Towards Live Language Development

Gabriël Konat Sebastian Erdweg Eelco Visser
Delft University of Technology
g.d.p.konat@tudelft.nl, s.t.erdweg@tudelft.nl, visser@tudelft.nl

Abstract

We would like to see live programming applied to language development, to get *live language development*. With live language development, a language developer gets fast feedback when they change their language, enabling experimentation with language design and development.

In this paper, we describe what live language development is and why it is useful, and we analyze what is needed to achieve live language development. Moreover, we describe our work in progress in supporting live language development in the Spoofax language workbench.

1. Introduction

A language workbench is a *language development environment* for developing domain-specific (DSL) and general-purpose programming languages. Language developers define their language as a language specification in the language workbench. From such a language specification, the language workbench derives a concrete language implementation.

We would like to see live programming applied to language development in language workbenches, to get *live language development*. With live language development, a language developer gets fast feedback when they change their language, enabling experimentation with language design and development. For example, if a developer changes a grammar production, we expect updated feedback for the grammar specification, but also an updated parser which produces new feedback in the form of syntax errors for all programs of that language.

In this paper, we present a mission statement for live language development. We describe what live language development is and why it is useful, and we analyze what is needed to achieve live language development. Moreover, we describe our work in progress in supporting live language development in the Spoofax [8] language workbench.

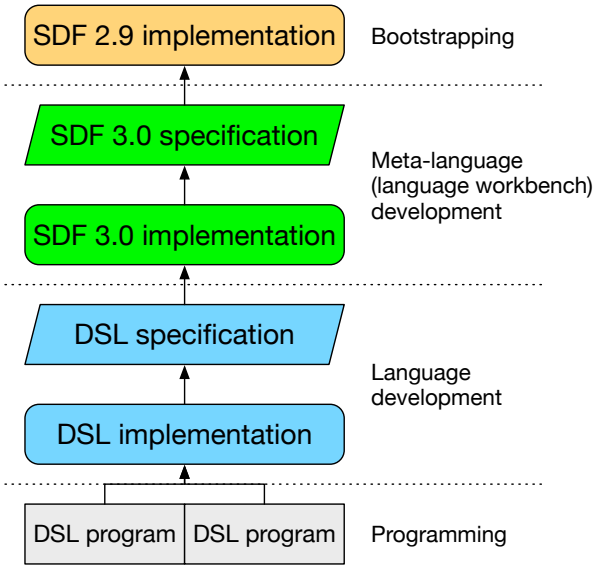

Figure 1. Overview of the different levels of specification. An arrow means 'is specified in'. A DSL program is specified in the DSL language implementation, the DSL specification is specified in the SDF meta-language, and so forth.

Note that this paper is about the liveness of a language development environment itself, not about support for designing or developing live programming languages.

2. Mission Statement

2.1 What do we want and why?

In short, we would like to get the entire stack from Figure 1 to be live. We will explain each part of the figure in this section.

Many IDEs such as Eclipse, IntelliJ, and Netbeans, and editors such as emacs, vim, and atom, provide fast feedback when a program changes. Feedback comes in the form of updated syntax highlighting, errors and warnings, outline, folding, and so forth. Programmers (or software developers) rely on fast feedback to quickly find and fix problems in their programs. But what about changes to the language a program is written in?

Language developers would also like to get fast feedback when a language is changed. A language is implemented or specified in a *language specification*. From the specification, a *language implementation* is derived. When a language specification is changed, we would like to get fast feedback for the change in the language specification itself, but also for all programs of that language. In other words, changes to a language specification should be cascaded into changes to the programs of that language. For example, if we change a typing rule in a language specification, we expect to see updated syntax highlighting and outlines for the specification, but also see type errors to appear or disappear in all programs.

With fast feedback for language changes, language developers can quickly explore the consequences of the changes made and react accordingly, enabling experimentation with language design and development.

Meta-languages are languages that are specifically made to specify or implement a (part of a) language. Language workbenches typically employ meta-languages for the specification of languages through language specifications. For example, the Spoofax Language Workbench provides the SDF [14] meta-language for specifying the grammar of a language, the TS language for specifying the type system of a language, and many more meta-languages for specifying different aspects of a language. A language specification in Spoofax consists of one or more programs of these meta-languages, from which a language implementation can be derived.

Language workbench developers (who develop meta-languages) would also like to get fast feedback when a meta-language is changed. When a meta-language is changed, we expect fast feedback for the change to the meta-language, all language specifications that use that meta-language, and all programs of those languages. This requires cascading of changes one level deeper.

A meta-language is frequently bootstrapped or self-hosting, meaning that it is specified in itself. Bootstrapping eases the development of the meta-language, and is a significant test case for the meta-language. Fast feedback for bootstrapped meta-languages should also be supported.

Programs, language specifications, and meta-languages can break, especially by experimentation with fast feedback. Breaking a program or language specification is not a big problem, since the change can easily be reverted by an undo operation or by fixing the problem. However, breaking a bootstrapped meta-language poses more problems, because it will not be able to bootstrap itself any more. A more sophisticated mechanism for reverting changes or fixing a broken bootstrapped meta-language is required.

To summarize: We would like to see fast feedback for language and (bootstrapped) meta-language development.

2.2 What do we need to achieve that?

To support fast feedback for changes to programs, languages, and (bootstrapped) meta-languages, we require *live language services*, and a *language registry* to support *live language development* and *live bootstrapping*.

A common requirement for fast feedback is that changes should be visible in the same environment. A *development environment* is an environment that hosts language implementations and their programs, such as the IDEs and editors mentioned earlier. For example, when a program is edited in Eclipse, feedback appears in the same environment. Starting or restarting the environment to get feedback requires manual steps and introduces a lot of latency.

A development environment needs to provide live language services. Language services include a parser, analyzers, transformations, editor services, etc. Live language services provide fast feedback when a program changes. A pipeline executes the relevant services when a change to the program is made, and presents feedback to the user. This is what IDEs and editors already provide an infrastructure for.

A *language development environment* is an environment that hosts meta-languages, language specifications, their derived language implementations, and programs of languages. An example of such an environment is the Spoofax language workbench.

To support live language development, language services should respond to changes in the language specification, within the same environment. A language registry keeps track of (multiple) language specifications and their derived implementations, and provides fast feedback when languages are added, removed, or changed. Whenever a language specification is changed, the language registry reexecutes the language service pipeline to provide new feedback for programs.

In essence, changes to a language specification are cascaded into language service changes. This cascading can occur multiple times in the case of changes to meta-language specifications, which may trigger changes to other languages, which in turn triggers the reexecution of language service pipelines, and so on.

Live bootstrapping is live language development applied to bootstrapped meta-languages. To support live bootstrapping, language specifications and implementations must be versioned, and dependencies from language specifications to language implementations must be supported.

Versioning and dependencies are required for two reasons. First, we need versioning and dependencies because a meta-language needs to depend on a previous version of itself to be able to bootstrap itself. For example, to bootstrap SDF 3.0 (i.e. compile the SDF 3.0 specification), we require a dependency on the (previously bootstrapped) SDF 2.9 language implementation. Second, we need versioning and dependencies because it needs to be possible to bootstrap a meta-language after an erroneous release by reverting

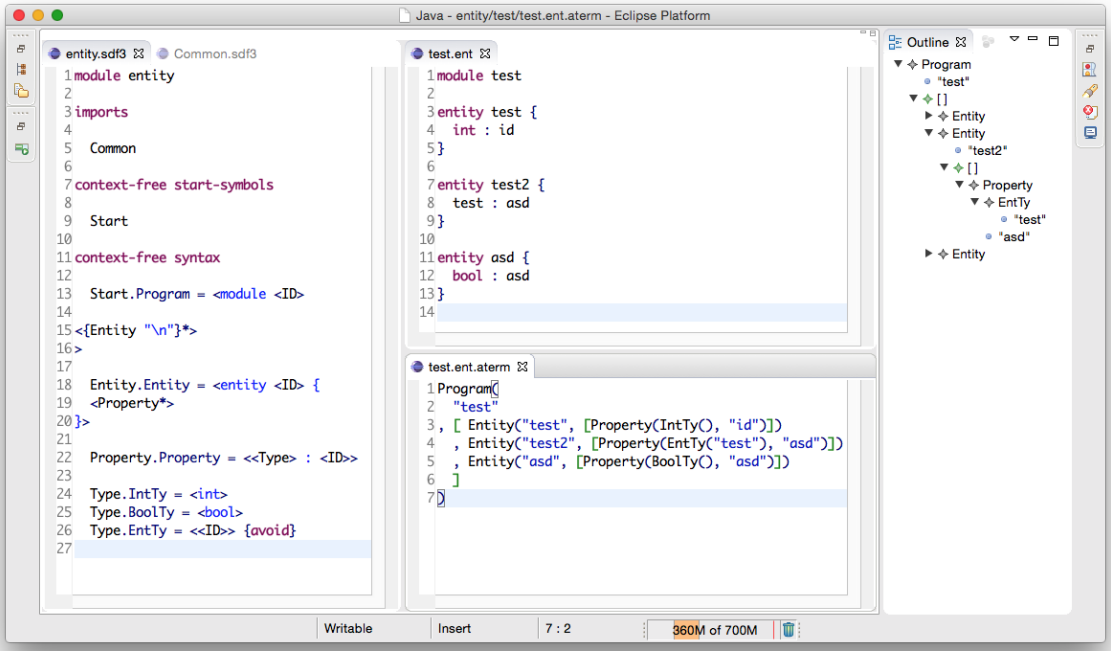

Figure 2. Language development in Spoofax. Left: the Entity language's syntax specification (specified in the SDF3 meta-language). Top: an example Entity program. Bottom: abstract syntax tree (AST) of the program. Right: outline of the program.

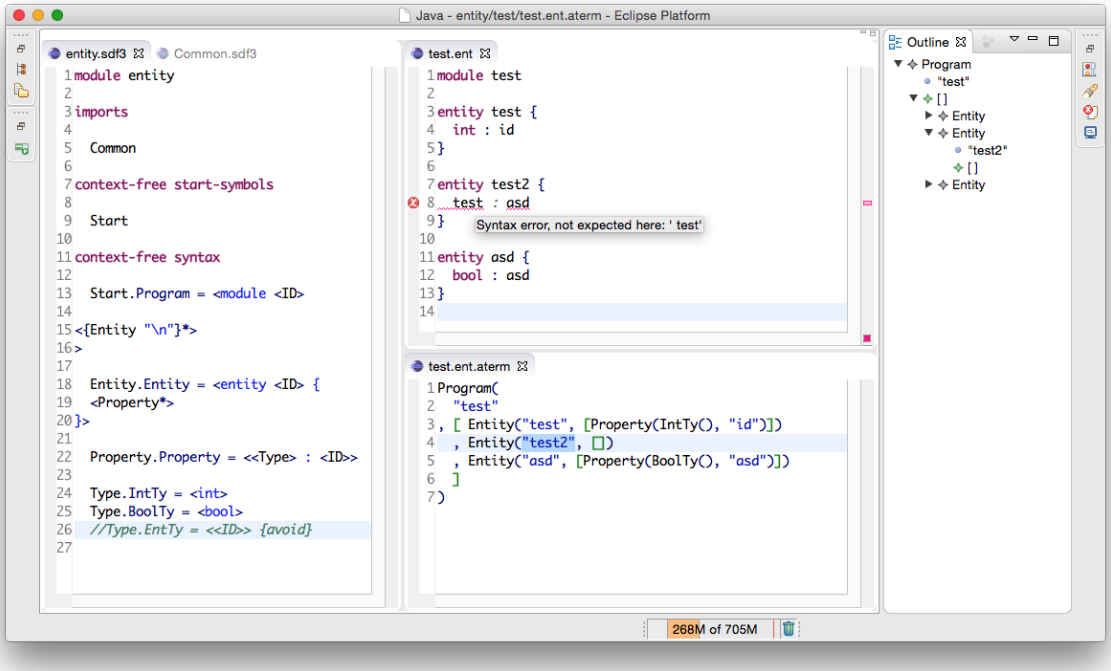

Figure 3. Live language development in Spoofax. A production in the syntax specification is commented out, which updates the rest of the environment. The example program now indicates an error, the AST has changed (even though parsing fails, we still approximate the AST through error recovery), and the outline has changed.

to a previous version. For example, if we bootstrap an erroneous SDF 3.0, SDF 3.1 must be based on SDF 2.9 instead of 3.0.

The language registry should support keeping track of multiple versions of a language, being able to switch between versions, and should execute the language service pipeline when versions change.

To summarize: For live language development and live bootstrapping to work, a language registry is required that keeps track of languages, supports versioning and dependencies between languages, and processes changes to languages, which triggers the execution of live editor services.

3. Live Language Development in Spoofax

Spoofax [8] is a textual language workbench in which a language is specified in several domain-specific meta-languages, such as SDF [14], Stratego [2], NaBL [12], and TS. From a language specification, Spoofax derives a language implementation with a full-fledged development environment. With Spoofax, language engineers can specify their language, and derive (instead of manually implement) the hairy implementation details of a language development environment for their language. Live language development in Spoofax is a work in progress, but with the latest version of Spoofax we have taken significant steps towards live language development and live bootstrapping.

Spoofax consists of a platform-independent core (a Java API), and several adapters of that core to specific platforms such as the Eclipse and IntelliJ IDE platforms, the Maven and Gradle build system platforms, and a command-line interface.

Spoofax is a development environment that supports live language services. We interface with the language services of the Eclipse and IntelliJ IDE platforms to provide a live development environment. When a source file is edited in an Eclipse or IntelliJ editor, Spoofax runs the language service pipeline, which updates the styling, error messages, outlines, etc. and displays that in editors. Other editors for affected source files of a change (e.g. deleting a class that is used in a different source file) are also updated through incremental name and type analysis [16].

Spoofax is also a language development environment that supports live language development and live bootstrapping. At the core of Spoofax is a language registry that keeps track of language specifications, implementations, and their programs. Language specifications and programs of those languages can be edited side-by-side, in the same environment, as seen in Figure 2.

Whenever a language specification is changed, the derived language implementation is compiled and reloaded into the same environment, which triggers the language registry to execute the language service pipeline for all programs of that language. Figure 3 shows an example of a change to the syntax specification (commenting out a production) of the Entity language. Spoofax updates the rest of the environment after a language specification change. This works for changes to meta-languages as well, cascading the changes to language specifications and programs.

Spoofax's language registry also supports live bootstrapping. The language registry versions language specifications and implementations, and supports dependencies between them. When a meta-language breaks, we can revert back to a consistent state by selecting a working version.

The liveness in Spoofax is different from most live programming systems. Instead of updating a running language implementation when changes are made, we generate a language implementation, compile it, and reload it into the environment. Using incremental compilation techniques [5], we try to make generation and compilation only take time relative to the size of the change.

Changes to the syntax are fast, since we can quickly generate, compile, and reload the parser implementation. However, changes to static analysis and transformations are not fast, since their compiler is not incremental. This means that feedback for syntax changes can be considered fast, whereas feedback for analysis and transformation changes are still slow. Since our meta-languages perform a lot of analysis and transformations, feedback from changes to meta-languages are also slow.

In the future, we would like to fully implement the mission statement from this paper. Missing right now is fast feedback for changes to the static analysis and transformations of a language, and fast feedback for changes to (bootstrapped) meta-languages. That requires an incremental compiler for the Stratego meta-language, which we use for analysis and transformation.

4. Related Work

Most language workbenches support some form of live language development, i.e. (re)loading a language into the same environment without restarting.

Rascal [10, 11] is an extensible metaprogramming language and IDE for source code analysis and transformation. Rascal's meta-language is interpreted, allowing changes to a language specification to be directly interpreted.

MPS [15] is an open-source projectional language workbench. A projectional language workbench providing a projectional editor that can display and edit an underlying tree structure. MPS applies changes to a language specification in the same environment and updates programs where needed.

MetaEdit+ [9] is a platform-independent graphical language workbench for domain-specific modeling. A graphical language workbench provides a graphical editor for displaying and editing models. MetaEdit+ processes changes to a language specification and applies them in the same environment, updating all models to conform to the new specification, or warning the developer of a change can break existing models.

SugarJ [4, 6] is a Java-based extensible programming language that allows programmers to extend the base language with custom language features. SugarJ uses self-extension to develop language implementations as extensions, which are loaded into the same environment by importing the extensions.

Racket [13] is an extensible programming language in the Lisp/Scheme family, which can serve as a platform for language creation, design, and implementation. Racket also uses self-extension, and supports live language development through the DrRacket IDE.

A notable exception to live language development is Xtext [1, 3, 7]; an open-source framework for development of programming languages and DSLs. Languages in Xtext are Eclipse or IntelliJ plugins, which cannot be dynamically loaded. A new Eclipse or IntelliJ instance has to be started to get feedback for changes.

The MPS, DrRacket, and MetaEdit+ language workbenches support live bootstrapping in addition to live language development. However, when a bootstrapped meta-language breaks, MPS and DrRacket cannot revert the change and fix the environment. The environment has to be restarted to revert to a working baseline.

MetaEdit+'s languages are changed through modeling transactions. Inside a transaction, an undo operation can go back one modeling step, and the entire transaction can also be abandoned. Undoing and abandoning can thus revert a change that breaks a meta-language implementation, and fix the environment.

5. Conclusion

We would like to see live programming applied to (meta-) language development, to support fast feedback for changes to programs, languages, and (bootstrapped) meta-languages. With fast feedback for language changes, language developers can quickly explore the consequences of the changes made and react accordingly, enabling experimentation with language design and development.

To support fast feedback for changes to programs, languages, and (bootstrapped) meta-languages, we require live language services, and a language registry to support live language development and live bootstrapping. Live language services provide fast feedback when a program changes, whereas the language registry provides fast feedback when a language changes.

Live bootstrapping requires that the language registry versions languages and supports dependencies between languages, such that a meta-language can be bootstrapped with its previous version, and reverting to a previous version is possible in case the meta-language breaks.

The Spoofax language workbench supports live language development and bootstrapping (Figures 2,3) in the sense that a language can be reloaded and feedback can be observed. Feedback for changes to the syntax specification is fast, but changes to the static analysis and transformation is not. In the future, we would like to make this kind of feedback fast as well.

References

[1] Lorenzo Bettini. Implementing java-like languages in xtext with xsemantics. In *SAC*, pages 1559–1564, 2013.

[2] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *SCP*, 72(1-2):52–70, 2008.

[3] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: implementing domain-specific languages for java. In *GPCE*, pages 112–121, 2012.

[4] Sebastian Erdweg. *Extensible Languages for Flexible and Principled Domain Abstraction*. PhD thesis, Philipps-Universität Marburg, March 2013.

[5] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. A sound and optimal incremental build system with dynamic dependencies. pages 89–106. ACM, 2015.

[6] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: library-based syntactic language extensibility. In *OOPSLA*, pages 391–406, 2011.

[7] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *OOPSLA*, pages 307–309, 2010.

[8] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA*, pages 444–463, 2010.

[9] Steven Kelly, Kalle Lyytinen, and Matti Rossi. Metaedit+: A fully configurable multi-user and multi-tool case and came environment. In *caise*, pages 1–21, 1996.

[10] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. Easy meta-programming with rascal. In *GTTSE*, pages 222–289, 2009.

[11] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *SCAM*, pages 168–177, 2009.

[12] Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *SLE*, pages 311–331, 2012.

[13] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *PLDI*, pages 132–141, 2011.

[14] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[15] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards user-friendly projectional editors. In *SLE*, pages 41–61, 2014.

[16] Guido Wachsmuth, Gabriël D. P. Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser. A language independent task engine for incremental name and type analysis. volume 8225 of *Lecture Notes in Computer Science*, pages 260–280. Springer, 2013.