Bootstrapping Domain-Specific Meta-Languages in Language Workbenches

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

Abstract

It is common practice to bootstrap compilers of programming languages. By using the compiled language to implement the compiler, compiler developers can code in their own high-level language and gain a large-scale test case. In this paper, we investigate bootstrapping of compiler-compilers as they occur in language workbenches. Language workbenches support the development of compilers through the application of multiple collaborating domain-specific meta-languages for defining a language's syntax, analysis, code generation, and editor support. We analyze the bootstrapping problem of language workbenches in detail, propose a method for *sound* bootstrapping based on fixpoint compilation, and show how to conduct breaking meta-language changes in a bootstrapped language workbench. We have applied sound bootstrapping to the Spoofax language workbench and report on our experience.

Categories and Subject Descriptors D.3.4 [*Compilers*]

Keywords bootstrapping, domain-specific, meta-language, language workbench

1. Introduction

A bootstrapped compiler can compile its own source code, because the compiler is written in the compiled language itself. Such bootstrapping yields four main advantages:

1. A bootstrapped compiler can be written in the compiled high-level language,
2. it provides a large-scale test case for detecting defects in the compiler and the compiled language,
3. it shows that the language's coverage is sufficient to implement itself, and
4. compiler improvements such as better static analysis or the generation of faster code applies to all compiled programs, including the compiler itself.

Compiler bootstrapping is common practice nowadays. For example, the GCC compiler for the C language is a bootstrapped compiler; its source code is written in C and it can compile itself. More generally for a language $\mathtt{L}$, a bootstrapped compiler $\mathtt{L_c}$ should apply to its own definition $\mathtt{L_d}$ such that $\mathtt{L_d} \in \mathtt{L}$ and $\mathtt{L_c}(\mathtt{L_d}) = \mathtt{L_c}$.

Language workbenches [9] are compiler-compilers that provide high-level meta-languages for defining domain-specific languages (DSLs) and their compilers. Thus, users of a language workbench implement the compiler $\mathtt{L_c}$ of language $\mathtt{L}$ not in $\mathtt{L}$ but in a high-level meta-language $\mathtt{M}$ such that $\mathtt{L_d} \in \mathtt{M}$ and $\mathtt{M_c}(\mathtt{L_d}) = \mathtt{L_c}$. Thus, bootstrapping of $\mathtt{L_c}$ is no longer required, which is good since many DSLs have limited expressiveness and are often ill-suited for compiler development.

What we desire instead is bootstrapping of a language workbench's compiler-compiler $\mathtt{M_c}$. We want to use our meta-languages for implementing our meta-language compilers, thus inheriting the benefits of bootstrapping stated above: high-level meta-language implementation, large-scale test case, meta-language coverage, and improvement dissemination. In short, bootstrapping of language workbenches supports meta-language development. However, bootstrapping of language workbenches also entails three novel challenges:

- Most language workbenches provide separate meta-languages $\mathtt{M}^{1..n}$ for describing the different language aspects such as syntax, analysis, code generation, and editor support. Thus, to build the defintion of any one meta-language compiler $\mathtt{M}^i_\mathtt{d}$, multiple meta-language compilers $\mathtt{M}^{1..n}_\mathtt{c}$ are necessary such that $\mathtt{M}^{1..n}_\mathtt{c}(\mathtt{M}^i_\mathtt{d}) = \mathtt{M}^i_\mathtt{c}$. This entails intricate dependencies that sound language-workbench bootstrapping needs to handle.
- Most language workbenches provide an integrated development environment (IDE). Typically, language workbenches generate or instantiate this IDE based on the definition of the meta-languages. In this setup, the meta-language developer needs to restart the IDE whenever the

definition of a meta-language is changed. However, to support bootstrapping, the definition of meta-language compilers should be available *within* the IDE and no restart should be required to generate and load the new bootstrapped meta-language compilers [18]. Importantly, since meta-language changes can be defective, it also needs to be possible to rollback to an older meta-language version if bootstrapping fails.

- Since meta-languages in language workbenches depend on one another, it can become difficult to implement breaking changes that require the simultaneous modification of a meta-language and existing client code. For example, renaming a keyword in one meta-language can require modifications in the compilers of the other meta-languages. To preserve changeability, we need to support implementing such breaking changes in a bootstrapped language workbench.

We present a solution to these challenges based on versioning and fixpoint bootstrapping of meta-language compilers. That is, we iteratively self-apply meta-language compilers to derive new versions until no change occurs. For this to work, we identified properties that meta-language compilers need to satisfy: explicit cross-language dependencies, deterministic compilation, and comparability of compiler binaries. To support meta-language engineers, we describe how to build interactive environments on top of fixpoint bootstrapping. Finally, we discuss how to implement and bootstrap breaking changes in the context of fixpoint bootstrapping.

To confirm the validity of our approach, we have implemented fixpoint bootstrapping for the Spoofax language workbench [14]. We use our implementation to successfully bootstrap eight meta-languages. We present our experience with seven changes to the meta-languages. We describe how we implemented the changes, how bootstrapping helped us to detect defects, and how we handled breaking changes.

We are the first to describe a method for bootstrapping the meta-languages of a language workbench. In summary, we make the following contributions:

- We present a detailed problem analysis and requirements for language-workbench bootstrapping (Section 2).
- We describe a sound bootstrapping method based on fixpoint meta-language compilation (Section 3).
- We explain how to build bootstrapping-aware interactive environments (Section 4).
- We investigate support for implementing breaking changes in a bootstrapped language workbench (Section 5).
- We validate our approach by realizing it in Spoofax and by investigating seven bootstrapping changes (Section 6).

2. Problem Analysis

To get a better understanding of bootstrapping in the context of language workbenches, we analyze the problem of bootstrapping in more detail. This problem analysis will help us answer why we need bootstrapping in the first place, and what is required to do bootstrapping in the context of language workbenches.

2.1 Bootstrapping Example

First, we need a more realistic example that shows the complexities of bootstrapping language workbenches. As an example, we use the SDF and Stratego meta-languages from the Spoofax language workbench. SDF [28] is a meta-language for specifying syntax of a language. Stratego [4] is a meta-language for specifying term transformations. SDF and Stratego are bootstrapped by self specification and mutual specification. That is, SDF's syntax is specified in SDF, and its transformations in Stratego. Stratego's syntax is specified in SDF, and its transformations in Stratego.

SDF also contains several generators. SDF contains a pretty-printer generator `PP-gen` that generates a pretty-printer based on the layout and concrete syntax in a syntax specification [30]. A pretty-printer (sometimes called an unparser) is the inverse of a parser. It takes a parsed abstract syntax tree (AST) and pretty-prints it back to a string. The generated pretty-printer is a Stratego program that performs this function. Besides generating a pretty-printer, SDF contains a signature generator `Sig-gen` that generates signatures for the nodes occurring in the AST. Since these signatures serve as a basis for AST transformations in Stratego, SDF describes these signatures in Stratego syntax and pretty-prints them using the generated Stratego pretty-printer.

Overall, our scenario entails various complex dependencies across languages. In the remainder of this section, we focus on the following dependency chain:

- The pretty-printer generator translates SDF ASTs into Stratego ASTs and thus requires the SDF and Stratego signatures.
- The SDF signatures are generated by the signature generator using the Stratego pretty-printer.
- The Stratego pretty-printer is generated by the pretty-printer generator, from the Stratego syntax definition.
- The pretty-printer generator is implemented as a Stratego program within SDF.

We want to apply bootstrapping to SDF and Stratego to detect defective changes to a language's implementation. In order to illustrate the difficulties of bootstrapping in the context of language workbenches, we will deliberately introduce a defect in the implementation of the pretty-printer generator. Normally, a pretty-printer needs to align with the parser such that $parse(pretty\text{-}print(ast)) = ast$. We break the the pretty-printer generator to violate this equation by generating pretty-printers that print superfluous semicolons. This is an obvious way to sabotage the pretty-printer generator and will cause parse failures when parsing a pretty-printed string. We expect bootstrapping to detect this defect. Figure 1 shows

an iterative bootstrapping attempt with relevant dependencies, illustrating code examples, and an explanation for each bootstrapping iteration.

We start with a baseline of language implementations. We introduce the defect in the pretty-printer generator and start rebuilding the whole system in Iteration 1 using the baseline. However, despite the defect, all components build fine in Iteration 1. This is because it takes multiple iterations for the defect to propagate through the system before it produces an error. In our example, the defective pretty-printer generator (Iteration 1) generates a broken pretty-printer (2), which is used by the signature generator (3), which then generates signatures in Stratego syntax but with superfluous semicolons (4). All defects remain undetected until the build of `PP-gen` or `Sig-gen` in Iteration 4 fails because of parse errors in the signatures.

Our example illustrates multiple points. First, dependencies between components in a language workbench are complex, circular, and across languages. Second, language bootstrapping yields a significant test case for language implementations and can successfully detect defects. Third, a single build is insufficient because many defects only materialize after multiple iterations of rebuilding.

This example still far removed from the complexity that language workbenches face in practice. For example, Spoofax features eight interdependent meta-languages and SDF alone has seven generators that uses pretty-printers from four other meta-languages.

2.2 Requirements

Based on our example above, we derive requirements for sound bootstrapping support in language workbenches.

Sound Bootstrapping In our example, we needed 4 bootstrapping iterations to find a failure caused by the defective pretty-printer. In general, there is no way to know how many iterations are necessary until a defect materializes or after how many iterations it is safe to stop. Therefore, for sound bootstrapping it is required to iterate until reaching a fixpoint, that is, until the build stabilizes.

To determine if a fixpoint has been reached, we must be able to compare the binaries that meta-languages generate. We have reached a fixpoint if the generated binaries in iteration $k + 1$ are identical to the binaries generated in iteration k. Since the binaries are the same, further rebuilds after reaching a fixpoint cannot change the implementation or detect new defects.

A further requirement for fixpoint bootstrapping is that compilers must be deterministic. That is, when calling a compiler with identical source files, the compiler must produce identical binaries.

Bootstrapping always requires a baseline of meta-language binaries to kickstart the process. Bootstrapping uses the baseline only to rebuild the meta-languages in the first bootstrapping iteration. After that, bootstrapping uses the bootstrapped binaries.

Finally, the bootstrapping system should be general; it should work for any meta-language in the language workbench.

Interactive Bootstrapping Environment Besides having a bootstrapping system that satisfies the requirements above, we also need to support bootstrapping in the interactive environments of language workbenches. In particular, an interactive environment needs to provide operations that (1) start a bootstrapping attempt, (2) load a new baseline into the environment after bootstrapping succeeded, (3) roll back to an existing baseline after bootstrapping failed, and (4) cancel non-terminating bootstrapping attempts.

Loading a baseline needs to be such that subsequent bootstrapping attempts use the new baseline. When bootstrapping fails, a rollback to the existing baseline is required such that the defect causing the failure can be fixed and a new bootstrapping attempt can be started. All operations should work within the same language workbench environment, without requiring a restart of the environment, or a new environment to be started.

Bootstrapping Breaking Changes Bootstrapping helps to detect changes that break a language implementation. However, sometimes breaking changes are desirable, for example, to change the syntax of a meta-language. If we change the syntax definition of some language L and the code written in L simultaneously, bootstrapping fails to parse the source code in Iteration 1 because the baseline only supports the old syntax of L. If we change the syntax definition of L but leave the code written in L unchanged, bootstrapping fails to parse the source code in Iteration 2 because the parser of L generated in Iteration 1 only supports the new syntax of L.

The bootstrapping environment should provide operations for bootstrapping breaking changes.

3. Sound Bootstrapping

Compiling or bootstrapping a meta-language is a complex operation that requires application of generators from many meta-languages, to a meta-language that consist of sources in several meta-languages. Therefore, we would like to find a general compilation and bootstrapping algorithm.

We describe a method for *sound bootstrapping* that fulfills the requirements from the previous section. As a first step towards compilation and bootstrapping, we introduce a general model for meta-language definitions and products. Using the model, we describe a general compilation algorithm that compiles a meta-language definition into a meta-language product. Finally, we show how to perform fixpoint bootstrapping operations based on the model and compilation algorithm. We use the bootstrapping scenario from Figure 1 as a running example in this section.

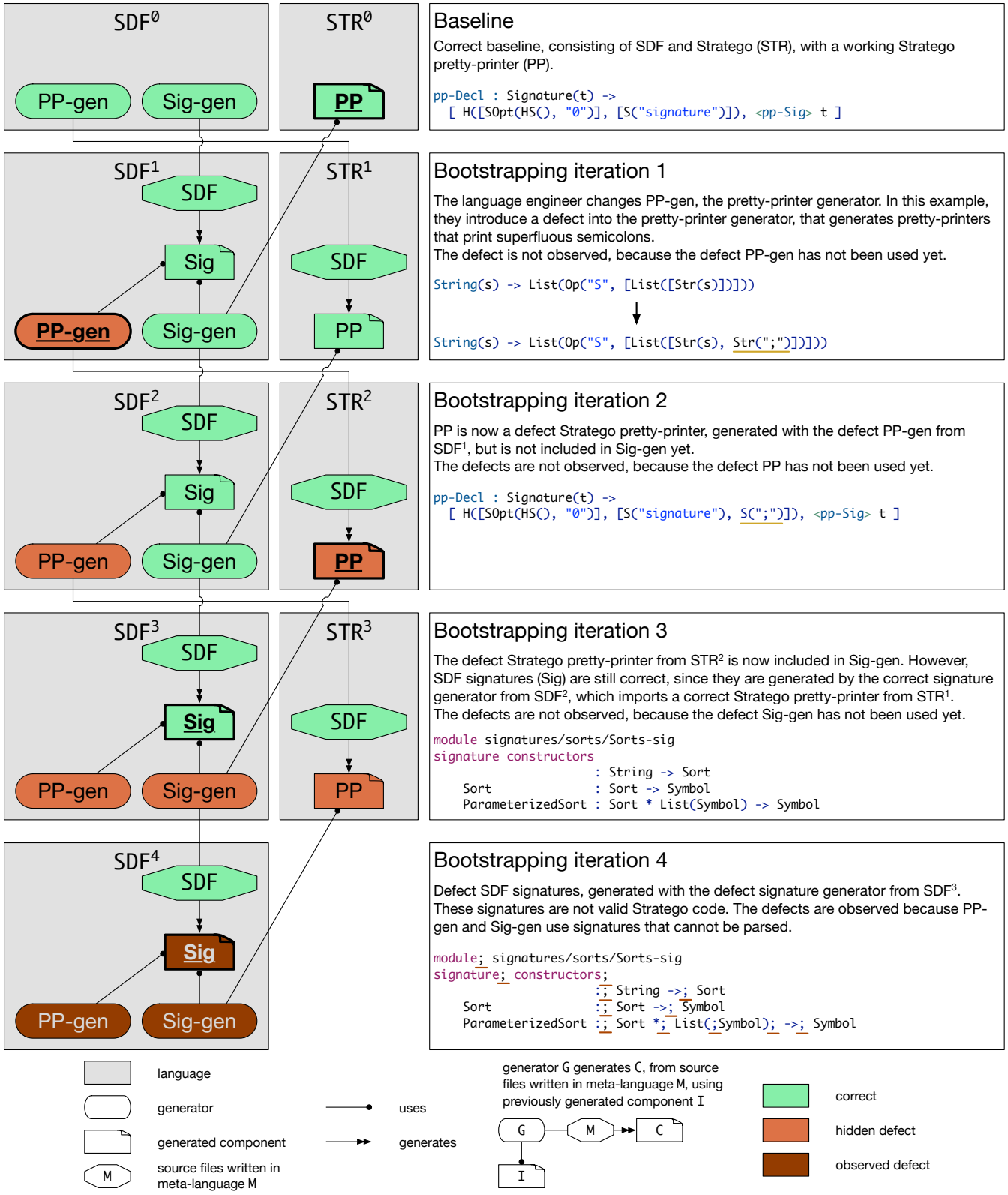

Figure 1. Bootstrapping flow for bootstrapping SDF and Stratego with a defect pretty-printer generator. In each iteration, SDF and Stratego are compiled, based on their previous versions. For example, $\texttt{SDF}^2$ is compiled with $\texttt{SDF}^1$ and $\texttt{STR}^1$. In the fourth iteration, bootstrapping fails because of a parse error, which can be traced back to the change which introduces a defect into the pretty-printer generator in the first iteration. Source code on the right belongs to underlined/bold components on the left.

3.1 Language Definitions and Products

As a first step towards bootstrapping, we introduce a general model for language definitions and products. We require such a model to describe a general compilation and bootstrapping algorithm for meta-languages. Figure 2 shows the model encoded in Haskell. In this subsection, we explain the model. In later subsections, we explain the compilation and fixpoint bootstrapping algorithm.

Language First of all, we use a *unique name* to identify each meta-language `Lang` of a language workbench, such as SDF and Stratego. However, a name alone is not enough to uniquely distinguish meta-languages. Multiple versions of the same meta-language exist when bootstrapping, for example, a baseline version of SDF and the first bootstrapping iteration of SDF. Therefore, we also use a *version* to identify a language, `LangID` in the model. We denote a language L with version 1 as $\mathtt{L}^1$. For example, with versioning, we can uniquely identify different versions of SDF and Stratego: $\mathtt{SDF}^0$, $\mathtt{STR}^0$, $\mathtt{SDF}^1$, $\mathtt{STR}^1$.

Bootstrapping applies the generators of a meta-language to the definition of its own and other meta-languages. Therefore, it is important to distinguish a meta-language *definition* from a meta-language *product*, which results from compiling the definition. The example in Figure 1 does not make this distinction to reduce its complexity, but we require this distinction here in order to precisely define compilation and bootstrapping.

Language Definition Each language definition `LangDef` defines a specific version of a language. We denote the definition of language L at version 1 as $\mathtt{L}^1_\mathtt{d}$. The definition consists of source artifacts written in different meta-languages (field `alang` of `Artifact`). To compile a language definition, we need to know what external artifacts and generators it requires. To this end, a language definition defines artifact and generator dependencies on previous versions of itself or on specific versions of other languages. We use these dependencies during compilation.

Language Product A language product `LangProd` models a compiled meta-language definition. We denote the product of compiling $\mathtt{L}^1_\mathtt{d}$ as $\mathtt{L}^1_\mathtt{p}$. A product exports artifacts and generators. A generator `Generator` transforms artifacts of some source language into artifacts of some target language. For example, `Sig-gen` in SDF transforms SDF artifacts into Stratego artifacts, or fails if the SDF artifacts are invalid, which we model as a dynamic exception of function `generate`.

Example Language definitions and products model the dependencies required to compile a definition into a product, which we describe in the next subsection. For example, $\mathtt{SDF}^1_\mathtt{d}$ requires the application of generator `Sig-gen` of $\mathtt{SDF}^0_\mathtt{p}$, whereas $\mathtt{STR}^1_\mathtt{d}$ requires the application of generator `PP-gen` of $\mathtt{SDF}^0_\mathtt{p}$. Moreover, $\mathtt{SDF}^1_\mathtt{d}$ requires the pretty-print table artifact `PP` of $\mathtt{STR}^0_\mathtt{p}$.

3.2 Compilation

Before we can bootstrap multiple meta-language definitions against a baseline of meta-language products, we must first be able to compile a single meta-language definition. We describe the compilation algorithm that compiles a single language definition using the model from above.

Function `compile` takes a language definition and a baseline of language products, and produces a new language product from the definition. The basic idea of the algorithm is to run the required generators on the source artifacts and the required external artifacts. This yields new generated artifacts that we package into a language product using `createLangProd`.

We first collect all generator inputs, which are the source artifacts `(dsources def)` of the definition and the required artifacts according to dependencies `(dartDeps def)`. We use the baseline to resolve dependencies; function `getProd` finds the product of the required `LangID`. Similar to required artifacts, we collect the required generators according to dependencies `(dgenDeps def)`.

When running generators, we need to make sure to call them in the right order: A generator must run later if it consumes an artifact produced by another generator. For example, $\mathtt{SDF}^1_\mathtt{d}$ requires the application of generator `Sig-gen`, which produces Stratego code. But $\mathtt{SDF}^1_\mathtt{d}$ also requires the application of the Stratego-compiler generator of $\mathtt{STR}^0_\mathtt{p}$, which translates Stratego code into an executable. Thus, we must run `Sig-gen` before the Stratego compiler. To this end, we sort all languages topologically according to their source and target languages.

Function `runGenerators` iterates over the sorted source languages and for each one applies all generators of the current source language `lang`. Function `runGeneratorsFor` finds all relevant generators `g` that take artifacts of `lang` as input and it finds all relevant artifacts `a` of `lang`. It then calls the `generate` function of all relevant generators `g` on all relevant artifacts `a` and collects and returns the generated artifacts. Function `runGenerators` passes the generated artifacts down when recursing to allow subsequent generators to compile them. If any `generate` function fails with a dynamic exception, the compilation fails.

Finally, after generating all artifacts, we create a language product from the language definition and the generated artifacts by calling `createLangProd`. This function must be implemented by the language workbench. We abstract over how a language workbench determines which artifacts to export and which generators to create from generated artifacts. For example, Spoofax determines which artifacts to export from a configuration file in the language definition, has built-in notions of generators to create based on generated artifacts, and allows a language definition to configure its own generators.

```haskell
-- Model for languages, language definitions with sources, and language products with artifacts and generators.
type Version = Int
type Lang = String
data LangID = LangID { name :: Lang, version :: Version }
data Artifact = Artifact { aname :: String, alang :: Lang, acontent :: String }
data LangDef = LangDef { dlang :: LangID, dsources :: [Artifact], dartDeps :: [LangID], dgenDeps :: [LangID] }
data Generator = Generator { gname :: String, gsource :: Lang, gtarget :: Lang, generate :: Artifact -> Artifact }
data LangProd = LangProd { plang :: LangID, partifacts :: [Artifact], pgenerators :: [Generator] }
type Baseline = [LangProd]

getProd :: LangID -> Baseline -> LangProd
getProd lang baseline = fromJust $ find (\prod -> lang == plang prod) baseline

-- Compile. Sort languages by generator source/target and run relevant generators against relevant artifacts.
compile :: LangDef -> Baseline -> LangProd
compile def baseline = createLangProd def (runGenerators sortedLangs generators inputs)
  where inputs = dsources def ++ [ a | l <- dartDeps def, a <- partifacts (getProd l baseline) ]
        generators = [ g | l <- dgenDeps def, g <- pgenerators (getProd l baseline) ]
        sortedLangs = topsort [ l | LangID l _ <- dgenDeps def ] [ (gsource g,gtarget g) | g <- generators]

runGenerators :: [Lang] -> [Generator] -> [Artifact] -> [Artifact]
runGenerators [] gens inputs = inputs
runGenerators (lang:langs) gens inputs = runGenerators langs gens (inputs ++ runGeneratorsFor lang gens inputs)

runGeneratorsFor :: Lang -> [Generator] -> [Artifact] -> [Artifact]
runGeneratorsFor lang gens inputs = [ generate g a | g <- gens, a <- inputs, gsource g == lang, alang a == lang ]

createLangProd :: LangDef -> [Artifact] -> LangProd -- Implemented by the language workbench

-- Fixpoint bootstrap language definitions with a baseline. Update versions in the first iteration, then fixpoint.
bootstrap :: Version -> [LangDef] -> Baseline -> (Baseline, [LangDef])
bootstrap v defs baseline =
  let firstBuild = [ compile (setVersion v def) baseline | def <- defs ] in
    bootstrapFixpoint (prepareFixpoint v defs) firstBuild

bootstrapFixpoint :: [LangDef] -> Baseline -> (Baseline, [LangDef])
bootstrapFixpoint defs baseline =
  let newBaseline = [ compile def baseline | def <- defs ] in
    if baseline == newBaseline
    then (newBaseline, defs)
    else bootstrapFixpoint defs newBaseline

setVersion :: Version -> LangDef -> LangDef
setVersion v (LangDef (LangID l _) srcs gdeps adeps) = LangDef (LangID l v) srcs gdeps adeps

prepareFixpoint :: Version -> [LangDef] -> [LangDef]
prepareFixpoint v defs = [ prepareFixpointDef v bootstrappedLangs def | def <- defs ]
  where bootstrappedLangs = [ l | LangDef (LangID l _) _ _ _ <- defs ]

prepareFixpointDef :: Version -> [Lang] -> LangDef -> LangDef
prepareFixpointDef v langs (LangDef (LangID l _) srcs adeps gdeps) =
  LangDef (LangID l v) srcs [ updateDep v langs dep | dep <- adeps ] [ updateDep v langs dep | dep <- gdeps ]

updateDep :: Version -> [Lang] -> LangID -> LangID
updateDep v langs (LangID l vold) = if l `elem` langs then LangID l v else LangID l vold
```

Figure 2. Model for sound bootstrapping, with algorithms for compilation and fixpoint bootstrapping, encoded in Haskell.

3.3 Fixpoint Bootstrapping

We can now use compilation to define fixpoint bootstrapping. In general, there is no way to know how many bootstrapping iterations are required before it is safe to stop. Therefore, we iteratively bootstrap meta-languages until reaching a fixpoint. We define a general fixpoint bootstrapping algorithm using the model and compilation algorithm from above.

Function `bootstrap` takes the version of the new baseline, a list of meta-language definitions, and an existing baseline, and it produces a new baseline of the given version. The basic idea of the algorithm is to compile meta-language definitions in iterations, until we reach a fixpoint. However, to avoid building against the old baseline repeatedly, we have to update the versions of the language definitions in the first iteration.

In the first iteration, function `bootstrap` calls `compile` on modified definitions `def` where we have set the version to `v`. This yields a list of language products `firstBuild` that contains products of version `v`. We use this as starting point for fixpoint computation. In addition, we update the dependencies in `defs` using function `prepareFixpoint`, which updates versions and dependencies of all bootstrapped languages to `v`.

To produce a new baseline, we repeat bootstrapping in `bootstrapFixpoint` until reaching a fixpoint. In each iteration, we compile all meta-language definition into meta-language products. If the new baseline is equal to the baseline from the previous iteration, we have reached a fixpoint and return the new baseline.

To compare the language products of a baseline, we compare the name, version, artifacts, and generators of products. To compare generators, we need to compare the executables of generators (not modeled in Haskell). In practice, this boils down to comparing binary files byte-for-byte, ignoring non-deterministic metadata such as the creation date or the last modified date. We change meta-language versions each iteration in Figure 1 for illustrative purposes. However, `bootstrap` only changes versions once, to prevent baseline comparison from always failing because of version differences.

Bootstrapping fails with a dynamic exception if any compile operation fails. Otherwise, our algorithm soundly produces a new baseline. In the next section, we explain how to manage baselines in interactive environments.

4. Interactive Bootstrapping

A language workbench provides an interactive environment in which a language engineer can import language definitions, make changes to the definitions in interactive editors, compile them into a language products, and test the changed languages. This allows a language engineer to quickly iterate over language design and implementation. Likewise, a meta-language engineer wants to quickly iterate over meta-language design and implementation [18]. Therefore, we need to support running bootstrapping operations in the interactive language workbench environment.

A language workbench manages an interactive environment with a *language registry* that manages all loaded language definitions and language products. The language registry loads language definitions and products *dynamically*, that is, while the environment is running without restarting the environment or starting a new one. The language workbench should react to loading, reloading, and unloading of language definitions and products, for example, by setting up file associations and updating editors.

To support interactive development, meta-language compilation interacts with the language registry. Instead of receiving a baseline as argument, in an interactive environment function `compile` from the previous section uses the language registry to retrieve language products. Thus, a change to the registry affects subsequent compilations.

Since bootstrapping relies on `compile`, in an interactive environment bootstrapping also interacts with the language registry. Instead of receiving a baseline as argument, in an interactive environment function `bootstrap` from the previous section uses the language registry to retrieve an initial baseline. Before calling `compile` in each iteration, `bootstrap` needs to load/reload the compiled products of version `v`. If bootstrapping succeeds, the new baseline stays in the language registry upon termination of `bootstrap`. But if bootstrapping fails with an exception, subsequent operations may not use the intermediate language products. To this end, `bootstrap` needs to rollback changes to the registry by unloading the language products of version `v`, and rolling back version changes in definitions.

Based on these changes to the algorithms and the language registry, our bootstrapping model supports interactive environments. Specifically, we can start a bootstrapping attempt with `bootstrap`, load a new baseline into the registry, and rollback the registry after bootstrapping failed or was canceled by the user.

5. Bootstrapping Breaking Changes

In the context of bootstrapping, a breaking change is a change to meta-language definitions such that fixpoint bootstrapping fails. Instead of treating such change as a whole, a breaking change needs to be decomposed into multiple smaller changes for which fixpoint bootstrapping succeeds.

For example, changing a keyword in the SDF meta-language is a breaking change, because it will cause parse failures for SDF source files elsewhere. Changing a keyword and all usages of the old keyword is also a breaking change, because we use the old baseline on the first build, which does not support the new keyword.

To perform such a breaking change, we need decompose it into *smaller non-breaking changes*. Using such decomposition, fixpoint bootstrapping succeeds after each change. However, it is actually sufficient to only perform a full fixpoint bootstrap after the final change and to only find defects then. For the intermediate changes, it is enough to bootstrap a single iteration in order to construct a new baseline for the subsequent builds. To support this, we propose to extend the interactive environment with an additional bootstrapping operation that bootstraps a single iteration only. This will still find all defects in the final fixpoint bootstrapping, but intermediate defects may go unnoticed until then.

A common breaking change that occurs when evolving a meta-language is the change of a feature F to F'. For example, this includes changing the syntax of Stratego or changing the `Sig-gen` generator in SDF. We can decompose a change of feature F to F' in `M` by (1) adding F' as an alternative to F in `M`, (2) executing a single bootstrap iteration, (3) changing all source artifacts written in `M` to use F' instead of F, (4) executing a single bootstrap iteration, (5) removing F from `M`, and finally (6) performing a fixpoint bootstrap.

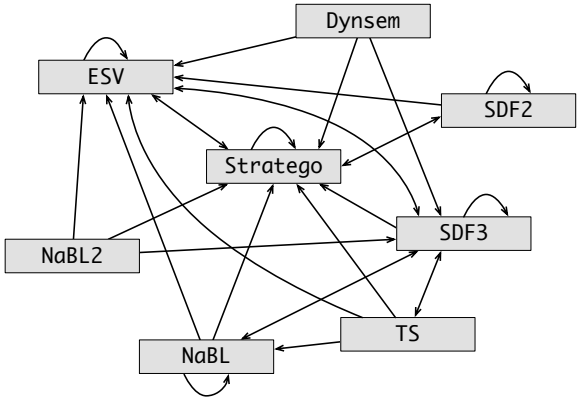

Figure 3. Generator dependencies between meta-languages. A generator dependency indicates that a meta-language requires (some) generators of a meta-language.

We have successfully used this decomposition for changing features in our evaluation.

6. Evaluation

To evaluate our bootstrapping method, we realized it in the Spoofax language workbench and bootstrapped Spoofax's eight meta-languages.

6.1 Implementation

We have implemented the model, general compilation algorithm, and general bootstrapping algorithm of our sound bootstrapping method in the interactive Eclipse environment of the Spoofax language workbench. With our implementation, a meta-language engineer can import meta-language definitions into Eclipse, make changes to the definitions, and run bootstrapping operations on the definitions to produce new baselines. The Eclipse console displays information about the bootstrapping process, e.g. when a new iteration starts, which artifacts were different during language product comparison, and any errors that occur during bootstrapping.

When bootstrapping fails, changes are reverted, and the console shows observed errors. Bootstrapping can also be cancelled by cancelling the bootstrapping job. When bootstrapping succeeds, the new baseline and meta-language definitions are dynamically loaded, such that the meta-language engineer can start making changes to the definitions and run new bootstrapping operations.

6.2 Meta-languages

To evaluate the bootstrapping method and implementation, we bootstrap Spoofax's meta-languages. Spoofax currently consists of eight meta-languages: SDF2, SDF3, Stratego, ESV, NaBL, TS, NaBL2, and DynSem. The generator dependencies between these meta-languages are shown in Figure 3.

Syntax used to be specified in the SDF2 [28] language, but we have since moved on to the more advanced SDF3 [30] language with syntax templates from which pretty-printers are automatically derived. SDF2 still exists because of compatibility reasons (some languages still use it) but also to use it as a target for generation. The SDF3 compiler generates SDF2, which the SDF2 compiler turns into a parse table. The SDF that we have been using as a running example in this paper is actually SDF3.

ESV is a domain-specific meta-language for specifying editor services such as syntax coloring, outlines, and folding. Every meta-language (including ESV itself) uses ESV to specify its editor services, such that Spoofax can derive an editor for the meta-languages.

Stratego [4, 29] is used for specifying term transformations and static semantics in several (meta)languages, and is also a common target for generation. For the name and type analysis domains, NaBL [19] is a domain-specific meta-language for specifying name analysis, and TS for specifying type analysis in terms of typing rules. NaBL2 is an evolution of NaBL that combines NaBL and TS in one language. Again, the older version of the language is kept for compatibility reasons. Finally, DynSem [27] is a meta-language for dynamic semantics specification through operational semantics.

We have successfully bootstrapped these meta-languages with our bootstrapping implementation.

6.3 Bootstrapping Changes

We evaluate our bootstrapping method and its implementation in Spoofax by bootstrapping changes to Spoofax's meta-languages. We could not test our bootstrapping implementation against existing changes made to the meta-languages, because our bootstrapping implementation expects meta-languages to be in a specific format which existing meta-languages are not. Therefore, we converted the meta-languages to this format and constructed realistic and interesting changes for evaluation.

We have logged the changes in the form of a Git repository[1], which contains a readme file explaining how to view the repository. Each change is tagged with a version in the Git repository. For each tag, sources and binaries of the created baseline are available. Tags for fixpoint bootstrapping operation also include the bootstrapping log. We now go over each change scenario to explain what we changed, why we made the change, and any issues that occurred.

Initial Bootstrap To be able to bootstrap Spoofax's meta-languages, we convert the meta-language definitions to work with our bootstrapping implementation. We successfully bootstrap the meta-languages by running a fixpoint bootstrapping operation. Version *v2.1.0* is the first fixpoint bootstrap that includes all meta-languages, and will be used as a baseline for the next change.

SDF2 in SDF3 SDF2 currently exports a handwritten pretty-printer, which is imported by SDF3 to pretty-print SDF2 source files. A handwritten pretty-printer is bad for

[1] `https://github.com/spoofax-bootstrapping/bootstrapping`

maintenance because it must be manually changed whenever the syntax changes. However, a generated pretty-printer is automatically updated in conjunction with changes to the syntax, which reduces the maintenance effort. Therefore, we convert SDF2's syntax to SDF3, such that SDF3's pretty-printer generator, generates a pretty-printer to replace the handwritten one.

This is a breaking change because SDF3 imports SDF2's pretty printer, and we change the pretty-printer that SDF2 exports, so SDF3's imports need to change. We decompose the change into three parts by applying the feature change decomposition shown previously: (1) we convert SDF2's syntax to SDF3 and export the generated pretty-printer, while still exporting the handwritten pretty-printer, (2) we change SDF3 to use the generated pretty-printer from SDF2, and (3) we remove the handwritten pretty-printer from SDF2. We apply each bootstrapping operation in the same environment, to test the interactive language workbench environment.

We first convert SDF2's syntax to SDF3, and run a single iteration bootstrapping operation to produce baseline *v2.1.1*. However, we have converted the syntax of SDF2 wrongly, constructor names are supposed to be in lowercase to retain compatibility with existing SDF2 transformations. Furthermore, lowercase constructor names such as `module` conflict with Stratego's syntax, which uses `module` as a reserved keyword. Therefore, we change SDF3 to support quotation marks in constructor names to support `'module`, which does not conflict with Stratego, and run a single iteration bootstrapping operation to produce baseline *v2.1.2*.

We convert SDF2's grammar again, with lowercase constructor names, and prefix reserved keywords with `'` where needed, run a single iteration bootstrapping operation to create baseline *v2.1.3*. We use the newly generated signatures and pretty-printer from SDF2 in SDF3, run a fixpoint bootstrapping operation (to confirm that the pretty-printer works), which succeeds, and produces baseline *v2.1.4*.

Finally, we clean up SDF2 by removing the handwritten pretty-printer. We also fix a bug in Stratego that causes some imports to loop infinitely during analysis, that was uncovered by the import dependencies between SDF2 and SDF3, which now mutually import each other. A fixpoint bootstrapping operation produces baseline *v2.1.5*.

Stratego in SDF3 We convert Stratego's syntax from SDF2 to SDF3 to also benefit from generated signatures and pretty-printers, instead of handwritten ones. This breaking change is decomposed in a similar way. However, we do not remove the handwritten pretty-printer yet, because multiple other meta-languages are using it, while we only change NaBL to use the generated one.

We convert Stratego's syntax to SDF3, run a single iteration bootstrapping operation to produce baseline *v2.1.6*. We change NaBL to use the newly generated Stratego signatures and pretty-printer, and run a fixpoint bootstrapping operation to produce baseline *v2.1.7*.

Results We were able to successfully bootstrap eight meta-languages with realistic changes, including complex breaking changes that required multiple bootstrapping steps. Bootstrapping is sound because it terminates, finds defects when we introduce them, and produces a baseline when bootstrapping succeeds. We were also able to run multiple bootstrapping operations in the interactive language workbench environment, where the baseline produced from bootstrapping is loaded into the environment and used to kickstart the next bootstrapping operation.

7. Related Work

We now discuss related work on bootstrapping.

Bootstrapped General-Purpose Languages Most general-purpose programming languages are bootstrapped. We discuss early languages and compilers that were bootstrapped.

The first programming language to be bootstrapped in 1959 is the NELIAC [12] dialect of the ALGOL 58 language. The main advantage of bootstrapping the compiler is implementation in a higher-level language. Instead of writing the compiler in assembly, it could be written in NELIAC itself, which is a much higher-level language than assembly. This allowed the compiler to be more easily be cross-compiled to the assembly of other machines, since the cross-compiled versions could be written in NELIAC.

Lisp was bootstrapped by creating a Lisp compiler written in Lisp, which was interpreted by an existing Lisp interpreter [23]. It is the first compiler that compiled itself by being interpreted by an existing interpreter of that language.

The Pascal P compiler [13] is a compiler for the minimal subset of standard PASCAL that it can still compile itself. It generates object code for a hypothetical stack computer SC. The first version of the compiler is written in assembly for SC. Using an assembler or interpreter for SC, the first compiler is compiled or executed. The compiler is bootstrapped by writing the compiler in itself, and compiling it with the existing compiler. The bootstrapped compiler is validated by comparing the assembled compiler binary and the bootstrapped compiler binary, a convention that we still apply to this day.

Bootstrapping We now look at literature on the art of bootstrapping itself.

Tombstone diagrams (also called T-diagrams) are a graphical notation for reasoning about translators (compilers), first introduced in [3] and extended with interpreters and machines in [6]. T-diagrams are most commonly used to describe bootstrapping, cross-compilation, and other processes that require executing complex chains of compilers, interpreters, and machines [20]. T-diagrams are a useful tool for graphically reasoning about compilers and bootstrapping, orthogonal to the bootstrapping framework presented in this paper.

Axiomatic bootstrapping [1] is an approach for reasoning about bootstrapping using axioms and equations between

those axioms, to verify if a change to a compiler will result in a successful bootstrap. They present axioms for an interactive ML runtime and compiler, which compiles to native code, and needs to deal with multiple architectures, calling conventions, and binary formats. For example, instantiating the axioms and equations show that changing the calling convention of the compiler causes bootstrapping to fail, and that a special cross-compilation operation can bootstrap the change.

Axioms are a useful tool to verify if a single bootstrapping iteration will work, and to reason about how a breaking change should be split up over multiple bootstrapping steps. However, axioms cannot be used to find hidden defects, such as the example from Figure 1, because those defects can manifest over an arbitrary number of iterations. Axioms also cannot be used to reason if a fixed point can be reached, for the same reason. Therefore, it is complementary to the bootstrapping framework presented in this paper.

Language Workbenches We now look at related work on bootstrapping in language workbenches.

Xtext [2] is a framework and IDE for the development of programming languages and DSLs. Its Grammar and Xtend meta-languages are bootstrapped. However, Xtext does not support dynamic loading, so it is not possible to bootstrap the meta-languages in the language workbench environment.

MPS [31] is a projectional language workbench. It has several meta-languages which are bootstrapped, but are read-only in the language workbench environment, meaning that they cannot be bootstrapped in the language workbench environment. MPS supports dependencies between languages through its powerful extension system. A meta-language can be extended, and that extension could be bootstrapped. However, MPS does not support versioning or undoing changes, making rollbacks impossible if defects are introduced.

MetaEdit+ [15] is a graphical language workbench for domain-specific modeling. Some of its meta-languages are bootstrapped, and can be bootstrapped in the language workbench environment. When changes are applied, they immediately apply to the bootstrapped meta-language and other languages. If an applied change breaks the language, the change can be undone or abandoned entirely to go back to a previous working state, after which the error can be fixed. However, they do not document their bootstrapping method, so it cannot be applied to other language workbenches.

Ensō [21, 26] is a project to enable a software development paradigm based on interpretation and integration of executable specification languages. Ensō's meta-languages are bootstrapped, but has no general framework for fixpoint bootstrapping or versioning of meta-languages. The meta-language engineer has to write code which handles fixpoint bootstrapping and versioning specifically for their meta-languages.

Rascal [16, 17] is a metaprogramming language and IDE for source code analysis and transformation. In the current version, Rascal's parser is bootstrapped, but the rest is implemented as an interpreter in Java. Development versions include a new Rascal compiler which is completely bootstrapped. However, the Rascal IDE has no general support for fixpoint bootstrapping or versioning of languages.

SugarJ [7, 8] is a Java-based extensible programming language that allows programmers to extend the base language with custom language features. In principle, SugarJ can be bootstrapped, because its compiler is written in Java and Java is a subset of SugarJ. However, in practice this was never done, and it is not obvious if that would actually work.

Racket [25] is an extensible programming language in the Lisp/Scheme family, which can serve as a platform for language creation, design, and implementation. DrRacket [10, 11] is the Racket IDE. Racket is mostly bootstrapped, but the core of the compiler and interpreter are implemented in C. The parts of Racket that are written in Racket can be changed interactively in DrRacket, which affects subsequently running Racket programs. A defect introduced in Racket's self definition may prevent bootstrapping to succeed, which requires a restart of the DrRacket IDE.

Staged Metaprogramming Staged metaprogramming approaches such as MetaML [24], MetaOCaml [5], Mint [32], and LMS [22] provide typesafe run-time code generation, which ensures that generated code does not contain typing defects. However, these approaches do not provide support for bootstrapping.

8. Conclusion

Bootstrapping is an efficient means for detecting defects in compiler implementations and should be useful for language workbenches as well. However, bootstrapping compiler-compilers of language workbenches needs to handle the intricate interactions between meta-languages. Unfortunately, previous literature on bootstrapping ignores these intricacies.

We present a sound method for meta-language bootstrapping. Given a baseline and updated meta-language definitions, our bootstrapping algorithm constructs a new baseline through fixpoint self-application of the meta-languages. We explain how our algorithms can be used in interactive environments and how to decompose breaking changes that occur when evolving meta-languages.

We have implemented the approach in the Spoofax language workbench and evaluated it by successfully bootstrapping eight interdependent meta-languages, and report on our experience with bootstrapping two breaking changes. This makes Spoofax into a laboratory for meta-language design experimentation.

Acknowledgements

This research was supported by NWO/EW Free Competition Project 612.001.114 (Deep Integration of Domain-Specific Languages) and NWO VICI Project (639.023.206) (Language Designer's Workbench).

References

[1] Andrew W. Appel. Axiomatic bootstrapping: A guide for compiler hackers. *ACM Transactions on Programming Languages and Systems*, 16(6):1699–1718, 1994.

[2] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2nd edition, 2016.

[3] Harvey Bratman. A alternate form of the "uncol diagram". *Communications of the ACM*, 4(3):142, 1961.

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

[5] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*, volume 2830 of *Lecture Notes in Computer Science*, pages 57–76. Springer, 2003.

[6] Jay Earley and Howard E. Sturgis. A formalism for translator interactions. *Communications of the ACM*, 13(10):607–617, 1970.

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

[8] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: library-based syntactic language extensibility. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 391–406. ACM, 2011.

[9] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly 0001, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.

[10] Robby Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: a programming environment for scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

[11] Robert Bruce Findler and PLT. DrRacket: Programming environment. Technical Report PLT-TR-2010-2, PLT Design Inc., 2010. `http://racket-lang.org/tr2/`.

[12] Harry D. Huskey, M. H. Halstead, and R. McArthur. NELIAC - dialect of ALGOL. *Commun. ACM*, 3(8):463–468, August 1960.

[13] K. Jensen H. H. Nägeli K. V. Nori, U. Ammann. The PASCAL <P> compiler: Implementation notes. Technical report, ETH Zürich, 1974.

[14] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM.

[15] Steven Kelly, Kalle Lyytinen, and Matti Rossi. Metaedit+: A fully configurable multi-user and multi-tool case and came environment. In Panos Constantopoulos, John Mylopoulos, and Yannis Vassiliou, editors, *Advances Information System Engineering, 8th International Conference, CAiSE 96, Heraklion, Crete, Greece, May 20-24, 1996, Proceedings*, volume 1080 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 1996.

[16] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. Easy meta-programming with rascal. In Joao M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*, volume 6491 of *Lecture Notes in Computer Science*, pages 222–289. Springer, 2009.

[17] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*, pages 168–177. IEEE Computer Society, 2009.

[18] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. Towards live language development. In *Proceedings of Workshop on Live Programming Systems (LIVE)*, 2016.

[19] Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer, 2012.

[20] Olivier Lecarme, Mireille Pellissier, and Marie-Claude Thomas. Computer-aided production of language implementation systems: A review and classification. *Software: Practice and Experience*, 12(9):785–824, 1982.

[21] Alex Loh, Tijs van der Storm, and William R. Cook. Managed data: modular strategies for data abstraction. In Gary T. Leavens and Jonathan Edwards, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2012, part of SPLASH '12, Tucson, AZ, USA, October 21-26, 2012*, pages 179–194. ACM, 2012.

[22] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In Eelco Visser and Jaakko Järvi, editors, *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*, pages 127–136. ACM, 2010.

[23] M. Levin T. Hart. Ai memo 39 - the new compiler. Technical report, MIT, 1962.

[24] Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[25] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 132–141. ACM, 2011.

[26] Tijs van der Storm, William R. Cook, and Alex Loh. Object grammars. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 4–23. Springer, 2012.

[27] Vlad A. Vergu, Pierre Neron, and Eelco Visser. DynSem: A DSL for dynamic semantics specification. In Maribel Fernández, editor, *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*, volume 36 of *LIPIcs*, pages 365–378. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

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

[29] Eelco Visser. A bootstrapped compiler for strategies (extended abstract). In B. Gramlich, H. Kirchner, and F. Pfenning, editors, *Strategies in Automated Deduction (STRATEGIES'99)*, pages 73–83, Trento, Italy, July 5 1999.

[30] Tobi Vollebregt, Lennart C. L. Kats, and Eelco Visser. Declarative specification of template-based textual editors. In Anthony Sloane and Suzana Andova, editors, *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*, page 8. ACM, 2012.

[31] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards user-friendly projectional editors. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering - 7th International Conference, SLE 2014, Västeras, Sweden, September 15-16, 2014. Proceedings*, volume 8706 of *Lecture Notes in Computer Science*, pages 41–61. Springer, 2014.

[32] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 400–411. ACM, 2010.