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

Tomas Petricek
Charles University, Prague, Czechia

Jan Liam Verter
Charles University, Prague, Czechia

Mikoláš Fromm
Charles University, Prague, Czechia

Abstract

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

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

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

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

1 Introduction

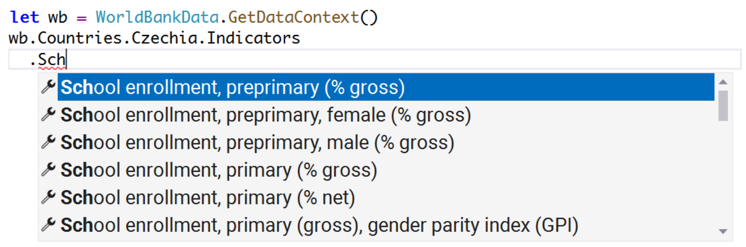
Multiple interactive programming systems, ranging from code editors for object-oriented programming languages to data exploration systems and interactive proof assistants, exhibit a remarkably similar pattern of interaction. They offer the user, who can be a programmer, a data scientist or a proof writer,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

There are subtle differences between different implementations of the general pattern. In some systems, the resulting source code will contain a trace of the choices made by the user. For example, when choosing an item from a list of class members, the code will contain the member name. In some systems, the interaction results in a block of code that can be included in the source file, but does not include a trace of the interaction. For example, invoking a proof search or case split in Idris [5] constructs a well-typed program, but leaves no trace of the command used to construct it. The nature of the generated options also varies. The list of choices may include all possible options that are valid at a given location, or it may list only a subset of the valid options. In some cases, it may also include incorrect options as, for example, in auto-completion for dynamic languages [8].

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

1. We motivate the formalism by reviewing four different systems that implement a variation on the interaction pattern. These include type providers for data access in F# [31], type providers for data exploration in The Gamma [22, 20], AI assistants for semi-automated data wrangling [25] and tooling for interactive proof assistants [2, 5, 32] (Section 2).
2. We introduce the *choose-your-own-adventure calculus*, which is a small formal structure that models an interactive system where a user constructs a program by repeatedly choosing from a list of options offered by the system (Section 3).
3. The calculus allows us to make the aforementioned subtle differences precise. We define the notions of *correctness* and *completeness* for the choose-your-own-adventure calculus. To distinguish the different ways of embedding the interactions in the edited programs, we also formally define *internal* and *external* mode of system integration.
4. We show that various programmer assistance tools, such as search and AI-based recommendations can be built on top of the primitives offered by the calculus, showing how the choose-your-own-adventure calculus supports of transfer of ideas across different kinds of interactive programming systems.

The main contribution of this paper is conceptual rather than technical. We capture a pattern that is perhaps not surprising in retrospect, but that is easy to overlook until it is given a name. We use formal programming language theory methods to precisely describe interesting aspects of the pattern. Moreover, our work also confirms that programming language theory methods can be extremely effective for studying not just *programming languages*, but also interactive *programming systems* [12].



```
let wb = WorldBankData.GetDataContext()
wb.Countries.Czechia.Indicators
  .Sch
```

School enrollment, preprimary (% gross)
School enrollment, preprimary, female (% gross)
School enrollment, preprimary, male (% gross)
School enrollment, primary (% gross)
School enrollment, primary (% net)
School enrollment, primary (gross), gender parity index (GPI)

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

2 Motivation

Computer scientists studying programming have long focused on programming languages as syntactic entities, sometimes neglecting the interactive environments in which they are inevitably embedded [9]. Notably, in many of the motivating examples that we draw from in this section, the interactive aspect of the system is only described in supplementary materials [5, 31, 2]. Only recently, programming language theory started to be used to study interactive environments [1, 15]. Our work contributes to this research direction.

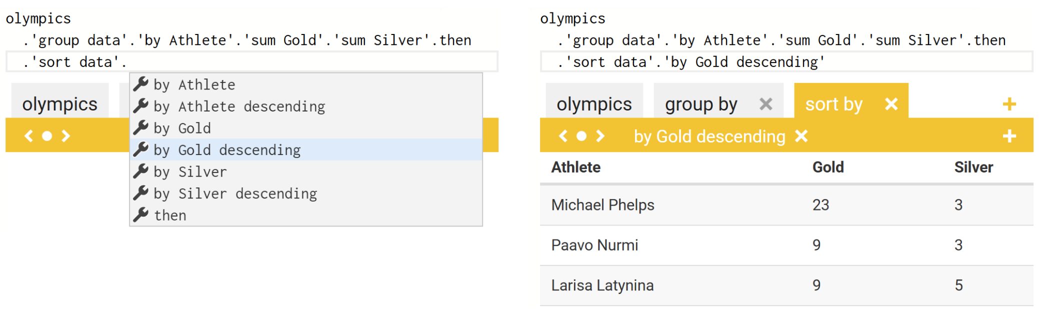
The following sections review four different instances of the choose-your-own-adventure interaction pattern. In all of those, an interactive editor offers the user some kind of a completion list during working with the system.

Type providers. F# type providers [31] are a mechanism for integrating external data sources into the F# type system. A type provider is a compiler extension, loaded and executed at compile-time and at edit-time. It can run arbitrary code to read the structure of external data and use it to generate a suitable statically-typed representation of the data, typically as objects with members. Type providers can, for example, infer the type from a sample JSON [24] or read a database schema.

The example in Figure 1 shows a simple type provider for accessing information from the World Development Indicators database. The provided wb object allows the programmer to access any indicator of any country in the database by choosing an appropriate [Country] and an [Indicator] in a chain of members wb.Countries.[Country].Indicator.[Indicator]. The result is a time series with values for the given indicator and a country. More generally, the example can be seen as a special case of a type provider for slicing n-dimensional data cube [22] – we choose a fixed value for two of the three dimensions (country, indicator, time).

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

[1] This interaction pattern has been lightheartedly called *dot-driven development* by Phil Trelford [29].



```
olympics
  .'group data'.'by Athlete'.'sum Gold'.'sum Silver'.then
  .'sort data'.
```

```
olympics
  .'group data'.'by Athlete'.'sum Gold'.'sum Silver'.then
  .'sort data'.'by Gold descending'
```

Athlete	Gold	Silver
Michael Phelps	23	3
Paavo Nurmi	9	3
Larisa Latynina	9	5

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

Data exploration. The Gamma [22] is a programmatic data exploration environment for non-programmers. In The Gamma, type providers are the primary programming mechanism. They are used not just for data access, but also for constructing queries.

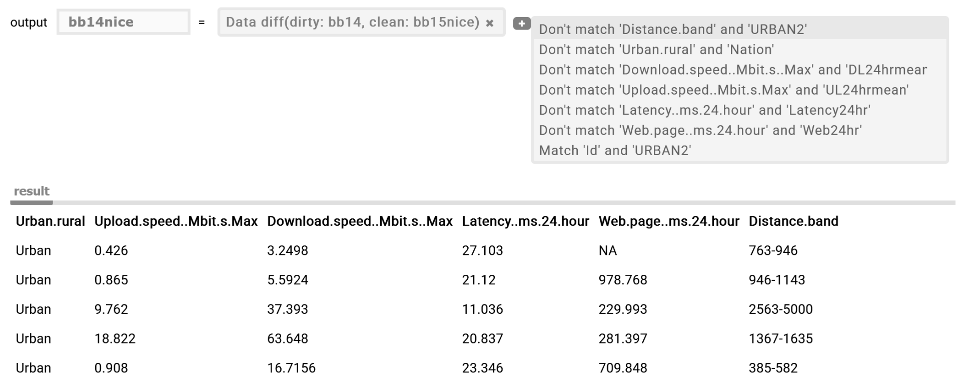
The type provider shown in Figure 2 lets the user construct an SQL-like query by repeatedly choosing operations and their parameters [20]. It keeps track of the schema and uses it to generate all possible valid parameters. When sortin data, it generates an object with two members for each columns – one for ascending and one for descending sort. Similarly, the grouping operation first offers all columns as possible grouping keys and then lets the user choose from a range of pre-defined aggregations (sum, count, average, concatenate). The system also evaluates the query on the fly, providing a live preview during editing [21].

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

AI assistants. The third instance of the choose-your-own-adventure interaction pattern comes from the work on semi-automatic data wrangling tools known as AI assistants [25]. An AI assistant guides the analyst through a data wrangling problem such as reconciling mismatched datasets, filling missing values or inferring data format and types. An AI assistant solves the problem automatically and suggests an initial data transformation, but it also generates a number of constraints that the user can choose from to refine the initial solution. If the initial solution is not correct, the user chooses a constraint and the AI assistant runs again, suggesting a new data transformation that respects the constraint.

Figure 3 shows an example. It uses the datadiff [30] AI assistant, running in a Wrattler notebook [23], to merge broadband quality data published by Ofcom for two subsequent years. The format of the CSV files for the two years differs. Columns were added, removed, renamed and their order has changed. In the example, we selected 6 columns from the year 2015 and want to find matching data from 2014.

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


output bb14nice = Data diff(dirty: bb14, clean: bb15nice)

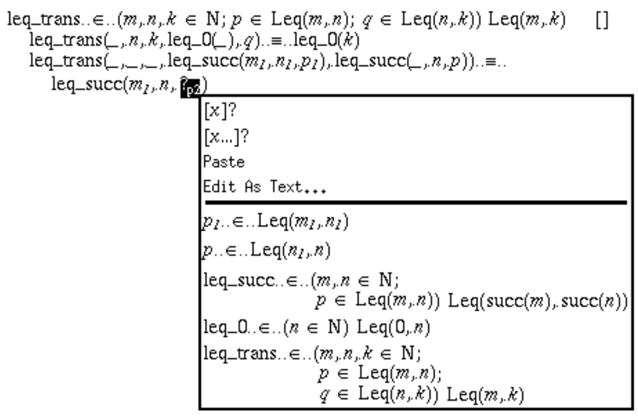
Don't match 'Distance.band' and 'URBAN2'
Don't match 'Urban.rural' and 'Nation'
Don't match 'Download.speed..Mbit.s..Max' and 'DL24hrmear
Don't match 'Upload.speed..Mbit.s.Max' and 'UL24hrmean'
Don't match 'Latency..ms.24.hour' and 'Latency24hr'
Don't match 'Web.page..ms.24.hour' and 'Web24hr'
Match 'Id' and 'URBAN2'

result

Urban.rural	Upload.speed..Mbit.s.Max	Download.speed..Mbit.s..Max	Latency..ms.24.hour	Web.page..ms.24.hour	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 prevent or force matching between specific columns.

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



```
leq_trans..∈..(m,n,k ∈ N; p ∈ Leq(m,n); q ∈ Leq(n,k)) Leq(m,k)    []
   leq_trans(_,n,k,leq_0(_),q)..≡..leq_0(k)
   leq_trans(_,_,_,leq_succ(m1,n1,p1),leq_succ(_,n,p))..≡..
      leq_succ(m1,n,?p2)
```

```
p1..∈..Leq(m1,n1)
p..∈..Leq(n1,n)
leq_succ..∈..(m,n ∈ N;
               p ∈ Leq(m,n)) Leq(succ(m),succ(n))
leq_0..∈..(n ∈ N) Leq(0,n)
leq_trans..∈..(m,n,k ∈ N;
               p ∈ Leq(m,n);
               q ∈ Leq(n,k)) Leq(m,k)
```

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

Interactive theorem provers. A fourth example of the choose-your-own-adventure interactive pattern can be found in interactive theorem provers. When writing programs in systems like Idris [6], the user typically works by stating the desired conclusion and filling the implementation with a hole. The system provides a range of interactive editing capabilities to fill the holes [16]. It can, for example, generate a case split or search for a proof [5].

Systems like Idris provide key bindings to invoke the completions, but the functionality could also be offered through a user interface. An example that illustrates this is the interactive editor for the ALF theorem prover [14], which is based on the refinement of an incomplete proof object [2]. This is illustrated in Figure 4. The user is proving the transitivity of the $\leq$ relation for Peano arithmetic natural numbers. They pattern match on the proof argument p and complete the first branch. For the second branch, they need to fill a hole $?_{p2}$ (called a wildcard in ALF). They trigger a completion and a pop-up menu shows the available variables and constructors, including `leq_trans` that can be used to complete the proof. After choosing `leq_trans`, two new holes are generated for its arguments. Those can be, again, filled interactively, by choosing p_1 and p from the completion.

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

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

3 Formal model

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

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

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

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

The definition is not a programming language calculus in the usual sense in that it does not define a concrete syntax with reduction rules. It is an abstract algebraic structure that captures the structure of a system that supports the choose-your-own-adventure interaction pattern. The definition is close to that of an AI assistant [25], which is written using a language specific for the data wrangling domain (such as cleaning scripts or input and output data) but is structurally similar. It is also worth noting that the definition may describe not just trees, but also graphs with cycles – a system can return to an already visited state. This is not practically useful, but it does not pose a theoretical problem.

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

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

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

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

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

[2] Serving as another evidence for the surprising effectiveness of the concept of a tree [17].

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

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

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

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

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

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

4 Examples

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

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

4.1 Data exploration

In The Gamma, the choose-your-own-adventure interaction pattern is used to construct a query that transforms the given input data. The query is a sequence of operations with parameters, $op(p_1, \ldots, p_n)$, loosely modelled after relational algebra [7].

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

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

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

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

The two approaches have different human-computer interaction trade-offs. In terms of cognitive dimensions [10, 4], the latter has a greater closeness of mapping, while the former is less cognitively demanding to read for a non-programmer. As discussed in Section 5, the two implementations of the choose-your-own-adventure interaction pattern also differ in terms of their formal properties.

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

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

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

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

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

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

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

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

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

4.2 Type providers

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

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

To model the completion mechanism, we also need to model information about types. We loosely follow the Foo calculus model [24] and write $\mathbb{C}$ for a set of class definitions, each consisting of an implicit class constructor and a collection of members M:

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

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

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

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

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

The model does not directly refer to type providers. Those are responsible solely for generating the type definitions in $\mathbb{C}$ as documented in earlier work [24]. It is worth noting that the type provider for data exploration, implemented by The Gamma, additionally needs to generate classes lazily [20]. To model this aspect, the simple lookup $\mathbb{C}(C)$ needs to be replaced with an operation that returns the type definition, alongside with a new context $\mathbb{C}'$ that contains additional generated type definitions (return types for all the members of the class C).

The model follows the internal mode of embedding the interaction in the program. It is easy to define the decode operation that takes the resulting generated expression and returns the sequence of choices, because the choices are items of the member chain. A slight caveat is that the completion is not invoked on an empty hole, but on a hole that contains the initial expression on which the completion is applied. We can model this using filled holes [18] and write $?_e$ for a hole containing the initial expression e. The $\mathsf{init}(?_e)$ operation then returns e alongside with an empty chain and the type of e.

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

4.3 AI assistants

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

An optimization-based AI assistant [25] thus provides another, very different, way of implementing the choose-your-own-adventure pattern. The assistant operates with respect to some input data X that does not change during the interaction and so we parameterize the choose-your-own-adventure calculus operations by the data. The input data X can be actual input data or a representative sample and so the AI assistant can be use past data to infer a cleaning script that will be used on new inputs.

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

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

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

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

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

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

The actual implementation of the optimization uses various machine learning techniques to find the optimal expression. In case of datadiff, X is a pair of datasets X_1, X_2 to be reconciled. The AI assistant uses the Hungarian algorithm [30] to construct a matching of columns from X_1 and X_2. The generated expression is a sequence of patches that can be applied to X_2 in order to reconcile its structure with the sturcture of X_1. The constraints specified by the user restrict the space of possible column matchings and so they affect $E_{\boldsymbol{c}}$. The scoring function $Q_{\boldsymbol{c}}$ is independent of the constraints and computes a sum of distances between the statistical distributions of the columns from X_1 and a patched version of X_2.

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

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

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

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

$\boxed{\Gamma \vdash \tau \Rightarrow e}$

$$\frac{}{\Gamma, x : \tau \vdash \tau \Rightarrow x}\ \textsf{(syn-var)} \qquad \frac{}{\Gamma \vdash \tau \Rightarrow ?_\tau}\ \textsf{(syn-hole)}$$

$$\frac{\Gamma, x : \tau_1 \vdash \tau_2 \Rightarrow e}{\Gamma \vdash \tau_1 \to \tau_2 \Rightarrow \lambda x.e}\ \textsf{(syn-lambda)} \qquad \frac{\Gamma \vdash \tau_1 \to \tau_2 \Rightarrow e_1 \quad \Gamma \vdash \tau_1 \Rightarrow e_2}{\Gamma \vdash \tau_2 \Rightarrow e_1\ e_2}\ \textsf{(syn-app)}$$

$$\frac{b \in \{\textsf{true}, \textsf{false}\}}{\Gamma \vdash \textsf{bool} \Rightarrow b}\ \textsf{(syn-bool)} \qquad \frac{\Gamma \vdash \tau \Rightarrow e_1 \quad \Gamma \vdash \tau \Rightarrow e_2 \quad \Gamma \vdash \textsf{bool} \Rightarrow e}{\Gamma \vdash \tau \Rightarrow \textsf{if}\ e\ \textsf{then}\ e_1\ \textsf{else}\ e_2}\ \textsf{(syn-cond)}$$

Figure 5 Illustrative set of simple type-directed program synthesis rules

4.4 Theorem proving

In the previous three sections, we showed how existing formally well-documented systems fit the choose-your-own-adventure interaction pattern. Although interactive theorem provers and editors for dependently typed languages implement similar kinds of interactions, there is no well-documented system that exactly fits the pattern. The closest example is perhaps the recently envisioned mixed-mode interaction theorem prover [32]. Rather than reframing the implementation of an existing system, this section thus outlines a possible implementation.

Theorem provers. There are two approaches to interacting with an interactive theorem prover. In Coq, the user writes a sequence of tactics that transform proof goals. In Agda or Idris, the user writes a term, or program, of a type that represents the theorem. Interactive editors exist for both types of systems. For Coq, the `Company-Coq` [26, 3] extension offers auto-completion, which recommends available tactics, hypotheses and local definitions, but it does not filter them based on what is valid in a given context. For Idris, the interactive editor [5] offers a range of commands that transform the selected term by adding a case split, a missing case or by automatically searching for a proof. In Idris, the system produces valid completions, but those cover only a small number of situations.

Despite the different ways of working, an implementation of the choose-your-own-adventure pattern for both types of systems would be similar. Based on the sub-goal that the user is currently proving, the system would recommend a range of tactics that can be applied to the sub-goal. In the case of Coq, the selected tactic would be added to the sequence. In Idris, the selected tactic would be applied to transform the current term. The difference is in the mode of embedding. A system for Coq would provide internal embedding in that the selected option is added to the proof source code. (Much like selecting a completion when using type providers appends a member access.) A system for Idris-like language would use the tactic to transform the term, making it impossible to reconstruct the sequence of applied tactics as in the external mode of embedding.

Type-directed synthesis. Thanks to the equivalence between programs and proofs, techniques akin to tactic-based proof construction have also emerged in work on type-directed program synthesis [13]. As illustrated in Figure 5, the synthesis process can be described as a set of rules of the form $\Gamma \vdash \tau \Rightarrow e$ that describe ways of synthesizing expressions e of a type τ. Existing implementations of the mechanism typically aim to automate program synthesis and use more precise type information, such as refinement types [27] and graded types [11], or include examples [19]. However, the same rules could be used to guide an interactive choose-your-own-adventure system. If the interaction was invoked to fill a typed hole $?_\tau$ in a context Γ, the system could collect multiple e such that $\Gamma \vdash \tau \Rightarrow e$ and offer a choice of such options. Note that the definition in Figure 5 synthesizes sub-expressions recursively, but a choose-your-own-adventure system may always fill those with a typed hole using (syn-hole).

Formal model. From the perspective of user interaction, a proof assistant where a user interactively constructs a term of a given type is very similar to an interactive tool for type-directed program synthesis. The key difference being that theorem provers like Idris and Agda use rich dependent type theories.

For example, 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 synthesizing terms of type τ. We again write $E[?_\tau]$ for a completion context containing a (typed) hole; we assume that the variables Γ available in the completion context of the hole can be obtained using $\mathsf{vars}(E[\_])$.

To model a choose-your-own-adventure interaction akin to Idris, the state of the system would be the term e itself, initially a typed hole. The choices operation synthesizes possible completions using $\Rightarrow$ (restricted, e.g., to only generate terms of a certain maximum size) 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. The choices operation suggests ways to fill a hole in the term:

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

The choices operation synthesizes possible terms of a type required by the hole. Since the state σ is a term, the choose operation simply returns it. The definition models the external embedding of the interaction, i.e., a system that behaves according to Idris. It constructs the term, but does not record the completion choices. That said, a system akin to Coq that constructs a sequence of tactics could be modelled too if the state was a sequence of tactics and choices appended the available tactics as options to the end of the current sequence.

5 Properties

The choose-your-own-adventure calculus lets us precisely discuss differences between how different programming systems interact with the user. We saw this in Section 3, which defines internal and external mode of embedding to distinguish between systems where the interaction leaves a reconstructible trace in the constructed program and systems where it does not. In this section, we make precise two properties that were introduced informally in the context of data exploration in The Gamma [22].

The choose-your-own-adventure system for data exploration in The Gamma is *correct*, which means that all programs that a user can construct using the system, by repeatedly choosing from the auto-completion list, are well-typed. The system is also *complete*, meaning that the user can use auto-completion to construct all possible programs. That is, there are no well-typed programs that cannot be constructed interactively, by repeatedly choosing options from the offered list of choices.

Correctness. The notions of correctness and completeness can, in general, be defined for any choose-your-own-adventure systems with respect to some system-specific distinction between correct and incorrect expressions. We write $\mathcal{E} \subseteq E$ for the subset of correct expressions.

For some systems, the set of correct expressions $\mathcal{E}$ is 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 complete proofs) are correct. For systems where the completion is string-based, we may treat all syntactically-correct programs as correct.

▶ **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:*

- $\forall \sigma_1, .., \sigma_n$ *and* $\iota_i, .., \iota_n$ *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 $\sigma_1, \sigma_2, \ldots$, then the programs we could generate from any of the intermediate states are correct.

The property depends on what we choose as the subset of correct expressions $\mathcal{E}$. Trivially, all systems are correct with respect to $\mathcal{E} = E$. However, the systems discussed in Section 4 are all correct with respect to non-trivial choices:

- For the data exploration system discussed in Section 4.1, we say that correct expressions are those where the parameters of all operations are fully-specified. That is, no operation requires further arguments. With respect to this definition, the system is correct. However, this is the case 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) one, the system would not be correct.
- For type providers (Section 4.2), correct expressions are those that are well-typed. With respect to this definition, the system is correct because the choices operation offers available members based on the type information. This also holds for 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.
- In the case of AI assistants (Section 4.3) the correctness of the system depends on the expressions returned by the optimization algorithm ($\arg\max$) from the set of all possible cleaning scripts $E_{\boldsymbol{c}}$. In general, the algorithm can return any $e \in E_{\boldsymbol{c}}$ and so system correctness is a matter of definition. The system is correct if and only if $E_{\boldsymbol{c}} \subseteq \mathcal{E}$ for

all possible sets of constraints $\boldsymbol{c}$. In practice, it is more important that the constraints generated by choices are well-formed.

- For the 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 [19], that is if $\Gamma \vdash \tau \Rightarrow e$ then also $\Gamma \vdash e : \tau$. Note that a correct choose-your-own-adventure system can be defined even using unsound synthesis rules – it would be sufficient to filter the recommended expressions in choices to the ones that are well-typed.

There may be useeful systems that violate the correctness property. An tool based on a large language model (LLM) may, for example, generate code with errors that the programmer can then correct. A more interesting case are cases like the data exploration systems discussed above where the program only becomes correct after multiple subsequent choices are made, for example to fully specify arguments of an operation.

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 does not support internal embedding. If the operation is dropped, we cannot reconstruct it from the generated source code. Generating code that includes the not-fully-specified operation allows external embedding, but makes the system incorrect. It would still satisfy a weaker definition of (eventual) correctness:

▶ **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 $\forall i \in k+1 \ldots n \,.\, \iota_i \mapsto \sigma_i \in \mathsf{choices}(\sigma_{i-1})$.*

Eventual correctness models systems where some sequences of choices result in invalid programs, but it is always possible to make further choices to reach a valid program. In general, it is always possible to turn an eventually correct system into a correct one:

1. As in the case of the data exploration, the system can remember the last state for which the choose operation returned a correct program and use it until the next correct state is reached. This makes any eventually correct choose-your-own-adventure system correct, but it does not support internal embedding.
2. Alternatively, we can construct a system that collapses all sequence of temporarilly invalid states $\sigma_1, \ldots, \sigma_n$ identified by $\iota_1, \ldots, \sigma_n$ 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 $\iota_1 \ldots \iota_n \mapsto \sigma_n$ designated by a joined identifier. This makes the system correct and also preserves external embedding, but it potentially generates too many choices that are difficult to understand.

There is more to be said about correctness of interactive programming systems, but the conceptual framework provided by the choose-your-own-adventure calculus makes it possible to take the first step. Similarly, the model lets us formally define the second property mentioned earlier.

Completeness. todo

..

..

..

▶ **Definition 6** (Completeness)**.** *Assume that E is a set of all possible expressions in a language and $\mathcal{E} \subseteq E$ is a set of all expressions that are correct with respect to some case-specific notion of correctness (e.g. well-typed). A choose-your-own-adventure system is complete if:*
$\forall e \in \mathcal{E}.\ \exists \sigma_1, \ldots, \sigma_n$ *such that* $\sigma_i \in \mathsf{choices}(\sigma_{i-1})$ *and* $e = \mathsf{choose}(\sigma_n)$.

That is, for any correct program, there is a sequence of choices that leads to a system state that choose turns into the given program. This is a more subtle property that not all of my above examples have:

* In The Gamma, the programs that can be generated are restricted - you can only use a fixed set of aggregation operations (rather than writing your own) and only a restricted set of parameters (sorting by a key, but not based on a custom expression). For those, the type provider is complete. However, if we treated a more general-purpose query language as the underlying language, the provider would not be complete.

* In AI assistants, the system offers a set of constraints that is generated based on the selected expression, but this does not let you construct arbitrary constraints. Moreover, because the choose operation is AI-based, it is not guaranteed that there is a way to get it to generate a specific program (unless we can supply constraints that restrict the set of programs $E_{\boldsymbol{c}}$ to just a single program).

* In interactive theorem prover, we could possibly offer all possible ways of filling a hole (up to renaming), but this would not be very practical. It is more likely that tactics will only generate a subset of valid proof/program steps and the user has to write some other steps manually.

The nice thing about the choose-your-own-adventure formalism is that it also lets us talk about (and think about!) properties that are more specifically about the user interaction with an interactive programming system.

6 Applications

ways of integrating automation with manual interaction - confirm each step vs. auto and retrace

[28]

[32]

7 Limitations

hard to edit - you cannot easily go back

References

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

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

3 David Aspinall. Proof general: A generic tool for proof development. In Susanne Graf and Michael Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 38–43, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

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

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

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

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

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

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

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

11 Jack Hughes and Dominic Orchard. Program synthesis from graded types. In Stephanie Weirich, editor, *Programming Languages and Systems*, pages 83–112, Cham, 2024. Springer Nature Switzerland.

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

13 Tristan Knoth. *Type-Directed Program Synthesis.* PhD thesis, University of California, San Diego, 2023. URL: https://escholarship.org/uc/item/4g11m7rq.

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

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

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

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

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

19 Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 619–630, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2737924.2738007`.

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

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

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

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

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

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

26 Clément Pit-Claudel and Pierre Courtieu. Company-coq: Taking proof general one step closer to a real ide. In *Second International Workshop on Coq for PL (CoqPL '16)*, January 2016. URL: `https://dspace.mit.edu/handle/1721.1/101149.2`.

27 Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 522–538, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2908080.2908093`.

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

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

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

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

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