Language Support for Multi-Paradigm Programming

Matthias M. Hölzl

hoelzl@informatik.uni-muenchen.de
Institut für Informatik
Ludwig-Maximilians-Universität
Oettingenstraße 67
80538 München

Abstract. One goal of programming language research should be to design languages that help programmers build better software. We argue that designing languages that are flexible enough to integrate different programming paradigms and expressive enough to allow the programmer to state his intention is a possible way toward this goal.

Complexity is *the business we are in, and complexity is what limits us.*

—F. P. Brooks, Jr.

1 Introduction

Where should language design be heading? One reasonable answer might be: "Toward languages that help us build better software."

How can a programming language help us to build better software? Well, it depends: Do you use RUP or XP as development process? How large is your team? How experienced are your programmers? What kind of software do you develop?... we therefore have to agree about a target audience for the language before we can answer this question.

The goals that governed the development of Java are clearly stated by James Gosling and Henry McGilton *"Primary characteristics of the Java programming language include a simple language that can be programmed without extensive programmer training [...]"* [4]. This is a reasonable point of view to adopt when designing a language for a broad audience and the success of Java shows that its designers were successful in realizing their goals.

The downside of this philosophy is that many powerful but dangerous language features were omitted from the language. Java is a simple language, but for the advanced programmer programming in Java is simply frustrating. I will therefore focus on the design of a language for a very different audience: Experienced programmers working with an agile development process. I expect them to be comfortable with abstractions and to posses a certain amount of mathematical knowledge.

So, how can a programming language help us to build better software?

- It can help us to build the right software.
- It can help us to reduce the complexity of software development.
- It can help us to develop the software as fast as possible.
- It can help us to keep the software changeable.
- It can help us to produce reliable software.

```
final File myFile = new File("test.log");
try {
  final Logger myLogger = new Logger(myFile);
  try {
    myLogger.log("...");
  } finally {
    myLogger.close();
} finally {
  myFile.close();
}
```

Fig. 1. Freeing operating system resources.

If you look closely at these goals you will see that the second goal implies all others except for the first one: If we can avoid the trap of exponentially increasing complexity we will achieve many other goals for free.

Many language designers consider languages that are "pure", i.e., that rely on a single programming paradigm, to be highly desirable. For example, James Gosling and Henry McGilton state the following goal for the design of the Java language: *"Java has no functions. Object-oriented programming supersedes functional and procedural styles. Mixing the two styles just leads to confusion and dilutes the purity of an object-oriented language."* [4].

In my opinion, a "pure" language *has* to be restricted in its expressiveness; it is highly unlikely that we will find the one paradigm that can express all problems equally well. Since I consider the expressiveness of a language to be more important than some concept of "purity" I think it is more promising to investigate the development of multi-paradigm programming languages.

There certainly is no "silver bullet" [3], but there are ways to keep complexity under control. This is borne out by the development of the *ConS/** family of languages (see [6, 5]). The *ConS/** family of languages extends functional or object oriented languages with facilities for non-deterministic computation and constraint solving. I implemented prototypes based on Scheme, Dylan and Common Lisp.

> *What can be said at all can be said clearly; and whereof one cannot speak thereof one must be silent.*
>
> —L. Wittgenstein

2 Expressing Intentions

Syntactic Extensibility. A recent article in a reputable German computer journal contained code along the lines given in Figure 1 to illustrate the freeing of operating system resources[1]. This solutions exposes an incredible amount of superfluous detail which cannot be eliminated in Java. The right solution ought to look like this:

```
withLogger (myLogger = "test.log") {
  myLogger.log("...");
}
```

A simple syntactic transformation decreases the number of lines by 66% and increases the expressiveness of the resulting code by a similar amount. And we have eliminated one possible source of obscure errors.

[1] Actually the code given in the article contained some mistakes; the above solution was given in a letter to the editor.

Syntactic abstraction is widely used in Lisp, but while there has been some research on syntactic transformations for infix languages, e.g., [1], no completely satisfactory mechanism has been proposed so far. Experience with *ConS/** has shown a problem with purely local syntactic abstractions: When designing a large language extension it is often useful to be able to inspect the surrounding context and inspect type or visibility information before deciding how to expand a construct. This hints at the necessity to export a clean user-visible interface to the first stage of the compiler.

The previous example illustrates a principle that all language designers should keep in mind: *Allow the programmer to express his intentions.* This principle has the following corollary: *Do not force the programmer to say something he does not want to say.*

Type Systems. We can illustrate these two principles with another example: the type system of the programming language. Most current object oriented languages violate both principles at once: they force the programmer to declare a type for every variable and parameter (even if he is just exploring the problem and does not know the correct type yet) and they identify types with classes which makes it rather hard to express many useful concepts. The following are all useful statements that a programmer might want to assert about a variable:

- This variable is either a `Foo` or `null`.
- This variable is a `Foo` (and it is not `null`).
- This variable is one particular instance of the class `Foo`.
- This variable is an integer between 35 and 42.

In most languages you can express only the first of these properties.

To increase the power of the type system it is useful to differentiate between *types* and *classes*. With this distinction it becomes possible to regard the union of two types as a type. This is not possible if types correspond to classes[2].

In Common Lisp it is possible to express all the above judgments[3]:

- `(type (or foo null))`
- `(type foo)`
- `(type (member` *`the-instance`*`))`
- `(integer 35 42)`

Any type system that allows the last judgment can obviously not be decidable, and so Common Lisp has to revert to run-time type checks for some type assertions. The programming language Cecil [2] offers optional static typing with polymorphic types and F-bounded polymorphism but no integer ranges. In section 4 we show how an undecidable type system can provide static type safety when it is tightly integrated into a programming language environment.

> *Lisp is a programmable programming language*
>
> —J. Foderaro

3 Programmable Programming Languages

Any given language can only support a few different paradigms. What if the one we need for our current problem is not included? To me, the most promising solution is

[2] Except for the case where two types are the only subtypes of a common super type.

[3] I have substituted the symbol `nil` for `null`, since there is no value that directly corresponds to `null` in Common Lisp. The type specifier `null` denotes the type that only contains the value `nil`.

to make the programming language flexible enough to incorporate new paradigms. John Foderaro coined the term "programmable programming language" for this property.

Lisp has indeed shown an amazing capability to incorporate new facilities into the language: Both the exception system and the Common Lisp Object System were originally implemented as "user level" extensions of the language. However, the development of *ConS/Scheme* and *ConS/Lisp* showed several limitations of the extensibility.

Advanced Control-Flow Manipulation. Adding non-determinism to Scheme is rather easy: Scheme offers first-class continuations with unlimited extent which can be used to implement the basic language constructs for a non-deterministic language in about 100 lines of code. This implementation, however, is only suitable for a prototype because it interacts badly with any other library using first-class continuations. Since, e.g., most exception systems for Scheme are based on first class continuations, the prototype implementation disrupts important system facilities like exception handling. Adding non-determinism to Dylan and Common Lisp required extensive modifications of the respective compilers. To implement new programming paradigms that require non-standard flow of control (e.g. forward or backward chainers, resolution systems) it would be very beneficial to develop a language facility that offers the same power as first class continuations for control-flow manipulation but that can be used safely by different libraries even if they are used in the same program.

We already discussed the need for syntactic extensibility in section 2.

4 Environment Support

It is indeed noteworthy that the alleged productivity of many RAD languages (like Visual Basic, Smalltalk or Lisp) is at least as much a function of the sophisticated development environments than of language features. On the other hand, at least in the case of Smalltalk and Lisp the quality of the programming environments is intimately related to fundamental features of the programming languages, most importantly to the reflective capabilities of the language. Therefore we regard some language features as essential, even if they are not commonly used in delivered programs: Reflection and run-time evaluation are capabilities that would be worthwhile for the benefit of the development environment even if they were rarely used in user level programs. On the other hand, many language features are only possible if they are supported by the right tools in the environment: While dynamic typing is convenient if an interactive top-level is available, it does not work well in an offline setting.

Socratic Reasoner. Theorem proving and model checking are mostly regarded as tool that have no practical applicability in software development, except in the case of safety critical systems. In my opinion this is mostly due to overambitious goals: It is indeed very hard to verify the correctness of a complete program. But this does not mean that an interactively guided theorem prover (a so-called Socratic reasoner [7]) cannot be used to prove assertions about the program behavior. Currently available interactive theorem provers like, e.g., ACL2 are relatively easy to use. In many cases it is quicker to prove some properties about an ACL2 program than to write the necessary test cases to check this property. The mechanism of existing theorem provers cannot be used directly to reason about object oriented programs, but it would certainly be possible to adapt them to object oriented languages.

Declarative information. Currently most of the information about the program is discarded when the implementation is written: We use a hash table because we expect a certain performance characteristic, but the reader of the program has

to infer this information from his own knowledge of data structures. The language should allow the programmer to make these high-level decisions explicit in the code. This also provides a step in the direction of making larger reusable components of code possible: If the programming environment knows the expected usage of a component it can adjust the data structures of the component to fulfill these expectations.

Undecidable Static Typing. Any sufficiently powerful static type system will be undecidable. In a system that provides an integrated Socratic reasoner a language with undecidable type system can in practice still be statically proved to be type correct. The PVS specification system is such a case: When type-checking a proof, the system may possibly generate so called *type correctness conditions (TCCs)* and submit them to the built-in theorem prover for verification. When the system is not automatically able to prove all TCCs it asks the user for advice. While it is theoretical possible that the system might generate very hard proof obligations for TCCs, practice shows that the theorem prover needs little assistance and those proof obligations that it cannot prove automatically require only minimal user intervention or point out a true problem in the specification.

Architecture Information. This is similar to the previous point: A programming language should allow the programmer to specify the architectural of the program (e.g., in the form of in/out connectors). The environment should provide facilities to extract this information from prototypes.

Acknowledgments

I want to thank Martin Wirsing for his comments about an early draft of this paper. I also want to thank my colleagues at PST, especially Hubert Baumeister, Alexander Knapp, Philipp Meier and Axel Rauschmayer for many helpful discussions.

References

1. Jonthan Bachrach and Keith Playford. The Java syntactic extender (JSE). In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, pages 31–42. ACM, 2001.
2. Craig Chambers. The Cecil language: Specification & rationale. Technical report, University of Washington, 1995.
3. Jr. Frederik P. Brooks. *The Mythical Man-Month (*20^{th} *anniversary edition).* Addison-Wesley, 1995.
4. James Gosling and Henry McGilton. The Java language environment— a white paper, 1996.
5. Matthias M. Hölzl. ConS/Lisp—a MOP-based non-deterministic lisp. In Raymond de Lacaze, editor, *Proceedings of the International Lisp Conference*, 2002.
6. Matthias M. Hölzl. Constraint-functional programming based on generic functions. In Slim Abdennadher, Thom Frühwirth, and Petra Hofstedt, editors, *Proceedings of the First International Workshop on Multiparadigm Constraint Programming Lanugages (MultiCLP)*, 2002.
7. Stuart Russel and Peter Norvig. *Artificial Intelligence, A Modern Approach, second edition.* Prentice Hall, 2003.