Transpiling Slang Methods to C Functions

An Example of Static Polymorphism for Smalltalk VM Objects

Tom Braun
tom.braun@student.hpi.de
Hasso Plattner Institute
University of Potsdam, Germany

Marcel Taeumel
marcel.taeumel@hpi.de
Hasso Plattner Institute
University of Potsdam, Germany

Eliot Miranda
eliot.miranda@gmail.com
Hasso Plattner Institute
University of Potsdam, Germany

Robert Hirschfeld
robert.hirschfeld@uni-potsdam.de
Hasso Plattner Institute
University of Potsdam, Germany

Abstract

The OpenSmalltalk-VM is written in a subset of Smalltalk which gets transpiled to C. Developing the VM in Smalltalk allows to use the Smalltalk developer tooling and brings a fast feedback cycle. However, transpiling to C requires mapping Smalltalk constructs, i.e., object-oriented concepts, to C, which sometimes requires developers to use a different design than they would use when developing purely in Smalltalk.

We describe a pragmatic extension for static polymorphism in Slang, our experience using it as well as the shortcomings of the new approach. While our solution extends the concepts developers can express in Slang, which reduces the burden of finding alternatives to well-known design patterns and by enabling the use of such patterns the modularity, it further complicates a fragile, already complicated system. While our extension solves the task it was designed for, it needs further enhancements, as does Slang itself in terms of understandability in the field.

CCS Concepts: • **Software and its engineering → Source code generation**; *Integrated and visual development environments.*

Keywords: object-orientation, modularity, polymorphism, virtual-machine development, tooling, Smalltalk, code generation

ACM Reference Format:
Tom Braun, Marcel Taeumel, Eliot Miranda, and Robert Hirschfeld. 2023. Transpiling Slang Methods to C Functions: An Example of Static Polymorphism for Smalltalk VM Objects. In *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '23), October 23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3623507.3623548

1 Introduction

When writing programs, one can choose between a multitude of programming languages. However, not all programming languages are created equal, and some are better suited for certain use cases. When writing a language virtual machine, maximum performance is desired, which typically requires a systems programming language such as C. If the implementers do not want to adopt such a language, they can transpile a chosen language to the performant one. The OpenSmalltalk-VM[3, 6] is based on this approach. The VM is developed in Slang, i.e. a subset of Smalltalk-80[4], which gets transpiled to C.

The advantage of developing the OpenSmalltalk-VM in Smalltalk is, that we can *simulate* the VM in the Smalltalk environment. This approach allows developers to use the Smalltalk debug tooling and the short feedback cycle in Smalltalk to be able to change the VM code during its execution. Simulation enables developers to develop new features without compiling the native VM once, as long as the simulation is fast enough. If the simulation's performance is not enough anymore or the feature is completed, developers have to shift to the compiled VM. When transpiling from Slang to C and when compiling the resulting code, developers are confronted with the limitations of the code generation. A mismatch in expressiveness exists between the two languages. While Smalltalk is object-oriented, C does not support this paradigm with concepts such as classes or polymorphic methods. When developing only using the simulator, developers can build software that cannot be translated as is.

Recently, we encountered such a case while writing a new garbage collector for the OpenSmalltalk-VM. We developed the collector only using the simulator and used a design pattern[2] the generator cannot translate. The new garbage collector acts differently compared with the existing collector, resulting in us wanting to use the existing collector in some special cases. In an object-oriented language, the encapsulation of different, interchangeable algorithms can be implemented using the strategy pattern[2, pp. 315–323]. However, the strategy pattern requires a common interface and different classes with concrete implementations of the interface. As long as only classes with distinctly named methods get generated, the pattern can be used. However, using multiple garbage collection strategies in the same binary requires to write all involved strategies into the C code. In an object-oriented language, multiple implementations on different classes pose no problem, but C does not allow multiple functions with the same declaration, i.e., signature or name. As Slang currently does not provide a production-ready way to bypass this limitation, we are not able to express this pattern easily.

As a part of our endeavor to make Slang easier to work with, we want to support a certain amount of polymorphism: static polymorphism. By static polymorphism, we mean that we can define which class' methods get generated at transpile time. This feature brings the mental model of the VM developers closer to the one in Smalltalk, instead of having to solve the problem as they would in C, which is quite different from Smalltalk best practices[5]. Additionally, static polymorphism improves modularity, while being pragmatic. In Slang we can use the strategy pattern. We do not have to replicate code somehow to use the existing collector. We also do not want to build an object mechanism (vtables) in C, because of the additional complexity and possible performance impact of always using indirections.

The paper is structured as follows: first, in section 2, we give background to the OpenSmalltalk-VM, how Slang works, and our motivation. Afterward, in section 3, we describe the extension. In section 4, we give an experience report and shortcomings we experienced. Finally, in section 5, we conclude the paper and point out further development for Slang.

2 Background and Motivation

In this section we describe the OpenSmalltalk-VM and its relation with Slang. Afterward, we outline Slang and the translation process to C. Finally, we explain our motivation to extend Slang by a new mechanism.

2.1 The OpenSmalltalk-VM and Slang

The OpenSmalltalk-VM[1] is the virtual machine for modern Smalltalk implementations, such as Squeak[2]. The OpenSmalltalk-VM is developed in Slang, a subset of Smalltalk. Because this subset is directly executable Smalltalk code, the OpenSmalltalk-VM can be simulated in a Smalltalk image[3].

[1]https://github.com/OpenSmalltalk/opensmalltalk-vm

[2]https://squeak.org/

[3]The image contains code for the VM and existing tools for development. The current OpenSmalltalk-VM development image can be found under: http://source.squeak.org/VMMaker/.

The advantages of this approach are (1) the flexibility of always being able to change the VM code with a minimal feedback loop and (2) the Squeak debugging tools are available. For developing the just-in-time compiler, the simulator uses a processor simulator to execute native code. On the downside, the simulation is slow. Therefore, for production use, Slang gets compiled to C. The resulting source code can be compiled with a normal C compiler. Note that the compiled VM is much faster than the simulated VM but significantly more difficult to debug. Therefore, it is desirable to work with the simulated VM as long as possible.

2.2 How Slang Works

Slang is a subset of Smalltalk that can be translated to C. The transpilation is guided using implementations of predefined class methods and pragmas[1] (annotations in methods). For example, a common task performed by these mechanisms is defining the C type of a variable. With the combination of these two mechanisms, the code generator translates the given classes to C code. As C does not have classes, the generator translates temporary and instance variables to local or global C variables, class variables to define directives, and methods to C functions. Additionally, classes can be translated to structs if marked accordingly. Method selectors and instance-variable names must be unique. C does not allow multiple declarations with the same name. These restrictions result in Slang being very close to C[7].

The following simplified code shows the process of translating Slang methods to C:

```
CCodeGenerator>>generate: classList
  classList do: [:class |
    self addMethodsFrom: class.
    self declareCVarsIn: class].
  self prepareMethods.
  self inline.
  self generateCFiles
```

The code generator gets a list of all classes to translate and iterates over the list. For every element the generator parses all methods and adds them to the methods dictionary. This dictionary stores the methods of classes. Consequently, when the generator finds a method with a previously added selector, the generator throws an error because it cannot generate unique functions as C requires. After all methods of a class are successfully added, the generator calls the declareCVarsIn: method of the class. In this method, the class can declare variables and execute other calls on the generator, too.

After the generator adds all methods, the generator prepares the methods for further steps. Most noticeable for our use case is that the generator traverses every method's abstract syntax tree (AST). Afterward, the generator inlines methods on the `Slang` level[4]. Finally, the generated AST can be translated to C and written to source files. These files can be compiled using a normal C compiler[5].

2.3 Motivation

In a recent project, we implemented a new garbage collector. The new collector has different trade-offs than the existing one. Consequently, in some cases, such as making a snapshot[6] of the running system, we want to use the existing collector. A common object-oriented way to implement different interchangeable algorithms is the strategy pattern[2]. The strategy pattern requires a common interface for all concrete strategies.

The OpenSmalltalk-VM's code is mostly comprised of singleton classes, such as the interpreter or memory manager. These classes use the strategy pattern and inheritance to represent different configurations. For example, the memory manager has a subclass that handles 32-bit systems and one that handles 64-bit systems. Using these language constructs in simulation causes no problem as they are native to Smalltalk. Until now, either one bit-flavor was generated or the other, never both. Therefore, no problems occurred in the generated code, as only one version was generated.

In our case, however, we require both garbage-collection strategies to be generated. When trying to translate both classes, at least the methods from the common interface cause an error. For example, both collectors have a `collect` method, which is different depending on the collection logic. They have the same name and therefore cannot be translated this way.

To solve this problem, the idea of static polymorphism existed in the OpenSmalltalk-VM. We took the idea and extended it to be usable for our case of the strategy pattern. Additionally, our goal was to achieve support for the strategy pattern without a big rewrite of the `Slang` code generation process, because our goal was to implement a garbage collector and not redesign `Slang`.

We had two alternative ideas: (1) we could implement complete polymorphism in the form of vtables. This change would introduce additional complexity, break `Slang`'s inlining as these calls are on function pointers and not known at compile time in this case, and add a performance overhead for always having the extra step of dereferencing the function pointers in the vtable. (2) We could duplicate the code into our new garbage collector and define unique names. We decided against this approach because we now would have multiple instances of the same code we have to keep in sync and it would be more complicated to define the interface, as the method selectors would have to reflect the different strategies. Additionally, we also would have to add extra functions in classes using the collector classes, depending on which methods use which strategy. Duplicating these methods could get out of hand quickly.

Both ideas have costs we want to avoid, therefore we decided to use the already introduced approach and extend `Slang`'s idea of static polymorphism to handle the deduplication of function names.

[4]<inline:> is one example of a pragma controlling the translation process. Using <inline:> the programmer can define if the method should be inlined into another, meaning in the C code happens no function call, but the inlined method got inserted instead of the function call.

[5]Even if the generator can generate C files, these files do not necessarily compile. One occasional error is the generator does not write out a function, which results in a compiler error.

[6]A snapshot saves the current state of the heap to a file. Afterward, the execution of a Smalltalk image can continue from the point in time when the snapshot was made.

3 Extension

We want static polymorphism to use the strategy pattern for multiple garbage collection strategies in the generated C code. This section describes the two different mechanisms we found to be necessary to support our use case. First, we describe the direct use of polymorphism. Then, we show a case where this mechanism is not enough to express what we want and finally our novel extension of `Slang`.

3.1 Direct Static Polymorphism

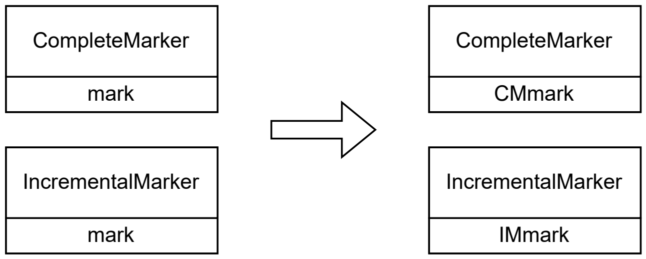


Figure 1. The left side shows the UML notion of the classes with equal method selectors. The right side shows how `Slang` resolves these methods internally

The direct static polymorphism tackles the problem of multiple classes having methods with the same selectors. When calling `declareCVarsIn:`, a call to the generator removes duplicate methods and adds them to the methods pool with a unique prefix. The class calculates which methods to remove by being manually given all different classes in the class hierarchy which conflicts with itself. Then, the class prefixes these methods with a unique prefix, which we currently generate from the class name. Figure 1 shows how `Slang` resolves such a method's selector.

The first step of prefixing the methods resolves the problem of the same selectors, but code using these methods needs to know which concrete method should be called. This is resolved by declaring the class to which an instance variable should be resolved. From now on, we will call this class the *type* of the instance variable. Declaring the type happens in `declareCVarsIn:`. In this method we define which instance variable should be resolved to which concrete class and in which scope. For example, we declare the variable `garbageCollector` in the memory manager to be the new garbage collector, and this should only apply to the memory manager. Should other classes have an instance variable with the same name, they will not be influenced by the declaration.

During processing all functions, the generator traverses the AST and resolves all calls to declared instance variables to the correct type. Additionally, calls to `self`, i.e., the object having polymorphic methods, are resolved automatically to the class itself. Therefore, the generator can resolve the correctly prefixed method and write it into the C code.

3.2 Limitations of the Direct Approach

The direct approach works fine as long as we only look at direct calls. But further down the call chain, the approach is insufficient. Imagine a class A which has static polymorphism for an instance variable `var` of class B. Consequently, normal calls to `var` are resolved to methods of class B. Class A has a variable `specialVar`, which gets resolved to C, too. When A calls a method on `specialVar`, the call gets resolved to C. The problem occurs when C calls A again. A's default case is still B, meaning ordinary methods of A call methods as B defines them. Calling B's method can invalidate assumptions made by the code in C.

3.3 Spreading Static Polymorphism

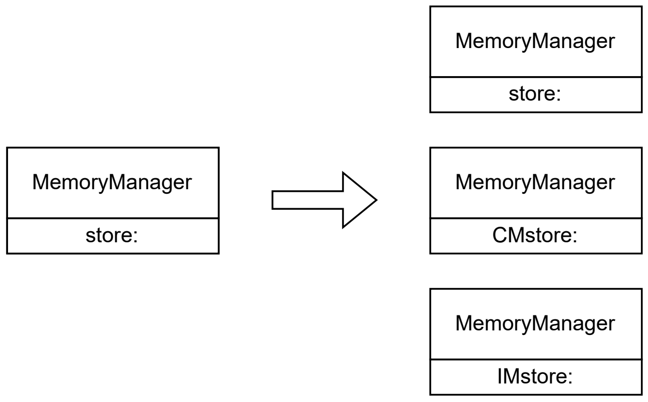


Figure 2. On the left side is the class MemoryManager with one method which uses an instance variable that needs polymorphism. The class resolves the polymorphism for the CompleteMarker (CM prefix) and the IncrementalMarker (IM prefix).

We want to keep the type of a method over multiple calls, even if methods called in between can be used in the context of multiple polymorphic classes. For example, the store method of the memory manager can be called by the existing marker and the new marker and should act accordingly.

As `Slang` internally does not use a call graph, but only looks at direct calls from methods while traversing the ASTs, we propagate types through the methods itself. This means, the method encodes which type its calls should be resolved to. For classes, we declare in the `declareCVarsIn:` method that instance variables are one of n given types, with one default type. The code generator then generates $n+1$ variants for all methods containing the instance variable. Additionally, the generator calculates the transitive closure of the methods using the instance variable. The generator creates the $n+1$ variants for these methods too, as the type gets only spread through polymorphic methods, and the methods from the transitive closure could interrupt the passing of polymorphic type information otherwise. n of these variants are prefixed for the n given types. Therefore, all calls in these methods are resolved to the given type. Additionally, the generator adds one unprefixed method for the default variant. Consequently, calls in this method are resolved to the default type. One example of this method generation can be seen in Figure 2.

We added the capability to resolve only a given subset of methods to the given types. This is necessary because some special classes get extra attention from `Slang`, and extending `Slang` to fully support these classes would have required a high engineering effort we did not want to make at this point. This decision causes problems, too, which we will discuss in section 4.

Determining which variant of a method to call happens through one of multiple mechanisms. First, if external code calls the method without any type information, the code calls the variant without any prefix, which is the default variant. We added this case to make using the extension more ergonomic, by not having to declare the default case in all classes. We also assume calls to the non-default cause are an exception and thus should be declared explicitly.

Secondly, some classes get exclusively used by one strategy. Consequently, we added the capability to associate classes with a strategy. The association gets declared in `declareCVarsIn:`, too. Calls from associated classes get automatically resolved to their associated strategy, meaning the method with the type of the associated class. For example, if a class associated with the `CompleteMarker` calls `store:` on the `MemoryManager`, the internal `CMstore:` method will be used.

Lastly, we added three pragmas to resolve special cases on a method level:

1. We can resolve a method to a specific type. Methods are assuming the existing logic, calling other methods that use static polymorphism. If these methods use other calls, we do not want to change, we can just resolve certain methods. For example:

```
<staticResolveMethod: 'setIsMarkedOf:to:'
                  to: #StopTheWorldGC>
```

is a pragma that sets the type of `setIsMarkedOf:to:` to `StopTheWorldGC`.

2. We can define the type all calls in a method should be resolved for. This approach gets used by methods in classes without polymorphism, calling always the same strategy. It is similar to associating classes with a strategy, only on a method level. For example:

   ```
   <declTypeForPolymorphism: #StopTheWorldGC>
   ```

3. We can define a type for a receiver in the method. This last approach can be used as an alternative to approach (2) if all calls to be resolved are to the same receiver. For example:

   ```
   <staticResolveReceiver: 'objectMemory gc'
                       to: #StopTheWorldGC>
   ```

   resolves all calls on `objectMemory gc` in the current method to `StopTheWorldGC`.

4 Experience Report and Open Challenges

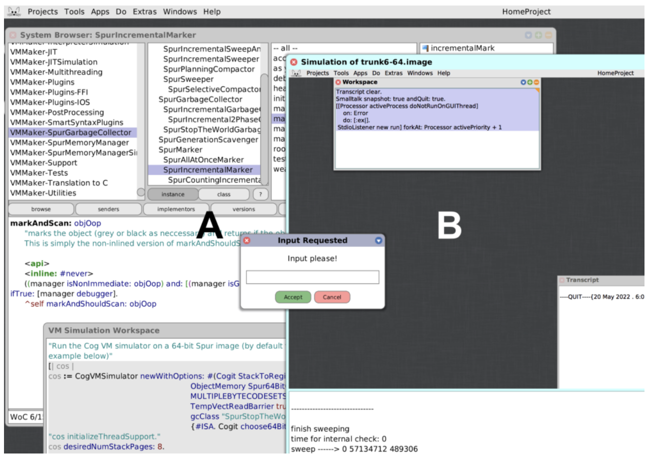
We got the initial implementation of the polymorphism to work in a week of coding. This undertaking took one person, who used the code generator but did not know its implementation, about 40 hours. We extended the previously existing hook for simple static polymorphism by the in this paper described approach. Using our approach was straightforward, as we designed it for our use case. We fiddled long enough with the approach and added the described pragmas as a workaround until the extension was able to express the semantics we desired.

Afterward, we could use the OpenSmalltalk-VM both with the new and the existing collector in the same binary. The first drawback was the renamed methods in the C code. During debugging the compiled VM, we rely heavily on helper methods defined in the code. Some of these methods changed their declaration to a prefixed one. This change is not too confusing, but sometimes unexpected when a helper function just defined in Smalltalk now has a different name in the debugger. In particular, with the extension, inexperienced developers might be surprised and assume that their code was not generated. Therefore, the renaming makes debugging more difficult.

Months after we got the first version of the garbage collector to run, we extended the collector to work with an additional component of the VM. At this point, we encountered subtle problems with our approach. Because the type information gets only spread by polymorphic methods, a method in the new component not being declared as such stops the correct spread of the type information. Consequently, code uses the existing logic, thereby calling code with the default logic, which is the new one. Additionally, an error also occurred the other way around. Code using the new logic called code with a type declaration for the existing logic. Swapping between the logic caused an error here. These errors do not trigger an error in `Slang`. The generated code compiles, but consequently intricate errors can enter the code base, which we experienced.

But why not warn about such transitions? The generator writes other warnings to the transcript[7]. In the case of one logic calling a method hardwired to another type, we could warn programmers and hope that they take the warning seriously. Transitions caused by entering methods without polymorphism and unwantedly calling the default type are more difficult to detect, as we would have to construct an analysis on the call-tree, to detect such transitions. The analysis would be complex and could generate false positives. For practical use, we used an external tool to keep the implementation minimal[8].

For checking transitions between types, we queried the call graph enriched with metadata about our types. We can easily generate a call graph from the ASTs the generator holds during the translation of `Slang` to C. We export this call graph enriched with meta information about potential types, a call got resolved to, to a CSV file which in term we import into Neo4j[9], a graph database. In this database, we can easily query paths from one resolution of a type to another one. Using queries that identified transitions from one type to another, we fixed multiple issues with the type declarations. Additionally, the presence of a query language for the graph enables us to quickly change our question in comparison to implementing such logic in Smalltalk.



```
markAndScan: objOop
    "marks the object (grey or black as neccessar
    This is simply the non-inlined version of markAndShouldS

    <api>
    <inline: #never>
    ((manager isNonImmediate: objOop) and: [(manager isG
ifTrue: [manager debugger].
    ^self markAndShouldScan: objOop
```

```
"Run the Cog VM simulator on a 64-bit Spur image (by default
example below)"
[| cos |
cos := CogVMSimulator newWithOptions: #(Cogit StackToRegi
                                        ObjectMemory Spur64Bit
                                        MULTIPLEBYTECODESETS
                                        TempVectReadBarrier tru
                                        gcClass "SpurStopTheWo
                                        {#ISA. Cogit choose64Bit
"cos initializeThreadSupport."
cos desiredNumStackPages: 8.
```

```
Transcript clear.
Smalltalk snapshot: true andQuit: true.
[[Processor activeProcess doNotRunOnGUIThread]
   on: Error
   do: [:ex|].
 StdioListener new run] forkAt: Processor activePriority + 1
```

```
----QUIT----{20 May 2022 . 6:0
```

```
------------------------------

finish sweeping
time for internal check: 0
sweep ------> 0 57134712 489306
```

Figure 3. In the VMMaker environment (A), the simulator (B) simulates a Smalltalk image using the VM under development. Developers can inspect the state of the running VM and even change the VM's code during execution, thanks to `Slang`.

[7]Writing to the transcript is the Smalltalk equivalent to writing to stdout or stderr.
[8]Additionally, we used the tool for querying other information about the code base.
[9]https://neo4j.com/

Finally, we want to comment on the state of `Slang` as a whole. `Slang` gets the job done, enabling the hybrid development of the OpenSmalltalk-VM in a Smalltalk image and compiling the code to a much more performant native application. The major advantage of using `Slang` is the short feedback cycles Smalltalk brings and the tooling (see Figure 3). On the downside, `Slang` is not intuitive to use. Errors such as a method not being generated to C code or a method not being inlined, although it should be, occur regularly. Furthermore, just because the generator generates C files, they do not necessarily compile. The generated C code is generally correct on a syntactic level, but often not all necessary methods are generated. Consequently, developers have to figure out the problem, thereby having to bridge the abstraction gap between a C compiler error and what the generator did or did not, and then they have to figure out how to convince `Slang` to generate the desired outcome. Another point is, that the generator already has a similar solution to a type problem: type inference for the types of the generated C functions. This feature in its use and dangers are similar to ours: great if they work, but quite a pain to debug if not. Once, the type inference generated a signed type signature for a function needing an unsigned signature. This error spreads an erroneous state to later execution, which was difficult to trace back. In summary, `Slang` is a tool that works, but this tool is sometimes brittle and could break easily even without adding a complex feature. Our extension works, but it makes `Slang` even more error-prone.

5 Conclusion

In this paper, we described our changes to `Slang` to bring the mental model between Smalltalk code, for use with the simulator, and C code, for production use, closer together. We described the high-level implementation of direct and spreading static polymorphism. Direct polymorphism renames the methods directly when the generator adds them before other classes can add conflicting methods. Spreading polymorphism allows methods of one class to be used by multiple strategies and not lose track of the type of the calling method. Together these two mechanisms allowed us to use the strategy pattern, which reduces the overhead for developers, who do not need to find another abstraction, that fits `Slang` and C. On the downside, our implementation can cause subtle bugs when (1) transitioning from methods with polymorphism to methods without polymorphism or (2) when calling a method of another type than the calling method's type.

As future work, `Slang` should receive a redesign to make it easier to debug, stable, and expressive:

1. We would wish for better explorability of why the generator took a certain decision, such as not generating a method. This error is far too common and sometimes requires intricate knowledge of the generator's implementation to understand.
2. The generator should be able to detect if the generated C code will compile, or at least not compilable C code should be an exception. The generated C code is syntactically correct, but sometimes the generator transpiles `Slang` and functions are missing. The generator should have knowledge of the present C code on the platform and which functions it has to generate. Otherwise, the generator should inform the developer and help him understand what went wrong. Possibly, idea (1) could help.
3. First class support for static polymorphism. It should be possible to mark classes as polymorphic and the generator figures out the rest. This would require a bigger redesign of the generator but would make it easier to use the polymorphism and because polymorphism would be applied to all classes, no bugs from transitioning to a method without polymorphism would occur anymore.

Overall, `Slang` and the tooling around it provide a productive environment for developing the OpenSmalltalk-VM. Although `Slang` has some rough edges, we are willing to endure possible quirks and shortcomings for the superb simulation-based VM tools for debugging and exploration.

References

[1] Stéphane Ducasse, Eliot Miranda, and Alain Plantec. 2016. Pragmas: Literal messages as powerful method annotations. In *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies*. 1–9. https://doi.org/10.1145/2991041.2991050

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional.

[3] Adele Goldberg and David Robson. 1983. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc.

[4] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* 318–326. https://doi.org/10.1145/263698.263754

[5] Edward J Klimas, Suzanne Skublics, and David A Thomas. 1995. *Smalltalk with style.* Prentice-Hall, Inc.

[6] Eliot Miranda. 2011. The cog smalltalk virtual machine. *Proceedings of VMIL* 2011 (2011).

[7] Eliot Miranda, Clément Béra, Elisa Gonzalez Boix, and Dan Ingalls. 2018. Two decades of smalltalk VM development: live VM development through simulation tools. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages.* 57–66. https://doi.org/10.1145/3281287.3281295

Received 2023-07-23; accepted 2023-08-28