Staging & Embedded Domain Specific Languages
Lionel Parreaux
I&C, EPFL

Abstract—Multi-Stage Programming (MSP) is a programming language feature that allows to define high-level abstractions describing generic, highly-parametrized code, while removing these abstractions at runtime during a succession of code generation phases (called stages): the last stage corresponds to the actual program execution. MSP removes these abstractions in a process akin to partial evaluation and program specialization. In particular, MSP can be used to facilitate the efficient implementation of embedded (domain-specific) languages achieving Jones optimality. Staging can also be used to implement custom optimizations as compiler passes, thereby reducing the complexity and cost of operations, based on domain-specific knowledge.

Index Terms—multi-stage programming, staging, embedded domain specific languages, abstraction, optimization

I. INTRODUCTION

Abstraction is an essential component of modern software development. Thanks to the ability to define high-level concepts that are generic and encapsulate details about their varying implementations, we can compose layers over layers of interoperating systems while keeping complexity reasonably low. However, abstraction often has a non-negligible cost in terms of performance, especially when the different layers don’t compose and communicate in efficient ways. Thus, partial evaluation and specialization are commonly used to automatically remove abstractions from the final program implementation, and allow more efficient execution of the code.

Partial Evaluation consists in evaluating the static parts of otherwise dynamic programs. Static program parts often arise from the use of a parametric abstraction with static actual parameters, which is fairly common in modern programming. Partial evaluation is not a trivial optimization, because it can be affected by the current program representation: binding-time improvements like transformation to Continuation-Passing Style (CPS), as well as other optimization passes, can make new static reduction opportunities appear.

Specialization is the adaptation of a generic program part to handle a particular kind of data, about which additional invariants are known, so its result can be computed faster thanks to the new assumptions the compiler can make about that data. Specialization can sometimes be viewed as partial evaluation of type tags in the target program. [7]

A. Multi-Stage Programming

Multi-Stage Programming (MSP) lets the programmer separate their code into a succession of runtime stages – or phases, that will be executed in sequence and result in runtime program generation. The first stages produce code to be interpreted in the next stages. Eventually, the last stage is executed, that is free of abstractions and can therefore compute results faster than the equivalent unstaged program.

Stage distinction is usually based on staging annotations, and is reflected in the type of expressions. The type system can thus enforce that generation of the subsequent stages is correct, and will not produce non-sensical (ill-typed) code.

B. Embedded Domain-Specific Languages

Domain-Specific Languages (DSL) are a useful tool to describe domain-specific computations in a clear and concise way, while being subject to domain-specific optimizations. However, building a new language from the ground-up is a long and non-trivial process. It involves building a lexer/parser, intermediate representations, binding resolution, transformation infrastructure, and target code generation. Fortunately, most of these steps can be removed and the rest simplified by embedding the language inside of an already existing meta-language, a language with a syntax and type system expressive enough to host constructs of the domain-specific language (called, in this context, the object language). Embedding a DSL also offers interesting possibilities of inter-operability with the host language, that were not readily available in a free-standing DSL, as well as access to the general-purpose optimizations offered by the host language.
We usually distinguish shallow embedding, where the DSL constructs map directly to constructs in the host language (thus essentially making the DSL a library written in the host language), from deep embedding, where an intermediate representation based on nodes encoded in the meta-language is produced before being interpreted or compiled and executed. Deeply embedding a DSL allows to perform domain-specific optimizations that the meta-language’s compiler cannot perform itself.

C. Jones Optimality

Jones-optimality is a concept related to self-interpreta- tion, where the meta-language and the object-language are the same. Introduced by Neil D. Jones in the context of partial evaluation of interpreters, it means that the code generated to run the interpreter on a particular program \( p \) is equivalent to the program \( p \) itself. This notion essentially boils down to the absence of any interpretation overhead (at least, after the initial stages of code generation).

D. High-Level Data Structures

High-Level Data Structures are a cornerstone of modern, functional programming. Indeed, modern styles of program- ming tend to adopt a more declarative structure, describing transformations of collections and other high-level data structures, instead of their low-level implementations. For example, instead of writing a sequence of error prone loops and manual collection reconstruction, one could write in Scala

\[
\text{persons}.\text{filter}(\_\_\_.\text{age} > 18).\text{map}(\_\_\_.\text{name})
\]

in order to produce a list containing the names of all the persons contained in source collection \text{persons}, and whose age is over 18.

E. Organization of the paper

The rest of this paper is organized as follows: in Section II, I present the three papers of my candidacy exam. In Section III, I present a short summary of my current work, and my thesis research proposal.

II. SURVEY OF THE SELECTED PAPERS

A. A Gentle Introduction to Multi-stage Programming [6]

This paper presents the basic ideas at the core of the Multi-Stage Programming (MSP) extension of OCaml, named MetaOCaml. It provides detailed examples on how to use it and what to use it for. The main, standard use case is the implementation of an interpreter for a simple embedded domain-specific language, so as to produce Jones-optimal code.

Comparison With Other Tools. Tools such as Template Haskell and C++, based on template metaprogramming, can achieve similar results as MetaOCaml, but they are fundamentally different. [2] Indeed, the only guarantee they offer about the code that is generated is that it is syntax-valid. Syntax-validity means that the code is a valid Abstract Syntax Tree (AST), but it could be a nonsensical one, failing to typecheck. In contrast, MetaOCaml uses its type system to ensure that the generated code is well-typed, which helps in writing correct staged code, and in debugging the source of code generation problems.

Basic Principles. The way MetaOCaml does that is by using the generic ‘a code type, which stores information about the type of the terms that it encodes. For example, \text{int code} represents a “staged” term: in the future stage, it will be of type \text{int}. Then, \text{code expression} are composed together using splicing into quasiquoted expressions.

Quasiquotation. MetaOCaml relies on quasi-quotation to build future stage computations.

• Brackets \(<\text{ and }\>)\ delay execution of code to the next stage. While expression \(<\text{ 2+2 }\>)\ represents the value 4 and will be immediately executed in the current stage, expression \(<\text{ 2+2 }\>)\ denotes a future stage computation, a term that will be generated to be executed in the next stage.

• Splicing (done with operator \text{.~})\ is the action of introducing local variables of \text{code type} into a bigger \text{code expression}. For example, assuming we have a value \text{let a = \text{< 2 >}, we can splice it in \text{< -a + -a >}, and obtain the same result as in the previous paragraph.}

Finally, operator \text{!} is used to actually execute the staged expression to generate code for the next stage.

The power function. It is shown how to implement the staged power function, prototypical example of MSP, which is concerned with partial evaluation of a recursive power function, given its static exponent. The main goal is to be able and compile a call to a power function with a static exponent (e.g., \text{let pow3 = \text{< ! x -> (.~(pow \text{< x>}) .3) as a simple sequence of multiplications}}, without recursive function call overhead (here, \text{let pow3 = fun x -> x**x})).

Staging a simple DSL interpreter. The paper goes on to explain the typical steps undertaken to stage a program, using the example of an interpreter for a simple DSL with arithmetics and recursion, applied to a program computing a factorial. The main problems encountered are:

• removing AST and environment traversal: staging is used to pre-compute things like AST interpretation and environment lookup, since these are static properties;

• binding-time improvements: how MSP can be used to convert a program to CPS, making error handling more efficient because involving less tags;

• controlled inlining: avoiding function calls by splicing the code directly in place of the call.

Criticism. My main criticism about this paper is that it seems to attribute the better error-handling generated code to their CPS transformation, whereas it is clear that what allowed them to remove the option type is their use of exceptions. Indeed, without exceptions they could not have generated such code, and with exceptions alone (without CPS), they could have produced equivalent code.

B. Finally Tagless, Partially Evaluated [1]

This paper deals with the encoding of a higher-order typed object language that allows several possible interpretations of the programs written in that encoding:
• execution: simply run the program by reading its Abstract Syntax Tree (i.e., direct interpretation);
• compilation: convert the AST into OCaml code that can be subsequently compiled and executed at runtime, using MetaOCaml;
• partial evaluation and optimization: the program can also be interpreted in a combination of the two previous interpretations, where terms formally known statically are interpreted on the fly and the rest is compiled (for example, \((\lambda x.x)\text{true} \) can be reduced to \(<\text{true}>\).
• optimization: related to partial evaluation, and performed at the same time in the paper, simple optimizations can be performed “online” (i.e., while reading the program), such as optimizations that exploit the ring and monoid properties of arithmetic operations (e.g., \(x+0 = 0\), \(x+1 = x\), etc.);
• Continuation-Passing Style: perform CPS transformation so as to potentially enable a larger degree of pattern-matching opportunities (this part is not presented in the paper itself for lack of space).

Modularity. Each of these interpretations (which translate to OCaml modules) is defined modularly, separately from the encoding definition (in the form of an OCaml module signature). As we will see later, this is very close to the way the Scala object system works, being able to declare abstract type members in a trait, that are defined in classes inheriting from the trait.

Taglessness. Moreover, none of these interpretations require the use of tags and pattern-matching to ensure correct execution; in fact, the correctness of the encoded program is verified by the meta-language itself, which allows for fast execution and Jones-optimal code generation.

How that is possible is that the main innovation of the paper, is that they use only functions to encode object programs, instead of the more traditional approach of using (Generalized) Algebraic Data Types or Universal Types. This allows them to fully take advantage of the parametric nature of functions in a Hindley-Milner type system, and prevent errors statically, even binding errors, thanks to their use of HOAS.

De Bruijn indices & Higher-Order Abstract Syntax (HOAS). In order to ensure that terms are closed, and can thus be executed safely without the need for pattern-matching and error reporting, the author present two approaches.

De Bruijn indices use an additional environment parameter to every functions, and two functions \(\text{varZ}\) and \(\text{varS}\) to pick variables from this environment, thereby requiring the environment to be of aperticular shape (so that programs composed with \(\text{varZ}\) and \(\text{varS}\) are well-typed). This shape restriction is what allows OCaml’s type system to reject object terms that are not closed. For example, term \(\text{bnot varZ}\) \((\text{b true () , ()})\) is valid, because \(\text{varZ}\) can find the value it is bound to in the head of the environment tuple. However, the following open term does not typecheck: \(\text{bnot varZ}()\).

However, the authors encode bindings in the meta-language itself (HOAS, which they argue is more convenient): instead of passing around an environment nested tuple and storing de-Bruijn indexes into this list as the variable bindings, they use lambda abstractions in the meta-language itself. This approach usually has the problem that non-terms can be constructed by using the meta-language to pattern-match on the bound variable, but the problem is not present here because the encoding abstracts over the interpretation of the object program, and is thus unable to perform such pattern-matchings.

Simplicity and Innovation. As the authors say themselves, their solution is remarkably simple and may even seem straightforward after the fact. However, it solves a problem that had been considered hard or even unsolvable without advanced type system features. That is telling about how simple ideas and encodings that seem straightforward a posteriori can change the way we think about certain problems.

Criticism. I found that the authors repeat too many times the exact same claims of the guarantees of their system (not getting stuck, well-typed, etc.), which are in fact not very advanced or hard to understand. One time saying that the guarantees apply throughought the paper would have been sufficient, in my opinion.

Another problem I had while reading the paper is that, contrary to the previous paper, it gives many function definitions without also giving their type. Type signatures are an extremely valuable tool to understanding implementations quickly. As a result, the reader has to end up doing the job of Hindley-Milner type inference in their head, which can be a bit tedious. Some implementations are also given without much explanations, although the choices they make are not always trivial.

Finally, their implementation of partial evaluation can diverge, making their optimizing compilation a partial function (it can fail). I found that this limitation of their system is not clearly stated as it should be.

C. Optimizing Data Structures in High-Level Programs [4]

This papers can be viewed as an application of the previous paper, used in the context of Scala and the Lightweight Modular Staging framework (LMS [3]), in order to implement an extensible compiler that can notably optimize high-level data structures, utilizing domain-specific knowledge.

Best of Both Worlds. The authors argue that none of compile-time macros or additional compiler passes can alone offer the required power to express the optimizations of high-level data structures. Indeed, macros are limited because they have a limited view of the program that will eventually be constructed, and internal passes have to be implemented as “low-level” IR-to-IR transformations, and suffer from ordering issues. The paper’s approach supposedly brings “the best of both world” by introducing an extensible compiler architecture in Scala that relies on staging and general optimization of the lowered code.

This is usually solved sing an additional dummy universal type.
1Like inferring that function:
2The authors argue that none of compile-time macros or additional compiler passes can alone offer the required power to express the optimizations of high-level data structures. Indeed, macros are limited because they have a limited view of the program that will eventually be constructed, and internal passes have to be implemented as “low-level” IR-to-IR transformations, and suffer from ordering issues. The paper’s approach supposedly brings “the best of both world” by introducing an extensible compiler architecture in Scala that relies on staging and general optimization of the lowered code.
3As the authors say themselves, their solution is remarkably simple and may even seem straightforward after the fact. However, it solves a problem that had been considered hard or even unsolvable without advanced type system features. That is telling about how simple ideas and encodings that seem straightforward a posteriori can change the way we think about certain problems.

3This is usually solved sing an additional dummy universal type.
Speculative Rewriting. The system relies on optimistic assumptions in order to more effectively combine independent optimizations, and rolls back in case the assumptions turn out to be too optimistic. This is essentially useful for optimizing loops by trying to aggressively removing as much unnecessary computations as possible.

Optimizations. The main optimizations presented in the paper are as follows:

- the usual Common Subexpression Elimination (CSE) – using the A-Normal Form (ANF), Dead Code Elimination (DCE), etc.;
- lowering: where high-level constructs are transformed into more low-level ones that can be further optimized and compiled efficiently;
- loop fusion and deforestation: this consists in merging loops iterating on the same data and removing intermediate objects that are constructed and deconstructed immediately, which is common in functional program code;
- closed inheritance to unions: adopting a closed-world assumption, we can reduce inheritance to C-like unions with tags, which allows further optimizations;
- array of struct to struct of array: popular optimization in the data-base community, this can allow more compact data representation and better cache locality, as well as dead code elimination, by allocating all the attributes of an array of object into separate arrays;
- other data format conversions (like object flattening);
- code generation for heterogeneous parallel devices, such as GPU, Hadoop, Spark, etc.

Experimental Results. The speedups demonstrated thanks to these optimizations are tremendous, typically of an order of magnitude. This goes to demonstrate the power of domain-specific optimizations and extensible compilation, which will almost surely become integrated into mainstream programming languages in the future, such great are their benefits in terms of performance.

Criticism. The paper describes in much details all the transformation implementations, and how they affect very simple code examples. However, it does not seem to describe how constraining the approach is to the end user; in particular, because of the need to use Rep types, the pervasive implicit conversions, and the increased compilation times.

III. RESEARCH PROPOSAL

SC (Systems Compiler) is a Scala framework being currently developed at the DATA lab. It aims at helping the design of domain-specific languages embedded in Scala and their optimization, while the embedding is made more seamless, similar to shallow embedding, with the conversion to deep embedding being automatic. The design of SC is based on LMS[3] (for staging), Yin-Yang[2] (shallow vs. deep and automatic deep generation), Forge[5] (automatic deep generation) and Stratego[8] (rewrite systems and quasiquotation).

Quasiquotation for SC. During my semester project, I have been working on implementing quasiquotes for SC, using macros and string interpolation, just like Scala quasiquotes do. However, SC quasiquotes are different from Scala quasiquotes in that they are more semantic than syntactic, and they act on deeply-embedded DSL nodes expressed in a normal form, instead of Scala AST nodes.

There are still many open questions as to how to extend these quasiquotes to make them more powerful (regular expression patterns, higher-order matching, etc.).

Applying SC to SC. One of the goals of SC is to eventually be able to handle general-purpose Scala code, so we can compile SC with SC to apply our optimizations and make SC faster, possibly generating C or LLVM code. This requires solving many challenges, one of the main ones being memory management.

Memory Management. I have been thinking about using an intermediate representation where ownership information (owned, shared or referenced) is explicit, which allows efficient compilation without garbage collection. This IR would be targeted directly by DSL developers, either in a straightforward way, or as the result of advanced analyses and other heuristics (alias and escape analysis, object representation transformations, etc.).

REFERENCES