The Choose-Your-Own-Adventure Calculus

Tomas Petricek
tomas@tomasp.net
Charles University
Prague, Czechia

Jan Liam Verter
verter@d3s.mff.cuni.cz
Charles University
Prague, Czechia

Mikolas Fromm
mikolas@fromm.one
Imperial College London
London, UK

Abstract

In many programming systems, the user can interactively construct a program by repeatedly triggering a completion mechanism and choosing one of the offered options. This is the case with code editors for object-oriented languages (choosing a member), data exploration environments (choosing a transformation), but also with theorem provers (choosing a tactic) and structure editors (choosing a grammar rule).

We capture the essence of this interaction pattern through a small formal model called the *choose-your-own-adventure calculus*. The model serves three roles. First, it reveals subtle differences between instances of the interaction pattern. Second, it enables transfer of interaction techniques across different domains by describing them abstractly in terms of the calculus. Examples include mixed-initiative interaction, AI assistants and programming by demonstration. Third, it lets us characterise desirable properties of a programming system, including correctness, completeness and uniqueness.

More generally, the paper shows that programming language theory can be used to study not just languages, but also rich interactive programming systems. We argue that such systems deserve at least as much attention as languages.

CCS Concepts: • **Software and its engineering** → **Formal language definitions**; • **Human-centered computing** → *Interaction paradigms.*

Keywords: Interactive programming systems, Type providers, Theorem provers, Structure editors, AI assistants, Mixed-initiative interaction

ACM Reference Format:
Tomas Petricek, Jan Liam Verter, and Mikolas Fromm. 2026. The Choose-Your-Own-Adventure Calculus. In *Proceedings of the 2026 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '26), October 4–9, 2026, Oakland, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3840586.3843213

Onward! '26, Oakland, CA, USA

ACM ISBN 979-8-4007-2934-8/2026/10
https://doi.org/10.1145/3840586.3843213

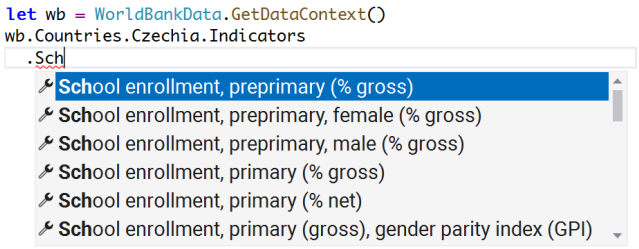


Figure 1. F# code editor showing all possible code completions offered by the World Bank type provider.

1 Introduction

Many interactive programming systems, ranging from code editors for object-oriented languages and data exploration systems, to interactive theorem provers and structure editors, exhibit a remarkably similar pattern of interaction. They offer the user, who can be a programmer, a data scientist or a proof engineer, a list of options that they can select from in order to complete their program, script or proof.

There are subtle differences between the various implementations of the general pattern. In some systems, the resulting source code contains 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. Invoking a proof search or case split in Idris [10] 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 completions that are valid at a given location, or it may list only a subset of the valid options. In some cases, it may even include incorrect options as is the case in auto-completion for dynamic languages [16].

This paper formally captures the recurring interaction pattern. Our contributions are:

- We motivate the formalism by reviewing five systems that implement a variation on the interaction pattern (Section 2). These include type providers in F# [56] and in The Gamma [40, 43], AI assistants for semi-automated data wrangling [46], tooling for interactive theorem provers [4, 10, 59] and structure editors [7].

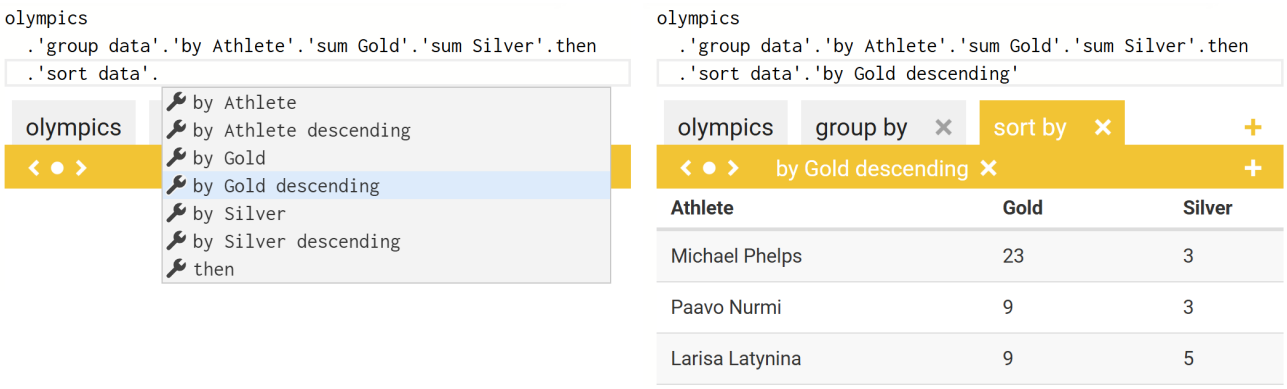


Figure 2. Constructing a query in The Gamma. We count the number of gold and silver medals for each athlete and sort the data by the number of gold medals. The entire query is constructed by repeatedly choosing one of the offered options.

- We capture the pattern as the *choose-your-own-adventure calculus*, a small formal model of interactive programming systems where a user constructs a program by repeatedly choosing from a list of options (Section 3).
- We define three desirable properties—*correctness*, *completeness* and *uniqueness*—and argue why they matter in practice (Section 5). To understand how the interaction pattern integrates in a programming environment, we define the notion of *expression completion* and capture *reconstructible* completion as a special case (Section 3).
- We show that multiple advanced interaction techniques—mixed-initiative interaction, AI assistants and programming by demonstration—can be defined on top of the calculus (Section 6), illustrating how the calculus supports transfer of ideas across different 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 programming language theory methods to precisely describe the pattern. Moreover, our work confirms that such methods can be effective for studying not just *programming languages*, but also interactive *programming systems* [23].

2 Motivation

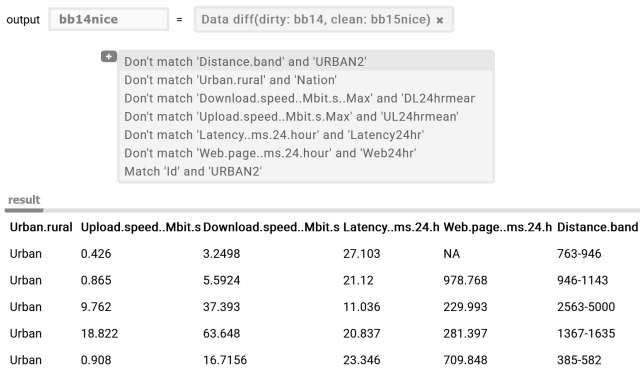
Research on programming has long focused on programming languages as syntactic entities, neglecting the interactive programming environments in which they are inevitably embedded [18]. Notably, in many of the motivating examples we discuss, the interactive aspect of the system is only described in supplementary materials [4, 10, 56]. Programming language theory has only recently started to be used to study interactive programming environments [1, 32]. We contribute to this line of work by studying a common interaction pattern. Five instances motivate the general pattern.

Type Providers. Type providers [13, 56] are a mechanism for integrating external data sources into the type system of a programming language. In F#, a type provider is a compiler extension, 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 statically typed representation of the data, typically objects with members. Type providers can infer the type from a JSON, XML or CSV sample [45] 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 object wb allows the programmer to access any indicator of any country in the database by choosing a [Country] and an [Indicator] in a chain of members wb.Countries.[Country].Indicators.[Indicator]. The result is a time series with values for the given indicator and country. The example is a special case of a type provider for slicing n-dimensional data cubes [43]. The programmer fixes a value for two of the three dimensions (country, indicator, time) and obtains data indexed by the remaining dimension.

When using the type provider, the programmer types the first line of code to initialise the type provider 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, a pattern playfully termed *dot-driven development* by Phil Trelford [49].

Data Exploration. The Gamma [43] is a programmatic data exploration environment for non-programmers, which uses type providers as the primary programming mechanism. Type providers are used not just for data access, but also for constructing queries. The example shown in Figure 2 lets the user construct an SQL-like query by repeatedly choosing operations and their parameters [40]. The type provider keeps track of the schema and uses it to generate all possible valid parameters. When sorting data, it generates an object with two members per column—one for ascending and one for descending order. The grouping operation first offers all



Urban.rural	Upload.speed..Mbit.s	Download.speed..Mbit.s	Latency..ms.24.h	Web.page..ms.24.h	Distance.band
Urban	0.426	3.2498	27.103	NA	763-946
Urban	0.865	5.5924	21.12	978.768	946-1143
Urban	9.762	37.393	11.036	229.993	2563-5000
Urban	18.822	63.648	20.837	281.397	1367-1635
Urban	0.908	16.7156	23.346	709.848	385-582

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

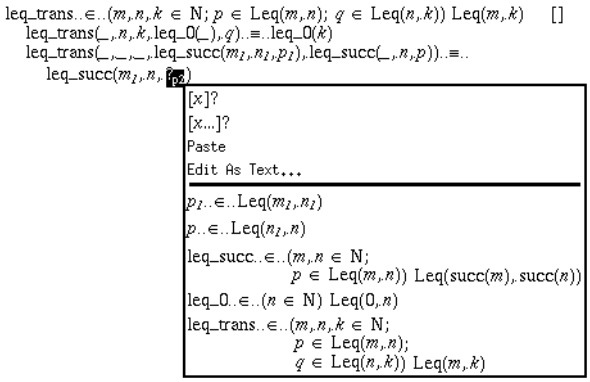


Figure 4. Constructing a proof of the transitivity of the ≤ relation in the Alf editor. The user is offered a range of variables and constructors in scope at the current location. [4]

columns as possible grouping keys and then lets the user choose from predefined aggregations (sum, count, average, concatenate). The system also evaluates the query on the fly, providing a live preview [42].

The interaction pattern is the same as in the previous example. After the user triggers 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, because the user can keep adding operations, and so the types have to be 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 [46] (not to be confused with modern LLM-based AI assistants).

An AI assistant guides the analyst through a data wrangling problem such as reconciling mismatched datasets, filling missing values or inferring data formats and types. It attempts to solve the problem automatically and suggests an initial data transformation, but it also generates 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 datadiff [54], running in a Wrattler notebook [44], to merge broadband quality data published by Ofcom for two consecutive 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 select six columns from the year 2015 and want to find matching data from 2014.

When the AI assistant 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 statistical distributions. 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.

The analyst again constructs the data transformation by repeatedly choosing from a list of options, but the way the interaction pattern is implemented differs in two respects. First, with type providers, we are gradually constructing a program by adding operations to a method chain. An AI assistant synthesises a data transformation and we are gradually adding constraints to control the synthesis. Second, the completion list for type providers offers all possible members of the object. Here, the list offers constraints recommended by the AI assistant and it may not be complete.

Interactive Theorem Provers. The fourth example of the pattern can be found in interactive theorem provers. When writing programs in systems such as Idris [11], the user typically works by stating the desired conclusion and filling the implementation with a hole. The system provides interactive editing capabilities to fill the holes [33]. It can, for example, generate a case split or search for a proof [10].

Idris provides key bindings to invoke completions, but the functionality could also be offered through a user interface. An example that illustrates this is the interactive editor for the Alf [29] theorem prover (Figure 4), which is based on a gradual refinement of an incomplete proof object [4]. The user is proving the transitivity of the ≤ relation for Peano arithmetic. 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 which can be used to complete the proof. After choosing leq_trans, two new holes are generated for its arguments. These can again be 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

a proof by filling holes. There are two subtle differences. First, each completion directly refines the proof that the user is editing. Second, a completion may generate multiple new holes, rather than just appending to a chain of operations.

Alf is a historical example, but a similar user interface could be built for Idris or Rocq. The two systems 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 Rocq, 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.

Structure Editors. The final instance of the pattern comes from structure editors. They make it possible to construct programs by manipulating the abstract syntax tree rather than by working with text. The interaction with a structure editor can be based on menus [57], key bindings [60], tiles [34] and various other approaches. Many of these fit the choose-your-own-adventure pattern.

Sandblocks [7] is a recent structure editor that is interesting for two reasons. First, the editor is automatically generated from a grammar. Second, the editor combines keyboard input with context menus (Figure 5). In the above example, the user is typing code and needs to fill a placeholder for an expression. After they type the start of the expression, the editor offers a completion list with production rules from the grammar that are valid in the given context and are compatible with the text typed so far. The user can continue typing (to refine the recommendation), but they can also select a rule from the menu. This completes the expression according to the production rule, generating further empty holes that can, in turn, be interactively filled with code.

In Sandblocks, context menus serve as hints, and users of the editor often construct programs through typing, but the example shows that structure editors can also be based on the pattern captured in this paper. The editor can offer all possible production rules based on the given grammar and enable the user to construct an entire program by repeatedly choosing one of the rules and manually entering string literals and identifier names.

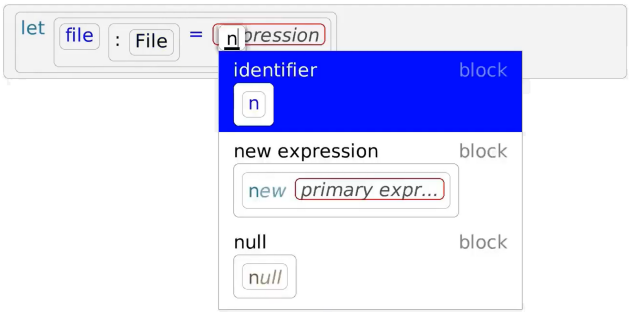


Figure 5. Constructing a program using Sandblocks. The user is typing the start of an expression and the context menu offers possible production rules according to a grammar. [7]

3 Formal Model

The five example systems discussed in the previous section share a common pattern of interaction. In all cases, the system repeatedly offers the user a range of options to choose from. Each of the options is designated by an identifier. During the process, the system maintains a state that determines subsequent options. The state may not be visible to the user, but the user can request the program constructed so far.

3.1 The choose-your-own-adventure Calculus

We now introduce the choose-your-own-adventure calculus, a small formal abstraction that captures the recurring interaction pattern. Despite its simplicity, the calculus will help us understand the differences between individual implementations of the pattern (Sections 4 and 5).

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 at each step (another system implementing the pattern, discussed in Section 4.4, refers to this structure as *programming by navigation* [28]). In the following definition, the 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* choices, choose *such that:*

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

The definition is not a conventional programming language calculus in that it does not define a concrete syntax with reduction rules. It is an abstract algebraic description of the structure of a system that supports the choose-your-own-adventure interaction pattern. The definition is close to that of an AI assistant [46], which uses data-wrangling terminology but is structurally similar. Note that the definition may also describe graphs with cycles. A system where the user can return to a previously visited state may not be practically useful, but it does not pose a theoretical problem.

3.2 Interactive Program Construction

The choose-your-own-adventure pattern is often used to interactively construct parts of larger programs. We thus also need to formally define how the calculus is used to complete programs in a host language. This definition will let us ask, for example, whether the steps of the original interaction can be reconstructed from the completed program.

Expression Completion. To model partial programs, 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. When invoked on an expression containing multiple holes, the system starts by completing the first hole.

We write $E[\cdot]$ 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 (Expression completion)**.** *An expression $E[?]$ is completed as $E[e]$ via an interaction with a choose-your-own-adventure system consisting of* choices *and* choose *if:*

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

As we will see when we revisit the earlier examples formally, this way of invoking a choose-your-own-adventure system is used, for example, in interactive theorem provers such as Idris. In these 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 which options, identified by $\iota_1, \ldots, \iota_n$, were selected in the process.

Reconstructible Completion. We asked whether a trace of the interaction can be reconstructed from the code of an interactively constructed program when considering several of the earlier examples. Systems that have this property include type providers, where the sequence of object members directly appears in the source code. The same would be the case in a completion system offering tactics to apply in Rocq, because the resulting proof would consist of a sequence formed by the selected tactics.

More formally, if we follow a sequence of choices $\iota_1, \ldots, \iota_n$ to construct a program e, is it possible to recover the sequence of choices through which it was constructed just from the program e? If this is the case, we say that expression completion is *reconstructible*. To capture the notion formally, we also need an operation decode that extracts identifiers of invoked completions from an expression. For some systems, the decode operation may also need the completion context $E[?]$, which defines, for example, variables in scope.

Definition 3 (Reconstructible expression completion)**.** *An expression $E[?]$ is reconstructibly completed as $E[e]$ through an interaction with a choose-your-own-adventure system consisting of* choices *and* choose *if:*

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

If the integration of a choose-your-own-adventure system with a host programming language uses reconstructible expression completion, we can recover the choices through which the user constructed the expression e in a completion context $E[e]$. 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 $\iota_1, \ldots, \iota_n$.

Filling Typed Holes. The notion of a hole is shared with other interactive and live programming environments. Hazel [38] equips incomplete programs with both a static and a dynamic semantics. The *livelits* [37] extension lets the user fill a typed hole using a user-defined graphical interface embedded in the code. This can be an arbitrary interface and so the mechanism could also support integrating choose-your-own-adventure pattern implementations.

4 Examples

We now show how the earlier examples fit our formal model. All five examples rely on some domain-specific logic. We describe what information the logic provides, but refer elsewhere for full details.

To illustrate how the formalism reveals subtle differences, we start with a data exploration system that is inspired by The Gamma but differs in that the interactively constructed program calls general-purpose data processing operations, rather than accessing a sequence of provided class members.

4.1 Data Exploration

In The Gamma, users interactively construct a query that transforms the given input data. The query is a sequence of operations with parameters, $op(p_1, \ldots, p_n)$, modelled after relational algebra [14], but it 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.

The model described in this section 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»

In The Gamma, the individual identifiers become object members and are included as a member chain in the generated code. In the simpler model presented here, the completion 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 [8, 19], the latter has a greater closeness of mapping, while the former

is less cognitively demanding to read for a non-programmer. In Section 5, we show that the two implementations of the pattern differ in 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 parameter values $p', p'', \ldots$ 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 generate the identifiers $\iota', \iota'', \ldots$ such as «by Gold descending» based on the state and the parameter value. 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):

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

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 append the additional operation:

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

Finally, the choose operation takes the state σ and generates an expression that represents the data transformation. Generating the expression 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:

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

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 wrapped and exposed through the common choose-your-own-adventure interface.

4.2 Type Providers

An F# type provider for a data source, such as the World Development Indicators database, generates a collection of types that model the external data source. The types are classes with members that retrieve the data at run time. The auto-completion mechanism that implements the interaction pattern in F# 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. We also need a representation of the type information. We loosely follow the Foo calculus model [45] and write $\mathbb{C}$ for a set of class definitions, each consisting of an implicit class constructor and a collection of members M:

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

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

The $\text{choices}_{\mathbb{C}}$ operation finds the class corresponding to the type of the last member in the chain and generates choices appending each of the available members to the current chain; $\text{choose}_{\mathbb{C}}$ returns the constructed member chain:

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

The model does not directly refer to type providers. Those are responsible solely for generating the type definitions in $\mathbb{C}$ as described in earlier work [45]. For simplicity, we omit the laziness that is necessary for the type provider for data exploration in The Gamma [40].

The example follows the reconstructible expression completion model. The decode operation takes the resulting generated expression and returns the sequence of choices, which are the items of the member chain. (In reality, completion is not invoked on an empty hole, but on an initial expression; this could be modelled using filled holes [38].)

As noted earlier, The Gamma does not embed query expressions directly into the generated code. It uses the model presented in this section 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 generate data cleaning scripts, taking into account constraints selected by the user. They typically obtain the script by performing statistical optimisation with respect to a set of constraints specified by the user [46]. They operate with respect to some input data X that do not change during the interaction. X can be the actual input or a representative sample, meaning that AI assistants can use past data to infer a cleaning script that will be used on new inputs.

The core logic is implemented in the choose operation, which finds the best data wrangling script for a given set of constraints. Rather than iterating over all possible expressions, AI assistants approximate the solution using machine learning. Formally, the logic can be captured using the arg max operator which finds an argument (script) for which the given scoring function is maximised. The user-specified constraints can restrict the set of possible expressions or influence the scoring function. We write c for individual constraints, $\boldsymbol{C}$ for a set of constraints and 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.

The state σ of the system consists of a set of constraints specified by the user. For a given set of constraints $\boldsymbol{C}$, the choose operation looks for $e \in E_{\boldsymbol{C}}$ with the largest score:

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

In datadiff, X is a pair of datasets X_1, X_2 to be reconciled. The optimisation looks for a matching of columns from X_1 and X_2 [54]. The solution is a sequence of patches that can be applied to X_2 in order to reconcile its structure with the structure 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 the patched version of X_2.

The choices operation generates constraints that the user may want to add to guide the inference. This is specific to each AI assistant; datadiff generates constraints that prevent an inferred matching or let the user force a specific matching.

To implement choices_X, optimisation-based AI assistants first call $\text{choose}_X(\sigma)$ 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. choices_X then offers a list of constraint sets with one of the additional constraints (the initial state σ_0 is an empty set):

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

The expression completion for an AI assistant is not reconstructible. The interaction results in a cleaning script, but there is no way of reconstructing the constraints used to guide the optimisation.

$$\frac{}{\Gamma, x : \tau \vdash \tau \Rightarrow x}\ \text{(syn-var)} \qquad \frac{b \in \{\text{true}, \text{false}\}}{\Gamma \vdash \text{bool} \Rightarrow b}\ \text{(syn-bool)}$$

$$\frac{}{\Gamma \vdash \tau \Rightarrow ?_\tau}\ \text{(syn-hole)} \qquad \frac{\Gamma, x : \tau_1 \vdash \tau_2 \Rightarrow e}{\Gamma \vdash \tau_1 \to \tau_2 \Rightarrow \lambda x.e}\ \text{(syn-lambda)}$$

$$\frac{\Gamma \vdash \tau_1 \to \tau_2 \Rightarrow e_1 \quad \Gamma \vdash \tau_1 \Rightarrow e_2}{\Gamma \vdash \tau_2 \Rightarrow e_1\ e_2}\ \text{(syn-app)}$$

$$\frac{\Gamma \vdash \tau \Rightarrow e_1 \quad \Gamma \vdash \tau \Rightarrow e_2 \quad \Gamma \vdash \text{bool} \Rightarrow e}{\Gamma \vdash \tau \Rightarrow \text{if } e \text{ then } e_1 \text{ else } e_2}\ \text{(syn-cond)}$$

Figure 6. Illustrative type-directed program synthesis rules.

Making the completion for AI assistants reconstructible is challenging in practice. Naively, the choose operation could explicitly include the collected constraints in the resulting expression. However, running the choose operation with the same constraints may result in a different cleaning script if the underlying machine learning algorithm is probabilistic. This also means that choices might offer different constraints to add and so the sequence of choices stored in the code may no longer match the available choices.

4.4 Program Synthesis

Although interactive theorem provers and editors for dependently typed languages implement interactions closely related to the choose-your-own-adventure calculus, there is no existing system that fits the pattern exactly. The closest fit is the recently envisioned mixed-mode interaction theorem prover [59]. This section sketches a possible implementation of the pattern for theorem proving and program synthesis.

Theorem Provers. There are two approaches to interacting with a theorem prover. In Rocq, the user writes a sequence of tactics that transform proof goals. Interactive editors for Rocq [5, 47] provide auto-completion, which recommends available tactics, hypotheses and local definitions, but these does not filter them based on what is valid in a given context.

In Agda or Idris, the user writes a term of a type that represents the theorem. Interactive editors [10] offer commands that transform the selected term by adding a case split, a missing case or by automatically searching for a proof, but those do not cover all possible completions.

The implementation of the choose-your-own-adventure pattern for both systems would be similar. Based on the current sub-goal, the system would recommend tactics that can be applied to the sub-goal. In Rocq, the selected tactic would be added to the sequence; in Idris, it would be applied to the current term. The difference between the two approaches is that the expression completion for Rocq would be reconstructible (the sequence of interactions can be recovered from the sequence of tactics in the generated proof).

Type-directed Synthesis. Thanks to the equivalence between programs and proofs, techniques akin to tactic-based theorem proving have also emerged in work on type-directed program synthesis [25]. As illustrated in Figure 6, the synthesis process can be described as a set of rules of the form $\Gamma \vdash \tau \Rightarrow e$ that describe ways of synthesising expressions e of a type τ. Existing implementations of the mechanism typically aim to automate program synthesis by using richer types or examples [22, 39, 48], but the rules could also guide an interactive choose-your-own-adventure system. If the interaction is invoked to fill a typed hole $?_\tau$ in a context Γ, the system could collect all expressions e such that $\Gamma \vdash \tau \Rightarrow e$ and offer a choice of those options. Note that the definition in Figure 6 synthesises sub-expressions recursively. A choose-your-own-adventure system may proceed one step at a time, filling sub-expressions with typed holes using (syn-hole).

Consider a system akin to Idris where the user aims to construct a term e of type τ. The term may contain typed holes written as $?_\tau$ and a relation $\Gamma \vdash \tau \Rightarrow e$ provides ways of synthesising terms of type τ. We write $E[?_\tau]$ for a completion context containing a (typed) hole and we assume that the variables Γ available in the completion context of the hole can be obtained using $\mathsf{vars}(E[?_\tau])$.

To model a choose-your-own-adventure interaction akin to Idris, the state of the system would be the term e. The choices operation synthesises possible completions using $\Rightarrow$ and offers the resulting terms as possible completions. The identifiers ι could be based either on the tactic name (rule name) or show a preview of the resulting term:

$$\begin{aligned} \mathsf{choices}(E[?_\tau]) &= \{\ \iota \mapsto E[e] \mid \mathsf{vars}(E[?_\tau]) \vdash \tau \Rightarrow e\} \\ \mathsf{choose}(e) &= e \end{aligned}$$

The choices operation synthesises possible terms, while choose simply returns the term. To avoid offering an infinite number of choices, the system could favour derivations where all sub-expressions are completed using the (syn-hole) rule.

The expression completion mode of the system behaves similarly to Idris. It constructs the term, but does not record the completion choices. A system akin to Rocq could be modelled by using a sequence of tactics as the state; choices would append the available tactics to the end of the current sequence.

Programming by Navigation. The programming by navigation framework [28] provides another way of looking at the problem of program synthesis. In the framework, program synthesis is described as a sequence of steps σ that transform expressions containing named holes $?_h$. The relation $e_1 \to^\sigma e_2$ holds if a step σ transforms an expression e_1 into e_2. A notion of validity, e valid determines expressions that are well-typed, satisfy the specified input-output pairs, or are valid in some other sense. The synthesis process is controlled by a step provider $\mathbb{S}$, which maps expressions to a set of possible steps, i.e., $\mathbb{S}(e) = \{\sigma_1, \ldots, \sigma_n\}$.

Any programming by navigation system can also implement the choose-your-own-adventure interaction pattern. As in the aforementioned program synthesis example, system states are expressions e. The operations are defined as:

$$\begin{aligned} \mathsf{choices}(e) &= \{\ \iota \mapsto \sigma(e) \mid \forall \sigma \in \mathbb{S}(e),\ \sigma(e)\ \mathsf{valid}\} \\ \mathsf{choose}(e) &= e \end{aligned}$$

Here, the choices operation offers all valid expressions that can be reached by one step from the current expression and the choose operation returns the completed expression. The programming by navigation framework is designed around two properties, *strong soundness* and *strong completeness*, which correspond directly to our definitions of *eventual correctness* and *completeness* discussed in Section 5.

4.5 Structure Editors

Structure editors illustrate further design considerations for choose-your-own-adventure systems. We contrast a system derived from a context-free grammar, inspired by Sandblocks [7], with a system that extends grammar-directed construction with a support for refactorings, akin to Deuce [21].

Formal model. We first model a system that generates choices based on a context-free grammar akin to Sandblocks. We assume the grammar $(\mathcal{N}, \Sigma, \mathcal{P}, S)$ consists of a set of non-terminal symbols $\mathcal{N}$, terminal symbols Σ, start symbol $S \in \mathcal{N}$ and production rules $\mathcal{P}$ of the form $n \mapsto \boldsymbol{s}$ where $n \in \mathcal{N}$ and $\boldsymbol{s} \in (\mathcal{N} \cup \Sigma)^*$ (we write n for non-terminals, t for terminals and s for all symbols; bold indicates a sequence of symbols).

When interacting with the choose-your-own-adventure system, the user is gradually constructing a program (sentence) of a form $\boldsymbol{s} \in (\mathcal{N} \cup \Sigma)^*$. The system state is the program (sentence) constructed so far, i.e., $\sigma = \boldsymbol{s}$. The process can continue as long as there are non-terminal symbols in the sentence and production rules applicable to them.

In general, choices could recommend applicable production rules for any non-terminal in the sentence. The following always rewrites the first non-terminal by assuming a sentence of a form $\boldsymbol{tns}$ (the need to sequentialise the construction is a limitation discussed in Section 7); the choices operation then finds all applicable production rules and offers them as choices. The identifiers ι_i would be formed by the names of the production rules:

$$\begin{aligned} \mathsf{choices}(\boldsymbol{tns}) &= \{\ \iota_i \mapsto \boldsymbol{ts_i s} \mid \forall (n \mapsto \boldsymbol{s_i}) \in \mathcal{P}\ \} \\ \mathsf{choose}(\boldsymbol{s}) &= \boldsymbol{s} \end{aligned}$$

Unlike in data exploration, the choose operation does not ensure that a program is fully constructed and the returned sentence may contain further non-terminals.

Program Refactoring. The definition based on a context-free grammar models structure editors such as Sandblocks [7], but many structure editors offer more advanced operations for program construction. For example, Deuce [21] makes

Stepwise construction based on grammar

1) Add function as the first *stmt*, fill the two *ident* leaves (name and parameters).

```
function sayHello(who) {
  stmt*
}
stmt*
```

2) Sequentially fill both holes with invocations.

```
function sayHello(who) {
  print("Hello " + who)
}
sayHello("world")
```

Stepwise construction using refactorings

1) Start with a small working program.

```
print("Hello world")
```

2) Extract literal into a variable.

```
let who = "world"
print("Hello " + who)
```

3) Extract function taking who as a parameter.

```
function sayHello(who) {
  print("Hello " + who)
}
sayHello("world")
```

Figure 7. Two approaches to program construction.

it possible to transform a program through a range of refactorings such as extract function, introduce a local variable, inline definition, or reorder arguments.

Figure 7 contrasts two ways of program construction. It shows some of the steps necessary to construct a simple program via the grammar rules and a more appealing alternative where the programmer starts with an executable program and then refines it through a sequence of refactorings.

Modelling refactorings formally requires a more expressive framework [53], so we omit the details. The choices operation would need to identify applicable refactorings for the current program and offer them to the user.

Note that a system based on an unambiguous context-free grammar is reconstructible. Although the constructed expression does not directly encode the sequence of operations, it is possible to infer which production rules have been applied. A choose-your-own-adventure structure editor that supports refactorings no longer has this property, because it may offer multiple ways of constructing the same program.

5 Properties

In many programming systems, there is a system-specific notion of validity or correctness that identifies some expressions from the set of all possible expressions as valid or correct. This may be the set of well-typed expressions, or expressions that do not contain further holes.

Using the choose-your-own-adventure calculus, we can now precisely capture several properties that are desirable for systems with such a notion. Two properties that exist in The Gamma [43] and in programming by navigation [28] are *correctness* and *completeness*. In systems satisfying the two properties, all programs constructed using the pattern are valid and, conversely, all valid programs can be constructed using the interaction pattern. In this section, we make these notions precise and also include a notion of *uniqueness*.

5.1 Correctness

To define correctness and completeness, we need a system-specific notion of validity [28], distinguishing between correct and incorrect expressions. We write $\mathcal{E} \subseteq E$ for the subset of correct expressions.

For systems based on statically typed programming languages, $\mathcal{E}$ may be a set of all well-typed expressions. For some systems, we may additionally want the set of correct expressions $\mathcal{E}$ to be hole-free, i.e., only programs that can run (or represent complete proofs) are correct. For systems where the completion is based on a grammar, we may require that correct programs do not contain non-terminals.

Definition 4 (Correctness). *Assume that $\mathcal{E} \subseteq E$ is a subset of correct expressions. A choose-your-own-adventure system is correct with respect to $\mathcal{E}$ if and only if:*

- $\forall \sigma_1, .., \sigma_n$ *and* $\iota_1, .., \iota_n$ *such that* $\iota_i \mapsto \sigma_i \in \mathsf{choices}(\sigma_{i-1})$ *it is the case that* $\mathsf{choose}(\sigma_i) \in \mathcal{E}$.

The definition states that, if we make any sequence of choices that start from an initial state σ_0 and result in intermediate states $\sigma_1, \ldots, \sigma_n$ then all the programs we could generate from any of the intermediate states are correct.

The property depends on how we define correct expressions $\mathcal{E}$. Trivially, all systems are correct with respect to $\mathcal{E} = E$. Three of the systems discussed in Section 4 are also correct with respect to a non-trivial set $\mathcal{E}$.

Data Exploration. For the system discussed in Section 4.1, correct expressions are those where the parameters of all operations are fully specified. The system is correct, but only because the choose operation drops the last operation if it is not fully specified. If choose returned all operations, including the partially constructed (but not yet completed) operation, the system would not have the property.

Type Providers. Correct expressions for type providers (Section 4.2) are those that are well-typed. The system is correct because the choices operation offers available members based on the type information. This reasoning also applies to the type provider behind The Gamma. In The Gamma, the generated members collect operation parameters and only invoke the operation once all parameters are known.

AI Assistants. The correctness of an AI assistant (Section 4.3) depends on the expressions that may be returned by the optimisation algorithm (arg max) from the set of all possible scripts E_C. If the algorithm can return any $e \in E_C$, then the system is correct if and only if $E_C \subseteq \mathcal{E}$ for all C.

Program Synthesis. For an interactive system based on type-directed synthesis (Section 4.4), correct expressions are those that are well-typed. The system is correct if the synthesis rules are sound [39], i.e., if $\Gamma \vdash \tau \Rightarrow e$ then $\Gamma \vdash e : \tau$. Correctness of a choose-your-own-adventure system based on programming by navigation [28] follows from strong correctness.

Structure Editors. For a structure editor based on a context-free grammar (Section 4.5), we may desire to obtain only fully complete completions and define $\mathcal{E}$ as the set of sentences that do not contain non-terminals. In such a case, a system based on stepwise construction is not correct, but a correct structure editor can be implemented if it is based solely on refactorings such as those in Figure 7.

There may be several useful systems that violate the correctness property. A heuristic completion tool, for example for a dynamically typed programming language, may generate code with errors that the programmer can later correct.

5.2 Eventual Correctness

The data exploration system discussed in Section 4.1 ensures correctness by dropping the last not fully specified operation in the choose operation. As a result, it is correct, but it does not support reconstructible expression completion. If the operation is dropped, we cannot reconstruct the sequence of interactions from the source code. Generating code that includes the not fully specified operation allows reconstructibility, but makes the system incorrect.

For such systems, it is useful to define weaker eventual correctness, modelling systems where a sequence of choices can always be completed to reach a correct program:

Definition 5 (Eventual correctness)**.** *Assume that $\mathcal{E} \subseteq E$ is a subset of correct expressions, a choose-your-own-adventure system is eventually correct with respect to $\mathcal{E}$ if:*

- *For any sequence $\sigma_1, \ldots, \sigma_k$ and $\iota_1, \ldots, \iota_k$ such that $\forall i \in 1 \ldots k \,.\, \iota_i \mapsto \sigma_i \in \mathsf{choices}(\sigma_{i-1})$ there exists an extension $\sigma_{k+1}, \ldots, \sigma_n$ and $\iota_{k+1}, \ldots, \iota_n$ such that $\mathsf{choose}(\sigma_n) \in \mathcal{E}$ and $\forall i \in k+1 \ldots n \,.\, \iota_i \mapsto \sigma_i \in \mathsf{choices}(\sigma_{i-1})$.*

Eventually correct systems include grammar-based structure editors (where correct expressions are not allowed to include non-terminals), as well as a data exploration system (that does not drop incomplete operations). The data exploration system suggests a more general construction.

Example 1. *Any eventually correct system can be turned into a correct one by remembering the last state for which the* choose *operation returned a correct program and using it in* choose *until the next correct state is reached. This makes any eventually correct choose-your-own-adventure system correct, but it breaks reconstructibility of expression completion.*

Example 2. *Alternatively, we can construct a system that collapses all sequences of temporarily invalid states $\sigma_1, \ldots, \sigma_n$ identified by $\iota_1, \ldots, \iota_n$ where $\forall i \in 1 \ldots n-1 \,.\, \mathsf{choose}(\sigma_i) \notin \mathcal{E}$ and $\mathsf{choose}(\sigma_n) \in \mathcal{E}$ into a single option $\iota' \mapsto \sigma_n$ where ι' is produced by concatenating identifiers $\iota_1, \ldots, \iota_n$. This makes the system correct and preserves reconstructibility, but it may generate too many choices that are difficult to navigate.*

5.3 Completeness

Another question about an implementation of the choose-your-own-adventure pattern is whether users can use it to construct all possible programs. For example, in The Gamma, the user can construct all possible chains of member accesses just by repeatedly choosing options from the offered lists. A system with the property is easier to use, because it follows the *recognition over recall* design heuristic [35].

Definition 6 (Completeness)**.** *Assume that $\mathcal{E} \subseteq E$ is a subset of correct expressions. A choose-your-own-adventure system is complete with respect to $\mathcal{E}$ if and only if:*

- *$\forall e \in \mathcal{E} \,.\, \exists \sigma_1, \ldots, \sigma_n$ and $\iota_1, \ldots, \iota_n$ such that $\iota_i \mapsto \sigma_i \in \mathsf{choices}(\sigma_{i-1})$ and $e = \mathsf{choose}(\sigma_n)$.*

A system is complete if, for any correct program, there is a sequence of choices that can be used to construct the program. This is a more subtle property than correctness.

Data Exploration. The system described in Section 4.1 would be complete only if the underlying query language had a fixed set of operations, a fixed set of aggregations (rather than letting users write their own) and a fixed set of values for each parameter. A system for a general-purpose query language, such as SQL, would be incomplete.

Type Providers. The mechanism for type providers (Section 4.2) is complete, because it offers all available members of the type. Consequently, the type provider in The Gamma is also complete. Even if the underlying query language is more expressive, it is hidden from the user and the system offers all available members.

AI Assistants. AI assistants (Section 4.3) offer a set of constraints based on the current inferred program. Moreover, because the arg max operation used in choose performs statistical optimisation, there is no guarantee that it can be used to generate a specific program. The system is complete for some AI assistants, including datadiff, where it is possible to choose a constraint set C that restrict the set of programs E_C to a single given program.

Program Synthesis. A type-directed synthesis system (Section 4.4), or an interactive theorem prover could offer all possible ways of filling a hole with an expression containing further holes as sub-expressions. This would make the system complete (up to renaming of variables), but the large number of generated options would be impractical. In a programming by navigation system [28], the property corresponds to strong completeness.

Structure Editors. Completeness holds for systems based on context-free grammars (Section 4.5). For each sentence that can be produced by the rules of the grammar, there is a sequence of choices that applies the necessary rules. Editors based on refactorings may not have the property as some programs may not be reachable just through refactorings.

Correctness and completeness are desirable properties of a choose-your-own-adventure system. Unlike, for example, type soundness, they are not strictly necessary in practice and are best seen as design trade-offs that designers should consider.

5.4 Uniqueness

In the grammar-based structure editor example, the generated program does not have a sequential structure, but it was still possible to reconstruct the interaction steps from the resulting expression. This is no longer possible if the program can be transformed through a richer set of refactorings.

The difference between the two examples is captured by the uniqueness property, which characterises systems where each expression can be reached by a unique path of choices:

Definition 7 (Uniqueness)**.** *A choose-your-own-adventure system has the uniqueness property if and only if, for all expressions $e \in E$ it is the case that:*

- *if there are $\sigma_1, .., \sigma_n$ and $\iota_1, .., \iota_n$ such that $\iota_i \mapsto \sigma_i \in \text{choices}(\sigma_{i-1})$ and $e = \text{choose}(\sigma_n)$,*
- *and there are $\sigma'_1, .., \sigma'_m$ and $\iota'_1, .., \iota'_m$ such that $\iota'_i \mapsto \sigma'_i \in \text{choices}(\sigma'_{i-1})$ and $e = \text{choose}(\sigma'_m)$,*
- *then $n = m$ and $\forall i \in 1 \ldots n \,.\, \iota_i = \iota'_i \wedge \sigma_i = \sigma'_i$.*

The property holds for the data exploration environment (Section 4.1) and type providers (Section 4.2) where the program directly mirrors the interactive steps. It is not the case for AI assistants (Section 4.3), where adding an irrelevant constraint may have no effect on the resulting program.

A program synthesis system or an interactive theorem prover (Section 4.4) would have the property if no two tactics produced the same term. This is the case for our small example, but it may not hold in general. Finally, a structure editor based on an unambiguous context-free grammar (Section 4.5) satisfies uniqueness, but a structure editor that allows further refactorings does not.

In theory, uniqueness guarantees that we can define the decode operation required by reconstructible expression completion. However, uniqueness does not guarantee that there is a computationally effective way to do so.

To decode an expression efficiently, we need to be able to start from the initial state and identify the correct choice at each step. This is possible, for example, for a structure editor based on a context-free grammar, but not in general. As a counter-example, consider a system that offers a large tree of choices and produces an expression that is a hash of the choices made. For such system, there is only one path to each hash, but reconstructing it requires searching the entire tree. To guarantee a computationally effective reconstructibility, we need more than uniqueness—in particular, it needs to be possible to uniquely determine what part of the constructed program corresponds to which of the choices from the sequence.

theorem sum-z-rh: **forall** n **exists** $n + (\mathsf{z}) = n$.
theorem sum-s-rh: **forall** $d_1 : n_1 + n_2 = n_3$ **exists** $n_1 + (\mathsf{s}\ n_2) = (\mathsf{s}\ n_3)$.

theorem sum-commutes: **forall** $d_1 : n_1 + n_2 = n_3$ **exists** $n_2 + n_1 = n_3$
  $d_2 : n_2 + n_1 = n_3$ **by induction on** d_1:
  case rule

$$\frac{}{dzc : (\mathsf{z}) + n = n}\ \text{sum-z}$$

  is
    $dz_1 : n + (\mathsf{z}) = n$ **by theorem** sum-z-rh **on** n
  end case
  case rule

$$\frac{dsp : n'_1 + n_2 = n'_3}{dsc : (\mathsf{s}\ n'_1) + n_2 = (\mathsf{s}\ n'_3)}\ \text{sum-s}$$

  is
    $ds_1 : n_2 + n'_1 = n'_3$ **by induction hypothesis on** dsp
    $ds_2 : n_2 + (\mathsf{s} n'_1) = (\mathsf{s} n'_3)$ **by theorem** sum-s-rh **on** ds_1
  end case
 end induction
end theorem

Figure 8. A proof of commutativity of + in Peano arithmetic constructed using mixed-initiative interaction. Parts generated by the system are highlighted in grey. After the user writes a theorem and specifies induction, the system completes the first case. The user then specifies how to apply the induction hypothesis and the system completes the proof.

6 Applications

The choose-your-own-adventure calculus lets us compare properties of related interactive programming systems, reuse components in system implementations, but also transfer ideas across different domains. We now discuss three ideas that emerged in the context of one specific interactive system, but could be applied to other systems based on the pattern.

A timely application that we discuss in more detail is using LLM-based assistants to recommend choices to the user. We describe and evaluate an implementation of a recommendation system for The Gamma, but note that the implementation relies only on the structure shared with other choose-your-own-adventure systems.

6.1 Mixed-initiative Interaction

When using a conventional interactive theorem prover, the user typically constructs the proof manually until an automated strategy can fill in the remaining gaps. This is the case with Idris proof search, as well as Rocq `auto` tactics.

Richer ways of interacting exist [27], but are less common. A recently proposed mixed-initiative theorem prover [59] supports a more collaborative way of working where the system completes some steps automatically, but defers to the user when it gets stuck.

Mixed-initiative Theorem Proving. Figure 8 shows a proof of commutativity of + in Peano arithmetic. The listing follows the SASyLF notation [2, 3] and is simplified for

brevity. After writing proofs of sum-z-rh and sum-s-rh (not shown), the user states sum-commutes and specifies the structure of the induction. They then invoke automatic proof search, which completes the first case, but gets stuck in the second case [59]. The user then specifies how to apply the induction hypothesis and the system completes the proof.

The interaction can be recast from the perspective of the choose-your-own-adventure system discussed in Section 4.4. An interactive theorem prover generates possible completions using available tactics and offers them to the user, who chooses a tactic and applies it to transform the proof. In the automatic mode, the system recursively searches through the available choices. If it finds one that results in a complete proof, it stops. Otherwise, it completes a number of steps (determined by some heuristic) and returns the control to the user who chooses the next step and completes the proof manually or invokes the automatic search again.

Generalised Mixed-initiative Interaction. The mixed-initiative mode of interaction combines manual interaction with automatic search. The same approach could be used in other interactive programming systems.

For example, a system for type-directed program synthesis could automatically synthesise parts of a program (e.g., pattern matching and implementation for some of the cases), but ask the user for input in order to complete branches where a solution was not found automatically. In data exploration, a system could perform automatic search through the available operations in order to transform data into a more regular format, according to a suitability metric as done, for example, in Proactive Wrangling [20].

We can model mixed-initiative interactive systems as extensions built on top of the choose-your-own-adventure calculus by requiring an additional operation suggest that recommends a sequence of choices for a given state:

Definition 8 (Mixed-initiative system)**.** *A choose-your-own-adventure system supports a mixed-initiative mode of interaction if it is equipped with an operation* suggest *such that:*

- $\mathsf{suggest}(\sigma_0) = (\iota_1, \ldots, \iota_n)$ *such that* $\forall i \in 1 \ldots n \,.\, \iota_i \mapsto \sigma_i \in \mathsf{choices}(\sigma_{i-1})$.

The definition only requires that the suggested sequence of choices is valid, but it does not specify how exactly the system should make the recommendations. In practice, the suggestion should improve the quality of the program so that $\mathsf{choose}(\sigma_n)$ is better than $\mathsf{choose}(\sigma_0)$, but we leave the definition flexible to accommodate a broader range of uses. An interactive theorem prover may try to find a proof (a correct program with no holes) or solve sub-goals (minimise the number of holes). A data wrangling system may try to improve the structure of data (using a suitability metric).

6.2 Language Model-based Completion

LLM-based assistants can generate data exploration code based on natural language prompts [61]. A recognised drawback is that the user may gain only cursory understanding of the code and overlook errors. Various systems address this by generating explanations alongside code [36].

We developed a system for The Gamma, inspired by an earlier prototype [17], that assists the user in making choices when using the type provider. In contrast to work on filling holes in Hazel [9], our LLM-based completion merely recommends choices to the user. Although our implementation is specific to The Gamma, we discuss the application in detail because it shows that the choose-your-own-adventure calculus can enable reuse of components of programming systems. The implementation of our assistant, as well as its evaluation, rely only on the common structure exposed by choose-your-own-adventure systems.

LLM Assistant for The Gamma. Our assistant controls The Gamma type provider using a large language model. It lets the user ask a natural language query and then recommends choices that the user should make in order to answer the query. We do not use the LLM to generate code, but to recommend an option offered by the type provider by issuing a prompt constructed as follows:

> *You are helping the user to complete a task in an interactive programming environment. The user's query is: [query]. The query built so far is: [path]. The environment offers the user possible options. Choose an option that should be applied next:*
>
> *[options]*
>
> *Answer with the number of the option and no further explanation.*

Here, the [query] is the natural language query, [path] is the sequence of identifiers selected so far and [options] is a numbered list of choices returned by the choices operation. Our system pre-selects the recommended item in the completion list offered to the user. In contrast to one-shot generation of Python or SQL code snippets, our approach guides the user through the program construction, providing them with an opportunity to review and check whether the program logic matches their expectations.

To evaluate the system, we use code snippets shared in The Gamma gallery (https://gallery.thegamma.net). The dataset consists of 61 code snippets, containing 75 method chains (a snippet comparing two datasets can use two distinct method chains) across type providers for exploring data cubes, tabular data and graph databases [43]. For each method chain, we constructed a natural language prompt from the snippet title and description posted by the user and compute a ratio of correct recommendations for individual choices in the sequence of operations, i.e., matching the choice in the hand-written snippet. This models a realistic scenario where users write title and description first, but do not necessarily aim to fully describe the task.

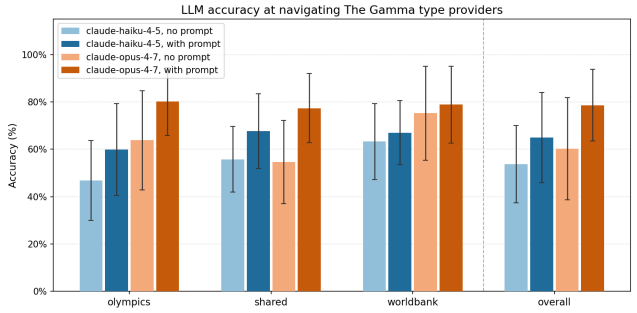


Figure 9. Accuracy of LLM recommendations for three different type providers, using four different configurations.

Figure 9 shows the result of the evaluation, comparing Claude Haiku 4.5 and Opus 4.7 models. The accuracy can further be improved by providing the LLM with a system prompt that explains basic structural patterns in the specific type providers (the '[any]' wildcard for graph databases; the 'then' and 'get series' operations for tabular data).

The results suggest that, when using a more advanced LLM system, the ratio of correct recommendations is high enough to be practically useful. Arguably, in many contexts, a mode of interaction where the user is offered a recommendation is preferable to providing an opaque one-shot solution, because it keeps the human in the loop [50] and avoids "ironies of automation" including deskilling [6].

Generalised LLM Assistant. An LLM-based recommendation system built for The Gamma can be developed for any choose-your-own-adventure system. The information needed to construct an LLM prompt comprises only the natural language query entered by the user, a sequence of previously made choices $\iota_1, \ldots, \iota_{n-1}$ and the identifiers $\iota_n, \iota'_n, \iota''_n, \ldots$ of the choices offered by the choices operation.

As the LLM-based recommendations are based on natural language analysis of the prompt, the quality of the recommendations depends on how semantically meaningful the generated identifiers are. Such a system could be useful for multiple systems discussed in this paper.

Data Exploration. We demonstrated the usefulness of the system for The Gamma. Type providers for structured data [45] typically generate a small number of choices that the user can navigate without assistance, but navigating the schemas generated by type providers for semantic knowledge bases [55] may be assisted by a natural language query.

AI assistants. The datadiff assistant (Section 2) matches columns based on statistical distribution and ignores column names. An LLM-based recommendation engine is useful in this case, because it is able to recommend the option Don't match 'Urban.rural' and 'Nation' based solely on its name. For other AI assistants, the LLM-based recommendations would need access to the input data.

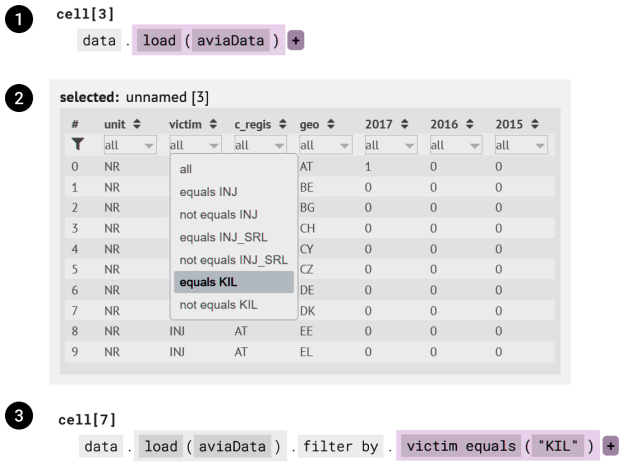


Figure 10. Programming by demonstration in Histogram. (1) the user constructs program to load data, (2) then they filter data using a graphical interface, which (3) records an operation corresponding to the interaction in code.

Theorem Proving. The problem of predicting steps in a proof is known as proofstep generation in the literature focused on deep learning for theorem proving [26]. Although most systems use models trained specifically for the task, there is also interest in using general-purpose LLMs [62]. Our approach suggests a prompting strategy.

Notably, the system sketched here combines a symbolic component (used for generating possible choices and checking their correctness) with an AI-based component (used for making choices). This is the same architecture that has been used by AlphaGeometry [58] for solving geometry problems.

6.3 Direct Manipulation

The Histogram data exploration environment [41] provides code completion that can be used both to choose operations (as with type providers) and to specify their parameters (by offering values available in the context). Histogram makes it possible to evaluate sub-expressions at a fine-grained level, refines type information based on runtime values, but it also supports programming by demonstration [15, 24].

As illustrated in Figure 10, when the user loads data, they can manipulate it in a table through a direct manipulation interface [51]. In Histogram, interacting with the table triggers a sequence of operations that construct code. A similar functionality is available in The Gamma, where data transformations can be selected not only through auto-completion, but also by interacting with the live preview.

Interactive Programming. The programming by demonstration interaction can be explained in terms of the choose-your-own-adventure calculus. When interacting with a live preview, the user is using a graphical interface to choose an

option offered by choices. Interacting with the interface appends an item to the sequence of choices, updates the system state and generates a new preview.

An interesting question is whether a program can be constructed solely through direct manipulation, by interacting with the live previews. For this to be possible, the live preview needs to offer links to all the options offered by choices. In Histogram (Figure 10), the live preview offers interactive elements for filtering based on equality test (row of drop-downs), sorting (up/down arrows) and indexing (indices in the "#" column), but some operations (such as aggregation) can only be constructed through code. Previews in The Gamma (Figure 2) provide interactive elements for all options, but they are not integrated with the data display and, in some situations, can only be accessed via a menu triggered using the "+" button shown on the right.

Generalised Direct Manipulation. We can use the choose-your-own-adventure calculus to make the question whether all programs can be constructed through direct manipulation more precise. Assume $\mathsf{preview}(e) = p$ is a preview constructed for an expression e and $\mathsf{links}(p) = \iota_1, \ldots, \iota_n$ are identifiers that can be invoked through the preview (akin to links in a hypertext document).

Definition 9 (Direct manipulation)**.** *A choose-your-own-adventure system with previews defined by* preview *and* links *supports full direct manipulation if:*

- $\forall \sigma, e$ *such that* $\mathsf{choose}(\sigma) = e$ *it is the case that* $\forall \iota' \mapsto \sigma' \in \mathsf{choices}(\sigma) \,.\, \iota' \in \mathsf{links}(\mathsf{preview}(e))$.

A system supports full direct manipulation if any program can be constructed by interacting with the live previews, created based on the gradually constructed programs. As illustrated by The Gamma, this can always be achieved by listing the options in a menu, but it is more interesting to see if the links can be directly embedded in the preview as in the data table interface of Histogram.

The idea of directly mapping elements in a user interface to underlying structure of the programming system has previously been implemented in pioneering user interface systems including the Alternate Reality Kit [52] and the Morphic framework for Self and Squeak [30, 31]. In those, user interface interactions directly map to messages sent to the underlying object and the halo element in Morphic plays a similar role to the additional menu in The Gamma. The difference is that those systems modify the state of a running programming system [23], rather than the state of a system that constructs static code.

7 Limitations

The study of programming systems is less well-developed than the study of programming languages. Our work is a step towards remedying the situation. As with all formal models, our system ignores a number of practical concerns.

Revisiting Earlier Choices. Programmers often revise their programs. For example, a data scientist may want to add further aggregation to a grouping or change the sorting key. The choose-your-own-adventure calculus does not model how such modifications are done. Modifying an earlier choice poses an interesting problem—if an earlier choice is changed, we may try to replay the remaining choices based on the identifiers ι, but there is no guarantee that they will be offered again and that this will yield the desired result.

Sequentialisation of Completion. In a number of examples, an expression contained multiple holes or non-terminals that can all be completed. We generally assume that they can be completed sequentially, but this may be an inconvenient limitation. It may be desirable to define a variant of expression completion where multiple independent recommendations are available for different parts of the program.

Searching Through Options. In some systems, the number of options to choose from may be large or even infinite. Similarly, systems may generate a very long chain of a small number of options. In those cases, choosing options from a menu may be inconvenient—a possible alternative interface would let the user search the tree of options.

Richer User Interfaces. The choose-your-own-adventure calculus models a simple interaction based on menus. Although we point out that it can be used as the basis for, e.g., direct manipulation, it may be interesting to look at richer interfaces. In particular, a number of systems are centred on entities that can be selected and modified through commands or interactions. This includes Morphic's halos [30, 31], the ARK [52], as well as text-based systems [12].

8 Conclusions

Working in many interactive programming environments has the same feel as following a choose-your-own-adventure book. You repeatedly choose one of the offered options to construct a program, proof or to explore data. The aim of this paper is to formally capture this kind of interaction. As with formal models of programming languages, our model focuses on the minimal essence of the interaction pattern. Yet, this is sufficient to recognise similarities across a broad range of systems, talk about key properties that they may have, transfer knowledge across multiple domains, as well as to suggest a more general way of building richer programming experiences ranging from AI assistants and mixed-initiative interaction to programming by demonstration.

Acknowledgments

We are grateful our many collaborators who worked on the systems analysed in this paper including Don Syme, Phil Trelford and colleagues from the AI for Data Analytics project at The Alan Turing Institute. We thank Vít Šefl and anonymous reviewers for invaluable feedback on the paper.

The work has been supported by the Charles University grant PRIMUS/24/SCI/021 and by the Czech Ministry of Education, Youth and Sports grant ERC-CZ LL2325.

References

[1] Michael D. Adams, Eric Griffis, Thomas J. Porter, Sundara Vishnu Satish, Eric Zhao, and Cyrus Omar. 2025. Grove: A Bidirectionally Typed Collaborative Structure Editor Calculus. *Proc. ACM Program. Lang.* 9, POPL, Article 73 (Jan. 2025), 29 pages. doi:10.1145/3704909

[2] Jonathan Aldrich and John Boyland. 2025. Formalization of integers, addition, and a proof that addition commutes (SASyLF example). https://github.com/boyland/sasylf/blob/master/examples/sum.slf Accessed: 2025-03-06.

[3] Jonathan Aldrich, Robert J. Simmons, and Key Shin. 2008. SASyLF: an educational proof assistant for language theory. In *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education (FDPE '08)*. Association for Computing Machinery, 31–40. doi:10.1145/1411260.1411266

[4] Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. 1994. *A user's guide to ALF*. Technical Report. Chalmers University of Technology, Sweden. https://people.cs.nott.ac.uk/psztxa/publ/alf94.pdf Unpublished Draft.

[5] David Aspinall. 2000. Proof General: A Generic Tool for Proof Development. In *Tools and Algorithms for the Construction and Analysis of Systems*, Susanne Graf and Michael Schwartzbach (Eds.). Springer. doi:10.1007/3-540-46419-0_3

[6] Lisanne Bainbridge. 1983. Ironies of Automation. In *Analysis, Design and Evaluation of Man–Machine Systems*, G. Johannsen and J.E. Rijnsdorp (Eds.). Pergamon, 129–135. doi:10.1016/B978-0-08-029348-6.50026-9

[7] Tom Beckmann, Patrick Rein, Stefan Ramson, Joana Bergsiek, and Robert Hirschfeld. 2023. Structured Editing for All: Deriving Usable Structured Editors from Grammars. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. ACM, Article 595, 16 pages. doi:10.1145/3544548.3580785

[8] Alan F. Blackwell and Thomas R. G. Green. 2003. Notational Systems – The Cognitive Dimensions of Notations Framework. In *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*, John M. Carroll (Ed.). Morgan Kaufmann, San Francisco, 103–134.

[9] Andrew Blinn, Xiang Li, June Hyung Kim, and Cyrus Omar. 2024. Statically Contextualizing Large Language Models with Typed Holes. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 288 (Oct. 2024), 31 pages. doi:10.1145/3689728

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

[11] Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming (ECOOP 2021) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 9:1–9:26. doi:10.4230/LIPIcs.ECOOP.2021.9

[12] Omar Antolín Camarena. 2025. Embark: Emacs Mini-Buffer Actions Rooted in Keymaps. https://github.com/oantolin/embark Accessed: 2025-03-06.

[13] David Raymond Christiansen. 2013. Dependent type providers. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming* (Boston, Massachusetts, USA) *(WGP '13)*. Association for Computing Machinery, New York, NY, USA, 25–34. doi:10.1145/2502488.2502495

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

[15] A. Cypher and D.C. Halbert. 1993. *Watch what I Do: Programming by Demonstration*. MIT Press.

[16] Damian Frölich and L. Thomas van Binsbergen. 2024. 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* (Pasadena, CA, USA) *(GPCE '24)*. Association for Computing Machinery, NY, USA, 107–120. doi:10.1145/3689484.3690734

[17] Mikoláš Fromm. 2024. Design of LLM Prompts for Iterative Data Exploration. Bachelor's thesis, Charles University.

[18] Richard P. Gabriel. 2012. 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)*. ACM. doi:10.1145/2384592.2384611

[19] Thomas R. G. Green. 1989. Cognitive Dimensions of Notations. In *People and Computers V: Proceedings of the Fifth Conference of the British Computer Society*, A. Sutcliffe and L. Macaulay (Eds.). Cambridge University Press, Cambridge, UK, 443–460.

[20] Philip J. Guo, Sean Kandel, Joseph M. Hellerstein, and Jeffrey Heer. 2011. Proactive wrangling: mixed-initiative end-user programming of data transformation scripts. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, 65–74. doi:10.1145/2047196.2047205

[21] Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. 2018. Deuce: a lightweight user interface for structured editing. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 654–664. doi:10.1145/3180155.3180165

[22] Jack Hughes and Dominic Orchard. 2024. Program Synthesis from Graded Types. In *Programming Languages and Systems*, Stephanie Weirich (Ed.). Springer Nature Switzerland, Cham, 83–112. doi:10.1007/978-3-031-57262-3_4

[23] Joel Jakubovic, J. Edwards, and T. Petricek. 2023. Technical Dimensions of Programming Systems. *Art Sci. Eng. Program.* 7, 3 (2023). doi:10.22152/PROGRAMMING-JOURNAL.ORG/2023/7/13

[24] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. ACM, 3363–3372. doi:10.1145/1978942.1979444

[25] Tristan Knoth. 2023. *Type-Directed Program Synthesis*. Ph. D. Dissertation. University of California, San Diego. https://escholarship.org/uc/item/4g11m7rq

[26] Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. 2024. A Survey on Deep Learning for Theorem Proving. In *First Conference on Language Modeling*.

[27] Helen Lowe and David Duncan. 1997. XBarnacle: Making theorem provers more accessible. In *Automated Deduction—CADE-14*, William McCune (Ed.). Springer Berlin Heidelberg, 404–407.

[28] Justin Lubin, Parker Ziegler, and Sarah E. Chasins. 2025. Programming by Navigation. *Proc. ACM Program. Lang.* 9, PLDI, Article 165 (June 2025), 28 pages. doi:10.1145/3729264

[29] Lena Magnusson and Bengt Nordström. 1994. The Alf proof editor and its proof engine. In *Types for Proofs and Programs*, Henk Barendregt and Tobias Nipkow (Eds.). Springer Berlin Heidelberg, 213–237.

[30] John Maloney. 2001. An Introduction to Morphic: The Squeak User Interface Framework. In *Squeak: Open Personal Computing and Multimedia*, Mark Guzdial and Kim Rose (Eds.). Prentice Hall.

[31] John H. Maloney and Randall B. Smith. 1995. Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology* (Pittsburgh, Pennsylvania, USA) *(UIST '95)*. Association for Computing Machinery, New York, NY, USA, 21–28. doi:10.1145/215585.215636

[32] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional evaluation with direct manipulation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 127 (Oct. 2018), 28 pages. doi:10.1145/3276497

[33] Conor McBride. 1999. *Dependently Typed Functional Programs and their Proofs*. Ph. D. Dissertation. University of Edinburgh. https://era.ed.ac.uk/handle/1842/374

[34] David Moon, Andrew Blinn, and Cyrus Omar. 2022. tylr: a tiny tile-based structure editor. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development* (Ljubljana, Slovenia) *(TyDe 2022)*. ACM, 28–37. doi:10.1145/3546196.3550164

[35] Jakob Nielsen. 1994. *Usability engineering*. Morgan Kaufmann.

[36] Farhad Nooralahzadeh, Yi Zhang, Jonathan Furst, and Kurt Stockinger. 2024. Explainable Multi-Modal Data Exploration in Natural Language via LLM Agent. arXiv:2412.18428 https://arxiv.org/abs/2412.18428

[37] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling typed holes with live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 511–525. doi:10.1145/3453483.3454059

[38] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live functional programming with typed holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (Jan. 2019), 32 pages. doi:10.1145/3290327

[39] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, 619–630. doi:10.1145/2737924.2738007

[40] Tomas Petricek. 2017. Data Exploration through Dot-driven Development. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21:1–21:27. doi:10.4230/LIPIcs.ECOOP.2017.21

[41] Tomas Petricek. 2019. Histogram: You have to know the past to understand the present. In *Workshop on Live Programming (LIVE 2019)*. Athens, Greece. https://tomasp.net/histogram Presented at the Workshop on Live Programming, Published online.

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

[43] Tomas Petricek. 2022. The Gamma: Programmatic Data Exploration for Non-programmers. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2022, Rome, Italy, September 12-16, 2022*, P. Bottoni, G. Costagliola, M. Brachman, and M. Minas (Eds.). IEEE, 1–7. doi:10.1109/VL/HCC53370.2022.9833134

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

[45] Tomas Petricek, Gustavo Guerra, and Don Syme. 2016. 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* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 477–490. doi:10.1145/2908080.2908115

[46] Tomas Petricek, Gerrit J. J. van den Burg, Alfredo Nazábal, Taha Ceritli, Ernesto Jiménez-Ruiz, and Christopher K. I. Williams. 2023. AI Assistants: A Framework for Semi-Automated Data Wrangling. *IEEE Trans. Knowl. Data Eng.* 35, 9 (2023), 9295–9306. doi:10.1109/TKDE.2022.3222538

[47] Clément Pit-Claudel and Pierre Courtieu. 2016. Company-Coq: Taking Proof General One Step Closer to a Real IDE. In *Second International Workshop on Coq for PL (CoqPL '16)*. https://dspace.mit.edu/handle/1721.1/101149.2

[48] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. ACM, 522–538. doi:10.1145/2908080.2908093

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

[50] Abigail Sellen and Eric Horvitz. 2024. The Rise of the AI Co-Pilot: Lessons for Design from Aviation and Beyond. *Commun. ACM* 67, 7 (July 2024), 18–23. doi:10.1145/3637865

[51] Ben Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *Computer* 16, 8 (1983), 57–69. doi:10.1109/MC.1983.1654471

[52] Randall B. Smith. 1986. Experiences with the alternate reality kit: an example of the tension between literalism and magic. In *Proceedings of the SIGCHI/GI Conference on Human Factors in Computing Systems and Graphics Interface* (Toronto, Ontario, Canada) *(CHI '87)*. Association for Computing Machinery, New York, NY, USA, 61–67. doi:10.1145/29933.30861

[53] Friedrich Steimann, Christian Kollee, and Jens von Pilgrim. 2011. A Refactoring Constraint Language and Its Application to Eiffel. In *ECOOP 2011 – Object-Oriented Programming*, Mira Mezini (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 255–280. doi:10.1007/978-3-642-22655-7_13

[54] Charles A. Sutton, Timothy Hobson, James Geddes, and Rich Caruana. 2018. 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*. ACM, 2279–2288. doi:10.1145/3219819.3220057

[55] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, Jack Hu, Tao Liu, Brian McNamara, Daniel Quirk, Matteo Taveggia, Wonseok Chae, Uladzimir Matsveyeu, and Tomas Petricek. 2012. *F# 3.0 - Strongly-Typed Language Support for Internet-Scale Information Sources*. Technical Report MSR-TR-2012-101. https://www.microsoft.com/en-us/research/publication/f3-0-strongly-typed-language-support-for-internet-scale-information-sources/

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

[57] Tim Teitelbaum and Thomas Reps. 1981. The Cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM* 24, 9 (Sept. 1981), 563–573. doi:10.1145/358746.358755

[58] Trieu H. Trinh, Yuhuai Wu, Quoc V. Le, He He, and Thang Luong. 2024. Solving olympiad geometry without human demonstrations. *Nature* 625, 7995 (01 Jan 2024), 476–482. doi:10.1038/s41586-023-06747-5

[59] Jan Liam Verter and Tomas Petricek. 2024. Don't Call Us, We'll Call You: Towards Mixed-Initiative Interactive Proof Assistants for Programming Language Theory. *CoRR* abs/2409.13872 (2024). arXiv:2409.13872 doi:10.48550/arXiv.2409.13872 Presented at the 5th International Workshop on Human Aspects of Types and Reasoning Assistants (HATRA 2024).

[60] Gregor Weber. 2017. Code is not just text: Why our code editors are inadequate tools. In *Companion Proceedings of the 1st International Conference on the Art, Science, and Engineering of Programming* (Brussels, Belgium) *(Programming '17)*. Association for Computing Machinery, New York, NY, USA, Article 35, 3 pages. doi:10.1145/3079368.3079415

[61] Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Oleksandr Polozov, and Charles Sutton. 2023. Natural Language to Code Generation in Interactive Data Science Notebooks. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 126–173. doi:10.18653/v1/2023.acl-long.9

[62] Shizhuo Dylan Zhang, Talia Ringer, and Emily First. 2023. Getting More out of Large Language Models for Proofs. *CoRR* abs/2305.04369 (2023). arXiv:2305.04369 [cs.FL] doi:10.48550/arXiv.2305.04369 Presented at the 8th Conference on Artificial Intelligence and Theorem Proving (AITP 2023).