Type-Directed Program Transformations for the Working Functional Programmer

Justin Lubin
University of Chicago, USA
justinlubin@uchicago.edu

Ravi Chugh
University of Chicago, USA
rchugh@uchicago.edu

Abstract

We present preliminary research on DEUCE$^+$, a set of tools integrating plain text editing with structural manipulation that brings the power of expressive and extensible type-directed program transformations to everyday, working programmers without a background in computer science or mathematical theory. DEUCE$^+$ comprises three components: (i) a novel set of *type-directed program transformations*, (ii) support for *syntax constraints* for specifying "code style sheets" as a means of flexibly ensuring the consistency of both the concrete and abstract syntax of the output of program transformations, and (iii) a domain-specific language for specifying program transformations that can operate at a high level on the abstract (and/or concrete) syntax tree of a program and interface with syntax constraints to expose end-user options and alleviate tedious and potentially mutually inconsistent style choices. Currently, DEUCE$^+$ is in the design phase of development, and discovering the right usability choices for the system is of the highest priority.

2012 ACM Subject Classification Human-centered computing → Human computer interaction (HCI); Software and its engineering → Domain specific languages; Software and its engineering → Integrated and visual development environments

Keywords and phrases program transformations, structured editing, refactoring, code formatting

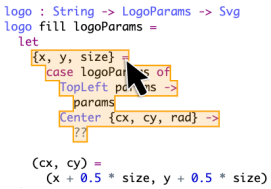
Digital Object Identifier 10.4230/OASIcs.PLATEAU.2019.23

Category Work-in-progress research

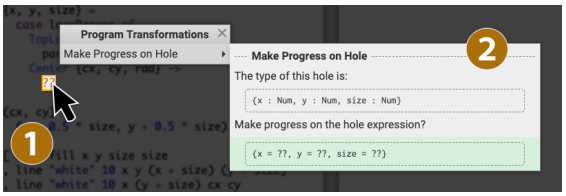
1 Introduction

As a medium for storing, transmitting, and interpreting information, text is as versatile as it is ubiquitous. Countless programs and interfaces operate and rely on text files, from the UNIX command line to nearly every programming language compiler. One particularly strong asset of text is its power to succinctly represent structured data in a way that is understandable to both humans and computers alike, as in CSV (comma-separated values) files, HTML (hypertext markup language) documents, and—the focus of this paper—programming language source files.

Unfortunately, this flexibility comes with a price: manual editing of structured text can be tedious and error-prone. On a basic level, one problem with manipulating structured text is that to do so requires knowledge of and adherence to rigid, static systems such as parsing and type checking. A single missed comma in a CSV file or improperly annotated variable in program source code can cause a complete failure on the part of the computer to interpret the text as the author intended. In the case of performing a nontrivial manual transformation on a program, the problem is exacerbated even further: programmers must also worry about the *runtime behavior* of their code and reason about changes in semantics (or lack thereof) that might be a result of their transformations.



(a) Structurally selecting a let-binding equation in DEUCE.



(b) Activating a program transformation in DEUCE by (1) structurally selecting an expression—in this case, a hole—and (2) interacting with the popup menu.

Figure 1 The DEUCE user interface.

Tools known as *structure* (or *projectional*) editors [9, 13, 24, 25] attempt to alleviate these difficulties by offering an interface that allows programmers to directly manipulate a *projection* of the underlying structure that is more faithful to the structure than is standard text. A major drawback of these systems is their reliance on non-standard file formats, and, as a result, their incompatibility with the large set of existing tools that operate on programs.

One attempt to reconcile the flexibility of plain text with the power of projectional editing is DEUCE [2, 8], a structure-aware code editor that operates on standard program source text, but augments the editing experience with direct manipulation capabilities for invoking relevant, automated program transformations. In DEUCE, the underlying structure of the program is revealed to the user via a set of overlaid polygons, as depicted in Figure 1a. After *structurally selecting* various parts of the program by pressing the shift key, hovering, and clicking on the polygons that appear, a "Program Transformations" menu appears that is automatically populated with a set of relevant transformations for the selected polygons, as depicted in Figure 1b. Hovering over the output of a program transformation previews it in the code panel, and clicking on the output updates the program with the transformed code.

While a good first step, DEUCE falls short in several regards. In particular, it has two main limitations:

(Limitation A) DEUCE only offers a relatively small number of ad-hoc program transformations; and

(Limitation B) the implementation of these transformations is tedious and non-compositional, requiring manual munging of abstract syntax trees annotated with low-level syntactic details such as whitespace.

To address these limitations, we propose and present initial work on a vast expansion of the DEUCE system—which we here call DEUCE$^+$—with the goal of bringing expressive and extensible program transformations to the working programmer.

Paper Outline. In Section 2, we run through a high-level example demonstrating the desired (as-of-yet unimplemented) workflow of the DEUCE$^+$ system. Then, in Section 3, we explain the work that is underway to make that workflow possible and explicitly describe how the improvements we propose will address Limitations A and B. Finally, we conclude in Section 4 with a discussion of the further usability challenges that arise from the DEUCE$^+$ workflow.

In our quest to realize these goals, we assert the necessity of harmony between (i) text and structured editing, and (ii) sound mathematical foundations and usable, accessible, tools for everyday programmers without assuming extensive expertise in advanced functional programming or abstract mathematics.

Table 1 The database schema with some example entries.

Flag	Identity	Homepage
1	alice	example.net
1	bob	
0	carol@example.net	
1	dan	example.org
0	eve@example.edu	

2 The Deuce$^+$ Workflow

Consider a website in which users register and have their information saved to a database. Originally, users could only register on this website with an email address, but this restriction was later relaxed to require only a username and, optionally, a homepage that lists further contact information. To encourage migration to the new username registration format, users may only list a homepage if they register with a username and *not* with an email. The database is structured into three columns, as shown in Table 1: the first tracks whether the user has registered with an email address (0) or a username (1), the second tracks the provided email or username, and the third tracks the (sometimes empty) user homepage.

To access this database, two library functions are provided:

```
identities : Database -> List (Either Username Email)
homepages : Database -> List (Maybe Homepage)
```

Using these two helpers, our task is to implement a function `show : Database -> String`.

We begin by taking a peek at our project's *code style sheet*, a mechanism provided by Deuce$^+$ to allow users to customize preferences about stylistic choices in their programs:

```
.type-alias[type=tuple] { newlines: per-component; }
.tuple { max-size: 3; }
.top-level-definition { eta-reduction: basic; }
```

The first of these three rules tells us that, when aliasing a tuple type with a new name, each component of the tuple should be on its own line. The second rule ensures that only tuples be of (at most) size three are permitted, encouraging larger tuples to be replaced with records. The final rule indicates that "basic" eta reductions should be performed for top level definitions (no term rewriting beyond dropping arguments to functions). None of the program transformations in Deuce$^+$ are aware of the particular code style in any given project, and, consequently, the authors of these transformations do not need to worry about adhering to style guidelines; Deuce$^+$ handles the stylistic details automatically.

We now sketch out the skeleton of our program using standard text editing.

```
showEntries : (List (Either Username Email), List (Maybe Homepage)) -> List String
showEntries (idents, homes) =
  ??

show : Database -> String
show d =
  String.concat (showEntries (identities d, homepages d))
```

In the definition of `showEntries`, we use a *hole expression* [17] as a placeholder until we are able to make further progress implementing the function.

Immediately, we notice that the type annotation on `showEntries` is long and unwieldy. As in Figure 1a (but not shown here or in the rest of the code listings), we *structurally select*

the tuple argument to the `showEntries` function and choose the Make Type Alias from Arguments transformation, entering "Entries" as the new type name.

```
type alias Entries =
  ( List (Either Username Email)
  , List (Maybe Homepage)
  )

showEntries : Entries -> List String
showEntries (idents, homes) =
  ??

show : Database -> String
show d =
  String.concat (showEntries (identities d, homepages d))
```

The new type alias automatically adheres to the specification from our style sheet that type aliases for tuples should have each component on a new line.

However, we are still struggling to make progress on the `showEntries` implementation because the types of our program alone do not ensure that the two lists passed into the function are the same length. In reality, we know that the state in which the two lists are of different lengths *should* be impossible. To take advantage of this fact, we can let the type system know it by structurally selecting the `Entries` type alias, choosing the Refine Type transformation, and entering the equality refinement `\xs -> length (first xs) == length (second xs)`.

```
type alias Entries =
  List (Either Username Email, Maybe Homepage)

convert : (List (Either Username Email), List (Maybe Homepage)) -> Maybe Entries
convert = ...

showEntries : Entries -> List String
showEntries entries =
  ??

show : Database -> String
show d =
  case convert (identities d, homepages d) of
    Just entries -> String.concat (showEntries entries)
    Nothing -> ??
```

The `Entries` type alias has been automatically updated to make the specified impossible state unrepresentable, the argument to `showEntries` has been switched from a (now-outdated) tuple pattern to a single variable name, and a new `convert` function has been introduced that is now used in the `show` function. The `convert` function translates our old `Entries` into the new representation, returning `Nothing` if the conversion fails. A hole expression is inserted in the `show` function to indicate that a default value is needed for when the conversion fails.

We now refactor the `Entries` type alias by structurally selecting the type argument to the `List` constructor and choosing the Introduce Type Alias transformation (which, as before, adheres to the code style sheet specification for type-aliased tuples). We also now provide a default value for `show` in the case of a conversion failure using standard text edits.

```
type alias Entry =
  ( Either Username Email
  , Maybe Homepage
  )

type alias Entries =
  List Entry

convert : (List (Either Username Email), List (Maybe Homepage)) -> Maybe Entries
convert = ...
```

```
showEntries : Entries -> List String
showEntries entries =
  ??

show : Database -> String
show d =
  case convert (identities d, homepages d) of
    Just entries -> String.concat (showEntries entries)
    Nothing -> "Conversion failure!"
```

The `Entries` type alias as well as the `convert` and `show` functions are now complete, so we will omit their definitions until the final code listing for brevity.

We can now make progress on the hole in `showEntries`. The notion of "making progress" on filling in a hole is captured by the Make Progress on Hole transformation. We structurally select the hole in `showEntries` and select this transformation. Deuce$^+$ recognizes that we are trying to fill a hole of type `List String` and that we have an unused argument of type `Entries`, or `List Entry`. Accordingly, one of the options that the transformation presents to the user is the Map over List option with the additional parameter `entries` as the list to map over. We select this option, and the resulting code is automatically eta-reduced in accordance with our code style sheet.

```
type alias Entry =
  ( Either Username Email
  , Maybe Homepage
  )

showEntries : Entries -> List String
showEntries =
  let
    showEntry : Entry -> String
    showEntry entry =
      ??
  in
    List.map showEntry
```

We can make progress on the newly-created hole by structurally selecting it and the `entry` argument, choosing the Make Progress on Hole transformation, and choosing the Pattern Match option.

```
type alias Entry =
  ( Either Username Email
  , Maybe Homepage
  )

showEntries : Entries -> List String
showEntries =
  let
    showEntry : Entry -> String
    showEntry entry =
      case entry of
        (Left username, Nothing) -> ??
        (Left username, Just homepage) -> ??
        (Right email, Nothing) -> ??
        (Right email, Just homepage) -> ??
  in
    List.map showEntry
```

Using standard text editing, we fill in the holes on the branches of the case expression, but we get stuck on the last branch.

```
type alias Entry =
  ( Either Username Email
  , Maybe Homepage
  )

showEntries : Entries -> List String
```

```
showEntries =
  let
    showEntry : Entry -> String
    showEntry entry =
      case entry of
        (Left username, Nothing) -> showUsername username
        (Left username, Just homepage) -> showUsername username ++ "; see " ++ showHomepage homepage
        (Right email, Nothing) -> showEmail email
        (Right email, Just homepage) -> ??
  in
    List.map showEntry
```

We are stuck on the last branch because it is actually an impossible state! We structurally select the branch and choose the Make Impossible code transformation. We are presented with two redefinitions of the `Entry` type alias: a "normalized" and a "standardized" option. The normalized option results in a sum-of-products representation of the valid states, and the standardized option results in a definition in terms of built-in library types.

```
-- Normalized
type Entry
  = Ctor0 Username
  | Ctor1 Username Homepage
  | Ctor2 Email
```

```
-- Standardized
type alias Entry =
  Either (Username, Maybe Homepage) Email
```

Using other code transformations, we can swap one of these implementations for the other at any time, so we do not need to dwell too long on the decision right now. For now, we choose the standardized option. The transformation updates the implementation (but not signature) of the `convert` function as well as the case expression inside the `showEntry` helper.

```
type alias Entry =
  Either (Username, Maybe Homepage) Email

type alias Entries =
  List Entry

convert : (List (Either Username Email), List (Maybe Homepage)) -> Maybe Entries
convert = ...

showEntries : Entries -> List String
showEntries =
  let
    showEntry : Entry -> String
    showEntry entry =
      case entry of
        Left (username, Nothing) -> showUsername username
        Left (username, Just homepage) -> showUsername username ++ "; see " ++ showHomepage homepage
        Right email -> showEmail email
  in
    List.map showEntry

show : Database -> String
show d =
  case convert (identities d, homepages d) of
    Just entries -> String.concat (showEntries entries)
    Nothing -> "Conversion failure!"
```

The function is complete. The data and types have been munged by the code transformations into a cleaner, more manageable form that allows us to focus on the core logic of our program: converting entries in the database to strings.

3 Designing and Implementing Deuce+

To fully realize the workflow described in Section 2, Deuce+ must comprise three distinct but interrelated components:

(Section 3.1) a set of *type-directed program transformations* informed by common code authoring patterns of functional programmers;

(Section 3.2) a *syntax constraint language and engine* to ensure these transformations are composable and produce stylistically consistent output; and

(Section 3.3) a *domain-specific language* for specifying these transformations.

The first of these components will address Limitation A of Deuce by providing a sizeable set of transformations justifiably rooted in existing programmer behaviors, and the second and third of these components will address Limitation B of Deuce by providing a system for building and composing the transformations that is accessible to any user of the system. Moreover, with such tools in place, a large library of automatically composable user-defined transformations will be made possible, further combating Limitation A.

3.1 Type-Directed Program Transformations

With the strong guarantees of a rich type system, programmers can leverage expressive program transformations to alleviate some of the pains caused by the flexibility of text (as demonstrated by [11]). As part of the Deuce$^+$ project, we would like to formalize some of the common, implicit techniques that functional programmers use to develop programs by crafting program transformations to perform these authoring patterns automatically.

From personal experience, we have identified a few such transformations, including the Refine Type, Make Progress on Hole, and Make Impossible transformations from Section 2. However, we would like to design and conduct a user study to observe how functional programmers author code. Do they start with a skeleton of a solution and later fill in the holes? Or do they, perhaps, write code from the top down? There are myriad other ways code can be written, too, and there may not even be a consensus among functional programmers on this issue; nevertheless, a formal user study is in order to even begin to answer these questions. In the meantime, we investigate the three aforementioned program transformations and how they relate to some common functional programming authoring patterns, as summarized briefly in Figure 2.

Refine Type. The Refine Type transformation mirrors the code authoring practice of *maintaining code invariants*. After structurally selecting a type, users may activate the Refine Type transformation to narrow down the type's set of possible values. Inspired by (but not reliant upon) refinement types [6, 7, 23], the Refine Type transformation prompts the user for an invariant that they wish to maintain about the program, and attempts to refactor the type to ensure that the invariant is satisfied. For example, consider a functional queue type (left, below) drawn from Chris Okasaki's *Purely Functional Data Structures* [16]. Given the queue type on the left and the invariant that `|front| >= |back|`, Refine Type might suggest to transform the type to that on the right:

```
type Queue a =
  Queue
    { front : List a
    , back : List a
    }
```

```
type Queue a =
  Queue
    { frontBack : List (a, a)
    , remainingFront : List a
    }
```

The first `|back|` components of `front` and `back` are stored together in the list `frontBack`, and the remaining `|front| - |back|` elements are stored in the list `remainingFront`, so it is now impossible to represent a `Queue` in which `|front| < |back|`. (This transformation is a generalization of the approach used in Section 2 to ensure that the lengths of the two lists in the `Entries` type

were equal.) To maintain ease of use and increase backward compatibility, the Refine Type tool might also provide helper functions corresponding to the previous API of the queue:

```
front : Queue a -> List a
front (Queue q) =
  List.map Tuple.first q ++ remainingFront
```

```
back : Queue a -> List a
back (Queue q) =
  List.map Tuple.second q
```

Make Progress on Hole. The Make Progress on Hole transformation mirrors the code authoring practice of *"following the types,"* an oft-heard adage in the functional programming community suggesting that the type system can guide the user to implement the task at hand essentially automatically. While such advice is clearly not universally applicable, the statement does hold some truth to it, as demonstrated by the practice of type-directed programming [1, 26, 27].

After structurally selecting a hole, users may activate the Make Progress on Hole transformation to fill a hole with an expression (that will likely contain further, more specific, holes to fill in the future). There are at least three type-based approaches that this transformation can rely on to fill holes with helpful expressions: expression templates, type-directed refinement, and program synthesis. *Expression templates* are pre-written generic code snippets that can be suggested to the user to fill the hole at hand based solely on the bindings that are in scope and the type of the hole to be filled, as was done to introduce the `List.map` function in the implementation of the `showEntries` function in Section 2. *Type-directed refinement* is the systematic destructuring of a type into its component types via pattern matching, as was done to pattern match on the `entry` variable in the implementation of the aforementioned `showEntries` function. Another example of type-directed refinement would be, for instance, the filling of a hole of type `(a, a)` with the expression `(??, ??)`, an expression whose holes have been refined to simpler types (albeit at the cost of introducing a greater number of holes/subproblems). A final, more general approach to the task of filling holes lies in the practice of type-directed *program synthesis*, or, generating programs to match a specification (which, in this case, is the type of the hole along with any additional information—such as examples—that the synthesis algorithm may need), as in [5, 10, 12, 18, 19, 20, 21].

Make Impossible. The Make Impossible transformation mirrors the code authoring practice of *making illegal states unrepresentable* [4, 15]. At the time of authoring a type, certain code decisions may seem like a good idea that only later reveal themselves to be cumbersome. For example, a programmer might write a record type representing the state of an application window with the field `content : Maybe String` (where `Nothing` represents a closed window). Some time later, the programmer may realize that windows should save their own position, and thus adds a field `position : Maybe (Int, Int)` (when the window is closed, they reason, it has no position, so `position` must be a `Maybe` type). But, in enacting this change, the programmer has made representable two states that should be illegal: when either one of the fields is `Just something` and the other is `Nothing`.

By structurally selecting the record type and providing the patterns that should be unrepresentable (or by merely structurally selecting the branches in a case expression that should be unrepresentable), the programmer can use the Make Impossible tool to make values of the selected type that match the specified patterns impossible to represent. In the windowing example from above, the Make Impossible tool would transform the record of `Maybe` types to `Maybe { contents : String, position : (Int, Int) }`.

This transformation is made possible by the algebraic nature of datatypes, as used

Refine Type	$\longleftrightarrow$	*maintaining code invariants*
Make Progress on Hole	$\longleftrightarrow$	*following the types*
Make Impossible	$\longleftrightarrow$	*making illegal states unrepresentable*

Figure 2 Summary of transformation and code authoring pattern correspondences.

in [14]. While work is still underway to make the intuition rigorous, we may view the Make Impossible transformation on a datatype as operating on the polynomial functor of which the type is a fixed point, represented suggestively as $\tau - P$, where τ is the datatype functor and P is the pattern we wish to eliminate. For example, viewing record types as product types, we can determine the solution to the windowing example from above using algebraic laws related to this transformation (which are still under investigation):

`(Maybe String, Maybe (Int, Int))` $-$ `(Just _, Nothing)` $-$ `(Nothing, Just _)`

$$\rightsquigarrow (1+s)*(1+(i*i)) - s*1 - 1*(i*i)$$
$$= 1*1 + 1*(i*i) + s*1 + s*(i*i) - s*1 - 1*(i*i)$$
$$= 1 + s*(i*i)$$

$\rightsquigarrow$ `Maybe (String, (Int, Int))`.

3.2 Syntax Constraints

Implementing program transformations can be difficult and time-consuming work; it is clear that even just the three transformation described in Section 3.1 will require great thought and careful execution in their design and implementation. On top of the inherent complexity of the program transformation at hand, transformation authors—in the most basic setting—will also need to worry about routine but difficult-to-get-right considerations such as whitespace handling, declaration ordering, and naming conventions, among many other stylistic considerations. Moreover, sets of transformations authored by different people might all work with mutually incompatible styles, resulting in a poor user experience.

Automatic formatting tools eliminate these problematic choices by applying language- or project-specific concrete formatting rules to abstract syntax trees. Although formatting tools can be extremely valuable, they are typically limited to concrete syntax (even though many structural choices can be viewed as stylistic, too) and based on hard-coded rules that are not easily adapted to multiple configurations.

To address this problem, we propose the notion of code style sheets, or, the notion that stylistic choices for both concrete and abstract syntax be made automatically based on *syntax constraints*. Syntax constraints are an expressive means for encoding a variety of stylistic choices to be optimized to meet different requirements (e.g. line widths, stylistic preferences, and surrounding context). Much like the distinction between HTML, CSS, and JavaScript, syntax constraints specify code style completely independently of the code itself and any transformations that operate on it, reducing the burden of transformation authors while also ensuring a more consistent and flexible experience for users of these transformations.

When needed, Deuce$^+$ will feed the syntax constraints (along with unformatted code) into a solver engine, which will output formatted code. Presently, the exact nature of this engine as well as of the syntax constraints themselves are under investigation.

3.3 A Program Transformation Language

Even assuming that both concrete and abstract formatting are taken care of by the code style sheet engine, writing the transformation code for manipulating abstract syntax trees can be challenging. Speaking from the personal experience of implementing thousands of line of program transformations, it is difficult to maintain a declarative, consistent, and reusable pattern of implementation that scales well for even a few dozen transformations.

To resolve these issues, we propose a domain-specific language for program transformations that can operate on three different levels of abstraction: the concrete syntax tree, the abstract syntax tree, and the generalized syntax tree. The concrete syntax tree and abstract syntax tree are familiar: the former including rigid details such as the exact whitespace of the code to be transformed and the latter including only the underlying structure that is fed to, for example, the evaluator of the language. The *generalized syntax tree* operates at an even higher level than the abstract syntax tree, and allows for an even more declarative approach to specifying program transformations. It relaxes certain relationships between nodes in the abstract syntax tree so that, for example, an ordered list of let-bindings becomes an unordered set of let-bindings and function application operators (such as `|>` in Elm, OCaml, and F# and `$` in Haskell) are unified into a single function application node. Translating from the transformed high-level description of the program back to the concrete syntax that the programmer sees can be done by the use of the code style sheet engine, and the concerns of style and transformation logic are cleanly separated.

The program transformation language should also be able to interface with the code style sheet, exposing transformation-specific properties that allow the programmer to fine-tune how the transformation operates on a granular basis. Moreover, the language should eventually be amenable to *synthesis* of program transformations, reducing the barrier to authorship of transformations even further; indeed, the synthesis of program transformations is an exciting area of related (as in [22]) and future work. As with the syntax constraints and solver, much further investigation is needed to understand, design, and implement the generalized syntax tree specification as well as the domain-specific language as a whole.

4 Further Usability Challenges for Deuce$^+$

If the goal of Deuce$^+$ is to make powerful program transformations backed by elegant mathematical ideas accessible to—and authorable by—the everyday programmer, then good usability is of utmost importance. Every single operation described thus far should be influenced and rooted in not only sound mathematics, but proper usability studies. Although the development of Deuce$^+$ is currently very early in the design process, there are a few preliminary usability considerations that need to be addressed as Deuce$^+$ develops.

- With so many program transformations available to the user (especially including transformations authored by third parties), it is critical that the user be able to find the correct transformation for the task at hand, even if the user does not know its name. We will need to investigate what mental models users have of the transformations available to them and determine heuristics (likely relating to the structurally selected expressions) and display mechanisms (such as code diffs or animation) for showing them relevant program transformations.
- After a transformation has been selected, its output may be hard to decipher, so how does a user know when a program transformation is "correct?" We will need to investigate how to help users be confident (and correct) in their output selections.

- The languages underlying the program transformations themselves must also be streamlined and intuitive for end-user transformation authors. Drawing from interdisciplinary programming language design [3], we will need to design and evaluate the style sheet and transformation specification languages holistically, aiming for providing an experience that supports and encourages expressive and extensible code.
- The structured editing user interface introduced by Deuce will need to be improved to support usability improvements such as drag and drop, fluid and dynamic animations, and novel code overlays beyond simple polygons (such as for type information).

Completion of these tasks will ensure that—at every step of the way—the usability of Deuce$^+$ is of the highest priority across all its components, from its graphical user interface to its program transformation authoring and end-user experience.

References

1. Edwin Brady. *Type-Driven Development with Idris*. Manning Publications Company, 2017. URL: https://books.google.com/books?id=eWzEjwEACAAJ.
2. Ravi Chugh. Structured Editing for Elm* in Elm. elm-conf, 2018.
3. Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. Interdisciplinary Programming Language Design. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2018, pages 133–146, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3276954.3276965, doi:10.1145/3276954.3276965.
4. Richard Feldman. Making Impossible States Impossible. elm-conf, 2016.
5. Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-Directed Synthesis: A Type-Theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 802–815, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2837614.2837629, doi:10.1145/2837614.2837629.
6. Tim Freeman and Frank Pfenning. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM. URL: http://doi.acm.org/10.1145/113445.113468, doi:10.1145/113445.113468.
7. Susumu Hayashi. Logic of refinement types. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 108–126, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.
8. Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. Deuce: A Lightweight User Interface for Structured Editing. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 654–664, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3180155.3180165, doi:10.1145/3180155.3180165.
9. JetBrains. JetBrains MPS. https://www.jetbrains.com/mps/.
10. Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-Guided Program Synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 253–268, New York, NY, USA, 2019. ACM. URL: http://doi.acm.org/10.1145/3314221.3314602, doi:10.1145/3314221.3314602.
11. Huiqing Li and Simon Thompson. Tool Support for Refactoring Functional Programs. In *Partial Evaluation and Program Manipulation*, pages 182–196, San Francisco, California, USA, January 2008. URL: http://www.cs.kent.ac.uk/pubs/2008/2634.
12. Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. Live Example-Directed Program Synthesis. July 2019.
13. John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The Scratch Programming Language and Environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15,

November 2010. URL: http://doi.acm.org/10.1145/1868358.1868363, doi:10.1145/1868358.1868363.

14 Conor McBride. The Derivative of a Regular Type is its Type of One-Hole Contexts. http://strictlypositive.org/diff.pdf.

15 Yaron Minksy. Effective ML. https://blog.janestreet.com/effective-ml-video/, 2010.

16 Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998.

17 Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live Functional Programming with Typed Holes. *PACMPL*, 3(POPL), 2019. URL: http://doi.acm.org/10.1145/3290327, doi:10.1145/3290327.

18 Peter-Michael Osera. *Program synthesis with types*. PhD thesis, University of Pennsylvania, 2015.

19 Peter-Michael Osera. Constraint-Based Type-Directed Program Synthesis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2019, pages 64–76, New York, NY, USA, 2019. ACM. URL: http://doi.acm.org/10.1145/3331554.3342608, doi:10.1145/3331554.3342608.

20 Peter-Michael Osera and Steve Zdancewic. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 619–630, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2737924.2738007, doi:10.1145/2737924.2738007.

21 Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 522–538, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2908080.2908093, doi:10.1145/2908080.2908093.

22 Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 404–415, Piscataway, NJ, USA, 2017. IEEE Press. URL: https://doi.org/10.1109/ICSE.2017.44, doi:10.1109/ICSE.2017.44.

23 Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM. URL: http://doi.acm.org/10.1145/1375581.1375602, doi:10.1145/1375581.1375602.

24 Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: A Syntax-directed Programming Environment. *Commun. ACM*, 24(9):563–573, September 1981. URL: http://doi.acm.org/10.1145/358746.358755, doi:10.1145/358746.358755.

25 Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards User-Friendly Projectional Editors. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering*, pages 41–61, Cham, 2014. Springer International Publishing.

26 Geoffrey Washburn and Stephanie Weirich. Good Advice for Type-directed Programming Aspect-oriented Programming and Extensible Generic Functions. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Generic Programming*, WGP '06, pages 33–44, New York, NY, USA, 2006. ACM. URL: http://doi.acm.org/10.1145/1159861.1159867, doi:10.1145/1159861.1159867.

27 Stephanie Weirich and Liang Huang. A Design for Type-Directed Programming in Java. *Electron. Notes Theor. Comput. Sci.*, 138(2):117–136, November 2005. URL: http://dx.doi.org/10.1016/j.entcs.2005.09.014, doi:10.1016/j.entcs.2005.09.014.