Computer Languages, Systems & Structures ∎ (∎∎∎∎) ∎∎∎–∎∎∎

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Modular interpreters with implicit context propagation

Pablo Inostroza *, Tijs van der Storm

Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands

ARTICLE INFO

Article history:
Received 3 January 2016
Received in revised form
15 July 2016
Accepted 1 August 2016

ABSTRACT

Modular interpreters are a crucial first step towards component-based language development: instead of writing language interpreters from scratch, they can be assembled from reusable, semantic building blocks. Unfortunately, traditional language interpreters can be hard to extend because different language constructs may require different interpreter signatures. For instance, arithmetic interpreters produce a value without any context information, whereas binding constructs require an additional environment.

In this paper, we present a practical solution to this problem based on implicit context propagation. By structuring denotational-style interpreters as Object Algebras, base interpreters can be retroactively *lifted* into new interpreters that have an extended signature. The additional parameters are implicitly propagated behind the scenes, through the evaluation of the base interpreter.

Interpreter lifting enables a flexible style of modular and extensible language development. The technique works in mainstream object-oriented languages, does not sacrifice type safety or separate compilation, and can be easily automated, for instance using macros in Scala or dynamic proxies in Java. We illustrate implicit context propagation using a modular definition of Featherweight Java and its extension to support side-effects, and an extensible domain-specific language for state machines. We finally investigate the performance overhead of lifting by running the DeltaBlue benchmark program in Javascript on top of a modular implementation of LambdaJS and a dedicated micro-benchmark. The results show that lifting makes interpreters roughly twice as slow because of additional call overhead. Further research is needed to eliminate this performance penalty.

1. Introduction

Component-based language development promises a style of language engineering where languages are constructed by assembling reusable building blocks instead of writing them from scratch. This style is particularly attractive in the context of language-oriented programming (LOP) [44], where the primary software development artifacts are multiple domain-specific languages (DSLs). Having a library of components capturing common language constructs, such as literals, data definitions, statements, expressions, declarations, etc., would make the construction of these DSLs much easier and as a result has the potential to make LOP much more effective.

Object Algebras [35] are a design pattern that supports type-safe extensibility of both abstract syntax and interpretations in mainstream, object-oriented (OO) languages. Using Object Algebras, the abstract syntax of a language fragment is defined

* Correspondence author.
E-mail addresses: pvaldera@cwi.nl, pinoval@acm.org (P. Inostroza), storm@cwi.nl (T. van der Storm).

http://dx.doi.org/10.1016/j.cl.2016.08.001
1477-8424/

using a generic factory interface. Operations are then defined by implementing these interfaces over concrete types representing the semantics. Adding new syntax corresponds to modularly extending the generic interface and any pre-existing operation. New operations can be added by implementing the generic interface with a new concrete type.

Object Algebras can be seen as extensible denotational definitions: factory methods essentially map abstract syntax to semantic denotations (objects). Unfortunately, the extensibility provided by Object Algebras breaks down if the types of denotations are incompatible. For instance, an evaluation component for arithmetic expressions might use a function type $() \rightarrow Val$ as semantic domain, whereas evaluation of binding expressions requires an environment and, hence, might be expressed in terms of the type $Env \rightarrow Val$. In this case, the components cannot be composed, even though they are considered to represent the very same interpretation, namely *evaluation*.

In this paper we resolve such incompatibilities for Object Algebras defined over function types using implicit context propagation. An algebra defined over a function type $T_0 \times \cdots \times T_n \rightarrow U$ is *lifted* to a new algebra over type $T_0 \times \cdots \times T_i \times S \times T_{i+1} \times \cdots \times T_n \rightarrow U$. The new interpreter implicitly propagates the additional context information of type S through the base interpreter, which remains blissfully unaware. As a result, language components do not need to standardize on a single type of denotation, anticipating all possible kinds of context information. Instead, each semantic component can be defined with minimal assumptions about its semantic context requirements.

We show that the technique is quite versatile in combination with host language features such as method overriding, side effects and exception handling, and can be naturally applied to interpretations other than dynamic semantics. Since the technique is so simple, it is also easy to automatically generate liftings using a simple code generator or dynamic proxies [37]. Finally, two case studies concerning a simple DSL and a simplified programming language illustrate the flexibility offered by implicit context propagation in modularizing and extending languages.

The contributions of this paper can be summarized as follows:

- We present *implicit context propagation* as a solution to the problem of modularly adding semantic context parameters to existing interpreters (Section 3).
- We show the versatility of the technique by elaborating on how implicit context propagation is used with delayed evaluation, overriding, mutable context information, exception handling, continuation-passing style, languages with multiple syntactic categories, generic desugaring of language constructs and interpretations other than dynamic semantics (Section 4).
- We present a simple, annotation-based Scala macro to generate boilerplate lifting code automatically and show how lifting can be implemented generically using dynamic proxies in Java (Section 5).
- To illustrate the usefulness of implicit context propagation in a language-oriented programming setting, we provide a case study of extending a simple language for state machines with 3 new kinds of transitions (Section 6).
- The techniques are furthermore illustrated using an extremely modular implementation of Featherweight Java with state [24,11]. This allows us to derive 127 hypothetical variants of the language, out of 7 given language fragments (Section 7).
- Interpreter lifting introduces additional runtime overhead. We investigate this overhead empirically by running the DeltaBlue [13] benchmark in Javascript on top of a modular implementation of LambdaJS (λ_{JS}) [16]. The results show that a single level of lifting makes interpreters roughly twice as slow. Executing a dedicated loop-based micro-benchmark shows that additional call overhead is the prime cause of the slow down (Section 8).

Implicit context propagation using Object Algebras has a number of desirable properties. First, it preserves the extensibility characteristics provided by Object Algebras, without compromising type safety or separate compilation. Second, semantic components can be written in direct style, as opposed to continuation-passing style or monadic style, which makes the technique a good fit for mainstream OO languages. Finally, the lifting technique does not require advanced type system features and can be directly supported in mainstream OO languages with generics, like Java or C#.

This paper extends an earlier paper [25] with additional examples of implicit context propagation in Sections 4.1 and 4.5, the implementation of lifting using dynamic proxies (Section 5.2), a new case study (Section 6), an investigation of the performance overhead of lifting (Section 8), and an expansion of the related work discussion in Section 9.

2. Background

2.1. Problem overview

Table 1 shows two attempts at extending a language consisting of literal expressions with variables in a traditional OO style.[1] The first row contains the base language implementation and the second row shows the extension. The columns represent two styles characterized as “anticipation” and “duplication”, respectively. In each column, the top cell shows the “base” language, containing only literal expressions (Lit). The bottom cell shows the attempt to add variable expressions to the implementation.

[1] All code examples are in Scala [34] (http://www.scala-lang.org). We extensively use Scala **trait**s, which are like interfaces that may also contain method implementations and fields. We also assume an abstract base type for values Val and a sub-type for integer values IntVal; throughout our code examples we occasionally elide some implicit conversions for readability.

Please cite this article as: Inostroza P, van der Storm T. Modular interpreters with implicit context propagation. Computer Languages, Systems & Structures (2016), http://dx.doi.org/10.1016/j.cl.2016.08.001