DOI: 10.2298/CSIS130115069L

Testing framework for embedded languages*

Dániel Leskó[1] and Máté Tejfel[1]

Eötvös Loránd University
Department of Programming Languages and Compilers
{ldani; matej}@elte.hu

Abstract. Embedding a new programming language into an existing one is a widely used technique, because it fastens the development process and gives a part of a language infrastructure for free (e.g. lexical, syntactical analyzers). In this paper we are presenting a new advantage of this development approach regarding to adding testing support for these new languages.
Tool support for testing is a crucial point for a newly designed programming language. It could be done in the hard way by creating a testing tool from scratch, or we could try to reuse existing testing tools by extending them with an interface to our new language. The second approach requires less work, and also it fits very well for the embedded approach. The problem is that the creation of such interfaces is not straightforward at all, because the existing testing tools are mostly not designed to be extendable and to be able to deal with new languages.
This paper presents an extendable and modular model of a testing framework, in which the most basic design decision was to keep the – previously mentioned – interface creation simple and straightforward. Other important aspects of our model are the test data generation, the oracle problem and the customizability of the whole testing phase.

Keywords: testing support for embedded languages, testing framework, abstraction over evaluation.

1. Introduction

Nowadays, embedding a language into an existing one (host language), is a well known and widely used approach to create a new programming language. This quickens the development process, because the host language's infrastructure (lexical, syntactical analyzer) can be reused. Modern functional host languages are flexible enough that the resulted combination has more the feel of a new language than just a library.

The "embedded" approach has proved to be an excellent technique for specifying and prototyping domain-specific languages (DSLs) [11]. Basically two approaches exist: shallow embedding, which directly maps the new language constructs to their semantics, while the deep embedding first builds an abstract syntax tree and later this tree is mapped to the language semantics.

* Supported by ELTE TÁMOP-4.2.2/B-10/1-2010-0030

Shallow embedding can be seen as an augmentation to an existing language. According to Mernik et al. [17] every library can be seen as a shallowly embedded language. While deep embedding really forms a new language on the foundations of the host language. Therefore the deep approach is more suitable for building a compilable and optimizable language. In this case a host language can be seen as a very powerful template or macro language.

There are numerous papers about embedded DSLs, such as how to design [15, 13], implement [10] or compile [7] them. However, as far as we know there are no specific paper, which aims to present a general solution for adding testing tool support – at low cost – for already existing embedded languages, while – as we all know – it is crucial for a language to have a proper testing tool support.

Implementing a testing system in a DSL is mostly not an option due to its labour-intensive nature and the fact that most DSLs are not designed for that kind of task. The realistic option is to use an already existing test environment, written in the host language. In this case we need to extend the existing framework with an interface to the embedded language, while the business logic remains untouched. Doing this is a much smaller task, than creating the testing support from scratch. This approach also fits nicely with the motivation of embeddedness, namely to save time and resources by reusing as much from the host language as possible.

The main problem with existing testing tools (QuickCheck [5], JUnit [2], HUnit [12]) that they were designed to test programs of *one* specific language and sometimes even for specific testing and test data generation methodologies. Therefore they are not easily extendible with an interface to another programming language or to another evaluation method. Furthermore they could be quite specific in certain aspects, like how the input test data are produced, or how the results are evaluated and decided whether a test case failed or passed. Another aspect regarding a testing framework is the viability and clear designability of supporting both property based and differential testing methodologies.

Our goal in this paper is to present a general and permissive model of a testing framework which can address properly all the previously mentioned aspects. The model is abstracted along four orthogonal aspects such as test data generation, the used evaluation method, the oracle problem and partly the used testing methodology.

Based on our model, a Haskell implementation was created. Its main characteristics are modularity and extensibility. The existence of such framework in Haskell (or in any host language) results that an embedded language can get a testing support by implementing only a simple and straightforward interface which spans the gap between the host and the embedded language.

2. The model

The following model was inspired by all the previously mentioned reasons and aspects, and it was designed to nicely fit for all of them. Figure 1 shows the data-flow model of a test case in our model.

The *generator*'s responsibility is to provide correctly typed input for *transformer*s. This input will be referred as the generated test data.

A *transformer* encapsulates the to-be-tested computation, and gives an universal interface, which hides such unnecessary details as how and by what the computation will be evaluated. A *transformer* can be thought as a function, whereat the generated test data is applied, and it yields the result of the computation. The number of the *transformer*s are not limited, it could be as many as the user wants.

A *property* is a function, which receives each *transformer*'s result, and also the originally generated test data. The outcome is a boolean value, which represents whether the specific property (given by the user) holds in that particular case, or not.

An *operator* is basically a driver, which controls the data-flow between the small boxes (in the figure). The configuration comes from the outside world (from the model's point of view), and it affects the *operator* (e.g. the number of performed tests).

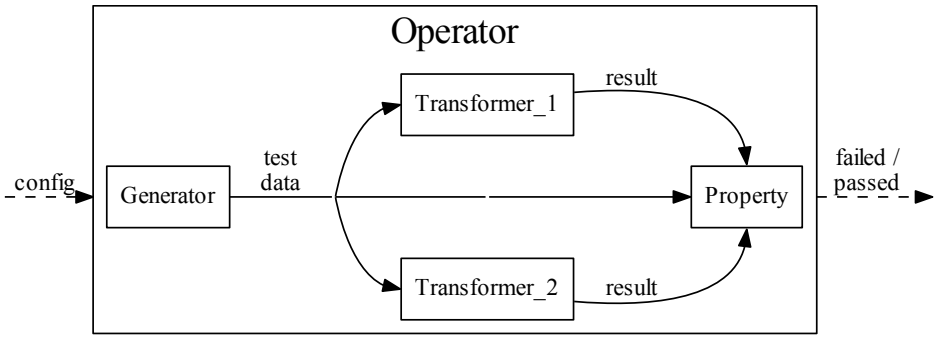

Fig. 1. The data-flow model of a test case with two transformers

To describe a test case in the terms of this model, we need one test data *generator*, a non-empty list of *transformer*s, and one *property*. To be able to run a test case, we also need an *operator*, which specifies the way how to do that.

One of the earliest design decision was that the model has to be as general as possible, in terms of that the four major parts of the model (*generator*, *transformer*, *property*, *operator*) have to be separate, independent and modular parts while the interactions between them should done through well defined and public channels.

2.1. Generator

The model's only expectation about a *generator* is that it has to supply test data with a matching type for the given *transformer*s. However there are a lot of uncontrolled aspects by the model, such as the used generation tactics (random, exhaustive), the test data distribution or controlling the size of the generated test data. All of these are entirely depending on the particular implementation of a specific *generator*.

To give control for the user over the previously mentioned aspects, we definitely need a small domain-specific language for building and specifying *generator*s. Since QuickCheck [5] and also SmallCheck [20] has some really powerful tools to do that, we can easily reuse those existing tools as a library. Note that the re-usage of such libraries not conflicts with the language independence of the model, since QuickCheck is re-implemented for more than 20 languages.

The model and the *generator* notion is not limited to the previously mentioned tools, any DSL or library which aims to generate test data could be integrated easily into the model and also into a testing framework, based on the model.

2.2. Transformer

Using a *transformer* is a way to abstract over specific evaluation methods (etc. interpreting, compiling) and specific programming languages. The model represents a *transformer* as one function, but under the hood it is a bit more complicated. A *transformer* is usually created by applying the to-be-tested function to a *transformer pattern*. Therefore the previously mentioned abstraction are done by the means of *transformer patterns*.

A specific *transformer pattern* could evaluate any program of a specific language with a specific evaluation method (e.g. C compiler, Haskell compiler, Haskell interpreter). On the implementation level it is a higher order function which takes a function (the to-be-tested) as its first argument and a complex data structure (holds the input data) as its second argument. Adding support for a new language or evaluation method can be done simply by creating a new *transformer pattern* for it.

2.3. Property

One of the most fundamental questions in automated software testing is to decide whether a test case is passed or failed, because it is hard to create an algorithm/oraculum which is general enough to correctly judge the results. However, if we somehow succeed, it is not a flexible solution to hardwire this decision method into a general framework.

A common resolution of this problem is to devolve this job to the user, who writes the actual test case. In our model it can be done with the use of the so-called *property*, which can be thought as a boolean function with two parameters. Note that our notion of *property* is quite far from QuickCheck's. Our

version means a much smaller part of the model. An expression evaluated to `true` indicates a successful test, `false` indicates a counter example. The first parameter of a *property* is the originally generated test data (the input of the transformers), the second is a list of transformer's results. The number of the compared transformers are not limited, but they have to have the same type signature.

The model does not specify what kind of results should be passed to a *property*, because that could depend on the used specific *transformer pattern*. For example a *transformer pattern* of a C compiler could pass compile time and ELOC (effective lines of code) information besides the result of the computation. Having also non-functional results can allow us to build more sophisticated *properties*.

2.4. Operator

The role of an *operator* is much more technical then the previously mentioned three parts of the model. This difference comes from the fact that an *operator* isn't part of a test case, it is only needed to perform the execution of it.

On the model level an *operator*'s job is to handle the data-flow from the *generator*s towards to a *property* through one of the *transformer*s. On the framework level there are several other technical responsibilities such as controlling the number of required iterations, the level of verbosity, setting logging and the working directory. The framework contains one predefined *operator*, which gives a standard way to handle the previously mentioned aspects. So normally a user doesn't have to create a new *operator*.

2.5. Testing framework - based on our model

The presented model is general in the sense that it doesn't require any specific programming language constructs, so it can be implemented in almost any host language. We have chosen Haskell, because it nicely fits for this job [11], and lately Haskell is a very favoured host language.

The implemented framework supports both property based testing and differential testing. As a property based tester it is a kind of generalization of QuickCheck and SmallCheck with the support for additional test data generation approaches. The most important feature – as a differential testing tool – is a "common ground" for different evaluation methods and also for different languages. The existence of this common ground makes comparability and modularity really easy.

In the following section we are discussing a detailed use case of the framework. At first sight it may look like that we are using it solely as a differential testing tool, but in reality it is rather about new *transformer pattern*s, mostly answering the why and how questions.

3. A detailed use case - Testing support for Feldspar

We used the framework to test the Feldspar[1] [1] language, which is a new embedded language for describing digital signal processing algorithms. We have more than 300 test cases, which were used on a daily basis to test the Feldspar language itself, and the existing Feldspar example programs. We also had to ensure that the Feldspar compiler and interpreter are in sync, and the results are valid (compared to trusted reference implementations or expected output sets). Besides testing equivalence, most of the test cases are checking non-functional properties too.

The actual implementation of the testing framework reused QuickCheck's `Gen` class and SmallCheck's `Serial` class as *generator*s. The used *properties* are mostly check equivalence either strictly or with a given epsilon threshold. We also used non-functional *properties*, which were about compile-time, run-time, ELOC, memory-usage.

The most important and interesting part is the *transformer*, which we present in the following subsections. Each subsection firstly introduces an aspect of Feldspar which will be tested, than shortly presents how such a specific *transformer pattern* can be created and how the testing can be achieved.

3.1. Testing the Feldspar interpreter

Unfortunately Feldspar doesn't have a written specification about semantics, so the interpreter couldn't be tested against that. But in many ways Feldspar is really similar to Haskell, in fact – by definition – a lot of primitive function and operator of Feldspar has the exact same semantics like their equivalents in Haskell.

This realization means, that we could test the Feldspar's primitive functions against their Haskell equivalents. In order to do that we need two new *transformer pattern*s. The first one will be responsible for the evaluation of a Feldspar program, using the Feldspar interpreter, while the other one will evaluate a Haskell function by the Haskell interpreter.

3.2. Testing the Feldspar compiler

The Feldspar compiler [6] has a capability of supporting several different back-ends to generate code for them. The main and mostly used platform is ANSI C, there are other developments like Ti specific instrinsics or LLVM back-end, but from the testing point of view, the C platform is the important one.

It is important from the testing point of view, that the generated C code is just a function, which has almost the same arguments, like the original Feldspar program.

[1] Developed by the Feldspar project, which was a joint research project of Ericsson; Chalmers University of Technology (Göteborg, Sweden) and Eötvös Loránd University (Budapest, Hungary).

As there is no written specification for Feldspar, the interpreter could be thought as some kind of executable reference implementation of the semantics. So the best way to verify the compiler is to test it against the interpreter.

The following steps have to be done, if we would like to compile and execute a Feldspar program and run from Haskell. So creating a *transformer pattern* – which evaluates a Feldspar program by the Feldspar compiler – requires that these steps have to be automated and built-in:

1. Compile a Feldspar program into a C function.
2. Generate a proper C main() function, which reads the input data from standard input, passes these data to the previously generated C function, and prints the result to standard output.
3. Compile the previous two C files with gcc.
4. Start an external process from Haskell to run the executable file, and feed it with proper test data.
5. Wait until the execution ends, read the result from the process, and close it.

3.3. Testing Feldspar programs against reference implementations

As it was mentioned earlier, Feldspar is a domain specific language (DSL), targeting digital signal processing (DSP). So it is obvious that there is a certain set of algorithms, which are very typical for that domain. We can assume that there are notable algorithms (e.g. Fast Fourier Transform [4]), which already have an implementation from a reliable source. These implementations can be treated as a reference to check and test the expressiveness, the usability and the correctness of the Feldspar language itself, and also the programs written in Feldspar.

Since DSP algorithms are mostly implemented in C, we need a *transformer pattern* that can evaluate an arbitrary C function. The function is either given as a string, or as an external file. Evaluation means here that the *transformer pattern* takes the C code, compiles it with a C compiler (e.g. `gcc`), generates a proper `main()` function, feeds this function with the generated input data, and reads the result back to the testing framework.

Since the C language is not embedded into Haskell, and there is no strong connection between those two languages, we are losing some type information there. While in case of Haskell and languages embedded into Haskell we could statically type-check the assembled test cases. This means that if a generator-transformer-property is ever wrongly paired or assembled, then we get a compile time type-check error.

Because of the less type information, the *transformer pattern* for C has to assume that a few invariant – regarding to the number and order of the parameters – have not been violated, otherwise we could end up with sudden run-time errors.

The typical use case of this pattern is to first have a *transformer* for a Feldspar program and then compare that with a reference C program. At first this testing approach could be a little bit confusing, because a failure doesn't

mean always a bug in the Feldspar language, it is always a possibility that the Feldspar implementation of the chosen algorithm is simply wrong. But as these test cases are based on real life examples, eventually the goal is to produce a properly working Feldspar implementation of those algorithms.

3.4. Simple testing approaches

In the following we will present, how the testing framework supports some basic testing methodology, such as negative testing, or testing with constant input.

Testing with constant data There are certain cases when the random test data generation is not enough, and we want to create some hardwired test case with specific input data. But in this case, we also might want to specify the result of the computation. It's basically the oldest and simplest version of testing, the input and the result are given manually, and we check it against the computed result.

To support this kind of testing, we need *transformer pattern*, which gets a constant as its first arguments, and results a constant transformer, always yielding the given constant.

Negative testing As it was mentioned earlier in subsection 2.2, a *transformer* could fail, but this failure is also handled as data. Besides the error message, there are some information about the source of error (e.g. Haskell, gcc, Feldspar compiler, Feldspar interpreter).

In order to build such a test case, which passes when one of the transformers fails, we need a new *transformer pattern* which constantly yields failure with the given error source.

The most likely use case for this kind of testing, if there are a certain set of Feldspar functions, which shouldn't be compiled to C, because they clearly hasn't got enough information (e.g. too general type signature) to produce a proper C code.

3.5. Concrete test case

The following is a Feldspar function, which takes an int stored on 32 bit (as the starting value of the accumulator) and a list of ints. The list is folded, while every element is added to the accumulator, which is the result of the function.

```
foldAdd :: Data Int32 -> DVector Int32 -> Data Int32
foldAdd = Feldspar.Vector.fold (+)
```

The `tc1` example test case tests the `foldAdd` function by comparing the Feldspar compiler, the Feldspar interpreter and the corresponding `foldl (+)` Haskell function.

```
tc1 = TestCase
    { tc_name = "testing fold with addition"
    , gen     = genInt32 ::> vectorOf 200 genInt32 ::> ()
    , trans   = [ refHaskellTP (Prelude.foldl (+))
                , evalTP foldAdd
                , compilerTP foldAdd]
    , prop    = base 1 strictEquality}
```

The results are strictly compared to the result of the Haskell function, which is treated as a reference implementation for Feldspar's `fold` function.

The test data is coming from a list of QuickCheck generators. Please note, that Feldspar uses fixed length lists (vectors) due to efficiency and optimization reasons, therefore we have to fix this information (200) at test data generation, otherwise we could use QuickCheck's `arbitrary` function or SmallCheck's `serial` function too, as a *generator*.

4. Improving transformers

Every testing system is made to support the development or maintenance by saving time and resources which allows to create and run more tests in an automated way.

The presented model and framework supports this goal, but still we have to create every test case manually for every function (like `foldAdd`) we would like to test, which is a very boring and time consuming work. Besides this obvious drawback there is an other disadvantage, namely that it is very easy to leave out a few functions during test case production.

This motivated us to enhance the previously presented *transformer* concept by generalizing the parameter of a *transformer pattern*. Previously a pattern expected a concrete function as its parameter to form a *transformer*. The new, enhanced *transformer pattern*s expects only a type signature instead of a concrete function.

Meta-transformers Type wise a meta-transformer looks and feels like an average *transformer*, but internally it is a bit more complicated. We can form a meta-transformer by applying only some type signature information on a *transformer pattern*. Based on this type information, the meta-transformer will internally generate the to be tested function with a matching signature to the given type information. This internal generation instantiates a meta-transformer and creates an ordinary *transformer*.

Basically we need type signature guided, automatic program generation. Since it is not an easy task, we solve it in two consecutive steps. The first phase does the real generation, and ensures the type correctness by producing closed and correctly typed lambda calculus terms. While the second phase translates the generated lambda term into a concrete program.

Palka et al. [18] successfully applies correctly typed lambda term generation to generate and test Haskell list expressions. Our approach is basically a generalization and improvement of their work.

The details and background of this two-phase generation go way beyond the scope of this paper and testing framework. Our point here is that the presented model and testing framework is so modular and flexible enough to accommodate even this kind of improvements too.

5. Related work

Helvetia [19] is a tool chain for developing an internal DSL by transforming an abstract syntax tree. The benefit of this approach is that a homogeneous tool support can be given for the newly created embedded languages. Therefore – in this case – there is no need for our testing framework. However our approach is applicable for already existing embedded languages, while to benefit from the homogeneous tool support of Helvetia we have to reimplement and embed our language into Helvetia, which is a much bigger task than just creating a *transformer* for our test framework.

So Helvetia only suits well for you if you are at the beginning of the language development process, but later it is not really an option to apply. While our solution is easily applicable for new and also existing languages too.

Testing embedded languages Grima et. al [8] developed an embedded language (in Haskell) addressing geometrical problems. The paper presents two different methodologies to test programs, written in that new language. The first simply reuses QuickCheck, while the second works on C level. Both using random input generation to verify the given properties, but they are two, completely separate solution on implementation level.

Our test framework could solve this in an *unified* way instead of those separate solutions. Furthermore the usage of our framework would save a considerable amount of time and resources, because we only need to create the two *transformer pattern*s (one for the Haskell level and one for the C level testing), the rest is already in the framework.

Test data generation The test data generation is always a crucial point in automated software testing. Numerous property or specification based testing tools are using some kind of test data generation. For example: QuickCheck [5], SmallCheck [20], Gast [14], Korat [3]. QuickCheck uses random generation (with the ability to shrink the founded counterexamples), while SmallCheck, Gast and Korat do exhaustive generation up to a limit given by the user.

It looks like that every tool supports only a specific programming language and a specific test data generation methodology. Our framework is designed to support arbitrary number of different test data generation methodologies and

also to hide their differences by giving a unified *generator* interface. For example the presented model can accommodate QuickCheck's random generator as well as SmallCheck's serial generator at the same time.

Differential testing McKeeman states the following: "The ugliest problem in testing is evaluating the result of a test." [16]. He was the first, who described the use of randomized differential testing for C compilers. His domain was essentially static: a test data was randomly generated based on a model of the valid inputs. The tested programs were compiled by different translators, and if the obtained results are different, the situation is considered to be potentially erroneous. The word "potentially" is important here, because the results – given by the two tested programs – may differ and yet still be correct depending on the requirements. This is the starting point of our *property* notion, which solves the oracle problem by simply porting it to the user.

McKeeman's model is really close to ours in the sense that he had a test data generator, translators – which corresponds with our *transformer* notion – and some kind of very simple property to check the results. However, his solution was specifically designed to test different C compilers with random test data, therefore there is no chance for such kind of extendability and flexibility like new test data generation methods, language interfaces and properties.

A reusable framework One of the simplest reusable framework is JUnit [2], and it's clones for other languages, like HUnit [12] for Haskell. Our model aims to preserve the simplicity of the previously mentioned tools, but also tries to support differential and property based testing, arbitrary test data generation methodologies and language independence in the sense of the tested program.

A test framework was developed for testing the Flash file system, and later it was reused for two other testing projects [9]. Their conclusion was that initial efforts to develop an effective test system pay off in re-use on similar projects, because the significant differences were less important than the similarities. Their experiences confirms that we have chosen the right design decision in case of our model. It also points out that resources can be saved in the long run, by developing a modular and extensible testing framework, like ours.

6. Conclusion and future work

We presented a permissive model of a modular and extensible testing framework. The main contributions of the model are the followings:

- testing support for embedded languages at low cost. To add support for a new language, we only have to create a new *transformer*, the test data generation and the basic properties are already there.
- an unified interface to support different test data generation methods. The integration of a new test data generator is very easy, nearly "plug and play".

– using abstraction over different evaluation methodologies. This latter assures that programs written in different languages can be tested against each other.

The model essentially supports property based and differential testing, where the oracle problem is ported to the user.

Language and platform independency was an early design decision for the model. A real test of this would be to try to create another (non-Haskell) implementation of the model, maybe in an object-oriented or imperative programming language.

A possible future work is to extend the framework with new *transformer patterns* to support new programming languages. For Feldspar, it could be a reasonable goal to add support for testing against reference MatLab programs.

References

1. Axelsson, E., Dévai, G., Horváth, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., Vajda, A.: Feldspar: A Domain Specific Language for Digital Signal Processing algorithms. In: Proc. Eighth MemoCode (2010)
2. Beck, K., Gamma, E.: Test infected: Programmers love writing tests. Java Report 3(7), 51–56 (1998)
3. Boyapati, R., Khurshid, S., Marinov, D.: Korat: Automated testing based on java predicates. In: In Proc. International Symposium on Software Testing and Analysis (ISSTA. pp. 123–133 (2002)
4. Brigham, E.O.: The Fast Fourier Transform (1974)
5. Claessen, K., Hughes, J.: Quickcheck: A Lightweight Tool for Random Testing of Haskell Programs. In: ACM SIGPLAN Notices. pp. 268–279 (2000)
6. Dévai, G., Tejfel, M., Gera, Z., Páli, G., Nagy, G., Horváth, Z., Axelsson, E., Sheeran, M., Vajda, A., Lyckegård, B., Persson, A.: Efficient Code Generation from the High-level Domain-specific Language Feldspar for DSPs. In: Proc. ODES-8: 8th Workshop on Optimizations for DSP and Embedded Systems, assoc. with IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (2010)
7. Elliott, C., Finne, S., Moor, O.d.: Compiling embedded languages. In: Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation. pp. 9–27. SAIG '00 (2000)
8. Grima, M., Pace, G.J.: An embedded geometrical language in haskell: Construction, visualisation, proof (2007)
9. Groce, A., Holzmann, G., Joshi, R.: Randomized differential testing as a prelude to formal verification. In: Proceedings of the 29th international conference on Software Engineering. pp. 621–631. ICSE '07 (2007)
10. Hudak, P.: Modular domain specific languages and tools. In: Proceedings of the 5th International Conference on Software Reuse. pp. 134–142. ICSR '98 (1998)
11. Hudak, P.: Building domain-specific embedded languages. ACM Comput. Surv. 28 (December 1996)
12. HUnit: Haskell unit testing. http://hunit.sourceforge.net/ (2012)
13. Kamin, S.N.: Research on domain-specific embedded languages and program generators. In: Electronic Notes in Theoretical Computer Science (1998)

14. Koopman, P., Alimarine, A., Tretmans, J., Plasmeijer, R.: Gast: Generic automated software testing. In: The 14th IFL'02, Selected Papers, volume 2670 of LNCS. pp. 84–100 (2002)
15. Leijen, D., Meijer, E.: Domain specific embedded compilers. SIGPLAN Not. 35, 109–122 (December 1999)
16. McKeeman, W.M.: Differential testing for software. Digital Technical Journal 10(1), 100–107 (December 1998)
17. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37, 316–344 (December 2005)
18. Palka, M.H., Claessen, K., Russo, A., Hughes, J.: Testing an optimising compiler by generating random lambda terms. In: Proceedings of the 6th International Workshop on Automation of Software Test. pp. 91–97. AST '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1982595.1982615
19. Renggli, L., Girba, T., Nierstrasz, O.: Embedding languages without breaking tools. In: In ECOOP 2010: Proceedings of the 24th European Conference on Object-Oriented Programming (2010)
20. Runciman, C., Naylor, M., Lindblad, F.: Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In: Proceedings of the first ACM SIGPLAN symposium on Haskell. pp. 37–48. Haskell '08 (2008)

Dániel Leskó is pursuing his Ph.D. in Test data generation and static analysis at Eötvös Loránd University in Hungary, where he received his B.Sc. and M.Sc. degree in software technology in 2008 and in 2010. His research interests include functional programming, compilers, automated testing and test data generation.

Máté Tejfel is an assistant professor at Eötvös Loránd University, Faculty of Informatics, Department of Programming Languages and Compilers. His main research topics are parallel programming, functional programming and verification.

Received: January 15, 2013; Accepted: September 10, 2013.