Interactive Data Analysis with Lively Typed Tables

ALEXANDER BANDUKWALA✉, University of Michigan, USA
CYRUS OMAR, University of Michigan, USA

Programming systems tailored for working with tabular data (*tabular programming systems*), such as spreadsheets and computational notebooks, are essential tools in data science. However, widely adopted systems are limited by the absence of static typing, which restricts the editor support they can provide, particularly when code is organized into reusable functions. Statically typed alternatives are limited by the fact that many useful operations on tables produce results whose column schema is data-dependent, e.g. pivots, unstack operations, or one-hot encodings.

This paper introduces *Hazel Lab*, a new tabular programming system built as an extension of Hazel, a live gradually typed functional programming environment. It aims to combine the expressivity of dynamically typed systems with the editor support of static typing by incorporating several novel mechanisms into Hazel. Tables are manipulated as sequences of labeled tuples, and we add several useful gradually typed operations on labeled tuples to increase expressiveness, including operations that convert field names to and from strings. These operations support best-effort static typing and fall back to the unknown type when a schema cannot be statically determined. We evaluate the expressiveness of these core abstractions using the Brown Benchmark For Table Types (B2T2), finding that Hazel Lab is able to reasonably express every example.

In order to improve type-based feedback, including error localization, when the fallback to the unknown type is needed, we introduce *live typing*, which builds on the fact that Hazel is a maximally live programming environment, even when there are static errors in the code, to feed dynamically observed instantiations of statically unknown types back into the static type checker. This feedback is complementary to the dynamic feedback that Hazel already distinctively provides by way of its live probes. We introduce rich probes—an extension of live probes with domain-specific table views that allow users to edit their functional data pipelines through direct manipulation interactions.

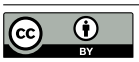
We evaluate the usability of our overall design by conducting a lab study with 7 participants, asking them to perform a variety of data cleaning and analysis tasks. The study evaluates the usability and usefulness of the proposed features for users already familiar with statically typed functional programming, rather than to assess transfer to data science workflows by scientists without that training. We find that participants could effectively use the table operations for data cleaning and transformation tasks, and that live typing helped them both understand and debug existing analyses. Most participants responded positively to adopting the evaluated features in their own programming environments, with none responding negatively.

CCS Concepts: • **Software and its engineering → Integrated and visual development environments**.

Additional Key Words and Phrases: tabular programming, gradual typing, live programming, labeled tuples, data science, type systems

ACM Reference Format:
Alexander Bandukwala and Cyrus Omar. 2026. Interactive Data Analysis with Lively Typed Tables. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 369 (October 2026), 34 pages. https://doi.org/10.1145/3839501

Authors' Contact Information: Alexander Bandukwala, University of Michigan, Computer Science and Engineering, Ann Arbor, USA, abanduk@umich.edu; Cyrus Omar, University of Michigan, Computer Science and Engineering, Ann Arbor, USA, comar@umich.edu.



ACM 2475-1421/2026/10-ART369
https://doi.org/10.1145/3839501

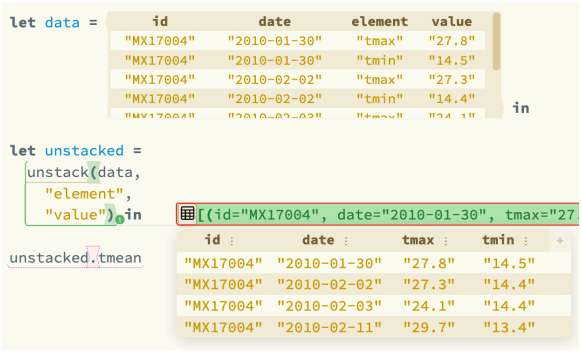


Fig. 1. A Hazel Lab program that reshapes a table of weather observations using unstack. Columns are generated dynamically from the element values; in the resulting table the `tmean` column does not exist. A live typing error highlights the invalid access, illustrating how runtime-determined schemas can be checked by live typing.

1 Introduction

Exploratory data analysis—iteratively exploring, transforming, and visualizing data to discover patterns—is central to data science [53]. In this paper, we focus on the early stages of data science workflows—particularly the *data acquisition* and *data preparation* stages where data is ingested, explored, filtered, and reshaped [8]. These stages are typically carried out in a programming system tailored for working with tabular data (a *tabular programming system* or *TPS*) consisting of a live programming environment (e.g., a computational notebook environment like Jupyter [22] or a spreadsheet environment), a programming language, and various libraries.

1.1 Limitations of Dynamically Typed TPSs

The most widely adopted contemporary TPSs provide dynamically typed tables, including R data frames [41], Pandas data frames [51], and spreadsheets. While these systems offer proven flexibility for interactive exploration, there are several well-understood limitations of dynamic typing.

First, editor services—such as autocompletion, type reporting, and error highlighting—are limited by the absence of static typing. To the extent that these services are available at all in these systems, they are tied to top-level (i.e. global) runtime values. Code operating on a function's arguments lacks these editor services. We call this the **function barrier problem**. The lack of support for writing reusable code is prevalent in existing data science tools: data scientists tend to copy-and-paste code to reuse functionality rather than building reusable abstractions [19], increasing duplication and making it harder to adapt analyses to new datasets or environments.

Existing TPSs also suffer from the **distal feedback problem**, where error feedback is distant both spatially—runtime failures are reported far from the offending syntax, via stack traces, cell outputs, or logs rather than inline in the source—and temporally, as evaluation-dependent feedback arrives only after potentially expensive computations complete.

1.2 Limitations of Statically Typed TPSs

Statically typed table libraries are used more heavily in data-engineering settings than in exploratory analysis, where workflows are less settled and analysts frequently reshape and reinterpret data. In principle, static types could support analysts by providing early, localized feedback independent of execution, mitigating the **function barrier** and **distal feedback** problems. These properties also support the transition to the Communication Phase of the data analysis pipeline [8], where analyses are stabilized, documented, and communicated to others.

In practice, however, several limitations make statically typed TPSs less suitable for data analysis. Compared to dynamically typed TPSs, live and notebook-style interfaces for statically typed languages are less developed, which limits the immediacy and locality of feedback while iteratively refining an analysis. In addition, statically typed systems commonly face the **external schema problem**: when data are ingested from external sources and parsed at runtime, the table's schema is unavailable for static analysis, preventing type-driven feedback unless the schemas are supplied explicitly through preprocessing, code generation, or reflection.

Finally, statically typed systems face an **expressivity problem** with two aspects. First, many common EDA transformations involve structural operations on tables—such as reshaping, reorganizing, or composing schemas—that are straightforward in dynamically typed environments but require sophisticated type-system features or compile-time metaprogramming to express statically. As observed in the Brown Benchmark for Table Types (B2T2), the ability of a type system to naturally express such operations varies widely, and limitations in expressivity can make certain analyses cumbersome or infeasible despite the benefits of static typing [24, 25].

Second, even when operations can be expressed, the type system may be unable to express the constraints needed to statically verify them. Many common table operations produce results whose structure depends on the values contained in the data. Fig. 1 illustrates a representative example, adapted from Wickham [58], in which a table of weather observations is reshaped from a long to a wide format. The `unstack` operation takes a table with `element` and `value` columns—here containing readings such as `"tmax"` and `"tmin"`—and produces a table with one column per distinct element. Crucially, the output schema of `unstack` depends on the runtime contents of the `element` column: if additional element values are present, corresponding columns are created automatically. In the example, the user subsequently attempts to access a `"tmean"` column that does not exist in the result. Because the set of output columns is determined only after inspecting the data, a traditional static type system cannot determine whether this access is valid. As a result, no feedback is provided until execution, at which point the program fails.

This pattern—where the structure of a table, and thus the validity of downstream operations, depends on values that are not known until runtime—is pervasive in exploratory workflows. Dynamically typed systems support these operations but do not provide type-directed editor services for the results. Full static verification would require scanning the entire dataset—prohibitively expensive for large data—so conventional static type systems either make these operations inexpressible or fail to detect errors.

1.3 Lively Typed Tabular Programming

We introduce *Hazel Lab*, a tabular programming system built as an extension of Hazel [29], a live functional programming environment. Hazel Lab aims to bring the benefits of static typing to tabular workflows without sacrificing the expressivity of dynamically typed TPSs. Hazel Lab makes three contributions.

To address the **expressivity problem** of statically typed tabular programming, our first contribution is a set of structural operations over labeled tuples and tables that are as statically typed as possible while remaining usable in exploratory settings. Structural operations over labeled records are not new (Section 5.2), but our design in a live, gradually typed language is. Hazel's pre-existing labeled tuples (Section 3.1) provide a flexible but typed representation of rows, and tables (Section 3.2) are represented as sequences of these tuples, allowing tabular structure to emerge compositionally from ordinary sequence operations rather than from a separate table abstraction. The structural operations—extension, contraction, and broadcast projection over tables—are gradually typed and preserve full static typing where feasible, alleviating the **distal feedback** and **function barrier** problems when applied to data with static types.

Our gradually typed approach to addressing the expressivity problem comes at the cost of introducing **distal feedback** for operations without statically determined types. Inspired by prior work on live [59] and gradual [47] typing, our second contribution extends static type checking with *live types* (Section 3.3): a hybrid approach that integrates dynamic observation of runtime values back into static type checking, easing the spatial aspect of the **distal feedback problem**. Live typing further addresses the **expressivity problem**: as operations such as `unstack` produce outputs whose structure depends on runtime data, live types discovered during execution are fed back into the static type checker. This allows Hazel Lab to detect type mismatches that would otherwise appear only at runtime, localizing errors directly to the relevant program expressions and providing actionable feedback early in the exploratory workflow. Editor feedback is grounded in these statically and dynamically refined types rather than in mutable kernel state, targeting the **function barrier** and **distal feedback** problems: a notebook such as Jupyter derives editor services from the global values currently in its kernel, so code operating on a function's arguments receives no feedback, whereas live types attach to expressions throughout the program, and editor services remain accurate inside functions.

Hazel provides live probes [10], which display the runtime values of expressions and patterns inline in the editor. Our third contribution extends these probes with domain-specific table views, yielding *rich probes* (Section 3.4) that allow users to interact with tabular data through direct manipulation while preserving textual source code as the authoritative representation. Rich probe actions rewrite the underlying syntax rather than modifying values directly, bridging the gap between spreadsheet-style interaction and programmatic data cleaning.

Hazel Lab also resolves the **external schema problem** through an in-editor CSV data loader (Section 2.1.3): external CSV files can be inserted directly as expressions, parsed into sequences of tuples, and typed by Hazel's standard bidirectional type system. No special preprocessing or ad-hoc schema language is required; the inferred type of the imported table serves as its schema, enabling type-directed feedback immediately after import.

We begin in Section 2 with a tutorial that walks through an example data analysis, demonstrating how live typing and static feedback combine in practice before turning to the system's design and implementation. To evaluate our system's expressiveness and error localization, we use the Brown Benchmark for Table Types (B2T2) [24, 25], which provides a curated set of table-oriented operations and error scenarios that allow systematic comparison with previous work. We also conduct an exploratory user study in which 7 participants experienced in statically typed functional programming performed data cleaning and analysis tasks, examining the usability of the table representation (**RQ1**), rich probes (**RQ2**), and live typing (**RQ3**). The study is intended as initial evidence to guide the system's design, not as a general usability claim; whether its benefits extend to data analysts without that background is left to future work. Participants found the table representation intuitive and live typing helpful for identifying bugs, and most responded positively to adopting the evaluated features in their own programming environments.

2 Tutorial Introduction

We illustrate the features of Hazel Lab through a simplified data analysis of fishery catch data shown across Figs. 2–6. An analyst incrementally prepares the data through a sequence of interactive transformations and then restructures it for presentation, ultimately producing a reusable pipeline that summarizes total biomass caught by region and sector. Each figure captures a successive stage and emphasizes a distinct capability of the system. A build of Hazel Lab with the example data is included in the artifact [6], and the reader is welcome to follow along interactively.

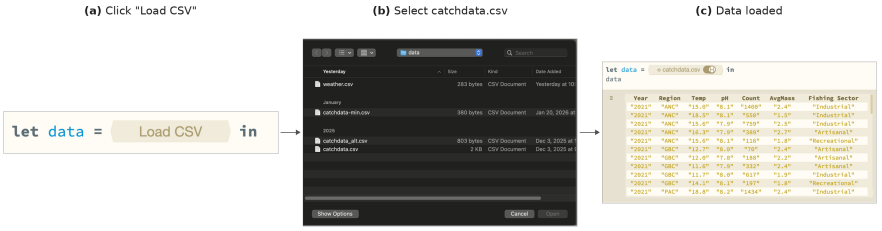


Fig. 2. CSV data loading workflow: (a) the user clicks the Load CSV button, (b) selects a file from the dialog, and (c) the data is loaded into the environment.

2.1 Data Acquisition

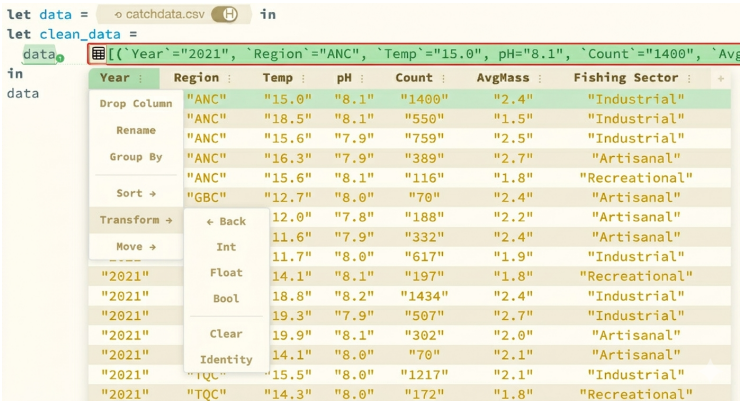
Most programming languages load, parse, and validate external data at runtime, giving rise to the **external schema problem**: the programming system and its editor tooling have little information about the structure of the ingested data.

2.1.1 Notebook Environments. Notebook environments such as Jupyter Notebooks [22] interleave edit-time and runtime operations: users can write code after partially executing the notebook, modify existing code, and rerun cells on the current kernel state. Because external data are typically loaded dynamically, editor feedback instead depends on the current state of the running kernel. Out-of-order execution [13, 38] and side effects from earlier data ingestion can make state irreproducible. As a result, editor feedback may be inaccurate or misleading in subsequent runs of the analysis.

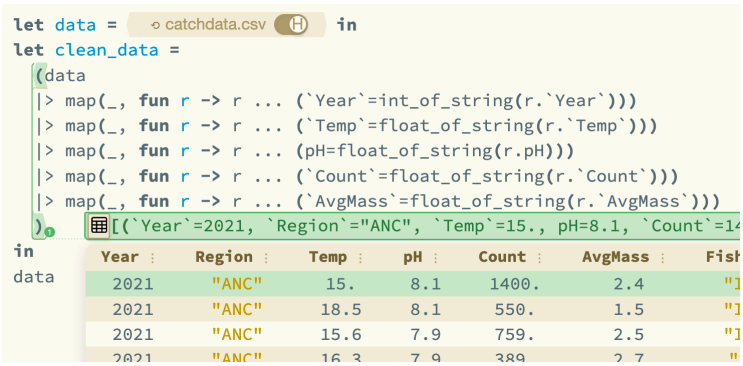
2.1.2 Type Providers. Type providers [37, 50] take a different approach to the **external schema problem**: at edit/compile time, a provider queries the external data source—a sample or the full dataset—and generates static types for the program to use, while the data remain external and are loaded at runtime. The generated types describe the runtime data as long as the source stays consistent with what was inspected, which the analyst can ensure by vendoring the data with the program. We discuss systems that extend type providers further into an analysis pipeline in Section 5.4.

2.1.3 Hazel Lab's Approach. Our approach, shown in Fig. 2, builds on livelits [28] to introduce a CSV data loader that inserts external data directly into the program source at any location where an expression is expected. When invoked, the loader prompts the user to select a CSV file, parses it into a list of tuples, and inserts the resulting syntax behind a projectional interface. The CSV loader performs no schema induction, so all fields are initially imported as strings. The interface also provides a toggle for interpreting the first row of the CSV as column headers; when enabled, the loader produces labeled tuples. This representation—tables encoded as sequences of labeled tuples—is described in Section 3.2. Because the imported data become ordinary expressions in the source code, the **external schema problem** is addressed by Hazel's standard bidirectional type system [61]: the inferred type of the embedded table serves as its schema, and any user-supplied schema is simply an ordinary type annotation, with any mismatch marked directly on the offending values within the data.

Hazel is a pure programming language, so placing data directly in the program makes it reproducible. Users may also switch away from the projectional interface to edit the embedded data as ordinary program text without invoking any external tool. These in-source edits are compatible with previously developed version-control and edit-tracking techniques for Hazel [1].



(a) Probed dataset with the column-header dropdown for `Year` showing suggested conversion functions



(b) Rewritten syntax and resulting dataset after applying the type conversions and removing temperature outliers

Fig. 3. Interactive data preparation using the projectional table editor

2.2 Data Preparation

We developed a live and rich projectional editor for tables [16] that is integrated directly alongside the textual syntax. Building on Hazel's existing *live probes* [10], which allow inspection of the runtime values of arbitrary terms and patterns, we introduce *rich probes*: interactive domain-specific views for probed values. When a probed expression evaluates to a list of labeled tuples, the system presents the result as a rich table rendered adjacent to the originating syntax. In Fig. 3a, the highlighted text beside the probe is the probed value's ordinary textual rendering—all a plain live probe would show—with the rich table view rendered beneath it. The rich table probe enables users to quickly explore large datasets using familiar table controls while maintaining a direct connection to the source code.

The table interface, seen in Fig. 3a, is interactive rather than read-only. It supports common spreadsheet-style operations such as column sorting, type conversions, moving, renaming, or dropping columns that are driven by the live type of the active table. Each interaction automatically rewrites the corresponding textual syntax so that the transformations are explicit and reproducible. These edits are encoded as a pipeline of function applications that operate over lists (Section 3.2), using labeled tuple extension (Section 3.1) to overwrite columns. Fig. 3b shows the rewritten syntax and data after conversions and outlier removal have been applied.

```
let enriched_data =
  (clean_data
  |> map(_, fun r -> r ... (`BioMass`=r.`Count` *. r.`AvgMass`))
  |> map(_, omit_labels(_, `Temp`))
  |> map(_, omit_labels(_, `Count`))
  ),  [(`Year`=2021, `Region`="ANC", pH=8.1, `AvgMass`=2.4, `Fishing Sector`
```

Year	Region	pH	AvgMass	Fishing Sector	BioMass
2021	"ANC"	8.1	2.4	"Industrial"	3360.
2021	"ANC"	8.1	1.5	"Industrial"	825.
2021	"ANC"	7.9	2.5	"Industrial"	1897.5

Fig. 4. The enrichment stage of the pipeline in which new columns are introduced and unnecessary ones removed

Users may fluidly alternate between manipulating the projectional table and editing the textual syntax. Some operations cannot be fully specified from the table interface's menus: adding a column requires an expression for its values, and filtering rows requires a predicate. For these, the system inserts expression holes [30] into the generated code—placeholders that users later complete via textual editing, with Hazel's type-directed feedback guiding them to a correct and well-typed program.

2.3 Feature Engineering

Fig. 4 shows an additional `BioMass` column being introduced via the table interface. Since the interface does not directly specify the value for this new column, an expression hole is inserted into the syntax, which the user then completes in the textual editor with `r.`Count` *. r.`AvgMass`)`, producing the calculated values in the probed table.

Structural operations such as `omit_labels` are similarly applied to drop columns like `Count` and `Temp` from each row. Because these operations are statically typed, the resulting transformed data retains fully typed schema information[1]. Crucially, static analysis provides quick feedback to the textual program—while evaluation is running asynchronously—allowing type errors and inconsistencies to be reported without waiting for expensive computations to complete, reducing the temporal aspect of the **distal feedback problem**.

2.4 Interpretation Using Live Typing

Fig. 5a illustrates a pivot table operation that transforms the values in the `Fishing Sector` column into new column headers, summarizing the data for easier interpretation. The function `pivot_table` (Fig. 5b) takes a table, projection functions that assign each row a column label (a string, since labels are second-class in our system) and a row key of arbitrary type, and an aggregation function. It abstracts over the two type variables `r` and `agg` with **`typfun`**, Hazel's expression-level abstraction over types, making `pivot_table` polymorphic in its row and aggregate types in the sense of System F (Hazel's parametric polymorphism is explicit[1]). It groups rows by the projected labels, applies the aggregation, and produces a new table whose columns correspond to the distinct column labels.

Notably, the inferred return type of the `pivot_table` function is `[?]`, a list whose element type is `?`, the *unknown* type from Hazel's gradual typing, which may itself appear within larger types (for example, `(Int, ?)` is a pair whose first component is an `Int` and whose second is unknown). Since column labels are constructed dynamically as strings, the set of column names cannot be

[1]In the current Hazel implementation, implicit parametric polymorphism is not yet supported, so `map` and `filter` produce results of unknown type. With implicit polymorphism in place, these combinators would receive fully determined types, and the entire transformation pipeline would be statically typed.

(a) Pivot table example with live typing errors

```
let pivot_table =
typfun r -> typfun agg ->
fun (table : [r], new_col : r -> String,
     index : r -> ?,
     aggregate : [r] -> agg) ->
  let indices = unique(map(table, index)) in
  let new_cols =
    unique(map(table, new_col)) in
  map(indices, fun idx ->
    (index = idx) ... from_lvs(
      map(new_cols, fun col ->
        (label = col,
         value = aggregate(
           filter(table, fun r ->
             index(r) == idx
             && new_col(r) == col))))))
```

(b) User-defined `pivot_table` function

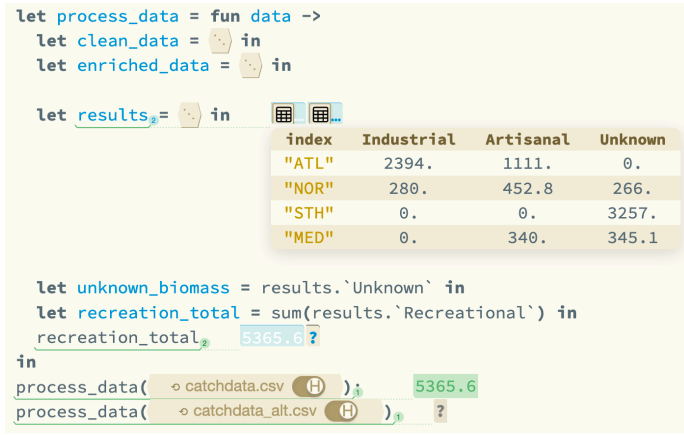
Fig. 5. Pivot tables in Hazel Lab. Column headers are dynamically constructed from `Fishing Sector` values; error marks and the lightning-bolt error message indicate live typing errors when accessing the non-existent Unknown sector.

determined statically—the same expressivity challenge as `unstack` (Fig. 1). To support expressing such dynamic operations, Hazel Lab provides two helper functions, `to_lvs` and `from_lvs`, which convert between lists of *labeled values* (i.e., label/value pairs, abbreviated as `lv`) and labeled tuples.

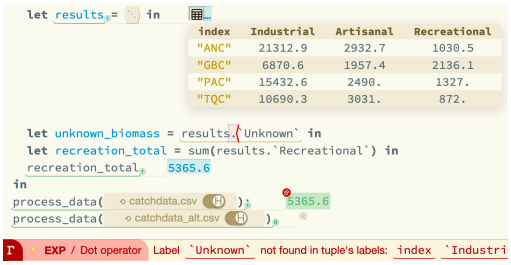
Since the types of value-dependent functions cannot be fully determined statically, Hazel Lab augments static checking with dynamically observed types via its live typing mechanism. As a live programming environment, Hazel provides immediate static feedback and reevaluates the program after each modification. Once evaluation completes, the system gathers the runtime values of statically underdetermined expressions, synthesizes dynamic types from them, and recomputes static analysis, producing secondary error marks in the editor (Section 3.3). In Fig. 5a, these marks indicate the missing column `Unknown` projected from the pivoted table. Traditional static type systems cannot assign a precise type for this operation, and gradually typed systems generally provide no in-editor feedback for such dynamic errors.

2.5 Reusable Pipeline

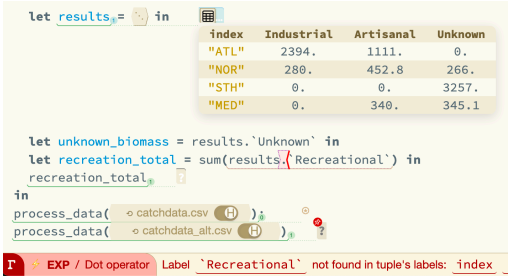
Fig. 6a shows the final step in building a reusable data pipeline: the pipeline has been abstracted into a function, with the input table as a parameter. Calling this function with the original dataset would preserve the static (assuming implicit polymorphism as previously discussed[1]) and dynamic marks observed in the original pipeline (Fig. 5a), solving the **function barrier problem**. Instead we are calling the function with two different datasets, and the dynamic marks have disappeared: the resulting pivot tables have inconsistent types, so no meet exists and the result keeps its static type—here the unknown type—from which no value-dependent marks can be derived (Section 3.3). Each probe collects a value per invocation—here, one per dataset. Live probes allow users to *pin* a probe on a function application, restricting every probe to the values from that invocation. Hazel Lab's live typing is then restricted to the currently pinned calls. In Fig. 6b, pinning the first invocation of the reusable pipeline highlights a dynamic mark on projecting the `Unknown` column because the first dataset contains no rows with the `"Unknown"` sector. Conversely, pinning the second invocation (Fig. 6c) highlights a mark on projecting `Recreational`, since the second dataset contains no rows for the `"Recreational"` sector after filtering. This mechanism enables users to inspect dynamic behavior and value-dependent errors for each dataset independently while viewing common errors when the resulting schemas agree. This diminishes the spatial aspect of **distal feedback** in the presence of disparate call sites.



(a) Pipeline moved into a function and applied to two datasets; the live typing marks disappear because each dataset produces a pivot table with a different schema.



(b) Invocation with `catchdata` pinned. Live typing is restricted to this invocation, producing a mark on `Unknown`.



(c) Invocation with `catchdata_alt` pinned. Live typing reflects the new dataset, producing a mark on `Recreational`.

Fig. 6. Reusable pipeline and the effect of pinning different invocations. The hexagonal icons are folds collapsing the pipeline stages from the preceding figures. Static type marks remain invariant, while dynamic marks are determined by the currently pinned invocation.

3 System Design

3.1 Labeled Tuples

Hazel provides *labeled tuples*: ordered heterogeneous collections whose elements may carry optional string-like labels, combining the order sensitivity of tuples with the named access of records. Order sensitivity is important for tabular programming since many table implementations and data formats (e.g., CSV) preserve column ordering; it also permits interleaving labeled and unlabeled entries, letting datasets evolve toward richer labeling over time.

3.1.1 Syntax and Types. Labeled tuples are written as a parenthesized, comma-separated list whose elements may be labeled or left unlabeled. Labels containing spaces, capital letters, or special characters are surrounded by backticks. For example, the tuple

```
(`Total Biomass`=42, `Region`="NOR", 7)
```

has elements labeled `Total Biomass` and `Region` followed by an unlabeled element, and its type mirrors this structure:

```
(`Total Biomass`=Int, `Region`=String, Int)
```

Hazel's type system tracks the types and labels of each element, letting operations refine or check types while preserving label information.

3.1.2 Structural Operations. Maintaining tidy data [58] requires structural modifications to rows: adding, removing, and reordering their entries. Hazel's labeled tuples provide only *projection* (`t.ℓ`), which reads a single entry. Hazel Lab therefore adds five primitives: *extension* (`t1 ... t2`), which concatenates two tuples, with labels in the right-hand tuple overriding matching labels in the left—a fresh label adds a column, a reused label overwrites that column's value—and four *contraction* operations that select entries and strip labels (*select labels*, *project labels*, *omit labels*, and *omit all labels*). Without these primitives, such transformations require decomposing each tuple into its constituent entries and reconstructing it, coupling programs to the concrete types and making it difficult to evolve the codebase as data change. A complete reference for each operation—with syntax, examples, and static error conditions—appears in Appendix A.

Although some of these operations appear syntactically like ordinary functions, they are primitive in the language semantics because they operate over labels, which are not first-class values in Hazel.[2]

3.1.3 Converting Between Tuples and Lists. While the structural operations above operate directly on labeled tuples, many data transformations require dynamically constructing and processing tuple entries. To support these workflows, we provide two primitives—`to_lvs` and `from_lvs`—that convert between labeled tuples and lists of label–value pairs.

`to_lvs` converts a labeled tuple to a list of label–value pairs, making it possible to reuse list-based abstractions when manipulating table rows while preserving label information. `from_lvs` reconstructs a labeled tuple from such a list, reifying labels back into the tuple structure.

These conversions are useful for operations that are awkward or inexpressible with fixed-label primitives, such as dynamically computing which columns to retain, rename, or reorder. `to_lvs` statically summarizes element types by computing their meet type (greatest lower bound), defaulting to the unknown type when inconsistent (see Appendix D). Reconstructing precise tuple types from a list generally requires observing runtime values, so `from_lvs` returns the unknown type statically, deferring to live typing. A complete reference is provided in Appendix C.

3.2 Tables as Sequences

Tables commonly have ordered columns and rows. To reflect this, we represent tables as ordered sequences of labeled tuples. A table therefore has no distinct runtime schema apart from the rows themselves. Our current implementation uses linked lists for simplicity; Section 5.6 discusses more efficient representations. This composition admits natural generalizations: for example, sets of labeled tuples can represent unordered row collections with ordered schemas, as in the relational algebra.

Many table operations, such as filtering and mapping, are naturally expressed in terms of lists of rows. This design unifies row-level operations with general-purpose programming constructs: transformations, aggregations, and selections use the same primitives as the rest of the language rather than a separate domain-specific language.

To accommodate the frequent need for column access, we provide a restricted form of APL-style rank polymorphism [49] that permits label projection over lists of labeled tuples. Thus, an expression of the form `table.label` is equivalent to `map(table, fun row -> row.label)`. This way of lifting field projection over a collection of records also appears in Cω, whose generalized dot maps member access over a stream [7].

[2]Hazel Lab also supports singleton unlabeled tuples using underscore notation (e.g., `(_=1)`); see Appendix B.

From the type system's perspective, the static type of a projection on a list is the projected type of its elements, wrapped in a list. For example, given

```
let rows : [(name=String, age=Int)] = [
  (name="Alice", age=30),
  (name="Bob", age=25),
  (name="Charlie", age=28)
]
```

the projection `rows.age` has type `[Int]` and evaluates to `[30, 25, 28]`—an ordinary list, so standard list operations compose directly with projections, as in `sum(rows.age)`, without a separate abstraction for table columns.

3.3 Live Typing

```
let x  = (if true then 1 else "") : ? in
(x + 1, x ++ "!");
let x  = (if false then 1 else "") : ? in
(x + 1, x ++ "!");
```

Fig. 7. A live-typing error whose location depends on the runtime condition: x is bound to an unknown-typed conditional and used both as an integer and in a string concatenation, and the system marks whichever use disagrees with the observed value.

Live typing extends Hazel's static type checker with type information observed at runtime, surfacing value-dependent errors directly in the editor where static checking alone cannot reach them. It is most useful when a program's types depend on its data.

Many EDA operations [53] and tidy data transformations [58] create and process table headers dynamically (Section 2.4): Wickham notes that "column headers are values, not variable names," and, conversely, several variables are often packed into one column, as in Entity–Attribute–Value and triplestore data [17, 43]. Statically typed languages usually reshape such data through external preprocessing—type providers [50], code generation, or reflection—whereas our `to_lvs` and `from_lvs` (Section 3.1.3) let statically typeable operations be checked while more dynamic ones use strings to build data-dependent schemas.

To address cases where expression types depend on values only present during evaluation, live typing augments static analysis with dynamic types collected during full program evaluation. Static analysis identifies the underspecified expressions, and evaluation records their runtime inhabitants. Once evaluation is complete, we synthesize a dynamic type from each recorded value and compute each expression's *live type*: the *type meet* (greatest lower bound with respect to precision) of its dynamic types, or its unchanged static type when the observations are inconsistent. We then rerun static analysis with live types in place of static ones and mark the errors this second analysis discovers in the editor as secondary error marks. The meet makes each live type more precise than the static type while remaining—up to preservation—consistent with the dynamic types of all runtime expressions that inhabit it. Full motivation and examples are given in Appendix D.

As a first illustration, Fig. 7 shows live typing on a variable used at two incompatible types: it marks whichever use disagrees with the observed runtime value. Placing the mark at the source rather than in evaluation output lessens the spatial aspect of **distal feedback**.

A Worked Example. Fig. 8 traces live typing on a `to_string` function over a dynamic-tagging convention, in which a value is paired with a string naming its intended type. Live typing tracks

```
let sum : [Int] -> Int = ··· in
let to_string = fun (d : (String, ?)) ->
  case d①
  | ("int", v②)   => string_of_int(v)
  | ("bool", v③)  => string_of_int(v)
  | ("nums", v④)  => string_of_int(sum(v))
  | (_, v⑤)       => "Unknown: " ++ v
  end
in
let values : [(String, ?)] =
  [("int",1), ("string","hello"),
   ("bool",true), ("nums",[1.,2.]),
   ("nums",[]), ("unit",())] in
map(values, to_string)
```

(a) The source program

```
let sum : [Int] -> Int = ··· in
let to_string = fun (d : (String, ?)) ->
  case d
  | ("int", v) => string_of_int(v)
  | ("bool", v) => string_of_int(v)
  | ("nums", v) => string_of_int(sum(v))
  | (_, v) => "Unknown: " ++ v
  end
in
let values : [(String, ?)] =
   [("int",1), ("string","hello"), ("bool",true),
    ("nums",[1.,2.]), ("nums",[]), ("unit",())] in
map(values, to_string)
```

```
≡ [
    "1",
    "Unknown: hello",
    string_of_int(true),
    string_of_int(0 + 1.000000:Int + 2.000000:Int),
    "0",
    "Unknown: "
    ++ (())
  ]
```

Γ ⚡ EXP : [Float] inconsistent with expected type [Int]

(b) The editor state and evaluation output

	node	observed	dynamic types	meet	live type
①	d	{("int",1), ("string","hello"),…}	{(String, Int),…}	∅	(String, ?)
②	v in "int"	1	Int	Int	Int
③	v in "bool"	true	Bool	Bool	Bool
④	v in "nums"	{[1.,2.],[]}	{[Float],[?]}	[Float]	[Float]
⑤	v in _	{"hello", ()}	{String, Unit}	∅	?

(c) The trace: each tracked node's observed values, their dynamic types, the meet (∅ when the observations are inconsistent), and the resulting live type

Fig. 8. Live typing on `to_string`: the source program (a); the editor (b), where the marks produced by refining nodes ③ (`string_of_int` on a `Bool`) and ④ (`sum` on a `[Float]`) sit above the evaluation output, which otherwise surfaces the failures only as stuck terms; and the node-by-node trace (c)

every expression and pattern whose static type is not fully known, refining each from the runtime values that reach it. Within `to_string`, those are the parameter `d` (of type `(String, ?)`) and, in each branch, the pattern variable `v` of static type `?` (Fig. 8c). Because `d` is matched on every call it observes all the inputs, whereas each branch's `v` observes only the values routed to that branch. Every branch nonetheless returns a `String`, so `to_string` is statically `(String, ?) -> String`. The result of `map` is statically unknown—Hazel's parametric polymorphism is explicit—so it too is refined by live typing, though we omit it from the trace.

During evaluation Hazel Lab records the runtime inhabitants of each tracked term and synthesizes their *dynamic types*; Fig. 8c traces each node's observations to its live type. Two cases are subtle: at node ④, the meet `[Float]`⊓`[?]` recovers `[Float]` from the empty list's uninformative `[?]`; at nodes ① and ⑤ the observations are mutually inconsistent—no meet exists—so each keeps its static type.

These live types re-enter typechecking: wherever subsumption reaches a pattern or expression whose synthesized type is incomplete, the node's live type refines it, and checking proceeds against the refined type. Refinement only fills in unknown parts, so no declared type is overwritten and

precision never regresses. `d` shows the fallback case: its inconsistent observations yield no refinement, so it is checked at its static type `(String, ?)` exactly as before evaluation, and no new mark can arise there.

Rerunning static analysis with these live types produces two marks (Fig. 8b). At node ③, the `"bool"` branch applies `string_of_int` to a value now known to be a `Bool`, while the `"int"` branch (②) applies the same `string_of_int(v)` yet is unmarked, because its `v` really is an `Int`: the error follows from the observed value, not the syntax, and is invisible to static checking, since `?` is consistent with `Int`. At node ④ the refinement itself has a downstream consequence: `v` is now `[Float]`, but `sum` is declared `[Int] -> Int`, so `v` is marked at the application site—an error that surfaces only once live typing has refined it from `?` to `[Float]`. The catch-all's `"Unknown: " ++ v` stays unmarked because its `v` is `?`; declining to type inconsistent observations avoids a false positive on the `String` it handles correctly, at the cost of not flagging the `Unit` it does not. Note that both marks are visible simultaneously: because evaluation in Hazel does not halt at a failure—failed applications remain as stuck terms while the rest of the program evaluates—a single run surfaces every error live typing discovers, in contrast to exception-based systems, which stop at the first failure and report one stack trace at a time.

Supporting live typing requires three capabilities of the host language: gradual typing, to represent values whose static types are incomplete; a means of instrumenting evaluation to capture the runtime inhabitants of those values; and integration with the static type checker, so the synthesized types feed back to surface new marks. Hazel already provides the first two; the integration is what live typing contributes. This second analysis is ordinary static type checking, so it applies equally to code that did not execute: a value's live type propagates into an unevaluated thunk or an uncalled function body just as into an executed branch, flagging an error before that code is ever run—which a dynamic runtime-error report, confined to code that has executed, cannot do. Hazel suits this design well, as its continuous reevaluation keeps the dynamic type information current and its purity makes the runtime-derived types deterministic and reproducible.

Both static type checking and live typing reduce the spatial distance of the **distal feedback problem** by colocating error marks with the source syntax responsible for them, rather than surfacing failures only in distant evaluation output, stack traces, or logs. For example, the live-typing mark in Fig. 5a identifies the invalid `Unknown` projection at its source rather than leaving the user to work backward from the evaluation output. Static type checking additionally addresses the temporal aspect: it does not depend on execution, so it localizes errors immediately, without waiting for potentially expensive computation. Live typing, by contrast, derives types from observed runtime values, so it extends source-localized feedback to the value-dependent operations that static typing cannot check but does not reduce temporal distance. We therefore claim only partial coverage in Table 3, since temporally-local feedback is available only for statically checkable errors.

Live typing's reliance on full program evaluation is a current limitation: longer-running analyses would delay live-typing feedback until evaluation completes. Lifting this limitation by streaming intermediate evaluation results into the type checker rather than waiting for full execution is future work (Section 5.6.1).

3.4 Rich Probes

We have extended Hazel's live probes [10] for expressions and patterns with projectional interfaces, yielding *rich probes* whose interfaces expose actions derived from their probed values. Rich probe compatibility is determined by the live type of the probed value and therefore is not dependent on the static type of an expression. Inspired by livelits [28], rich probes follow a model–view–update architecture extended with a dynamically observed expression and an action that rewrites the probe's underlying syntax.

The table rich probe, featured in Fig. 3, is applicable to any probed value consisting of a list of fully labeled tuples with identical labels. Once invoked, the list is displayed as a table with the labels as column headers. The available column actions are determined by the live type of the currently viewed value. Consequently, suggested actions reflect the currently selected probed value and may not be compatible with probed values from other invocations. We adopt this design to encourage exploratory, direct manipulation of the currently inspected data rather than restricting actions to those uniformly valid across all probed values.

Actions performed through a rich probe transform the underlying syntax rather than directly modifying the resulting value. This allows users to interact with probed values from a single invocation and observe the effects on the others. When placed on patterns, rich probes currently operate in a read-only mode, as no existing actions have a pattern-specific analogue. Future work may explore specialized actions for rewriting pattern syntax.

3.5 Implementation

The system is implemented as an extension of the open-source Hazel project. Hazel is an integrated editor and language runtime implemented in OCaml using the `js_of_ocaml` compiler to target JavaScript web browsers. The current evaluator is a research prototype: an unoptimized interpreter that runs after every program modification.

4 Evaluation

4.1 B2T2 Benchmark

We use the Brown benchmark for table types (B2T2) [24, 25] to analyze Hazel Lab. B2T2 facilitates comparison of type systems for tabular programming by testing their expressivity in representing common table operations and determining what constraints are enforceable. The benchmark defines a table as a schema—"an ordered sequence of column names and corresponding sorts"—plus a "rectangular collection of cells". Our full implementation of the benchmark including a datasheet is in our artifact [6].

4.1.1 Representation and API. As outlined in Section 3.2, we represent tables as ordinary lists of labeled tuples, with the tuple's type as the schema and its labels the header. This differs from the benchmark's representation in three ways: a table has no standalone, data-level schema, so an empty table carries none; missing data must be tracked explicitly with an `Option` type; and column names, which the benchmark treats as first-class values, are second-class labels in our system, with `to_lvs` and `from_lvs` converting between a labeled tuple and a runtime list of `(string, value)` pairs. Hazel's gradual typing lets these transformations be written without fully specified static types, relying on live typing (Section 3.3) for feedback—a pragmatic compromise in the spirit of gradual typing for expressivity [15].

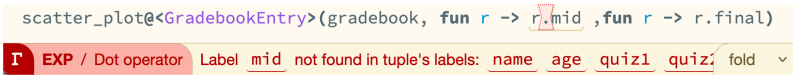
B2T2 documents each operation with *requires* (preconditions) and *ensures* (postconditions) that capture properties a programmer may want enforced. Many of these operations are ordinary list operations, since our tables are lists—appending rows or stacking tables is just concatenation—and are polymorphic over the element type. Lacking row polymorphism, operations that generically alter a schema have no precise type and fall back to the unknown type; the structural operations of Section 3.1 are what make them expressible, as Table 1 quantifies.

4.1.2 Errors. B2T2 also provides a set of buggy programs to help analyze how type systems provide feedback to users in the presence of errors. To improve error reporting we have modified many of the functions in the errors section to take higher-order functions projecting values from labels

Table 1. B2T2 expressibility and constraint enforcement for Hazel Lab, for pre-existing *Hazel*, and for an intermediate ablation *Extension* (defined in text). For each, the columns report how many of B2T2's 49 operation signatures it expresses, and how many of the 72 `requires` and 142 `ensures` it enforces—statically, and for `ensures` also under live typing.

	Expressible	Requires	Ensures (of 142)	
	(of 49)	(of 72)	static	live
Hazel	17	4	29	29
Extension	20	4	30	35
Hazel Lab	49	5	38	96

rather than taking column names as values. This idiom is used only for the error-reporting programs; the counts in Table 1 reflect the benchmark-style API, in which column names are passed as values. Many of the errors are localized as an invalid label being projected out of a tuple as in Fig. 9a.



(a) Invalid label projection

(b) Tables with inconsistent schemas

Fig. 9. Example static error messages

Another common error is a table with an incorrect schema being passed as an argument to a function that expects a different schema. Hazel Lab marks this with an ordinary inconsistent type error and displays both types. An example error message is shown in Fig. 9b.

For programs whose bug the system cannot express statically, the error manifests as a stuck term displayed in the evaluator, a type casting error, or a `None` return value. For example, we do not verify list bound checking statically so using `nth_opt` returns an option type while `nth` results in a stuck term. Notably, Hazel currently requires explicit types for parametric polymorphism, so the user must add type annotations not present in the benchmark programs. These types aid in localizing errors and the live types from Section 3.3 can be used to identify which types to instantiate. The B2T2 error benchmark does not include programs whose schemas are dynamically generated from row values—such as those produced by the benchmark's `pivotWider` operator or the malformed example in Fig. 1. These are precisely the programs where live typing is most effective.

4.1.3 Expressivity and Constraint Enforcement. To isolate the expressivity the structural operations of Section 3.1 contribute, we implemented the full B2T2 Table API for Hazel Lab and two ablations of it (Table 1). *Hazel* provides only the pre-existing labeled-tuple types, literals, and row projection (`t.ℓ`), with none of Hazel Lab's operations; *Extension* additionally provides tuple extension (`t1 ... t2`);[3] and Hazel Lab additionally provides the `to_lvs` and `from_lvs` conversions between a row and a list of label–value pairs, from which the remaining label operations of Section 3.1 are derivable. *Hazel* expresses 17 of the 49 signatures—the *column-agnostic* operations, which name no column. *Extension* adds three more, which concatenate or merge whole rows using `t1 ... t2`. Hazel Lab can at least partially express all 49: `to_lvs` and `from_lvs` unlock the

[3] *Extension* also includes broadcast projection over tables (Section 3.2); this affects neither the expressibility nor the static-guarantee results in Table 1.

29 *first-class-label* operations—those that take, return, or enumerate column names as runtime values—none of which is expressible without them (four only under a side condition, e.g. `leftJoin` needs a row to recover the right table's schema).

This expressivity costs static guarantees: because `to_lvs` and `from_lvs` thread runtime column names through the unknown type, the first-class-label operations enforce few of their `ensures` statically, and the static count reaches only 38 of 142. Live typing closes much of the gap—run to completion on a table of one concrete schema, the result is re-typed to a concrete labeled-tuple type, making postconditions about its schema, header, and column sorts checkable and raising enforced `ensures` to 96 of 142 (for the 29 first-class-label operations alone, from 8 to 61 of 89). Live typing cannot recover the `requires`, which constrain inputs rather than the result, or cardinality postconditions such as row counts, which the type system does not track.

4.1.4 Takeaways. Our list-of-labeled-tuples representation can express almost all of B2T2. Many operations are ordinary list and functional operations, since a table is just a list. When operations take column names as first-class values—which our labels are not—we represent the names as strings, and few of these operations' constraints are checkable statically. This is the price of keeping data-dependent schemas expressible without a richer type system: row or dependent types could recover more guarantees ahead of time, but at the cost of the simplicity and flexibility the representation is designed for. Live typing offers another route to those guarantees: by re-typing finished results, it recovers much of the lost `ensures`—the postconditions on a result's schema and header, not the `requires` on its inputs—after execution, and without adding any machinery to the type system.

4.2 User Study

We conducted a user study to evaluate how effectively users can perform data manipulation tasks using our table representation and API, and to gather qualitative feedback on their experience.

We considered the following research questions:

- **RQ1**: Can users use Hazel Lab's table operations to do basic data transformation and data cleaning tasks?
- **RQ2**: Can users effectively use rich probes in data cleaning and transformation tasks? What problems do users face when trying to use rich probes?
- **RQ3**: Do users find live typing errors useful in identifying programming errors? Do users find live typing helpful when trying to understand program behavior?

4.2.1 Participants. Participants were recruited with self-reported experience with functional or expression-oriented and statically typed programming languages by posting on Bluesky and X offering compensation of $50 for a 2-hour session. Our study had 7 participants (3 male, 3 female, 1 declined to answer; 4 professional programmers, 3 students); 9-30 years of programming experience ($\mu = 14$). Participants were also screened to have a familiarity with the list functions `map` and `fold_left` as they are used extensively through the tasks.

4.2.2 Procedure. Each two-hour session was conducted remotely with screen recording, and consisted of a guided background tutorial on Hazel's language and interface before introducing the features under evaluation. Participants were encouraged to ask questions throughout. Tutorials for new features (labeled tuples, tables, rich probes, and live typing) were interspersed through tasks to better isolate feedback to the specific features under evaluation.

There were six tasks in total (Table 2):

- Task 1 is a baseline programming task in Hazel that does not use any newly introduced features, serving to assess each participant's ability to program in Hazel.

Table 2. Study tasks

Task	Type	Description
1	Programming	Compute the mean of a list of strings representing integers
2	Programming	Compute mean of gradebook midterms
3	Programming	Tidy gradebook and add `overall_grade` column
4	Programming	Tidy gradebook `term` column
5	Bug identification	Pearson correlation between student age and final exam score
6	Bug identification	Weighted mean of overall course grade per year and semester

- Task 2 is a simple data analysis task designed to evaluate whether participants are comfortable with the table representation and can process tabular data (**RQ1**).
- Tasks 3 and 4 both involve data tidying through type-conversions and the introduction of new columns, evaluating the usability of rich probes in data cleaning and transformation (**RQ2**).
- Tasks 5 and 6 are bug identification tasks where participants are given a buggy program and asked to verbally (1) describe what the program is intended to compute, (2) identify the bug(s), and (3) propose a fix for each bug found (**RQ3**).

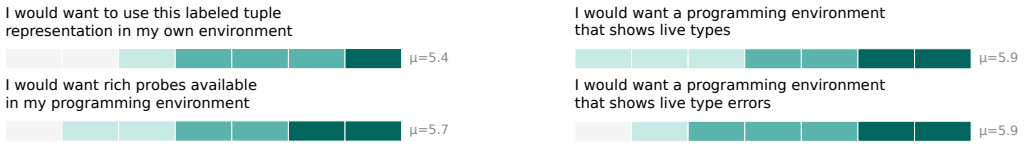
The order within each task pair (3/4 and 5/6) is counterbalanced across participants, with the evaluated feature enabled on the latter task of each pair: rich probes for Tasks 3/4, and live typing for Tasks 5/6, letting participants see live type errors and the live type of the term under the caret. Both bug identification tasks have a 10-minute time limit and are presented in a read-only editor, though participants may move the cursor throughout the program to inspect the typing information of different terms and observe the program output, where type casting failures are visible in the resulting values.

The bug identification programs are designed to be realistic scenarios that involve a straightforward analysis component and a data tidying component that would be non-trivial to statically type. For example, in Task 5 a `students_vertical` dataset is provided exhibiting what Wickham [58] calls "multiple variables stored in one column," and the program performs what Wickham refers to as an *unstack* operation to transform a long table into a wide one. Static type checking shows no errors for either program, but both produce live typing errors.

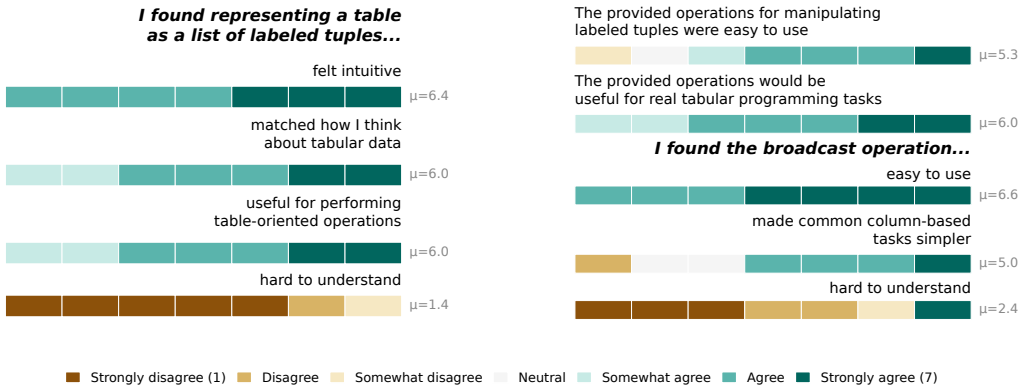
4.2.3 Results. Participant assessments of feature desirability are summarized in Fig. 10a; every response to the adoption-intent questions was at least neutral, and the majority were positive. Additional participant feedback appears in Appendix E.

RQ1: Table Representation and Operations. As shown in Fig. 10b, participants found the labeled tuple representation intuitive and consistently disagreed that it was "hard to understand." Participants valued structural typing, which eliminates the need to declare new types for each transformation. P1 noted that type inference "just works" without requiring types for all variations of rows, and P6 similarly appreciated not needing a new datatype for each transformation. P4, however, expressed a preference for unordered labels and asked about partial type specification (row typing). P4 further remarked on the tension between strong typing and the table API, noting that while live typing helps recover precise types, it breaks down when unknown types arise from columns with inconsistent dynamic types.[4] P7 wished for first-class labels and greater reflective capabilities beyond `to_lvs`/`from_lvs`.

[4]At the time of the study, live typing fell back to the unknown type when dynamic observations were inconsistent; it has since been changed to fall back to the expression's static type (Section 3.3). The study examples did not meaningfully exercise this difference.



(a) Adoption-intent questions across all three research questions



(b) Table representation and labeled tuple operations (**RQ1**)

Fig. 10. Exit survey responses

P2 noted that the broadcasting projection operator felt "even more declarative than normal functional programming," but raised concerns about accidental misuse because the same syntactic form is used whether the operand is a single labeled tuple or a list of labeled tuples, and this ambiguity is especially problematic when the operand type is unknown. P4 similarly felt that broadcasting was too table-specific, expressing a desire for more general rank-polymorphic support.

P7 praised the tuple extension operator (. . .) as "nice, both semantically and syntactically" and easier to use than equivalents in other languages. However, P2 expressed concern that the same operator is used for both type-preserving updates and type-changing extensions, noting that this "seems like it could introduce hard-to-see bugs."

Overall, the representation was well-received, but participants identified areas for improvement: partial type specification (row typing), more reflection, syntactic disambiguation for broadcasting between single tuples and lists, and separating type-preserving updates from schema-altering extensions.

RQ2: Rich Probes. As shown in Fig. 11 (left), the highest-rated aspect of rich probes was their utility for understanding intermediate values. Participants responded positively to rich probes as a way to perform data transformations through a visual interface while preserving textual source code as the source of truth. P1 remarked: "Good, so the textual source code stays the source of truth, but we have these fancy UIs to manipulate that without having to remember how to map over the lists." P2 drew a comparison to spreadsheets, noting that unlike Excel where type conversions are invisible and not preserved in the file format, rich probes make transformations explicit in source code. However, responses were more divided on whether rich probes helped participants work faster. P3 and P6, who used rich probes during the tutorial, did not apply them on the data-transformation task where the probes were available: P6 did not find it obvious how to apply

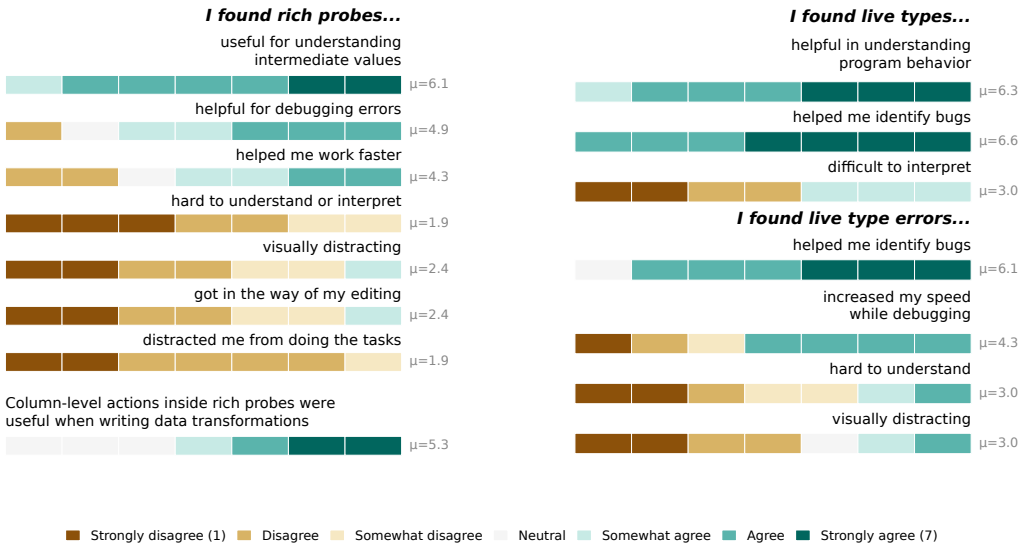


Fig. 11. Exit survey responses for rich probes (**RQ2**) and live typing (**RQ3**)

rich probe actions to Task 4, and P3 did not use probe actions despite expressing that they were useful in the tutorial. This suggests the speed benefit may depend on whether participants used probe actions during the tasks or on differences in existing programming familiarity. Separately, P3 remarked that they were "so in on the widget, I didn't notice it was emitting code for me," highlighting a disconnect between probe interactions and awareness of the generated source code.

Overall, rich probes bridge the gap between direct manipulation interfaces and programmatic data cleaning. P2 captures the core value proposition, comparing the experience to Excel while noting that "the process gets defined": rich probes offer the discoverability and immediacy of a graphical interface while maintaining a textual program as the authoritative representation. This addresses a fundamental limitation of spreadsheet-based data cleaning, where transformation steps are often invisible and non-reproducible. However, the code-generation model created a disconnect for some participants who were focused on the probe widget and did not notice changes to the source code, suggesting that tighter coupling between probe interactions and source navigation may be needed.

RQ3: Live Typing. As shown in Fig. 11 (right), all seven participants found live types helpful for understanding program behavior and for identifying bugs. P5 found that live typing helped keep track of tuple shapes and fix type errors quickly, and P1 noted that live types provide a way to recover from the unknown type. During Task 5 with live typing enabled, P4 reacted: "Wow. Look at that! Live typing has just given us the type, so I can… don't even have to read this." P2 said "Oh, that's helpful" upon hovering over a variable and seeing its live type, and reported that "without even looking at the code first, just looking at the type and using the name of the variable" they were able to determine the purpose of a piece of code. After completing Task 6 with live typing enabled, P3 turned the feature off to compare and remarked: "Yeah, it would be impossible to spot that needle in this haystack of code in the given time frame." However, responses to "difficult to interpret" were split. P3, P5, and P7 found that having to move the cursor to inspect types was cumbersome, and P7 suggested "an option to display certain types inline, such as for function signatures and variable types." Live type errors showed a similar pattern: while most participants

agreed they helped identify bugs, responses to "increased my speed while debugging" were mixed. This may be partly attributable to Hazel's error and type display mechanisms in general, which some participants found difficult to use independently of the live typing features.

Overall, live types provided genuine utility for program understanding and debugging: participants reasoned about code from type information alone, using live types as lightweight, real-time documentation. That a participant could determine a variable's purpose "without even looking at the code first" suggests live types can serve as an effective entry point for understanding unfamiliar code, particularly in data analysis programs where the shape of the data is central to comprehension.

4.2.4 Threats to Validity.

Internal Validity. The within-subjects design for rich probes (Tasks 3/4) and live typing (Tasks 5/6) introduces potential order effects: participants may perform better on later tasks due to increased familiarity with Hazel rather than the features being evaluated. We mitigate this by counterbalancing task order across participants, though feature enablement is not counterbalanced—the feature is always enabled on the latter task in each participant's ordering. This means we cannot fully disentangle improvement due to learning from improvement due to the feature. Additionally, the tutorial segments preceding each task group may prime participants toward the evaluated features. The study was also limited to a 2-hour session in which participants had to learn both Hazel and Hazel Lab's new features through a tutorial before completing tasks. This compressed timeline limited the fluency participants could develop, and the tutorial quality may influence performance independently of the features being evaluated.

External Validity. Our participant pool was recruited from social media and screened for experience with functional programming and static typing, since Hazel is an expression-oriented, statically typed functional language and recruiting without this background would have required substantially more tutorial time. These seven participants self-selected and were likely already favorably disposed toward typed functional programming. This study should therefore be read narrowly: it offers evidence that the evaluated features are usable and useful *for programmers already fluent in statically typed functional programming*, not that the proposal is usable in general or that its benefits transfer to the broader population of data analysts—many of whom work primarily in dynamically typed languages such as Python or R—who lack that background. We read its positive results as evidence that, once Hazel Lab is production-ready, investing in the requisite background could be worthwhile for a working scientist, rather than as a general usability claim. Whether lively typed tables help data scientists without a typed functional background is left to future work. The tasks, while designed to reflect realistic data tidying scenarios, are smaller in scope than real-world workflows, which may involve larger datasets, more complex schemas, and iterative exploration over longer time horizons.

Construct Validity. Participants were working in an unfamiliar environment and encountered usability challenges unrelated to the features under evaluation. Hazel's error reporting was an unfamiliar interaction model, and its partial application syntax required additional learning time and posed challenges during the tasks. These incidental difficulties make it harder to attribute observed performance and survey responses solely to the evaluated features.

The bug identification tasks (Tasks 5/6) use a read-only editor without access to probes, preventing participants from experimenting with fixes or using probes for debugging—a strategy they would normally employ. P4 noted that live typing may not have been necessary with probe access, suggesting our study may overestimate live typing's importance relative to a realistic setting. Hazel does not provide stack traces; because errors do not halt execution, participants must locate

failures by inspecting output values and type information rather than being directed to the source. The bug-identification tasks (the latter with live typing enabled) are where this *spatial* aspect of the **distal feedback problem** is exercised—live typing marks the offending source location rather than leaving participants to work backward from output—but these programs are small and do not reflect the scale of a real pipeline in which an error introduced early surfaces far downstream. We do not evaluate the temporal aspect here: the study programs each run in under five seconds, so waiting on evaluation does not arise.

Our exit survey relies on self-reported Likert responses, which may be subject to acquiescence or social desirability bias, particularly given the researcher-participant interaction during the study.

5 Related and Future Work

Tabular programming systems (TPSs) vary along multiple dimensions—including the editor or notebook interface, the programming language and runtime, and the table library. We first evaluate the problems from Section 1 (summarized in Fig. 12) across a set of illustrative TPSs, then discuss related work on individual components of our system.

5.1 Comparison with Illustrative TPSs

Table 3 compares four representative systems—Scala with a conventional IDE and Frameless [54], Polynote [39] (a Scala notebook), Jupyter with Pandas [51], and Hazel Lab. Classifications are necessarily approximate: a partial designation indicates that a system mitigates a problem in common scenarios but leaves important limitations unaddressed. Detailed per-system analysis is provided in Appendix F.

Table 3. Comparison of illustrative tabular programming systems by the extent to which they address the problems outlined in Fig. 12. **Legend:** ● = fully addresses or mitigates the problem; ◑ = partially addresses; ○ = does not address.

Problem	Scala w/ IDE	Polynote (Scala)	Jupyter + Pandas	Hazel Lab
Function barrier problem	●	◑	○	●
Distal feedback problem	◑	◑	○	◑
External schema problem	◑	◑	◑	●
Expressivity problem	◑	◑	◑	●
Out-of-order execution	●	◑	○	●

5.2 Labeled Tuples

Hazel's labeled tuples, over which our structural operations are defined, most closely match labeled tuples in OCaml (available since OCaml 5.4) [12]. In OCaml, labeled tuples are dual to labeled function arguments; in Hazel, all multi-argument functions take tuples, so labeled arguments are naturally labeled tuple entries. Both systems are order-aware, structurally typed, and disallow duplicate labels. Our primitive tuple operations are an order-preserving variant of record operations such as in Cardelli and Mitchell [11]; similar operations exist for heterogeneous lists [21].

Problem definitions.

- **Function barrier problem** — code written inside functions lacks type-directed services such as autocomplete, inline error highlighting, and refactorings.
- **Distal feedback problem** — error feedback is distant both spatially, where runtime failures are reported far from the offending syntax (e.g., via stack traces, cell outputs, or logs), and temporally, where type-directed services require running expensive or long-running computations, forcing analysts to wait for feedback.
- **External schema problem** — when data are ingested and parsed at runtime, the table's schema is unavailable to static analysis, preventing type-directed services unless schemas are explicitly provided.
- **Expressivity problem** — common EDA transformations involving structural changes to tables are difficult to express statically without advanced type-system features or metaprogramming. In the hardest cases, a table's structure depends on values unknown until runtime, hindering static verification of downstream operations.
- **Out-of-order execution** — editor services depend on mutable execution history rather than program structure, leading to brittle, misleading, or non-reproducible feedback.

Fig. 12. Problems considered when comparing tabular programming systems

5.2.1 Row Polymorphism for Labeled Tuples. Our system does not support row-polymorphism for labeled tuple operations and instead defers to gradual typing's unknown type when a principal type cannot be statically inferred. Supporting row types [44–46, 55, 56] would allow many of these operations to have principal types, enabling pipelines to be typed over many different schemas without resorting to the unknown type. This lessens the **distal feedback problem**—particularly its temporal aspect, since feedback no longer waits on evaluation—and makes more code safe regardless of the caller. Our live typing approach provides a pragmatic alternative: rather than requiring richer static types, it recovers type information from runtime observations while preserving the ability to execute programs without fully specifying all types. Any integration of row types should maintain this property.

5.2.2 First-Class Labels and Dependent Types. First-class labels or dependent types would allow programmers to express stronger invariants about table structure and column relationships. Dependent types have been used to encode extensible records and heterogeneous lists [57]; a similar formulation could encode labeled tuples. Dependent types could also improve performance by inspecting schema metadata rather than fully executing the program. However, integrating dependent types with gradual typing remains an active area of research [14, 23, 26]. Our live typing approach retains a simple type system while detecting errors that other simply typed systems miss. Match types "enable a lightweight form of dependent typing" [9]: Scala 3 leverages them to implement structural operations on unlabeled tuples. Scala 3.5 Named Tuples [27] add labels to tuple entries; unlike our system, Named Tuples are treated as subtypes of their unnamed counterparts.

5.3 Table Representations and Types

R and Pandas data frames are dynamically typed and column-oriented, requiring explicit conversion into lists of rows; our system is row-oriented. Data frames also carry row indices—in Pandas, including hierarchical multi-indices—which our representation does not model. Index values could be encoded as ordinary labeled entries, but efficient keyed access would depend on the more efficient representations discussed in Section 5.6.

Lifting labeled tuple projection to lists of labeled tuples is inspired by APL's rank polymorphism. Remora integrates ML-style records with rank polymorphism and static typing [48, 49].

MODIN [33] is a drop-in replacement for Pandas built on a core set of dataframe operations. It includes schema induction to infer column types by parsing data, whereas our system uses ordinary type inference for literal tables and live typing (Section 3.3) for dynamically created schemas.

MODIN likewise distinguishes schema-preserving operations from those requiring inference, paralleling our distinction between precise types and the unknown type. MODIN's opportunistic evaluation, which passes control back to the user while incremental evaluation continues, could be adapted to our system as future work.

Hazel Lab currently brings external data into an analysis by embedding it in the program source: the CSV loader (Fig. 2) parses a file into an ordinary table expression whose inferred type serves as its schema, and for data that is not yet available, a static type annotation describes the expected schema, providing type-directed feedback before any data arrive. Writing such annotations by hand is tedious for wide tables: a `typeof` operator, as in TypeScript, could derive a static type from an existing dataset—akin to type providers [50] and MODIN's schema induction, but within the language rather than a preprocessing step—so that a dataset's schema can be named as a type alias and reused to annotate functions or describe expected data.

SQL requires declaring a schema before processing data, making exploratory analysis difficult [33]. Unlike our tables, SQL makes no guarantees about row ordering and relies on normalization rather than composition to resolve hierarchical structures.

Dependent types have been used to encode tables in Lean [42] and Idris [60], both analyzed using the B2T2 benchmark [24, 25].

5.4 Live Typing

5.4.1 The Gamma. The Gamma [34, 36] extends Type Providers [37] to support "dot-driven" data exploration by lazily generating member methods from available data in a pipeline. This mechanism is similar to our live typing but is limited to linear pipelines and provides no typing information without concrete values. Our system additionally works across multiple invocations by computing the type meet over all observed values.

5.4.2 Dynamic Type Inference. RubyDust [2] infers types for Ruby programs by executing them and collecting usage constraints on runtime values. Our system instead reports types synthesized from data present during execution, surfacing potential operations not exposed by current usages, and is integrated into ordinary static type checking so it applies only where static inference is insufficient. RubyDust also requires exercising every path through each method body for completeness, whereas our approach observes the program as currently edited and is thus intrinsically complete. Type-hole filling [61] is a complementary static approach to collecting usage constraints.

Histogram [35] represents programs as lists of user interactions rather than syntax. It permanently ties function types to the concrete values they were first applied to, which ensures stability but limits generalization; our system instead aggregates observed values across all invocations to build general types. We compare Histogram's editing and probe aspects in Section 5.5.2.

5.4.3 Occurrence Typing. Live typing of user-defined type tagging (Fig. 8) superficially resembles occurrence typing in Typed Scheme [52], where type predicates narrow variable types in conditionals. Our system similarly allows multiple occurrences of the same variable to have different types, but the variation arises from dynamic observations rather than explicit predicates. Whereas occurrence typing is fully static, our live typing relies on evaluation, enabling it to detect implicit type flow as in Fig. 7.

5.4.4 Live Environments. REPLs [32] and Jupyter Notebooks provide live tab-completion by introspecting runtime objects in their current state, so feedback is only valid at that point in execution. Obtaining feedback for a variable defined earlier requires re-executing its code, leading to out-of-order execution issues. Our system instead tracks expression types throughout execution.

5.4.5 Live Typing in Smalltalk. Cuis Smalltalk [18, 59] uses runtime execution to infer types for instance variables and method return values, extended with a live type checker that detects incompatible method calls. That system records the set of observed classes per variable or return site, whereas our approach records live types at any expression and maintains only their type meet, ensuring live types remain representable within Hazel's ordinary type system. Because Cuis supports mutable variables, a single variable may accumulate values of different types across assignments, producing false-positive errors; Hazel's immutable bindings and expression-level granularity avoid this. Cuis also filters possible classes within conditionals to avoid impossible errors [31], while our system observes live types at any evaluated expression without special cases. Our live types are recorded only where static typing cannot fully determine a type, and live probe pinning allows selectively filtering dynamic observations.

5.4.6 User Control over Live Typing. Our current implementation allows the user to enable/disable live typing for the entire program. Since Hazel uses the unknown type as its general notion of type hole, this applies live typing to every incomplete type. Future work could provide finer-grained control, letting analysts opt into live typing for only value-dependent operations or mark parts of the program to be typed statically. One appealing approach is a distinct `Dyn` type in the surface syntax, signaling that an expression's type should be inferred by live typing.

5.5 Rich Probes for Tables

5.5.1 Livelits. Livelits [28] provide user-definable, composable [16] GUIs for filling expression holes by expanding to a literal of a fixed, statically known model type, guaranteeing the generated literal fits the surrounding typing discipline. Our CSV data loader is effectively a livelit, but CSVs produce tables with varying schemas that cannot be expressed under the fixed expansion type without falling back to the unknown type. Rich probes are inspired by livelits but focus on interacting with dynamic values rather than constructing literals: they operate on arbitrary probed expressions, translating user interactions into AST transformations. They are not yet user-definable or compositional; making rich probes extensible and reconciling them with livelits is future work.

5.5.2 Rich Direct Manipulation TPSs. The Gamma's [36] exploratory editor supports interactive data transformation and table visualization through direct manipulation, but relies on dynamically generated methods via type providers for a linear sequence of method calls. Our system instead uses live typing of expressions and drives the editor entirely from type information, allowing it to operate with unknown values or across multiple datasets by pinning invocations.

Histogram [35] includes a rich table editor with similar actions, but records them as explicit interactions rather than code. Because it permanently attaches functions to their initial argument, intermediate values reflect only that original invocation; our probes observe intermediate values for every invocation. It also enforces an append-only model where programs cannot be edited directly and changes to functions do not affect existing callers, whereas Hazel Lab supports direct manipulation at arbitrary points, reflecting edits across all relevant datasets.

Mage [20] adds interactive GUIs to notebooks, synchronizing GUI interactions with notebook cell code. Its table tool resembles our projectional editor but must be summoned explicitly on a variable bound to data—typically at notebook global scope—preventing speculative use and inspection of intermediate values inside functions. Our probe interface attaches speculatively to arbitrary expressions or patterns, providing type-directed visualizations without restructuring code.

5.6 Performance and Scalability

5.6.1 Incrementality across Typing and Evaluation. Live typing currently waits for whole-program evaluation before feeding observations back into static typechecking (Section 3.3). Streaming intermediate evaluation results into the type checker as they are produced would surface live-typing marks during long-running computations rather than after them, though how to present partially observed live types to the user requires further design. Recent work on incrementalizing bidirectional typechecking could speed up static typechecking in our system [40]. A unified incremental framework across static typing, evaluation, and live typing would more closely match the responsiveness of notebook-style environments. MODIN-style opportunistic evaluation could prioritize visible computations while background evaluation continues, and pipelined operations could be fused to reduce redundant scans.

5.6.2 Efficient Table Representation. Our current table representation prioritizes simplicity over performance. Future work could adopt more efficient layouts such as persistent vectors, struct-of-arrays, or columnar formats like Apache Arrow [4] or Parquet [5], while preserving the row-oriented labeled tuple abstraction.

5.6.3 Foreign Data and Offloaded Computation. Datasets must currently be embedded in the program source (Fig. 2): the CSV loader copies the full dataset into the program text, so the editor, type checker, and evaluator all process it on every reevaluation, which does not scale to large data. A *foreign data interface* would keep the data in external storage while still presenting it as an embedded value—typed, probeable, and displayed like an ordinary table expression. Program text and typechecking would then no longer grow with the data, evaluation could materialize only the rows and columns a computation or probe actually demands, and schemas and live types of unchanged external data could be cached across reevaluations. To preserve Hazel's purity—and with it the determinism and reproducibility of live types—the interface would need snapshot semantics, treating external data as an immutable versioned value rather than a mutable resource.

Once the data live externally, computation can move to the data. Projections and filters could be pushed down to columnar stores, and suitable pipelines could be JIT-translated into programs for a distributed engine such as Apache Spark, leveraging its distributed execution, optimization, and fault tolerance. Offloading can be opportunistic: pipelines admitting efficient external execution are offloaded, while the Hazel evaluator handles cases that depend on live interaction.

6 Conclusion

This paper presents Hazel Lab, a tabular programming system that extends Hazel's gradual typing discipline with live typing. Live typing alleviates the spatial aspect of **distal feedback** using runtime observations to type value-dependent table operations, while Hazel's ordinary static type checking surfaces conventional type errors without waiting for evaluation.

We introduce statically typed structural operations on labeled tuples, letting common data tidying and presentation patterns be written and checked with best-effort static types even in the absence of data. A CSV data loader addresses the **external schema problem**: the imported table's inferred type serves as its schema, reducing schema validation to Hazel's existing type checking.

Finally, rich probes for tables, powered by live typing, enable multi-modal programming that combines direct manipulation of tabular data with textual code. We evaluate expressiveness and error localization using the B2T2 benchmark. In our user study ($n = 7$), participants found labeled tuples intuitive, rich probes effective at bridging direct manipulation and programmatic data cleaning, and live types useful for program understanding and debugging.

Data Availability Statement

The artifact accompanying this paper is archived on Zenodo [6]. It includes the study materials given to participants in the user study, our implementations of the B2T2 benchmark and the ablations reported in Table 1, the appendix (Appendices A–F) referenced in the body of the paper, and the full source code with build instructions. It provides two pre-compiled versions of Hazel Lab, runnable in a Chromium-based web browser: the version used in the user study, and the current version, whose live typing falls back to the static type when observations are inconsistent (Section 3.3).

Acknowledgments

We thank Anthony Li for work on the initial implementation of labeled tuples in Hazel. We are also grateful to Andrew Blinn for his work on the implementation of probes, which made rich probes possible, and for helpful feedback about rich probes, and to Matthew Keenan and Thomas J. Porter for helpful conversations on labeled tuples. We thank the anonymous reviewers for their insightful feedback, which substantially improved the paper.

This material is based upon work supported by the National Science Foundation CISE Graduate Fellowships under Grant No. 2313998 and by the National Science Foundation under Grant No. 2238744. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

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. *Proceedings of the ACM on Programming Languages* 9, POPL (Jan. 2025), 2176–2204. doi:10.1145/3704909

[2] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. 2011. Dynamic inference of static types for Ruby. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '11)*. Association for Computing Machinery, New York, NY, USA, 459–472. doi:10.1145/1926385.1926437

[3] Apache Foundation. 2025. Apache Spark. https://spark.apache.org/

[4] Apache Software Foundation. 2026. Apache Arrow. https://arrow.apache.org/

[5] Apache Software Foundation. 2026. Parquet. https://parquet.apache.org/

[6] Alexander Bandukwala and Cyrus Omar. 2026. Artifact for Interactive Data Analysis with Lively Typed Tables. doi:10.5281/zenodo.21458220

[7] Gavin Bierman, Erik Meijer, and Wolfram Schulte. 2005. The Essence of Data Access in Cω: The Power is in the Dot!. In *ECOOP 2005 – Object-Oriented Programming*, Andrew P. Black (Ed.). Springer, Berlin, Heidelberg, 287–311. doi:10.1007/11531142_13

[8] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. 2022. The art and practice of data science pipelines: A comprehensive study of data science pipelines in theory, in-the-small, and in-the-large. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2091–2103. doi:10.1145/3510003.3510057

[9] Olivier Blanvillain, Jonathan Immanuel Brachthäuser, Maxime Kjaer, and Martin Odersky. 2022. Type-level programming with match types. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–24. doi:10.1145/3498698

[10] Andrew Blinn. 2025. *Structured Semantic Context for Programming Processes.* Proposal Prospectus. University of Michigan. https://andrewblinn.com/papers/Thesis-Proposal-Andrew%20Blinn-December-2025.pdf

[11] Luca Cardelli and John C. Mitchell. 1991. Operations on records. *Mathematical Structures in Computer Science* 1, 1 (March 1991), 3–48. doi:10.1017/S0960129500000049

[12] Chris Casinghino and Ryan Tjoa. 2024. Labeled Tuples. In *Proceedings of Higher-order, Typed, Inferred, Strict: ML Family Workshop 2024*. ACM, Milan, Italy, 1–3. https://tyconmismatch.com/papers/ml2024_labeled_tuples.pdf

[13] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. 2020. What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3313831.3376729

[14] Joseph S Eremondi. 2023. *On The Design of a Gradual Dependently Typed Language for Programming*. Ph. D. Dissertation. University of British Columbia. doi:10.14288/1.0428823

[15] Michael Greenberg. 2019. The Dynamic Practice and Static Theory of Gradual Typing. In *3rd Summit on Advances in Programming Languages, SNAPL 2019, Providence, RI, USA, May 16-17, 2019 (LIPIcs, Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:20. doi:10.4230/LIPICS.SNAPL.2019.6

[16] Joshua Horowitz and Jeffrey Heer. 2023. Live, Rich, and Composable: Qualities for Programming Beyond Static Text. In *13th Annual Workshop on the Intersection of HCI and PL (PLATEAU 2023)*. doi:10.48550/arXiv.2303.06777 arXiv:2303.06777 [cs].

[17] Stephen B. Johnson. 1996. Generic Data Modeling for Clinical Repositories. *Journal of the American Medical Informatics Association* 3, 5 (Sept. 1996), 328–339. doi:10.1136/jamia.1996.97035024

[18] Juan Vuletich. 2009. Cuis-Smalltalk. https://cuis.st/

[19] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018*, Regan L. Mandryk, Mark Hancock, Mark Perry, and Anna L. Cox (Eds.). ACM, 174. doi:10.1145/3173574.3173748

[20] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. ACM, Virtual Event USA, 140–151. doi:10.1145/3379337.3415842

[21] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*. ACM, Snowbird Utah USA, 96–107. doi:10.1145/1017472.1017488

[22] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and power in academic publishing: Players, agents and agendas*. IOS press, 87–90. doi:10.3233/978-1-61499-649-1-87

[23] Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. 2022. Gradualizing the Calculus of Inductive Constructions. *ACM Trans. Program. Lang. Syst.* 44, 2 (April 2022), 7:1–7:82. doi:10.1145/3495528

[24] Kuang-Chen Lu, Ben Greenman, and Shriram Krishnamurthi. 2021. B2T2: Brown Benchmark for Table Types (Version 1.0). doi:10.5281/ZENODO.5507463

[25] Kuang-Chen Lu, Ben Greenman, and Shriram Krishnamurthi. 2022. Types for Tables: A Language Design Benchmark. *The Art, Science, and Engineering of Programming* 6, 2 (2022), 8:1–8:30. doi:10.22152/PROGRAMMING-JOURNAL.ORG/2022/6/8

[26] Kenji Maillard, Meven Lennon-Bertrand, Nicolas Tabareau, and Éric Tanter. 2022. A reasonably gradual type theory. *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 931–959. doi:10.1145/3547655

[27] Martin Odersky. 2024. Named Tuples. https://docs.scala-lang.org/sips/58.html

[28] 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 (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 511–525. doi:10.1145/3453483.3454059

[29] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live functional programming with typed holes. *Proc. ACM Program. Lang.* 3, POPL (2019), 14:1–14:32. doi:10.1145/3290327

[30] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: a bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 86–99. doi:10.1145/3009837.3009900

[31] Julián Francisco Gutiérrez Ostrovsky. 2024. *TypeCheckerDragon: Chequeo de Tipos Contextualizados en un Lenguaje de Tipado Dinámico*. Tesis de Licenciatura en Ciencias de la Computación. Universidad de Buenos Aires, Buenos Aires. https://bibliotecadigital.exactas.uba.ar/download/seminario/seminario_nCOM000841_GutierrezOstrosky.pdf

[32] Fernando Perez and Brian E. Granger. 2007. IPython: A System for Interactive Scientific Computing. *Computing in Science & Engineering* 9, 3 (May 2007), 21–29. doi:10.1109/MCSE.2007.53

[33] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. 2020. Towards scalable dataframe systems. *Proc. VLDB Endow.* 13, 12 (July 2020), 2033–2046. doi:10.14778/3407790.3407807

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

[35] Tomas Petricek. 2019. Histogram: You have to know the past to understand the present. http://tomasp.net/histogram
[36] 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*, Paolo Bottoni, Gennaro Costagliola, Michelle Brachman, and Mark Minas (Eds.). IEEE, 1–7. doi:10.1109/VL/HCC53370.2022.9833134
[37] 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 (PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 477–490. doi:10.1145/2908080.2908115
[38] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, Montreal, QC, Canada, 507–517. doi:10.1109/MSR.2019.00077
[39] Polynote Contributors. 2025. Polynote. https://polynote.org/
[40] Thomas Porter, Marisa Kirisame, Ivan Wei, Pavel Panchekha, and Cyrus Omar. 2025. Incremental Bidirectional Typing via Order Maintenance. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 1865–1892. doi:10.1145/3763117
[41] R Core Team. 2021. R: A Language and Environment for Statistical Computing. https://www.R-project.org/
[42] Joseph Rotella. 2024. *Dependently Typed Tables.* Bachelor of Science Thesis. Brown University, Providence, Rhode Island. https://cs.brown.edu/media/filer_public/b8/d7/b8d70bb1-9c0e-467f-aec6-aaf42019f169/rotellajoseph.pdf
[43] Jack Rusher. 2003. Rhetorical Device: Triple Store. https://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/rusher.html
[44] D. Rémy. 1989. Type checking records and variants in a natural extension of ML. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89*. ACM Press, Austin, Texas, United States, 77–88. doi:10.1145/75277.75284
[45] Didier Rémy. 1992. Typing record concatenation for free. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '92)*. Association for Computing Machinery, New York, NY, USA, 166–176. doi:10.1145/143165.143202
[46] Didier Rémy. 1994. Type inference for records in a natural extension of ML. In *Theoretical aspects of object-oriented programming: types, semantics, and language design*. MIT Press, Cambridge, MA, USA, 67–95. https://dl.acm.org/doi/10.5555/186677.186689
[47] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Seventh Workshop on Scheme and Functional Programming*. University of Chicago, Portland, OR, USA, 81–92. http://ecee.colorado.edu/~siek/pubs/pubs/2006/siek06_gradual.pdf
[48] Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An Array-Oriented Language with Static Rank Polymorphism. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 27–46. doi:10.1007/978-3-642-54833-8_3
[49] Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2019. Records with rank polymorphism. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. ACM, Phoenix AZ USA, 80–92. doi:10.1145/3315454.3329961
[50] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, Jack Hu, Tao Liu, Brian McNamara, Daniel Quirk, and Matteo Taveggia. 2012. *Strongly-typed language support for internet-scale information sources.* Technical Report MSR-TR-2012-101. Microsoft Research. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/information-rich-themes-v4.pdf
[51] The pandas development team. 2025. pandas-dev/pandas: Pandas. doi:10.5281/zenodo.16918803
[52] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '08)*. Association for Computing Machinery, New York, NY, USA, 395–406. doi:10.1145/1328438.1328486
[53] John W. Tukey. 1977. *Exploratory data analysis.* Addison-Wesley. https://www.worldcat.org/oclc/03058187
[54] Typelevel. 2026. Frameless. https://typelevel.org/frameless/
[55] Mitchell Wand. 1987. Complete Type Inference for Simple Objects. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science (LICS 1987)*. IEEE Computer Society Press, Ithaca, NY, USA, 37–44. https://www.ccs.neu.edu/home/wand/papers/wand-lics-87.pdf
[56] Mitchell Wand. 1989. Type Inference for Record Concatenation and Multiple Inheritance. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. IEEE Computer Society, 92–97. doi:10.1109/LICS.1989.39162
[57] Gonzalo Waszczuk, Alberto Pardo, and Marcos Viera. 2017. Extensible records in Idris. In *Proceedings of the 21st Brazilian Symposium on Programming Languages (SBLP '17)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/3125374.3125384

[58] Hadley Wickham. 2014. Tidy Data. *Journal of Statistical Software* 59, 10 (2014), 1–23. doi:10.18637/jss.v059.i10

[59] Hernán Wilkinson. 2019. VM support for live typing: automatic type annotation for dynamically typed languages. In *Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming (Programming '19)*. Association for Computing Machinery, New York, NY, USA, 1–3. doi:10.1145/3328433.3328443

[60] Robert Wright, Michel Steuwer, and Ohad Kammar. 2022. Idris2-Table: evaluating dependently-typed tables with the Brown Benchmark for Table Types (Extended Abstract). In *TyDe 2022*. Association for Computing Machinery, Ljubljana, Slovenia. https://tydeworkshop.org/2022-abstracts/paper6.pdf

[61] Eric Zhao, Raef Maroof, Anand Dukkipati, Andrew Blinn, Zhiyi Pan, and Cyrus Omar. 2024. Total Type Error Localization and Recovery with Holes. *Proc. ACM Program. Lang.* 8, POPL (2024), 2041–2068. doi:10.1145/3632910

A Structural Operations Reference

Hazel Lab extends Hazel's pre-existing labeled tuples with the structural operations described in the main paper. The following tables provide a complete reference for these operations, each with its syntax, an illustrative example, a description, and the static errors that may arise; projection pre-exists in Hazel and is included for completeness.

Projection

Syntax	`t.ℓ`
Example	`(a=1, 2, b=3).a` ⇒ `1`
Description	Projects a specified labeled value out of a tuple.
Static Errors	• The argument's static type is not a tuple • The label is not present in the static type of the tuple

---

Extension

Syntax	`t1 ... t2`
Example	`(a=1, 2, b=3) ... (c=4, a=5, 6)` ⇒ `(a=5, 2, b=3, c=4, 6)`
Description	Concatenates two tuples. Labels in the right-hand tuple override labels in the left-hand tuple; element order is preserved.
Static Errors	• One of the argument's static types is not a tuple

---

Select Labels

Syntax	`select_labels(t, ` ℓ_1 `, ` ℓ_2 `, ...)`
Example	`select_labels((a=1, b=2, 3, c=4), \`a\`, \`b\`)` ⇒ `(a=1, b=2)`
Description	Keeps only the specified labels, preserving the order given as arguments.
Static Errors	• The first argument's static type is not a tuple • One or more of the labels does not appear in the tuple

---

Project Labels

Syntax	`project_labels(t, ` ℓ_1 `, ` ℓ_2 `, ...)`
Example	`project_labels((a=1, b=2, 3, c=4), \`a\`, \`b\`)` ⇒ `(1, 2)`
Description	Equivalent to `select_labels`, but discards labels in the result.
Static Errors	• The first argument's static type is not a tuple • One or more of the labels does not appear in the tuple

---

Omit Labels

Syntax	`omit_labels(t, ` ℓ_1 `, ` ℓ_2 `, ...)`
Example	`omit_labels((a=1, b=2, 3, c=4), \`b\`, \`c\`)` ⇒ `(a=1,3)`
Description	Removes the specified labels from a tuple.
Static Errors	• The first argument's static type is not a tuple • One or more of the labels does not appear in the tuple

Omit All Labels	
Syntax	`omit_all_labels(t)`
Example	`omit_all_labels((a=1, b=2, 3, c=4))` ⇒ `(1, 2, 3, 4)`
Description	Removes all labels while preserving element order.
Static Errors	• The argument's static type is not a tuple

Remarks on structural operations. Some of the operations listed above appear syntactically like ordinary functions, but they are primitive in the language semantics because they operate over labels. In Hazel labels are not first-class expressions and can not be bound to variables, so these operations are implemented natively and their usage is constrained:

- **Labels are not first-class.** For operations that accept explicit label arguments (e.g., `select_labels`, `project_labels`, `omit_labels`), the label literals must be provided at the use site. While the tuple expression argument may be *deferred* via partial application, labels cannot be deferred or passed indirectly, as they are not representable as first-class values in the surface language. The parser and static rules specifically recognize label literals in these call sites to support this behavior.
- **Unknown / deferred tuple arguments yield unknown results.** If a operation is given a tuple expression whose type is unknown (or if the parameter is deferred), the operation's result is given the unknown type. This reflects the fact that the operation's static result depends on the statically-known shape of the tuple.
- **Infers a precise result type when the argument synthesizes one.** When the tuple argument synthesizes a labeled-tuple type, the primitive's static rule returns a correspondingly precise type (for example, selecting known labels yields a labeled tuple with the expected component types). In short, the precision of the inferred result type follows the precision of the argument's synthesized type.
- **Error marking** The static type checking for these operations marks errors in the expected ways: when a parameter expected to be a tuple is not a tuple (or is obviously ill-formed), and when a primitive expects certain labels but those labels are not present in the tuple's synthesized type. Importantly, these checks are built into the language implementation and are *not* expressible in the surface type system: a user cannot write a type that enforces these constraints. This allows the editor to provide precise static diagnostics even for constraints beyond the reach of ordinary type annotations.

B Singleton Unlabeled Tuples

To support structural operations on tuples, Hazel Lab allows singleton unlabeled tuples to be distinguished from their contained element by using an underscore `_` in the label position. For example, `(_=1)` represents a single-element tuple containing the value 1. Singleton unlabeled tuples ensure more predictable behavior when elements are dropped from higher-arity tuples and are also useful when concatenating single fields through tuple extension; for instance, the expression `(1, 2, 3) ... (_=4)` evaluates to `(1, 2, 3, 4)`.

C `to_lvs` and `from_lvs` Reference

The following reference cards describe the syntax, examples, and static error conditions for the `to_lvs` and `from_lvs` primitives.

`to_lvs`	
Syntax	`to_lvs(t)`
Example	`to_lvs((a=1, b=2))` ⇒ `[(label="a",value=1),(label="b",value=2)]`
Description	Converts a fully labeled tuple into a list of `(label, value)` pairs, preserving tuple order.
Static Errors	• The argument's static type is not a tuple

`from_lvs`	
Syntax	`from_lvs(list_of_pairs)`
Example	`from_lvs([(label="a",value=1),(label="b",value=2)])` ⇒ `(a=1, b=2)`
Description	Constructs a labeled tuple from a list of `(label, value)` pairs.
Static Errors	• The argument is analyzed against `[(label=String, value=?)]`

Remarks. The operations `to_lvs` and `from_lvs` play a key role in mediating between statically structured tuples and dynamic processing of data necessary to address the **expressivity problem**. In particular, they serve as a boundary where static typing hands off to live typing: list-based transformations can be written without committing to a fixed schema, while live observations allow the system to recover precise tuple structure when values are available.

D Motivation for Meet Types

Hazel is gradually typed with the primary goal of migrating programs toward fully static types, using the unknown type for incomplete programs (expression holes) and incomplete types (type holes) with expressivity being a secondary goal. Since the primary use of the unknown type is to represent program incompleteness, we prefer to compute the *most precise type* from dynamic observations so that unknown types in dynamic expressions, such as expression holes, do not overshadow other observed types.

For example, two dynamically observed types `(Int, ?)` and `(?, String)` will yield the live type `(Int, String)` under the meet, whereas using a *join type* would result in `(?, ?)`. Our choice of the meet type reflects the expectation that programmers are gradually converging toward precise types, rather than treating unknown types as a top element ($\top$) in a subtyping system.

In cases where the observed types are inconsistent and no meet exists, the expression keeps its static type. This avoids adding potentially incorrect type marks. An expression that is never evaluated has no observations at all and likewise receives no refinement: Fig. 13 illustrates an example where a branch is never executed, and therefore more accurate live types are unavailable.

E Additional Participant Feedback

The following participant feedback from the user study provides additional detail on specific aspects of the system design.

Inline Table Projector. Participants valued the inline table projector for providing immediate visibility into the structure of tabular data within the editor. P2 said that their biggest gripe with pandas and dataframes generally was the lack of visibility into the data—needing to perform extra steps just to figure out what you are working with. Having the table projector directly in the editor for quick reference was very useful, particularly for verifying that column names were correct.

```
let dynamically_select =
  fun (is_int, x) ->
    if true
    then "disabled"
    else
      if is_int then string_of_int(x) else x
in

dynamically_select(false, "");
dynamically_select(true, 1)
```

Fig. 13. A function conditionally calls function `string_of_int(x)` based on an `is_int` argument. Both calls are in branches that are never executed (here, the outer `if true` disables the inner `if` expression). Live typing would correctly infer the type of `x` if the inner `if` ran. Since it does not, `x`'s type is left unrefined, avoiding an incorrect mark.

Tuple Operation Syntax. P4 noted that `omit_labels` uses function-call syntax but takes labels rather than ordinary expressions as arguments, stating: "That seems very confusing to me. I feel like in a more polished version, I want this to somehow be syntactically differentiated." P4 appreciated that `omit_labels` supports deferrals but was unsure about the syntax: "What you can also do deferral… oh, that's interesting. That's cool. It's weird." P4 also acknowledged the utility of `to_lvs` and `from_lvs` for converting between tables and labeled value lists, but noted a desire for stronger typing: "Although, obviously, it would be nice if it was strongly typed."

Rich Probe Interaction Model. Both P3 and P6 expressed a desire to interact more directly with the table visualization rather than navigating back to the source code. P6 stated: "I would prefer if the column addition operation did not require me to go back to the source code." P3 similarly suggested that clicking into the table visualization should navigate the cursor to the corresponding source code. P5 noted: "It saves me from having to write a lot of boilerplate code to locate the field I want to modify, and it also lets me directly check whether my logic is correct."

Live Typing Observations. P6 noted that live typing "looks almost like dependent typing," reflecting how the value-dependent nature of live types differs from traditional static type systems.

F Detailed TPS Comparison

The following descriptions provide a detailed per-system analysis of how each illustrative tabular programming system addresses the problems outlined in the main paper.

- **Scala w/ IDE** — a conventional statically typed batch program written in Scala using a modern IDE and the `frameless-dataset` [54] library for type-safe DataFrames. Frameless does not provide a built-in way to load external schemas as internal types; instead, users define a Scala case class in the program source and load data into it at runtime, leaving the **external schema problem** unaddressed. Because Scala is statically typed, the **function barrier problem** is addressed through compile-time type-checking in the editor. The **distal feedback problem** is partially addressed: static type errors are reported inline without waiting for evaluation, but any dynamic errors are still reported far from the offending syntax. Being a full batch program, out-of-order execution is also eliminated; however, there is no incremental evaluation, so modifying an analysis requires rerunning the entire program. The

expressivity problem is partially addressed by converting to untyped Spark DataFrames, though this comes at the cost of losing static typing guarantees. Value-dependent operations are expressible through this mechanism but require the user to explicitly declare the resulting columns in the program text, typed as `Option` to accommodate missing data. This forces the user to anticipate concrete values in advance, hindering exploration and discoverability and limiting the usefulness of editor feedback.

- **Polynote [39]** — a notebook-style interface embedding Scala, which provides interactive evaluation and partial type-directed services within cells. For tabular programming, we use the Apache Spark [3] library to manipulate DataFrames loaded at runtime (e.g., from CSV files). Polynote partially addresses the **distal feedback problem** by running static checks immediately before evaluating a cell and highlighting expressions causing errors within a cell using the stack trace generated during evaluation; however, it does not typecheck as the user edits the program, and expressions that cannot be statically typed depend on evaluating the program. Polynote also mitigates out-of-order execution by limiting variable visibility to those defined in previous cells; however, it does not guarantee that a cell's value reflects the most recent upstream changes, so stale data can still occur. Polynote partially mitigates the **external schema** and **expressivity** issues by exposing metadata of runtime Spark DataFrames. Nevertheless, because these DataFrame schemas are not represented in Scala's type system, their ability to aid in error detection remains limited.
- **Jupyter + Pandas** — a Jupyter Notebook running Python with the Pandas library for tabular programming. Autocomplete works based on the current state of the kernel, so if a global DataFrame exists, column names are suggested; this partially addresses the **external schema problem** and provides feedback after data transformations, partially mitigating the **expressivity problem**. However, there is no static type feedback for errors, and function arguments inside functions are not checked beyond verifying that the variable exists, leaving the **function barrier problem** and **distal feedback problem** unaddressed. Since cells can be executed in arbitrary order, out-of-order execution is also unaddressed. Pandas provides rich operations for reshaping and transforming tables, including value-dependent operations, so the operations themselves are fully expressible in practice.
- **Hazel Lab** — a live programming environment with support for tables and the primary focus of this paper. Hazel Lab addresses the **function barrier problem** by performing traditional static type checking after every edit. It partially addresses the **distal feedback problem**: static type checking colocates errors with the source syntax and, because it does not require execution, mitigates both the spatial and temporal aspects for statically checkable errors, while live typing extends source-localized feedback to value-dependent errors but, requiring full evaluation, does not reduce their temporal distance. Out-of-order execution is addressed by re-evaluating the full program on each change. The **external schema problem** is addressed through our CSV data loader, which loads external data directly into the program source. Hazel Lab provides rich operations for reshaping and composing tables, and live typing provides feedback for value-dependent operations, together addressing the **expressivity problem**. Rich probes provide additional type-directed feedback and enable edits based on live typing.

Received 2026-03-17; accepted 2026-08-06