Experience Report: Growing Programming Languages for Beginning Students

Marcus Crestani
University of Tübingen
crestani@informatik.uni-tuebingen.de

Michael Sperber
DeinProgramm
sperber@deinprogramm.de

Abstract

A student learning how to program learns best when the programming language and programming environment cater to her specific needs. These needs are different from the requirements of a professional programmer. Consequently, the design of teaching languages poses challenges different from the design of "professional" languages. Using a functional language by itself gives advantages over more popular, professional languages, but fully exploiting these advantages requires careful adaptation to the needs of the students—as-is, these languages do not support the students nearly as well as they could. This paper describes our experience adopting the didactic approach of *How to Design Programs*, focussing on the design process for our own set of teaching languages. We have observed students as they try to program as part of our introductory course, and used these observations to significantly improve the design of these languages. This paper describes the changes we have made, and the journey we took to get there.

Categories and Subject Descriptors D.2.10 [*Software Engineering*]: Design—Methodologies; K.3.2 [*Computers and Education*]: Computer and Information Science Education—Computer Science Education

General Terms Design, Languages

Keywords Introductory Programming

1. Introduction

Functional programmers know that the choice of language affects the thinking of programmers and thus the design of software. The choice of language also matters when it comes to teaching introductory programming: It profoundly affects the students' thinking repertory, as well as their learning experience. An "off the rack" language poses significant challenges for beginners and tends to be an obstacle to learning (Felleisen et al. 2004; Findler et al. 2002).

In 1999, the University of Tübingen started revising its introductory course: A functional-programming-based course replaced more traditional previous offerings using Pascal, C++, or Java. The course was, to a large degree, based on the classic *Structure and Interpretation of Computer Programs* (or *SICP*) (Abelson et al. 1996). We were aware at the time of Rice PLT's efforts, led by Matthias Felleisen, that would result in *How to Design Programs* (or *HtDP*) (Felleisen et al. 2001), which, however, had not been published yet—consequently, we only had a vague idea of its central tenets. We used PLT's DrScheme (now called DrRacket) for the course, seeing it mainly as a graphical IDE for Scheme, and thus easier to use for students than traditional Scheme systems. Specifically, we ignored its hierarchy of language levels and instead ran DrScheme in its R^{5}RS mode. Our underlying assumption was the same as that of SICP, namely that the sheer power of functional programming combined with the syntactic simplicity of Scheme would make both teaching and learning so easy that we would fix all the problems of the previous courses instantly. However, while Scheme fixed many problems, significant issues remained.[1]

After having written a textbook on this approach, it took us until 2004 to realize that SICP's example-driven approach to teaching did not work as well as we had expected with a large portion of our students: SICP admirably explains how many concepts of software development—abstraction in particular—work, but this is not enough to enable students to solve problems on their own. By this time, HtDP had appeared, and we started to adopt its central didactic concept, the *design recipes*, which implement an explicit programming process, driven by a data analysis.

Adopting the design recipes meant expressing their concepts in code. However, pure R^{5}RS Scheme is a poor match for the design recipes—it lacks native constructs for compound data, mixed data ("sum types"), and noting violations. Consequently, we started implementing our own "language level," which included the missing features and allowed us to adopt HtDP's design recipes, while staying close to "standard" Scheme. We still shunned HtDP's own language levels, as they deviate significantly from R^{5}RS Scheme. However, even though we did not know it at the time, we had replicated the first step of PLT's own journey towards the language levels, and we would replicate more.

For reasons initially unrelated to the language, we started observing our students as they worked on exercises (Bieniusa et al. 2008). Soon, we saw that, despite Scheme's simplicity, students were making syntactic and other trivial mistakes. Experienced programmers "see" these mistakes immediately, but students often do not. This can be immensely frustrating, and a significant number of students gave up on programming on their own as a result. Disturbingly, many of these mistakes could have been detected by the Scheme implementation if only the language and the error messages were restricted to what the students knew. Consequently, we started implementing restrictions in line with the course, and custom error messages—replicating another step of PLT's experience.

Moreover, we saw that some students would read ahead and make use of programming-language features not yet covered in the course (most popular: assignments), which destroyed important didactic points: Thus, we implemented a sequence of progressively bigger language levels, replicating and thus confirming the final essential step of PLT's development of the HtDP language levels. In 2006, after adding more add-on features analogous to HtDP's, such as the handling of images and functional animations, we had lan-

[1] We were wrong about the course on other aspects as well (Bieniusa et al. 2008).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-60558-794-3/10/09…$10.00

guage levels almost completely analogous to HtDP's. We also published our own follow-up textbook, *Die Macht der Abstraktion* (or *DMdA*, German for *The Force of Abstraction*) (Klaeren and Sperber 2007). In retrospect, we could have gotten there much faster and cheaper. However, at the time, we did not have PLT's experience and the rationale for their design, and thus had proudly assumed a "not-invented-here" stance.

Nevertheless, the design process for our teaching languages did not end there: The HtDP language levels still had insufficient support for some aspects of the design recipes—in particular, test cases, mixed data, and contracts. Moreover, new desirable aspects of teaching languages emerged—most recently, the support for formal properties of programs. This paper documents our experience with adopting the HtDP approach and evolving our teaching languages to better meet the students' needs.

2. HtDP's Language Levels

The HtDP (and DMdA) languages have evolved from Scheme, which, at the time, had been the basis for many introductory textbooks and courses, as its small size make it attractive for classroom use, and beginning students take well to the simple Lisp-style parenthesized prefix syntax. However, standard Scheme does not solve all language problems of the introductory course. Thus, improving the students' experience meant changing and improving the language, as PLT's TeachScheme! project has been doing since 1995 (Felleisen et al. 2004; Findler et al. 2002).

In particular, students make mistakes when writing code. If the student is to make independent progress, the programming environment must provide feedback that enables the student to fix the mistakes on her own. These mistakes are often trivial: Syntax errors (which occur even with the "trivial" Scheme syntax) and type errors can be detected by the programming environment. Helping students fix other kinds of mistakes—misunderstanding the syntax or using features not yet covered in class—require actual changes to the languages beginners program in.

In DrScheme, at any given time, the beginner uses one of several *language levels*. A language level is an operation mode of DrScheme that provides a language subset tailored to the needs of the beginner at that time. As the student progresses, she switches to more advanced language levels, each of which is a superset of the previous level. Each language level has its own implementation of error reporting tailored to the beginner's needs. The error messages only mention terms that the course has introduced up to that point.

3. Popularity $\neq$ Success

Adopting HtDP's insights for what would become DMdA was a lengthy process: Prior to the 2004 course, we only had a vague idea what the students were doing when they were on their own. That did not keep us from *believing* we had a fairly good idea of what they were doing, namely solving their homework problems using the techniques we had taught them. Only when we started personally supervising lab exercises, we found out that the students did not always follow the path we had laid out for them, and encountered numerous difficulties. This was easy to address during personal supervision, but would have kept the students from solving homework problems when on their own. In fact, many students resorted to copying somebody else's homework (Bieniusa et al. 2008), and our impressions of what the students were doing turned out to be quite wrong, even though we thought we had good reason to believe they were right: The course was popular with students, and passing rates were higher than with the previous, "traditional" courses, even though we had covered more difficult material.

When we realized this, we started observing our students more closely. Specifically, we recorded the mistakes they made, the error messages from DrScheme that reported the mistakes, and the students' reaction to the error messages. The authors did this personally, and additionally trained our student TAs to look for mistakes, and report their observations to us. We also tried to raise the students' awareness of these issues and report them. However, most of the helpful observations we made ourselves, closely followed by the TAs reports—we received very little unsolicited feedback from the students, and even this was mostly ad-hoc in-class feedback.

The following insights from our experience have stayed with us:

- We did not even know we had a problem, even though we have always maintained an open door and open ears for our students. Consequently, it was extremely easy to deceive ourselves that "everything was fine."
- Mistakes made by one student were often repeated by other students.
- What seems easy or natural to us does not necessarily appear that way to the students.
- We could not expect the students to give us, on their own initiative, the specific feedback we need to improve the course and the software for the course.

The design decisions documented in this paper were mostly direct consequences of this action research, which is ongoing. Student's scores in the programming exercises of the final exams have continually risen since we have adopted design recipes and started improving our teaching languages.

4. Simple Differences

The original DMdA languages of 2006 differed from the HtDP languages in several minor ways—partly to reduce the differences with standard Scheme, and partly to cater more specifically to our German audience. The HtDP languages generally appeal to the students' prior training in algebra, sacrificing some of the original Scheme syntax, whereas the DMdA languages stay closer to the original Scheme. The differences illustrate some of the decisions designers of languages for beginners face.

4.1 Procedure/Function Definitions

The difference in the handling of algebra is most visible in procedure definitions: In HtDP, procedures (called "functions" there) are defined with the usual Scheme syntactic sugar:

```
(define (f x)
  ...)
```

This emphasizes the similarity to function definitions in mathematics as well as the visual congruence between function definitions and calls, and makes it easy to "see" the substitution that occurs. Conversely, DMdA's procedure definitions use an explicit `lambda`:

```
(define f
  (lambda (x)
    ...))
```

This makes it easier later to introduce higher-order procedures, as it is straightforward to move the `lambda` somewhere else as opposed to explaining the concept of syntactic sugar, but loses the visual congruence. This is no great loss, however, as German students typically cannot identify the mathematical substitution principle, anyway—the subject does not play the explicit role in German high school that it enjoys in US curricula.[2] Explaining it from scratch with `lambda` is thus no more difficult than explaining it using the syntactic sugar.

[2] Ironically, Felleisen traces back the algebraic aspect to his training in German high school, where algebra sadly has since been de-emphasized.

4.2 Record Definitions

An important part of HtDP and DMdA is the treatment of *compound data*. Instructors teach students to recognize compound data, and use record definitions as implementations of the resulting data definitions. Teaching compound data effectively is surprisingly difficult, as beginning students tend to get confused about the idea of "several things becoming one." Both DMdA and HtDP instructors teach simple heuristics such as that the number of components in the data definition should match the number of fields. ("How many parts does a calendar date have? Three! How many fields does the record-type definition for calendar dates have? Three!") This means that the programming aspects of compound data ought to be as simple as possible, to not add to the students' burden.

Scheme has a long history of "record wars" (Clinger et al. 2005), hence it is no surprise that DMdA and HtDP chose different syntaxes for their record-type-definition forms. HtDP has chosen a so-called "implicit-naming" form. For example, consider the following HtDP "struct definition":

```
(define-struct ant (weight loc))
```

This is in fact a definition of four procedures: A record constructor called `make-ant`, a predicate `ant?`, and two selectors `ant-weight` and `ant-loc`. The names are not explicitly mentioned in the form, hence "implicit-naming."

The DMdA teaching languages provide an "explicit-naming" form. Here is a definition equivalent to the above:

```
(define-record-procedures ant
  make-ant ant?
  (ant-weight ant-loc))
```

This is more verbose than the HtDP form, but makes it easier for the students to see that the form defines identifiers, and what those identifiers are. Also, `define-record-procedures` allows choosing arbitrary names for the various procedures, even though we emphasize the value of the conventions used above. Moreover, the DrScheme "Rename" menu entry works with explicit naming form, but not with the implicit naming.

Some instructors in Germany experimenting with the HtDP languages reported that a significant number of students had difficulty understanding the "magic" of implicit naming. This particular problem is not as significant in DMdA courses; signatures (see Section 5.2) further alleviate any problems the students may have with writing record-type definitions.

4.3 Print Format

The REPL of a typical Scheme implementation accepts an expression and then prints its value. While the output format of the value is not standardized, most Scheme implementations output the (standard) external representation of the value: 5 prints as `5`, "true" prints as `#t`, and the list with elements 1, 2, and 3 prints as `(1 2 3)`. While the use of the external representation has advantages for dealing with advanced features of Scheme such as representing program source code as data, `eval` and `quote`, it confuses many beginning students about the difference between expressions and values. For example, the expression `(list '+ 1 2)` evaluates to `(+ 1 2)`, which looks like an expression that evaluates to 3.

HtDP and DMdA avoid this confusion by using output formats different from the external representation. As HtDP emphasizes the relationship between algebra and programming, it prints out each value as a canonical form that evaluates to it. Thus, the list with elements 1, 2, 3 prints as `(cons 1 (cons 2 (cons 3 empty)))` or `(list 1 2 3)` (depending on the language level), which, as an expression, again evaluates to a list with elements 1, 2, 3. Record values are printed as constructor calls—for example, an ant will print out as `(make-ant` w `(make-posn` x y`))`.

With DMdA, we instead chose to emphasize the distinction between the expression `(make-posn 1 2)` and its value. This is particularly relevant in DrScheme's stepper (Findler et al. 2002), which displays intermediate reductions as expression. In DMdA, the list prints as `#<list 1 2 3>`, and the ant prints as `#<record:ant` w `#<record:posn` x y`>>`. This has the technical disadvantage of not being usable as an expression, but also prevents certain abstraction violations: In particular, it prevents students from cutting and pasting the result directly into a test case.

Both approaches have been successful at avoiding the confusion associated with the standard external representation.

4.4 Minor Language Changes

We made additional minor changes over the HtDP languages. One example is the omission of symbols in favor of strings: HtDP (and an ordinary Scheme programmer) uses symbols for enumerations (`'solid`, `'liquid`, `'gaseous`) where DMdA uses strings. This avoids the notational difficulties of using symbols, in particular the syntactic restrictions (no spaces etc.), and also the notational confluence between symbols and variables. We had observed these problems in earlier incarnations of the course, and switching to strings solved them all. (One might argue that this is less efficient, but it is the introductory course, after all.)

Delaying symbols enables DMdA to also relegate `quote` (including quoted lists) to the very end, the general notion of which was quite confusing to students when introduced earlier. The inconvenience—`(list "solid" "liquid" "gaseous")` instead of `'(solid liquid gaseous)`—is well worth it.

5. Growing the Teaching Languages

In 2006, when the DMdA teaching languages had become roughly analogous to the HtDP languages, we could focus on further improvements. In particular, we adopted and improved upon newer developments in the HtDP languages such as the support for testing. We have also developed two new additions: support for signatures, and the formulation of general, checked properties of procedures.

5.1 Encouraging Testing

Writing test cases is an early step of the design recipes. In particular, students should write test cases before they write the procedure definition itself.

When we originally introduced testing as a mandatory part of the design recipes, we adopted graphical test boxes, which HtDP had implemented previously, that the students had to insert via a menu and fill out like a form. A test box would contain "Test" and "Should be" fields, that would be tested for equality. DrScheme would decorate test boxes of successful tests with green marks and failed tests with red marks and the actual value. The idea was that the graphical and form-like approach would make testing more attractive to students, but in fact the opposite was the case: The students found the GUI manipulation required to use test boxes too cumbersome. Moreover, the test boxes had to come *after* the procedure definition of the procedure they were supposed to test even though the design recipes specify that the students write them *before* writing the procedure definition. As a result, many students wrote their test cases after completing the procedure body.

To encourage the students to test more, we replaced the mechanism for writing tests by one HtDP had implemented earlier: Instead of graphical test boxes, test cases are formulated as plain code using the `check-expect` form that accepts a test expression and a should-be expression as operands. The test case for `is-5?` can be formulated as a `check-expect` form like this:

```
(check-expect (is-5? 7) #f)
```

When we replaced graphical test boxes by `check-expect`, the students wrote significantly more test cases. The `check-expect` form allows quick creation, keyboard-based manipulation and easy duplication.[3] Also, `check-expect`-based tests run after the rest of the program, and can be placed above the procedure definition. This successfully encourages the students to write test cases before writing the procedure definition.

Thus, even though the difference between the graphical test boxes and `check-expect` is linguistically insignificant, the results differ dramatically: Details matter.

5.2 Signatures

An important part of the design recipes is the formulation of a *contract* for every procedure. In HtDP the contracts are comments:

```
;; is-5? : number  ->  boolean
(define (is-5? n)
  (= n 5))
```

The HtDP language of contracts is informal. (HtDP predates PLT's well-known research on contracts as part of the programming language.) Most contracts look like type signatures. (Some represent more complex predicates, but this is not the main point here.)

Writing down contracts is important for the students, as it helps answer typical questions, such as how many arguments they should supply in a procedure call, or how they should order them. Thus, contracts further guide decisions students have to make when they write their programs, and, once written, do so without requiring the student to think about the concrete problem at hand. Consequently, the remove the process of constructing the program from "solving the whole problem" by one—often crucial—step. Furthermore, TAs use contracts as anchors for giving helpful instructions,

As contracts are not subject to static type checking, type errors do not keep a student from running the program and observing its behavior. Consequently, while *writing down* a type signature would have the same benefits as writing down the contract, the effects of doing this in a statically typed language would be detrimental for the beginning student when *trying to run* the program.

The complete lack of checking also creates problems: Many students quickly realize that the contract comments have no bearing on the running program, and as a result they are sloppy with more complicated contracts. This led DMdA to add *signatures* as formal parts of the teaching languages in 2008, which take the place of HtDP's informal contracts. Here is a signature declarations:

```
(: is-5? (number -> boolean))
```

Any signature violation is logged like a test-case violation—see Figure 1. The feedback to the student includes the expression in the program whose evaluation violated the signature, the signature that was violated, and the value that violated it. The value is important for the student, as it provides concrete evidence that the program did something wrong (rather than a type system's assertion that the program *might* do something wrong), and helps the student figure out the source of the problem.

While replacing contracts with signatures does not significantly alter the pedagogy of the course, automatic checking plays the role of the lab supervisor for the students, and provides more immediate and precise feedback. The introduction of signatures showed instant results in class: The students were more thorough about writing them, and programming was more in line with the design recipes, as each part of a data definition now results in an actual piece of program code: The code for a definition for mixed data (the terminology used by DMdA and HtDP for "sums"), which previously had no counterpart in the code, looks like this:

```
(define animal
  (signature (mixed ant armadillo bigfoot)))
```

This definition can be read as "an animal is an ant, armadillo, or a bigfoot" or, more precisely, "a value matching the signature `animal` must match one of the signatures `ant`, `armadillo`, `bigfoot`." The `signature` keyword marks the expression as written in signature syntax.[4]

Compound data requires no new special form with signatures—students write regular signatures for the constructors, predicates, selectors and mutators. For the ants record definition from Section 4.2, students would typically write the following signatures:

```
(: make-ant (real posn -> ant))
(: ant? (%a -> boolean))
(: ant-weight (ant -> real))
(: ant-loc (ant -> posn))
```

The first line declares that the constructor for ants accepts a real number and a position, and returns an `ant` record, the next that `ant?` accepts any value and returns a boolean, and the two following lines that the selectors for the weight and loc fields accept an `ant` record and return a real number and position, respectively. The first declaration already says all there is to say about ants—all predicates have the same signature. The selector signatures simply mirror the constructor signature, and we originally taught our students to only write this first line. To our (pleasant) surprise, the students soon insisted on writing all signatures, which have since been consistently helpful in getting students to understand the concepts of predicate and selector.

The `%a` signature is a signature variable, as is every identifier appearing in a contract that starts with a `%`. This notation allows formulating typical "polymorphic" signatures like this:

```
(: map ((%a -> %b) (list %a) -> (list %b)))
```

The implementation views any such signature as meaning "any"—hence, the system does not check correct use of parametric polymorphism, and thus fails to prevent students from being sloppy with proper use of signature variables. However, this problem is quite minor compared with the sloppiness we had observed earlier.

Note that signatures work as invariants for procedure calls. Conversely, the "real" contracts that are available in Racket monitor the flow of values across module boundaries (Flatt et al. 2010).

5.3 Properties

We noticed in the Tübingen 2008 course that some students, when the course introduced `check-expect`, would ask whether it might be possible to check for properties rather than examples. This struck a nerve with the DMdA team, as the textbook includes a section on formal specification using equational properties based on ADTs. This section had never worked particularly well, as it requires talking about semantics in terms of universal algebra. This was time-consuming and too obscure for students to grasp in the first semester. Moreover, we found that formulating interesting properties—such as fundamental properties of search trees—was

[3] In hindsight, this seems obvious, but it was far from obvious at the time, considering the prevalence of graphical paradigms in professional development environments.

[4] The `signature` syntax could almost but not quite be expressed as a combinator library, or individual macros for `mixed` etc.: The `signature` syntax delays references to signature variables and invocations of signature abstractions to allow recursive signatures. Moreover, it attaches fresh locations to the various parts of the syntax to enable intuitive error reporting. For example, when the `number` signature of `is-5?` above is violated, the visual feedback marks the particular occurrence of `number` in `is-5?`'s signature. To enable this, the system must treat `number` differently from a generic variable reference.

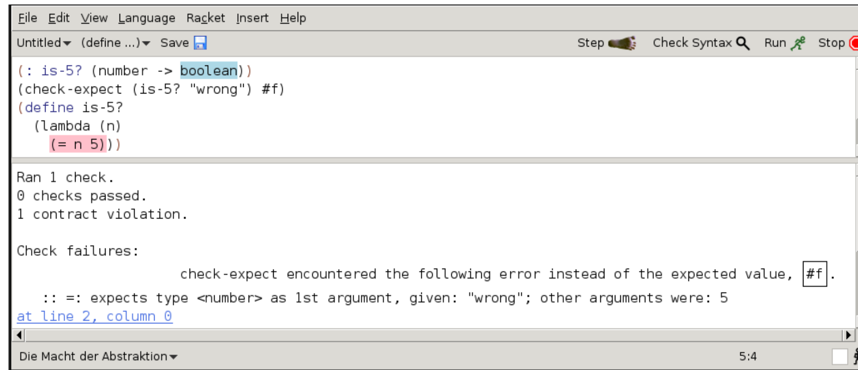

Figure 1. Signature violation in DrScheme

beyond the reach of the framework we had introduced, which was already too complex.

Consequently, we decided to instead introduce properties in the concrete context of programming and add support for them to the DMdA languages. Here is an example:

```
(define +-is-commutative
  (for-all ((a number) (b number))
    (= (+ a b) (+ b a))))
```

The range of variables in the new `for-all` construct is specified using signatures. Thus, adding signatures to the language paid off in an unexpected way. Properties are objects, which can be composed. The new `check-property` form can be used to check a property:

```
(check-property +-is-commutative)
```

This invokes a QuickCheck clone (Claessen and Hughes 2000), and DrScheme displays counterexamples along with the test results.

As signatures are run-time objects, the system constructs the value generators needed for QuickCheck using "regular programming" rather via type-class-based overloading. The fact that signatures are objects enables simple abstractions accessible to beginners, such as this:

```
(: commutativity
   ((number number -> number) signature
    -> property))
(define commutativity
  (lambda (op sig)
    (for-all ((a sig) (b sig))
      (= (op a b) (op b a)))))
```

This enables concrete practice dealing with abstract properties—this is helpful for our beginning students who struggle with the general concept of "commutativity" when divorced from arithmetic.

Properties have now replaced the ADT-based approach to formal specification in the course, and the course segues from the QuickCheck testing to actual proofs of properties. Initial feedback from the 2009/2010 courses in Tübingen and Freiburg has been positive. In the Tübingen course, which placed more emphasis on properties, the students invented properties—typically simple algebraic properties such as commutativity, associativity, distributivity—throughout the course. Consequently, we are confident that properties will play a more prominent and supportive role in future courses. However, we will need to assess more systematic feedback and gather more experience to fully realize this potential.

6. Assessing Success

Many pedagogic interventions have unexpected effects: Often, the best intentions are not sufficient to make a good idea work in practice. We generally assess the success of our own interventions through frequent testing, final exams, and direct observation, always comparing the results to those of previous courses, some of which have yielded significant empirical effects (Bieniusa et al. 2008). However, it is difficult to isolate the effects of individual changes in the teaching languages in empirical measurements. In particular, it is difficult to measure how many problems students were unable to solve because of language-design issues. Thus, we rely on direct observation in our supervised lab exercises, where our TAs log any problems the students have where the program environment or the programming language may help.

We were able to observe some specific effects, however: For example, before the introduction of signatures, most contracts written by the students contained errors, whereas afterwards, most signatures did not contain errors. The effect of properties is not empirical, as they *enable* a particular didactic approach—we believe the basic approach is already validated, as many students are able to write properties on their own, whereas the previous ADT-based approach to specification was a disaster, as students were not able to formulate properties on their own.

7. Growing Teaching Languages

While it has become clear that standard Scheme *as-is* was not an ideal teaching language, it was still a good starting point for our endeavors: Functional programming is a more appropriate beginners' paradigm than imperative or object-oriented programming; Scheme, being a functional language, supports the paradigms needed for implementing the design recipes, and its general abstraction mechanisms make it ideal for practicing abstraction. Its simple syntax makes classroom treatment easy.

Educators and implementors can improve the learning experience with any (functional) language. This requires substantial action research and observation-driven improvement as part of a long-running process, as our experience has demonstrated. Moreover, educators do well to clearly define their teaching goals. Appropriate goals are defined in terms of the actual learning experience rather than the subject coverage in class. The following principles have served us well on our journey:

- Observe your students directly and closely.
- Be willing to abandon your favorite aspects of the course or teaching language—at least be willing to move them to a different place.
- Keep making changes, evaluate them, and be willing to abandon them if they do not work.
- Cooperate with others who are doing similar work. Learn from their mistakes.

8. Related Work

There are surprisingly few constructive investigations of how particular design elements of a programming language can support or hinder a beginner's effort to learn programming. Wadler's critique of Scheme for teaching (Wadler 1987) is such a constructive investigation; Wadler stresses the importance of a type-based approach to program construction, recognizes the problems of Scheme's external representation, and the importance of algebraic techniques in understanding programs. The work on support for testing in ProfessorJ (Gray and Felleisen 2007) shows the importance of a concise and lightweight notation for tests, and thus mirrors the experience we had with test boxes and `check-expect`.

The paper by McIver and Conway (McIver and Conway 1996) identifies a number of issues in the design of languages for introductory programming. The paper aptly concludes:

> This implies that the most important tool for pedagogical programming language design is usability testing, and that genuinely teachable programming languages must evolve through prototyping rather than springing fully-formed from the mind of the language designer.

The work on Helium (Heeren et al. 2003) demonstrates the Haskell community's insight that beginners have needs different from those of professionals—specifically, that they require better (type) error messages. Also, Helium, lacking type classes, is effectively a beginner's language level for Haskell. The Helium project uses concrete observations of students' interactions with the system to improve it (van Keeken 2006). Generally, producing comprehensible type error messages in Hindley-Milner-typed languages is ongoing research (Rahli et al. 2009). Marceau et al. have recently studied the quality of the error messages in DrScheme more systematically and concluded that there is still significant room for improvement (Marceau et al. 2010). DrJava (Hsia et al. 2005) has picked up the concept of language levels from DrScheme.

9. Conclusions

The programming language used by an introductory course can be either a help to the student, or an obstacle. However, even though the typical professional functional language is less complex than the typical professional object-oriented language, problems remain. Improving this situation requires language design specifically geared towards beginning students. The properties of these languages arise from the pedagogic principles of the course—the design recipes—and continual improvement from an ongoing process and observation of the students.

The HtDP and DMdA languages have come a long way in supporting the beginning student. However, work on them is ongoing, and we believe further refinements are possible. In the near future, we will continue to work on the error messages, again following PLT's lead (Marceau et al. 2010). We have also ported the work on signatures in the DMdA levels to the HtDP levels, which will be available in a future version of DrRacket. As many signatures already look like types, we also plan to experiment with adding additional levels that treat the signatures as type declarations. Moreover, we expect experience to guide us towards further improvements. In the future, we may benefit from a more systematic approach to evaluating our success instead of our past action research. We welcome new adopters and their feedback. We call on educators who teach programming using other languages to use similar or improved processes to tailor their tools to the needs of their students.

10. Acknowledgments

Many people were involved in shaping the DMdA and HtDP language levels: Matthias Felleisen and the members of the PLT group—particularly Matthew Flatt, Robby Findler, Shriram Krishnamurthi, and John Clements—are responsible for the ongoing development of DrRacket. Martin Gasbichler helped develop the DMdA language levels. Peter Thiemann and Torsten Grust and their groups provided helpful suggestions on the design of the DMdA languages, based on their own intro courses. Carl Eastlund suggested adding randomized testing to the language levels.

References

Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., second edition, 1996.

Annette Bieniusa, Markus Degen, Phillip Heidegger, Peter Thiemann, Stefan Wehr, Martin Gasbichler, Marcus Crestani, Herbert Klaeren, Eric Knauel, and Michael Sperber. HtDP and DMdA in the battlefield. In Frank Huch and Adam Parkin, editors, *Functional and Declarative Programming in Education*, Victoria, BC, Canada, September 2008.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In Philip Wadler, editor, *Proceedings International Conference on Functional Programming 2000*, pages 268–279, Montreal, Canada, September 2000. ACM Press, New York. ISBN 1-58113-202-6. doi: http://doi.acm.org/10.1145/351240.351266.

Will Clinger, R. Kent Dybvig, Michael Sperber, and Anton van Straaten. SRFI 76: R6RS records. `http://srfi.schemers.org/srfi-76/`, September 2005.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs*. MIT Press, 2001.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The TeachScheme! project: Computing and programming for every student. *Computer Science Education*, March 2004.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul A. Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, pages 159–182, March 2002.

Matthew Flatt, Robert Bruce Findler, and PLT. *Guide: Racket*. PLT, 2010. Available from `http://pre.plt-scheme.org/docs/`.

Kathryn E. Gray and Matthias Felleisen. Linguistic support for unit tests. Technical Report UUCS-07-013 2007, University of Utah, 2007.

Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In Johan Jeuring, editor, *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop*, pages 62–71, Uppsala, Sweden, August 2003.

James I. Hsia, Elspeth Simpson, Daniel Smith, and Robert Cartwright. Taming Java for the classroom. In *SIGCSE 2005*, February 2005.

Herbert Klaeren and Michael Sperber. *Die Macht der Abstraktion*. Teubner Verlag, 1st edition, 2007.

Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmiers. In *2010 Workshop on Scheme and Functional Programming*, Montréal, Québec, Canada, August 2010.

Linda McIver and Damian Conway. Seven deadly sins of introductory programming language design. In *Proceedings Software Engineering: Education & Practice*, pages 309–316, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

Vincent Rahli, J. B. Wells, and Fairouz Kamareddine. Challenges of a type error slicer for the SML language. Technical Report HW-MACS-TR-0071, Heriot-Watt University, School of Mathematics & Computer Science, September 2009.

Peter van Keeken. Analyzing Helium programs obtained through logging — the process of mining novice Haskell programs —. Master's thesis, Utrecht University, October 2006. INF/SCR-05-93.

Philip Wadler. A critique of Abelson and Sussman or why calculating is better than scheming. *SIGPLAN Notices*, 22(3):83–94, March 1987.