3689484.3690734`.

8 Richard P. Gabriel. The structure of a programming language revolution. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, page 195–214, New York, NY, USA, 2012. Association for Computing Machinery. `doi:10.1145/2384592.2384611`.

9 Thomas R. G. Green. Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay, editors, *People and Computers V: Proceedings of the Fifth Conference of the British Computer Society*, pages 443–460, Cambridge, UK, 1989. Cambridge University Press.

10 Joel Jakubovic, J. Edwards, and T. Petricek. Technical dimensions of programming systems. *Art Sci. Eng. Program.*, 7(3), 2023. `doi:10.22152/PROGRAMMING-JOURNAL.ORG/2023/7/13`.

11 Lena Magnusson and Bengt Nordström. The alf proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 213–237, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

12 Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. Bidirectional evaluation with direct manipulation. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. `doi:10.1145/3276497`.

13 Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999. URL: `https://era.ed.ac.uk/handle/1842/374`.

14 Jaroslav Nešetřil. Strom jako matematická struktura – i v umění. KAM Series 742, Department of Applied Mathematics, Charles University, 2005. URL: `https://kam.mff.cuni.cz/~kamserie/serie/clanky/2005/s742.pdf`.

15 Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live functional programming with typed holes. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. `doi:10.1145/3290327`.

16 Tomas Petricek. Data exploration through dot-driven development. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 21:1–21:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPICS.ECOOP.2017.21`.

17 Tomas Petricek. Foundations of a live data exploration environment. *Art Sci. Eng. Program.*, 4(3):8, 2020. `doi:10.22152/PROGRAMMING-JOURNAL.ORG/2020/4/8`.

18 Tomas Petricek. The gamma: Programmatic data exploration for non-programmers. In Paolo Bottoni, Gennaro Costagliola, Michelle Brachman, and Mark Minas, editors, *2022 IEEE*

Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2022, Rome, Italy, September 12-16, 2022, pages 1–7. IEEE, 2022. doi:10.1109/VL/HCC53370.2022.9833134.

19 Tomas Petricek, James Geddes, and Charles Sutton. Wrattler: Reproducible, live and polyglot notebooks. In *10th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2018)*, London, July 2018. USENIX Association. URL: https://www.usenix.org/conference/tapp2018/presentation/petricek.

20 Tomas Petricek, Gustavo Guerra, and Don Syme. Types from data: making structured data first-class citizens in F#. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 477–490, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2908080.2908115.

21 Tomas Petricek, Gerrit J. J. van den Burg, Alfredo Nazábal, Taha Ceritli, Ernesto Jiménez-Ruiz, and Christopher K. I. Williams. AI assistants: A framework for semi-automated data wrangling. *IEEE Trans. Knowl. Data Eng.*, 35(9):9295–9306, 2023. doi:10.1109/TKDE.2022.3222538.

22 Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. Exploratory and live, programming and coding: A literature study comparing perspectives on liveness. *The Art, Science, and Engineering of Programming*, 3(1):1, 2019. doi:10.22152/programming-journal.org/2019/3/1.

23 Mark Seemann. *Code That Fits in Your Head: Heuristics for Software Engineering*. Addison-Wesley Professional, Boston, 2021.

24 Charles A. Sutton, Timothy Hobson, James Geddes, and Rich Caruana. Data Diff: Interpretable, Executable Summaries of Changes in Distributions for Data Wrangling. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2279–2288. ACM, 2018. doi:10.1145/3219819.3220057.

25 Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, and Tomas Petricek. Themes in information-rich functional programming for internet-scale data sources. In Evelyne Viegas, Karin K. Breitman, and Judith Bishop, editors, *Proceedings of the 2013 Workshop on Data Driven Functional Programming, DDFP 2013, Rome, Italy, January 22, 2013*, pages 1–4. ACM, 2013. doi:10.1145/2429376.2429378.

26 Jan Liam Verter and Tomas Petricek. Don't call us, we'll call you: Towards mixed-initiative interactive proof assistants for programming language theory. *CoRR*, abs/2409.13872, 2024. Presented at the 5th International Workshop on Human Aspects of Types and Reasoning Assistants (HATRA 2024). arXiv:2409.13872, doi:10.48550/ARXIV.2409.13872.

8 Typesetting instructions – Summary

LIPIcs is a series of open access high-quality conference proceedings across all fields in informatics established in cooperation with Schloss Dagstuhl. In order to do justice to the high scientific quality of the conferences that publish their proceedings in the LIPIcs series, which is ensured by the thorough review process of the respective events, we believe that LIPIcs proceedings must have an attractive and consistent layout matching the standard of the series. Moreover, the quality of the metadata, the typesetting and the layout must also meet the requirements of other external parties such as indexing service, DOI registry, funding agencies, among others. The guidelines contained in this document serve as the baseline for the authors, editors, and the publisher to create documents that meet as many different requirements as possible.

Please comply with the following instructions when preparing your article for a LIPIcs proceedings volume.

Minimum requirements

- Use pdflatex and an up-to-date LaTeX system.
- Use further LaTeX packages and custom made macros carefully and only if required.
- Use the provided sectioning macros: `\section`, `\subsection`, `\subsubsection`, `\paragraph`, `\paragraph*`, and `\subparagraph*`.
- Provide suitable graphics of at least 300dpi (preferably in PDF format).
- Use BibTeX and keep the standard style (`plainurl`) for the bibliography.
- Please try to keep the warnings log as small as possible. Avoid overfull `\hboxes` and any kind of warnings/errors with the referenced BibTeX entries.
- Use a spellchecker to correct typos.

Mandatory metadata macros

Please set the values of the metadata macros carefully since the information parsed from these macros will be passed to publication servers, catalogues and search engines. Avoid placing macros inside the metadata macros. The following metadata macros/environments are mandatory:

- `\title` and, in case of long titles, `\titlerunning`.
- `\author`, one for each author, even if two or more authors have the same affiliation.
- `\authorrunning` and `\Copyright` (concatenated author names)
  The `\author` macros and the `\Copyright` macro should contain full author names (especially with regard to the first name), while `\authorrunning` should contain abbreviated first names.
- `\ccsdesc` (ACM classification, see `https://www.acm.org/publications/class-2012`).
- `\keywords` (a comma-separated list of keywords).
- `\relatedversion` (if there is a related version, typically the "full version"); please make sure to provide a persistent URL, e. g., at arXiv.
- `\begin{abstract}...\end{abstract}` .

Please do not ...

Generally speaking, please do not override the `lipics-v2021`-style defaults. To be more specific, a short checklist also used by Dagstuhl Publishing during the final typesetting is given below. In case of **non-compliance** with these rules Dagstuhl Publishing will remove the corresponding parts of LaTeX code and **replace it with the `lipics-v2021` defaults**. In serious cases, we may reject the LaTeX-source and expect the corresponding author to revise the relevant parts.

- Do not use a different main font. (For example, the `times` package is forbidden.)
- Do not alter the spacing of the `lipics-v2021.cls` style file.
- Do not use `enumitem` and `paralist`. (The `enumerate` package is preloaded, so you can use `\begin{enumerate}[(a)]` or the like.)
- Do not use "self-made" sectioning commands (e. g., `\noindent{\bf My Paragraph}`).
- Do not hide large text blocks using comments or `\iffalse` ... `\fi` constructions.
- Do not use conditional structures to include/exclude content. Instead, please provide only the content that should be published – in one file – and nothing else.
- Do not wrap figures and tables with text. In particular, the package `wrapfig` is not supported.
- Do not change the bibliography style. In particular, do not use author-year citations. (The `natbib` package is not supported.)

This is only a summary containing the most relevant details. Please read the complete document "LIPIcs: Instructions for Authors and the `lipics-v2021` Class" for all details and don't hesitate to contact Dagstuhl Publishing (`mailto:publishing@dagstuhl.de`) in case of questions or comments: `http://drops.dagstuhl.de/styles/lipics-v2021/lipics-v2021-authors/lipics-v2021-authors-guidelines.pdf`

9 Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetur adipiscing elit [**?**]. Praesent convallis orci arcu, eu mollis dolor. Aliquam eleifend suscipit lacinia. Maecenas quam mi, porta ut lacinia sed, convallis ac dui. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse potenti. Donec eget odio et magna ullamcorper vehicula ut vitae libero. Maecenas lectus nulla, auctor nec varius ac, ultricies et turpis. Pellentesque id ante erat. In hac habitasse platea dictumst. Curabitur a scelerisque odio. Pellentesque elit risus, posuere quis elementum at, pellentesque ut diam. Quisque aliquam libero id mi imperdiet quis convallis turpis eleifend.

▶ **Lemma 4** (Lorem ipsum). *Vestibulum sodales dolor et dui cursus iaculis. Nullam ullamcorper purus vel turpis lobortis eu tempus lorem semper. Proin facilisis gravida rutrum. Etiam sed sollicitudin lorem. Proin pellentesque risus at elit hendrerit pharetra. Integer at turpis varius libero rhoncus fermentum vitae vitae metus.*

Proof. Cras purus lorem, pulvinar et fermentum sagittis, suscipit quis magna.

Just some paragraph within the proof. Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

▷ Claim 5. content...

Proof. content...

1. abc abc abc ◁

◀

Listing 1 Useless code.

```
for i:=maxint to 0 do
begin
    j:=square(root(i));
end;
```

▶ **Corollary 6** (Curabitur pulvinar, [?]). *Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.*

▶ **Proposition 7.** *This is a proposition*

Proposition 7 and Proposition 7 . . .

9.1 Curabitur dictum felis id sapien

Curabitur dictum Corollary 6 felis id sapien Corollary 6 mollis ut venenatis tortor feugiat. Curabitur sed velit diam. Integer aliquam, nunc ac egestas lacinia, nibh est vehicula nibh, ac auctor velit tellus non arcu. Vestibulum lacinia ipsum vitae nisi ultrices eget gravida turpis laoreet. Duis rutrum dapibus ornare. Nulla vehicula vulputate iaculis. Proin a consequat neque. Donec ut rutrum urna. Morbi scelerisque turpis sed elit sagittis eu scelerisque quam condimentum. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Aenean nec faucibus leo. Cras ut nisl odio, non tincidunt lorem. Integer purus ligula, venenatis et convallis lacinia, scelerisque at erat. Fusce risus libero, convallis at fermentum in, dignissim sed sem. Ut dapibus orci vitae nisl viverra nec adipiscing tortor condimentum [?]. Donec non suscipit lorem. Nam sit amet enim vitae nisl accumsan pretium.

9.2 Proin ac fermentum augue

Proin ac fermentum augue. Nullam bibendum enim sollicitudin tellus egestas lacinia euismod orci mollis. Nulla facilisi. Vivamus volutpat venenatis sapien, vitae feugiat arcu fringilla ac. Mauris sapien tortor, sagittis eget auctor at, vulputate pharetra magna. Sed congue, dui nec vulputate convallis, sem nunc adipiscing dui, vel venenatis mauris sem in dui. Praesent a pretium quam. Mauris non mauris sit amet eros rutrum aliquam id ut sapien. Nulla aliquet fringilla sagittis. Pellentesque eu metus posuere nunc tincidunt dignissim in tempor dolor. Nulla cursus aliquet enim. Cras sapien risus, accumsan eu cursus ut, commodo vel velit. Praesent aliquet consectetur ligula, vitae iaculis ligula interdum vel. Integer faucibus faucibus felis.

- Ut vitae diam augue.
- Integer lacus ante, pellentesque sed sollicitudin et, pulvinar adipiscing sem.
- Maecenas facilisis, leo quis tincidunt egestas, magna ipsum condimentum orci, vitae facilisis nibh turpis et elit.

▶ Remark 8. content...

10 Pellentesque quis tortor

Nec urna malesuada sollicitudin. Nulla facilisi. Vivamus aliquam tempus ligula eget ornare. Praesent eget magna ut turpis mattis cursus. Aliquam vel condimentum orci. Nunc congue, libero in gravida convallis [?], orci nibh sodales quam, id egestas felis mi nec nisi. Suspendisse tincidunt, est ac vestibulum posuere, justo odio bibendum urna, rutrum bibendum dolor sem nec tellus.

▶ **Lemma 9** (Quisque blandit tempus nunc). *Sed interdum nisl pretium non. Mauris sodales consequat risus vel consectetur. Aliquam erat volutpat. Nunc sed sapien ligula. Proin faucibus sapien luctus nisl feugiat convallis faucibus elit cursus. Nunc vestibulum nunc ac massa pretium pharetra. Nulla facilisis turpis id augue venenatis blandit. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.*

Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque.

11 Morbi eros magna

Morbi eros magna, vestibulum non posuere non, porta eu quam. Maecenas vitae orci risus, eget imperdiet mauris. Donec massa mauris, pellentesque vel lobortis eu, molestie ac turpis. Sed condimentum convallis dolor, a dignissim est ultrices eu. Donec consectetur volutpat eros, et ornare dui ultricies id. Vivamus eu augue eget dolor euismod ultrices et sit amet nisi. Vivamus malesuada leo ac leo ullamcorper tempor. Donec justo mi, tempor vitae aliquet non, faucibus eu lacus. Donec dictum gravida neque, non porta turpis imperdiet eget. Curabitur quis euismod ligula.

References

1 Michael D. Adams, Eric Griffis, Thomas J. Porter, Sundara Vishnu Satish, Eric Zhao, and Cyrus Omar. Grove: A bidirectionally typed collaborative structure editor calculus. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. `doi:10.1145/3704909`.

2 Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. A user's guide to alf. Technical report, Chalmers University of Technology, Sweden, 1994. Unpublished Draft. URL: `https://people.cs.nott.ac.uk/psztxa/publ/alf94.pdf`.

3 Alan F. Blackwell and Thomas R. G. Green. Notational systems – the cognitive dimensions of notations framework. In John M. Carroll, editor, *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*, pages 103–134. Morgan Kaufmann, San Francisco, 2003.

4 Edwin Brady. *The Idris Programming Language*, pages 115–186. Springer International Publishing, Cham, 2015. `doi:10.1007/978-3-319-15940-9_4`.

5 Edwin Brady. Idris 2: Quantitative type theory in practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:26, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECOOP.2021.9`.

6 E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970. `doi:10.1145/362384.362685`.

7 Damian Frölich and L. Thomas van Binsbergen. On the soundness of auto-completion services for dynamically typed languages. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE '24, page 107–120, NY, USA, 2024. Association for Computing Machinery. `doi:10.1145/3689484.3690734`.

8 Richard P. Gabriel. The structure of a programming language revolution. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, page 195–214, New York, NY, USA, 2012. Association for Computing Machinery. `doi:10.1145/2384592.2384611`.

9 Thomas R. G. Green. Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay, editors, *People and Computers V: Proceedings of the Fifth Conference of the British Computer Society*, pages 443–460, Cambridge, UK, 1989. Cambridge University Press.

10 Joel Jakubovic, J. Edwards, and T. Petricek. Technical dimensions of programming systems. *Art Sci. Eng. Program.*, 7(3), 2023. `doi:10.22152/PROGRAMMING-JOURNAL.ORG/2023/7/13`.

11 Lena Magnusson and Bengt Nordström. The alf proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 213–237, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

12 Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. Bidirectional evaluation with direct manipulation. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. `doi:10.1145/3276497`.

13 Conor McBride. *Dependently Typed Functional Programs and their Proofs.* PhD thesis, University of Edinburgh, 1999. URL: `https://era.ed.ac.uk/handle/1842/374`.

14 Jaroslav Nešetřil. Strom jako matematická struktura – i v umění. KAM Series 742, Department of Applied Mathematics, Charles University, 2005. URL: `https://kam.mff.cuni.cz/~kamserie/serie/clanky/2005/s742.pdf`.

15 Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live functional programming with typed holes. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. `doi:10.1145/3290327`.

16 Tomas Petricek. Data exploration through dot-driven development. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 21:1–21:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPICS.ECOOP.2017.21`.

17 Tomas Petricek. Foundations of a live data exploration environment. *Art Sci. Eng. Program.*, 4(3):8, 2020. `doi:10.22152/PROGRAMMING-JOURNAL.ORG/2020/4/8`.

18 Tomas Petricek. The gamma: Programmatic data exploration for non-programmers. In Paolo Bottoni, Gennaro Costagliola, Michelle Brachman, and Mark Minas, editors, *2022 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2022, Rome, Italy, September 12-16, 2022*, pages 1–7. IEEE, 2022. `doi:10.1109/VL/HCC53370.2022.9833134`.

19 Tomas Petricek, James Geddes, and Charles Sutton. Wrattler: Reproducible, live and polyglot notebooks. In *10th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2018)*, London, July 2018. USENIX Association. URL: `https://www.usenix.org/conference/tapp2018/presentation/petricek`.

20 Tomas Petricek, Gustavo Guerra, and Don Syme. Types from data: making structured data first-class citizens in F#. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 477–490, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2908080.2908115`.

21 Tomas Petricek, Gerrit J. J. van den Burg, Alfredo Nazábal, Taha Ceritli, Ernesto Jiménez-Ruiz, and Christopher K. I. Williams. AI assistants: A framework for semi-automated data wrangling. *IEEE Trans. Knowl. Data Eng.*, 35(9):9295–9306, 2023. `doi:10.1109/TKDE.2022.3222538`.

22 Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. Exploratory and live, programming and coding: A literature study comparing perspectives on liveness. *The Art, Science, and Engineering of Programming*, 3(1):1, 2019. `doi:10.22152/programming-journal.org/2019/3/1`.

23 Mark Seemann. *Code That Fits in Your Head: Heuristics for Software Engineering.* Addison-Wesley Professional, Boston, 2021.

24 Charles A. Sutton, Timothy Hobson, James Geddes, and Rich Caruana. Data Diff: Interpretable, Executable Summaries of Changes in Distributions for Data Wrangling. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2279–2288. ACM, 2018. `doi:10.1145/3219819.3220057`.

25 Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, and Tomas Petricek. Themes in information-rich functional programming for internet-scale data sources. In Evelyne Viegas, Karin K. Breitman, and Judith Bishop, editors, *Proceedings of the 2013 Workshop on Data*

Driven Functional Programming, DDFP 2013, Rome, Italy, January 22, 2013, pages 1–4. ACM, 2013. doi:10.1145/2429376.2429378.

26 Jan Liam Verter and Tomas Petricek. Don't call us, we'll call you: Towards mixed-initiative interactive proof assistants for programming language theory. *CoRR*, abs/2409.13872, 2024. Presented at the 5th International Workshop on Human Aspects of Types and Reasoning Assistants (HATRA 2024). arXiv:2409.13872, doi:10.48550/ARXIV.2409.13872.

A Styles of lists, enumerations, and descriptions

List of different predefined enumeration styles:

- `\begin{itemize}...\end{itemize}`
- ...
- ...

1. `\begin{enumerate}...\end{enumerate}`
2. ...
3. ...

(a) `\begin{alphaenumerate}...\end{alphaenumerate}`
(b) ...
(c) ...

(i) `\begin{romanenumerate}...\end{romanenumerate}`
(ii) ...
(iii) ...

(1) `\begin{bracketenumerate}...\end{bracketenumerate}`
(2) ...
(3) ...

Description 1 `\begin{description} \item[Description 1]  ...\end{description}`
Description 2 Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque.
Description 3 ...

Proposition 13 and Proposition 13 ...

B Theorem-like environments

List of different predefined enumeration styles:

▶ **Theorem 10.** *Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque.*

▶ **Lemma 11.** *Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque.*

▶ **Corollary 12.** *Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque.*

▶ **Proposition 13.** *Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque.*

▶ **Conjecture 14.** *Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque.*

▶ **Observation 15.** *Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque.*

▶ **Exercise 16.** *Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque.*

▶ **Definition 17.** *Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque.*

▶ **Example 18.** Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque.

▶ Note 19. Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque.

▶ Note. Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque.

▶ Remark 20. Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque.

▶ Remark. Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque.

▷ Claim 21. Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque.

▷ Claim. Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque.

Proof. Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque. ◀

Proof. Fusce eu leo nisi. Cras eget orci neque, eleifend dapibus felis. Duis et leo dui. Nam vulputate, velit et laoreet porttitor, quam arcu facilisis dui, sed malesuada risus massa sit amet neque. ◁