Illicium

A modular transpilation toolchain from Pharo to C

Pierre Misse-Chanabier
Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL, France
pierre.misse-chanabier@inria.fr

Vincent Aranega
Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRIStAL, France
vincent.aranega@inria.fr

Guillermo Polito
CNRS - UMR 9189 - CRIStAL, Univ. Lille, Centrale Lille, Inria, France
guillermo.polito@inria.fr

Stéphane Ducasse
Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL, France
stephane.ducasse@inria.fr

Abstract

The Pharo programming language runs on the OpenSmalltalkVM. This Virtual Machine (VM) is mainly written in Slang, a subset of the Smalltalk language dedicated to VM development. Slang is transpiled to C using the Slang-to-C transpiler. The generated C is then compiled to produce the VM executable binary code.

Slang is a powerful dialect for generating C because it benefits from the tools of the Smalltalk environment, including a simulator that runs and debugs the VM. However, the Slang-to-C transpiler is often too permissive. For example, the Slang-to-C transpiler generates invalid C code from some Smalltalk concepts it does not support. This makes the Slang code hard to debug as the errors are caught very late during the development process, which is worsen by the loss of the mapping between the generated C code and Slang. The Slang-to-C transpiler is also hard to extend or adapt to modify part of the translation process.

In this paper we present Illicium, a new modular transpilation toolchain based on a subset of Pharo targeting C through AST transformations. This toolchain translates the Pharo AST into a C AST to generate C code. Using ASTs as source and target artifacts enables analysis, modification and validation at different levels during the translation process. The main translator is split into smaller and replaceable translators to increase modularity. Illicium also allows the possibility to introduce new translators and to chain them together, increasing reusability. To evaluate our approach, we show with a use case how to extend the transpilation process with a translation that requires changes not considered in the original C AST.

Keywords Transpilation, Pharo, C language, Tools

1 Introduction

Smalltalk code is traditionally compiled to bytecode, which is interpreted by a virtual machine (VM) [2, 6]. The OpenSmalltalkVM is the current virtual machine used by many Smalltalk dialects, such as Squeak, Pharo and Newspeak. The OpenSmalltalkVM is mostly developed in Slang. Only platform dependant code is directly developed in C. Slang is described as a subset of Smalltalk that is trans-compiled (or transpiled) to C [7]. Developing in Slang enables developer to use Smalltalk live programming features and simulation of the VM [1, 8, 10]

Slang. The Slang-to-C transpiler uses the Smalltalk abstract syntax tree (AST), transforms it, and writes the corresponding C code in a file. We observe several problems.

- Slang does not have the same semantics as Smalltalk. However, since Slang is described as a subset of Smalltalk, and developed in a Smalltalk environment, it is misleading for newcomers to Smalltalk VM development.
- The Slang-to-C transpiler accepts erroneous Slang codes as input, and translates them into invalid C code without throwing errors. Errors are caught during the compilation of the generated C code making them hard to debug.
- The Slang-to-C transpiler is tightly coupled to the Smalltalk AST to produce code, making it difficult to extend.

While Slang has been successfully applied for twenty years to produce Pharo and Squeak Virtual Machines [8, 10], the work presented in this article is a reflection on how to provide a basis for a modular architecture for a transpiler, improving reusability. To experiment with this idea, we designed, Illicium, a new toolchain transpiling Pharo to C.

Illicium. We created a new transpilation toolchain from Pharo to C that we named Illicium. We use a metamodel approach to describe a subset of the C AST and generate corresponding code. We also use the metamodel to generate some tooling for the C AST, in the form of diverse visitors such as walker and structural analysis visitors. This approach allows us to create tooling around the C AST that is independent of the transpilation process.

Illicium's translator is split in several modular and replaceable components. A main translator dispatches the translation to smaller, specialized translators, each taking care of one of

IWST'19, July, 2019, Cologne, Germany
2019.

the Pharo AST node types. Each of these specialized translators are responsible to accept or reject their input, to allow for early error detection. Moreover, it uses a mechanism to treat messages depending on their receiver's types, rather than only on their selectors.

The paper is structured as follow: We first give an overview of how the Slang-to-C transpiler works, and focus on problems we want to solve in Section 2. We then introduce our solution in the form of the Illicium compilation toolchain in Section 3 and how it solves the problems we outlined in Section 2. To support our claim, we show how to extend our tool in Section 4. Next, we explain some of the semantic gaps between Pharo and C that have to be kept in mind when extending Illicium in Section 5. Section 6 compares our solution with other approaches. Finally Section 7 describes several possible future work before concluding in Section 8.

2 Slang

To respect a long tradition of writing everything in Smalltalk, the Pharo Virtual Machine is developed in Smalltalk itself. Yet, it was decided to translate it to C to produce a reliable and efficient VM [8]. To allow the VM developer to express C code in Smalltalk, Slang, a subset of the Smalltalk language is used [7]. Slang code is translated to C code during a process called source to source compilation, or transpilation. The generated C code is then compiled.

Using Slang in a Smalltalk environment allows for a quicker VM development through live programming and simulation [8, 10]. Slang is actively used for the OpenSmalltalk VM development and more specifically for the development of the operating system independent part of the virtual machine as well as plugins. Two different kinds of plugins are developed: internal and external. Internal plugins are compiled with the VM sources, whereas external plugins are compiled on their own, and dynamically linked to a running VM.

Slang has been used for decades and offers a powerful language to express C concepts. However, Slang induces a huge learning curve and the Slang-to-C transpiler tends to make the generated C code hard to debug, especially for newcomers. We identified three main problems that are error prone for VM plugins developers:

P1 Slang does not have the same semantics as Smalltalk.
P2 The Slang-to-C transpiler does not check for semantic violations.
P3 The Slang-to-C transpiler is hard to extend.

In the following sections, we show how these points are source of mistakes and introduce bugs that should be caught earlier in the development process.

2.1 Semantic mismatch

New Slang users usually come from Smalltalk to understand the underlying system or add a custom plugin for the VM they're using. They often have a Smalltalk background and mindset, rather than a C one.

Slang allows the developer to use the Smalltalk syntax, and most of its concepts,

Slang allows the developer to use almost all Smalltalk syntax and concepts, but it has its own semantics and only a subset is translated (*e.g.,* dynamic arrays are not supported). Therefore, Slang is a subset of Smalltalk from a syntactic point of view, but a different language. The Slang-to-C transpiler does not prevent the use of all unsupported Smalltalk features. These unsupported Smalltalk concepts are translated in erroneous C code, that the C compiler rejects at compilation time. As the Slang-to-C transpiler does not keep the mapping between the Slang code and the generated C code, debugging this kind of errors is often complicated. The Slang-to-C transpiler also inlines significant amount of code, which worsen this problem. Let's do a first plugin in Slang to illustrate one of the issue of the current transpilation.

A first plugin as illustrative example. To create a new plugin, we start by subclassing InterpreterPlugin, which gives us access to plugins properties such as the stack manipulation. We add to this class a class variable (Listing 1), as it is common to do in Smalltalk.

```
1 InterpreterPlugin subclass: #MyFirstPlugin
2   instanceVariableNames: ''
3   classVariableNames: 'AClassVariable'
4   category:''
```

Listing 1. A first plugin.

In Smalltalk semantics, class variables values are assigned in any method in the class, or through accessors. In this example, the class variable is set to 5 in aPluginMethod, a instance side method (Listing 2).

```
1 MyFirstPlugin >> aPluginMethod
2   AClassVariable := 5
```

Listing 2. An instance side method assigning a value to a class variable.

As shown in Listing 3, when translated to C, the class variable is transformed in a #define macro. The value of this macro is the class variable value obtained during the transpilation phase. The assignment is translated simply as a C assignment and stays equivalent to the Slang code.

```
1 #define AClassVariable null
2
3 static sqInt APluginMethod(void){
4   AClassVariable = 5;
5   return 0;
6 }
```

Listing 3. Generated C code for aPluginMethod.

This results in the following compilation error from the C compiler:

error : lvalue required as left operand of assignment

This happens because during the C pre-processing phase macros are replaced with their values. This means that after this phase, AClassVariable = 5; is replaced by null = 5;[1], which is an attempt to assign a value to a value and isn't tolerated by the C compiler. This particular error can be hard to understand when the developer is unaware of the way the Slang-to-C transpiler translates class variables.

This example illustrates a difference between Slang's semantic and Smalltalk's (**P1**), and that the semantic violation is not caught by the transpiler (**P2**).

2.2 Slang's development process

Slang's development process requires more steps than Smalltalk's as shown in Figure 1. In a Smalltalk development process, the written Smalltalk code is directly compiled into executable bytecode (*i.e.,* a Compiled Method) and interpreted by the VM. When developing in Slang, the written code goes through three stages, with different sets of constraints and enforcements.

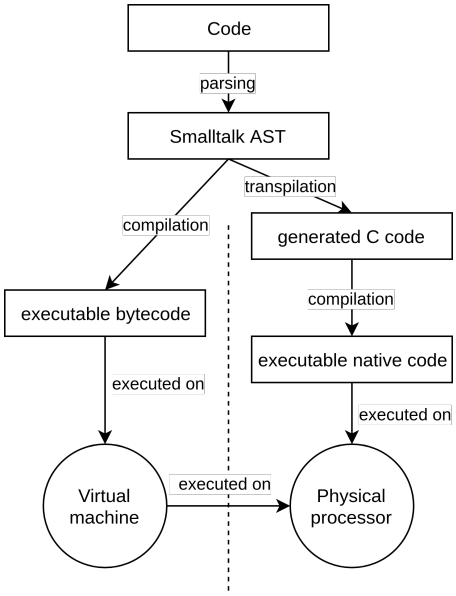

Figure 1. Code compilation process. Slang code is first parsed in a Smalltalk AST, before being compiled as a Smalltalk method. The Slang-to-C transpiler then uses the Smalltalk AST to generate C code. Finally, the C code is compiled.

As Slang is developed in a Smalltalk environment and is a subset of Smalltalk, it first goes through the same steps as regular Smalltalk code and is therefore also compiled to executable bytecode. During this first stage, the Smalltalk compiler first parses the code and creates the method's AST. This compilation process then detects some errors and mistakes such as syntactic errors or undeclared variables.

Next, The Slang-to-C transpiler computes a set of classes to be translated and processes them in the second stage of the transpilation. During the transpilation process, the transpiler handles the AST of input classes and methods, prepares them to be translated and emits the corresponding C code. The Slang-to-C transpiler checks for some unsupported features usage such as the usage of static arrays, and interrupts the translation process with an error if it encounters any of them. It then flushes the emitted C code into a file. Finally, the generated C code is compiled using a traditional C compiler. During this last stage of translation, translation errors, *i.e.,* invalid C code, are caught by the C compiler.

2.3 Surprising translations

In Smalltalk, all interactions between entities, besides assignments, are done via message sending. Therefore, their translations are an important part of the process and they require a special care. Slang introduces a distinction between message sends: it considers special message sends and normal message sends. This distinction is made during the transpilation process and induces a difference in the message send translations.

Special messages. Let us suppose a user overloads the #= binary method for a plugin's class. When sending this newly defined message to self, the overload is ignored, and is translated as the C == operator systematically without warning or error from the transpiler. Special messages are messages that denote a special symbol or a specific function in Slang. To each of these messages an *ad-hoc* translation is applied, regardless of their usage. There are 125 special messages. These messages and their associated translations are statically predefined in the CCodeGenerator » initializeCTranslationDictionary method.

Normal messages. Normal message sends are translated as simple function calls. Messages are translated without verification that the corresponding function will be defined. For example, if a developer sends the #sqrt message to a Boolean instance, it is translated as a function call sqrt(...) even if this message is not understood by Booleans. This shows that a function call is generated from a message send even if the function is not compatible with the receiver's type. This translation generates invalid C code. This error is caught during the generated C code compilation.

Method generation. A plugin must inherit directly or indirectly from the InterpreterPlugin class. The transpiler generates code for methods contained in the plugin class and every one of its superclasses until the InterpreterPlugin class, which is excluded. As a consequence, the #class method is not inherited

[1] null is also a macro defined in the plugin file produced by Slang as 0, this assignment is therefore actually 0=5 after the pre-processing phase

from Object. Class side methods are therefore not usable from Slang code and only instance sides methods are translated. The Slang-to-C transpiler translates unary and keywords methods as function definitions, and inlines binary messages rather than generating their methods. Methods corresponding to a message send to the super pseudo variable are inlined, leaving no traces of inheritance in the generated code. Method redefining a special messages are ignored.

In this part, we saw that inheritance in Slang has a different meaning than the one in Smalltalk (**P1**). We also saw that the Slang-to-C transpiler allows for invalid function calls (**P2**). Finally we explained Slang special messages are stored in a table, making it hard to derive a dedicated translation regarding the context of the message send, limiting extensions (**P3**).

All those reasons make starting to code in Slang a challenge. Since errors are often caught very late in the generation process (*i.e.,* during C code compilation), as the transpiler does not keep the mapping between the C and Slang codes and it heavily uses method inlining, it is very complex to infer from which parts of the code the errors come from. Moreover, the semantic mismatch also makes it hard to reason about the written code, and forces the developer to code in a Smalltalk like language while thinking about the results she wants in C.

3 Illicium: a modular toolchain

To solve the limitations of Slang and the Slang-to-C transpiler, we created a new compilation toolchain from a Limited Pharo to C as a reflection on how to improve the transpilation process with a modular design. The toolchain must implement the three following requirements:

- **R1** The generated C code must be semantically close from what has been expressed in Smalltalk.
- **R2** If features are not supported by the toolchain, the error must be caught as soon as possible.
- **R3** The whole toolchain must be modular to support easy modifications and extensions.

3.1 Structure Overview

An overview of our transpiler is illustrated in Figure 2.

Illicium relies on AST transformations. Using ASTs as source and target artifacts of the translation increases the analysis, validation and transformation potential of the toolchain (**R2** and **R3**). Limited Pharo is used as toolchain input. Limited Pharo defines a subset of Smalltalk that will be translated while trying to keep an equivalent semantic (**R1**). Not all concepts of Pharo are translated, but as the toolchain is designed to be modular, it can be easily extended to add or replace feature support. Once the Limited Pharo code is compiled by the standard Pharo compiler, several analyses are run on the produced AST. These analyses check for unsupported features or modify the AST. Although most analyses and modifications are optional, Illicium requires of the the Pharo AST to be

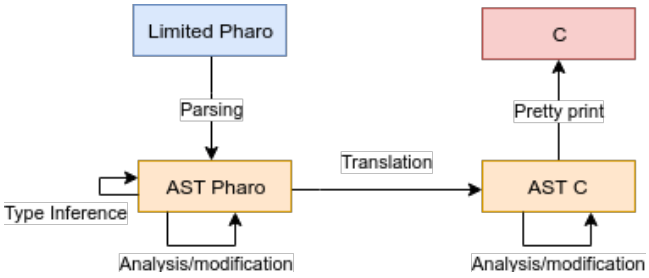

Figure 2. Illicium structure. Limited Pharo code is parsed in an AST. A type inferencer annotates Pharo nodes with concrete types. T he typed AST is then translated in the C AST. Finally, the C AST is pretty-printed.

typed. We currently use the Phineas type inferencer [12] to annotate the AST with concrete Pharo types. Such type information enables us to translate messages depending on the type of the receiver, as well as detect messages misuse early in the transpilation process. After the preliminary passes, the modified and annotated Pharo AST is translated into a C AST using a dedicated translator. Exactly as for the Pharo AST, analysis and modification phases are run in the produced C AST if needed. Finally, the C AST is generated by pretty-printing it in files. The generated C code is then compiled to produce a binary that can be used as a plugin for the VM.

3.2 Generative approach to AST creation

To provide a modular generation toolchain, we decide to heavily rely on AST transformations either to change the AST structure or to pass from an AST to another. In the case of Illicium, the translation is performed between the Pharo AST and the C AST. The Pharo AST is part of the Pharo system and can be directly used as such. To produce the C AST, we used a metamodel approach. We manually created a C AST metamodel using FAMIX [4], and we used code generation to automatically derive validation visitors and a simplified C AST in Pharo. This approach allows us to reuse and easily extend the C AST through the metamodel and to automatically generate common artifacts.

An excerpt of the C AST metamodel is depicted in Figure 3. We see 4 main concepts from the C AST: Statement, Expression, Block and ExpressionStatement. The metamodel states that a Statement is either a Block or an ExpressionStatement. The Block is composed of zero or many other Statements such as Blocks or ExpressionStatements. An ExpressionStatement represents the top node of an Expression.

AST Code generation. From this metamodel, we generate a simplified code for the AST, free from FAMIX dependencies. The class hierarchy generated from the metamodel presented in Figure 3 is shown in Figure 4. The generator generates the necessary accessors for each relationship in each corresponding class. For each container relationship that can contain

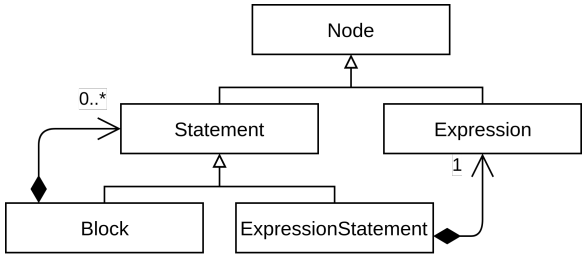

Figure 3. C AST metamodel excerpt. A Block is a Statement containing of a variable number of Statements. An ExpressionStatement is a kind of Statement containing an Expression.

many instances of other classes in the metamodel, it generates an initialize method that creates default, empty, ready to use collections. On top of these basic accessors and initializers, the generator generates class testers (*e.g.,* #isBlock) and a basic visitor mechanism.

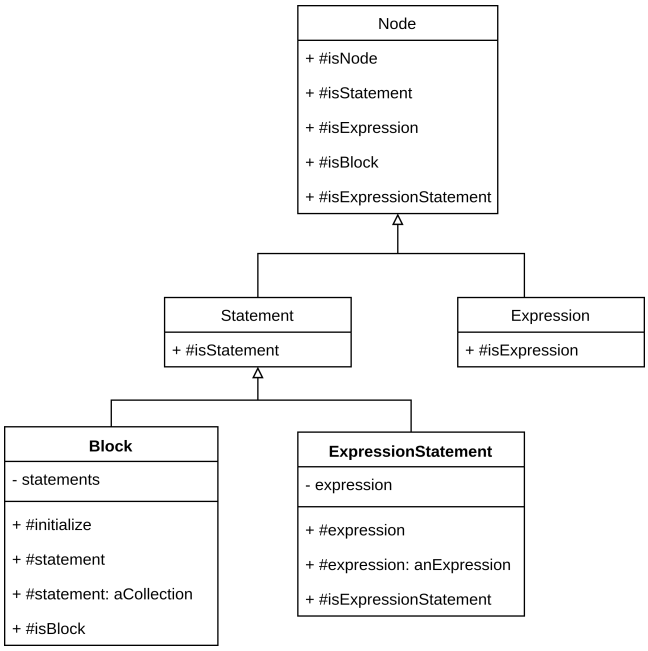

Figure 4. Simplified generated code's model for the C AST. Accessors methods are generated, as well as class tester methods. A default initialize method is generated to initialize an instance variable that contain zero or many entities with a default collection.

Visitors. The generator also generates visitors that are applied to the C AST if required. Figure 5 shows the generated visitor hierarchy from the C AST metamodel. Abstract and Walker are used as base classes and Walker gives an automatic way of navigating through the AST. Printer gives a first display of the AST. StructureValidator and StructuralErrorCount provide a structural validation of the AST code. Their generation relies on the type of each properties in the metamodel. For each property of a class, a special assert is generated in the code of the visitor ensuring that a C AST is structurally correct. StructuralErrorCount uses the StructureValidator, but does not stop on each false assertion and counts the errors instead.

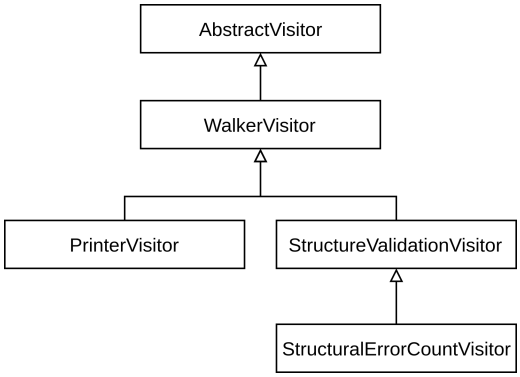

Figure 5. Visitors generated from the metamodel. Abstract and Walker are used as base classes. Printer gives a first display of the AST. StructureValidator and StructuralErrorCount provide a structural validation of the AST code.

The C AST code and tools generated from the metamodel are purely standalone and are independent from the translator and FAMIX. We can therefore develop tools dedicated to the C AST, and reuse them in various contexts where the C AST must be handled. This metamodel approach improves the modularity of the transpilation process and facilitates its extension (**R3**).

3.3 Modular translation

Supporting a modular translation goes with the idea of distributing as much as possible the translation efforts into small interchangeable translation units. In Illicium, we created translation units for the dedicated translation of each Pharo AST node type. These units are responsible for producing from a Pharo AST node the equivalent C AST fragment. They also detect some AST input patterns and adapt their translation, raising errors if required. In this section, we briefly present how the translator is built, show how translation units are written and report errors before focusing on message translation.

3.3.1 Translation structure

The current Illicium translator is built upon a visitor of the Pharo AST. This main translator is composed of smaller translators, specialized to translate only one type of AST node. For each node the main translator visits, it invokes the corresponding translator unit for that kind of node. Each node translator translates the node it gets, and asks the main translator to translate the sub-AST of this node.

An example of the translation process is pictured in Figure 6 for a simple message send in a method. The main translator is visiting a message send #ifTrue: and dispatches this node

to the MessageNodeTranslator. The MessageNodeTranslator takes care of the current selector and asks the main translator to dispatch the receiver and the arguments onto the correct specialized translators. Consequently, the true literal is sent to the LiteralValueNodeTranslator while the empty array is sent to the LiteralArrayNodeTranslator. The process repeats until every node in the method's AST is visited.

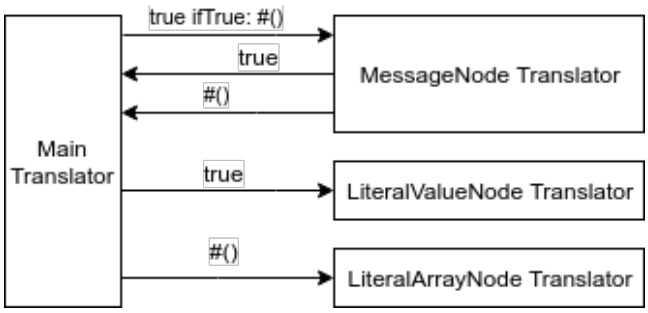

Figure 6. Translation process. The Main Translator dispatches the MessageNode containing the selector #ifTrue: to the MessageNodeTranslator. The MessageNodeTranslator then asks the Main translator to take care of the receiver and the argument of the message, which the Main Translator dispatches onto specialized visitor.

3.3.2 Node translator

Each of these specialized translation units is independent and replaceable. They only need to provide the method: #translateNode:withMainTranslator:. This method must return a fragment of a C AST.

For example, a simplified version of this method for the AssignmentNodeTranslator is shown in the Listing 4. This assignment translator creates a new AST node corresponding to a C AssignmentOperator, and initialize its operands with the AST fragments that are returned by the dedicated translation unit for these nodes.

```
1 AssignmentNodeTranslator >> translateNode:
      anAssignmentNode withMainTranslator: aTranslator
2   ↑AssignmentOperator new
3     leftSideOperand:
4       (anAssignmentNode variable acceptVisitor: aTranslator)
5     rightSideOperand:
6       (anAssignmentNode value acceptVisitor: aTranslator)
```

Listing 4. Simplified translation of an AssignmentNode

3.3.3 Transpilation errors

Each unit translator is responsible for accepting or rejecting the AST node it receives as input. In the example described in Section 2.1, we saw that the Slang-to-C transpiler translates class variable as C macro definition, therefore it does not support class variable assignment. However, the transpiler does not enforce the read only property implied by this translation choice. The assignment AClassVariable := 5. from the code should throw an error to forbid the user to use this unsupported feature. This expression's AST is shown in Figure 7.

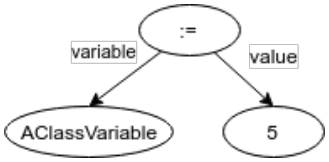

Figure 7. Pharo AST for the expression: AClassVariable := 5. It is represented by an AssignmentNode assigning to the variable AClassVariable the literal value 5.

For the translation of such an AST fragment, three translation units are used: a ClassVariableTranslator, an AssignmentNodeTranslator and a LiteralValueNodeTranslator. The ClassVariableNodeTranslator receives as input the ClassVariableNode representing the class variable AClassVarNode and performs the read-only check from it (Listing 5).

The translator checks if its parent node is an AssignmentNode and if it is, if the variable of the assignment is the node the translator is currently treating. (lines 2-4). The translator throws an error if it identifies that the ClassVariableNode is accessed in write, otherwise, it translates it as an Identifier node instance from the C AST (lines 5-6).

Each node translator is responsible for its own transpilation errors, the semantic enforcement is dedicated to each kind of node (**R2**). Moreover, this approach by small translation units introduces a high modularity as they can be easily modified or replaced (**R3**).

```
1 ClassVariableNodeTranslator >> translateNode:
      aClassVarNode withMainTranslator: aTranslator
2   ( aClassVarNode parent isAssignment
3     and: [ aClassVarNode parent variable = aClassVarNode ])
4     ifTrue: [ self error: ' ClassVariables are read only in Limited
        Pharo.' ].
5   ↑ Identifier new
6     id: aClassVarNode name
```

Listing 5. Class Variables Translation process

3.3.4 A focus on message translation

As we mentioned in Section 2.3, messages are the basis of interactions between objects. We saw that Slang makes the distinction between special message sends and normal message sends. More specifically, the translation architecture behind special messages makes it hard to extend. Moreover, the translation as function calls or inlines for normal messages is ad-hoc and cannot be changed depending on the type of the

receiver. In that regard, we focused on providing a unified and modular approach for message sends translations depending on the receiver's type.

Translation principle. In Illicium, the Pharo AST is required to be annotated with type information. This type information is used by the translator to conduct the code generation. It also allows us to detect invalid message uses during the transpilation process. When the MessageNodeTranslator is called with an input node, it looks for the type of the receiver. Depending on the type of the receiver, it redirects the translation to a special class, a translation class, that represents the corresponding type in the target language. The methods of the translation classes define how each message will be translated. We created translation classes for primitive types. This translation class is instantiated with the receiver's AST as value. The message is dynamically sent to this newly created instance of the translation class using the #perform:withArguments: message. Finally, the translation classes method returns the C AST fragment corresponding to this message send. The #perform:withArguments: message is overridden for those translation classes to have control over the translation and to prevent messages from Object to be inadvertently accepted by the transpiler. Those translation classes implement the methods that are authorized for the translation and return C AST fragments. A message not understood by a translation class is considered invalid, and throws a compilation error.

Taking the same input as the previous example (Figure 6), the #ifTrue: selector is performed on an instance of ILBoolean type which is the translation class for the Boolean type. ILBoolean understands the #ifTrue: selector and knows how to produce the corresponding C AST fragment. This method asks the Main Translator to translate the LiteralNode containing the true literal, and the LiteralArrayNode that contains the empty static array. Finally, the #ifTrue: method returns an If node from the C AST to the main translator.

Advantages. Having translation classes present several advantages:

- Methods and translation type for Smalltalk classes that are available for translation are browsable, modifiable, and extendable. This feature also improves the modularity of the translator (**R3**). As a consequence, several messages with the same selector have different translations depending on the receiver's type.
- Translation classes being used only for the translation process, they also offer a dedicated documentation. This enables transpilation developers to explain the semantic chosen for each types and methods, as well as exposing their limitations (**R2**).
- Statically detectable invalid message errors come up during the translation process rather than in the C compilation process (**R2**).

Users are able to browse and tweak the translation, replace part of the translation process without affecting the target language and create new analyses on the target code using generated visitor base classes. This infrastructure improves the area of the translations and provides a solution to three requirements exposed in Section 2.

4 Extending Illicium: an illustration

To illustrate the modularity and extensibility of Illicium (**R3**), we describe the process of adding a translation for the #to:do: message on SmallInteger.

Analysis. The #to:do: method is a for loop. It is defined on SmallInteger and takes two arguments: the upper bound of the loop and a block representing the sequence of instructions that is executed by each iteration. Each SmallInteger value between the two bounds is passed as argument to the block.

```
1 (2 + 3) to: 10 do: [ :i | "do a computation with i" ]
```

Listing 6. #to:do: call in a smalltalk method

There are three ways of translating this message send in C: a for loop, a while loop or a call to a recursive function. In this example, we decided to implement it as a while loop where the loop variable has the same name as the block's parameter.

The following C fragment (Listing 7) shows the C code we aim to generate. Note that the variable i does not exist outside of the Pharo BlockClosure. We therefore enclose it in another C Block.

```
1 {
2   int i = (2 + 3);
3   while(i <= 10){
4     /* generated code for aBlock */;
5     ++i;
6   }
7 }
```

Listing 7. C code corresponding to #to:do:

Extending the Metamodel. The current C AST does not have a While loop concept, we therefore need to add it. A C While is a Statement containing an Expression which is the condition and a Statement which is its body. This is represented in the metamodel in Figure 8.

After this addition to the metamodel, the metamodel code for the new C AST node and the new version of the visitors are generated.

Extending the Translator. #to:do: is a message and is therefore translated by the MessageNodeTranslator. As we described in the Section 3.3.4, the MessageNodeTranslator send a dynamically created message to an instance of a translation class. Here, as we want to support this message for SmallInteger, we implement the method #to:do: on its translation

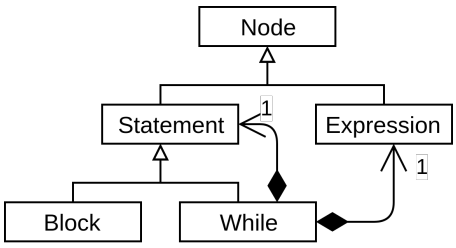

Figure 8. Addition of the While concept to the C AST meta-model. A C while is a Statement, that contains an Expression, the condition, and a Statement representing its body.

class: ILSmallInteger. This method returns the C AST fragment corresponding to the message send.

```
1  ILSmallInteger >> to: stop do: aBlock
2    |iterName|
3    aBlock argument size ~=1
4      ifTrue: [self error: 'Wrong number of arguments.' ]
5      ifFalse: [iterName := (aBlock argument at: 1) accept: visitor].
6    ↑Block new
7        add:
8          (VariableDeclaration new
9                type: self class asCType;
10               id: (Identifier new id: iterName);
11               init: (self value accept: visitor));
12       add:
13         (While new
14            condition:
15               (LesserEqualOperator new
16                  left: (Identifier new id: iterName);
17                  right: (stop accept:visitor));
18            body:
19               (Block new
20                  addAllStatements:(aBlock accept: visitor);
21                  add:
22                     (PreIncrementOperator new
23                        operand: (Identifier new id: iterName));
24                  yourself));
25       yourself.
```

Listing 8. A simplified implementation of the #to:do: for ILSmallInteger.

We see in the Listing 8 a simplified version of the C AST to return. In the first place, we ensure that aBlock takes exactly one argument (line 3-4). We then retrieve the name of this argument to use it as the loop variable name (line 5). The methods finally builds the AST fragment and returns it (lines 6-25). In this Block there is the declarations of a variable initialized with the AST fragment corresponding to the receiver's translation (lines 8-11) and a While statement (line 13-24). This While contains an Expression which is a condition (line 14-17) and a body which is a Block (line 18-23). The body of the While loop contains the AST fragment of aBlock's translation (line 20) and the loop variable incrementation (line 22-23).

Pretty-printing. Finally the last part of the process is the production of the C code pretty-printed from the C AST. The PrettyPrinterVisitor therefore has to implement the #visitWhile: method.

```
1  PrettyPrinterVisitor >> visitWhile: aWhile
2    stream << 'while('.
3    aWhile condition acceptVisitor: self.
4    stream << ')'.
5    aWhile body acceptVisitor: self.
```

Listing 9. Pretty-printing method for the C While concept.

Listing 9 shows how the method is implemented. We observe that only little code is related to the While concept and that the code generation is quickly distributed to the condition and body of the While statement.

5 The challenges of defining a Limited Pharo

C and Smalltalk have different designs, and a different level of abstraction. Mapping the features offered by each language is a difficult task. While defining the supported input, a number of semantic gaps appear. The transpiler developer should be aware of such gaps to provide correct translations or warn the user if they are not supported.

We showed in the Use Case Section (Section 4) an example of how Illicium is extended to enable transpiler developers to add new translations, modify existing ones or remove them. It is the responsibility of the transpiler developer to choose the semantic she gives to a feature in the limited language. Therefore she can decide to overlook a problem, if it does not happen in the context she manipulates. For example, Slang ignores the integer overflow problem and therefore does not pay the run-time cost of checking overflows after operations on integers.

Regardless of the translation decisions, those problems exist, and have to be dealt with when designing a limited version of Pharo to be translated to C. We outline in this section the challenges a transpiler developer faces when translating Pharo code to C code. Both languages offering numerous features, we are focusing on features we think are the most important.

5.1 Primitive data-types

The C language provides four primitive data-types: integer, float, character and boolean. C provides a handful of built-in operators for each of them based on hardware available operators. On the other hand, equivalent to those primitives types are Objects in Pharo and they come with rich libraries allowing their uses and manipulations. It means that although

a few messages have a direct mapping to their C operators counter parts, most of them have to be translated by hand to be available.

Floats and booleans types have a similar semantic on both sides, but integers and characters are problematic.

Integers. In Pharo, Integers are considered infinite because they return a different representations of an Integer when it is needed, in a transparent way for the user. In C, however, the developer has the responsibility to choose the size of the integer to be big enough for its purpose.

When computing an arithmetic operation, if the result is too big for the representation the developer chose, the operation causes an overflow. In Pharo, the SmallInteger's method detects the overflow, and returns a LargeInteger instead. But in C, the result is just different. If the developer does not expect the overflow, the resulting program has an undefined behavior.

Characters. The character type in C is just a byte and does not involve encoding. Character's encoding in C is interpreted as ASCII by many libraries. In Pharo characters are represented by unicode points. The first 128 Unicode code points are the ASCII characters and are encoded on seven bits. Those characters are therefore not a problem. Any Unicode code point's interpretation, when the code point is superior to 128, depends on the library used and will be problematic if they are encoded on more than a byte.

Strings. Although Strings support is not available as a primitive data-type in C, they are as useful as they are used. In C, as an approximation, the convention is to terminate an array of characters by a null character (\0) which is added automatically in the case of a string literal by the compiler.

Pharo represents them using a Pharo Array of Characters. Pharo String's representation is chosen transparently for developers by the system, depending on their content. They are an instance of ByteString if every character it contains is encodable on a byte, or an instance WideString otherwise. They inherit the encoding problems from Characters.

Symbols. Symbols are interned Strings *i.e.,*Strings that are only created once and shared across the program[2] to minimize memory footprint for example[3]. Symbols are a kind of literal in Pharo but they do not have a C native equivalent, although libraries implementing them exist. Similarly to Strings, Pharo supports ByteSymbols and WideSymbols transparently for the user and also inherit the encoding problems from Characters.

5.2 Object Oriented Programming

C does not support Object-Oriented Programming (OOP) features whereas Pharo is basing everything on them. This discrepancy is the most obvious one to notice and is dealt with in a several different ways ranging from ignoring OOP completely, to create a support for OOP features.

For example Slang uses OOP as an organization system on the Smalltalk side and removes any trace OOP features in the generated code. SPiCE[16] on the other hand recreates an equivalent of the Smalltalk environment, and implements OOP features such as dynamic message passing.

This issue is well documented in the literature and is beyond the scope of this paper. Still, Slang offers an interesting static limited inheritance model[10].

Slang's inheritance. In this model, plugin classes actually inherit other plugins instances side methods. Every method that is not redefined, is translated as one of the current plugin. The usage of the super pseudo variable is supported as well and the method call using super as receiver is inlined. This mechanism allows to reuse Slang code, while having no trace of inheritance in the generated code.

5.3 Arrays and collections

An Array is a chunk of contiguous memory of a specified size in C. It can be accessed using the square bracket operator. C requires for all elements to have the same type.

Pharo provides a rich collection library, of which Array is the closest counter part. Pharo's Array is a collection of a fixed size that contain elements of any types as well as the knowledge of its own size. Pharo's collections also provides developers with a wide range of methods allowing to iterate over them without having to know how they are represented whereas user have to describe how to iterate over a C array.

5.4 Block Closures

BlockClosure is an important feature that has several properties in Pharo:

- lexical closure
- deferred execution
- non local returns

A block closure also accepts arguments, as well as defining new temporary variables. It is heavily used, most notably used for iteration and control flow It is syntactically represented by square brackets.

However in C, a block is a simple statement that contains statements surrounded by curly braces and they only support the creation of temporary variables.

5.5 Additional problematic Pharo features

Typing. Although not as visible as the previous features, dynamic typing is an important feature in Pharo.

Message passing is the mechanism allowing all interactions between entities in Smalltalk. It is therefore primordial. The principle of message passing is to send a message to an entity, without caring for its type. All that is required from this receiving entity, is that it understands that message. This method corresponding to the message is looked up in the

[2] In the Smalltalk context, shared across the image

[3] See the Flyweight design pattern

class of the receiver and through the inheritance hierarchy if the it does not know it directly. An error mechanism is triggered if the method is nowhere to be found.

C takes another approach: static typing. The compiler enforce statically, that all functions used, are known, and that types used to call them are correct. C static typing allows the compiler to resolve bindings between a function calls and a function definitions at compile time. It also makes type verification possible during the compilation process, to prevent run-time type errors in the resulting program.

Memory management. In Pharo, memory management is taken care of by the VM, relieving the developer of that responsibility. In Pharo, developers only have to ask for a resource. This resource is handled by a Garbage Collector. When the Garbage Collector notices that the resource is not used anymore, it gives it back to the system.

C on the other hand, only manipulates memory, both static and dynamic. When manipulating dynamic memory in C, developers have to ask the underlying system to allocate a chunk of memory of a specific size. The developers are responsible for its manipulation, and to tell the system when they do not need that chunk of memory anymore, so the system can allocate it to another process. Developers are also able to unsafely access memory directly, which often results in crashes of programs. This access is not restrained and is a common errors when programming in C.

6 Related Work

Slang is a subset of Smalltalk that has been used successfully for two decades to develop the OpenSmalltalkVM [8, 10] (Section 2). The languages delimitation is hard to grasp, resulting in a huge learning curve for newcomers. The Slang-to-C transpiler is too permissive, and accepts in the input code Smalltalk features it does not support. We improve its design with a modular approach, where error management is distributed on small, specialized translations units.

Several other projects successfully translated Smalltalk to C. Among them, Orchard [11] and later SPiCE [16] aim at translating general purpose Smalltalk. They provide a runtime system replacement for the Smalltalk VM as well as replacement classes which implements similar methods as the ones available in the Smalltalk standard class library directly in C. They are called using dynamic look up, to preserve Smalltalk's messaging semantics. We use translation classes at compilation time to guide transpilation in a static manner as well as enabling early errors throw rather than implementing a substitute.

RPython[4] is a toolchain that provides a Restricted Python to C transpiler as well as a support framework to implement dynamic languages. Unlike Slang, RPython offers more features and feels more like writing Python than C [10]. The cost for this abstraction is a less direct mapping to C. This requires long analyses to transform, which results in long compilation times [13] and the generated code is quite unreadable making it difficult to debug.

Using successive ASTs transformations to pass from a formalism to another is equivalent to Model Driven Engineering (MDE) transformation chains [3]. For example, Rodrigues *et al.,* present a MDE approach to generate OpenCL code from UML [15]. This kind of approaches rely on model transformations and models that conforms to metamodel. Our approach could have followed the same principle by directly handling C models that conform to the C AST FAMIX metamodel we built and use model transformation languages. Instead we decided to produce a simpler C AST version to cut FAMIX dependency, and to use Pharo reflexive capabilities as model transformations languages replacement.

Rivera *et al.,* worked upon the ability to chain model transformations [14]. They proposed a modeling language to express how one can link model transformation one to the other. Their system relies on an execution engine that interprets input models describing the transformations chain. Currently, Illicium does not provide an abstraction to express the ASTs transformations chaining, but we consider the production of a Domain Specific Language (DSL) dedicated to this task as future work.

7 Future work

Better way to write AST. As we saw in Section 4, writing an AST by hand is a tedious task. We would like to improve the way we write AST fragments to be able to ease the extension of the current translation classes.

Inlining. Generally speaking, inlining is the process to replace a function call by its code which prevents to pay for a function call at the cost of code size. Slang's Inlining improved the VM performances significantly when it was first introduced [8]. An advantage of transpiling over compiling, is that we rely on the target language's compiler. C compilers have been the subject of research and optimizations and have evolved tremendously. After twenty years, does it still make sense to inline at the transpilation level, rather than allowing the C compilers to take care of it?

Experimenting with the translation. Illicium currently translates classes following a semantic similar to Slang's. We aim at enabling the user to choose how to translate their classes, for example to translate instance variables in a structure rather than as global variables. Illicium can and should also be extended to support higher level concepts with careful attention to the costs they induce. This will enable us to benchmark features, and measure the impact of using one translation rather than another.

Ease plugin testing. Plugins are currently compiled outside of the Pharo environment by a C compiler and their interfaces

[4] https://rpython.readthedocs.io/en/latest/#

have to be created by hand. We would like to have a compilation pipeline that transpiles the plugin, compiles it, and plugs it in the current VM to be tested. A default interface to interact with the plugin could be generated as well.

Composing transformations for an adaptive toolchain. Illicium is designed around ASTs transformations. Transformations introduce validation, refactoring, going from one abstraction level to another... Expressing their composition is currently hardcoded in the system. We will propose a DSL dedicated to the expression of transformations compositions. This language will also allow the toolchain designer to express tasks that could be executed in parallel, increasing the transpilation speed. Such a language invites developers to code smaller composable transformations units that focus on specific independent tasks, increasing reusability, modularity and testing. It will be also interesting to explore how all these small transformations can be enabled or disabled at run time thus producing an adaptive toolchain. For example, the toolchain could adapt its optimisation or validation level regarding patterns identified in the user code.

Multi-level debugging. We showed that the toolchain we designed reduces potential surprises and errors thanks to small transformations units. However, trusting the toolchain does not imply that the user code does not contain bugs. Some bugs can be observed and caught at Smalltalk level, but others are only observed at run time. As future direction, Illicium will maintain a link between the Smalltalk code and the C code to allow the toolchain to report errors into the Smalltalk code (*e.g.,* C syntactical errors). Moreover, conjointly used with C debug format information as DWARF [5], this mapping would allow Illicium to propose a step by step run time debugging directly from Smalltalk.

8 Conclusion

In this paper we present Illicium, a new transpilation toolchain from Pharo to C that is extensible and allows early validations in the transpilation chain instead of waiting until the C compiler fails. This is possible because Illicium uses a C AST model that can be validated and transformed before the C compilation. For this, we use a metamodel approach that describes a subset of the C language and generates the corresponding AST. This metamodel approach also allows us to automatically generate tooling such as structural checkers.

Illicium's translator is split in several replaceable components. A main translator dispatches the translation to smaller, specialized translators, that take care of each of the Pharo AST node types. Each of these specialized translators are responsible to accept or reject their input, to allow for early error detection. Moreover, it uses a mechanism to treat messages depending on their reveiver's types, rather than only on their selectors.

We show how Illicium can be easily extended to be able to transpile new features. Finally, we describe the semantic gaps between Pharo and C that a transpiler developer should keep in mind when extending the Illicium transpilation chain.

Acknowledgments

This work was supported by Ministry of Higher Education and Research, Nord-Pas-de-Calais Regional Council, CPER Nord-Pas-de-Calais/FEDER DATA Advanced data science and technologies 2015-2020. The work is supported by I-Site ERC-Generator Multi project 2018-2022. We gratefully acknowledge the financial support of the Métropole Européenne de Lille.

References

[1] Clément Bera. 2017. *Sista: a Metacircular Architecture for Runtime Optimisation Persistence*. Ph.D. Dissertation. Université de Lille. http://rmod.inria.fr/archives/phd/PhD-2017-Bera.pdf

[2] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. 2009. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland. 333 pages. http://rmod.inria.fr/archives/books/Blac09a-PBE1-2013-07-29.pdf

[3] Jesús Sánchez Cuadrado and Jesús García Molina. 2008. Approaches for Model Transformation Reuse: Factorization and Composition. In *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations (ICMT '08)*. Springer-Verlag, Berlin, Heidelberg, 168–182. https://doi.org/10.1007/978-3-540-69927-9_12

[4] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. 2001. *FAMIX 2.1 — The FAMOOS Information Exchange Model*. Technical Report. University of Bern.

[5] Michael Eager. 2007. *Introduction to the DWARF debugging format*. Technical Report. Eager Consulting. http://www.dwarfstd.org/doc/Debugging%20using%20DWARF.pdf

[6] Adele Goldberg and David Robson. 1983. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass. http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf

[7] Mark Guzdial and Kim Rose. 2001. *Squeak — Open Personal Computing and Multimedia*. Prentice-Hall.

[8] 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 (OOPSLA'97)*. ACM Press, 318–326. https://doi.org/10.1145/263700.263754

[9] 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, VMIL@SPLASH 2018, Boston, MA, USA, November 4, 2018*. 57–66. https://doi.org/10.1145/3281287.3281295

[10] 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*. ACM, 57–66.

[11] C. Nash and William Haebich. 1991. An accidental translator from Smalltalk to ANSI C. *OOPS Messenger* 2, 3 (1991), 12–23. https://doi.org/10.1145/122242.122244

[12] Nicolás Passerini, Pablo Tesone, and Stéphane Ducasse. 2014. An extensible constraint-based type inference algorithm for object-oriented

dynamic languages supporting blocks and generic types. In *International Workshop on Smalltalk Technologies (IWST 14)*.

[13] Armin Rigo and Samuele Pedroni. 2006. PyPy's approach to virtual machine construction. In *Proceedings of the 2006 conference on Dynamic languages symposium*. ACM, New York, NY, USA, 944–953. https://doi.org/10.1145/1176617.1176753

[14] José E Rivera, Daniel Ruiz-Gonzalez, Fernando Lopez-Romero, José Bautista, and Antonio Vallecillo. 2009. Orchestrating ATL model transformations. *Proceedings of MtATL* 9 (2009), 34–46.

[15] Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. 2013. An MDE Approach for Automatic Code Generation from UML/MARTE to OpenCL. *Computing in Science and Engineering* 15, 1 (2013), 46–55. https://doi.org/10.1109/MCSE.2012.35

[16] Kazuki Yasumatsu and Norihisa Doi. 1995. SPiCE: A System for Translating Smalltalk Programs Into a C Environment. *IEEE Trans. Software Eng.* 21, 11 (1995), 902–912. https://doi.org/10.1109/32.473219