Embedding Languages Without Breaking Tools*

Lukas Renggli, Tudor Gîrba, Oscar Nierstrasz

Software Composition Group, University of Bern, Switzerland
`http://scg.unibe.ch/`

Abstract. Domain-specific languages (DSLs) are increasingly used as embedded languages within general-purpose host languages. DSLs provide a compact, dedicated syntax for specifying parts of an application related to specialized domains. Unfortunately, such language extensions typically do not integrate well with the development tools of the host language. Editors, compilers and debuggers are either unaware of the extensions, or must be adapted at a non-trivial cost. We present a novel approach to embed DSLs into an existing host language by leveraging the underlying representation of the host language used by these tools. HELVETIA is an extensible system that intercepts the compilation pipeline of the Smalltalk host language to seamlessly integrate language extensions. We validate our approach by case studies that demonstrate three fundamentally different ways to extend or adapt the host language syntax and semantics.

1 Introduction

General purpose languages, by being "good enough" to code software for arbitrary domains, are necessarily suboptimal for many specialized domains. They may be overly verbose, confusing or just plain awkward to use. Many DSLs have been developed to address the needs of these specialized domains, but most of these languages typically do not integrate well with the host language and tools, making it clumsy to develop and debug programs written in these languages.

Further complicating matters, multiple DSLs may be active within the same application software. These DSLs may either be globally available, or they may be *context-dependent*, being active only within selected packages or classes.

DSLs come in many flavors. At one extreme we have so-called *internal DSLs* which simply make creative use of APIs and of the host syntax. Such DSLs are sometimes referred to as *fluent interfaces* [1]. They provide a seamless integration in the host language, and as such they can benefit from the tools provided by the development environment (*e.g.,* code editor, debugger) of the host language. However, the expressiveness of internal DSLs is confined by the host syntax. At the other end of the spectrum we find *external DSLs* [2]. These are typically developed as a preprocessing step or through an extensible compiler, thus providing freedom for expressing diverse syntax and semantics. However, while doing so they break the tools of the host development environment.

* In Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP 2010), LNCS 6183, pp. 380–404, Springer-Verlag, 2010.

In between these two extremes we find *embedded DSLs*, which extend a host language with new syntax and semantics. Language workbenches support the development of embedded DSLs by introducing a common representation for all languages and by integrating multiple languages into a common toolset. Mernik *et al.* [3] point out that such an embedded approach [4] leads to better reuse of existing host language features and tools, and significantly reduces development and training costs. In practice however, language workbenches do not leverage the existing tools but provide their own environment. Often they introduce a non-standard language representation and thus pose compatibility problems with existing code.

Table 1. Taxonomy for Pidgin, Creole and Argot embedded languages.

	Syntax	**Vocabulary**	**Semantics**	**Description**
Pidgin		✓	✓	A pidgin is a simplified form of the host language. It introduces a new vocabulary and new semantics to the code.
Creole	✓		✓	A creole changes the syntax of the host language and defines new semantics.
Argot			✓	An argot switches the semantics of the existing language, without affecting its syntax.

We propose to address these shortcomings by developing an approach that enables multiple, context-dependent embedded DSLs that leverage existing tools. Embedded languages extend a host language by adapting the existing syntax and semantics, or by introducing new syntax. In Table 1 we identify essentially three different ways this can be done:[1]

Pidgin. A *pidgin* bends the syntax of the host language to extend its semantics [5]. This kind of embedded language reuses a limited part of the host syntax and combines it with a new vocabulary.

Creole. A *creole* introduces a completely new syntax by defining its own grammar and a custom transformation to the host language that defines the semantics.

Argot. An *argot* uses the existing host language syntax, but changes its semantics. An argot reinterprets the semantics of valid host language code, whereas

[1] In the domain of natural language, a “pidgin” is “a grammatically simplified form of a language, used for communication between people not sharing a common language”; a “creole” is “a mother tongue formed from the contact of two languages through an earlier pidgin stage”; an “argot” is a “jargon or slang of a particular group or class” [New Oxford American Dictionary].

pidgin code is only syntactically correct host code — it has meaning only for the pidgin.

An embedded language must either introduce new syntax to the host language for the concepts it introduces (a creole), or it must adopt the host syntax as is. If the host syntax is reused, it must either be overloaded, reinterpreting the syntax in a novel way (pidgin), or it must alter the semantics of the host (argot). A fully general approach to integrating new embedded languages into an existing host language and environment must therefore support all three classes of embedded language.

Our approach. In this paper we present a language workbench called Helvetia for defining embedded languages and for integrating them into the host language. Helvetia accommodates new languages through extension points of the existing compiler and tools. All languages are transformed and represented in terms of the abstract syntax tree (AST) of the host language. The transformations are expressed as rules and they can be scoped to various contexts. Furthermore, these rules can be active at the same time, allowing us to embed different languages into a common host language, and to integrate them into the host environment and its tools. Since our approach builds on top of the existing infrastructure of the host language, existing tools, such as editors and debuggers, continue to work with minimal adaptation. Helvetia provides the necessary low-level infrastructure for *Language Boxes* [6], an adaptive language model for fine-grained language changes and language composition.

Outline. Section 2 discusses the related work and enumerates the shortcomings of these approaches. In Section 3 we present concrete language extensions that exemplify the three types of languages our solution supports. Section 4 illustrates how the different types of embedded languages can be specified with Helvetia. In Section 5 we explain how Helvetia leverages the host toolchain to seamlessly embed new languages. Section 6 evaluates our approach in comparison with related work, and in Section 7 we summarize the paper and discuss future work.

2 Related Work

We review the state of the art in systems for authoring embedded languages to establish the open challenges facing a new approach. In particular, we consider how these systems support the development of pidgin, creole and argot embedded languages, and we assess how well these systems facilitate integration of embedded languages with their respective host language and host language tools. Furthermore, we compare the existing approaches with respect to the following properties of language embedding:

Multiple Context-Dependent Languages. Switching between different languages should be possible at arbitrary points and not enforce the use of special syntactic markers. It should be practicable to mix and match different language extensions and the host language. Language changes should be scopable at a fine-grained level, to make it possible to use several otherwise conflicting language extensions in the same compilation unit.

Homogeneous Tool Integration. A uniform tool set is important to software engineers. To ease the development and use of embedded languages all development activities should happen in the same familiar programming environment of the host language. No special code browser, editors, debuggers or source control should be necessary; all existing tools should continue to work transparently with different languages. To facilitate debugging a precise bi-directional connection between the original source, the various transformation stages and the final executable code should be maintained.

Homogeneous Language Integration. The meta-language used to specify new language features should be the same as the host language [7]. Homogeneous language integration is central for several reasons: A homogeneous system is a requirement to transparently pass values between the different meta-levels and to use reflection to reason about static and dynamic structure of the application [8].

Table 2 provides an overview of the related work split into four categories. For each considered system we indicate the host language, the capability of defining pidgins, creoles and argots, and the support for the aforementioned characteristics. The following sections offer details for each individual system.

2.1 Extensible Compilers

Extensible Compilers are best described as open toolboxes that provide entry points into the toolchain to extend and change the host language.

The *Java Annotation Processing Tool* (APT) enables a compile time, read-only view of the Java program structure. ASTs can be transformed only using a private API, which is not supported and may be subject to change or deletion. As such, argots and pidgins can be implemented, but a creole would require an additional preprocessing step. In a similar way people are using the weaving mechanism of AOP to achieve semantic changes for pidgins and argots.

ableJ [9], *Dryad* [10], *JastAddJ* [11], and *Polyglot* [12] are extensible Java compiler frameworks and provide the necessary infrastructure to build argots, pidgins and creoles. Language extensions are composable and modular, and are transformed into Java for execution. Most extensible compilers define the syntax changes using a external language definition. The Dryad compiler uses bytecode as its central representation, and thus it is not homogeneous as well. None of the systems offers tight IDE integration, and the transformed code cannot be debugged at the source level.

Xoc [13] is an extensible C compiler. Xoc uses a source-to-source translator that reads the input, analyzes and transforms it, to eventually generate standard C code. The system provides no reflective facilities and no tool integration, neither at compile time nor at run time.

Table 2. Comparison of different systems for embedded language authoring. We indicate custom host languages with parentheses.

Type	System	Host Language	Pidgin	Creole	Argot	Multiple Languages	Homogeneous Tools	Homogeneous Language
	Helvetia	Smalltalk	✓	✓	✓	✓	✓	✓
	Java Annotation Processing	Java	✓	·	✓	✓	·	✓
	ableJ	Java	✓	✓	✓	✓	·	·
Extensible	Dryad	Java	✓	✓	✓	✓	·	·
Compilers	JastAddJ	Java	✓	✓	✓	✓	·	·
	Polyglot	Java	✓	✓	✓	✓	·	✓
	Xoc	C	✓	✓	✓	✓	·	·
Meta	Cola	(various)	·	✓	·	✓	·	✓
Programming	Converge	(Python)	·	✓	·	✓	·	✓
Systems	MetaOCaml	OCaml	✓	·	✓	✓	·	✓
	Scheme	Scheme	✓	·	✓	✓	✓	✓
	JetBrains MPS	(Java)	·	✓	·	✓	✓	✓
	Intentional Software	(C#)	?	✓	?	?	✓	?
Language	openArchitectureWare	Java	·	✓	·	·	·	·
Workbenches	Java Development Tools	Java	·	·	·	·	✓	·
	IDE Metatooling Platform	Java	·	·	·	·	✓	·
	WholePlatform	Java	·	✓	·	·	·	·
	Katahdin	(C#)	·	✓	·	✓	·	✓
	Ceteva XMF	(Java)	·	✓	·	✓	·	✓
Language	Khepera	C	✓	✓	✓	·	·	·
Transformation	MontiCore	Java	·	✓	·	·	·	·
Systems	MetaBorg		✓	✓	✓	✓	·	·

2.2 Meta-Programming Systems

Meta-Programming Systems are programming languages that come with meta-programming facilities targeted at code generation.

Cola [14] implements an open object model for experimentation with different programming paradigms. Cola is bootstrapped in itself using a Smalltalk-like language called Pepsi. Jolt is a Lisp-like language that serves as a common abstract syntax and executable representation for other languages. OMeta [15] is an object-oriented pattern matcher based on Parsing Expression Grammars that is used to transform new languages to Jolt. Even if all languages are built on top of the same infrastructure, the authors do not provide mechanisms to easily embed them into each other. Furthermore there is no common tool support for editing and debugging the different languages.

Converge [16] is a dynamic programming language resembling Python. A special block construct `$<<language>>` is used to embed languages into the source code. The creation of argots and pidgins, *i.e.,* the modification or extension of existing languages, is not supported. Meta-programming is possible at compile time only. There is no IDE integration.

MetaOCaml [17] uses multi-stage programming to generate and transform code at runtime. Similar quoting mechanisms are available in programming languages like *Scheme*, Lisp or Template Haskell, but these mechanisms alone do not allow new syntax to be introduced. Both system have powerful macro programing constructs, making it possible to tweak the default towards pidgins or argots. Typically these kind of systems are used together with traditional text editors and thus do not allow an easy adaptation to language changes. Debuggers are available.

2.3 Language Workbenches

Language Workbenches are characterized by a specialized IDE with a well-defined workflow to specify and use different languages. Language designers are required to follow clearly defined steps to describe syntax, semantics and editor behavior of a new language.

The *Meta Programming System* (MPS) by JetBrains [18] and *Intentional Software* [19] both provide a programming environment to define new languages and to change existing ones. Neither system uses text representation for source code, but instead they provide a graphical cell editor that maps valid programs directly to an underlying abstract code representation. MPS defines new languages using different concepts for structure (semantic model), editor (parser), constraints, behavior, type systems, data flow and code generators for Java. MPS 1.1 does not come with a source level debugger; debugging and error reporting happens at the level of the generated Java code. Similarly Intentional Software requires language developers to define edit, display, and transformation concerns for language extensions. As no product previews and no detailed documentation is available, the exact properties of this system are not clear. For example it is unknown how closely the system integrates with the host language and if it is possible to transparently step with a debugger through different languages.

openArchitectureWare and *Whole Platform* [20] are language workbenches that are tightly integrated into the Eclipse platform. In both cases templating systems are used to generate executable Java code. openArchitectureWare provides a strong integration with the Eclipse meta-modeling facilities, such as the Eclipse Modeling Framework (EMF) and the Graphical Modeling Framework (GMF). They both provide basic support for editor integration such as syntax highlighting and code completion of textual languages. However, both systems lack debugging support and the ability to change the semantics of their host language.

The *Java Development Tools* (JDT) provide the basic tools to build Eclipse plugins. The *IDE Metatooling Platform* (IMP) [21] is an extensible IDE architecture for the Eclipse platform. Contrary to all other tools listed in this section JDT and IMP do not provide functionality for language design and embedding themselves. The only purpose of IMP is to closely integrate existing languages into the Eclipse IDE using a service architecture. At the time being there is no support for interaction with language runtimes and debuggers.

Katahdin [22] is a programming language implemented on top of C#. Katahdin is a dynamically typed language that syntactically resembles C#. New constructs such as expressions or statements are defined by subclassing existing parse-tree nodes that can then be added to the host language at runtime. Languages can be enabled on a per-file basis, or can be integrated into the host language by extending it with a specific keyword such as `language { ... }` and connecting the two grammar-trees. The semantics of user-defined parse-tree nodes is specified by overriding certain methods in the respective node-classes. Code is interpreted by traversing the parse-tree nodes and calling methods defining the semantics. There is a simple debugger available visualizing the internal parse-tree structure of the interpreter. The debugger is not able to work at the level of Katahdin programs.

The Extensible Programming Language (XMF) by Ceteva is a specification of a “superlanguage” [23]. A superlanguage is characterized through usability (interactive, dynamic, reflection, interfaces), expressiveness (high-level, dynamic typing, garbage-collection), and extensibility (aspects, reflexive, extensible syntax). XMF is written in itself, and allows one to easily define new languages. A special `@language` construct is used to switch between different languages. Although a Java interface is available, XMF uses its own proprietary virtual machine written in Java. XMF does not provide an IDE integration.

2.4 Language Transformations

Language transformation systems define languages through the transformation and composition of language models.

Khepera [24] is a preprocessor that transforms source-to-source and pretty-prints the generated code to C before compiling with a traditional compiler. It parses input into an abstract syntax tree and performing complex tree-based analysis and manipulation. All transformations preserve the knowledge of the origin of each node. Thus, in theory Khepera makes it possible to develop debuggers for new languages. However, the system provides no ready-to-use IDE or debugger. Furthermore, it does not support multiple languages to be used simultaneously.

MontiCore [25] provides a framework for language inheritance and language embedding. MontiCore has its own syntax to define grammars and their mapping to Java types. The parser is created using the ANTLR [26] parser generator. The abstract syntax tree is automatically derived from the grammar. Language inheritance allows one to subclass existing grammars to modify and extend. Language embedding is achieved by manually introducing a superordinate parser for every pair of languages that are used together. ANTLR has been patched to support swapping between the grammars on the fly. Visitors are used to add new behavior for productions, to generate code and to build editors for Eclipse. There is no IDE integration or support for debugging.

MetaBorg [27] is a method for embedding DLSs and extending existing languages. MetaBorg is based on the Stratego/XT [28] toolkit, a language independent program transformation engine, hence there is no integration into development environments and model level debuggers of the host language.

3 Helvetia Exemplified

In this section we introduce three cases of language extensions. The first two cases take the same underlying API of a graphical engine and transform it into a pidgin (Section 3.1) and a creole (Section 3.2). We then describe the use of an argot for introducing a transactional memory mechanism without changing the syntax of the host language (Section 3.3).

The prototype of Helvetia[2] and the examples presented in this paper are implemented in Pharo[3], an open-source Smalltalk [29] dialect. Readers unfamiliar with the syntax of Smalltalk might want to read the code examples in the following sections aloud and interpret them as normal sentences.

An invocation to a method named `method:with:`, using two arguments looks like: `receiver method: arg1 with: arg2`. The semicolon separates messages that are sent to the same receiver. For example, `receiver method1: arg1; method2: arg2` sends the messages `method1:` and `method2:` to `receiver`. Other syntactic elements of Smalltalk are: the dot to separate statements: `statement1. statement2`; square brackets to denote code blocks or anonymous functions: `[ statements ]`; single quotes to delimit strings: `'a string'`; and double quotes delimit comments: `"comment"`. The caret `^` returns the result of the following expression.

3.1 A Pidgin: Mondrian

Mondrian [30] is a graph based visualization framework that provides a declarative Smalltalk API for users to specify new visualizations and compose existing ones.

One of the features of Mondrian is an API to compose custom shapes out of basic ones, called FormsBuilder. The FormsBuilder is inspired by CSS 3 and uses a grid to align primitive graphical elements such as text labels and boxes. For example, the code below in Listing 1 creates a UML package shape as depicted in Figure 1. The package shape is built from a 2×2 grid.

The first column and row are told to `grow` to enclose their children. The second column and row are told to `fill` the remaining space. In cell (1, 1) we place a bordered `LabelShape`. In cell (1, 2) we place a bordered `RectangleShape` that spans two horizontal cells.

[2] The implementation along with its source code and examples can be downloaded from `http://scg.unibe.ch/research/helvetia`.

[3] `http://www.pharo-project.org`

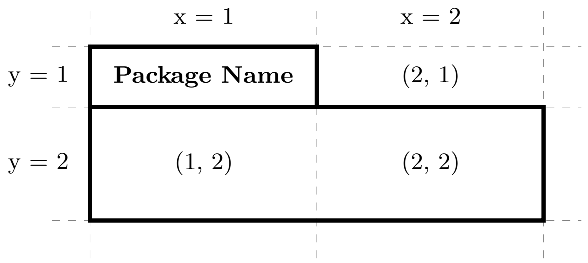


Fig. 1. A UML package shape in Mondrian.

```
aBuilder row grow.                                      " defines row sizing "
aBuilder row fill.

aBuilder column grow.                                 " define column sizing "
aBuilder column fill.

aBuilder x: 1 y: 1 add: (LabelShape new                   " define the cells "
   text: [ :each | each name ];
   borderColor: #black;
   borderWidth: 1;
   yourself).
aBuilder x: 1 y: 2 w: 2 h: 1 add: (RectangleShape new
   borderColor: #black;
   borderWidth: 1;
   width: 200;
   height: 100;
   yourself)
```

Listing 1. Traditional Mondrian Forms Builder API.

Mondrian provides an internal DSL that offers a high-level interface for composing visualizations. While it makes the composition easy, there is still a considerable amount of syntactic noise that makes the script hard to read.

We incrementally bend the syntax of the host language towards a more suitable DSL, first by creating a pidgin, and then by creating a creole. Our final goal is to be able to define the visualizations using a simple syntax resembling cascading style-sheets (CSS), which offers a compact notation to programmers and designers to declaratively specify layout and design of web sites.

If we have a look at the code in Listing 1 we see that the noise is caused by certain semantic elements that are required to make this example run as Smalltalk code conforming to the original Mondrian API. We discover three things that are repetitive and that could be simplified:

1. The variable `aBuilder` is referenced in every rule as an entry point to construct and configure the different parts of the forms.

2. The specification of the cells and their content is repetitive and rather hard to read.
3. The instantiation of different shapes is cumbersome as in this case the host language syntax is rather verbose.

The following code addresses these issues:

```
row = grow.
row = fill.
column = grow.
column = fill.
(1 , 1) = label
         text: [ :each | each name ];
         borderColor: #black;
         borderWidth: 1.
(1 , 2) - (2 , 1) = rectangle
         borderColor: #black;
         borderWidth: 1;
         width: 200;
         height: 100.
```

Listing 2. Pidgin: Eliminating syntactic noise.

While the above code is syntactically valid and is parsed by the standard Smalltalk parser, it is not semantically valid. For example numbers do not implement a , method, and `column` is an unknown variable.

In our implementation (described in Section 4.1), the above pidgin example is transformed transparently into the code from Listing 1. This kind of transformation simplifies the amount and complexity of source code significantly. The transformation is specified at the AST level using two transformation rules that are applied by the compiler after parsing.

3.2 A Creole: Mondrian

The pidgin shows an improvement over the original Smalltalk code, but our goal is to obtain an even more concise CSS-like language as in the listing below:

```
shape {
   cols: #grow, #fill;
   rows: #grow, #fill;
}
label {
   position: 1 , 1;
   text: [ :each | each name ];
   borderColor: #black;
   borderWidth: 1;
}
rectangle {
   position: 1 , 2;
   colspan: 2;
   borderColor: #black;
   borderWidth: 1;
   width: 200;
   height: 100;
}
```

Listing 3. Creole: A CSS-like syntax.

The code above does not follow Smalltalk syntax. At this point, the assumption of a pidgin relying on the host syntax starts to get in our way.

The solution is to allow the definition of a new parser that handles the creole syntax. We typically also want to integrate the new language constructs with the host language or with other language constructs. In our example, the code `text: [ :each | each name ]` provides such a case in which we parameterize the shape specification with a Smalltalk expression.

As shown in Section 4.2, Helvetia offers a mechanism for writing a custom parser that can also include productions external to the language at hand. Like this we can accommodate any syntax.

3.3 An Argot: Transactional Memory

In our previous work [31] we have presented a solution for introducing software transactional memory (STM) [32,33] at the language level of dynamic programming languages without requiring changes to the virtual machine. We achieved this by patching the compiler. As a result we were able to make all applications, libraries and system code transaction-aware. However since our changes at the compiler level were rather ad hoc, we lost the ability to use the debugger within a transaction.

Introducing STM into an existing language provides a concrete use case for changing the execution semantics without changing the syntax of the language. A piece of library code that is used as part of transactional code should continue to work without requiring any adaptation. Unlike a pidgin, which bends the host syntax in ways that break the semantics, an argot more subtly reinterprets the semantics of otherwise valid code.

In the example below the global counter `value` is incremented by `1`. The code does not ensure mutual exclusion, thus it might happen that some of the updates are lost when multiple threads run this method concurrently.

```
incrementBy: anInteger
   value := value + anInteger
```

When running the above method from within a transaction, the change is deferred to the end of the transaction, instead of incrementing the variable immediately. This allows the system to check for conflicts and revert the changes if necessary. Thus, even if the source code looks exactly the same, its behavior changes. We describe the transformations applied to transactional code in Section 4.3.

4 Specifying Embedded Languages with Helvetia

In this section we present the specifications of the three examples given in the previous section.

4.1 Specifying the Mondrian Pidgin

The syntax of the Mondrian pidgin can be parsed by the traditional parser of the Smalltalk host language. However, we need to apply several transformations to get the semantics right. We define a set of transformation rules that are applied by Helvetia after parsing the code from Listing 2:

```
1  MondrianPidgin class>>rowColumnTransformation
2     <transform>
3     ^ TreeRule new
4        expression: 'row = `@expr';
5        expression: 'column = `@expr';
6        action: [ :ast |
7           ast swapWith: ``(aBuilder
8              `,(ast receiver)
9              `,(ast at: '`@expr')) ]
10
11 MondrianPidgin class>>cellTransformation
12    <transform>
13    ^ TreeRule new
14       expression: '(`@x , `@y) = `@expr';
15       expression: '(`@x , `@y) - (`@w , `@h) = `@expr';
16       action: [ :ast |
17          ast swapWith: ``(aBuilder
18             x: `,(ast at: '`@x')
19             y: `,(ast at: '`@y')
20             w: `,(ast at: '`@w' ifAbsent: [ 1 ])
21             h: `,(ast at: '`@h' ifAbsent: [ 1 ])
22             add: ``(`,(Shapes at: (context at: '`var') name)
23                new `,(ast at: '`@expr'))) ]
```

The transformation rules are split into two methods. Each of these methods is tagged with the method annotation `<transform>` (lines 2 and 12), so that the compiler knows that it has to apply these transformations before performing semantic analysis. Each rule consists of two match expressions (lines 4–5 and 14–15) to find particular parse-tree nodes. This functionality is part of the Refactoring Engine [34] and is provided by the host environment. In our context these patterns match the specific constructs we introduced in Listing 2.

The action blocks (lines 6–9 and 16–23) perform a transformation on the matched AST node. For example the first action block transforms expressions of the form `row = grow` into `aBuilder row grow`. It does so by using a syntactic extension of Smalltalk with partial evaluation [35]. Everything that follows the quasiquote meta-character ``` `` ``` is delayed in execution and represents the AST of the enclosed expression at runtime. Similarly everything that follows the unquote meta-character `` `, `` is again executed when performing the code and is used to combine smaller delayed values (*e.g.,* from matched AST nodes) to larger ones. A third operator to compile, evaluate and splice in the result at compile-time is available too, but not used in the examples of this paper.

The two mentioned methods are all that is needed to implement the Mondrian pidgin. The `swapWith:` method call replaces the matched AST node with the new code. Since all AST nodes carry information about their original source origin, a debugger is able to step through and properly highlight the recomposed code fragments. Newly generated code is marked as hidden, so that the user of the pidgin does not see it in the debugger.

4.2 Specifying the Mondrian Creole

In contrast to a pidgin, a creole requires a custom parser and Helvetia offers the possibility to define one. For example, for the creole we presented in Listing 3 we define the following grammar rules defined as individual methods of the class `CSSParser`:

```
CSSParser>>rules = { rule }
CSSParser>>rule = selector "{" declarations "}"
CSSParser>>selector = #identifier
CSSParser>>declarations = declaration { ";" declaration }
CSSParser>>declaration = #keywordMessage
```

This grammar looks very similar to Extended Backus-Naur Form (EBNF) [36]. In fact, it is a DSL for parser generators implemented in Helvetia. As an extension to EBNF we allow productions to reference grammar rules of other languages. The name of external grammar rules are prefixed with a hash character `#`. For example, the CSS selector is simply a Smalltalk identifier, and the declaration of a property is a keyword message (a Smalltalk method name with arguments, but without receiver) of the host language.

Next we create a new subclass of `CSSParser` called `CSSTranslator`, to reuse the abstract grammar definition and to augment it with productions to transform the parse tree nodes to the host language AST [37]. Again we use quasiquoting to build the AST of the host language. Two of `CSSTranslator`'s parse tree transformations look like in the following listing. The other grammar productions are similarly defined.

```
CSSTranslator>>rules
   ^ super rules ==> [ :ast | ``(buildOn: aBuilder `,ast) ]

CSSTranslator>>rule
   ^ super rule ==> [ :ast | self transform: (ast at: 'selector')
     declarations: (ast at: 'declarations') ]
```

This assigns semantic actions to the productions defined in the superclass. The argument `ast` is a collection of parse nodes built by the grammar productions

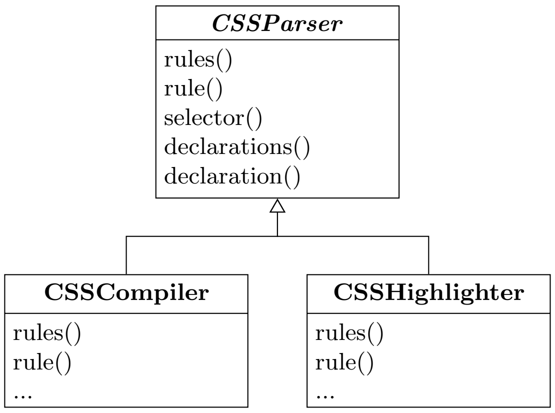


Fig. 2. The CSS Parser Hierarchy.

of the superclass. Lines 3–4 use quasiquoting to define the method header and to embed the AST nodes of the rules into its body. Lines 7–9 call the helper method `build:declarations:` with the selector token and a collection of declaration messages to build a Smalltalk AST.

To tell the system to use our custom parser instead of the default one, we use a method annotation `<parse>` on the classes where we want to use the custom syntax. The code in Listing 3 is parsed, transformed and eventually compiled to bytecode identical to the methods we manually wrote in Listing 1 and Listing 2.

```
MondrianCreole class>>cssParser
   <parse>
   ^ CSSTranslator
```

One small problem at this point is that the syntax highlighter in the code editor is broken. As before, we create a new subclass of `CSSParser` named `CSSHighlighter` that underlines the selectors and dispatches to standard Smalltalk highlighting for the definitions. Again this is achieved by overriding the appropriate methods of `CSSParser`. For example, the selector method is defined in `CSSHighlighter` like this:

```
CSSHighlighter>>selector
   ^ super selector ==> [ :ast | ast -> TextEmphasis underlined ]
```

Using a `<highlight>` annotation we declare the handler to be responsible for syntax highlighting of the CSS code:

```
MondrianCreole class>>cssHighlighter
    <highlight>
    ^ CSSHighlighter
```

Adding a pidgin or creole over an existing framework can simplify its use and reduce a considerable amount of syntactic noise. Without touching the original framework we are able to provide different language skins that might be an appealing alternative to the internal DSL that was used before.

The changes shown in this section are all that is needed to also affect the debugger. Figure 4 shows a live result of stepping through the execution of the script building the UML package shape.

4.3 Specifying the Transactional Memory Argot

In a nutshell, our software transactional memory implementation works as follows. We compile every method in the system twice, once for the transactional and once for the non-transactional context. On the transactional code we apply two transformations: (1) all state access is reified to be dispatched through the transactional context, and (2) method names and method sends are prefixed with `__atomic__`. Furthermore we use method annotations to disable or customize these transformations in certain places, such as when primitive code is called or in the transactional infrastructure itself. A transaction is started by assigning a transaction manager to a thread-local variable and by calling an `__atomic__` method. At the end of a transaction the cached changes are atomically checked for conflicts and applied to the involved objects. The transaction boundaries are handled at the language level using the reflective facilities of the host language. For details on the implementation of the semantic model please refer to our previous work [31].

Through the use of Helvetia the argot specification becomes simple. The complete set of transformation rules are presented below:

```
1  Object class>>transformAtomic
2     <attribute>
3     ^ ConditionRule new
4         if: [ :context | context isTransactional ]
5         then: (TreeRule new
6           expression: '`@receiver `@msg: `@args' do: [ :ast |
7              ast swapWith: ``(`,(ast at: '`@receiver')
8                 `,('__atomic__' , (ast at: '`@msg:'))
9                 `,(ast at: '`@args')) ];
10          expression: '`var := `@expr' do: [ :ast |
11             ast swapWith: ``(self
12                atomicInstVarAt: `,(ast binding index)
13                put: `,(ast at: '`@expr')) ];
14          expression: '`var' do: [ :ast |
15             ast swapWith: ``(self
16                atomicInstVarAt: `,(ast binding index)) ])
```

The code uses the `<attribute>` method annotation (line 2), to tell the compiler that the rules are expected to run after the symbols have been resolved (attributed). Line 4 makes sure that the transformation is only performed when compiling code for the transactional context. Lines 5–16 implement the actual transformations, exemplified in the table below:

6–9 Transform Message Sends
`self printString` $\rightarrow$ `self __atomic__printString`

10–13 Transform Instance Variable Write
`value := 'Atomic'` $\rightarrow$ `self atomicInstVarAt: 2 put: 'Atomic'`

14–16 Transform Instance Variable Read
`value` $\rightarrow$ `self atomicInstVarAt: 2`

All message sends are prepended with `__atomic__` to ensure that the execution stays in the atomic context. All state accesses, such as instance variable reads and writes, are transformed to message sends and dispatched through the transaction manager. This allows us to delay modifications to objects, so that the changes are only visible within the current transaction. The number `2` in the examples above refer to the index of the named instance variable `value`. This slot index is retrieved from the attributed AST.

To trigger the compilation of a transactional and a non-transaction version of every method we hook into the parser using the `<parser>` annotation. We copy the compilation context and spawn a new compilation path for the transaction context. (Details follow in Section 5.)

```
Object class>>compileTransactional: aContext
   <parser>
   aContext isTransactional ifFalse: [
      aContext copy
         beTransactional;
         perform ].
   ^ nil
```

The implementation of transactional memory using Helvetia has numerous advantages over our previous implementation: The code base is not integrated into the compiler in an ad hoc manner anymore, but through clearly defined Helvetia extension points. In our previous implementation, we directly patched in several places the existing compiler. Following the support of Helvetia, the transformations are specified at a single place that is external to the compiler. As a consequence the code is nicely modularized and more concise. Without additional work the use of the debugger becomes viable, because the transformations preserve location integrity. In our previous example it was not possible to transparently step through transactional code, as the tools would display the generated code. The new implementation takes advantage of Helvetia and single stepping through transactional code looks exactly the same as the regular code.

5 Leveraging the Host Toolchain with Helvetia

Helvetia manages to integrate multiple embedded languages with existing tools by leveraging and intercepting the existing toolchain and the underlying representation of the host language. Helvetia provides hooks to intercept parsing, AST transformation and semantic analysis of the standard compilation toolchain.

5.1 Homogeneous Language Integration

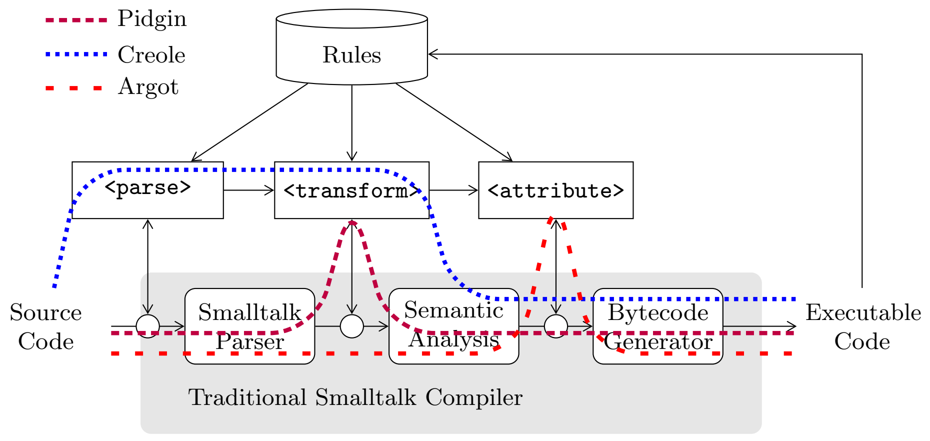


Fig. 3. The code compilation pipeline showing multiple interception paths.

In Smalltalk the unit of compilation is the method. Whenever a method is saved, it automatically triggers the Smalltalk compiler within the same environment. The source code and compiled methods as well as the compiler are all available at runtime. As seen in Figure 3 we have enhanced the standard compilation pipeline to be able to intercept the data passed from one compilation step to the other. We are able to perform source-to-source transformations or to bypass the regular Smalltalk parser altogether. Furthermore we are able to perform AST transformations either before, instead of, or after semantic analysis.

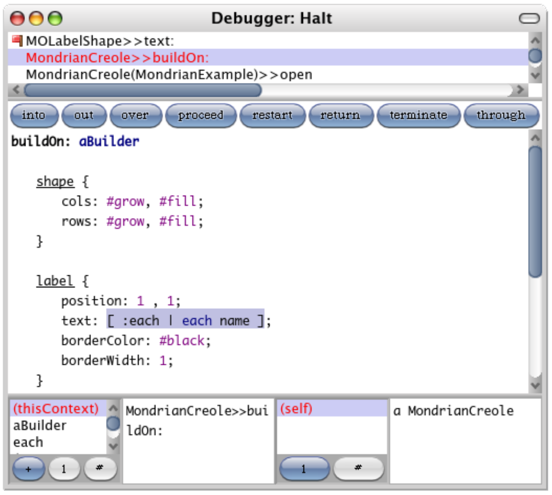
The rules to intercept this transformation pipeline are defined using annotated methods. These methods constitute conventional Smalltalk code that is called at compile time [38]. The interception rules allow us not only to modify data in the pipeline but also to bypass conventional components.

- A rule marked with `<parser>` allows one to intercept the parsing of the source code. The result of a parser rule can be either a new source string (in case of a source-to-source transformation) or a Smalltalk AST (in which the original Smalltalk parser is skipped).
- A rule marked with `<transform>` is performed on the AST after parsing and before semantic analysis. It allows developers to apply arbitrary transformations on the AST. Furthermore, it is possible to change the default semantic analysis and instead perform a custom one.
- A rule marked with `<attribute>` is performed after symbol resolution and before bytecode generation. This makes it possible to perform transformations on the attributed AST as well.

Compilation errors are handled by the standard toolchain. Since all data passed from one step to the next carries information on its original source location, the error location is determined automatically and it is revealed to the the user through the traditional means of the compiler. For example, when a variable is undeclared, its occurrence is highlighted and the user is asked to correct the problem.

5.2 Homogeneous Tool Integration

To control syntax highlighting, we use the rule database to change the default highlighting. The traditional Smalltalk syntax highlighter is only applied to normal Smalltalk methods. As soon as there is a custom parser involved, the affected part of the source code remains black unless a custom highlighter is provided. The annotation `<highlight>` is used to define a highlighting rule. Figure 4 shows the result of the custom highlighter in the source pane of the debugger. Helvetia provides similar extension points for code-completion and contextual menus.



```
MOLabelShape>>text:
MondrianCreole>>buildOn:
MondrianCreole(MondrianExample)>>open
```

```
buildOn: aBuilder

   shape {
      cols: #grow, #fill;
      rows: #grow, #fill;
   }

   label {
      position: 1 , 1;
      text: [ :each | each name ];
      borderColor: #black;
      borderWidth: 1;
   }
```

```
(thisContext)
aBuilder
each
```

```
MondrianCreole>>buildOn:
```

```
(self)
```

```
a MondrianCreole
```

Fig. 4. Traditional Smalltalk debugger with language specific syntax highlighting stepping through a mixture of Smalltalk and the creole defined in Section 3.2.

The lack of dedicated tools to find and fix bugs in a new language is one of the major drawbacks when designing and using embedded languages. Since our approach uses the code abstraction of the host language, the standard debugging tools continue to work. One can set breakpoints as in a conventional methods. Stepping through code written in a mixture of languages poses no problem either. The AST of a debugged method carries information about the source range in the original code. Generated code either reuses the source ranges of the parent node, or has no source range and is therefore invisible in the debugger. With this information from the original AST the debugger is able to accurately highlight the current execution point and step to the next statement, without having to know anything about the structure of the source string.

Helvetia currently does not change the way the debugger presents information, *e.g.,* the stack frames and variables are displayed at the level of host language. However, we envision the addition of new rules to enable the customization of the debugger's user-interface.

Figure 4 shows how we step through the code of our creole example. The top part shows the execution stack, with the top method being `LabelShape>>text:` which sets the Mondrian text of the shape. The main editor shows the creole code corresponding to the creation of the shape. This example shows how the debugger accommodates both the creole code and the called framework code.

5.3 Multiple Context-Dependent Languages

Helvetia uses *annotated methods* class-side (static) methods to define a rule database that is queried by the compiler and other tools. The rules affect instance-code of the corresponding class and its subclasses. To define a system-wide rule, it has to be installed within an extension method for `Object`, the root of the class hierarchy. The following primitive rule types are currently supported:

– The `ConditionRule` behaves like a case statement. An ordered list of conditions is checked and the first matching action is executed. If no match is found a default action is executed as an alternative. Both the condition and the match action are implemented using the host language and can check for arbitrary conditions using the reflective API. This rule type is typically used to scope the effect of rules to specific parts of the system.
– `MatchRule` and `RangeRule` use regular expressions to match source code. This is useful to check for specific strings in the code when no parse tree is available yet. For example, regular expressions are sometimes used to provide custom syntax highlighting within string literals of the host language. In many cases matching the parse tree is simpler. This rule type is only supported for `<parse>` rules.
– The `TreeRule` is a parse tree matcher. Unlike string matching these patterns work on the AST and make it possible to efficiently find all occurrences of particular node combinations. Again, action code can be supplied that is executed when a match is found. This rule type is only supported for `<transform>` and `<attribute>` rules.

As we have seen in transactional example in Section 4.3, rules can be arbitrarily nested. Instead of attaching Smalltalk code to an action, another rule can be used that is subsequently applied in the context of the parent match. Furthermore, as we have seen in the creole example in Section 4.2, it is possible to supply a custom rule object such as a custom parser or syntax highlighter.

6 Evaluation

In the related work, support for pidgin, creole and argot embedded languages is variable. In the category of extensible compilers usually all types are supported. Meta programming systems either do not provide a model of the host language that can be modified (Converge) or do not provide the possibility to change the syntax (MetaOCaml, Scheme). Language workbenches are designed to implement creoles, that is to build new language elements and combine them with other languages.

6.1 Host Language Choice

Meta programming systems and language workbenches provide large toolsets to define new languages, however in many cases (Converge, MPS, Intentional Software, Katahdin, XMF) they use derivatives. In some cases (Katahdin, XMF) they implement a new runtime layer that makes it difficult to reuse existing code and libraries.

We believe that it is beneficial for the adaptation of a language authoring system to build into an existing host language and leverage as many features as possible. Helvetia reuses the Smalltalk code representation, the complete compiler toolchain and the existing IDE to provide a lightweight language integration. Helvetia code shows performance penalty as it uses the same runtime infrastructure as the host language.

In our previous work [39] we have evaluated several host language choices for a system like Helvetia. Smalltalk has proven to be a good practical choice, though not a requirement:

– In Smalltalk the compiler is part of the development environment and can be changed on the fly. For Helvetia, we did so by carefully introducing interception points before and after the different compilation steps (Section 5.1). Rules are defined using annotated methods that are evaluated at compilation time.
– Rules that that work on AST nodes need to preserve the source mapping with every transformation. In our case we use the refactoring engine of the host language to query the AST nodes. Meta-programming facilities, such as the quasiquoting facilities [40] in Scheme or OMetaCaml's staging constructs [41], greatly simplify the generation of code.
– As with the compiler, editors are required to support extension points for custom highlighting, code completion, error reporting, *etc.* In Smalltalk the editors are implemented within the host language and can be customized by extending or changing the existing code. In our case we did so by consulting the rule database for every method being edited. It is essential that the environment have full access to the rules.
– To support debugging of different languages, the debugger must be able to use an arbitrary source mapping between the custom language and the executable representation of the host environment. In our case we maintain

this mapping from the source string through all transformation stages down to the bytecode. The debugger is fed with a custom function that maps source ranges to bytecode ranges. Since the debugger reuses the normal code editor of the programming environment, syntax highlighting works without additional support.
– Since all languages use the same underlying representation, there are no difficulties to share application state between different parts of the system. For example, a new language construct can access temporary variables, instance variables or globals. When a method is evaluated, it does not matter in what language it has been implemented. Block closures can be passed around, no matter what origin they have and from what language context they are evaluated.

We see the following main challenges to implement a system like Helvetia in an existing environment like Eclipse: (1) replace the default editor, compiler and debugger with a customized ones, (2) connect these to a central rule database (this requires communication between different Java VMs), and (3) establish a fine-grained mapping between byte code and source code (by default Java only supports a line based mapping).

6.2 Multiple Embedded Languages

The integration of new embedded languages into each other and into the host language is solved in different ways. Most existing systems do this on a per-file basis. Some systems require special tokens to switch between languages (Converge, XMF, MontiCore). In Katahdin this token is freely definable by changing the host language.

In Helvetia the scope of each language is defined in a way that differs from the systems discussed. Language extensions are defined in rules that use reflection capabilities of the host language to check for specific conditions. Namespaces, packages, class-hierarchies, or annotated classes can define a scope. At a method level we are able to look for specific annotations in the source string or simply try different parsers. At a sub-method level we are able to look for certain code statements to transform, either using regular-expressions (before parsing) or using parse-tree matching (after parsing). These techniques enable a fine-grained control over the languages, however for end users it is often less evident what parts of the system belong to the host language or are externally defined. This can be addressed by means of tailored highlighting of such code.

6.3 Combining Languages

Since multiple rule-sets can be active at the same time, different language extensions can be combined. For example, both the transactional memory extension and any of the Mondrian languages can be active at the same time, since they do not perform transformations at the same place in the compiler toolchain. If conflicting rules are active, for example two language extensions that define their own parser, Helvetia raises an error.

Transformation rules typically don't conflict, since they work on the same AST model. Rules are performed in a deterministic order based on their priority. Thanks to the reflective capabilities of the system, each rule can detect other active rules and choose to disable itself or other rules on the fly. In practice conflicts are rare, because language extensions are typically scoped to a small portion of the system, such as a class hierarchy or a package.

6.4 Homogeneous Languages

The language transformation systems use a preprocessor. This considerably slows down the compile cycles, as several transformation passes and compilation cycles of different independent tools are involved. Furthermore, it can be difficult to debug the generated code, as it is often impossible to provide a correct mapping from generated code back to the original source. Different host and meta languages make interoperability more difficult.

Helvetia maintains this mapping throughout a single compiler pipeline that allows one to use this information in the standard Smalltalk debugger. Transformation rules are defined in the host language and take advantage of the reflective capabilities of the system.

6.5 Homogeneous Tools

Most systems provide debugging tools for language developers, however they mostly lack sophisticated debugging support for application developers. We believe that it is crucial for the end users of a language to have good debugging support. Implementing custom debuggers is expensive and thus seldom done in practice. Furthermore switching between different debuggers in a multilingual environment is cumbersome. End users do not want to be forced to learn new tools, but instead prefer the familiar tools provided by the host language in use.

Helvetia supports the use of the existing debugging facilities for language developers and end users. While the host language debugger might not offer the optimal abstraction for all languages, it offers a free live view on the untransformed source code and the current execution point. This is something that most other systems to not provide without additional development effort.

7 Conclusion

In this paper we have presented Helvetia, an environment for defining embedded languages and for integrating them into the host language. We have shown how we can bend the syntax and semantics of the host language, by introducing a few extension points into the standard compiler pipeline. Our contributions are the following:

1. We have identified and demonstrated three fundamental types of embedded languages: *pidgins* adopt the syntax of the host language while extending its semantics; *creoles* further refine pidgins with their own dedicated syntax; *argots* switch the semantics of the host language without changing the syntax and without requiring changes in existing application code.
2. We have presented a novel approach to language embedding that leverages the host language toolchain. Reusing the traditional code representation of the host system has numerous advantages. We have achieved a tight integration of different languages that work seamlessly with each other. We specify transformation rules using annotated methods, and specify the scope of these transformations using reflective facilities of the host language. Our approach works nicely with existing code, and integrates well into the existing toolset. Fine-grained customizations such as syntax highlighting are readily supported.
3. We have demonstrated a fully working prototype of the HELVETIA system and shown the implementation of three non-trivial embedded DSLs in detail. We have also mentioned two other language extensions in the domain of language engineering that have been implemented using HELVETIA, the grammar specification language and the quasiquoting facility. Furthermore the HELVETIA infrastructure has been used for the implementation of *Language Boxes* [6], an adaptive language model for fine-grained language changes, language composition and language re-use in terms of grammar transformation. A collection of HELVETIA example languages can be found at `http://scg.unibe.ch/research/helvetia/examples`.
4. We have identified the basic requirements for the host language and have compared HELVETIA with other systems for embedding languages.

As future work we plan to validate our approach on a wide variety of other language extensions that we have collected from an industrial context. We also intend to look into ways to automatically refactor code from the host language towards a DSL, as well as how to automate the creation of DSL layers to improve the use of existing frameworks.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 – Sept. 2010). We thank CHOOSE, the special interest group for Object-Oriented Systems and Environments of the Swiss Informatics Society, for its financial contribution to the presentation of this paper. We also like to thank Damien Cassou, Stéphane Ducasse, Ralf Lämmel, Jorge Ressia, David Röthlisberger, and Erwann Wernli for their feedback on earlier drafts of this paper.

References

1. Fowler, M.: FluentInterface, on Martin Fowler's blog (December 2005) `http://www.martinfowler.com/bliki/FluentInterface.html`.

2. Fowler, M.: Language workbenches: The killer-app for domain-specific languages (June 2005) `http://www.martinfowler.com/articles/languageWorkbench.html`.
3. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4) (2005) 316–344
4. Hudak, P.: Building domain specific embedded languages. ACM Computing Surveys **28**(4es) (December 1996)
5. Spinellis, D.: Notable design patterns for domain specific languages. Journal of Systems and Software **56**(1) (February 2001) 91–99
6. Renggli, L., Denker, M., Nierstrasz, O.: Language boxes: Bending the host language with modular language changes. In: Software Language Engineering: Second International Conference, SLE 2009, Denver, Colorado, October 5-6, 2009. LNCS, Springer (2009) to appear.
7. Sheard, T.: Accomplishments and research challenges in meta-programming. In: SAIG 2001: Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation, London, UK, Springer-Verlag (2001) 2–44
8. Gybels, K., Wuyts, R., Ducasse, S., D'Hondt, M.: Inter-language reflection — a conceptual model and its implementation. Journal of Computer Languages, Systems and Structures **32**(2-3) (July 2006) 109–124
9. Van Wyk, E., Krishnan, L., Bodin, D., Schwerdfeger, A.: Attribute Grammar-Based Language Extensions for Java. LNCS **4609** (2007) 575
10. Kats, L.C.L., Bravenboer, M., Visser, E.: Mixing source and bytecode. A case for compilation by normalization. In Kiczales, G., ed.: Proceedings OOPSLA 2008, Nashville, Tenessee, USA, ACM (October 2008) 91–108
11. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In Gabriel, R.P., Bacon, D.F., Lopes, C.V., Jr., G.L.S., eds.: Proceedings OOPSLA 2007, New York, NY, USA, ACM Press (2007) 1–18
12. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An extensible compiler framework for java. In: Compiler Construction. Volume 2622 of LNCS. Springer-Verlag (2003) 138–152
13. Cox, R., Bergan, T., Clements, A.T., Kaashoek, F., Kohler, E.: Xoc, an extension-oriented compiler for systems programming. SIGARCH Comput. Archit. News **36**(1) (2008) 244–254
14. Piumarta, I., Warth, A.: Open reusable object models. Technical report, Viewpoints Research Institute (2006) VPRI Research Note RN-2006-003-a.
15. Warth, A., Piumarta, I.: OMeta: an object-oriented language for pattern matching. In: DLS '07: Proceedings of the 2007 symposium on Dynamic languages, New York, NY, USA, ACM (2007) 11–19
16. Tratt, L.: The Converge programming language. Technical Report TR-05-01, Department of Computer Science, King's College London (February 2005)
17. Calcagno, C., Taha, W., Huang, L., Leroy, X.: Implementing multi-stage languages using ASTs, GenSym, and Reflection. In: Proceedings GPCE. Volume 2830 of LNCS., Springer-Verlag (2003) 57–76
18. Dimitriev, S.: Language oriented programming: The next programming paradigm. onBoard Online Magazine **1**(1) (November 2004)
19. Simonyi, C., Christerson, M., Clifford, S.: Intentional software. In: Proceedings OOPSLA 2006, ACM (2006) 451–464
20. Solmi, R.: Whole Platform. PhD thesis, University of Bologna (March 2005)
21. Charles, P., Fuhrer, R.M., Jr., S.M.S., Duesterwald, E., Vinju, J.J.: Accelerating the creation of customized, language-specific ides in eclipse. In Arora, S., Leavens, G.T., eds.: Proceedings OOPSLA 2009, ACM (2009) 191–206

22. Seaton, C.: A programming language where the syntax and semantics are mutable at runtime. Technical Report CSTR-07-005, University of Bristol (June 2007)
23. Clark, T., Sammut, P., Willans, J.: Superlanguages, Developing Languages and Applications with XMF. Volume First Edition. Ceteva (2008)
24. Faith, R.E., Nyland, L.S., Prins, J.F.: KHEPERA: a system for rapid implementation of domain specific languages. In: DSL'97: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997, Berkeley, CA, USA, USENIX Association (1997) 19–19
25. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: Modular development of textual domain specific languages. In Paige, R., Meyer, B., eds.: Proceedings of the 46th International Conference Objects, Models, Components, Patterns (TOOLS-Europe), Springer-Verlag (2008) 297–315
26. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Programmers (May 2007)
27. Bravenboer, M., Visser, E.: Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Schmidt, D.C., ed.: Proceedings OOPSLA 2004, Vancouver, Canada, ACM Press (oct 2004) 365–383
28. Visser, E.: Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer, C., et al., eds.: Domain-Specific Program Generation. Volume 3016 of LNCS. Spinger-Verlag (June 2004) 216–238
29. Goldberg, A., Robson, D.: Smalltalk 80: the Language and its Implementation. Addison Wesley, Reading, Mass. (May 1983)
30. Meyer, M., Gîrba, T., Lungu, M.: Mondrian: An agile visualization framework. In: ACM Symposium on Software Visualization (SoftVis'06), New York, NY, USA, ACM Press (2006) 135–144
31. Renggli, L., Nierstrasz, O.: Transactional memory in a dynamic language. Journal of Computer Languages, Systems and Structures **35**(1) (April 2009) 21–30
32. Herlihy, M.P.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems **13**(1) (January 1991) 124–149
33. Herlihy, M.P., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20. Annual International Symposium on Computer Architecture. (1993) 289–300
34. Roberts, D., Brant, J., Johnson, R.E.: A refactoring tool for Smalltalk. Theory and Practice of Object Systems (TAPOS) **3**(4) (1997) 253–263
35. Futamura, Y.: Partial evaluation of computation process: An approach to a compiler-compiler. Higher Order Symbol. Comput. **12**(4) (1999) 381–391
36. Wirth, N.: What can we do about the unnecessary diversity of notation for syntactic definitions? Commun. ACM **20**(11) (1977) 822–823
37. Bracha, G.: Executable grammars in Newspeak. Electron. Notes Theor. Comput. Sci. **193** (2007) 3–18
38. Tratt, L.: Domain specific language implementation via compile-time meta-programming. ACM TOPLAS **30**(6) (2008) 1–40
39. Renggli, L., Gîrba, T.: Why Smalltalk wins the host languages shootout. In: Proceedings of International Workshop on Smalltalk Technologies (IWST 2009), ACM Digital Library (2009) To appear.
40. Bawden, A.: Quasiquotation in Lisp. In: Partial Evaluation and Semantic-Based Program Manipulation. (1999) 4–12
41. Taha, W.: A gentle introduction to multi-stage programming. In: Domain-Specific Program Generation. (2003) 30–50