STRUCTURING LANGUAGES AS OBJECT-ORIENTED LIBRARIES

PABLO INOSTROZA VALDERA

Structuring Languages as Object-Oriented Libraries

Structuring Languages as Object-Oriented Libraries

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. K. I. J. Maex
ten overstaan van een door het College voor Promoties
ingestelde commissie,
in het openbaar te verdedigen in de Agnietenkapel
op donderdag 29 november 2018, te 10.00 uur

door

Pablo Antonio Inostroza Valdera

geboren te Concepción, Chili

Promotiecommissie

Promotores:	Prof. Dr. P. Klint	Universiteit van Amsterdam
	Prof. Dr. T. van der Storm	Rijksuniversiteit Groningen
Overige leden:	Prof. Dr. J.A. Bergstra	Universiteit van Amsterdam
	Prof. Dr. B. Combemale	University of Toulouse
	Dr. C. Grelck	Universiteit van Amsterdam
	Prof. Dr. P.D. Mosses	Swansea University
	Dr. B.C.d.S. Oliveira	The University of Hong Kong
	Prof. Dr. M. de Rijke	Universiteit van Amsterdam

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Centrum Wiskunde & Informatica

The work in this thesis has been carried out at Centrum Wiskunde & Informatica (CWI) under the auspices of the research school IPA (Institute for Programming research and Algorithmics) and has been supported by NWO, in the context of Jacquard Project 638.001.214 “Next Generation Auditing: Data-Assurance as a Service”.

Contents

1 Introduction

Implementing programming languages is a non-trivial endeavor. A typical language processing pipeline consists of several possibly interdependent components. Thus, if an existing programming language is extended with a new construct, its tooling has to be extended accordingly. A case in point could be adding an async/await mechanism to a Java-like language. This would imply that all the processors of this language will need to be extended to consider the new constructs. It is therefore desirable that one could reuse the implementation of existing processors, extending them in a modular fashion.

However, this kind of language extension poses a number of challenges. For instance, a newly introduced feature can demand different context information than the one assumed by the existing processors, or it can interact with the existing features in intricate ways. These scenarios show how complex is to achieve true modular language extensibility. In this thesis we develop techniques to reduce this complexity, contributing to make the implementation of modular languages a reality.

The results of this dissertation contribute to the idea of language-oriented programming [Dmi04; FFF+18; War94], a vision that puts programming languages at the center of the software development practice. Instead of using one general-purpose programming language (GPL), the idea is to represent the diversity of domains that crosscut traditional software development using specialized Domain-Specific Languages (DSL) [Fow10; vDKV00]. This allows programmers to concentrate on the elements of the domain and their interactions instead of the details about their representation.

In language-oriented programming it is essential to have tools and practices that make the development of new languages less costly. In particular, in this thesis we exploit linguistic reuse [Kri01] by means of better modularity. We consider linguistic

reuse as the application of the principles of software reuse to the domain of language development.

According to Krueger [Kru92], "*Software Reuse is the process of creating software systems from existing software rather than building software systems from scratch*". A canonical example of reuse is library-based development. In fact, in McIllroy's seminal paper on software reuse he presents the vision of a library of reusable components [McI68]; software can be thus developed by assembling such components. A software library is a collection of interrelated functions, classes or modules that cover a particular domain or application concern (e.g., accessing the database) and is intended to be reused as is.

The main motivation of this thesis is to provide new mechanisms to implement libraries of language fragments. A *language fragment* is the specification of a language that might or might not be usable on its own, but can be composed with others to form increasingly complex languages. In other words, a language fragment is a module that represents the implementation of part of a language. The essential question that follows is how to define modular and extensible language fragments.

One of the keys of the success of object-oriented programming is its support for extensibility via inheritance [Ald13]. Statically typed object-oriented languages such as Java or C# support static safe incremental extensibility which enables modular development. In this work we posit the question of how to use object-oriented techniques to implement reusable language fragments.

The object algebra pattern [OC12], encoded with standard object-oriented features, is a technique in that direction, as it enables incremental extensibility along two axes that fit the language development scenario: to develop a language we can both extend its syntax and add new semantic interpretations.

In this thesis we exploit the object algebra pattern to develop modular language components that form libraries of language fragments. We focus our attention on two kinds of libraries. On the one hand, we want to develop extensions of an existing GPL using object algebra-based language fragments, in the spirit of "Growing a Language" [Ste98]. On the other hand, we want to assemble several object algebra-based language fragments that can help us to build different languages from scratch, as if these fragments were language LEGO blocks.

The next section provides an overview of the idea of language libraries and component-based language specifications, alongside the background on related techniques for language extensibility and language modularity. Section 1.2 introduces the central technique that we make use of: object algebras, while Section 1.3 describes how we have applied object algebras to develop language libraries, introducing our research questions and how they correspond to the chapters in this thesis. Furthermore, related work that is specific to each research question is also discussed. In Section 1.4 we list our contributions in terms of peer-reviewed publications. Section 1.5 presents

the software artifacts produced in the context of this thesis. Finally Section 1.6 describes the structure of this dissertation.

1.1 Language Libraries

The idea of component-based language development has originated work in different communities. In [TSC+11], for example, the authors coin the concept of "Language as Libraries" and illustrate it with the Racket language. Racket programmers can write language modules that reuse the base-level meta-infrastructure by having access to a well-defined interface to the static semantics. These modules themselves are written in Racket supporting the argument that writing a language is equivalent to writing a library in the base language, instead of more complex approaches such as extensible compilers [NCM03].

Another embodiment of the idea of language libraries is SugarJ [ERK+11], a Java-like language in which libraries are just like regular Java libraries together with syntactic sugar definitions to cater for extensible syntax.

On the more formal side, Bergstra et al. propose an axiomatic algebraic calculus of modules to dissect the essence of composition [BHK90]. More pragmatically, Heering and Klint consider a library of reusable semantic components as an essential element of Language Design Assistants [HK00].

In [CMS+15; CMT14] Churchill et al. propose fundamental constructs (*funcons*), a fixed set of reusable components of semantic specifications defined using I-MSOS [MN09], an improved, modular variant of Structural Operational Semantics. Using funcons as basic building blocks, one can specify arbitrary compositions that provide the specification of complex general-purpose languages.

In this dissertation, a language library is a *library of language fragments written in an object-oriented host language*. The basic mechanism for implementing the idea of language libraries is language composition. It is therefore necessary to establish some basic terminology in this regard.

In algebraic terms, if we have two modular definitions for language fragments L_A and L_B, we want to have a composition operation $\oplus$ such that $L_A \oplus L_B$ gives us L_{A+B}. But, of which nature is this $\oplus$ operation? In order to answer this question and clarify some terminology, Erdweg et al. [EGR12] have characterized five cases of language composition based on the idea of unchanged reuse, i.e., reusing language components without modifying them (known as black-box reuse in traditional Software Engineering terminology): language extension, language restriction, language unification, self-extension (including language embedding), and extension composition. In particular, in this dissertation we are interested in language extension and language embedding.

$$
\begin{array}{rcl}
e & := & \ldots \\
 & | & \texttt{unless}(e, e, e)
\end{array}
$$

Figure 1.1: Abstract syntax of an unless extension to an expression-based language

Language extension. A language L_B is an extension of a base language L_A if L_A can be reused without changes when implementing L_B. Some language-development systems supporting this property are Lisp with macros (such as Racket) or SugarJ. For example, a Racket language definition implementing a state machine language consists of a series of macros. This state machine language extends Racket via its macro mechanism, without altering Racket's implementation.

Language embedding. A language L_B is embedded in a language L_A if a program P_B written in L_B is also a valid L_A program. The converse does not hold. In other words, programs of the embedded language L_B are nothing but calls to an API written in the host language L_A. The Akka framework, for instance, allows programmers to write distributed applications in Scala (or Java) using an actor model. An Akka program is simply a Scala (or Java) program.

These two kinds of composition differ in which kind of abstraction they enable. While language extension provides syntactic abstraction, language embedding reuses the host language mechanism for procedural abstraction. In other words, a language extension allows programmers to use new syntax (with respect to the host language) for expressing specific concepts captured by the extension. A language embedding, on the other hand, uses the abstraction mechanisms of the host language to support the higher-level concepts captured by the embedding, just as a well-designed API.

Next, we review related work on language libraries in the light of the two aforementioned forms of language composition and discuss the challenges for modular language development. As an illustrative artifact, we will use a running example of a simple `unless` extension to an expression-based language, whose abstract syntax can be found in figure 1.1. The `unless` expression extends the syntactic case for expressions e and it has three sub-expressions. The first sub-expression is a condition that in case of being successful leads to the execution of the third sub-expression, or otherwise to the execution of the second sub-expression.

1.1.1 Related Work in Language Extension

Language extension enables customization by means of syntactic abstraction. Already in the 60s, early Lisps provided mechanisms to specialize the language for specific tasks within the language, defining in consequence custom "special forms". Of

these mechanisms, macros have persisted to this day. Macros [McI60] are syntactic transformations that can automate common patterns of use. They are expanded at compile-time, and therefore were regarded as an optimal way of customizing the language. Programmers (acting as de facto metaprogrammers) write macros that capture a specialized pattern, and these macros can be invoked in a program. Lisp together with a set of macros is thus an enriched, extended Lisp.

A contemporary Lisp descendant is the aforementioned Racket language. Racket has been designed to support language extensions inheriting the tradition of using macros as the internal mechanism to extend the base language, though benefitting from several advancements in macro technology such as hygiene and lexical macros. This is complemented with a module system and a rich interface to the internals of the base language.

The following snippet shows the encoding of the `unless` extension as a Racket macro:

```
(define-syntax-rule (unless c e1 e2)
  (if (not c) e2 e1))
```

After defining this macro, we can write expressions such as `(unless (> x y) 0 1)` as if they were part of Racket's built-in syntactic forms. Each occurrence of an `unless` call is statically expanded to an if-expression.

Macros have inspired ideas such as Generative Programming, whose vision consists of configurable components that form system families which can be customized for different domains [CE00]. C++ template-based metaprogramming and Intentional Programming [SCC06] are examples of Generative Programming techniques.

The aforementioned SugarJ also falls in the category of systems for language extension, as it allows programmers to create libraries of syntactic extensions. The syntax of an extension is defined using the syntax definition formalism SDF [HHK+89; Vis97] while the desugaring of an extension into Java-like code is defined using transformations, written in the transformation language Stratego [VB98]. Hence, a reusable language fragment consists of a syntax definition plus a transformation describing how to desugar a program using the extended syntax into plain GPL code.

1.1.2 Related Work in Language Embedding

An embedded language is a domain-specific API encoded in a host language. Programs of the embedded language are calls to this API and, therefore, also valid programs in the host language. Figure 1.2 shows the encoding of the `unless` extension as a Java static method.

The main difference with the previous (macro) example is that in this case there is no syntactic abstraction. In order to model lazy evaluation of the consequent and the alternative we needed to resort to an internal host-language mechanism, in this

```
class Unless {
  public static <T> T unless(boolean c, Supplier<T> e1, Supplier<T> e2) {
    if (c) return e2.get(); else return e1.get();
  }
}
```

Figure 1.2: An encoding of the `unless` extension in Java

case, Java closures represented by the `Supplier` interface. The next expression uses the `unless` extension:

```
Unless.unless(x > y, () → 0, () → 1);
```

Such expressions are just regular Java expressions (no static expansion is performed whatsoever). Since Java does not feature lazy evaluation, the use of the `unless` abstraction needs to consider the accidental complexity of deferring the evaluation, and it is therefore verbose.[1]

A common object-oriented pattern to embed languages are fluent interfaces [Fow10], which attempt to improve the readability of the embedded programs by using method chaining.

In functional programming, there is a long tradition of language embedding. In particular, Hudak focuses the discussion on the modularity of the embeddings. In a series of articles where he presents various language interpreters using Haskell, he refers to the embedded languages as Domain-Specific Embedded Languages (DSEL) [Hud98; Hud96; LHJ95]. Hudak's modular interpreters are extensible in three dimensions, as seen in the definition of the evaluation function `interp`:

```
interp :: Term -> InterpM Value
```

The three components of this function are configurable: `Term` corresponds to the syntax of the language, `Value` corresponds to the domain of values, and `InterpM` is a monad representing the nature of the computations involved in the evaluation. The former two components can be extended by using open unions[2] – also known as extensible union types – while the computations can be extended using monad transformers.

Unfortunately, the introduction of new variants to any of these components implies a full recompilation; programs using the not yet extended components need to be

[1] The discussion about the tradeoff between the syntactic convenience of language extension and the implementation convenience of language embedding goes beyond the extent of this dissertation. Taking this tradeoff into consideration, it is the language engineer who decides to implement a DSL using a syntactic extension or an embedding.

[2] For instance, the open union `OR` allows us to represent the disjoint union of two types `a` and `b` as **data** `OR a b = L a | R b`.

recompiled to take the extensions into account. Hudak's embeddings showcase the complexity of modular language development, the subject of next section.

1.1.3 Challenges of Modular Language Development

Modularity fosters reuse, and reuse has the potential to improve language engineering. In order to objectively assess language modularity solutions, a characterization of the requirements of a modular language implementation is needed.

Since the idea of language embedding is well-known in the context of functional programming languages, let us introduce the modularity problem in that specific context. The gist of the idea of language modularity has been expressed as a problem of extensibility known as the Expression Problem [Wad98]. The Expression Problem poses the question of how to extend a datatype definition along two axes: adding new variants and adding new operations. In its original formulation, Wadler specifies four requirements for possible solutions, while Zenger and Odersky [ZO01] introduce an additional one, listed in the last place:

- *Extensibility in both dimensions*: Both new variants and new operations can be added.
- *Strong static type safety*: An operation cannot be applied to a variant that is not permitted by the static checker.
- *No modification or duplication*: The extended code cannot be modified or duplicated, in other words, it must be reused "as is".
- *Separate compilation and type checking*: Safety checks or compilation steps must not be deferred until linking or runtime. In other words, an extension must be compiled and type checked, only depending on the interface of the code being extended, but not on its implementation.
- *Independent extensibility*: Two independently developed extensions can be composed in order to be used together.

To explain why the Expression Problem is relevant to the language modularity challenges, we first point out that datatype definitions are analogous to definitions of the abstract syntax of a language, and operations are analogous to language processors, such as interpreters, type checkers, compilers, etc. Furthermore, the listed requirements cater for modular definitions: The first four ones require the solution to be incrementally extensible, while the last requirement results in even stronger modularity: independently-developed extensions can also be composed. The related work we discuss below considers mostly the earlier formulation of the expression problem, i.e., without taking into account independent extensibility.

In [Swi08], Swierstra consolidates the ideas of catamorphisms (*folds* over arbitrary data structures) and open unions to provide a solution to the Expression Problem. However, there is no discussion of the composition of interpreters with different

context requirements or "computations", a matter that Hudak and Liang's interpreters address with monad transformers .

Also in the context of functional programming, Hinze introduced the encoding of datatypes as Haskell type classes [Hin06]. Later, Oliveira et al. showed that similar type-class based encodings are extensible and therefore solve the expression problem [dSHL06]. Similar techniques were used in [CKS09] to introduce Tagless Interpreters, presented as a solution to the Expression Problem where Haskell type classes or OCaml modules are used to encode syntax. Different semantics to syntax encoded this way can be provided as many times as desired, accounting for different operations.

In the context of object-oriented programming languages some solutions to the Expression Problems have been presented too. Torgersen [Tor04] discusses 4 solutions of which 3 use advanced features such as recursive F-bounds or wildcards while the last one, being non type-safe, does not comply with all the aforementioned minimal requirements of the Expression Problem.

Polymorphic Embeddings in Scala [HOR+08], on the other hand, are analogous to finally tagless interpreters, where traits are used as typeclasses and the types of the operations performed on the syntax are left abstract using type members. Another recent solution to the Expression Problem in the context of object-oriented languages does not require the language to feature parametric polymorphism, but instead, covariant type refinement of return types [WO16].

In this work we use a solution to the Expression Problem that imposes minimal requirements on the host object-oriented language: Object algebras.

1.2 Object Algebras

Object algebras [dSHL06] were introduced as a simple mechanism to solve the expression problem in object-oriented programming languages, requiring only inheritance and generics (parametric polymorphism). The pattern is based on algebraic specification of abstract datatypes [GH78] and encodes the basic elements of this formulation using an object-oriented embedding.

An object algebra interface corresponds to an algebraic signature, specifying syntactic cases. Consider an arithmetic expression-based language that features literals and addition. This is its object algebra interface, in Java:

```
interface Arith<E> {
    E lit(Integer n);
    E add(E e1, E e2);
}
```

For each syntactic case, we define a method signature. The interface has one type parameter `E` known as the carrier type of the algebra. Implementing this interface as

	Arith	**...**	**Arith + Neg**
Evaluation	`class EvArith implements Arith<Integer> { Integer lit(Integer n) { return n; } Integer add(Integer e1, Integer e2) { return e1 + e2; } }`	⋯	`class EvNeg extends EvArith implements Neg<Integer> { Integer neg(Integer n) { return -n; } }`
⋮	⋮	⋮	⋮
Pretty-printing	`class PPArith implements Arith<String> { String lit(Integer n) { return n.toString(); } String add(String e1, String e2) { return e1 + " + " + e2; } }`	⋯	`class PPNeg extends PPArith implements Neg<String> { String neg(String n) { return "-(" + n + ")"; } }`

Figure 1.3: Extensibility along two dimensions with object algebras

many times as desired over different concrete carrier types accounts for extensibility of operations. For example a class implementing this interface over carrier type `Integer` can serve to represent the "evaluation" operation, while one over carrier type `String`, "pretty-printing", and so on.

On the other hand, extending the object algebra interface with more abstract methods accounts for syntactic extensibility. For instance, a new interface could extend `Arith` by adding the new method definition `neg` that corresponds to a new syntactic case representing negation:

```
interface class Neg<E> extends Arith<E> {
    E neg(E e);
}
```

Figure 1.3 illustrates the different kinds of extensibility supported by object algebras. Along the vertical axis, new operations can be added by providing new implementations of the object algebra interface. The first column shows two implementations of the interface `Arith`: the one on the top row is implemented over carrier type `Integer` representing the operation "evaluation", and the one on the bottom row over carrier type `String`, representing the operation "pretty-printing". This kind of

extension enables reuse of specifications via multiple implementations. For instance, classes `EvArith` and `PPArith` reuse the specification of the `Arith` algebra by implementing the corresponding interface.

Along the horizontal axis, the classes implement the extended object algebra interface that account for the new syntax (in this case, `Neg`). In the right column we see the implementations of the new evaluator and pretty-printer that now consider the "negation" syntactic case, added by the object algebra interface `Neg`. This kind of extension enables reuse of implementations via inheritance. For example, class `EvNeg` reuses the implementation from class `EvArith`, adding only the new behaviour corresponding to the syntactic extension represented by interface `Neg`.

The bottom-right frame showcases both kinds of extensibility relative to the top-left frame, adding a new kind of operation "pretty-printing" and a new syntactic case "negation".

The simple idea of object algebra has been extended with other mechanisms in order to give answers to other requirements, such as independent extensibility [OvdSL+13] or the support for attribute grammars [RBO14]. Also, the pattern has been applied in scenarios such as extensible languages [GPS14] or to "scrap boilerplate" in language processors [ZCO+15]. Besides the application of object algebras in the context of programming languages, Leduc et al. have studied how to adapt the object algebra pattern to support modular extensibility of executable metamodels [LDC+17].

This thesis explores to which extent we can use object-oriented techniques, in particular object algebras, to address some of the aforementioned challenges of language-oriented programming, with a focus on the construction of modular and reusable language components that can serve as the foundation for an ecosystem of language libraries.

1.3 Language Libraries with Object Algebras

Object algebras are a natural candidate to structure modular language components using an object-oriented general-purpose programming language. The simple examples of the preceding section show simple language processors for an arithmetic DSL. In this thesis we want to explore the capability of object algebras to act as supporting technology for language-driven development. In particular, we exploit the object algebra pattern in various ways in order to improve the reuse of OO-based language definitions, leveraging the idea of object-oriented language libraries. This motivation is reflected in our research questions, which we discuss in the following section.

1.3.1 Modular Extension of a General-purpose Programming Language using Object Algebras

Syntactic extensions to a GPL enable notations that adjust more to diverse domains (e.g., asynchronous computations), without abandoning the host language. Recent examples of GPL extension are Scala-Virtualized [RAM+12] and F# computation expressions [PS14]. In both cases, a number of constructs of the host-language syntax can be virtualized in order to accommodate arbitrary semantics, which are specified in the host-language as well.

However, in both cases, the set of syntactic constructs that can be virtualized is fixed. It would be desirable to generalize this to support arbitrary semantics for arbitrary syntactic constructs. In the light of language libraries, this would enable the distribution of arbitrary syntactic and semantic extensions to a GPL as libraries. These libraries would extend a language like Java in a plug-and-play fashion. The object algebra pattern can be applied to provide the semantics for this sort of syntactic extensions. This leads to our first research question.

Research Question 1 (RQ1): How can object algebras facilitate modular language extension of a General-purpose Programming Language?

Results. We have designed *Recaf*, a lightweight tool to modularly extend Java. Recaf uses a syntactic transformation that generically transforms designated method bodies to code that is parametric on an Object Algebra that provides its runtime semantics in Java, enabling thus a form of "virtualization" of the Java syntax. A modular language extension consists of a class that extends the object algebra implementation of the base Java semantics with added methods that specify the semantics of the syntactic extensions. The Recaf system is discussed in Chapter 2.

1.3.2 Tracing Program Transformations

Program transformations are pervasive in language-oriented programming. In fact *Recaf*, the tool we have just introduced as a product of the research associated to RQ1, also relies on a syntactic transformation. A well-known problem in systems that depend on syntactic transformations is how to provide adequate IDE support, for instance, in terms of debuggability or error-marking. Since the high-level abstractions used by programmers go away at the moment of execution since they are transformed into GPL code, it is difficult for the programmers to relate the code that actually runs with the mental model they had in mind when programming.

Various techniques have been proposed to mitigate this, such as *origin tracking* [vDKT93], a mechanism to trace the output of the transformations back to its source, along a chain of transformations; *macro debugging* [CF10]; and *resugaring* [PK14],

a technique to compute reduction steps executed in the target language in terms of the high-level surface syntax.

Traceability, in particular, presents a series of benefits in terms of debuggability (as we can trace errors in the generated code back to its origin), understandability of code (as we can, if needed, look at the generated code in order to understand what the high level code means), etc.

In the context of language libraries, it is desirable that if any code generation takes place, as in the case of *Recaf*, the link between the input and output programs is explicit. Language extensions shipped as libraries would gain in usability if the programmers can relate the extended language code back to the plain GPL code that is generated behind the scenes. The benefits that traceability brings to Language-oriented development, in particular in the spirit of lightweight tools such as Recaf, motivate our second research question.

Research Question 2 (RQ2): How to trace program transformations in a lightweight, language-agnostic manner?

Results. *String origins* are a lightweight, generic and portable technique, inspired by origin tracking, that enables to establish a tracing relation between the input and the output of a textual program transformation, such as the one employed by *Recaf*, by linking each generated character to its origin. The technique works by reimplementing the standard string operations used by transformations, allowing them to keep the links to the original input and thus to propagate these links through the entire transformation pipeline. *String origins* are presented in depth in Chapter 3.

1.3.3 Composing Interpreters with Different Context Requirements

Object algebras are a natural technique to implement modular language processors, in the object-oriented programming landscape. This is posited in [GPS14] where the authors also present a simple annotation-based technique to link the language processors to concrete syntax. By doing so, object algebras support the complete modular definition of external DSLs.

Still there are, however, a number of open challenges and opportunities in applying object algebras to modular language development. If we have modular language definitions we want to make sure that we can compose them in all possible scenarios. If two independently developed language components assume the same requirements in terms of semantic context (also known as semantic entities), for instance, that an environment mapping variable names to values is present, then composition is trivial as the context parameters of both components are compatible. In the case of object algebras, the composition consists of creating a new algebra that extends the two modules that need to be composed.

However, if the context requirements differ, simple composition using object-oriented extension is not possible. If there are two language components A and B, where B requires one more context parameter than A, there are two options. It is possible to develop a new component A' that duplicates all the code in A just to thread the extra context parameter. This alternative implies no reuse whatsoever. The other alternative, in case we want to develop a language component once and for all, is to anticipate all the possible context requirements at the moment of writing such language component. If we aim at using languages as libraries, this is an unacceptable constraint, as it implies a closed world in which all the context requirements need to be known before composition time. This leads to our third research question.

Research Question 3 (RQ3): How to compose modular interpreters with different context-requirements using object algebras?

Results. We present Implicit Context Propagation for object algebra-based modular interpreters. The technique relies on a lifting from base (context oblivious) interpreters to interpreters that have an extended signature accepting an arbitrary number of new context parameters. These additional parameters are implicitly propagated through the evaluation of the base interpreter. Context parameters (representing semantic entities) inherit the host-language semantics; for instance, to represent mutable state we use a mutable data structure from the host language.

This question is the subject of Chapter 4 and the scientific output is summarized in Section 1.4.3.

1.3.4 Built-in Effect Handling in an Object-Oriented Language

In Implicit Context Propagation using Object Algebras, the context parameters reuse the native mechanisms of the host language to implement side-effects. As a result, the language developer cannot control the interaction between the side-effects. For example, assuming that a backtracking context is implemented using exceptions and a stateful context using mutable data structures, what happens if we want to compose a component that requires backtracking with one that requires state? How does the native exception mechanism of the host language interact with the native support for mutation?

There are several solutions to this problem presented in the community of functional programming, such as monad transformers [LHJ95] or algebraic effects [PP03; Pre15]. In particular, algebraic effects have gained traction in the last years (e.g. [Lei17; LMM17]) thanks to their desirable properties in terms of modularity and flexibility. An algebraic effect acts as an interface that defines effectful operations, leaving the actual implementation open. Handlers provide the concrete semantics to these operations in a custom, pluggable manner. Considering the limitations

of Implicit Context Propagation in terms of explicit side-effect interaction, we are interested in a similar embodiment of the idea of algebraic effects and handlers, but in the context of object-oriented programming. This interest leads to our final research question.

Research Question 4 (RQ4): How to integrate effect handling in an object-oriented Language?

Results. *JEff* is a Java-like language where effects and handlers are defined natively; the former as so-called effect interfaces, and the latter as classes that implement such interfaces. This integration of effect handling and object orientation allow user-defined effects to benefit from interface polymorphism. We introduce the dynamic and static semantics of *JEff* by means of a core calculus dubbed *Featherweight JEff*. The dynamic semantics is inspired by traditional algebraic effect systems found in functional programming, while the static semantics is given by a type and effect system that assign types to programs. The typing relation is parameterized by a set of effect privileges that are declared using method-level annotations. *JEff* is presented in depth in Chapter 5.

1.4 Origins of the Chapters

The work associated with our research questions has resulted in five peer-reviewed publications. I am the main author of four of them, while in one of them I have contributed as a second author. This section provides the details about these publications.

1.4.1 Research Question 1 (Chapter 2)

A. Biboudis, P. Inostroza, and T. van der Storm. "Recaf: Java Dialects As Libraries". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2016. Amsterdam, Netherlands: ACM, 2016, pp. 2–13

Recaf allows programmers to define extensions to Java. Being Java a statement- and expression-based language, extensions can be defined both at the level of statement-like syntax, or expression-like syntax. My technical contribution as a second author is the implementation of expression virtualization. I also contributed to the writing and discussion of the ideas reported in the paper, during the visit of Aggelos Biboudis to CWI, in 2016.

1.4.2 Research Question 2 (Chapter 3)

P. Inostroza, T. van der Storm, and S. Erdweg. "Tracing Program Transformations with String Origins". In: *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings*. Ed. by D. D. Ruscio and D. Varró. Vol. 8568. Lecture Notes in Computer Science. Springer, 2014, pp. 154–169

1.4.3 Research Question 3 (Chapter 4)

P. Inostroza and T. van der Storm. "Modular Interpreters for the Masses: Implicit Context Propagation Using Object Algebras". In: *Proc. of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 171–180

P. Inostroza and T. van der Storm. "Modular interpreters with Implicit Context Propagation". In: *Computer Languages, Systems & Structures* 48 (2017). Special Issue on the 14th International Conference on Generative Programming: Concepts & Experiences (GPCE'15), pp. 39–67

1.4.4 Research Question 4 (Chapter 5)

P. Inostroza and T. van der Storm. "JEff: Objects for Effect". In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2018. Boston, Massachussets, USA: ACM, 2018

1.5 Software Artifacts

In the context of this thesis a number of software artifacts were produced and shared as online open source repositories, facilitating the reproducibility of our techniques in the scientific community.

Recaf The Recaf repository[3] contains the framework for creating extensions as libraries in Java. The framework consists of, on one hand, the generic transformation that desugars programs in the customized Java into regular Java code, and on the other hand, the object algebra-based interpreters implementing the default Java semantics to facilitate the development of extensions. The user can extend these interpreters

[3] https://github.com/cwi-swat/recaf

(possibly overriding parts of their behavior) in order to implement the desired custom semantics. For the transformation, the framework requires the library containing the interpreter of Rascal [KSV09], a metaprogramming language for source code analysis and transformation.

String Origins String Origins have been implemented as an experimental feature of the Rascal metaprogramming language and made available as a separate branch in the official Rascal repository.[4] The applications and the example code discussed in Chapter 3 have all been prototyped in the String Origins-enabled Rascal and made available in their own repository.[5] Rascal extended with String Origins has also been used in [EvdSD14].

Implicit Propagation The lifting presented in order to implicitly propagate the context parameters has been encoded as a Scala macro and made available online.[6] This repository also contains examples of modularly defined base interpreters for subfeatures of Featherweight Java [IPW99]. These features can be combined after the automatic macro-based lifting of the base interpreters.

JEff The grammar, dynamic semantics and type system of JEff's core calculus, Featherweight JEff, have been encoded using PLT Redex [KCD+12], an embedded domain-specific language for modeling the semantics of programming languages. The source code of the models is available online.[7] In order to execute the models, it is necessary to install Racket (PLT Redex's host language). Besides the PLT Redex model, the repository also hosts a Rascal transformation that allows users to write FJEff programs using a syntax that is closer to the one introduced in Chapter 5. Such FJEff programs are transformed into the PLT Redex syntax in order to be executed. This transformation can be invoked using an executable JAR.

1.6 Dissertation Structure

The coming four chapters of this thesis provide answers to research questions RQ1, RQ2, RQ3 and RQ4, respectively. Finally, Chapter 6 summarizes the conclusions reached in each chapter and discusses open challenges and possibilities for future work.

[4] https://github.com/usethesource/rascal/tree/string-origins
[5] https://github.com/cwi-swat/string-origins
[6] https://github.com/cwi-swat/implicit-propagation
[7] https://github.com/cwi-swat/jeff-model

2 Recaf: Java Dialects as Libraries

Mainstream programming languages like Java have limited support for language extensibility. Without mechanisms for syntactic abstraction, new programming styles can only be embedded in the form of libraries, limiting expressiveness.

In this chapter, we present Recaf, a lightweight tool for creating Java dialects; effectively extending Java with new language constructs and user defined semantics. The Recaf compiler generically transforms designated method bodies to code that is parameterized by a semantic factory (Object Algebra), defined in plain Java. The implementation of such a factory defines the desired runtime semantics.

We applied our design to produce several examples from a diverse set of programming styles and two case studies: we define i) extensions for generators, asynchronous computations and asynchronous streams and ii) a Domain-Specific Language (DSL) for Parsing Expression Grammars (PEGs), in a few lines of code.

This chapter is based on the following published article: A. Biboudis, P. Inostroza, and T. van der Storm. "Recaf: Java Dialects As Libraries". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2016. Amsterdam, Netherlands: ACM, 2016, pp. 2–13.

2.1 Introduction

Programming languages are as expressive as their mechanisms for abstraction. Most mainstream languages support functional and object-based abstractions, but it is well-known that some programming patterns or idioms are hard to encapsulate using these standard mechanisms. Examples include complex control-flow features, asynchronous programming, or embedded DSLs. It is possible to approximate such features using library-based encodings, but this often leads to code that is verbose and tedious to maintain.

Consider the example of a simple extension for automatically closing a resource in Java:[1]

```
using (File f: IO.open(path)) { ... }
```

Such a language feature abstracts away the boilerplate needed to correctly close an IO resource, by automatically closing the `f` resource whenever the execution falls out of the scope of the block.

Unfortunately, in languages like Java, defining such constructs as a library is limited by inflexible statement-oriented syntax, and semantic aspects of (non-local) control-flow. For instance, one could try to simulate `using` by a method receiving a closure, like `void using(Closeable r, Consumer<Closeable> block)`. However, the programmer can now no longer use non-local control-flow statements (e.g., `break`) within the closure block, and all variables will become effectively final as per the Java 8 closure semantics. Furthermore, encodings like this disrupt the conventional flow of Java syntax, and lead to an atypical, inverted code structure. More sophisticated idioms lead to even more disruption of the code (case in point is "call-back hell" for programming asynchronous code).

In this work we present Recaf[2], a lightweight tool[3] to extend Java with custom dialects. Extension writers do not have to alter Java's parser or write any transformation rules. The Recaf compiler generically transforms an extended version of Java, into code that builds up the desired semantics. Hence, Recaf is lightweight: the programmer can define a dialect without stepping outside the host language.
Recaf is based on two key ingredients:

- Syntax extension: Java's surface syntax is liberated to allow the definition of new language constructs, as long as they follow the pattern of existing control-flow or declaration constructs. (For instance, the `using` construct follows the pattern of Java's `for`-each.) A pattern that is used, drives a corresponding transformation that Recaf performs.

[1] similar to C#'s `using` construct, or Java's try-with-resources
[2] As in *recaffeinating coffee* which describes the process of enhancing its caffeine content [CV08].
[3] The code is available at `https://github.com/cwi-swat/recaf`.

- Semantics extension: a single, syntax-directed transformation maps method bodies (with or without language extensions) to method calls on polymorphic factories which construct objects encapsulating the user-defined or customized semantics.

The factories are developed within Java as Object Algebras [OC12], which promote reusability and modular, type-safe extension of semantic definitions.

The combination of the two aforementioned key points enables a range of application scenarios. For instance, Recaf can be used to: extend Java with new syntactic constructs (like `using`), modify or instrument the semantics of Java (e.g., to implement aspects like tracing, memoization), replace the standard Java semantics with a completely different semantics (e.g., translate Java methods to Javascript source code), embed DSLs into Java (e.g., grammars, or GUI construction), define semantics-parametric methods which support multiple interpretations, and combine any of the above in a modular fashion (e.g., combine `using` with a tracing aspect). Developers can define the semantics of new and existing constructs and create DSLs for their daily programming needs. Language designers can experiment with new features, by quickly translating their denotations in plain Java. In other words, Recaf brings the vision of "languages as libraries" to mainstream, object-oriented programming languages.

The contributions of this chapter are summarized as follows:

- We present a transformation of Java statement Abstract Syntax Trees (ASTs) with extended syntax to virtualize their semantics, and we show how the semantics can be defined as Object Algebras (Section 2.3).
- We generalize the transformation to virtualize Java expressions, widening the scope of user defined semantics (Section 2.4).
- We describe the implementation of Recaf, and how it deals with certain intricacies of the Java language (Section 2.5).
- We evaluate the expressiveness of Recaf with two case studies: i) providing language support for generators and asynchronous computations and ii) creating a DSL for parser combinators (Section 2.6).

The results of the case studies and directions for future work are discussed in Section 2.7.

2.2 Overview

2.2.1 Recaffeinating Java with Recaf

Figure 2.1 gives a bird's eye overview of Recaf. It shows how the `using` extension is used and implemented using Recaf. The top shows a snippet of code illustrating how the programmer would use a Recaf extension, in this case consisting of the

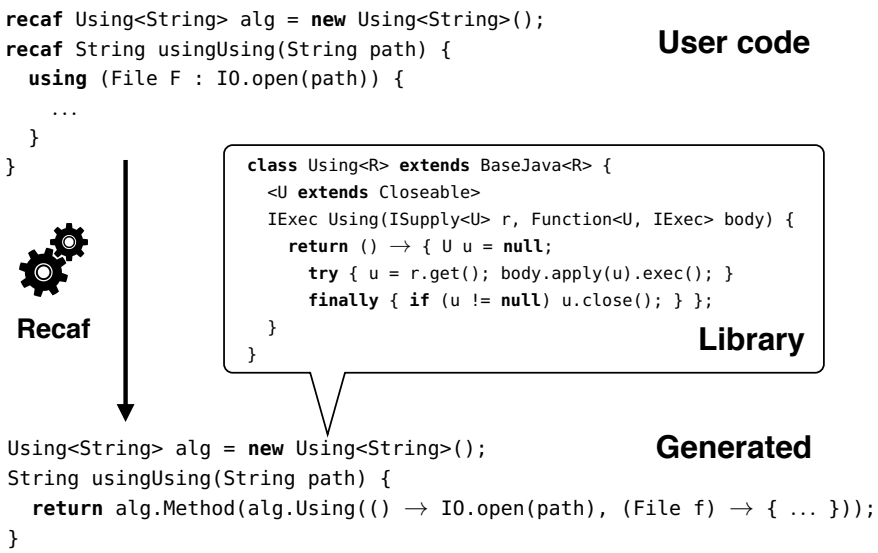


Figure 2.1: High level overview of Recaf

`using` construct. The programmer writes an ordinary method, decorated with the **`recaf`** modifier to trigger the source-to-source transformation. To provide the custom semantics, the user also declares a **`recaf`** field, in scope of the **`recaf`** method. In this case, the field `alg` is initialized to be a `Using` object, defined over the concrete type `String`.

The downward arrow indicates Recaf's source-to-source transformation which virtualizes the semantics of statements by transforming the Recaf code fragment to the plain Java code at the bottom. Each statement in the user code is transformed into calls on the `alg` object. The `using` construct itself is mapped to the `Using` method. The `Using` class shown in the call-out, defines the semantics for `using`. It extends a class (`BaseJava`) capturing the ordinary semantics of Java, and defines a single method, also called `Using`. This particular `Using` method defines the semantics of the `using` construct as an interpreter of type `IExec`.

In addition to using a **`recaf`** field to specify the semantics of a **`recaf`** method, it is also possible to decorate a formal parameter of a method with the **`recaf`** modifier. This allows binding of the semantics at the call site of the method itself. Thus, Recaf supports three different binding times for the semantics of a method: static (using a static field), at object construction time (using an instance field), and late binding (method parameter). Recaf further makes the distinction between statement-only virtualization and full virtualization. In the latter case, expressions are virtualized

too. This mode is enabled by using the **recaff** keyword, instead of **recaf**. Section 2.4 provides all the details regarding the difference.

2.2.2 Object Algebras

The encoding used for the `Using` class in Figure 2.1 follows the design pattern of Object Algebras [OC12] which has already been applied to numerous cases in the literature [BPF+15; IvdS15; OvdSL+13]. Object Algebras can be seen as an object-oriented encoding of tagless interpreters [CKS09]. Instead of defining a language's abstract syntax using concrete data structures, it is defined using generic factories: a generic interface declares generic methods for each syntactic construct. Implementations of such interfaces define a specific semantics by creating semantic objects representing operations like pretty printing, evaluation, and so on.

Object Algebras are a simple solution to the *expression problem* [Wad98]. As such they provide type-safe, modular extensibility along two axes: adding new data variants and adding new operations over them without changing existing code. For instance, the `Using` algebra extends the base Java semantics with a new syntactic construct. On the other hand, the generic interface representing the abstract syntax of Java can also be implemented *again*, to obtain a different semantics. In the remainder of this chapter we define the algebras as Java 8 interfaces with default methods to promote additional flexibility for modularly composing semantic modules.

2.3 Statement Virtualization

In this section we describe the first level of semantic and syntactic polymorphism offered by Recaf, which restricts virtualization and syntax extension to statement-like constructs.

2.3.1 μJava

μJava is a simplified variant of Java used for exposition purposes. In μJava all variables are assumed to be final, there is no support for primitive types nor **void** methods, and all variable declarations have initializers. Figure 2.2, shows the abstract syntax of μJava statements and method bodies in the form of Object Algebra interfaces.

Both interfaces are parametric in two generic types, `R` and `S`. `R` represents the return type of the method, and `S` the semantic type of statements. The method `Method` in `MuJavaMethod` mediates between the denotation of statements (`S`) and the return type `R` of the virtualized method. The programmer of a Recaf method needs to ensure that `R` returned by `Method` corresponds to the actual return type declared in the method. Note that `R` does not have to be bound to the same concrete type in both `MuJavaMethod`

```
interface MuJavaMethod<R, S> { R Method(S s); }

interface MuJava<R, S> {
        S Exp(Supplier<Void> e);
        S If(Supplier<Boolean> c, S s1, S s2);
        <T> S For(Supplier<Iterable<T>> e, Function<T, S> s);
        <T> S Decl(Supplier<T> e, Function<T, S> s);
        S Seq(S s1, S s2);
        S Return(Supplier<R> e);
        S Empty();
}
```

Figure 2.2: Object Algebra interfaces defining the abstract syntax of μJava method bodies and statements.

and `MuJava`. This means that the return type of a virtualized method can be different than the type of expressions expected by `Return`.

The `MuJava` interface assumes that expressions are represented using the standard Java `Supplier` type, which represents thunks. Java expressions may perform arbitrary side-effects; the thunks ensure that evaluation is delayed until after the semantic object are created.

The constructs `For` and `Decl` employ higher-order abstract syntax (HOAS [PE88]) to introduce local variables. As a result, the bodies of declarations (i.e., the statements following it, within the same scope) and **`for`**-each loops are represented as functions from some generic type `T` to the denotation `S`.

2.3.2 Transforming Statements

The transformation for μJava is shown in Figure 2.3, and consists of two transformation functions $\mathcal{M}$ and $\mathcal{S}$, respectively transforming method bodies, and statements. The transformation folds over the syntactic structure of μJava, compositionally mapping each construct to its virtualized representation. Both functions are subscripted by the expression a, which represents the actual algebra that is used to construct the semantics. The value of a is determined by the **`recaf`** modifier on a field or formal parameter.

As an example consider the code shown on the left of Figure 2.4. The equivalent code after the Recaf transformation is shown on the right. The semantics of the code is now virtualized via the algebra object `a`. The algebra `a` may implement the same semantics as ordinary Java, but it can also customize or completely redefine it.

$$
\begin{array}{rcl}
\mathcal{M}_a[\![S]\!] & = & \texttt{return}\ a\texttt{.Method(}\mathcal{S}_a[\![S]\!]\texttt{);} \\
 & & \\
\mathcal{S}_a[\![e\texttt{;}]\!] & = & a\texttt{.Exp(() } \rightarrow \texttt{ \{}e\texttt{; return null;\})} \\
\mathcal{S}_a[\![\texttt{if (}e\texttt{) }S1\texttt{ else }S2\,]\!] & = & a\texttt{.If(() } \rightarrow e\texttt{, }\mathcal{S}_a[\![S1]\!]\texttt{,}\mathcal{S}_a[\![S2]\!]\texttt{)} \\
\mathcal{S}_a[\![\texttt{for(}T\ x\texttt{: }e\texttt{) }S]\!] & = & a\texttt{.For(() } \rightarrow e\texttt{, (}T\ x\texttt{) } \rightarrow \mathcal{S}_a[\![S]\!]\texttt{)} \\
\mathcal{S}_a[\![T\ x\ \texttt{=}\ e\texttt{; }S]\!] & = & a\texttt{.Decl(() } \rightarrow e\texttt{, (}T\ x\texttt{) } \rightarrow \mathcal{S}_a[\![S]\!]\texttt{)} \\
\mathcal{S}_a[\![S1\texttt{; }S2]\!] & = & a\texttt{.Seq(}\mathcal{S}_a[\![S1]\!]\texttt{, }\mathcal{S}_a[\![S2]\!]\texttt{)} \\
\mathcal{S}_a[\![\texttt{return}\ e\texttt{;}]\!] & = & a\texttt{.Return(() } \rightarrow e\texttt{)} \\
\mathcal{S}_a[\![\texttt{;}]\!] & = & a\texttt{.Empty()} \\
\mathcal{S}_a[\![\texttt{\{ }S\texttt{ \}}]\!] & = & \mathcal{S}_a[\![S]\!]
\end{array}
$$

Figure 2.3: Virtualizing method statements into statement algebras.

```
for (Integer x: l)
  if (x % 2 == 0)
   return x;
  else ;
return null;
```

```
return a.Method(
 a.Seq(
  a.For(() → l, (Integer x) →
    a.If(() → x % 2 == 0,
        a.Return(() → x),
        a.Empty())),
  a.Return(() → null)));
```

Figure 2.4: Example method body (left) and its transformation into algebra `a` (right).

2.3.3 Statement Syntax

Statement syntax is based on generalizing the existing control-flow statement syntax of Java. Informally speaking, wherever Java requires a keyword (e.g., **`for`**, **`while`** etc.), Recaf allows the use of an identifier. This identifier will then, by convention, correspond to a particular method with the same name in the semantic algebra.

The following grammar describes the syntax extensions of statements (S) for μJava:

$$
\begin{array}{rcll}
S & ::= & x!\ e\ ; & \text{Return-like} \\
 & | & x\ (T\ x\texttt{: }\ e)\ S & \text{For-each like} \\
 & | & x\ (e)\ \{S\} & \text{While-like} \\
 & | & x\ \{S\} & \text{Try-like} \\
 & | & x\ T\ x = e; & \text{Declaration-like}
\end{array}
$$

This grammar defines a potentially infinite family of new language constructs, by using identifiers (x) instead of keywords. Each production is a generalization of existing syntax. For instance, the first production, follows syntax of **`return`** e, with the difference that an exclamation mark is needed after the identifier x to avoid ambiguity. The second production is like **`for`**–each, the third like **`while`**, and the fourth follows the

$$
\begin{array}{rcl}
\mathcal{S}_a[\![x\,!\;e;]\!] & = & a.x(() \to e) \\
\mathcal{S}_a[\![x\;\;(T\;y\colon e)\;S]\!] & = & a.x(() \to e,\;\; (T\;y) \to \mathcal{S}_a[\![S]\!]) \\
\mathcal{S}_a[\![x\;T\;y\;=e;\;S]\!] & = & a.x(() \to e,\;\; (T\;y) \to \mathcal{S}_a[\![S]\!]) \\
\mathcal{S}_a[\![x\;\;(e)\;S]\!] & = & a.x(() \to e, \mathcal{S}_a[\![S]\!]) \\
\mathcal{S}_a[\![x\;\;\{\;S\;\}]\!] & = & a.x(\mathcal{S}_a[\![S]\!])
\end{array}
$$

Figure 2.5: Transforming syntax extensions to algebra method calls.

pattern of **`if`** without **`else`**. Finally, the last production supports custom declarations, where the first identifier x represents the keyword.

Transforming the extension into an algebra simply uses the keyword identifier `x` as a method name, but follows the same transformation rules as for the corresponding, non-extended constructs. The transformation rules are shown in Figure 2.5.

2.3.4 Direct Style Semantics

The direct style interpreter for μJava is defined as the interface `MuJavaBase`, implementing it using default methods and is declared as follows:

```
interface MuJavaBase<R> extends MuJava<R, IExec> { ... }
```

The type parameter `R` represents the return type of the method. `S` is bound to the type `IExec`, which represents thunks (closures):

```
interface IExec { void exec(); }
```

The algebra `MuJavaBase` thus maps μJava statement constructs to semantic objects of type `IExec`. Most of the statements in μJava have a straightforward implementation. Non-local control-flow (i.e., **`return`**), however, is implemented using exception handling.

The method `Method` ties it all together and mediates between the evaluation of the semantic objects, returned by the algebra, to the actual return type of the method:

```
default R Method(IExec s) {
  try { s.exec(); }
        catch (Return r) { return (R)r.value; }
        catch (Throwable e) { throw new RuntimeException(e); }
  return null;
}
```

Since the mapping between the statement denotation and the actual return type of a method is configurable it is not part of `MuJavaBase` itself. This way, `MuJavaBase` can be reused with different `Method` implementations.

Example: Maybe As a simple example, similar to the `using` extension introduced in Section 2.1, consider a `maybe` construct, to unwrap an optional value (of type `java.util.Optional`). In a sense, `maybe` literally overrides the semicolon, similar to the bind operator of Haskell. Syntactically, the `maybe` operator follows the declaration-like syntax. It is defined as follows:

```
interface Maybe<R> extends MuJavaBase<R> {
  default <T> IExec Maybe(Supplier<Optional<T>> x, Function<T, IExec> s) {
    return () → {
      Optional<T> opt = x.get();
      if (opt.isPresent()) s.apply(opt.get()).exec();
    };
  }
}
```

The `Maybe` method returns an `IExec` closure that evaluates the expression (of type `Optional`), and if the optional is not empty, executes the body of `maybe`.

2.3.5 Continuation-Passing Style Semantics

The direct style base interpreter can be used for many extensions like `using` or `maybe`. However, language constructs that require non-local control-flow semantics require a continuation-passing style (CPS) interpreter. This base interpreter can be used instead of the direct style interpreter for extensions like coroutines, backtracking, call/cc etc. It also shows how Object Algebras enables the definition of two different semantics for the same syntactic interface.

The cps style interpreter is defined as the interface `MuJavaCPS`, similarly to `MuJavaBase`:

```
interface MuJavaCPS<R> extends MuJava<R, SD<R>> { ... }
```

The `MuJavaCPS` algebra maps μJava abstract syntax to CPS denotations (`SD`), defined as follows:

```
interface SD<R> { void accept(K<R> r, K0 s); }
```

`SD<R>` is a functional interface that takes as parameters a return and a success continuation. The return continuation `r` is of type `K<R>` (a consumer of `R`) and contains the callback in the case a statement is the return statement. The success continuation is of type `K0` (a thunk) and contains the callback in the case the execution falls off without returning.

To illustrate the CPS interpreter, consider the following code that defines the semantics of the **`if-else`** statement:

```
default SD<R> If(Supplier<Boolean> c, SD<R> s1, SD<R> s2) {
  return (r, s) → { if (c.get()) s1.accept(r, s);
                    else s2.accept(r, s);}; }
```

```
interface Backtrack<R>
    extends MuJavaCPS<R>, MuJavaMethod<List<R>, SD<R>> {

  default List<R> Method(SD<R> body) {
       List<R> result = new ArrayList<>();
    body.accept(ret → { result.add(ret); }, () → {});
    return result;
  }
  default <T> SD<R> Choose(Supplier<Iterable<T>> e, Function<T, SD<R>> s) {
    return (r, s0) → {
      for (T t: e.get()) s.apply(t).accept(r, s0);
    };
  }
}
```

Figure 2.6: Backtracking Extension

Based on the truth-value of the condition, either the then branch or the else branch is executed, with the same continuations as received by the if-then-else statement.

Example: Backtracking The CPS interpreter serves as a base implementation for language extensions, requiring complex control flow. We demonstrate the backtracking extension that uses Wadler's list of successes technique [Wad85] and introduces the `choose` keyword. As an example, consider the following method which finds all combinations of integers out of two lists that sum to 8.

```
List<Pair> solve(recaf Backtrack<Pair> alg) {
  choose Integer x = asList(1, 2, 3);
  choose Integer y = asList(4, 5, 6);
  if (x + y == 8) {
    return new Pair(x, y);
  }
}
```

In Figure 2.6 we present the extension for μJava. Note that `Method` has a generic parameter type in the `MuJavaMethod` interface. In this case we change the return type of method to `List<T>` instead of just `T`. The result is the list of successes, so the return continuation should add the calculated item in the return list, instead of just passing it to the continuation. `Choose` simply invokes its success continuation for every different value, effectively replaying the execution for every element of the set of values.

```
interface MuStmJava<S, E> {
        S Exp(E x);
        <T> S Decl(E x,Function<T,S> s);
        <T> S For(E x,Function<T,S> s);
        S If(E c, S s1, S s2);
        S Return(E x);
        S Seq(S s1, S s2);
        S Empty();
}
```

```
interface MuExpJava<E> {
        E Lit(Object x);
        E This(Object x);
        E Field(E x, String f);
        E New(Class<?> c, E...es);
        E Invoke(E x, String m, E...es);
        E Lambda(Object f);
        E Var(String x, Object it);
}
```

Figure 2.7: Generic interfaces for the full abstract syntax of μJava.

2.4 Full Virtualization

The previous section discussed virtualization of declaration and control-flow statements. In this section we widen the scope of Recaf for virtualizing expressions as well.

2.4.1 Expression Virtualization

Until know we have dissected the `MuJava` interface of Figure 2.2. That interface still does not support virtualized expressions since it requires the concrete type `Supplier` in expression positions. To enable full virtualization, we have to use the algebraic interfaces shown in Figure 2.7, where expressions are represented by the generic type `E`. `MuExpJava` specifies the semantic objects for μJava expressions. The interface `MuStmJava` is similar to `MuJava`, but changes the concrete `Supplier` arguments to the generic type `E`.

Compared to full Java, the expression sub language of μJava makes some additional simplifying assumptions: there are no assignment expressions, no super calls, no array creation, no static fields or methods, no package qualified names, and field access and method invocation require an explicit receiver. For brevity, we have omitted infix and prefix expressions.

To support expression virtualization, the transformation of statements is modified according to the rules of Figure 2.8. The function $\mathcal{E}$ folds over the expression structure and creates the corresponding method calls on the algebra a. Consider, for example, how full virtualization desugars the μJava code fragment `for (Integer x: y) println(x + 1);`:

```
a.For(a.Var("y", y), Integer x →
  a.Exp(a.Invoke(a.This(this), "println",
                 a.Add(a.Var("x", x), a.Lit(1)))));
```

$$
\begin{array}{rcl}
\mathcal{S}_a[\![e;]\!] & = & a.\texttt{Exp}(\mathcal{E}_a[\![e]\!]) \\
\mathcal{S}_a[\![\texttt{if }(e)\ S1\ \texttt{else}\ S2]\!] & = & a.\texttt{If}(\mathcal{E}_a[\![e]\!], \mathcal{S}_a[\![S1]\!], \mathcal{S}_a[\![S2]\!]) \\
\mathcal{S}_a[\![\texttt{for}(T\ x\texttt{:}\ e)\ S]\!] & = & a.\texttt{For}(\mathcal{E}_a[\![e]\!], (T\ x)\ \rightarrow \mathcal{S}_a[\![S]\!]) \\
\mathcal{S}_a[\![T\ x\ \texttt{=}\ e;\ S]\!] & = & a.\texttt{Decl}(\mathcal{E}_a[\![e]\!], (T\ x)\ \rightarrow \mathcal{S}_a[\![S]\!]) \\
\mathcal{S}_a[\![\texttt{return}\ e;]\!] & = & a.\texttt{Return}(\mathcal{E}_a[\![e]\!]) \\
 & & \\
\mathcal{E}_a[\![v]\!] & = & a.\texttt{Lit}(v) \\
\mathcal{E}_a[\![x]\!] & = & a.\texttt{Var}(\texttt{"}x\texttt{"}, x) \\
\mathcal{E}_a[\![\texttt{this}]\!] & = & a.\texttt{This}(\texttt{this}) \\
\mathcal{E}_a[\![e.x]\!] & = & a.\texttt{Field}(\mathcal{E}_a[\![e]\!], \texttt{"}x\texttt{"}) \\
\mathcal{E}_a[\![e.x(e1,...,en)]\!] & = & a.\texttt{Invoke}(\mathcal{E}_a[\![e]\!], \texttt{"}x\texttt{"}, \mathcal{E}_a[\![e1]\!], ..., \mathcal{E}_a[\![en]\!]) \\
\mathcal{E}_a[\![\texttt{new}\ T(e1,...,en)]\!] & = & a.\texttt{New}(T.\texttt{class}, \mathcal{E}_a[\![e1]\!], ..., \mathcal{E}_a[\![en]\!]) \\
\mathcal{E}_a[\![(x1,...,xn) \rightarrow S]\!] & = & a.\texttt{Lambda}((x1,...,xn)\ \rightarrow \mathcal{S}_a[\![S]\!])
\end{array}
$$

Figure 2.8: Transforming statements (modified) and expressions.

Note how the HOAS representation of binders carries over to expression virtualization, through the `Var` constructor. The additional `String` argument to `Var` is not essential, but provides additional meta data to the algebra.

Recaf does not currently support new syntactic constructs for user defined expression constructs. We assume that in most cases ordinary method abstraction is sufficient.[4] The examples below thus focus on instrumenting or replacing the semantics of ordinary μJava expressions.

2.4.2 An Interpreter for μJava Expressions

Just like statements, the base semantics of μJava expressions is represented by an interpreter, this time conforming to the interface `MuExpJava`, shown in Figure 2.7. This interpreter binds `E` to the closure type `IEval`:

```
interface IEval { Object eval(); }
```

As `IEval` is not polymorphic, we do not make any assumptions about the type of the returned object. The interpreter is fully dynamically typed, because Java's type system is not expressive enough to accurately represent the type of expression denotations, even if the Recaf transformation would have had access to the types of variables and method signatures.

The evaluation of expressions is straightforward. Field access, object creation (**`new`**), and method invocation, however are implemented using reflection. For instance, the following code defines the semantics for field lookup:

[4] The only situation where a new kind of expression would be useful is when arguments of the expression need to evaluated lazily, as in short-circuiting operators.

```
default IEval Var(String x, Object v) {
  return () → {
    System.err.println(x + " = " + v);
    return MuExpJavaBase.super.Var(x, v).eval();
  };
}
```

Figure 2.9: Intercepting field accesses for tracing.

```
default IEval Field(IEval x, String f) {
   return () → {
      Object o = x.eval();
      Class<?> clazz = o.getClass();
      return clazz.getField(f).get(o);
   };
}
```

The class of the object whose field is requested, is discovered at runtime, and then, the reflective method `getField` is invoked in order to obtain the value of the field for the requested object.

Example: Aspects A useful use case for expression virtualization is defining aspect-like instrumentation of expression evaluation. The algebra methods of the base interpreter are overridden, they implement the additional behavior, and delegate to the parent's implementation with the **`super`** keyword. As an example, consider an aspect defining tracing of the values of variables during method execution. The algebra extends the base interpreter for Java expressions and overrides the `Var` definition, for variable access. Figure 2.9 shows how the overridden `Var` method uses the variable name and the actual received value passed to print out the tracing information, and then calls the **`super`** implementation.

Example: Library embedding In the following example we demonstrate library embedding of a simple constraint solving language, *Choco* [PFL15], a Java library for constraint programming. Choco's programming model is heavily based on factories nearly for all of its components, from variable creation, constraint declaration over variables, to search strategies.

We have developed a Recaf embedding which translates a subset of Java expressions to the internal constraints of Choco, which can then be solved. The `Solve` algebra defines the `var` extension to declare constraint variables. The `solve!` statement posts constraints to Choco's solver. This embedding illustrates how the expression

virtualization allows the extension developer to completely redefine (a subset of) Java's expression syntax.

```
recaf Solve alg = new Solve();
recaff Iterable<Map<String,Integer>> example() {
  var 0, 5, IntVar x;
  var 0, 5, IntVar y;
  solve! x + y < 5;
}
```

2.5 Implementation of Recaf

All Recaf syntactic support is provided by Rascal [KSV09], a language for source code analysis and transformation. Rascal features built-in primitives for defining grammars, tree traversal and concrete syntax pattern matching. Furthermore, Rascal's language workbench features [EvdSV+15] allow language developers to define editor services, such as syntax highlighting or error marking, for their languages.

2.5.1 Generically Extensible Syntax for Java

Section 2.3.3 introduced generic syntax extensions in the context of μJava, illustrating how the base syntax could be augmented by adding arbitrary keywords, as long as they conform to a number of patterns, e.g. `while`- or declaration-like. We implemented these patterns and a few additional ones for full Java using Rascal's declarative syntax rules. These rules modularly extend the Java grammar, defined in Rascal's standard library using productions for each case that we identify as an extensibility point: return-like, declaration-like, for-like, switch-like, switch-like (as a for) and try-like.

2.5.2 Transforming Methods

Recaf source code transformation transforms any method that has the `recaf` or `recaff` modifier. If the modifier is attached to the method declaration, the algebra is expected to be declared as field in the enclosing scope (class or interface). If the modifier is attached to a method's formal parameter, that parameter itself is used instead. Furthermore, if the `recaff` modifier is used, expressions are transformed as well.

The transformation is defined as Rascal rewrite rules that match on a Java statement or expression using concrete-syntax pattern matching. This means that the matching occurs on the concrete syntax tree directly, having the advantage of preserving comments and indentation from the Recaf source file. As an example, the following rule defines the transformation of the `while`-statement:

```
Expr stm2alg((Stm)`while (<Expr c>) <Stm s>`,Id a,Names ns)
  = (Expr)`<Id a>.While(<Expr c2>, <Expr s2>)`
  when
    Expr c2 := injectExpr(c, a, ns),
    Expr s2 := stm2alg(s, a, ns);
```

This rewrite rule uses the actual syntax of the Java **while** statement as the matching pattern and returns an expression that calls the `While` method on the algebra `a`. The condition `c` and the body `s` are transformed in the `when`-clause (where `:=` indicates binding through matching). The function `injectExpr` either transforms the expression `c`, in the case of transformations annotated with the **recaff** keyword, or creates closures of type `Supplier` otherwise. The body `s` is transformed calling recursively `stm2alg`.

The `ns` parameter represents the declared names that are in scope at this point in the code and is the result of a local name analysis needed to correctly handle mutable variables. Local variables introduced by declarations and for-loops are mutable variables in Java, unless they are explicitly declared as **final**. This poses a problem for the HOAS encoding we use for binders: the local variables become parameters of closures, but if these parameters are captured inside another closure, they have to be (effectively) final. To correctly deal with this situation, variables introduced by declarations or for-loops are wrapped in explicit reference objects, and the name is added to the `Names` set. Whenever such a variable is referenced in an expression it is unwrapped. For extensions that introduce variables it is unknown whether they should be mutable or not, so the transformation assumes they are final, as a default. In total, the complete Recaf transformation consists of 790 SLOC.

2.5.3 Recaf Runtime

The Recaf runtime library comes with two base interpreters of Java statements, similar to `MuJavaBase` and `MuJavaCPS`, and an interpreter for Java Expressions. In addition to **return**, the interpreters support the full non-local control-flow features of Java, including (labeled) **break**, **continue** and **throw**. The CPS interpreter represents each of those as explicit continuations in the statement denotation (`SD`), whereas the direct style interpreter uses exceptions.

The main difference between `MuExpBase` and the full expression interpreter is handling of assignments. We model mutable variables by the interface `IRef`, which defines a setter and getter to update the value of the single field that it contains. The `IRef` interface is implemented once for local variables, and once for fields. The latter uses reflection to update the actual object when the setter is called. In addition to the `Var(String,Object)` constructor, the full interpreter features the constructor `Ref(String,IRef<?>)` to model mutable variables. The expression transformation uses the local name analysis (see above) to determine whether to insert `Var` or `Ref` calls.

Since the Recaf transformation is syntax-driven, some Java expressions are not supported. For instance, since the expression interpreter uses reflection to call methods, statically overloaded methods are currently unsupported (because it is only possible to dispatch on the runtime type of arguments). Another limitation is that Recaf does not support static method calls, fields references or package qualified names. These three kinds of references all use the same dot-notation syntax as ordinary method calls and field references. However, the transformation cannot distinguish these different kinds, and interprets any dot-notation as field access or method invocation with an explicit receiver. We consider a type-driven transformation for Recaf as an important direction for future work.

2.6 Case Studies

2.6.1 Spicing up Java with Side-Effects

The Dart programming language recently introduced `sync*`, `async` and `async*` methods, to define generators, asynchronous computations and asynchronous streams [MMB15] without the typical stateful boilerplate or inversion of control flow. Using Recaf, we have implemented these three language features for Java, based on the CPS interpreter, closely following the semantics presented in [MMB15].

Generators. The extension for generators is defined in the `Iter` class. The `Iter` class defines `Method` to return a plain Java `Iterable<R>`. When the `iterator()` is requested, the statement denotations start executing. The actual implementation of the iterator is defined in the client code using two new constructs. The first is `yield!`, which produces a single value in the iterator. Its signature is `SD<R> Yield(ISupply<R>)`.[5] Internally, `yield!` throws a special `Yield` exception to communicate the yielded element to a main iterator effectively pausing the generator. The `Yield` exception contains both the element, as well as the current continuation, which is stored in the iterator. When the next value of the iterator is requested, the saved continuation is invoked to resume the generator process. The second construct is `yieldFrom!` which flattens another iterable into the current one. Its signature is `SD<R> YieldFrom(ISupply<Iterable<R>> x)` and it is implemented by calling `ForEach(x, e→ Yield(() → e))`. In the code snippet below, we

[5] `ISupply` is a thunk which has a **`throws`** clause.

present a recursive implementation of a `range` operator, using both `yield!` and `yieldFrom!`:

```
recaf Iterable<Integer> range(int s, int n) {
  if (n > 0) {
    yield! s;
    yieldFrom! range(s + 1, n - 1);
  }
}
```

Async. The implementation of `async` methods also defines `Method`, this time returning a `Future<R>` object. The only syntactic extension is the `await` statement. Its signature is `<T> Await(Supplier<CompletableFuture<T>>, Function<T, SD<R>>)`, following the syntactic template of for-each. The `await` statement blocks until the argument future completes. If the future completes normally, the argument block is executed with the value returned from the future. If there is an exception, the exception continuation is invoked instead. `Await` literally passes the success continuation to the future's `whenComplete` method. The Async extension supports programming with asynchronous computations without having to resort to call-backs. For instance, the following method computes the string length of a web page, asynchronously fetched from the web.

```
recaf Future<Integer> task(String url) {
  await String html = fetchAsync(url);
  return html.length();
}
```

Async*. Asynchronous streams (or reactive streams) support a straightforward programming style on observables, as popularized by the Reactive Extensions [Mei10] framework. The syntax extensions to support this style are similar to `yield!` and `yieldFrom!` constructs for defining generators. Unlike the `yield!` for generators, however, `yield!` now produces a new element asynchronously. Similarly, the `yieldFrom!` statement is used to splice one asynchronous stream into another. Its signature reflects this by accepting an `Observable` object (defined by the Java variant of Reactive Extensions, RxJava[6]): `SD<R> YieldFrom(ISupply<Observable<R>>)`. Reactive streams offer one more construct: `awaitFor!`, which is similar to the ordinary for-each loop, but "iterates" asynchronously over a stream of observable events. Hence, its signature is `<T> SD<R> AwaitFor(ISupply<Observable<T>>, Function<T, SD<R>>)`. Whenever, a new element becomes available on the stream, the body of the `awaitFor!` is executed again. An `async*` method will return an `Observable`.

[6] `https://github.com/ReactiveX/RxJava`

Here is a simple method that prints out intermediate results arriving asynchronously on a stream. After the result is printed, the original value is yielded, in a fully reactive fashion.

```
recaf <X> Observable<X> print(Observable<X> src) {
  awaitFor (X x: src) {
    System.out.println(x);
    yield! x;
  }
}
```

2.6.2 Parsing Expression Grammars (PEGs)

To demonstrate language embedding and aspect-oriented language customization we have defined a DSL for Parsing Expression Grammars (PEGs) [For04]. The abstract syntax of this language is shown in Figure 2.10. The `lit!` construct parses an atomic string, and ignores the result. `let` is used to bind intermediate parsing results. For terminal symbols, the `regexp` construct can be used. The language overloads the standard sequencing and **`return`** constructs of Java to encode sequential composition, and the result of a parsing process. The constructs `choice`, `opt`, `star`, and `plus` correspond to the usual regular EBNF operators. The `choice` combinator accepts a list of alternatives (`alt`). The Kleene operators bind a variable x to the result of parsing the argument statement S, where the provided expression e represents the result if parsing of S fails.

The PEG language can be used by considering methods as nonterminals. A PEG method returns are parser object which returns a certain semantic value type. A simple example of parsing primary expression is shown in Figure 2.11. The method `primary` returns an object of type `Parser` which produces an expression `Exp`. Primaries have two alternatives: constant values and other expressions enclosed in parentheses. In the first branch of the `choice` operator, the `regexp` construct attempts to parse a numeric value, the result of which, if successful, is used in the **`return`** statement, returning an `Int` object representing the number. The second branch, first parses an open parenthesis, then binds the result of parsing an additive expression (implemented in a method called `addSub`) to `e`, and finally parses the closing parenthesis. When all three parses are successful, the `e` expression is returned. Note that the **`return`** statements return expressions, but the result of the method is a parser object.

Standard PEGs do not support left-recursive productions, so nested expressions are typically implemented using loops. For instance, additive expression could be defined as `addSub ::= mulDiv (("+"|"-") mulDiv)*`. Here's how the `addSub` method could define this grammar using the PEG embedding:

S	::=	`lit!` e`;`	Literals
	|	`let` T x `=` e `;`	Binding
	|	`regexp String` x `=` e `;`	Terminals
	|	S `;` S	Sequence
	|	`return` e `;`	Result
	|	`choice{` C`+}`	Alternative
	|	`opt` T x `=(`e`)` S	Zero or one
	|	`star` T x `=(`e`)` S	One or more
	|	`plus` T x `=(`e`)` S	Zero or one
C	::=	`alt` l`:` S`+`	Alternative (l = label)

Figure 2.10: Abstract syntax of embedded PEGs.

```
recaf Parser<Exp> primary() {
  choice {
    alt "value":
      regexp String n = "[0-9]+";
      return new Int(n);
    alt "bracket":
      lit! "("; let Exp e = addSub(); lit! ")";
      return e;
  }
}
```

Figure 2.11: Parsing primaries using Recaf PEGs.

```
recaf Parser<Exp> addSub() {
  let Exp l = mulDiv();
  star Exp e = (l) {
    regexp String o = "[+\\-]";
    let Exp r = mulDiv();
    return new Bin(o, e, r);
  }
  return e;
}
```

The first statement parses a multiplicative expression. The `star` construct creates zero or more binary expressions, from the operator (`o`), the left-hand side (`e`) and the right-hand side (`r`). If the body of the `star` fails to recognize a + or − sign, the `e` will be bound to the initial seed value `l`. The constructed binary expression will be fed back into the loop as `e` through every subsequent iteration.

The (partial) PEG for expressions shown above and in Figure 2.11 does not support any kind of whitespace between elements of an expression. Changing the PEG definitions manually to parse intermediate layout, however, would be very tedious and error-prone. Exploiting the Object Algebra-based architecture, we add the layout handling as a modular aspect, by extending the PEG algebra and overriding the methods that construct the parsers.

For instance, to insert layout in between sequences, the PEG subclass for layout overrides the `Seq` as follows:

```
<T, U> Parser<U> Seq(Parser<T> p1, Parser<U> p2) {
  return PEG.super.Seq(p1, PEG.super.Seq(layout, p2));
}
```

Another concern with standard PEGs is exponential worst-case runtime performance. The solution is to implement PEGs as packrat parsers [For02], which run in linear time by memoizing intermediate parsing results. Again, the base PEG language can be modularly instrumented to turn the returned parsers into memoizing parsers.

2.7 Discussion

Static Type Safety The Recaf source-to-source transformation assumes certain type signatures on the algebras that define the semantics. For instance, the transformation of binding constructs (declarations, for-each, etc.) expects `Function` types in certain positions of the factory methods. If a method of a certain signature is not present on the algebra, the developer of a Recaf method will get a static error at the compilation of the generated code.

The architecture based on Object Algebras provides type-safe, modular extensibility of algebras. Thus, the developer of semantics may enjoy full type-safety in the development of extensions. The method signatures of most of the examples and case-studies accurately describe the expected types and do not require any casts.

On the other hand, the statement evaluators represent expressions as opaque closures, which are typed in the expected result such as `Supplier<Boolean>` for the **`if-else`** statement. At the expression level, however, safety guarantees depend on the denotation types themselves. More general semantics, like the Java base expression interpreter, however, are defined in terms of closures returning `Object`. The reason is that Java's type system is not expressive enough to represent them otherwise (lacking features such as higher-kinded types and implicits). As a result, potentially malformed expressions are not detected at compile-time.

Another consequence of this limitation is that the `Supply`-based statement interpreters described in Section 2.3 cannot be combined out-of-the-box with expression interpreters in the context of Full Virtualization, as both interpreters must be defined in terms of generic expressions. Fortunately, the `Supply`-based statement interpreters

can be reused by applying the Adapter pattern [VHJ+95]. In the runtime library, we provide an adapter that maps a `Supplier`-based algebra to one that is generic in the expression type. As we have discussed earlier, this is unsafe by definition. Thus, although we can effectively integrate statement and expression interpreters, we lose static type guarantees for the expressions.

To conclude, Recaf programs are type-correct when using Statement Virtualization, as long as they generate type-correct Java code. However, in the context of Full Virtualization, compile-time guarantees are overridden as the expressions are fully generic, and therefore, no static assumptions on the expressions can be made.

Runtime Performance. The runtime performance depends on the implementation of the semantics. The base interpreters are admittedly naive, but they serve to illustrate the modularity and reusability enabled by Recaf for building language extensions on top of Java. The Dart-like extensions, reuse the CPS interpreter. As such they are too slow for production use (closure creation to represent the program increases heap allocations). But these examples illustrate the expressiveness of Recaf's embedding method: a very regular syntactic interface (the algebra), may be implemented by an interpreter that completely redefines control flow evaluation. On the other hand, the constraint embedding case study only uses the restricted method syntax to build up constraint objects for a solver. Solving the constraints does not incur any additional overhead, the DSL is used merely for construction of the constraint objects.

Further research is still needed, however, to remove interpretive overhead in order to make extensions of Java practical. One direction would be to investigate "compiler algebras", which generate byte code or (even native code) at runtime. Frameworks like ASM [BLC02] and Javassist [Chi98] could be used to dynamically generate bytecode, which could then be executed by the `Method` method.

2.8 Related Work

Syntactic and semantic extensibility of programming languages has received a lot of attention in literature, historically going back to Landin's, "Next 700 Programming Languages" [Lan66]. In this section we focus on work that is related to Recaf from the perspective of semantic language virtualization, languages as libraries, and semantic language customization.

Language Virtualization. Language virtualization allows the programmer to redefine the meaning of existing constructs and define new ones for a programming language. Well-known examples include LINQ [MBB06]) that offers query syntax over custom data types, Haskell's **`do`**-notation for defining custom monadic evaluation sequences, and Scala's **`for`**-comprehensions. Scala Virtualized [RAM+12] and

Lightweight Modular Staging (LMS) [RO10; RO12] are frameworks to redefine the meaning of almost the complete Scala language. However, these frameworks rely on the advanced typing mechanisms of Scala (higher-kinded types and implicits) to implement concrete implementations of DSL embeddings. Additionally, compared to Scala, Java does not have support for delimited continuations so we rely on a CPS interpretation to mitigate that. Recaf scopes virtualization to methods, a choice motivated by the statement-oriented flavor of the Java syntax, and inspired by how the `async`, `sync*` and `async*` modifiers are scoped in Dart [MMB15] and `async` in C# [BRM+12].

Another related approach is the work on F#'s computation expressions [PS14] which allow the embedding of various computational idioms via the definition of concrete computation builder objects, similar to our Object Algebras. The F# compiler desugars ordinary F# expressions to calls into the factory, in an analogous way to the transformation employed by Recaf. Note that the semantic virtualization capabilities offered by computation expressions are scoped to the expression level. Both, Recaf and F# support custom operators, however in F# they are not supported in control flow statements [Sym12]. Carette et al. [CKS09] construct CPS interpreters among others. In Recaf we use the same approach to enable control-flow manipulation extensions.

Languages as Libraries. Recaf is a framework for library-based language extension. The core idea of "languages as libraries" is that embedded languages or language extensions exist at the same level as ordinary user code. This is different, for instance, from extensible compilers (e.g., [NCM03]) where language extensions are defined at the meta level.

The SugarJ system [ERK+11] supports language extension as a library, where new syntactic constructs are transformed to plain Java code by specifying custom rewrite rules. The Racket system supports a similar style of defining libary-based languages by transformation, leveraging a powerful macro facility and module system [TSC+11]. A significant difference to Recaf is that in both SugarJ and Racket, the extension developer writes the transformations herself, whereas in Recaf the transformation is generic and provided by the framework.

Language Customization. Language extension is only one of the use cases supported by Recaf. Recaf can also be used to instrument or modify the base semantics of Java. Consequently, Recaf can be seen as specific kind of meta object protocol [KDB91], where the programmer can customize the dynamic semantics of a programming language, from within the host language itself. OpenC++ [Chi95] introduced such a feature for C++, allowing the customization of member access, method invocation and object creation.

2.9 Conclusion

In this chapter we have presented Recaf, a lightweight tool to extend both the syntax and the semantics of Java methods just by writing Java code. Recaf is based on two techniques. First, the Java syntax is generalized to allow custom language constructs that follow the pattern of the regular control-flow statements of Java. Second, a generic source-to-source transformation translates the source code of methods into calls to factory objects that represent the desired semantics. Furthermore, formulating these semantic factories as Object Algebras enables powerful patterns for composing semantic definitions and language extensions.

3 Tracing Program Transformations with String Origins

Program transformations, such as the one employed in Recaf, play an important role in domain-specific languages and model-driven development. Tracing the execution of such transformations has well-known benefits for debugging, visualization and error reporting. In this chapter, we introduce string origins, *a lightweight, generic and portable technique to establish a tracing relation between the textual fragments in the input and output of a program transformation. We discuss the semantics and the implementation of string origins using the Rascal meta programming language as an example. We illustrate the utility of string origins by presenting data structures and operations for tracing generated code, implementing protected regions, performing name resolution and fixing inadvertent name capture in generated code.*

This chapter is based on the following published article: P. Inostroza, T. van der Storm, and S. Erdweg. "Tracing Program Transformations with String Origins". In: *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings*. Ed. by D. D. Ruscio and D. Varró. Vol. 8568. Lecture Notes in Computer Science. Springer, 2014, pp. 154–169.

3.1 Introduction

Program transformations play an important role in domain-specific language (DSL) engineering and model-driven development (MDD). In particular, DSL compilers are often structured as a sequence of transformations, starting with an input program and eventually generating code. It is well-known that origin tracking [vDKT93] and model traceability [ANR+06; Jou05; ON06; OO07; RPK+08] provide valuable information for debugging, error reporting and visualization.

In this chapter, we focus on traceability for transformations that generate (fragments of) text. We propose *string origins*, a lightweight technique that links each character in the generated text to its origin. A string either originates directly from the input model, occurs as a string literal in the transformation definition, or is synthesized by the transformation (e.g., by string concatenation or substitution). We represent string origins using a combination of unique resource identifiers (URIs) and offset and length values that identify specific text fragments in a resource. We propagate string origins through augmented versions of standard string operators, such that the propagation is fully transparent to transformation writers. In particular, parsing and unparsing retains string origins for text fragments that appear in the AST, such as variable names.

Through applications of string origins we further confirm the usefulness of model traceability by realizing generic solutions to common problems in program-transformation design. First, string origins allow us to link generated elements back to their origin. In Section 3.3.1, we show how this enables the construction of editors with embedded hyperlinks to inspect generated code. Second, we present an example of attaching additional information to generated code via string origins. Section 3.3.2 describes how this enables protected regions in generated code. Third, string origins can be interpreted as unique pointers that identify subterms. In Section 3.3.3, we use the origins of symbolic names (variables, type names, method names, etc.) to implement name resolution. Finally, string origins can be used to systematically replace fragments of the generated code that have the same origin. In Section 3.3.4, we show a generic solution for circumventing accidental variable capture (hygiene) by systematic renaming of generated names.

In Section 3.4, we discuss the implementation of string origins in the context of Rascal [KSV09]. Overall, we found that string origins have a number of important benefits that can improve the design of program transformations and transformation engines:

- Totality: Unlike existing work in origin tracking and model traceability [ON06], string origins induce an origin relation which is total. That is, the origin relation maps every character in the output text of a transformation back to its origin.

- Portability: Since the origin relation is based on string values and string operations instead of inferred from transformation code, the structure or style of the transformation language is largely irrelevant. As a result, string origins are portable across transformation systems, transformation styles, and technological spaces. Even in the case of graphical modeling languages, embedded strings (e.g., names, labels, etc.) could be annotated with their location in the serialization format used to store such models.
- Universality: String origins are independent of the source or target language, since they only apply to the primitive type string. In particular, origin propagation is independent of the AST structure or meta model.
- Extensibility: String origins are automatically propagated as annotations of substrings. As such, string origins can serve as general carriers of additional, domain-specific information. Marking certain subsstrings as protected (Section 3.3.2) is an example of this.
- Non-invasiveness: Transformation languages that support string manipulation during program transformation can support string origins by modifying the internal representation of strings, without changing the programming interface of strings. The only visible change is at input boundaries where strings are constructed.

We have implemented string origins as an experimental feature of Rascal, a meta programming language for source code analysis and transformation [KSV09]. The applications and example code presented in this chapter have all been prototyped in Rascal. The full code of the examples can be found online at `https://github.com/cwi-swat/string-origins`.

3.2 String Origins

We illustrate the basic idea of string origins in Figure 3.1. The code in the middle shows a simple transformation which converts name and email address specifications to the VCARD format. Arrows and shading indicate the origin relation. The white-on-black substrings in the output are introduced by the transformation; their origins point to the string template in the transformation code in the middle. In contrast, the substrings with gray backgrounds (name and email) are copied over from the input to the output, and hence point back to the input model. The substrings in the result are partitioned according to the origin relation: a fragment originates in either the input, or the transformation.

Note that the transformation processes the input by splitting the string. It is important to realize that this does not break the origin relation, but instead makes it more fine-grained: the output fragments “Pablo Inostroza” and “pvaldera@cwi.nl” have distinct origins pointers to the exact corresponding substrings in the input.

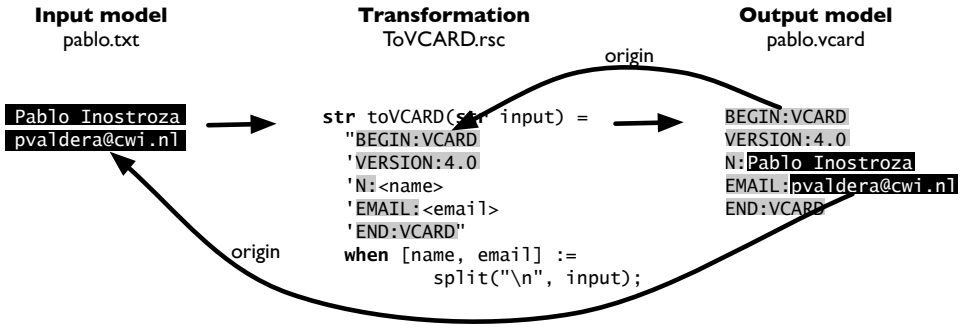


Figure 3.1: Example of a simple Rascal transformation with trace links

3.2.1 Representing String Origins

Many transformations take text files as input and, eventually, produce text files as output. Moreover, the transformations themselves are expressed often as transformation code that is stored in text files as well. String origins exploit this fact by representing origins as *source locations*. Conceptually, a source location is a tuple consisting of a URI identifying a particular resource and an *area* identifying a text fragment within the resource. We represent an area by its start offset and length.

In the context of Rascal, source locations are represented by the built-in **loc** data type. To give an example, |file:///etc/passwd|(0, 50) identifies the first 50 characters in the file /etc/passwd, starting at offset 0. Rascal's source locations also represent begin and end line and column numbers, but for the remainder of this chapter we will abstract from this technical detail. Although source locations are built into Rascal, they are easily implemented in any other transformation system.

The propagation of string origins is transparent: The transformation writer can fully ignore their presence and simply uses standard string operations such as concatenation or substitution. We discuss the details of the propagation in Section 3.4. Here, we want to highlight how to build generic tools on top of origin information. To this end, we provide an API for accessing locations and origins of substrings. First, we provide a function for decomposing a string into its atomic substrings (called chunks):

```
alias Index = rel[loc pos, str chunk];
Index index(str x, loc output);
```

Function `index` constructs an `Index` by collecting the atomic substrings of a string at a given location (e.g., a file path). The type `Index` is defined as a binary relation from the location of a substring to the corresponding chunk. The relation type **rel** is native

in Rascal and is equivalent to a set of tuples. Second, each of the chunks in an `Index` has an associated origin which can be retrieved with the function `origin`.

```
loc origin(str x); // require: x is a chunk
```

For example, we can call `index` on the generated VCARD shown in Fig. 3.1. Assuming the output location is |file:///pablo.vcard|, we get the following index:

```
{<|file:///pablo.vcard|(0,28),  "BEGIN:VCARD\nVERSION:4.0\nN:">,
 <|file:///pablo.vcard|(28,14), "Pablo Inostroza">,
 <|file:///pablo.vcard|(42,7),  "\nEMAIL:">,
 <|file:///pablo.vcard|(49,14), "pvaldera@cwi.nl">,
 <|file:///pablo.vcard|(63,9),  "\nEND:VCARD">}
```

Applying the `origin` function on any of the chunks retrieves the location where that particular chunk of text was introduced. Combining both functions gives us the origin relation, modeled by the `Trace` data type, which relates output locations to their corresponding origins:

```
alias Trace = rel[loc pos, loc org];
Trace trace(str s, loc out) = {<l, origin(chunk)> | <l, chunk> ← index(s, out)}
```

Function `trace` maps function `origin` over all chunks of the index. Considering again the example of Fig. 3.1, the trace relation of the generated VCARD looks as follows:

```
{<|file:///pablo.vcard|(0,28),  |file:///ToVCARD.rsc|(28, 28)>,
 <|file:///pablo.vcard|(28,14), |file:///pablo.txt|(0,14)>,
 <|file:///pablo.vcard|(42,7),  |file:///ToVCARD.rsc|(66, 7)>,
 <|file:///pablo.vcard|(49,14), |file:///pablo.txt|(15,14)>,
 <|file:///pablo.vcard|(63,9),  |file:///ToVCARD.rsc|(86, 9)>}
```

Note that the URIs in the origins distinguishes chunks originating in the input (`pablo.txt`) from chunks introduced by the transformation (`ToVCARD.rsc`). Both the index and trace relations are the stepping stones for the generic tools developed in the subsequent section.

3.2.2 String Origins in M2T and M2M Transformations

The previous example illustrates the use of string origins for text-to-text transformations. However, string origins are also useful in model-to-text and model-to-model transformations. More specifically, when parsing text into an AST, the string fragments that appear as leaves of the AST have string origins attached, pointing to the corresponding text fragment in the input file. Model-to-model transformations preserve the origins of strings copied from the input model and generate new origins for synthesized string fragments. Similarly, unparsing and other model-to-text transformations preserve the origins of strings in the AST. Again, the origin propagation

```
state opened
  close => closed
end
state closed
  open => opened
  lock => locked
end
state locked
  unlock => closed
end
```

(a) An example state machine

```
controller(
  [ ... /* event declarations */ ... ],
  [state("opened"@{|input|(62,6)}, [],
     [transition("close"@{|input|(70,5)},

  "closed"@{|input|(79,6)})]),
   state("closed"@{|input|(100,6)},[],
     [transition("open"@{|input|(108,4)},

  "opened"@{|input|(116,6)}),
      transition("lock"@{|input|(124,4)},

  "locked"@{|input|(132,6)})]),
    state("locked"@{|input|(152,6)},[],

  [transition("unlock"@{|input|(160,6)},

  "closed"@{|input|(170,6)})])])
```

(b) Parsed AST of the state machine

```
prog([
  fdef("opened"@{|input|(62,6)},[],val(nat(0))),
  fdef("closed"@{|input|(100,6)},[],val(nat(1))),
  fdef("locked"@{|input|(152,6)},[],val(nat(2))),
  ... // dispatch functions per state
  fdef( // main dispatch
    "main"@{|meta|(1280,13)},
    ["state"@{|meta|(1307,5)},
     "event"@{|meta|(1316,5)}],
    cond(equ(var("state"@{|meta|(1515,5)}),
                 call("opened"@{|input|(62,6)},
  [])),
      call("opened-dispatch"
             @{|meta|(1565,9),|input|(62,6)},
           [var("event"@{|meta|(1583,5)})]),
      cond(equ(var("state"@{|meta|(1515,5)}),

  call("closed"@{|input|(100,6)},[])),
        call("closed-dispatch"
               @{|input|(100,6),|meta|(1565,9)},
             [var("event"@{|meta|(1583,5)})]),
        cond(equ(var("state"@{|meta|(1515,5)}),

  call("locked"@{|input|(152,6)},[])),
         call("locked-dispatch"
                {|meta|(1565,9),|input|(152,6)},
              [var("event"@{|meta|(1583,5)})]),
         val(error("UnsupportedState"
            @{|meta|(1375,16)})))))], [])
```

(c) Generated AST of the compiled state machine

Figure 3.2: The names in the state machine code (a) end up as strings in the AST (b), the origins of which are propagated to the compiled AST (c). State machine input is represented by URI *input*, the transformation definition by URI *meta*.

is transparent to transformation writers, parsing and unparsing because origins are propagated through standard string operators.

Tracing origin information for string fragments in an AST is often useful. For example, variable names typically occur as string fragments in an AST. Figure 3.2 illustrates tracing of variable names in the context of a DSL for state machines. Figure 3.2(a) shows the source code of a state machine. Parsing the state machine produces an abstract syntax tree (AST), which is shown in Figure 3.2(b). Note that all strings in this AST are annotated with their origin, using the pseudo-notation "@". The AST is then translated to an imperative program which is shown in Figure 3.2(c). Some

strings have *input* origins (e.g., "opened"), some are introduced by the transformation and have *meta* origins (e.g., "main"), and some strings have origins in both the input and transformation because of concatenation (e.g., "opened-dispatch").

3.3 Applications of String Origins

3.3.1 Hyperlinking Generated Artifacts

One of the foremost applications of string origins is relating (sub)strings of the output back to the input of a transformation [KRP+12; ON06]. Applications of this information include embedding links back to the source program in generated code, inspectors, debuggers (e.g., using SourceMaps [Sed12]), or translating back errors produced by further transformations (e.g., general-purpose language compiler errors). In this section we show an example of inspecting the result of a program transformation where the output is shown in an editor with embedded hyperlinks to the input or transformation code.

To display hyperlinks for parts of the generated code, the offsets of the chunks in the generated code must be mapped back to the origin associated with each corresponding chunk. Fortunately, the trace relation introduced in Section 3.2.1 contains exactly this information. The hyperlinks are created by finding the location of a click in the `Trace` mapping and moving the focus and cursor to the corresponding editor.

A demonstration of this feature is shown in Fig. 3.3. The screenshot shows three editors in Rascal Eclipse IDE. The first column shows generated Java code. The substrings highlighted in red are the substrings originating from the input, a textual model for state machines (shown in the middle). The other substrings (in black) are introduced by the code generator, which is shown in the right column. Clicking anywhere in the first column will take you to the exact location where the clicked substring originated.

3.3.2 Protecting Regions of Generated Code

In many cases, a model-to-text transformation is intended to generate just a partial implementation that has to be completed by the programmer. Normally, if the transformation is re-run, the manually edited code is overwritten. In general, this problem is addressed by explicitly marking certain zones of the generated text as *editable*. The MOF Models to Text Standard [Obj08], for instance, introduces the `unprotected` keyword at the transformation level to specify whether a region can be editable by the end programmer or not. Another traditional solution is the generation gap pattern [Fow10], in which the generated code and the code that is expected to be handwritten are related by inheritance. This, however, demands that the generated

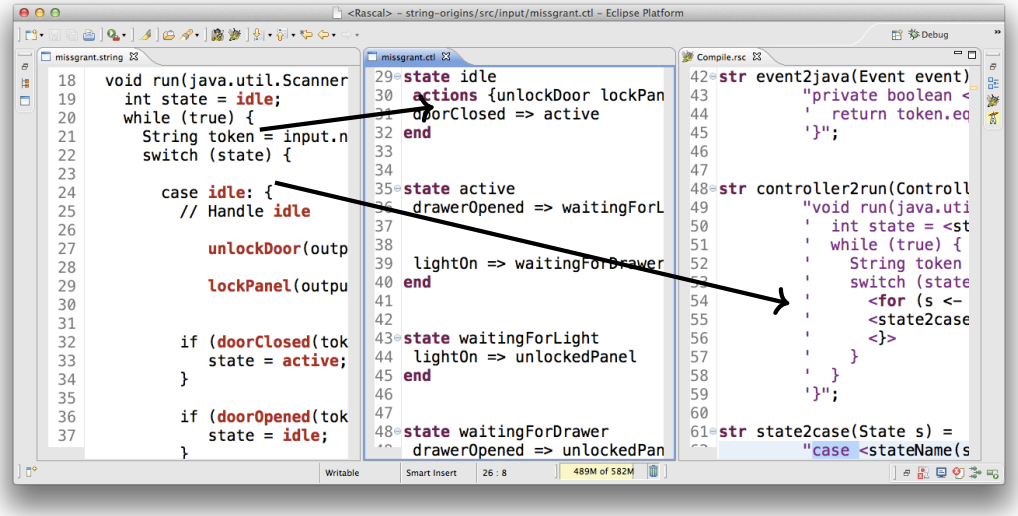


Figure 3.3: Three editors showing (1) generated code with embedded hyperlinks (2) the input state machine model and (3) the transformation code. Fragments of the generated code that originate from the input are in bold red.

code is written in a language that features inheritance and also that the writer of the transformation encodes this design pattern in the transformation.

String origins allow us to tackle this problem in a language and transformation design agnostic way. Since locations correspond to extended URIs, they can be enriched with meta data in the form of query string parameters. We provide three functions `tagString(key,value)`, `getTagValue(key)` and `isTagged(key)`, as an abstract interface to these query strings. The `tagString` function could be used in a transformation to tag regions of text as editable. For instance, the following code snippet marks a substring as being editable in the code generator for a state machine language:

```
str command2java(Command command) =
  "private void <command.name>(Writer output) {
  '  output.write(\"<command.token>\\n\");
  '  <tagString("// Add more code here", "editable", command.name)>
  '}";
```

The function `tagString` transparently marks the origin of the inserted string (`"// Add more code here"`) to be an `editable` region and names it as the name of the command input to `command2java`.

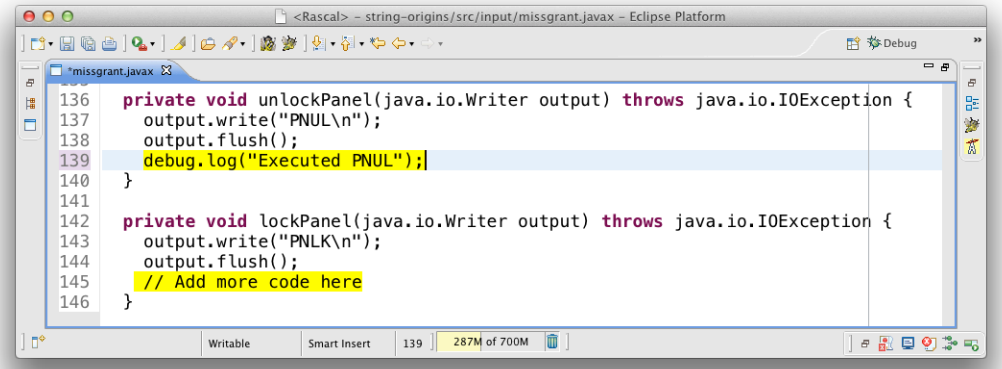


Figure 3.4: Editor featuring highlighted editable regions.

To provide editor support for editable regions, the marked substrings need to be extracted from the generated code. The function `extract` constructs a map from output location to region name using the `index` function introduced in section 3.2.1.

```
alias Regions = map[loc pos, str name];
Regions extract(str s, loc l) =
  (l: getTagValue(x, "editable") | <l, x> ← index(s, l), isTagged(x,
  "editable") );
```

From the index computed on the generated code `s` and the target location `l`, the function `extract` collects all locations which have an associated string value that is tagged as editable. An editor for `s` can then use the locations in the domain of this map to allow changes to the regions identified by the locations. In fact, it maintains another map, this time from region name (range of the result of `extract`) to the contents of each region.

When the code is regenerated, the edited contents of the regions need to be plugged back into the newly generated code, to restore the manual modifications. The function `plug` performs this task:

```
alias Contents = map[str name, str contents];
str plug(str s, loc l, Contents c) = substitute(s, extract(s, l) o c);
```

The `Contents` type captures the edits made in the editable regions. The function `plug` uses a generic substitution function (`substitute`) which receives a map from location to string and performs substitution based on the locations. To obtain this map, `plug`

```
// Event handlers for DSL editors
void onSave(Tree_L1 p.L1) {
  str s = L1toL2(p.L1);
  Regions r = extract(s);
  if (exists(p.regions))
    r = merge(read(p.regions));
  write(p.regions, r);
  write(p.L2, s);
}
```

```
// Event handlers for generated code editors
Tree_L2 onEdit(str p.L2) {
  return decorate(parse(p.L2),
  read(p.regions));
}

void onSave(Tree_L2 p.L2) {
  write(p.regions, lookup(p.L2));
}
```

Figure 3.5: Managing editable regions across edit sections

composes the map returned by `extract` with the contents `c`, where the map composition operator `o` is similar to relational composition.

As a proof of concept, we have added a feature to the Rascal editor framework that uses the presented infrastructure in order to provide consistent editing of generated artifacts with editable areas. When a transformation that produces editable regions is executed, a file with information about the editable offsets is generated as well. When the user opens a generated file, the editor checks if the region information is available. If so, the editor restricts the editing of text just to the regions marked as editable, ensuring that the fixed generated code stays as it is. Fig. 3.4 shows a screenshot of the editor with highlighted editable regions.

3.3.3 Resolving Symbolic Names

Textual DSLs or modeling languages employ symbolic names to encode references, for instance from variables to declarations. As a result, DSL compilers and type checkers require name analysis to resolve references to referenced entities, in fact imposing a graph structure on top of the abstract syntax tree (AST) of the DSL. The names themselves cannot be used as nodes in this graph, since then different occurrences of the same name will be indistinguishable. A solution to this problem is to assign unique labels to each name occurrence in the source code. Since no two names can occupy the same location in the source code, string origins are excellent candidates to play the role of such labels.

Figure 3.6(a) shows the abstract syntax of the state machine language used in Fig. 3.2. Note that states, events and transitions contain strings. Each of these strings will be annotated with an origin by the state machine parser as in Fig. 3.2b. Figure 3.6(b) shows the generic type `Ref` for *reference graphs*: a tuple consisting of the set of all name occurrences (`names`), and a relation mapping uses of names to declarations. The function `resolve` computes a reference graph by first constructing two relations mapping names of states and events to declarations of states and events,

```
data Controller
  = controller(list[Event] events,
               list[State] states);
data State
  = state(str name,
          list[Transition] trans);

data Event
  = event(str name, str token);

data Transition
  = transition(str event, str state)
```

(a) AST data type of state machines

```
alias Ref = tuple[set[loc] names,
                  rel[loc use, loc def] refs];

Ref resolve(Controller ctl) {
  sds = { <x, origin(x)> | state(x, _) ← ctl.states };
  eds = { <x, origin(x)> | event(x, _) ← ctl.events };
  v = range(sds) + range(eds);
  e = { <origin(e),ed>, <origin(s),sd>
            | /transition(e, s) := ctl,
              <e, ed> ← eds, <s, sd> ← sds};
  return <v, e>;
```

(b) Name resolution for state machines

Figure 3.6: Implementing name resolution for state machines

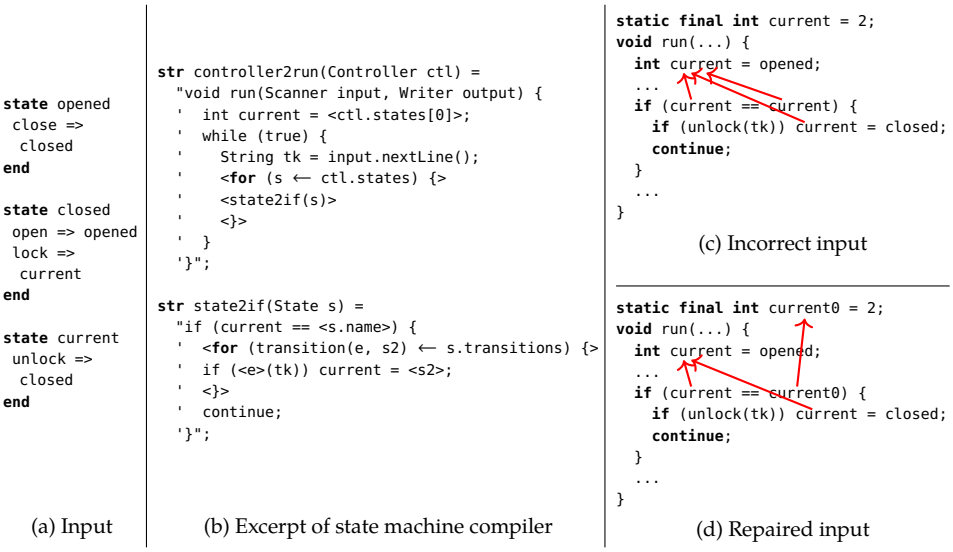
respectively (`sds` resp. `eds`). The last comprehension uses the deep matching feature of Rascal (`/`) to find transitions arbitrarily deep in the controller `ctl`. Each transition then contributes two edges to the relation `e`.

Reference graphs such as returned by `resolve` have numerous generic applications in the context of DSL engineering. For instance, reference graphs can be used to implement jump-to-definition hyperlinking of editors: when the user clicks on the use of a name, the reference graph can be used to find the location of its declaration. Another application is rename refactoring: given a reference graph, and the locations of a name occurrence, it is possible to track other names that reference it or are referenced by it and consistently rename them. Finally, if `Ref` is slightly modified to distinguish uses from declarations in the `names` component, reference graphs can be used to report unbound names or unused declarations.

3.3.4 Enforcing a Same Origin Policy for References

A common problem with code generation is that names used in the input (source names) which pass through a transformation and end up in the output might interact with names introduced by the transformation (introduced names). For instance, the declaration of a name introduced by the transformation might capture a reference to a source name, or vice versa. This is the problem that is traditionally solved in the work on macro hygiene [CR91].

The problem of inadvertent name capture is best illustrated using an example. Figure 3.7(a) shows the simple state machine used earlier in Fig. 3.2(a), but this time the last state is named `current`. The code generator of state machines – partially shown in Fig. 3.7(b) – introduces another instance of the name `current` to store the current



(a) Input

(b) Excerpt of state machine compiler

(c) Incorrect input

(d) Repaired input

Figure 3.7: Example of repairing name capture: the input (a) contains the name `current`, but this name is introduced in the transformation as well (b). Consequently, the introduced variable in the output shadows the constant declaration (c). The `fix` function renames all occurrences of `current` originating in the input to `current0` so that capture is avoided (d). The arrows in (c) and (d) link variable uses to their declarations.

state in the generated Java implementation of the state machine. As a result, the declaration of this `current` captures the reference to the state constant `current`.

The reference arrows in Fig. 3.7(c) show that both `current` variables in the if-condition are bound by the `current` state variable declaration. However, the right-hand side of the equals expression should be bound by the constant declaration corresponding to the state `current`. Moreover, the Java compiler will not signal an error: even though the code is statically correct, it is still wrong.

To avoid name capture, the algorithm described below renames the source names in the output of a transformation if they are also in the set of non-source names. The result can be seen in Fig. 3.7(d): the source occurrences of `current` are renamed to `current0`, and inadvertent capture is avoided. Effectively, the technique amounts to enforcing a same origin policy for names, similar to how a same origin policy avoids cross-site scripting attacks in Web application security[1]: names originating from different artifacts should not reference each other.

In [EvdSD14] the authors showed how string origins proved to be instrumental in automatically repairing the problem of unintended variable capture. In this section

[1] `http://en.wikipedia.org/wiki/Same-origin_policy`

we present a technique that is simpler but also more conservative: it might rename more identifiers than is actually needed. Whereas the method of [EvdSD14] is parameterized in the scoping rules of both source and target language, the technique of this section is language agnostic, and does not require name analysis of the source or target language.

The key observation is that whenever name capture occurs it involves a source name and a name introduced by the transformation. This difference is reflected in the origins of the name occurrences in the output: the origins' source locations will have different URIs. The same origin policy then requires that for every reference in the generated code from x to y, both x and y originate from the input or neither. The same origin policy is enforced by ensuring that the set of source names is disjoint from the set of names introduced by the transformation. This can be realized by consistently renaming source names in the generated code when they collide with non-source names.

To formalize the same origin policy, let $t = f(s)$ be the result of some transformation f on input program s, inducing a trace relation $\tau \in \mathit{Trace}$, and let $G_s = \langle V_s, E_s \rangle$, $G_t = \langle V_t, E_t \rangle$ be the reference graphs of the source s and target t, respectively. The same origin policy then requires that

$$\forall \langle l_1, l_2 \rangle \in E_t, \langle l_1, o_1 \rangle \in \tau, \langle l_2, o_2 \rangle \in \tau \ : \ o_1 \in V_s \Leftrightarrow o_2 \in V_s$$

To enforce the same origin policy, one more assumption on reference graphs is needed, namely that the locations in every reference edge point to the same textual name. In other words: every use is bound by a declaration with the same name. For instance, the reference edges drawn in Fig. 3.7c and Fig. 3.7d satisfy this invariant since variable uses l_1, l_2, l_3 point to occurrences of the name `current`, which is also the name used in the declaration l_0.

If we assume that the same name invariant is true for E_t, then the same origin policy is satisfied if the set of source names is disjoint from the set of names introduced by the transformation. The same name invariant ensures that for every $\langle l_1, l_2 \rangle \in E_t$, we have that l_1 and l_2 point to the same name. Consequently, it is not possible that one name originates from the input (e.g., through o_1) but the other does not (e.g., through o_2) because that would contradict disjointness of names.

The code for restoring disjointness is shown in Fig. 3.8. The function `fix` has three parameters: the generated code `gen`, the index `names` capturing the names occurring in `gen`, and a source location identifying the input program `inp`. The latter is used by the predicate `isSrc` to determine whether a name `x` is a source name by checking if the path in the origin of `x` is the input path.

The **`for`**-loop iterates over the index `names` that represents all names in the generated string `gen`. If such a name `x` originates in the source and is also used as an `other` name, an entry is created in the substitution `subst`, mapping location `l` to a new name. The

```
str fix(str gen, Index names, loc inp) {
  bool isSrc(str x) = origin(x).path == inp.path;
  set[str] other = { x | <_, x> ⟵ names, !isSrc(x) };
  set[str] allNames = { x | <_, x> ⟵ names };
  map[loc,str] subst = ();
  map[str,str] renaming = ();
  for (<l, x> ⟵ names, isSrc(x), x in other) {
    if (x notin renaming) {
      <y, allNames> = fresh(x, allNames);
      renaming[x] = y;
    }
    subst[l] = renaming[x];
  }
  return substitute(gen, subst);
}
```

Figure 3.8: Restoring disjointness by fixing source names.

new name is retrieved from the `renaming` map which records which source names should be renamed to which new name. The function `fresh` produces a name that is not used anywhere (i.e., it is not in `allNames`). The variable `allNames` is updated by `fresh` to ensure that consecutive renames do not introduce new problems.

Note that `fix` could also be parameterized with an additional set of external names which might capture or be captured by source names. External names could include the reserved identifiers (keywords) of the target language or (global) names that are always in scope (e.g., everything in java.lang). The only required change is to add the external names to `other`.

3.4 Implementation

The implementation of string origins requires changes to the internal representation of strings used by the transformation engine. In this section we discuss the implementation of string origins in Rascal.

As Rascal is implemented in Java, we have implemented string origins in Java as well. Rascal string values (of type **`str`**) are internally represented as objects conforming to the interface `IString`. We have reimplemented this interface to support string origins, changing only the internal representation. Instances of `IString` are constructed through a factory method `IString string(java.lang.String)` in the Rascal factory interface for creating values (`IValueFactory`).

To ensure that the propagation of string origins is complete, every created string now needs a location to capture its origin. We have extended `IValueFactory` with

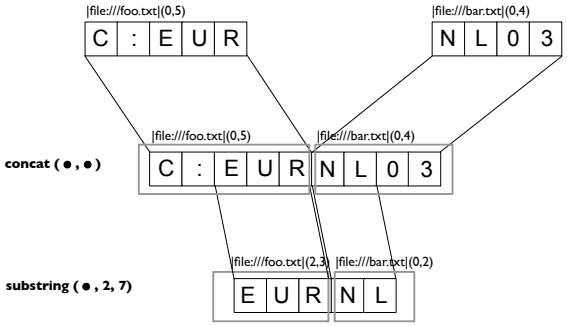


Figure 3.9: The `concat` and `substring` operations defined on origin strings

another factory method IString string(java.lang.String, ISourceLocation) to support this. Calls to the original `string(...)` method were changed to the new one, everywhere in the Rascal implementation. The locations where changes have been made correspond to the following three categories:

- Input: any function that reads a resource into a string must be modified to install origins on the result. In Rascal, these are built-in library functions like `readFile(loc)`, `readLines(loc)`, `parse(loc)`, etc.
- String literals: constant string values that are created as part of a Rascal program get the origin of the string literal in the Rascal file. Whenever a string literal is evaluated, its location is looked up in its AST and passed to the factory method. This category also covers interpolated string templates.
- Conversions: converting a value to a string in Rascal is achieved through string interpolation. For instance, "<x>" returns the string representation of `x`. If `x` evaluates to a string, the result of the conversion is that string itself (including origin); otherwise, the newly created string gets the locations of the expression `x` in the Rascal source.

String origins are propagated through all string operations. As a result, all operations provided in the `IString` interface have been reimplemented. The two most important operations are `concat` and `substring`. Their semantics is illustrated in Fig. 3.9. The top two string values are annotated with source locations |file:///foo.txt|(0,5) and |file:///bar.txt|(0,5). Concatenating both strings (middle row) produces a new, composite string, where the original arguments to `concat` are still distinguishable, and have their respective origins. Finally, the substring operation computes a new composite string with the origin of each component updated to reflect which part of the original input is covered. Besides `concat` and `substring`, all common string

operations such as `indexOf`, `replaceAll`, `split` etc. can be defined on strings with origins, with full propagation.

Internally, Rascal strings with origins are represented as binary trees. A string is either a *chunk* object which has a source location attached to it, or it is a *concat* object which represents two concatenated strings. A string represented as a binary tree can be flattened to a list containing elements with a string value and source location for each chunk object at the leaves. This list is the basis for the functions `index` and `origin` introduced in Section 3.2.1.

Although in our experience the performance penalty introduced by representing strings as binary trees is acceptable in practice, further benchmarking is needed to assess the overall impact. In particular, it will be interesting to see how the choice of representation affects different use cases. For instance, when generating code, concatenation is one of the most frequently executed string operations. The binary tree representation is optimized for that: concatenation is an $O(1)$ operation. On the other hand, analyzing strings (e.g., substring, parsing, matching) is much more expensive if a string is a binary tree. But then again, the penalty will be most significant if these operations apply to strings resulting from concatenation in the first place. We consider investigating these and other aspects of performance an important area for further research.

3.5 Related Work

String origins are related to previous work in origin tracking, taint propagation and model traceability in model-driven engineering. Below we discuss each of these areas in turn.

Origin tracking. The main inspiration of string origins is origin tracking [vDKT93]. In the context of term-rewriting systems, this technique relates intermediate subterms matched and constructed during a rewriting process. Origin tracking was proposed as a technique to construct automatic error reporting, generic debuggers and visualization in program transformation scenarios. String origins are related in that the result is a relation between input and output terms. However, for string origins, only string valued elements are in this relation. Furthermore, the origin relation of [vDKT93] is derived from analyzing rewrite rules. As a result the transformation writer is restricted to this paradigm. With string origins, a transformation can be arbitrary code.

Taint propagation. In Web applications, untrusted user input might potentially end up as part of a database query, a command-line script execution or web page. Malicious input could thus compromise the system security in the form of code injection attacks.

Taint propagation [HCF05] is a mechanism to raise the level of security in such systems by tracking potentially risky strings at runtime. It consists of three main phases: mark certain *sources* of strings as tainted, propagating taint markers across the execution of the program, and disallowing the use of tainted strings at certain specific points called *sinks*. The propagation is achieved by annotating the string values themselves and making sure that string operations propagate taintedness.

Although in general the taint information is coarse-grained: any string that is computed from any number of tainted strings is tainted as well. A finer granularity is employed in character-based taint propagation [CW09]. String origins are very similar to this approach in that the origin is known for each character in a string. On the other hand, string origins can be considered more general, because origins capture more information than just taintedness. In fact, taint propagation could easily be realized using string origins by considering certain input locations as tainted.

In [DMS+10], the authors present an application of taint propagation to the domain of model-to-text transformations, specifically, to support debugging of failures introduced in a transformation. Their approach consists in instrumenting the transformation in order to add so-called *tainted marks* to each identifiable element of the input. On the other hand, the user of the transformation has to identify erroneous sections in the output. Since the taints from the input are consistently propagated by the instrumented transformation, it is possible to relate the errors in the output to specific elements of the input. In this work, the input is an XML document and the transformation, an XSLT file. The granularity of this technique is at the level of XML nodes, which provides quite precise information for the error tracking analysis.

Traceability in model-driven engineering. In model-driven engineering, models are refined through transformations to produce executable artifacts. In [ANR+06], the authors argue for the need for automatic generation of trace information in such a setting. Several endeavors towards this goal have been reported in the context of different model transformation systems, such as ATL, MOF, and Epsilon.

For instance, ATL transformations can be manually enriched with traceability rules that conform to a traceability metamodel [Jou05]. Besides the target models, the enriched transformations will also automatically produce trace models when executed. In order to avoid the manual work of adding these specifications to existing transformations, the authors present a technique for automatically weaving the trace rules into the transformation. Unlike string origins, this approach relies on the structure of the ATL rules to derive the trace links, and such links just relate a subset of the elements in the target model to certain elements in the source model, but not to the transformation itself.

Another approach to address traceability is the MOF Models to Text Transformation Language standard [Obj08]. In this specification, transformations can be decorated

with a `trace` annotation so when the transformation is executed, a relation between its output and its input is constructed. As in the case of [Jou05], the transformation conveys the traceability information explicitly. To overcome this, [ON06] and [OO07] introduce an alternative technique for managing traceability in MOFScript, a language for defining model to text transformations based on the MOF standard. In this case, "any reference to a model element that is used to produce text output results in a trace between that element and the target file". Like string origins, this technique provides implicit propagation and fine-grained tracing. However, no relation between the output and the text fragments coming from the transformation is created. Just as in the case of ATL, MOFScript depends on the structure of the rules to analyze the transformation and generate trace information.

Finally, The Epsilon Generation Language (EGL) is a model-to-text transformation language defined at the core of the Epsilon Platform [RPK+08]. EGL provides an API to construct a transformation trace. However, this API is coarse-grained (file-level).

3.6 Conclusion

String origins identify the exact origin of a fragment of text. By annotating string values with their origins, the origins are automatically propagated through program transformations, independent of transformation style or paradigm. The result is that for every string valued element in the output of a transformation, we know where it came from, originating in the input program or introduced by the transformation itself.

String origins have diverse applications. They address traditional model traceability concerns by linking output elements to where they were introduced. We have shown two applications in this space, namely hyperlinked editors for generated code and protected regions. Moreover, string origins can be used to uniquely identify sub terms, which is instrumental for implementing name resolution, rename refactoring, jump-to-definition services and error marking. Finally, we have shown that by distinguishing source names from introduced names, accidental name capture in generated code can be avoided in a reliable and language agnostic way.

The implementation of string origins is simple and independent of any specific meta-model, transformation engine or technological space. Any transformation system or programming language that manipulates string values during execution can support string origins by changing the internal representation of strings. The standard programming interface on strings remains the same. As a result, code that manipulates strings does not have to be changed, except for the code that creates strings in the first place. Although conceptually simple, we have shown that string origins, nevertheless, provide a powerful tool to improve the understandability and reliability of program transformations.

4 Modular Interpreters with Implicit Context Propagation

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. As Recaf showcased, object algebras are a suitable technique to encode modular interpreters, which in the case of Recaf correspond to modular Java (and extended Java) interpreters.

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 chapter, 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 retro-actively 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,

The content of this chapter was first published at the GPCE2015 conference, and later extended to a COMLAN journal publication. This chapter is based on the latter: P. Inostroza and T. van der Storm. "Modular interpreters with Implicit Context Propagation". In: *Computer Languages, Systems & Structures* 48 (2017). Special Issue on the 14th International Conference on Generative Programming: Concepts & Experiences (GPCE'15), pp. 39–67.

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 [FM89] benchmark program in Javascript on top of a modular implementation of LambdaJS [GSK10], 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.

4.1 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) [War94], 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 [OC12] are a design pattern that supports type-safe extensibility of both abstract syntax and interpretations in mainstream, object-oriented (OO) languages. Using effect handlings, the abstract syntax of a language fragment is defined 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 $() \to \mathit{Val}$ as semantic domain, whereas evaluation of binding expressions requires an environment and, hence, might be expressed in terms of the type $\mathit{Env} \to \mathit{Val}$. In this case, the components cannot be composed, even though they are considered to represent the very same interpretation, namely *evaluation*.

In this chapter we resolve such incompatibilities for object algebras defined over function types using implicit context propagation. An algebra defined over a function type $T_0 \times \ldots \times T_n \to U$ is *lifted* to a new algebra over type $T_0 \times \ldots \times T_i \times S \times T_{i+1} \times \ldots \times T_n \to U$.

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 [Ora15a]. 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 presented throughout this chapter 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 4.3).
- 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 4.4).
- 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 4.5).
- 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 4.6).
- The techniques are furthermore illustrated using an extremely modular implementation of Featherweight Java with state [FKF98; IPW99]. This allows us to derive 127 hypothetical variants of the language, out of 7 given language fragments (Section 4.7).
- Interpreter lifting introduces additional runtime overhead. We investigate this overhead empirically by running the DeltaBlue [FM89] benchmark in Javascript on top of a modular implementation of LambdaJS (λ_{JS}) [GSK10]. 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 4.8).

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#.

4.2 Background

4.2.1 Problem Overview

Table 4.1 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.

The first style (left column) captures the traditional OO extension where a new AST class for variables (`Var`) is added. The extension is successful, since the base language anticipates the use of the environment. Unfortunately, the anticipated context parameter (`env`) is not used at all in the base language. Furthermore, anticipating additional context parameters, such as stores, leads to more unnecessary pollution of the evaluation interface in the base language. The main drawback of this style is that it breaks open extensibility. At the moment of writing the base language implementation, the number of context parameters is fixed, and no further extensions are possible without a full rewrite.

The second style (right column) does not anticipate the use of an environment, and the implementation of `Lit` is exactly as one would desire. No environment is used, and so it is not referenced either. To allow the recursive evaluation of expressions in the extension, however, the abstract interface `Exp` needs to be replaced to require an environment-consuming `eval`. Consequently, the full logic of `Lit` evaluation needs to be reimplemented in the extension as `Lit2`. If more context parameters are needed later, the extended classes need to be reimplemented yet again. In fact, in this style, there is no reuse whatsoever.

To summarize, the traditional OO style of writing an interpreter supports extension of syntax (data variants), but only if the evaluation signatures are the same. As a result, any context parameters that might be needed in future extensions have to be anticipated in advance to realize modular extension. In the next section we reframe

[1] All code examples are in Scala [OAC+14] (`http://www.scala-lang.org`). We extensively use Scala traits, 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 the specification of some implicit conversions for readability.

Anticipate

Base Language

```
trait Exp { def eval(env: Env): Val }

class Lit(n: Int) extends Exp {
  def eval(env: Env) = n
}

class Add(l: Exp, r: Exp)
    extends Exp {
  def eval(env: Env) =
    l.eval(env) + r.eval(env)
}
```

Extended Language

```
class Var(x: String) extends Exp {
  def eval(env: Env) = env(x)
}
```

Duplicate

Base Language

```
trait Exp { def eval: Int }

class Lit(n: Int) extends Exp {
  def eval = n
}

class Add(l: Exp, r: Exp)
    extends Exp {
  def eval = l.eval + r.eval
}
```

Extended Language

```
trait Exp2 {
  def eval(env: Env): Val
}

class Lit2(n: Int) extends Exp2 {
  def eval(env: Env) = n
}

class Add2(l: Exp2, r: Exp2)
    extends Exp2 {
  def eval(env: Env)
    = l.eval(env) + r.eval(env)
}

class Var(x: String) extends Exp2 {
  def eval(env: Env) = env(x)
}
```

Table 4.1: Two attempts at adding variable expressions to a language of addition and literal expressions. On the left, the `Lit` and `Add` classes anticipate the use of an environment, without actually using it. On the right, the semantics of `Lit` and `Add` need to be duplicated.

the example language fragments in the object algebras [OC12] style, providing the essential ingredient to solve the problem using implicit context propagation.

4.2.2 Object Algebras

Using object algebras the abstract syntax of a language is defined as a generic factory interface. For instance, the base language abstract syntax of Table 4.1 is defined as the following trait:

```
trait Arith[E] {
  def add(l: E, r: E): E
  def lit(n: Int): E
}
```

Because the trait `Arith` is generic, implementations of the interface must choose a concrete semantic type of `Arith` expressions. In abstract algebra parlance, the factory interface corresponds to an algebraic *signature*, the generic type `E` is a syntactic *sort*, and implementations of the interface are *algebras* binding the generic sort to a concrete *carrier type*. Carrier types can be any type supported by the host language (i.e. Scala), but in this text we only consider function types.

An evaluation algebra for the `Arith` language could be implemented as follows:

```
type Ev = () => Val

trait EvArith extends Arith[Ev] {
  def add(l: Ev, r: Ev)
    = () => IntVal(l() + r())

  def lit(n: Int)
    = () => IntVal(n)
}
```

The type alias `Ev` defines a carrier type consisting of nullary functions returning a value of type `Val`. Terms over the algebra are constructed by invoking the methods of the algebra:

```
def onePlusTwo[E](alg: Arith[E]): E = alg.add(alg.lit(1), alg.lit(2))

val eval = onePlusTwo(new EvArith {})
println(eval()) // => 3
```

The generic function `onePlusTwo` accepts an algebra of type `Arith` and constructs a term over it. Invoking this function with the evaluation algebra `EvArith` gives an object of type `Ev` which can be used to evaluate the expression `add(lit(1), lit(2))`.

Now let us extend `Arith` with variable expressions, as was attempted in Table 4.1. First the abstract syntax is defined using a generic trait:

```
trait Var[E] { def vari(x: String): E }
```

The syntax for both fragments can be combined using trait inheritance:

```
trait ArithWithVar[E] extends Arith[E] with Var[E]
```

For evaluating variables, we can implement the interface over a carrier type `EvE` which accepts an environment:

```
type EvE = Env => Val
trait EvVar extends Var[EvE] {
  def vari(x: String) = env => env(x)
}
```

Unfortunately, this trait cannot be composed with `EvArith` because the carrier types are different: `EvArith` is defined over `Ev` whereas `EvVar` is defined over `EvE`. In order to compose the two syntactic interfaces, both carrier types have to be the same. In this case, however, the evaluation semantics of the language fragments require different context information, which prevents the components from being combined. We actually observe the same problem as shown in Table 4.1!

Fortunately, object algebras also support modular extension with new operations. This means that it is possible to modularly define a trait for a different interpretation of the same syntax:

```
trait EvEArith extends Arith[EvE] {
  ...
}
```

This trait defines arithmetic expressions over the carrier type `EvE` instead of `Ev`. Internally this trait will delegate to the original `EvArith` which was defined over the type `Ev`. In the next section we describe this pattern in more detail.

4.3 Implicit Context Propagation

We have seen how the incompatibility between object algebras defined over different function types precludes extensibility. In this section we introduce implicit context propagation as a technique to overcome this problem, by first extending the `Arith` language to support variable binding, and then generalizing the pattern to support the propagation of other kinds of the context information.

4.3.1 Adding Environments to Arithmetic Expressions

The language fragment of expressions that require environments is shown in Figure 4.1. The `Binding` language defines four constructs: `lambda` (functions), `vari` (variables), `apply` (function application) and `let` (binding). The carrier type is `EvE`, a function from

```
trait Binding[E] {
  def lambda(x:Str, b:E): E
  def vari(x:Str): E
  def apply(e1:E, e2:E): E
  def let(x:Str, e:E, b:E): E
}

type EvE = Env => Val
type Env = immutable.Map[Str,Val]

class Clos(x:Str,b:EvE,e:Env) extends Val {
  def apply(v: Val): Val = b(e + (x -> v))
}
```

```
trait EvEBinding extends Binding[EvE] {
  def lambda(x: Str, b: EvE)
   = env => new Clos(x, b, env)

  def vari(x: Str): EvE = env => env(x)

  def apply(e1: EvE, e2: EvE)
   = env => e1(env).apply(e2(env))

  def let(x:Str, e:EvE, b:EvE)
   = env => b(env + (x -> e(env)))
}
```

Figure 4.1: A language fragment with binding constructs

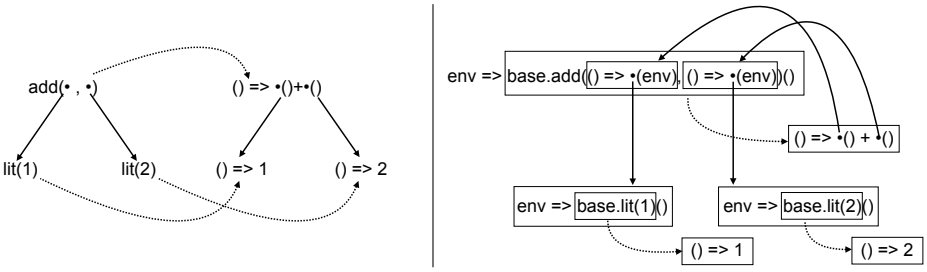


Figure 4.2: The left column shows how the expression `add(lit(1), lit(2))` is mapped to its denotation by `EvArith`; the nodes in the tree are of type `Ev` (`() => Val`). On the right, the result of lifting the denotation produced by `EvArith` to the type `EvE` (`Env => Val`) to propagate environments. The dotted arrows indicate evaluation of Scala expressions; the solid arrows represent references.

environments to values. To support lambdas, the `Val` domain is extended with closures (`Clos`). The interpreter on the right evaluates `lambda`s to closures. Variables are looked up in the environment. Function application expects that the first argument evaluates to a closure and applies it to the value of the second argument. Finally, `let` evaluates its third argument in the environment extended with a variable binding.

We now discuss the implementation of the environment-passing interpreter for the `Arith` language using implicit context propagation. As described in Section 4.2, two object algebra interpreters can be combined if they are defined over the same carrier type. In this case, this means that `EvArith` needs to be lifted to an `EvEArith` which is defined over the carrier type `EvE`, i.e., `Env => Val`:

```
trait EvEArith extends Arith[EvE] {
  private val base = new EvArith {}
```

```
  def add(l: EvE, r: EvE): EvE = env => base.add(() => l(env), () => r(env))()

  def lit(n: Int): EvE = env => base.lit(n)()
}
```

Instead of reimplementing the semantics for the arithmetic operations, the code for each variant delegates to the `base` field initialized with `EvArith`. The interpreter `EvEArith` shows the actual propagation of the environment in the method for `add`. Invoking `add` on the base algebra requires passing in arguments of type `Ev`. This is achieved with the inline anonymous functions. Each of these closures calls the actual arguments of type `EvE` (`l` and `r`). Since both these arguments expect an environment, we pass in the original `env` that corresponds to the argument of the closure denoted by `add`.

In order to visualize lifting, Figure 4.2 shows the evaluation of `add(lit(1), lit(2))` over `EvArith` (left) and over the lifted algebra `EvEArith`. On the left the result is a tree of closures of type `Ev`. The right shows how each closure is lifted to a closure of type `EvE`. Note that each of the closures in the denotation on the left is also present in the denotation on the right, but that they are connected via intermediate closures on the right.

The two languages can now be combined as follows:

```
trait EvEArithBinding extends EvEArith with EvEBinding
```

The following client code shows how to create terms over this language:

```
def makeLambda[E](alg: Arith[E] with Binding[E]): E = {
  import alg._
  lambda("x", add(lit(1), vari("x")))
}

val term: EvE = makeLambda(new EvEArithBinding {})
```

The method `makeLambda` provides a generic way of creating the example term `lambda("x", add(lit(1), vari("x")))` over any algebra defining arithmetic and binding expressions. Invoking the method with an instance of the combined interpreter EvEArith-Binding creates an object of type `EvE`.

4.3.2 Generating Implicit Context Propagation Code

The general pattern for generating context propagating code is shown in Figure 4.3. The template is written in pseudo-Scala and defines a trait $\texttt{Alg}_{(T,U*)\Rightarrow V}$, implementing the language interface `Alg` over the function type `(T, U*) => V`. The asterisks indicate splicing of formal parameters. For instance, `U*` capture zero or more type parameters

```
trait Alg_{(T,U*)⇒V} extends Alg[(T, U*) => V] {

  val base = new Alg_{U*⇒V} {}

  def C_i(f_1: (T, U*) => V, ... , f_n: (T, U*) => V):
   (T, U*)=> V  =
     (t, u*) => base.C_i((u_1*) => f_1(t, u_1*), ..., (u_n*) => f_n(t, u_n*))(u*)

  ...
}
```

Figure 4.3: Template for generating lifted interpreters that propagate environment-like context parameters.

in the function signature `(T, U*) => V`. The same notation is used on ordinary formal parameters, as shown in the closure returned by constructor method C_i.

As shown in Figure 4.3, the `base` algebra is instantiated with an algebra over function type `U* => V`, which accepts one fewer parameter than the carrier type of $Alg_{(T,U*)⇒V}$. For each constructor, C_i, the lifting code follows the pattern as shown. For presentation purposes, primitive arguments to C_i are omitted, and only arguments of the function types are shown as f_j, for $j \in 1,...,n$.

This template concisely expresses the core mechanism of lifting. Notice, however, that it assumes that the added parameter is prepended at the front of the base signature. A realistic generation scheme would consider permutations of parameters. The macro-based code generator discussed in Section 4.5 supports inserting the parameter anywhere in the list.

4.4 Working with Lifted Interpretations

The example languages we have discussed so far only considered expressions in a purely functional framework. In this section, we discuss how implicit context propagation can be used for introducing delayed evaluation, semantics overriding, mutable parameters to model side-effects, exception handling for non-local control, lifting of continuation-passing style interpreters, many-sorted languages, implementation by desugaring, and interpretations other than dynamic semantics.

4.4.1 Delaying Evaluation

The interpreter for arithmetic expressions introduced in Section 4.2.2 is defined in terms of the carrier type `() => Val`. Accordingly, the algebra produces closures that need to be applied in order to obtain the final result. In this case, however, the

denotations could have simply had the type `Val`, as there is no need for delayed evaluation in this language module.

In a certain sense using the carrier type `() => Val` for arithmetic expressions represents a form of anticipation: it is expected that arithmetic expressions will be used together with expressions that do require delayed evaluation. For instance, defining conditional expressions in an eager host language like Scala requires delayed evaluation, otherwise both branches of the conditional are evaluated. This is clearly not intended, especially in the presence of side effects.

It turns out, however, that lifting can be used to delay evaluation on top of an algebra that computes results eagerly. Consider the following implementation of `Arith` expressions:

```
trait ValArith extends Arith[Val] {
  def add(l: Val, r: Val) = IntVal(l + r)
  def lit(n: Int) = IntVal(n)
}
```

The carrier type is simply `Val` and expression evaluation is "immediate": when expressions are constructed over this algebra, they are actually immediately evaluated without creating any intermediate closures. `ValArith` can now be lifted to produce thunks instead of values, as follows:

```
trait DelayedValArith extends Arith[Ev] {
  private val base = new ValArith {}

  def add(l: Ev, r: Ev): Ev = () => base.add(l(), r())
  def lit(n: Int): Ev = () => base.lit(n)
}
```

This version of arithmetic evaluation corresponds to the original `EvArith` introduced in Section 4.2.2 and can be composed with, for instance, implementations of conditional expressions. Lifting was characterized as interleaving the construction of the expressions on the base algebra with evaluating them. In this case, one can see that construction and evaluation actually coincide.

4.4.2 Overriding Interpretations: Dynamic Scoping

The propagation of environments presented in Section 4.3 obeys lexical scoping rules for all implicitly-propagated parameters. Some context information, however, should not be lexically scoped, but dynamically scoped. Typical examples include the binding of **`self`** or **`this`** in OO languages, dynamic contexts in context-oriented programming [HCN08], or simply dynamically scoped variables [HP01].

Consider the following language fragment for introducing dynamically scoped variables, similar to `fluid-let` in Scheme [Han00]:

```
trait DynLet[E] { def dynlet(x: String, v: E, b: E): E }
```

The construct `dynlet` binds a variable `x` to a value in both the lexical and dynamic environment. The dynamic variable can then be referenced in the scope of `dynlet` using the ordinary `vari` of the `Binding` fragment (cf. Figure 4.1).

As an example of the dynamically scoped propagation, consider the following example term defined over the combination of `Arith`, `Binding`, and `DynLet`. The left column shows the abstract syntax, the right column shows the same program in pseudo concrete syntax:

```
dynlet("x", lit(1),
  let("f", lambda("_", add(vari("x"), lit(1))),
    dynlet("x", lit(2),
      let("z", dynlet("x",
                 lit(3),
                 apply(vari("f"), lit(1))),
        add(vari("z"), vari("x"))))))
```

```
dynlet x = 1 in
  let f = λ _ . x + 1 in
    dynlet x = 2 in
      let z = (dynlet x = 3 in f(1)) in
        z + x
```

This program dynamically binds `x` to 1, in the scope of the `let` which defines `f` as a `lambda` dynamically referring to `x`. The value of `x` thus depends on the dynamic scope when the closure `f` is applied to some argument. Nested within the `let` is another dynamic let (`dynlet`) which overrides the value of `x`. The innermost `let` then defines a variable `z` with the value of applying `f` to 1. This application is itself inside another dynamic let, yet again redefining `x`. So the result of this application will be 4, as the innermost dynamic scope defines `x` to be 3. In the body of the innermost normal `let`, however, the active value of `x` is 2, so the final addition `z + x` evaluates to 6.

The implementation of `dynlet` is straightforward by using an extra parameter of type `Env` representing the dynamic environment.

```
type EvEE = (Env, Env) => Val

trait EvEEDynLet extends DynLet[EvEE] {
  def dynlet(x: String, v: EvEE, b: EvEE)
    = (env, denv) => {
        val y = v(env, denv); b(env + (x -> y), denv + (x -> y))
      }
}
```

Notice that since the static and the dynamic environment both have the same type, the signature of the carrier type makes the interpretation order-dependent. We discuss strategies for disambiguation in these scenarios when presenting automated lifting in Section 4.5.

To combine the lexically scoped `Binding` fragment with the dynamically scoped `DynLet` fragment, `EvEBinding` (cf. Figure 4.1) needs to be lifted so that it propagates the dynamic environment. Implicit propagation can be used to obtain `EvEEBinding`. Unfortunately, the dynamic environment is now inadvertently captured when `lambda` creates the `Clos` object.

To work around this problem, the implementation of `lambda` and `apply` in `EvEEBinding` should be overridden, to support the dynamic environment explicitly:

```
class DClos(x: String, b: EvEE, env: Env) extends Val {
  def apply(denv: Env, v: Val): Val
    = b(env ++ denv + (x -> v) , denv) // denv shadows env
}

trait EvEEBindingDyn extends EvEEBinding {
  override def lambda(x: Str, b: EvEE): EvEE
    = (env, denv) => new DClos(x, b, env)

  override def apply(e1: EvEE, e2: EvEE): EvEE
    = (env, denv) => e1(env, denv).apply(denv, e2(env, denv))
}
```

The closure class `DClos` differs from `Clos` only in the extra `denv` parameter to the `apply` method. The supplied dynamic environment `denv` is added to the captured environment, so that a dynamically scoped variable `x` (introduced by `dynlet`) will shadow a lexically scoped variable `x` (if any).

Although the existing `lambda` and `apply` could not be reused, one could argue that adding dynamically scoped variables to a language is not a proper extension which preserves the semantics of all base constructs. In other words, adding dynamic scoping literally *changes* the semantics of `lambda` and `apply`.

4.4.3 Mutable Parameters: Stores

The previous section showed how the lexical scoping of propagated parameters was circumvented through overriding the semantics of certain language constructs. Another example of context that should not be lexically scoped is a store for modeling side effects. In this case, however, the parameter should also not obey stack discipline as it did for the dynamic environment. Instead, we achieve this by propagating mutable data structures. Consequently, all interpreter definitions will share the same store, even when they are captured when closure objects are created.

Consider a language `Storage` which defines constructs for creating cells (`create`), updating cells (`update`) and inspecting them (`inspect`):

```
trait Storage[E] {
  def create(): E
```

```
  def update(c: E, v: E): E
  def inspect(c: E): E
}
```

The simplest approach to implement an interpreter for such expressions is to use a mutable store as a parameter to the interpreter. For instance, the following type declarations model the store as a mutable `Map` and the interpreter as a function from stores to values:

```
type Sto = mutable.Map[Cell, Val]
type EvS = Sto => Val
```

The interpreter for `Storage` could then be defined as follows:[2]

```
trait EvSStorage extends Storage[EvS] {
  def create() = st => ...
  def update(c: EvS, v: EvS) = st => ...
  def inspect(c: EvS) = st => ...
}
```

To compose the `Arith` language defined in Section 4.2.2 with `Storage`, the `EvArith` interpreter needs to be lifted in order to propagate the store. Since `Sto` is a mutable object, side-effects will be observable even though the propagation follows the style of propagating environments.

Unsurprisingly, perhaps, mutable data structures are an effective way of supporting side-effecting language constructs. It is interesting to contemplate whether it is possible instead to lift interpreters that thread an immutable store through the base evaluation process, without depending on mutation. We have experimented with a scheme that uses a private mutable variable, local to the traits containing the lifted methods.

The following example is a failed attempt at lifting `EvArith` to thread an immutable store (represented by the type `ISto`). Since the store is immutable, the carrier type `EvS2S` takes an `ISto` and produces a tuple containing the return value and the (possibly) updated store.

```
type ISto = immutable.Map[Cell,Val]
type EvS2S = ISto => (Val, ISto)

trait EvS2SArith extends Arith[EvS2S] {
  private val base = new EvArith {}
  private var _st: ISto = _

  def add(l: EvS2S, r: EvS2S)
    = st => { _st = st;
              (base.add(() => {val (v1, s1) = l(_st); _st = s1; v1},
```

[2] For brevity, we have elided the actual, straightforward implementation of the storage constructs.

```
class Fail extends Exception

trait Choice[E] {
  def or(l: E, r: E): E
  def fail: E
}
```

```
trait EvChoice extends Choice[Ev] {
  def or(l: Ev, r: Ev): Ev
    = () => try { l() } catch { case _:Fail => r() }

  def fail(): Ev = () => throw new Fail
}
```

Figure 4.4: Implementing local backtracking with exception handling.

```
                        () => {val (v2, s2) = r(_st); _st = s2; v2})(),
                _st)
        }
    ...
  }
```

At every evaluation step, the private variable `_st` is synchronized with the currently active store returned by sub expressions; since the current value of `_st` is also passed to the subsequent evaluation of sub terms, side effects are effectively threaded through the evaluation.

Unfortunately, this scheme breaks down when two different lifted traits have their own private `_st` field. As a result, expressions only see the side-effects enacted by expressions within the same lifting, but not the side-effects which originate from other lifted traits. It would be possible to share this "current store" using an ambient, global variable, allowing different traits (lifted or not) to synchronize on the same store. Such a global variable, however, compromises the modularity of the components and would complicate the code generation considerably, especially in the presence of multiple store-like context parameters.

Since simulating mutable context information by threading immutable data breaks modularity and modularity is key to our approach, we instead depend on the support for mutable data structures in the host language in order to represent side-effects in the object language.

4.4.4 Exception Handling: Backtracking

Many non-local control-flow language features can be simulated using exception handling. A simple example is shown in Figure 4.4, which contains the definition of a language fragment for (local) backtracking. The `or` construct first tries to evaluate its left argument `l`, and if that fails (i.e., the exception `Fail` is thrown), it evaluates the right argument `r` instead. Note that `EvChoice` does not require any context information and is simply defined over the carrier type `Ev`.

If `EvChoice` is lifted to `EvEChoice` to implicitly propagate environments, the exception handling still provides a faithful model of backtracking, because the environments are simply captured in the closures `l` and `r`. In other words, upon backtracking – when the `Fail` exception is caught – the *original* environment is passed to `r`.

```
trait EvEChoice extends Choice[EvE] {
  private val base = new EvChoice {}

  def or(l: EvE, r: EvE): EvE = env => base.or(() => l(env), () => r(env))()

  def fail() = env => base.fail()()
}
```

For instance, evaluating the following term using this algebra, results in the correct answer (1):

```
  let("x", lit(1), or(let("x", lit(2), fail()), vari("x")))
```

Notice, however, that we face a limitation if we want to compose backtracking with mutable context, e.g., the store. Since the store inherits the mutable semantics from the host language, it is not trivial to customize the interaction with the backtracking behavior. For instance, in order to support transaction-reversing choice we might naively assume that we just need to lift `EvChoice` to `EvSChoice` (an interpreter that accepts the store). Unfortunately, the resulting interpreter has the wrong semantics. It does not rollback the transactions upon failure, because the captured stores are mutated at the level of the host language. An alternative is to override the semantics of the `Choice` interpreter in order to keep track of mutable stores at each choice point (like dynamic let required overriding lambda and apply). This leads to a contrived implementation that works around the limitations of the host language semantics. Moreover, in the case there are interactions with more mutable parameters, all of them must be considered, leading to more overriding. In summary, lifting does not automatically handle the interaction between non-local control flow extensions and extensions that require mutable parameters.

4.4.5 Continuation-Passing style

In the introduction, we argue that one of the benefits of our approach to language modularity is that the semantic components can be written in direct style (as opposed to continuation-passing style or monadic style). However, some language features might in fact require a different formulation of interpreters. For instance, not all non-local control flow features are conveniently expressed using exception handling. One case in point is Scheme's call-with-current-continuation (`callcc`) [ADH+98], which allows arbitrary capturing of the "remainder of a computation" (the continuation).

```
type EvK = (Val => Unit) => Unit

trait EvKArith extends Arith[EvK] {
  def add(l: EvK, r: EvK): EvK = k => l(v1 => r(v2 => k(IntVal(v1+v2))))
  def lit(n: Int): EvK = k => k(IntVal(n))
}
```

Figure 4.5: CPS evaluator for arithmetic expressions.

We show that interpreter lifting can still be applied if all interpreters are coded in continuation-passing style (CPS) [Rey93].

Consider, for instance, CPS evaluators for the `Arith` language shown in Figure 4.5. The carrier type `EvK` is defined as a function from a continuation (a function consuming a value) to the unit type (`Unit` in Scala). CPS interpreters never return a value, but always call the given continuation to continue evaluation. For instance, `add` is defined by a call to the `l` argument, passing a new continuation, which, when called, invokes the `r` argument with yet another continuation. If that continuation is invoked, the original continuation `k` is invoked with the result of the addition. The key aspect of CPS interpreters is that all forms of sequencing are made completely explicit in terms of function composition.

Assuming that we want to propagate an environment through the evaluation of `EvKArith` evaluator in order to combine it with binding expression, then the propagating strategy is the same as the one we have already observed:

```
trait EvEKArith extends Arith[EvEK] {
  private val base = new EvKArith {}

  def add(l: EvEK, r: EvEK): EvEK =
    (e, k) => base.add(k_ => l(e, k_), k_ => r(e, k_))(k)

  def lit(n: Int): EvEK = (e, k) => base.lit(n)(k)
}
```

The base `add` function receives functions of type `EvK` which call the original `l` and `r` propagating the environment `e`.

Note that the current continuation (`k`) acts just like any other context parameter. It is therefore tempting to think that lifting could be used to convert a direct style interpreter into a CPS interpreter. Unfortunately, direct style interpreters depend on the host language for their evaluation strategy. As a result, it is impossible to recover the implicit sequencing going on in base interpreters and make it explicit using CPS.

4.4.6 Many-sorted Languages: If-Statements

Up till now, the language components only have had a single syntactic category, or *sort*, namely expressions. In this section we discuss the propagation in the presence of multiple syntactic categories, such as expressions and statements.

In the context of object algebras, syntactic sorts correspond to type parameters of the factory interfaces. For instance, the following trait defines a language fragment containing if-then statements:

```
trait If[E, S] { def ifThen(c: E, b: S): S }
```

The `ifThen` construct defines a statement, represented by `S`, and it contains an expression `E` as a condition.

The interpreter for `ifThen` makes minimal assumptions about the kinds of expressions and statements it will be composed with. Therefore, `E` is instantiated to `Ev` (`() => Val`; see above), and `S` is instantiated to the type `Ex`:

```
type Ex = () => Unit

trait EvIf extends If[Ev, Ex] {
  def ifThen(c: Ev, b: Ex): Ex = () => if (c()) b()
}
```

The type `Ex` takes no parameters, and produces no result (`Unit`). The `ifThen` construct simply evaluates the condition `c` and if the result is true, executes the body `b`.

A first extension could be the combination with statements that require the store, like assignments. Statements that require the store are defined over the type `ExS = Sto => Unit`. As a result, `EvIf` needs to be lifted to map type `Ex` to `ExS`. Since the only argument of `ifThen` that has type `Ex` is the body `b`, lifting is only applied there. In the current language, expressions do not have side-effects, so they do not require the store, and consequently do not require lifting:

```
type ExS = Sto => Unit

trait ExSIf extends If[Ev, ExS] {
  private val base = new EvIf {};

  def ifThen(c: Ev, b: ExS) = st => base.ifThen(c, () => b(st))()
}
```

Note that the argument `c` is passed directly to `base.ifThen`.

An alternative extension is to add expressions which require an environment. In Section 4.3.1 such expressions were defined over the type `EvE = Env => Value`. In this case, `EvIf` needs to be lifted so that `ifThen` can be constructed with expressions requiring the environment. In other words, `c: Ev` needs to be lifted to `c: EvE`. However,

since an actual environment is needed to invoke a function of type `EvE`, the result sort `Ex` also needs to be lifted to accept an environment:

```
type ExE = Env => Unit

trait EvEIf extends If[EvE, ExE] {
  private val base = new EvIf {}

  def ifThen(c: EvE, b: ExE) = env => base.ifThen(() => c(env), () => b(env))
}
```

In this lifting code, the invocation of `c` requires an environment, and thus the closure returned by `ifThen` needs to be of type `ExE` to accept the environment and pass it to `c`.

The context parameters propagate outwards according to the recursive structure of the language. At the top level, the signature defining the semantics of a combination of language fragments will accept the union of all parameters needed by all the constructs that it could transitively contain.

4.4.7 Desugaring: Let

Desugaring is a common technique to eliminate syntactic constructs ("syntactic sugar") by rewriting them to more basic language constructs. As a result, the implementation of certain operations (like compilation or interpretation) becomes simpler because there are fewer cases to consider.

Desugaring in object algebras is realized by directly calling another factory method in the algebra. Note that methods in traits in Scala do not have to be abstract. As a result, desugarings can be generically implemented directly in the factory interface. The same, generic desugaring can be reused in any concrete object algebra implementing the syntactic interface.

As an example, recall the `Binding` language of Figure 4.1. It defines a `let` constructor which was implemented directly in the right column of Figure 4.1. Instead, `let` can be desugared to a combination of `lambda` and `apply`:

```
trait Let[E] extends Binding[E] {
  def let(x:Str, e:E, b:E) = apply(lambda(x, b), e)
}
```

This trait generically rewrites `let` constructs to applications of lambdas, binding the variable `x` in the body `b` of the `let`. Since the desugaring is generic, it can be reused for multiple interpreters, including the ones resulting from lifting. If `EvEBinding` (Figure 4.1) is lifted to propagate the store, for instance, the desugaring would automatically produce lifted `lambda` and `apply` denotations.

```
type EvES = (Env, Sto) => Val

trait EvESBinding extends Binding[EvES]  {
   ... // store propagation code
}

trait EvESBindingWithLet extends EvESBinding with Let[EvES]
```

Generic desugarings combined with traits (or mixins) provide a very flexible way to define language constructs irrespective of the actual interpretation of the constructs themselves. Keeping such desugared language constructs in separate traits also makes them optional, so that may remain unexpanded (such as for pretty printing).

4.4.8 Multiple Interpretations: Pretty Printing

Object algebras support the modular extension of both syntax and operations. Thus, implicit propagation can be applied to interpretations of a language other than dynamic semantics. Examples include type checking, other forms of static analysis, and pretty printing.

Consider the example of pretty printing. Here is a pretty printer for `Arith` expressions, `PPArith`, defined over the carrier type `PP` (`() => String`):

```
type PP = () => Str // "Pretty Print"

trait PPArith extends Arith[PP] {
  def add(l: PP, r: PP) = () => l() + " + " + r()

  def lit(n: Int) = () => n.toString
}
```

Pretty printing of arithmetic expressions does not involve the notion of indentation. However, to pretty print the `ifThen` construct of Section 4.4.6 we would like to indent the body expression. This is realized with a context parameter `i` that tracks the current indentation level:

```
type PPI = Int => Str

trait PPIIf extends If[PPI,PPI] {
  def ifThen(c: PPI, b: PPI) = i => "if " + c(0) + "\n" + " " * i + b(i + 2)
}
```

Both modules can be combined after lifting `PPArith` to propagate the parameter representing the current indentation:

```
trait PPIArith extends Arith[PPI] {
  private val base = new PPArith {}
```

```
  def add(l: PPI, r: PPI) = i => base.add(() => l(i), () => r(i))()
  def lit(n: Int) = i => base.lit(n)()
}
```

4.5 Automating Lifting

We have introduced implicit context propagation and illustrated how the liftings work in diverse scenarios. Although the liftings can be written by hand, they represent a significant amount of error-prone boilerplate code. We first introduce a Scala macro-based code generator for single-sorted algebras, which generates the lifting code automatically. Second, we describe how dynamic proxies in Java can be used to perform lifting at runtime.

4.5.1 Lift using a Scala Macro

The code generator is invoked by annotating an empty trait. In the compiled code, the code generator fills in the required lifting methods for this annotated trait. Here is an example showing how to lift the `EvArith` interpreter to propagate the environment and the store:

```
@lift[Arith[_], ()=> Val, EvArith, (Env, Sto) => Val]
trait EvESArith
```

The `@lift` annotation receives four type parameters: the trait that corresponds to the generic object algebra interface representing the language's syntax (`Arith`), the carrier type of the base implementation (`()=>Val`), the trait that provides the base level implementation (`EvArith`), and finally, the target carrier type (`(Env,Sto)=>Val`).

The annotated trait produces an implementation for the lifted trait that extends the factory interface instantiating the type parameter to the extended carrier type. The compiled code will contain the lifted methods that delegate to the specified base implementation. Note that the code generation does not break independent compilation or type safety: the generator only inspects the interfaces of the types that are specified in the annotation without needing access to the source code where these types are defined.

The `@lift` annotation is implemented as a Scala macro annotation [Bur13]. Macro annotations are definition-transforming macros that can be used to generate boilerplate code at definition level. Figure 4.6 shows the implementation of the `@lift` macro annotation. The class extending `StaticAnnotation` defines a macro annotation and defines a method `macroTransform` that contains the logic of the compile-time transformation. The companion object contains the `impl` method referenced by `macroTransform`. This method receives as arguments a collection of `annottees` that represents all the definitions in

the scope of the annotation. First, the four annotation arguments are extracted as trees and then type checked in order to obtain their types. Then, an intermediate representation is created from these types using a custom `InternalImporter`. The annottees are then pattern-matched in order to get the trait's name. The crucial step is calling the `liftTraitTo` method on the representation of the trait that corresponds to the algebra interface (instance of the class `Trait`, not shown). This method receives the name of the trait being transformed, the source carrier type, the trait corresponding to the base algebra implementation, and the target carrier type, and performs all the necessary transformations in order to return the representation of the lifted trait. This resulting trait is then serialized, concluding the transformation process.

The code generator does not simply prepend a new parameter at the front of the parameter list (as in the template of Figure 4.3), but performs the necessary permutations to appropriately lift the base signature to the target. This permutation logic is encoded in the already mentioned method `liftTraitTo`. This is important for components that need to be "mutually lifted". Consider a component which is lifted from `Sto=>Val` to `(Env,Sto)=>Val`. To combine this component with a component of type `Env=>Val`, the latter should be lifted to `(Env,Sto)=>Val` as well, but in this case, the parameter is added at the end.

The current version of `@lift` does not disambiguate parameters with the same type. It is always possible to create artificial wrapping types to distinguish between two context objects of the same type. It is, however, also conceivable to implement this by requiring the user to provide the disambiguation information in the annotation. This is an opportunity for future work.

4.5.2 Dynamic Lifting in Java

The Scala macro approach generates method implementations for a trait introduced by the programmer. Java does not have macro facility that supports a similar style of code generation. One would think the Java annotation processing framework [Ora15b] could be used for this, but, unfortunately, annotation processing only allows the generation of new classes or types, but not filling in the implementations of existing (abstract) classes or interfaces. As a result, client code becomes dependent on generated types which, in turn introduces temporal dependencies within the build cycle of the code.

Fortunately, it is also possible to perform lifting dynamically using the concept of dynamic proxies [Ora15a]. Since Java 8, function types are represented using *functional interfaces*: interfaces with a single method that acts as the "apply" method of functions. Dynamic proxies can be used to provide a generic implementation of such interfaces. Thus, instead of generating methods that encode the lifting of an interpreter, the context parameters are implicitly propagated at runtime.

```
class lift[ALG, FROM, BASEALG, TO] extends StaticAnnotation{
  def macroTransform(annottees: Any*) = macro lift.impl
}

object lift{

  def impl(c: whitebox.Context)(annottees: c.Expr[Any]*) = {
     import c.universe._

     // Get the annotation arguments as trees
     val (algAST: Tree, srcFunAST: Tree, baseAlgAST: Tree, tgtFunAST: Tree)
       = c.macroApplication match{
          case q"new lift[$a, $from, $baseA, $to].macroTransform($_)" =>
            (a, from, baseA, to)
          case _ =>
            c.abort(c.enclosingPosition, "Invalid type parameters")
     }

     // Get the types of the annotation arguments
     val (algType: Type, srcType: Type, baseImplType: Type, tgtType: Type)
       = Checker.typeCheck(c)(algAST, srcFunAST, baseAlgAST, tgtFunAST)

     // Create importer that allows to create intermediate representations of traits and
     // function types based on the annotation arguments
     val (alg: Trait, srcFun: FunType, baseImpl: Trait, tgtFun: FunType)\newline
       = InternalImporter.importTypes(c)(algType, srcType, baseImplType, tgtType)}

     // At least one annottee must be a trait
     annottees.map(_.tree) match {
      case (q"$mods trait $name") ::Nil => {

        // This is the code that triggers the lifting on the intermediate representations
        val lifted: Trait = alg.liftTraitTo(name.decoded, srcFun, tgtFun, "base"+alg.name)

        // Serialize the lifted trait and return it as the result of the transformation
        val result: Expr[Any] = Render.serialize(c)(mods, lifted, alg, baseImpl, srcFun)
        result
      }
      case _ => c.abort(c.enclosingPosition, "Invalid annottee")
     }
  }
}
```

Figure 4.6: The `lift` macro annotation for lifting object algebra at compile time

```
interface ExpAlg<E> {
  E lit(int n);
  E add(E l, E r);
}

@FunctionalInterface
interface IEval {
  int eval();
}
```

```
interface EvalExp extends ExpAlg<IEval> {
  default IEval lit(int n) {
    return () → n;
  }

  default IEval add(IEval l, IEval r) {
    return () → l.eval() + r.eval();
  }
}
```

Figure 4.7: Arithmetic expression evaluation using Java 8 functional interfaces and default methods

Figure 4.7 shows a simple expression evaluator in object algebra style using Java 8 functional interfaces and interface default methods. Note how the closure notation automatically creates objects that are instances of the `IEval` interface.

Of course, another language fragment could require additional context parameters, which are not reflected in the type `IEval`. For instance, denotations requiring a environment could be represented by the following interface:

```
@FunctionalInterface
interface IEvalEnv {
  int eval(Env env);
}
```

We need an implementation of `ExpAlg` over the carrier type `IEvalEnv`. Figure 4.8 shows a slightly contrived implementation of expression evaluation with dynamic environment propagation using dynamic proxies. If this code would be implemented by hand there would be no need for dynamic proxies: the implementation would simply follow the pattern of the manual Scala lifting in Section 4.3.1. However, the example illustrates how dynamic proxies can be used to simulate functions in Java.

The lifting is realized using the helper methods `lift` and `lower`. Both methods return proxies created using `java.lang.reflect.Proxy::newProxyInstance`. This method takes a class loader, an array of interfaces the proxy is supposed to export, and an instance of the `java.lang.reflect.InvocationHandler` interface to handle method requests on the proxy. Since `InvocationHandler` is a functional interface in Java 8, one can directly provide closures as invocation handlers. The returned object will behave as an instance of all the provided interface types, but all method invocations will be routed to the invocation handler.

The `lift` method takes a function from the extra parameter (`Env`) to `IEval` and uses it to create proxy objects of type `IEvalEnv`. The returned proxy object provides the

```
interface EvalExpEnv extends ExpAlg<IEvalEnv> {
  static final ExpAlg<IEval> base = new EvalExp() {};

  default IEvalEnv lit(int n) {
    return lift(env → base.lit(n));
  }

  default IEvalEnv add(IEvalEnv l, IEvalEnv r) {
    return lift(env → base.add(lower(l, env), lower(r, env)));
  }

  static IEvalEnv lift(Function<Env, IEval> f) {
    return (IEvalEnv) Proxy.newProxyInstance(IEvalEnv.class.getClassLoader(),
      new Class<?>[] { IEvalEnv.class }, (p, m, as) → f.apply((Env) as[0]).eval());
  }

  static IEval lower(IEvalEnv e, Env env) {
    return (IEval) Proxy.newProxyInstance(IEval.class.getClassLoader(),
      new Class<?>[] { IEval.class }, (p, m, as) → e.eval(env));
  }
}
```

Figure 4.8: Manually lifting arithmetic expression evaluation in Java to propagate environments

extra argument (`as[0]`) to the closure `f` to obtain an `IEval` object and then calls `eval()` on it. The `lower` function has the inverse effect of lift: it turns objects of a "larger" type (e.g., `IEvalEnv`) into objects of a smaller type (e.g., `IEval`), again using proxies.

The closure `f` provided to `lift` abstracts over how to turn a base interpreter into the lifted type given the extra parameter which should be propagated. For instance, in the case of `add`, the provided function to `lift` calls the `base.add` constructor with lowered versions of `l` and `r`. Lowering is realized by the helper method `lower`, which turns `IEvalEnv` objects into `IEval` objects. Whenever `eval` is called on such an `IEval`, it delegates back to the original `IEvalEnv` providing the extra parameter `env` that was captured when `lower` was invoked.

The interface `EvalExpEnv` can be composed with other interfaces over the same carrier type, similar in style to Scala trait inheritance. However, the propagation code is specific for two carrier types (i.e. `IEval` and `IEvalEnv`). This problem is solved by introducing yet another level of dynamic proxies, this time at the level of the algebras themselves.

Since signatures of object algebras are represented by Java interfaces, dynamic proxies can also be used to simulate object algebras themselves. The following `lifter` method creates such "lifting" algebras automatically:

```
static <F, S, T> F lifter(Class<F> ialg, Class<S> source, Class<T> target,
    F base) {
  return (F) Proxy.newProxyInstance(ialg.getClassLoader(), new Class<?>[] {ialg},
    new Lifter<>(source, target, base));
}
```

Intuitively, this method turns an object algebra of type `F<S>` into an algebra of type `F<T>`, assuming that both `S` and `T` are functional interfaces, where the single method in `T` has one extra parameter. Method invocations on the resulting algebra of type `F<T>` are handled by the invocation handler `Lifter` which will create proxy objects delegating to the `base` algebra (of type `F<S>`) and propagating the extra parameter behind the scenes.

The `Lifter` class is shown in Figure 4.9. The entry point of the `Lifter` class is always an invocation of a factory method (e.g., `add`, `lit`, etc.) which will be handled by the `invoke` method from Java's `InvocationHandler` interface. The `invoke` method receives the current proxy object, the called method and the method's arguments (`kids`). It immediately returns the result of calling `lift` which produces the desired target type `T`.

The `lift` method follows the same pattern as the `lift` method of Figure 4.8. The main difference is that the argument `f` now accepts both a method object representing the method supporting the extra parameter in addition to the extra argument itself. Whereas the closure provided to `lift` in Figure 4.8 directly called the appropriate factory method, in this case the base interpreter is created using reflection, and the arguments are lowered using a loop in `lowerKids`: for every constructor argument in `kids` that is an instance of the target type `T`, a proxy is created in the `lower` method. This proxy will call the method `extEval` on the original `T` object (`kid`), extending the list of arguments with the extra argument originally received in the closure provided to `lift`; the `extend` helper method creates a copy of `args` with the `extra` object appended to it.

After the `evaluator` object is obtained from the `f` closure in `lift`, the corresponding method in type `S` is looked up using reflection on `source`. We simply look for a method with the same name, but with one fewer parameter, and then invoke it on the base `evaluator` ignoring the extra parameter. Finally, this `eval` method is invoked on the base `evaluator` object, ignoring the last element of the extended argument array (`extArgs`).

Using the `lifter` method any object algebra `F<S>` defined over a functional interface **`interface`** `S { U m(`C_1 c_1`, ..., `C_n c_n`); }` can now be converted to an algebra `F<T>` defined in terms of **`interface`** `T { U m(`C_1 c_1`, ..., `C_n c_n`, `C_{n+1} c_{n+1}`); }`. The extra parameter c_{n+1} will be dynamically propagated.

Implicit context propagation using dynamic proxies operates at the level of runtime objects, whereas the Scala macro operated at the trait level. In other words: the result of `lifter` is a runtime object, not a trait or class. As a result, it is not possible

```
class Lifter<S, T, F> implements InvocationHandler {
  private final Class<S> source;
  private final Class<T> target;
  private final F base;

  Lifter(Class<S> s, Class<T> t, F base) {
    this.source = s;
    this.target = t;
    this.base = base;
  }

  public T invoke(Object proxy, Method constructor, Object[] kids) {
    return
      lift((extEval, extra) ⟶
             (S)constructor.invoke(base, lowerKids(extEval, kids, extra)));
  }

  private T lift(BiFunction<Method, Object, S> f) {
    return proxy(target, (p, extEval, extArgs) ⟶ {
          S evaluator = f.apply(extEval, extArgs[extArgs.length - 1]);

      // Get the method in S corresponding to extEval in T
      Method eval =
        source.getMethod(extEval.getName(),
          Arrays.copyOf(extEval.getParameterTypes(), extArgs.length - 1));

      // Invoke it on the base interpreter ignoring the extra parameter.
      return eval.invoke(evaluator, Arrays.copyOf(extArgs, extArgs.length - 1));
    });
  }

  private S lower(Method extEval, T kid, Object extra) {
    return proxy(source, (p, eval, args) ⟶ extEval.invoke(kid, extend(args,
  extra)));
  }

  private Object[] lowerKids(Method eval, Object[] kids, Object extra) {
    Object lowered[] = Arrays.copyOf(kids, kids.length);
    for (int i = 0; i < kids.length; i++)
      if (target.isInstance(kids[i]))
        lowered[i] = lower(eval, (T)kids[i], extra) ;
    return lowered;
  }
}
```

Figure 4.9: The `Lifter` class for lifting object algebra using dynamic proxies

anymore to compose language fragments using trait/interface inheritance. Once again, dynamic proxies can be used to mitigate the problem.

The following code defines a `union` combinator which multiplexes method invocations from a single object algebra interface between actual implementations of subsets of the interface (see [GPS14; OvdSL+13] for similar incarnations of this idea):

```
static <T> T union(Class<T> ialg, Object ...algs) {
  return
    (T) Proxy.newProxyInstance(
          ialg.getClassLoader(),
          new Class<?>[] { ialg },
          (x, m, args) → {
            for (Object alg: algs)
              try { return m.invoke(alg, args); }
              catch (Exception e) { continue; }
            throw new UnsupportedOperationException("no such method"); });
}
```

Let's assume we have two language fragments, the simple arithmetic expressions and a language for binding constructs. At the interface level these fragments can be combined using interface extension:

```
interface ExpBindAlg<E> extends ExpAlg<E>, BindAlg<E> { }
```

The dynamically lifted version of `EvalExp` can now be combined with an implementation of the binding fragment (say, `EvalBind`):

```
ExpAlg<IEvalEnv> evalExp = lifter(ExpAlg.class, IEval.class, IEvalEnv.class,
                                  new EvalExp() {});
BindAlg<IEvalEnv> evalBind = new EvalBind() {};
ExpBindAlg<IEvalEnv> evalExpBind = union(ExpBindAlg.class, evalExp, evalBind);
```

The algebra `evalExpBind` can now be used to create expressions just like any ordinary algebra.

4.5.3 Discussion

We have presented two approaches to automate lifting: Scala macros and Java dynamic proxies. As the macro transformation is a compile-time mechanism, all the type information is available when generating the code for the lifted algebras, and thus all the generated code is type safe. On the other hand, in the case of the dynamic proxies, the lifting is realized at runtime and requires casts in order to make the proxied object conform to the lifted interfaces. All the code at the points where explicit casting occur (e.g., the cast that returns a proxied lifted object from the `lifter` method) is unsafe. Having said that, this unsafety is limited to code that is provided by the framework as

a reusable mechanism for dynamic lifting. Provided that the framework is correctly implemented, the end user is not affected by it.

4.6 Case study 1: Extending a DSL for State Machines

In this section we present a case study, based on an extensible DSL for state machines, inspired by the example DSL introduced in [Fow10]. The exploration of this DSL is motivated by the following considerations.

First, state machines emphasize the DSL perspective: state machines are different from expression-oriented or even statement-oriented programming languages. The contexts involved are not just environments or stores, but may contain arbitrary interfaces to some unspecified outside world.

Second, whereas the examples of Section 4.4 are mostly single sorted expression languages, state machines are inherently many sorted, since such a language typically involves at least the syntactic categories of state machines, states and transitions. Thus, this case study illustrates more clearly that adding context parameters to nested AST types requires lifting surrounding AST types, similar to how statement interpreters required lifting in Section 4.4.6.

Third, the state machine example strongly illustrates that implicit context propagation can be seen as a form of "scrapping your boilerplate" [LP03]. In a sense, implicit context propagation supports the creation of *structure shy* [Lie96] language extensions: adding language features deep down within the syntactic structure does not require to change the definitions of the semantics of surrounding, context unaware node types. Similar concerns were addressed, for instance, by implicit parameters (as described in [LLM+00]): many cases in the definition of a recursive function do not use the context information, but some leaves of the recursion need the information.

Finally, instead of building upon the assumption that a language is constructed by assembling the smallest possible building blocks, the state machine case study is presented from the perspective of language extension. Given an existing definition of a state machine language, we will extend it with three new types of transitions: conditional transitions, transitions with token output, and conditional transitions with token output.

4.6.1 State Machines

A state machine consists of a list of named state definitions. Each state contains a list of outgoing transitions. A transition fires on an event (a string) and transitions to a target state (identified by name). The abstract syntax of the state machine language is shown in Figure 4.10.

An interpreter for the base language is shown in Figure 4.11. The carrier type `EvM` captures functions from the current state and event to the next state, if any. A

```
trait Stm[M, S, T] {
  def machine(name: String, states: Seq[S]): M
  def state(name: String, transitions: Seq[T]): S
  def transition(event: String, target: String): T
}
```

Figure 4.10: The abstract syntax of the state machine language

```
type EvM = (String, String) => Option[String]

trait EvalStm extends Stm[EvM, EvM, EvM] {
  override def machine(name: String, states: Seq[EvM]): EvM
    = (st, ev) => states.map(_(st, ev)).find(_.isDefined).flatten

  override def state(name: String, transitions: Seq[EvM]): EvM
    = (st, ev) =>
        if (name == st) transitions.map(_(st, ev)).find(_.isDefined).flatten else None

  override def transition(event: String, target: String): EvM
    = (st, ev) => Option(if (ev == event) target else null)
}
```

Figure 4.11: A simple interpreter for state machines. The carrier type `EvM` captures functions from current state and event to next state (if any).

machine simply finds the first state with a firing transition. A transition may fire if it is defined in the state we are in (`st`), and if the event `ev` matches the event in the transition. If a transition fires, the target state is returned.

As an example, consider a simple state machine controlling the opening and closing of doors:

```
def doors[M, S, T](alg: Stm[M, S, T]): M =
  alg.machine("doors", Seq(
      alg.state("closed", Seq(alg.transition("open", "opened"))),
      alg.state("opened", Seq(alg.transition("close", "closed")))))
```

This state machine can be used as follows:

```
val stm = doors(new EvalStm {})
val Some(st1) = stm("closed", "open")
println(st1);
val Some(st2) = stm(st1, "close")
println(st2);
```

Executing the code will print out:

```
trait TokenTrans[T] {
  def transition(event: String, target: String, tokens: Set[String]): T
}

trait CondTrans[E, T] {
  def transition(cond: E, event: String, target: String): T
}

trait CondTokenTrans[E, T] {
    def transition(cond: E, event: String, target: String, tokens:
  Set[String]): T
}
```

Figure 4.12: Three language extensions defining the abstract syntax transitions with token output, transitions with conditions, and transitions with both token output and conditions.

```
opened
closed
```

4.6.2 Modular Extension of State Machines

The abstract syntax of the three state machine extensions is shown in Figure 4.12. Each extension is defined in its own trait. `TokenTrans` defines transitions that output a set of tokens. `CondTrans` defines conditional transitions, introducing an additional `E` sort representing expressions. Finally, `CondTokenTrans`, combines both extensions to support conditional transitions with token output.

Transitions with Token Output The definition of transitions with token output is as follows:

```
type EvTT = (String, String, Writer) => Option[String]

trait EvalTokenTrans extends TokenTrans[EvTT] {
  def transition(event: String, target: String, tokens: Set[String]): EvTT
    = (st, ev, w) => if (ev == event) {
        tokens.foreach(t => w.append(t + "\n")); Some(target)
      }
      else None
}
```

The type `EvTT` describes that transitions with token output require another context parameter, in this case a `Writer` object that the output tokens will be written to. The

`transition` method then returns a function that outputs the given `tokens` whenever the transition fires.

To combine this language extension with the base interpreter of state machines shown in Figure 4.11, the latter has to be lifted to propagate the `Writer` parameter. The lifted version of `EvalStm` has the following structure:

```
trait LiftEvalStm extends Stm[EvTT, EvTT, EvTT] {
  private val base = new EvalStm {}

  def machine(name: String, states: Seq[EvTT]): EvTT
    = (st, ev, w) => base.machine(...)

  def state(name: String, transitions: Seq[EvTT]): EvTT
    = (st, ev, w) => base.state(...)

  def transition(event: String, target: String): EvTT
    = (st, ev, w) => base.transition(...)
}
```

Note that all three type parameters of `Stm` (`M`, `S`, `T`) are bound to the extended signature `EvTT` since the writer object needs to be provided from the top in order to be propagated down to the level of transitions.

Given the syntax traits `Stm` and `TokenTrans`, state machines may now contain transitions that declare tokens that will be output upon firing:

```
def doorsTokens[M, S, T](alg: Stm[M, S, T] with TokenTrans[T]): M =
  alg.machine("doors",
    Seq(
      alg.state("closed",
        Seq(alg.transition("open", "opened", Set("TK1OP", "TK2OP")))),
      alg.state("opened",
        Seq(alg.transition("close", "closed", Set("TK1CL", "TK2CL"))))))
```

To execute such extended state machines, the interpreters `LiftEvalStm` and `EvalTokenTrans` need to be combined:

```
val stm = doorsTokens(new LiftEvalStm  with EvalTokenTrans {})
val w = new StringWriter()
val Some(st1) = stm("closed", "open", w)
...
```

Printing out the contents of the writer object `w` will contain the tokens as sequentially output by the new kind of transitions.

Conditional Transitions The extension with conditional transitions follows a similar pattern as the extension with transitions outputting tokens. In this case, however,

the extension introduces a new syntactic category for expressions. As a result, this extension also requires a separate language fragment defining the syntax and semantics of expressions. We assume this language is defined in its own trait `Cond` and that its interpreter `EvalCond` is defined over the type `EvE = Env => Boolean`. As a result, the propagated context parameter is the environment used to evaluate a transition's condition.

The semantics of conditional transitions are then defined as follows:

```
type EvET = (Env, String, String) => Option[String]

trait EvalCondTrans extends CondTrans[EvE, EvET] {
  override def transition(cond: EvE, event: String, target: String): EvET
    = (env, st, ev) => Option(if (ev == event && cond(env)) target else null)
}
```

Creating state machines with conditional transitions is now defined over the algebra interfaces `Stm`, `Cond` and `CondTrans`. Executing such state machines combines the lifted base interpreter to propagate the environment, with `EvalCond` and `EvalCondTrans`.

Conditional Transitions with Token Output The final extension combines both conditions and token output in transitions. Although this extension can be considered in isolation, it makes intuitively more sense to allow this kind of transition to coexist with the two extensions described above. As a result, adding conditional transitions with token output requires two-level lifting of the base interpreter. In other words, one of the lifted interpreters for the previous extensions is lifted once again to propagate the additional context (i.e. writer object or environment).

Additionally, to combine this extension with the previous extensions, both interpreters `EvalTokenTrans` and `EvalCondTrans` need to be lifted. `EvalTokenTrans` needs to propagate the environment, and `EvalCondTrans` needs to propagate the writer object. Note however, that the interpreter for conditions (`EvalCond`) does not require lifting.

The only code that remains to be written is the interpreter for the new kind of transitions itself:

```
type EvETT = (Env, String, String, Writer) => Option[String]

trait EvalCondTokenTrans extends CondTokenTrans[EvE, EvETT] {
  override def transition(cond: EvE, event: String, target: String,
      tokens: Set[String]): EvETT
    = (env, st, ev, w) => if (ev == event && cond(env)) {
        tokens.foreach(t =>w.append(t + "\n"))
        Some(target)
      }
      else None
}
```

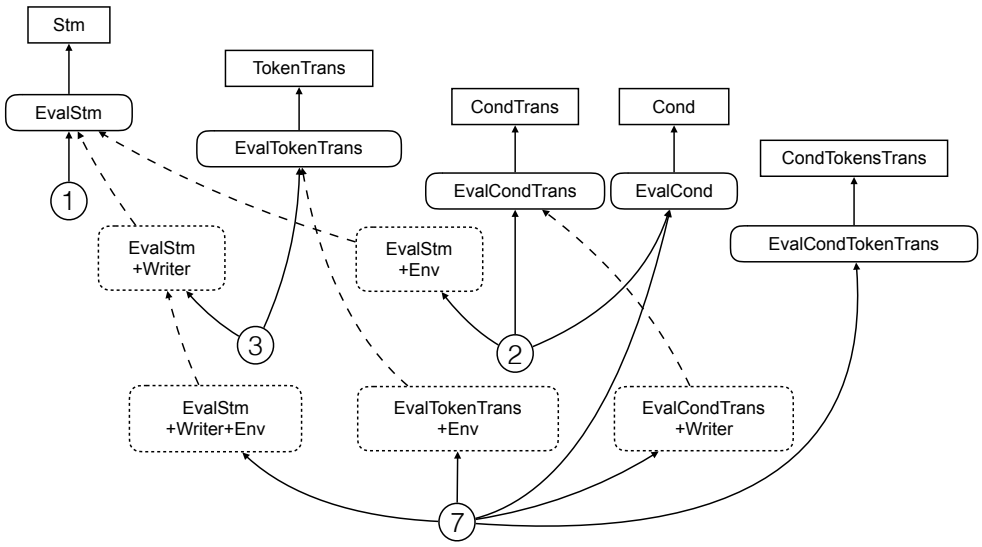


Figure 4.13: Modular extension of a state machine DSL with conditions and/or token output. Rectangles define syntactic constructs. Rounded rectangles are interpreters; dotted borders indicate lifted interpreters. Solid arrows represent trait inheritance and dashed arrows represent delegation inherent in lifting. Each circle represent an executable composition of modules.

Note that this definition duplicates the logic of ordinary transitions, conditional transitions and transitions with token output. This may seem unfortunate, but understandable: the new kind of transition represents feature interaction between transition firing, condition evaluation and token output, which can never be automatically derived from the given interpreters.

Summary To summarize, given the manually written language modules `EvalStm`, `EvalTokenTrans`, `EvalCondTrans` and `EvalCondTokensTrans` and an additional module defining conditional expressions (`EvalCond`), we can derive the following 7 language variants:

1. `Stm`
2. `Stm` **`with`** `CondTrans`
3. `Stm` **`with`** `TokenTrans`
4. `Stm` **`with`** `CondTokenTrans`
5. `Stm` **`with`** `TokenTrans` **`with`** `CondTokenTrans`
6. `Stm` **`with`** `CondTrans` **`with`** `CondTokenTrans`
7. `Stm` **`with`** `CondTrans` **`with`** `TokenTrans` **`with`** `CondTokenTrans`

Compositions 1, 2, 3, and 7 make the most sense and are depicted graphically in Figure 4.13. Each solid rectangle defines a syntactic trait, the semantics of which is implemented in the rounded rectangles (interpreters); the solid arrows represent trait inheritance or extension. The dashed rounded rectangles represent liftings of interpreters, and the dashed arrows represent delegation to the base language. The circles represent compositions of language fragments.

4.7 Case Study 2: Modularizing Featherweight Java

To examine how implicit context propagation helps in modularizing a programming language implementation, we present a second case study using *Featherweight Java* (FJ) [IPW99]. The case study consists of a modular interpreter for FJ and its extension to a variant that supports state (SFJ), inspired by [FKF98].

The case study addresses two questions:

- What is the flexibility that implicit context propagation provides to support the definition of languages by assembling language fragments?
- How much boilerplate code is avoided by implicit context propagation?

In this section, these questions are answered by analyzing the number of hypothetical languages that can be defined from the combination of SFJ fragments, and by counting the possible liftings.

4.7.1 Definition of FJ and SFJ

FJ was introduced as a minimal model of a Java-like language, small enough to admit a complete formal semantics. In FJ, there are no side-effects and all values are objects; it supports object creation, variables, method invocation, field accessing and casting. To study how to extend a language to a variant that requires more context information, we introduce SFJ, which also features field updating and sequencing.

We have modularly implemented FJ and its extension to SFJ defining one language module per alternative in the abstract grammar. Each language construct is represented as a single object algebra interface to allow for maximum flexibility. As a consequence, the semantics of each construct is defined in its own trait assuming only the minimal context information necessary for the evaluation of that particular construct.

A complete definition of SFJ requires four kinds of context information:

- An `Obj` that represents the object being currently evaluated (i.e., **`this`**). In FJ, the `Obj` simply contains the object's class name and the list of arguments that are bound to its fields.

		Syntax	Signature
FJ	Field access	`e.f`	`CT=>Obj`
	Object creation	`new C(e,...)`	`()=>Obj`
	Casting	`(C) e`	`CT=>Obj`
	Variables	`x`	`(Obj,Env)=>Obj`
	Method call	`e.m(e,...)`	`(Obj,CT,Env)=>Obj`
SFJ	Sequencing	`e ; e`	`()=>Obj`
	Field assignment	`e.f = e`	`(CT,Sto)=>Obj`
	Object creation	`new C(e,...)`	`(CT, Sto)=>Obj`
	Variables	`x`	`(Obj,CT,Env,Sto)=>Obj`

Table 4.2: Signatures per (S)FJ language construct

- The class table `CT` which contains the classes defined in an FJ program. The classes contain the meta information about objects, in particular, how the ordering of constructor arguments maps to the object's field names.
- The environment `Env` which maps variables to `Obj`s.
- The store `Sto` modeling the heap (just needed in the case of SFJ).

As shown in Table 4.2, six different signatures are used to implement nine constructs. For presentation purposes, we solely focus on the expression constructs:

- Object creation does not require any context information.
- Field access and casting only require the class table to locate fields in objects by offset.
- Variables require the current object to evaluate the special variable **`this`**, and the environment to lookup other variables.
- Method calls require the class table to find the appropriate method to call; the current object is needed to (re)bind the special variable **`this`** and the environment is needed to bind formal parameters to actual values.
- Sequencing does not depend on any context parameters.
- Field assignment uses the class table to locate fields and the store to modify the object.

Notice too that the cases for object creation and variable referencing had to be redefined in SFJ over the signatures`(CT,Sto) => Obj` and `(Obj,CT,Env,Sto) => Obj` in order to allocate storage for the newly created object and inspecting the referenced object in the store, respectively. In particular, variable referencing needs all the context parameters as it needs to "reconstruct" the object structure by inspecting the store and finding the information about the order of arguments in the class table.

```
@lift[Field[_], CT => Obj, EvCT2ObjField, (Obj, CT, Env, Sto) => Obj]
trait EvSlfCtESt2ObjField

@lift[New[_], (CT, Sto) => Obj, EvCtSt2ObjNew, (Obj, CT, Env, Sto) => Obj]
trait EvSlfCtESt2ObjNew

@lift[Cast[_], CT => Obj, EvCt2ObjCast, (Obj, CT, Env, Sto) => Obj]
trait EvSlfCtESt2ObjCast

@lift[Call[_], (Obj, CT, Env) => Obj, EvSlfCtEnv2ObjCall, (Obj, CT, Env, Sto) => Obj]
trait EvSlfCtESt2ObjCall

@lift[Seq[_], () => Obj, Ev2ObjSeq, (Obj, CT, Env, Sto) => Obj]
trait EvSlfCtESt2ObjSeq

@lift[SetField[_], (CT, Sto) => Obj, EvCtSt2ObjSetField, (Obj, CT, Env, Sto) => Obj]
trait EvSlfCtESt2ObjSetField
```

Figure 4.14: Automatic lifting of SFJ language components using the `@lift` macro annotation

For implementing FJ, four of the base interpreters (for variables, field access, object creation and casting) are lifted to the function type `(Obj,CT,Env)=>Obj`. Combining these lifted interpreters results in an implementation of basic FJ.

In order to obtain a full implementation of SFJ, the FJ interpreters need to be lifted to also propagate the store and the stateful fragments need to be lifted to propagate the environment, class table and current object, where needed. The result is a set of interpreters defined over the "largest" signature `(Obj,CT,Env,Sto) => Obj`. We have implemented the lifting using the `@lift` macro annotation discussed in Section 4.5.1. The relevant code is shown in Figure 4.14.

4.7.2 Analyzing Hypothetical Subsets of SFJ

The previous subsection detailed how the implementation of a complete language, in this case FJ and SFJ, can be constructed from assembling language fragments. Here we discuss hypothetical subsets of such languages. Even though these subsets might not (and probably will not) be meaningful in any practical sense, they illustrate the flexibility that implicit context propagation promotes.

Table 4.3 shows how many interpreters can be derived per interpreter signature using implicit context propagation. The second column lists the number of given base interpreters over a specific signature. The third column indicates the number of lifting opportunities. Finally, the last column shows the total number of possible

Signature	Base	Liftings	Derived	Total
`CT=>Obj`	2	O/E/S	14	16
`(Obj,Env)=>Obj`	1	C/S	3	4
`(Obj,CT,Env)=>Obj`	1	S	1	2
`()=>Obj`	2	C/O/E/S	30	32
`(CT,Sto)=>Obj`	2	O/E	6	8
`(Obj,CT,Env,Sto)=>Obj`	1		0	1
				63

Table 4.3: Number of Base interpreters per signature, possible Liftings (C = `CT`, O = `Obj`, E = `Env`, S = `Sto`), number of possible Derived interpreters and Total number of possible interpreters.

interpreters, including the base interpreters. Note that the next-to-last row shows two base interpreters because the interpreter for object construction needed to be rewritten to allocate storage.

Lifting opportunities are described using a shorthand indicating which types of parameters could be added to the signatures using implicit context propagation (C = `CT`, O = `Obj`, E = `Env`, S = `Sto`). For instance, the string "O/E/S" in the first row means that an interpreter over `CT=>Obj` can be lifted to any of the following 7 signatures:

```
(Obj,CT)=>Obj, (Env,CT)=>Obj, (Sto,CT)=>Obj, (Obj,CT, Env)=>Obj,
(Obj,CT, Sto)=>Obj, (Env,CT,Sto)=>Obj, (Obj,Env,CT,Sto)=>Obj
```

The number of possible lifted interpreters given n base interpreters can be computed using the following formula $n \times (2^k - 1)$, where k represents the number of possibly added context parameters. Since there are $n = 2$ base interpreters in the first row for which $k = 3$, 7 opportunities apply to each of them, and thus the total number of derivable interpreters is 14.

Summing the last column in Table 4.3 gives an overall total of 63 possible interpreters, of which only 9 are written by hand. The other 54 can be derived automatically using implicit context propagation. Thus, our technique eliminates considerable amount of boilerplate when deriving new variants of languages from base language components.

The 63 interpreters include 7 over the "largest" signature `(Obj,CT,Env,Sto) => Obj` for each of the 7 language constructs. These 7 fragments allow $2^7 - 1 = 127$ combinations representing hypothetical subsets of SFJ (excluding the empty language). The interpreter for full SFJ is just one of these 127 variants. This gives an idea of the flexibility that implicit context propagation provides in defining multiple language variants from assembling the different language modules.

4.8 Performance Overhead of Lifting

A lifted interpreter exhibits more runtime overhead than its equivalent non-lifted version. This is because the lifting works creating new closures at runtime to adapt the signatures of the arguments and adds additional call overhead to go from one closure to the other.

4.8.1 Benchmarking Realistic Code: Executing the DeltaBlue Benchmark

In order to have an idea of the impact of lifting, we performed an experiment using the DeltaBlue benchmark [FM89]. DeltaBlue represents an incremental constraint solver and is used to benchmark programming languages. DeltaBlue was originally developed in Smalltalk; in our experiment we use the Javascript version.[3] To execute the benchmark we developed a modular interpreter of λ_{JS}, based on the semantics described in [GSK10].

Using the desugaring framework published at [GSK15] the DeltaBlue Javascript code was desugared to λ_{JS} and input to our interpreter. To assess the performance impact of lifting we ran the benchmark using two interpreters: one that employed lifting, and one that propagated the context parameters explicitly, i.e., written in the "anticipation style" discussed earlier.

The interpreter that propagates all the context information explicitly represents our baseline, as we consider it to be a straightforward implementation that follows the interpreter pattern [GHJ$^+$94] (except it uses closures as AST objects).

In order to make the comparison as fair as possible regarding to the impact of lifting, both interpreters were modularized in logically-related units representing sublanguages of λ_{JS}, namely: *Binding*, *Control*, *Mutability*, *Core*, and *Exceptions*. This means that both styles of interpreters have the same trait inheritance structure.

In the case of the lifted interpreter, each module is defined using carrier signatures that consider only the needed context. For the non-lifted interpreter, however, every module is defined over the maximal signature, anticipating all required context information even though not all of it is required in each and every module. For instance, the *Control* module does not need any context because it only deals with syntactic forms that represent control flow. While the lifted interpreter's carrier type `()->Val` reflects this, the non-lifted version is defined over the largest signature: the signature with both environment and store as context parameters (`(Sto, Env)->Val`).

Table 4.4 shows the signatures of the carrier type that corresponds to expressions, in the base components of the λ_{JS} implementation that depends on lifting. Since there is no single module defined over the largest signature, all modules have to be lifted to either propagate the store or to propagate the environment, or both. As a result, all

[3] See `https://github.com/xxgreg/deltablue`

Component	Expression Carrier Type
Binding	`Env => Val`
Control	`() => Val`
Core	`Sto => Val`
Exceptions	`Env => Val`
Mutable	`Sto => Val`

Table 4.4: Signatures of expression carrier types for the base components of the λ_{JS} implementation that depends on lifting

modules exhibit some level of lifting, and thus all nodes created by the desugared DeltaBlue program will be affected by the performance overhead of lifting.

The benchmarks were executed on a MacBook Pro with OS X Yosemite (version 10.10.3) and 8 GB RAM, running on an Intel Core i5 CPU (2.7 GHz). We use Oracle's JVM (JDK 8u51), executed in server mode. The DeltaBlue benchmark can be run for a variable number of constraint solving iterations. We executed the program for an increasing number of iterations from 5 to 30 in steps of 5, taking 15 time measurements per number of iterations using Java `System.nanoTime()`.

Figure 4.15 shows a box plot of the execution times. For each number of iterations, we show the execution time for the non-lifted interpreter (white fill) and the lifted interpreter (gray fill). Calculating the difference in slowdown relative to the size, we observe that the slowdown is in the range of 63% and 81% with a median of 77.87%. We conjecture that the slowdown is due to the allocation of new closures at runtime and additional call overhead between the lifted closure and base closures. To zoom in on these effects, we now describe a micro-benchmark that partially confirms this hypothesis.

4.8.2 Micro-benchmark: Executing Lifted Interpreters in a Loop

To isolate the effect of lifting we created a micro-benchmark based on a simple expression language, containing literals, addition and a sum construct. The sum construct receives an integer n and an argument expression and sums the evaluated arthis progument n times in a loop. For $n \in [0...1,000,000]$ with step size $10,000$ the benchmark executes the expression $\mathsf{sum}(n, \mathsf{add}(\mathsf{lit}(1), \mathsf{lit}(2)))$. We report the average running time measured using `System.nanoTime()` over 100 executions per n.

The benchmark is executed for three versions of the expression interpreter: one without lifting (explicitly propagating an environment), one that is lifted to implicitly propagate and environment, and an optimized lifted interpreter that we discuss below.

The benchmark results are shown in Figure 4.16. It is immediately clear from Figure 4.16 that lifting does incur significant overhead. The non-lifted interpreter

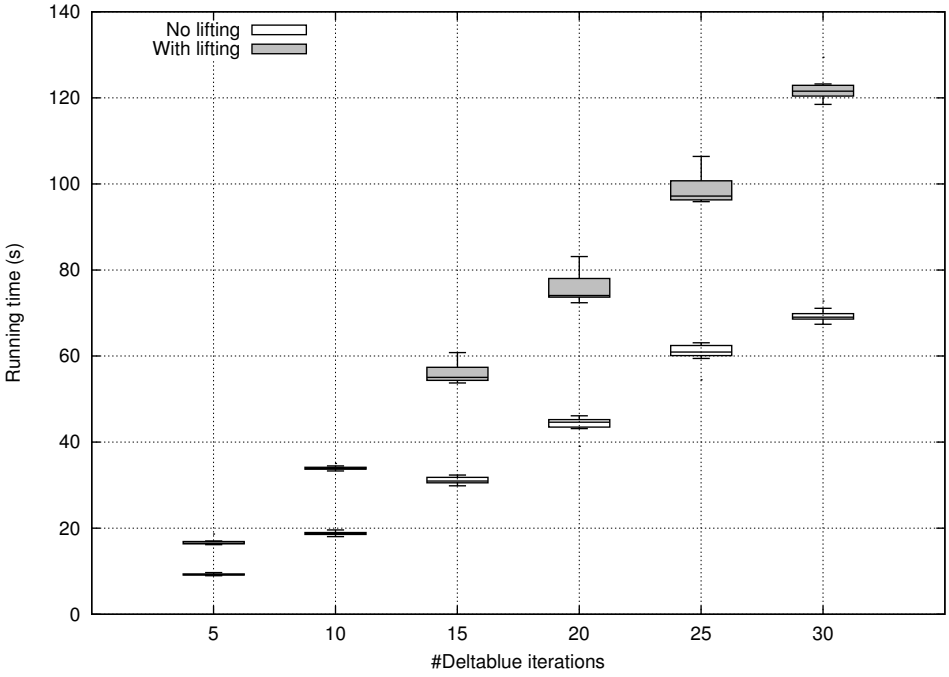


Figure 4.15: Runtime of the lifted interpreter compared to the baseline interpreter.

is about twice as fast as the lifted versions, and gets even faster as soon as the JIT compiler kicks in (around 540,000). This is because the non-lifted interpreter involves one call per expression evaluation, whereas the lifted interpreters always involve two.

The "fast" lifting results show a small improvement over vanilla lifting. The optimization in fast lifting is based on moving the creation of new closures outside of dynamic expression evaluation, and using side-effects to bind the propagated parameter. As a result, all additional closures are created once and for all when the expression itself is created. For instance, the optimized lifting of the addition construct is defined as follows:

```
def add(l: EvE, r: EvE): EvE = {
  var env: Env = null
  val lw = () => l(env)
  val rw = () => r(env)
  val add = base.add(lw, rw);
  e => { env = e; add() }
}
```

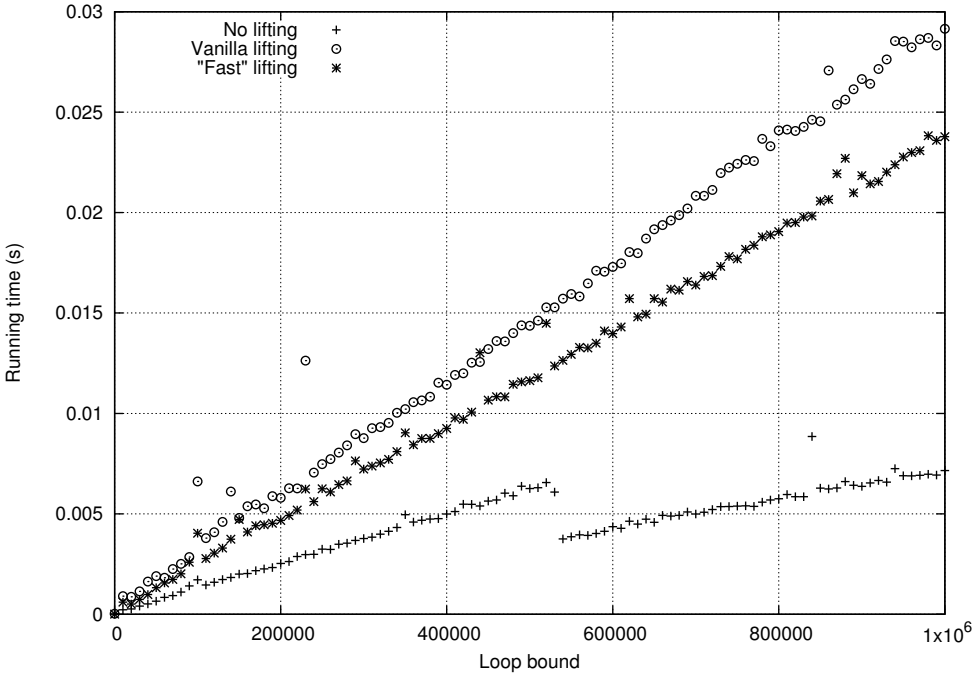


Figure 4.16: Executing `sum(n, add(lit(1), lit(2)))` without lifting, with vanilla lifting, and with "fast" lifting.

Instead of creating the lowered argument closures `lw` and `rw` within the body of the returned closure, they are hoisted to the level of `add` itself. Similarly, the call to `base.add` is also hoisted, avoiding additional runtime overhead. To ensure that the environment is propagated correctly, the returned closure assigns the received environment `e` to the mutable variable `env`.

The results of the micro-benchmark confirm the observed slowdown in the DeltaBlue benchmark. However, the primary cause seems to be additional call overhead, and not the allocation of closures, since that accounts for only a small part of the slowdown. Further research is needed to investigate ways of optimizing lifting interpreters.

4.9 Related Work and Discussion

In this section we discuss related work and then provide a qualitative assessment of implicit context propagation as a technique.

4.9.1 Related Work

We discuss the related work in three categories: modular interpreters, component-based language development, and, finally, other techniques to implicitly propagate context.

Modular Interpreters

In this section, we present other approaches to modular interpreters. We first elaborate on the concept of monad transformers since this technique is the most well-known approach for defining modular interpreters. We then discuss some limitations of the original presentation of monad transformers and existing work that addresses them. Finally, we review other functional programming approaches to modular interpreters.

Monad transformers The use of monads to structure interpreters is a well-known design pattern in functional programming. Monads, as a general interface for sequencing, allow idioms such as environments, stores, errors, and continuations to be automatically propagated. However, monads themselves do not allow these different effects to be combined. Liang et al. [LHJ95] consolidated much of the earlier work (e.g., [Esp95; Ste94]) on how *monad transformers* (MT) can be used to solve this problem. The complete presentation of modular interpreters in their work exposes a monadic interpretation function whose signature is `interp :: Term -> InterpM Value`. This interpreter is extensible because all three components – the term type, the value type and the monad – can be composed from individual components. Both the term and the value types can be modularly defined and later extended/composed using extensible unions, while the monad `InterpM` can be defined using monad transformers.

To illustrate modular interpreters in monadic style, the left column of Table 4.5 shows the `Arith`, `Binding` and `Storage` languages introduced in Section 4.4 in Haskell. For reference, the object algebra implementations are shown in the right column. The monadic interpreters are defined as instances of the type class `InterpC`, where the `interp` operation is defined. The code fragments assume that extensible unions are used to combine the syntactic data types of each language module and the resulting `Value` data type used in `InterpM Value`. As a result, the algebraic data types in Table 4.5 recurse on the open type `Term`. For instance, the `Arith` and `Binding` modules can be combined using the **`Either`** type constructor and a **`newtype`** definition of `Term` to "tie the knot":

```
type ArithBinding = Either Arith (Either Binding ())
```

```
newtype Term = Term ArithBinding
```

The extensible `Value` is defined similarly, and requires auxiliary functions for injecting values into and projecting values out of the value domain. The functions `returnInj` and `bindPrj` are helper functions representing the monadic `return` and `bind` functions performing injection and projection. For instance, the `Arith` language injects integers into the value domain. Similarly, `Binding` and `Storage` require closures and locations to be part of the domain, respectively. Here, however, we focus on the modular effects part of the presentation.

The key observation about Table 4.5 is that the `interp` functions return monadic values in the type `InterpM`, but each module imposes different constraints on the actual defintion of this type. For instance, the `Arith` module does not make any assumptions on `InterpM` except that it is a monad. The `Binding` module, however, requires that the monad supports the functions `rdEnv` and `inEnv` in order to obtain the current environment and evaluate an expression in a particular environment, respectively. Finally, the `Storage` language requires allocating, inspecting and updating locations.

To support both environments and a store for side effects, `InterpM` should be defined as a stack of monad transformers, each of which allows lifting the operations from one kind of monad into another one. For example, the following definition of `InterpM` suffices for the composition of the `Arith`, `Binding` and `State` interpreters (notice that at the bottom of the stack we have the identity monad `Id`):

```
type InterpM = EnvT Env (StateT Store Id)
```

Given a composition of the syntactic data types representing `Arith`, `Binding` and `Storage` the interpreters can be composed where "effect oblivious" code propagates the environment and store as defined in the monad transformers `EnvT` and `StateT`.

Note that the definition of `InterpM` is a global definition for all interpreter modules, which defines the set of effects and their composition once and for all. This means that the ordering of the effects and their interaction is defined and fixated at this point; it is not possible to change the interaction on a per module basis. Furthermore, any change to the `InterpM` type definition requires re-typechecking each module. As such, this formulation does not support separate compilation. It is also impossible to reuse a composition of interpreters as a black box component in further compositions.

Monad transformers are defined in a pair-wise fashion. This means that if there are feature interactions between two monads, they have to be explicitly resolved. With implicit context propagation, however, the interactions between effects are implicit, as they depend on the behavior of the context objects in the host language. Note however, that inadvertent feature interactions can always be explicitly resolved in our case by *overriding* lifted interpretations. An example of this is preventing capture of the dynamic environment as discussed in Section 4.4.2. Nevertheless, when the interactions are more complex (e.g., when the host language does not natively

Monad transformers in Haskell

```
data Arith
    = Lit Int
    | Add Term Term

instance InterpC Arith where
    interp (Add l r) = interp l `bindPrj` \i
                       -> interp r `bindPrj` \j
                       -> returnInj ((i+j)::Int)
    interp (Lit n) = returnInj n
```

```
data Binding
    = Lambda Name Term
    | Vari Name
    | Apply Term Term

instance InterpC Binding where
    interp (Lambda x b) = rdEnv >>= \env
                        -> returnInj $ Clos x b env
    interp (Vari x) = rdEnv >>= \env
                        -> case lookupEnv x env of
                             Just v    -> v

    interp (Apply e1 e2) = interp e1 `bindPrj` \f ->
                           case f of
                             Clos x e env ->
                              inEnv
                               (extendEnv (x, interp e2) env)
                               (interp e)
```

```
data Storage
    = Create
    | Update Term Term
    | Inspect Term

instance InterpC Storage where
  interp (Create) = do loc <- allocLoc
                       updateLoc (loc, returnInj (0::Int))
                       returnInj loc
  interp (Update c v) = interp c `bindPrj` \loc
                          -> interp v >>=  \val
                          -> updateLoc (loc, return val) >>
                             return val
  interp (Inspect c) = interp c `bindPrj` lookupLoc
```

Implicit context propagation in Scala

```
type Ev = () => Val

trait EvArith extends Arith[Ev] {
  def add(l: Ev, r: Ev)
    = () => IntVal(l() + r())

  def lit(n: Int)
    = () => IntVal(n)
}
```

```
type EvE = Env => Val

trait EvEBinding extends Binding[EvE] {
  def lambda(x: Str, b: EvE)
    = env => new Clos(x, b, env)

  def vari(x: Str)
    = env => env(x)

  def apply(e1: EvE, e2: EvE)
    = env => e1(env).apply(e2(env))
}
```

```
type EvS = Sto => Val

trait EvMStorage extends Storage[EvS] {
  def create()
    = st => {
        val c = new Cell(st.size + 1)
        st += c -> IntVal(0); c
      }
  def update(c: EvS, v: EvS)
    = st => {
        val c1: Cell = c(st)
        val v1 = v(st)
        st += c1 -> v1; c1
      }
  def inspect(c: EvS)
    = st => {
        val c1: Cell = c(st) ; st(c1)
      }
}
```

Table 4.5: Arithmetic, Binding and Storage language building blocks implemented in Haskell with monad transformers (on the left) and in Scala using object algebras (on the right).

	Monad transformers [LHJ95]	This work
Syntactic modularity	Extensible unions	Trait composition
Value extensibility	Extensible unions (with inj/prj)	Subtyping (with casts)
Context propagation	Monad transformers	Implicit propagation
Interpreter style	Monadic	Direct
Part of signature	Return type	Formal parameters
Purely functional	Yes	No
Type safe	Yes	Yes
Separate compilation	No	Yes
Effect interaction	Explicit	Implicit
Scope of interaction	Global, fixed	Locally overridable
Compositional propagation	No	Yes

Table 4.6: Comparing characteristics of modular interpreters using monad transformers vs. implicit context propagation

support the effects being modeled), the overriding code can be quite involved (cf. Section 4.4.4).

As a summary, Table 4.6 provides a qualitative appraisal of using monad transformers vs. implicit context propagation as presented in this chapter, in terms of a number of meta level characteristics. The table shows that both approaches have different strengths and weaknesses. The most apparent feature of monad transformers is that they represent a purely functional approach to modular interpreters; no side-effects are needed. Implicit context propagation, on the other hand, emphasizes extensibility and simplicity, at the cost, perhaps, of sacrificing purity and explicitness. In particular, implicit context propagation supports compositional propagation: a lifted interpreter can be lifted yet again, to propagate additional context. The lifting operation is oblivious as to whether the interpreter to be lifted is a base interpreter or has been lifted earlier. This is a crucial aspect for incremental, modular development of languages.

Extensible syntax and multiple interpretations Monad transformers in their original presentation do not feature separate compilation. Although we can define the language components in a modular fashion, at the moment of composition all the type synonyms (e.g. the type `InterpM`) who are referred by each module, must be resolved and therefore, the compilation is monolithic. Similarly, syntax definitions are not truly extensible either: adding data types to the extensible union requires recompilation of the existing interpreter code. To overcome these problems, Duponcheel [Dup95] extended the work of [LHJ95] by representing the abstract syntax of a language as algebras, and interpreters as catamorphisms over such algebras to cater for extensible

syntax. Additionally, this style support extensible operations as well, similar to the object algebra style employed in this text.

Other approaches to modular interpreters A different approach to extensible interpreters was pioneered by Cartwright and Felleisen [CF94]. They present *extended direct semantics*, allowing orthogonal extensions to base denotational definitions. In this framework, the interpreters execute in the context of a global authority which takes care of executing effects. A continuation is passed to the authority to continue evaluation after the effect has been handled. In extended direct semantics, the semantic function $\mathcal{M}$ has a fixed signature $Exp \to Env \to C$ where C is an extensible domain of computations. The fixed signature of $\mathcal{M}$ allows definitions of language fragments to be combined.

Kiselyov et al. [KSS13] generalized the approach of [CF94], allowed the administration functions to be modularized as well, and embedded the framework in Haskell using open unions for extensible syntax, and free monads for extensible interpreters. This approach to define modular interpreters excels at the definition of imperative embedded languages, where the monad sequencing operator is reused for the sequencing operators of the embedded language.

Component-Based Language Development

The vision of building up libraries of reusable language components to construct languages by assembling components is not new. An important part of the Language Development Laboratory (LDL) [HLR97] consisted of a library of language constructs defined using recursive function definitions. Heering and Klint considered a library of reusable semantic components as a crucial element of Language Design Assistants [HK00]. Our work can be seen as a practical step in this direction. Instead of using custom specification formalisms, our semantic components are defined using ordinary programming languages, and hence, are also directly executable.

More recently, Cleenewerck investigated reflective approaches to component-based development [Cle03; Cle07]. In particular, he investigated the different kinds of interfaces of various language aspects and how they interact. Implicit context propagation can be seen as a mechanism to address one such kind of feature interaction, namely the different context requirements of interpreters.

Directly related to our work is Mosses' work on component-based semantics [CMS+15; CMT14]. Languages are defined by mapping abstract syntax to fundamental constructs (*funcons*), which in turn are defined using I-MSOS [MN09], an improved, modular variant of Structural Operational Semantics (SOS) which also employs implicit context propagation. The modular interpreters of this chapter can be seen as the denotational, executable analog of I-MSOS modules. In fact, our implicit context propagation technique was directly inspired by the propagation strategies of I-MSOS.

Finally, first steps to apply object algebras to the implementation of extensible languages have been reported in [GPS14]. In particular, this introduced Naked object algebras (NOA), a practical technique to deal with the concrete syntax of a language using Java annotations. We consider the integration of NOA to the presented modular interpreter framework as future work. In particular, we want to investigate designs to support multiple concrete syntaxes for an abstract semantic component.

Implicit Propagation

Implicit propagation has been researched in many forms and manifestations. The most related treatment of implicit propagation is given by Lewis et al. [LLM+00], who describe implicit parameters in statically typed, functional languages. A difference to our approach is that implicit parameters cannot be retro-actively added to a function: a top-level evaluation function would still need to declare the extra context information, even though its value is propagated implicitly.

Another way of achieving implicit propagation in functional languages is using extensible records [Lei05]. Functions consuming records may declare only the fields of interest. However, if such a function is called with records containing additional fields, they will be propagated implicitly.

Implicit propagation bears similarity to dynamic scoping, as for instance, found in CommonLisp or Emacs Lisp. Dynamic scoping is a powerful mechanism to extend or modify the behavior of existing code [HP01]. For instance, it can be used to implement aspects [Cos03] or context-oriented programming [HCN08].

Another area where implicit propagation has found application is in language engineering tools. For instance, [Vis01] introduced scoped dynamic rewrite rules to propagate down dynamically scoped context information during a program transformation process. Similarly, the automatic generation of copy rules in attribute grammars is used to propagate attributes without explicitly referring to them [Kas84].

Finally, the implicit propagation conventions applied in the context of I-MSOS [MN09], have been implemented in DynSem, a DSL for specifying dynamic semantics [VNV15]. In both I-MSOS and DynSem, propagation is made explicit by transforming semantic specifications.

4.9.2 Discussion

Although most of the code presented throughout this chapter, as well as the code of the case studies, is written in Scala, it is easy to port implicit context propagation to other languages. For instance, Java 8 introduces default methods in interfaces, which can be used for trait-like multiple inheritance. These interfaces were used in the discussion of using dynamic proxies for automating lifting in Section 4.5.2. Without a trait-like composition mechanism, the technique can still be of use, except that extensibility would be strictly linear. This loses some of the appeal for constructing a

library of reusable semantic building blocks, but still enjoys the benefits of type safety and modular extension.

As we could conclude from the analysis of existing work on modular interpreters, the main strength of implicit context propagation is its simplicity. For context information other than read-only, environment-like parameters, we depend on the available mechanisms of the host language. For instance, read-write effects (stores) are modeled using mutable data structures (cf. Section 4.4.3). Other effects, such as error propagation, local backtracking (Section 4.4.4), non-local control flow (`break`, `continue`, `return`, etc.), and gotos and coroutines [All89] can be simulated using the host language's exception handling mechanism. Support for concurrency or message passing can be directly implemented using the host language's support for threads or actors (cf. [HO09]).

A limitation of implicit context propagation is that certain complex feature interactions may occur when simulating propagation patterns that are not native to the host language (such as transaction-reversing choice). There is always the option to override semantic definitions to resolve the interaction manually, but this can be quite involved. Semantic feature interactions are more explicitly addressed in the work on monad transformers or effect handlers.

Another drawback of implicit context propagation is that, even though the boilerplate code can be automatically generated, the user still has to explicitly specify which liftings are needed and compose the fragments herself. Instead of using the annotations, it would be convenient if one could simply extend a trait over the right signature and have the actual implementation completely inferred. For instance, instead of writing the `@lift` annotation described in Section 4.5, one would like to simply write:

```
trait Combined extends EvEBinding with Arith[EvE]
```

The system would then find implementations of `Arith[_]` to automatically define the required lifting methods right into the `Combined` trait. If multiple candidates exist, it would be an error. This is similar to how Scala implicit parameters are resolved [OAC+14]. We consider this as a possible direction for future work.

Another open challenge is the disambiguation of parameters with the same type when automating lifting. For instance, in the case of the Dynamic Scoping example (Section 4.4.2), one wants to explicitly indicate on the lifted carrier type `(Env, Env) => Val` which `Env` corresponds to the dynamic environment, and which one to the static one. In our current implementation, there is no way to specify this either with the macro approach or with dynamic proxies. When there are two parameters with the same type, the behavior of the `@lift` is currently undefined. In order to work around this, additional metadata (e.g., via annotations) should be provided in order to guide the disambiguation of same-type parameters and produce the right lifting.

4.10 Conclusion

Component-based language engineering would bring the benefits of reuse to the construction of software languages. Instead of building languages from scratch, they can be composed from reusable building blocks. In this work we have presented a design for modular interpreters that support a high level of reuse and extensibility. Modular interpreters are structured as object algebras, which support modular, type safe addition of new syntax as well as new interpretations. Different language constructs, however, may have different context information requirements (such as environments, stores, etc.), for the same semantic interpretation (evaluation, type checking, pretty printing, etc.).

We have presented **implicit context propagation** as a technique to eliminate this incompatibility by automatically *lifting* interpretations requiring n context parameters to interpretations accepting $n+1$ context parameters. The additional parameter is implicitly propagated, through the interpretation that is unaware of it. As a result, future context information does not need to be anticipated in language components, and opportunities for reuse are increased.

Implicit context propagation is simple to implement, does not require advanced type system features, fully respects separate compilation, and works in mainstream OO languages like Java. We have shown how the pattern operates in the context of overriding, mutable context information, exception handling, continuation-passing style, languages with multiple syntactic categories, generic desugaring and interpretations other than dynamic semantics. Furthermore, the code required for lifting can be automatically generated using a simple annotation-based code generator or lifting can be generically performed at runtime using dynamic proxies. We have illustrated the usefulness of implicit context propagation in an extensible implementation of a simple DSL for state machines. Our modular implementation of Featherweight Java with state shows that the pattern enables an extreme form of modularity, bringing the vision of a library of reusable language components one step closer.

Since lifting is based on creating intermediate closures, lifted interpreters can be significantly slower than directly implemented base interpreters. Running the well-known DeltaBlue benchmark on top of a modular interpreter for LambdaJS shows that lifting makes interpreters almost twice as slow. Further research is required to explore techniques to eliminate the additional call overhead during evaluation. One possible direction would be light-weight modular staging (LMS) [RO10; RO12] in Scala, although this approach would compromise separate compilation of base interpreters.

Other directions for further research include the integration of concrete syntax (cf. [GPS14]), and the application of implicit context propagation in the area of DSL engineering. We expect that DSL interpreters require a much richer and diverse set of context parameters, apart from the standard environment and store idioms. Finally,

we will investigate the design of a library of reusable interpreter components as a practical, mainstream analog of the library of fundamental constructs of [CMS+15; CMT14].

5 JEff: Objects for Effect

Modular interpreters with Implicit Context Propagation represent context using the side-effect facilities of the host language. This limits their composability as the feature interaction between the different effects is not explicit.

Effect handling is a way to structure and scope user-defined side-effects which is gaining popularity as an alternative to monads in purely functional programming languages. Languages with support for effect handling allow the programmer to define idioms for state, exception handling, asynchrony, backtracking etc. from within the language. Functional programming languages, however, operate within a closed world assumption, which prohibits certain patterns of polymorphism well-known from object-oriented languages. In this chapter we introduce JEff, an object-oriented programming language with native support for effect handling, to provide first answers to the question what it would mean to integrate OO programming with effect handling. We illustrate how user-defined effects could benefit from interface polymorphism, and present its runtime semantics and type system.

This chapter is based on the following published article:P. Inostroza and T. van der Storm. "JEff: Objects for Effect". In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2018. Boston, Massachussets, USA: ACM, 2018.

5.1 Introduction

Effect handlers [PP09; PP13] are a programming language mechanism to structure, scope, and compose side-effects in purely functional languages. Languages that support effect handling natively, such as Koka [Lei17], Eff [BP15], Frank [LMM17], allow side effects (state, IO, exceptions, coroutines, asynchrony, etc.) to be defined within the programming language, as libraries. Effect handlers have been touted as a simpler and more composable way of programming with effects than other techniques, such as, for instance, monads. As a result, perhaps, existing research on effect handling has mainly focused on functional programming languages.

Functional programming languages, however, operate on a closed world assumption [Cooo9]. As a result, certain patterns of modularity and reuse well-known in object-oriented languages do not apply. Yet the huge success of object-orientation in practice [Ald13] seems to suggest these mechanisms are valuable programming tools. This raises the question: what would it mean to integrate effect handling with object-oriented programming?

```
interface StdOut { eff Unit print(String s) }

class MyStdOut<T>(List<String> o) implements
  StdOut, Handler<Tuple<List<String>, T>, T> {...}

class Main() {
  Unit hello()@StdOut = StdOut::print("Hello world!")

  Tuple<List<String>, Unit> main() =
    with (new MyStdOut<Unit>([])) { this.hello() }
}
```

Figure 5.1: Skeleton of a simple JEff program

In this chapter we present first steps towards answering this question in the form of JEff, an object-oriented Java-like language without side-effects or inheritance, but with built-in support for programming with effect handlers. As a simple example, consider the code snippet shown in Figure 5.1. It defines an effect interface `StdOut` declaring a single effect method `print` for printing to the console. The class `MyStdOut` implements the interface and also marks itself as being a `Handler`; we defer the details of implementing handlers to Section 5.2. The `StdOut` interface is used in the `hello` method, which declares it as a required effect using `@`. In its body it calls the `print` method. Finally, the `main` method – which is pure – installs a `MyStdOut` handler with the **`with`**-construct, providing the printing capability to `hello`. Since `MyStdOut` models

the console as a list of strings, this, together with a trivial unit value, will be the result of `main`.

The example already highlights the most important aspect of JEff, namely that the declaration of an effect is decoupled from its implementation by a handler through interfaces. For instance, the `StdOut` interface can be implemented by multiple handler classes like `MyStdOut`, but encapsulating different internal representations. Nevertheless, the `hello` method only refers to the effect interface, and hence can be executed over any such implementation. JEff thus leverages dynamic dispatch and interface-based encapsulation (two of the corner-stones of OO programming) for defining effects.

The contributions of this chapter can be summarized as follows:

- We present JEff, the first object-oriented language with native support for effect handling (Section 5.2).
- We illustrate how common effects like exception handling (Section 5.2.2) and state (Section 5.2.3) are realized in JEff, thus providing an object-oriented introduction to effect handling, and how effect handling facilitates structuring interpreters (Section 5.2.4).
- We present the formal semantics of a core language of JEff, called Featherweight JEff (FJEff) (Section 5.3).
- We present the formal type system of FJEff and discuss its soundness properties (Section 5.4).

The syntax and semantics of FJEff have been modeled using PLT Redex [KCD+12], an embedded domain-specific language for mechanizing programming languages. The source code of the models is available online. [1]

The chapter is concluded with a discussion of open problems and directions for further research. We hope that JEff can contribute to a better understanding of effectful programming in the context of object-oriented languages without built-in notions of state, identity, or inheritance [Coo12].

5.2 JEff: Programming with Objects and Effects

5.2.1 Introduction

JEff is a Java-like language where custom effects can be defined as effect interfaces. The implementation of these effects is provided by handler classes. In this section we will explore the main characteristics of JEff using some illustrative scenarios.

Like Java, JEff features both classes and interfaces. It has multiple inheritance of interfaces, and it features both subtyping and parametric polymorphism (generics). To focus on the core aspects of combining object-orientation with effect handling,

[1] https://github.com/cwi-swat/jeff-model

JEff does not feature implementation inheritance for classes. [2] Furthermore, JEff is a side-effect free language: there is no mutation, I/O, exceptions, etc. These effects are to be provided by libraries of effect handlers that simulate such effects. Programmers can design and implement their own custom effects, and provide new handlers for existing ones.

Effect interfaces can be used to define the signature of the effect methods, such as `print` in the introduction. Effect methods are implemented in handler classes which provide the effect semantics. Effect methods can resume execution, transferring control back to the point where the effect method was called, using the special context variable **`there`** which denotes a special object conforming to the predefined interface `Resume` that defines a single method `resume`. If an effect method does not resume, the current execution stack is ignored and execution proceeds at the point where the handler was installed using the **`with`**-construct. The **`with`**-construct thus acts as a delimiter of the dynamic context in which effect invocations are handled. In order to specify what to do with the value that is produced after executing the body of the **`with`**-expression – in the manner of a wrapper – handler classes must implement the predefined `Handler` interface, whose single `return` method acts as the required wrapper.

JEff features a type and effect system that assigns types to expressions, methods, interfaces and classes. A method whose body calls an unhandled operation needs to be annotated with a type that declares the called effect method. The client of that method must, therefore, provide at some point a handler for that particular effect, similar to how checked exception declarations propagate in Java.

We now illustrate effectful progamming in JEff using the standard examples of exception handling and state.

5.2.2 Effectful Programming in JEff: Exception Handling

In JEff, an effect type is defined by declaring an effect interface. For instance, the following effect interface (indicated by the keyword **`eff`** on the effect method `raise`) defines the effect of raising an exception:

```
interface Raise { eff Nothing raise(String s) }
```

The `Raise` effect interface declares the effect method `raise`, as indicated by the **`eff`** keyword. This operation receives a string as an argument and returns an object of type `Nothing` (which represents the bottom type). In this case in particular, this return type signifies that the `raise` method will never return.

[2] As a reference, in [ALZ01] authors report that the interaction between exceptions, a particular case of effects, and inheritance is non-trivial.

The following code illustrates how the `raise` effect is triggered:

```
Int divide(Int x, Int y)@Raise = if (y != 0) x / y else
  Raise::raise("Division by zero")
```

The method signature of `divide` reflects its required effects in its signature through the `Raise` annotation. If the divisor is equal to zero, the method `divide` throws the exception. The syntax to call operations is the name of the type corresponding to the effect interface followed by two colons, the name of the operation, and the arguments.

The code above shows how to trigger an effect, but not how to handle it. In order to provide an interpretation to the `raise` effect, a handler object must be installed using the **`with`**-construct. This handler object must be an instance of a class that implements the `Raise` interface. For instance, the following expression installs a handler `h` to handle the effects triggered by `divide`:

```
with (h) { divide(4, 0) }
```

The **`with`**-expression acts as a dynamic scoping construct so that the `raise` invocation becomes a method call on the handler object `h`.

Since all the effects invoked in the code within the context of a **`with`**-construct are eventually handled, the body of the **`with`**-construct evaluates to a value. Handlers may capture this value and transform it before it is returned as the result of the **`with`**-expression itself. This is realized by the requirement that all handler objects must be instances of classes that implement the predefined `Handler` interface:

Definition 1 (Handler)

```
interface Handler<Out, In> { Out return(In in) }
```

The `Handler` interface declares a single `return` method that captures the result of the **`with`**-construct when no more effects are triggered in its body. The two type parameters `In` and `Out` capture the type of the body of the **`with`** expression, and the type of the **`with`** expression itself, respectively. There is nothing special about the `Handler` interface; it simply functions as the interface between the **`with`**-construct and its body and context, similar to how the `Iterable` interface interacts with the **`for`**-construct in Java. Note however, that the `return` method must be pure, since it has no effect annotations.

Now let's look at a potential implementation of a handler for `Raise`. The following `DefaultRaise` class defines a handler for the `raise` effect which simply returns a default value in case of an exception. The default value is provided when the class is instantiated through the `x` field. This default value is used as the value of the **`with`** expression in case its body raises an exception.

```
class DefaultRaise<T>(T x) implements Raise, Handler<T, T> {
  T return(T t) = t
  eff Nothing raise(String s) = this.x
}
```

In that case, both type parameters of `Handler` coincide and correspond to the type parameter `T`. This means that if an object of class `DefaultRaise` is used as a handler in a **`with`** expression, both the handled body of the expression and the **`with`** expression should have the same type. The `return` method, in this case, is the identity function. Hence, if the body does not raise an exception, as in **`with`** `(`**`new`** `DefaultRaise<Int>(-1)) { divide(4, 2) }`, the `return` method of the handler object will be called with the value returned by the body. In this case the value of the **`with`** expression will therefore be `2`.

Looking closely at the implementation of `raise`, however, makes it clear that effect methods are special, since the type of their body does not match the declared return type at all. In fact, in this case, the return type is `Nothing`, whereas the type of the body expression is `T`! The reason for this is that the declared return type corresponds to the type of value that will be sent back to the calling context upon resumption using `resume`. The type of the body of an effect method should always correspond to the `Out` type parameter of the `Handler` interface. Since in this case, the `raise` method does not resume, it simply returns a value of that type, the default value.

The `divide` method only refers to the `Raise` effect interface, so it can be run in the context of any number of handler implementations of the `Raise` interface. For instance, here is another implementation of the `Raise` interface, where the result of a computation is wrapped in a `Maybe` (option) type:

```
class MaybeRaise<T>() implements Raise, Handler<Maybe<T>, T> {
  Maybe<T> return(T t) = new Some<T>(t)
  eff Nothing raise(String s) = new None()
}
```

In `MaybeRaise`, the `In` type parameter corresponds to `T` but the `Out` type parameter, that is, the one that corresponds to the type of the **`with`**-expression, is `Maybe<T>`. In this case, the `return` method wraps the value resulting from the evaluation of the **`with`**-body in a `Some` object. Dually, the `raise` method produces the empty value **`new`** `None()`. Note again that the `None` object will be the result of the **`with`**-expression.

We have seen that the `Nothing` return type of `raise` means that the method will never "return", in other words, that it will never transfer control back to the point of invocation. In the next section we illustrate the scenario in which effect methods resume execution at the point where they were called.

5.2.3 Resuming After Handling: State

When the `raise` effect is triggered in the body of a **`with`** expression, the normal flow of computation is aborted and evaluation proceeds at the level of the **`with`** expression. Most effects, however, require resuming execution at the point where the effect was invoked, after handling. JEff realizes resumption through the special variable **`there`**. The **`there`** object is implicitly brought in scope when an effect method executes, just like **`this`** is available in all method executions.

The type of **`there`** is defined by the `Resume` interface:

Definition 2 (Resume)

```
interface Resume<T, Out, In> { Out resume(T x, Handler<Out, In> h) }
```

The `resume` method accepts two arguments. The first argument represents the value that is sent back to the calling context as the result of the effect invocation. The second argument represents the (possibly new) handler to install for the remainder of the execution. The concrete types for the type parameters are inferred from the context, but note that the second argument always needs to be a `Handler`.

Figure 5.2 shows how the state effect can be defined in JEff. The `State` effect interface (left) defines two operations: `get`, to retrieve the state, and `put` to update the state. The interface is generic in what is stored, and it abstracts from how state itself is represented.

The right-hand side of the figure shows the handler class `TupleState`, which implements the `State` interface and the `Handler` interface. Both `get` and `put` resume the computation using the **`there`** object. The method `get` resumes execution with the current state **`this`**`.s`, within the context of the current handler object **`this`**. Alternatively, `put` resumes with the unit value `()`, and installs a new handler by constructing a new `TupleState` object with the updated state `x`. Note how the type of the first argument to the `resume` method always corresponds to the declared return type of the effect methods. Finally, `TupleState` defines the `return` method from the `Handler` interface, wrapping the current state **`this`**`.s` and result of the **`with`**-body `x` in a tuple.

The `State` effect and the `TupleState` handler can be used as follows:

```
Unit countDown()@State<Int> =
  Int i = State<Int>::get();
  if (i >= 0) {
    State<Int>::put(i - 1);
    this.countDown();
  }
```

```
with (new TupleState<Int,Unit>(2)) {
  this.countDown()
}
// evaluates to Tuple(0, ())
```

```
interface State<T> {
  eff T get()
  eff Unit put(T x)
}
```

```
class TupleState<T,U>(T s) implements State<T>,
  Handler<Tuple<T,U>,U> {
  eff T get() = there.resume(this.s, this)
  eff Unit put(T x) = there.resume((), new TupleState(x));
  Tuple<T,U> return(U x) = new Tuple<T,U>(this.s, x)
}
```

Figure 5.2: The `State` effect interface (left), and a handler implementation `TupleState` using tuples (right)

The method `countDown` requires the `State` effect over integers, as witnessed by the annotation. It simply decreases the stored value until it is zero. This method can then be invoked by bringing `TupleState` handler in scope using **`with`**, as shown on the right.

Note again that `countDown` is independent from any implementation of state, and only depends on the effect interface `State`, in the same way that the `divide` method was only dependent on the effect interface `Raise`, and not on any particular implementation. Decoupling the interface of effect operations from handler operations allows client code to be independent of concrete handlers.

For instance, consider the following handler for state, which maintains a history of updates:

```
class LogState<T, U>(List<T> log) implements State<T>,
  Handler<Tuple<List<T>, U>, U>> {
  eff T get() = there.resume(this.log.last(), this)
  eff Unit put(T x) = there.resume((), new LogState<T,U>(log.append(x)));
  Tuple<List<T>, U> return(U x) = new Tuple<List<T>, U>(this.log, x)
}
```

Note that the `State` interface is still implemented over type parameter `T`, but the handler itself now uses `List<T>` as its representation to save the history of values that have been assigned to the state. Calling `countDown` in the context of a `LogState` handler, as in **`with`** `(new LogState<Int, Unit>(2)) { countDown() }`, will evaluate to the value `Tuple([2, 1, 0], ())`.

5.2.4 Structuring Effectful Interpreters

One of the benefits of object-oriented programming is open extensibility of data types [Cooo9]. Given an interface defining a data type, (third-party) programmers can add new representation variants to the type without changing (or even recompiling) existing code. One use case where this is valuable is extensible AST-based interpreters.

```
interface Exp {
  Val eval()@Store,Env
}

class Lit(Val v)
    implements Exp {
  Val eval() = this.v
}

class Var(String x)
    implements Exp {
  Val eval()@Env =
    Env::get().lookup(this.x)
}
```

```
class Deref(Exp cell)
    implements Exp {
  Val eval()@Store =
    Store::get(cell.eval())
}

class Assign(Exp loc, Exp val) implements Exp {
  Val eval()@Store =
    Store::put(this.loc.eval(),
                  this.val.eval())
}

class Let(String x, Exp v, Exp b) implements Exp {
  Val eval()@Env =
    with (Env::get().extend(x, v.eval())){
      b.eval()
   }
}
```

Figure 5.3: Modular interpreters

Figure 5.3 shows the definition of an `Exp` data type with five classes realizing different kinds of expressions. The `Exp` interface defines a single `eval` method, annotated with `Store` and `Env` (environment) effect types. Figure 5.4 shows example implementations of `Env` and `Store`. Notice that both the `Env` and the `Store` classes are a data structure and a handler at the same time. In particular, they represent a domain-specific type of handler, that in this case is suitable for the "interpretation of expressions" domain. Because of this, we have fixed the incoming type of the handler to `Val`, which brings as consequence that any **`with`**-expression installing these handlers will enclose a value-producing expression. In this manner, domain-specific handlers can make code enclosed by **`with`**-expressions domain-specific.

Coming back to Figure 5.3, each concrete class implements the `Exp` interface.[3] The `eval` method in `Lit` does not use any effect, since it simply returns the field `v`. The `Var` class, however, requires `Env` reader effect to lookup bindings for variables. The classes `Deref` and `Assign` use the `Store` effect (similar to `State` of Figure 5.2) to realize cell dereferencing and assignment, respectively. Finally, the `Let` class models a lexically scoped binding construct, by first obtaining the current environment, extending it with a binding for `x`, and providing it as context for the evaluation of the body `b`.

Even though the `Exp` interface fixes the effect privileges of all interpreters, the effect handling mechanism of JEff makes it unnecessary to accept and propagate stores and environments explicitly, which would be needed in, e.g., Java, even when they are

[3]The effect annotations in the implementation classes specify only the effects that are actually used in their body. This is valid due to JEff's definition of overriding.

```
class Map<T, U>() {
  U get(T t)@Raise = ...
  Map<T, U> put(T t, U u) = ...
}

class Env(Map<String,Val> map = new Map<String,Val>()) implements Handler<Val, Val> {

  eff Env get() = there.resume(this, this)
  Val return(Val v) = v
  Val lookup(String x) = with (new DefaultRaise<Val>(new Nil())) { this.map.get(x) }
  Env extend(String x, Val v) = new Env(this.map.put(x, v))
}

class Store(Int id = 0, Map<Val,Val> map = new Map<Val,Val>())
  implements Handler<Tuple<Store, Val>, Val> {

  eff Val get(Val c) =
    there.resume(with (new DefaultRaise<Val>(new Nil())) { this.map.get(c) }, this)
  eff Val put(Val c, Val v) = there.resume(v, new Store(this.id, this.map.put(c, v)))
  eff Val alloc() = {
    Cell c = new Cell(this.id);
    there.resume(c, new Store(this.id + 1, this.map.put(c, new Nil())))
  }
  Tuple<Store, Val> return(Val v) = new Tuple<Store,Val>(this, v)
}
```

Figure 5.4: This code provides implementations of the `Env` and `Store` types. Both `Env` and `Store` use an immutable `Map` class. Note how both `Env` and `Store` use the `DefaultRaise` handler to deal with missing keys. To make client code less unwieldy, JEff features default values for fields as notational short-hand; since object construction is always pure in JEff, the only allowed expressions as default values are object constructor calls with **`new`** or literals.

not used. Note also that the set of effect privileges on `eval` must be recursively closed, since `eval` methods might call effectful methods on dependencies. In this example all dependencies receiving method calls (i.e., `cell`, `loc`, `val`, `v`, and `b`) are `Exp` objects themselves, so this is trivially satisfied.

The AST classes of Figure 5.3 could then be used with the following `run` method:

```
Tuple<Store<Val>,Val>
  run(Exp<Val> exp) =
  with (new Store<Val>()) {
    with (new Env<Val>()) {
      exp.eval()
    }
}
```

```
run(
  new Let<Val>("x", new Cell(0),
    new Assign<Val>(
      new Var<Val>("x"),
      new Lit<Val>(new Num(42)))))
// evaluates to
// Tuple(Store(0, Map(Cell(0) -> Num(42))), Num(42))
```

The `run` method receives an expression and evaluates `exp` in the context of a fresh store and environment. The result will be a tuple of the store and the result. Note that

`run` is pure, since all effects of `eval` are handled. Note further that `Store` and `Env` act both as handlers for their respective effects as well as data containers for cell-value and name-value pairs respectively; it is perfectly fine for JEff effect handlers to have ordinary methods as well.

Although the effect signature of `eval` is not extensible itself, it is still possible to extend the code of Figure 5.3 with new AST classes, without changing any of the existing classes, and without having to modify `run`. Achieving the same at the level of effects remains an open research question (see, e.g., [IvdS17], for a discussion and non-effectful solution).

5.2.5 Ad Hoc Overloading of Effects

A key feature offered by JEff's effect system is that dispatch of an effect invocation happens through both subtype polymorphism and parametric polymorphism (generics). This means that multiple handlers for the same effect interface can be in scope for a fragment of code, and, more importantly, they can be distinguished as well, since the effect invoking code explicitly qualifies the invoked effect.

This unique feature of JEff is illustrated in Figure 5.5. The left-hand side shows a `ToStr` effect, which allows converting some value of type `X` to be converted to a string. Without going into details of implementing handlers for this effect, the left-hand side shows three specializations of this effect: two sub-interfaces (`IntToStr` and `IntToHex`) representing different ways of converting an integer to a string, and a specialization for converting booleans to strings (`BoolToStr`).

The right-hand side of Figure 5.5 shows two methods invoking `ToStr` effects. The method `main1`, requires abstract `ToStr` effects, instantiated for both `Int` and `Bool`; it returns the concatenation of converting both method parameters to string. So the same effect interface is required to be in scope, instantiated over different argument types.

Assuming we have handler implementations `AnIntToStr`, and `ABoolToStr`, the `main1` can be invoked as follows:

```
with (new AnIntToStr<String>())) {
  with (new ABoolToStr<String>()) {
    this.main1(42, true)
  }
} // ⇒ "42 true"
```

The invocation `toString(n)` will dispatch to `AnIntToStr`, because `AnIntToStr` is a subtype of `ToString<Int>`. Similarly, `toString(b)` will dispatch to `ABoolToStr`, because `ABoolToStr` is a subtype of `ToString<Bool>`. Note, however, that `main` still only refers to `ToString`, so still benefits from subtype polymorphism. For instance, an handler implementation

```
interface ToStr<X> {
  eff String toString(X x);
}

interface IntToStr extends
  ToStr<Int> { }

interface IntToHex extends
  ToStr<Int> { }

interface BoolToStr extends
  ToStr<Bool> { }
```

```
// same effect, different type parameter
String main1(Int n, Bool
  b)@ToStr<Int>, ToStr<Bool>
  = ToStr<Int>::toString(n) + " "
  + ToStr<Bool>::toString(b)

// same effect, same type parameter
String main2(Int n)@IntToStr, IntToHex
  = IntToStr::toString(n) + ": "
  + IntToHex::toString(n)
```

Figure 5.5: `ToStr` effects

of `IntToHex` (Figure 5.5) could also be installed to adapt the behavior of `main1` from the outside.

Figure 5.5 shows three interfaces specializing the `ToStr` effect interface. This allows us to go even one step further, and distinguish between different handlers over the same effect interface and argument type(s). This is illustrated in `main2` on the right of Figure 5.5.

In this case, the method prints out a single number using different presentations, the default one (`IntToStr`), and another one, in this case `IntToHex`. Again, assuming two handler implementations of these respective interfaces, allows `main2` to be invoked as follows:

```
with (new AnIntToStr<String>()) {
  with (new AnIntToHex<String>()) {
    this.main2(42)
  }
} // ⇒ "42: 2A"
```

Again, the combination of subtyping and generic type instantiation provide additional flexibility in effectful programming.

5.3 Dynamic Semantics

In this section we present the semantics of Featherweight JEff (FJEff), a core calculus focusing on JEff's more distinctive semantic characteristics.

5.3.1 Syntax

T, S, U, V, W	::=	$X \mid N$
N, P, Q	::=	$C\langle\overline{T}\rangle$
L	::=	**class** $C\langle\overline{X} \triangleleft \overline{N}\rangle(\overline{T}\,\overline{f}) \triangleleft \overline{N}\{\overline{M}\}$ | **interface** $C\langle\overline{X} \triangleleft \overline{N}\rangle \triangleleft \overline{N}\{\overline{H}\}$
Ξ	::=	$\overline{N}$
H	::=	$[\textbf{eff}]\langle\overline{X} \triangleleft \overline{N}\rangle T\, m(\overline{T}\,\overline{x})@\Xi$
M	::=	$H{=}e$
e, d	::=	$x \mid e.f \mid e.\langle\overline{T}\rangle m(\overline{e}) \mid$ **new** $N(\overline{e}) \mid N::\langle\overline{T}\rangle m(\overline{e}) \mid$ **with**$(e)\{e\}$
v, w	::=	**new** $N(\overline{v})$ | **new** `Resume` $\{$`resume`$(x, x) = e\}$

Figure 5.6: FJEff syntax

The grammar of FJEff is shown in Figure 5.6. Metavariables B, C and D range over class and interface names; f and g over field names; m over method names; X and Y over type variables; and finally x and y over variables, including the special variables **this** and **there**. Comma-separated sequences are represented by overlined symbols, for example $\overline{M}$ represents a sequence of method declarations. Consecutive sequences represent sequencing tuples of elements as in $\overline{C}\ \overline{f}$ for field declarations.

Types can be either type variables X or non-variable types N. A type definition L can be either a class or an interface definition. Class definitions consist of the class's name, an optional list of bounded type parameters $X \triangleleft N$, followed by a possibly empty list of fields, and an optional list of implemented interfaces (preceded by $\triangleleft$), and finally a list of methods. An interface declaration follows the same structure, but does not feature fields and may only contain method headers. An effect set Ξ is syntactically a list of non-variable types where order is irrelevant. Method headers H may be marked as being an effect method using the keyword **eff** and consist further of the return type T, the method name m, a list of formal parameters $\overline{C}\ \overline{x}$, and an optional sequence of effect annotations Ξ. Method definitions M consists of a header and a body expression `e`.

Expressions e can be a variable, field reference, method invocation, object instantiation, effect method invocation, or a **with**-expression. The first four are standard. An effect method call consists of the type of an interface or class that defines the effect method, followed by two colons and then the name of the method and the arguments. The handling expression **with** contains two sub-expressions. The first one corresponds to the (handler) object that will handle (some of) the effects that will be be triggered during execution of the body. Finally, objects v, w correspond to a fully-evaluated instantiation expressions with **new**, or resumption object containing a definition of the `resume` method as defined by the `Resume` interface (see Definition 2).

$$
\begin{array}{lcl}
E & ::= & [] \mid E.f \mid E.{<}\overline{T}{>}m(e...) \mid v.{<}\overline{T}{>}m(v...E\,e...) \mid \textbf{new}\ N(v...E\,e...) \mid N::{<}\overline{T}{>}m(v...E\,e...) \\
 & \mid & \textbf{with}(E)\{e\} \mid \textbf{with}(v)\{E\} \\
X_N & ::= & [] \mid X_N.f \mid X_N.{<}\overline{T}{>}m(e...) \mid v.{<}\overline{T}{>}m(v...X_N\,e...) \mid \textbf{new}\ N(v...X_N\,e...) \mid N::{<}\overline{T}{>}m(v...X_N\,e...) \\
 & \mid & \textbf{with}(X_N)\{e\} \mid \textbf{with}(\textbf{new}\ Q(v...))\{X_N\} \qquad \text{when} \bullet \vdash Q \not<: N
\end{array}
$$

Figure 5.7: Evaluation contexts for reduction semantics

$$\frac{\mathit{fields}(N){=}\overline{T}\,\overline{f}}{\textbf{new}\ N(\overline{v}).f_i \longrightarrow v_i}\ \textsc{r\_field}$$

$$\frac{\mathit{mbody}(m{<}\overline{V}{>},N){=}\overline{x}.e}{\textbf{new}\ N(\overline{v}).{<}\overline{V}{>}m(\overline{w}) \longrightarrow [\textbf{new}\ N(\overline{v})/\mathtt{this}, \overline{w}/\overline{x}]e}\ \textsc{r\_invk}$$

$$\frac{}{\textbf{new}\ \mathtt{Resume}\{\mathtt{resume}(x_v, x_h) = e\}.\mathtt{resume}(v_1, v_2) \longrightarrow [v_1/x_v, v_2/x_h]e}\ \textsc{r\_resume}$$

$$\frac{}{\textbf{with}(\textbf{new}\ N(\overline{v}))\{v\} \longrightarrow \textbf{new}\ N(\overline{v}).\mathtt{return}(v)}\ \textsc{r\_return}$$

$$\frac{\begin{array}{c}\bullet \vdash Q{<:}N \qquad \mathit{mbody}(m{<}\overline{V}{>},Q){=}\overline{x}.e \\ v_k = \textbf{new}\ \mathtt{Resume}\{\mathtt{resume}(x_v, x_h){=}\textbf{with}(x_h)\{X_N[x_v]\}\}\end{array}}{\textbf{with}(\textbf{new}\ Q(\overline{v}))\{X_N[N::m{<}\overline{V}{>}(\overline{w})]\} \longrightarrow [\textbf{new}\ Q(\overline{v})/\mathtt{this}, v_k/\mathtt{there}, \overline{w}/\overline{x}]e}\ \textsc{r\_eff\_invk}$$

Figure 5.8: Reduction rules

5.3.2 Reduction Semantics

We present the operational semantics of FJEff using Felleisen-style evaluation contexts [WF94]. We use two evaluation contexts, E and X_N, shown in Figure 5.7.

Figure 5.8 shows the evaluation rules of FJEff. The semantics uses two auxiliary lookup functions *fields* (to map field names to field indices) and *mbody* (to obtain the body of a method); their definition can be seen in Figure 5.9.

The E context is the usual context for call-by-value evaluation, while the X_N context is used for the context that a handler delimits. It follows the same structure as E except that it matches **with**-expressions so that the body X_N does not contain **with**-expressions which handle effect N. In other words, it captures the nearest

Field lookup:

$$\frac{\textbf{class}\ C{<}\overline{X}\triangleleft\overline{N}{>}(\overline{S}\,\overline{f})\triangleleft\overline{P}\{\ldots\}}{\mathit{fields}(C{<}\overline{T}{>}) = [\overline{T}/\overline{X}]\overline{S}\,\overline{f}}\ \textsc{f\_class}$$

Method type lookup:

$$\frac{\textbf{class}\ C{<}\overline{X}\triangleleft\overline{N}{>}(\overline{S}\,\overline{f})\triangleleft\overline{P}\{\overline{M}\}\qquad[\textbf{eff}]{<}\overline{Y}\triangleleft\overline{Q}{>}\,U\,m(\overline{U}\,\overline{x})\texttt{@}\Xi_m{=}e\ \in\ \overline{M}}{\mathit{mtype}(m, C{<}\overline{T}{>}) = [\textbf{eff}][\overline{T}/\overline{X}](<\overline{Y}\triangleleft\overline{Q}{>}\overline{U}\rightarrow\overline{U}\texttt{@}\Xi_m)}\ \textsc{mt\_class}$$

Method body lookup:

$$\frac{\textbf{class}\ C{<}\overline{X}\triangleleft\overline{N}{>}(\overline{S}\,\overline{f})\triangleleft\overline{P}\{\overline{M}\}\qquad[\textbf{eff}]{<}\overline{Y}\triangleleft\overline{Q}{>}\,U\,m(\overline{U}\,\overline{x})\texttt{@}\Xi_m{=}e\ \in\ \overline{M}}{\mathit{mbody}(m{<}\overline{V}{>}, C{<}\overline{T}{>}) = \overline{x}.[\overline{T}/\overline{X}, \overline{V}/\overline{Y}]e}\ \textsc{mb\_class}$$

Figure 5.9: Auxiliary definitions

enclosing **with**-expression that is able to handle effect N. The side-condition ensures this by disallowing Q to be a subtype of N.

Rule R_FIELD defines field lookup. It maps a field name to its position in the sequence of values in an object using the *fields* lookup function. Rule R_INVK defines the semantics for ordinary method invocation by obtaining the body of the method m using the *mbody* lookup function. The expression then reduces to the method body e with substitutions applied for **this** and the formal parameters $\overline{x}$. The rule R_RESUME is similar to R_INVK but works on synthesized resumption objects.

There are two cases for **with**-expressions. The first one, R_RETURN, deals with the case in which the body expression has been fully evaluated to a value. In that case the expression reduces to a `return` invocation expression on the handler object.

The second rule R_EFF_INVK applies when **with**-bodies contain remaining effect method invocations, and thus implements effect dispatch. The context X_N ensures that the Q object is the directly enclosing handler servicing the effect N. The effect method m is looked up in the Q object, and the **with**-expression is reduced to its body e with substitutions applied for **this**, **there**, and the formal parameters $\overline{x}$. The special variable **there** is substituted for a resumption object v_k whose `resume` method installs the handler x_h and continues execution of context X_N with x_v plugged in as the result of the effect method. As a consequence, the value of an effect method invocation at the call site will be x_v whenever the effect method resumes.

Compared to other calculi for effects and handlers that are of functional nature, e.g. [Lei17; LMM17], FJEff's rule for effect dispatch is special in that the syntax for the

Bound of type:

$$bound_\Delta(X) = \Delta X \qquad bound_\Delta(N) = N$$

Subtyping:

$$\frac{}{\Delta \vdash T <: T}\ \text{S\_REFL} \qquad \frac{}{\Delta \vdash X <: \Delta(X)}\ \text{S\_VAR} \qquad \frac{\Delta \vdash S <: T \qquad \Delta \vdash T <: U}{\Delta \vdash S <: U}\ \text{S\_TRANS}$$

$$\frac{\textbf{class}\ C\langle \overline{X} \triangleleft \overline{N} \rangle(\ldots) \triangleleft \overline{P}\{\ldots\} \qquad Q \in \overline{P}}{\Delta \vdash C\langle \overline{T} \rangle <: [\overline{T}/\overline{X}]Q}\ \text{S\_CLASS\_M}$$

$$\frac{\textbf{interface}\ C\langle \overline{X} \triangleleft \overline{N} \rangle \triangleleft \overline{P}\{\ldots\} \qquad Q \in \overline{P}}{\Delta \vdash C\langle \overline{T} \rangle <: [\overline{T}/\overline{X}]Q}\ \text{S\_IFACE}$$

Well-formed types:

$$\frac{}{\Delta \vdash \texttt{Object}\ \text{ok}}\ \text{WF\_OBJECT} \qquad \frac{X \in dom(\Delta)}{\Delta \vdash X\ \text{ok}}\ \text{WF\_VAR}$$

$$\frac{\textbf{class}\ C\langle \overline{X} \triangleleft \overline{N} \rangle(\ldots) \triangleleft \overline{P}\{\ldots\} \qquad \Delta \vdash \overline{T}\ \text{ok} \qquad \Delta \vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{N}}{\Delta \vdash C\langle \overline{T} \rangle\ \text{ok}}\ \text{WF\_CLASS}$$

$$\frac{\textbf{interface}\ C\langle \overline{X} \triangleleft \overline{N} \rangle \triangleleft \overline{P}\{\ldots\} \qquad \Delta \vdash \overline{T}\ \text{ok} \qquad \Delta \vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{N}}{\Delta \vdash C\langle \overline{T} \rangle\ \text{ok}}\ \text{WF\_IFACE}$$

Predefined interfaces:

```
interface Object { }

interface Handler<Out, In> { Out return(In in) }

interface Resume<T, Out, In> { Out resume(T x, Handler<Out, In> h) }
```

Valid method overriding:

$$\frac{mtype(m, N) = \langle \overline{Z} \triangleleft \overline{Q} \rangle \overline{U} \to U_0 \texttt{@} \Xi_0 \text{ implies } \overline{P}, \overline{T} = [\overline{Y}/\overline{Z}](\overline{Q}, \overline{U}) \text{ and } T_0 = [\overline{Y}/\overline{Z}]U_0 \text{ and } \overline{Y} <: \overline{P} \vdash [\overline{Y}/\overline{Z}]\Xi_0 \preceq \Xi}{override(m, N, \langle \overline{Y} \triangleleft \overline{P} \rangle \overline{T} \to T_0 \texttt{@} \Xi)}$$

Figure 5.10: Subtyping and type well-formedness rules

invocation includes an effect qualifier N that enables effect selection using subtyping and parametric polymorphism, as it has been discussed in the pretty-printing example of Section 5.2.5. In R_EFF_INVK, it is clear that the handler Q that is selected among those in the runtime stack, is the one that is a subtype of the type specified by qualifier N. The pretty-printing example illustrates the consequences of this language design and the opportunities that become available in terms of new patterns for structuring effectful code, unavailable in functional languages with effects.

5.4 Type System

5.4.1 Introduction

JEff is a statically typed language with a nominal type system, where both classes and interfaces introduce types, arranged in a subtype lattice. In this section we present the type system of the simplified core language FJEff. The type system of FJEff is mostly standard, except that it ensures that JEff methods are effect-safe: whenever an effect is triggered it is either handled using a syntactically enclosing **`with`**-construct, or the enclosing method is annotated with a type defining the effect. The type system further makes use of the `Handler` and `Resume` interfaces for checking effect method declarations, and the **`with`**-construct itself. The definitions for method, method header, class and interface typing are shown in Figure 5.11. The rules are similar to the typing rules used in Featherweight Generic Java [IPW01] and employ auxiliary definitions for subtyping, well-formedness and overriding, shown in Figure 5.10.

Below we describe in detail the type rules for method definitions and expressions.

5.4.2 Method Typing

Figure 5.11 shows the two typing rules for method definitions in classes. The rule T_METH_REG checks the validity of ordinary, non-effect method declarations. The body of the method e_0 is type checked in context of the effect set Ξ_m, the type variable environment Δ (derived from the type parameters of the method and class), and an initial type environment defining the type of **`this`**. The set Ξ_m can be seen as the set of effect privileges available in the body of m. A method declaration is then valid if its header is valid and the type of e_0 is a subtype of the declared return type T.

The second rule defines the type correctness of effect methods in a similar fashion. The first difference, however, is that, in this case, C needs to implement the `Handler` interface. Second, the initial type environment also defines the type of the **`there`** variable in terms of the `Resume` interface. Finally, since the return type of an effect method corresponds to the type of the value used in resumptions, in this case the derived type of e_0 (S) must be a subtype of T_{out}, the return type of the `return` method defined by C.

Method typing:

$$\frac{\Delta = \overline{X}<:\overline{N}, \overline{Y}<:\overline{P} \qquad \Xi_m ; \Delta ; \overline{x}:\overline{T}, \texttt{this}:C<\overline{X}> \vdash e_0 : S \qquad \Delta \vdash S<:T \qquad <\overline{Y}\triangleleft\overline{P}>T\ m(\overline{T}\,\overline{x})@\Xi_m \text{ OK IN } C<\overline{X}\triangleleft\overline{N}>}{<\overline{Y}\triangleleft\overline{P}>T\ m(\overline{T}\,\overline{x})@\Xi_m = e_0 \text{ OK IN } C<\overline{X}\triangleleft\overline{N}>} \quad \text{T\_METH\_REG}$$

$$\frac{\bullet \vdash C<\overline{X}> <: \texttt{Handler}<T_{\text{out}}, T_{\text{in}}> \qquad \Delta = \overline{X}<:\overline{N}, \overline{Y}<:\overline{P} \qquad \Xi_m ; \Delta ; \overline{x}:\overline{T}, \texttt{this}:C<\overline{X}>, \texttt{there}:\texttt{Resume}<T, T_{\text{out}}, T_{\text{in}}> \vdash e_0 : S \qquad \Delta \vdash S<:T_{\text{out}} \qquad \texttt{eff}\ <\overline{Y}\triangleleft\overline{P}>T\ m(\overline{T}\,\overline{x})@\Xi_m \text{ OK IN } C<\overline{X}\triangleleft\overline{N}>}{\texttt{eff}\ <\overline{Y}\triangleleft\overline{P}>T\ m(\overline{T}\,\overline{x})@\Xi_m = e_0 \text{ OK IN } C<\overline{X}\triangleleft\overline{N}>} \quad \text{T\_METH\_EFF}$$

Method header typing:

$$\frac{\texttt{class}\ C<\overline{X}\triangleleft\overline{N}>(\ldots)\triangleleft Q_1 \ldots Q_{\text{n}}\{\ldots\} \qquad \overline{X}<:\overline{N}, \overline{Y}<:\overline{P} \vdash \overline{T}, T, \overline{P}, \Xi_m \text{ ok} \qquad \mathit{override}(m, Q_1, <\overline{Y}\triangleleft\overline{P}>\overline{T}\to T@\Xi_m) \ldots \mathit{override}(m, Q_{\text{n}}, <\overline{Y}\triangleleft\overline{P}>\overline{T}\to T@\Xi_m)}{[\texttt{eff}]<\overline{Y}\triangleleft\overline{P}>T\ m(\overline{T}\,\overline{x})@\Xi_m \text{ OK IN } C<\overline{X}\triangleleft\overline{N}>} \quad \text{T\_HEADER\_CLASS}$$

$$\frac{\texttt{interface}\ C<\overline{X}\triangleleft\overline{N}>\triangleleft Q_1 \ldots Q_{\text{n}}\{\ldots\} \qquad \overline{X}<:\overline{N}, \overline{Y}<:\overline{P} \vdash \overline{T}, T, \overline{P}, \Xi_m \text{ ok} \qquad \mathit{override}(m, Q_1, <\overline{Y}\triangleleft\overline{P}>\overline{T}\to T@\Xi_m) \ldots \mathit{override}(m, Q_{\text{n}}, <\overline{Y}\triangleleft\overline{P}>\overline{T}\to T@\Xi_m)}{[\texttt{eff}]<\overline{Y}\triangleleft\overline{P}>T\ m(\overline{T}\,\overline{x})@\Xi_m \text{ OK IN } C<\overline{X}\triangleleft\overline{N}>} \quad \text{T\_HEADER\_IFACE}$$

Class and interface typing:

$$\frac{\overline{X}<:\overline{N} \vdash \overline{N}, \overline{T}, \overline{P} \text{ ok} \qquad \overline{M} \text{ OK IN } C<\overline{X}\triangleleft\overline{N}>}{\texttt{class}\ C<\overline{X}\triangleleft\overline{N}>(\overline{T}\,\overline{f})\triangleleft\overline{P}\{\overline{M}\}} \quad \text{T\_CLASS}$$

$$\frac{\overline{X}<:\overline{N} \vdash \overline{N}, \overline{P}, \text{ ok} \qquad \overline{H} \text{ OK IN } C<\overline{X}\triangleleft\overline{N}>}{\texttt{interface}\ C<\overline{X}\triangleleft\overline{N}>\triangleleft\overline{P}\{\overline{H}\}} \quad \text{T\_IFACE}$$

Figure 5.11: Method, class and interface typing rules

$$\frac{}{\Xi;\Delta;\Gamma \vdash x : \Gamma(x)} \quad \text{T\_VAR}$$

$$\frac{\Xi;\Delta;\Gamma \vdash e_0 : T_0 \qquad \mathit{fields}(\mathit{bound}_\Delta(T_0)) = \overline{T}\,\overline{f}}{\Xi;\Delta;\Gamma \vdash e_0.f_i : T_i} \quad \text{T\_FIELD}$$

$$\frac{\Delta \vdash N\,\mathrm{ok} \qquad \mathit{fields}(N) = \overline{T}\,\overline{f} \qquad \Xi;\Delta;\Gamma \vdash \overline{e} : \overline{S} \qquad \Delta \vdash \overline{S} <: \overline{T}}{\Xi;\Delta;\Gamma \vdash \mathbf{new}\ N(\overline{e}) : N} \quad \text{T\_NEW}$$

$$\frac{\begin{array}{c}\Xi;\Delta;\Gamma \vdash e_0 : T_0 \qquad \mathit{mtype}(m, \mathit{bound}_\Delta(T_0)) = <\overline{Y} \triangleleft \overline{P}> \overline{U} \to U\,@\,\Xi_m \qquad \Delta \vdash \Xi \preceq \Xi_m \\ \Delta \vdash \overline{V}\,\mathrm{ok} \qquad \Delta \vdash \overline{V} <: [\overline{V}/\overline{Y}]\overline{P} \qquad \Xi;\Delta;\Gamma \vdash \overline{e} : \overline{S} \qquad \Delta \vdash \overline{S} <: [\overline{V}/\overline{Y}]\overline{U}\end{array}}{\Xi;\Delta;\Gamma \vdash e_0.m<\overline{V}>(\overline{e}) : [\overline{V}/\overline{Y}]U} \quad \text{T\_INVK}$$

$$\frac{\begin{array}{c}\mathit{mtype}(m, N) = \mathbf{eff}\ <\overline{Y} \triangleleft \overline{P}> \overline{U} \to U\,@\,\Xi_m \qquad \Delta \vdash \Xi \preceq \Xi_m \qquad \Delta \vdash \Xi \preceq N \\ \Delta \vdash \overline{V}\,\mathrm{ok} \qquad \Delta \vdash \overline{V} <: [\overline{V}/\overline{Y}]\overline{P} \qquad \Xi;\Delta;\Gamma \vdash \overline{e} : \overline{S} \qquad \Delta \vdash \overline{S} <: [\overline{V}/\overline{Y}]\overline{U}\end{array}}{\Xi;\Delta;\Gamma \vdash N::m<\overline{V}>(\overline{e}) : [\overline{V}/\overline{Y}]U} \quad \text{T\_EFF\_INVK}$$

$$\frac{\begin{array}{c}\Xi;\Delta;\Gamma \vdash e_0 : T_0 \qquad \mathit{mtype}(\mathtt{return}, \mathit{bound}_\Delta(T_0)) = U_{\mathrm{in}} \to U_{\mathrm{out}} \qquad \mathit{bound}_\Delta(T_0), \Xi;\Delta;\Gamma \vdash e_1 : T_1 \\ \Delta \vdash \mathit{bound}_\Delta(T_0) <: \mathtt{Handler}<U_{\mathrm{out}}, U_{\mathrm{in}}> \qquad \Delta \vdash T_1 <: U_{\mathrm{in}}\end{array}}{\Xi;\Delta;\Gamma \vdash \mathbf{with}\,(e_0)\,\{\,e_1\,\} : U_{\mathrm{out}}} \quad \text{T\_WITH}$$

Figure 5.12: Expression Typing

5.4.3 Expression Typing

The rules for expression typing are shown in Figure 5.12. Here we highlight the most salient differences with respect to those used in Featherweight Generic Java. First of all, there are two different rules for checking method invocation: T_INVK for regular method invocation and T_EFF_INVK for effect method invocation. These rules enforce that effect methods can only be called using normal method invocation syntax, and regular methods only via effect call syntax by requiring that the method type returned by the *mbody* metafunction has the right effect annotation. Also, next to the type variable environment and the type environment, expressions are typed in the context of an effect privilege set Ξ, as introduced by the method typing rules shown in Figure 5.11.

For instance, the rule for ordinary method invocation T_INVK checks that the declared effects of m (Ξ_m) are included in the privilege set Ξ using the condition

$\Delta \vdash \Xi \preceq \Xi_m$. The rule for effect invocation T_EFF_INVK performs the same check, but additionally enforces that the requested effect C is also in the privilege set Ξ. Dually, the rule for **with**-expressions (T_WITH) *extends* the privilege set for e_1 with the type of e_0 (T_0).

The relation $\preceq$ defines a preorder between two sets of types, and intuitively extends subtyping over sets of types. It is defined by first defining $\preceq$:

$$\Delta \vdash \Xi \preceq T \;\equiv\; \exists T' \in \Xi : \Delta \vdash T' <: T \qquad \text{(A type in } \Xi \textit{ handles} \text{ type } T\text{)} \tag{5.1}$$

The full relation is then obtained as follows:

$$\Delta \vdash \Xi_1 \preceq \Xi_2 \;\equiv\; \forall T \in \Xi_2 : \Delta \vdash \Xi_1 \preceq T \qquad (\Xi_1 \textit{ handles} \text{ all types in } \Xi_2\ ) \tag{5.2}$$

This relation is used in determining whether a privilege set is powerful enough to handle all effects requested by a certain expression. The relation is further used in checking the validity of method implementations, where the effect annotations of method definitions in a class may be less demanding according to $\preceq$ than the declared annotations in a declaring interface. In plain language this means that the effect payload of a method, i.e. the effects it might invoke, may be less than what is declared. Note in particular that pure method implementations (i.e. without any effects) conform to well-typed interface method declarations with arbitrary sets of effect annotations, because $\Xi \preceq \bullet$.

5.4.4 Soundness

Soundness is often stated as "well-typed programs cannot go wrong". We sketch the proof of soundness using progress and preservation.

For progress, we make a distinction by considering that expressions in normal form are not only values but also expressions $X_N[N{::}m{<}V{>}(\overline{e})]$ whose next evaluation step requires the handling of an operation call. By pairing the latter with a constraint to a set of effect privileges Ξ, we have a new class of expressions that we do not consider stuck.

Definition 3 (Normal Form) *An expression e is in normal form with respect to Ξ if either (a) is a value v, or (b) is an expression of the form $X_N[N{::}m{<}V{>}(\overline{e})]$ such that $\bullet \vdash \Xi \preceq N$.*

Definition 4 (Non-stuckness) *An expression e is non-stuck with respect to Ξ if either (a) is in normal form with respect to Ξ, or (b) there is an e', such that $e \longrightarrow e'$.*

Lemma 5.4.1 (Progress) *If $\Xi; \bullet; \bullet \vdash e : T$, then e is non-stuck with respect to Ξ.*

Proof sketch. By induction on the structure of type derivations, with a case analysis on the last rule used. The interesting case is T_WITH, in particular when e is **with** (**new** Q($\overline{v}$)){e_1}. By rule T_WITH, we have $Q, \Xi; \bullet; \bullet \vdash e_1 : T_1$ for some T_1. Then, by hypothesis, e_1 is non-stuck, thus, it is either (a) a redex, in which case e progresses; (b) a value, in which case rule R_RETURN applies; or (c) of the form $X_N[N::m<V>(\overline{e})]$, such that $\bullet \vdash Q, \Xi \preceq N$, in which case, by the definition of $\preceq$ (cf. section 5.4.3), either: (c.1) $Q <: N$, which implies that rule R_EFF_INVK applies; or (c.2) $Q \not<: N$, which implies that $\Xi \preceq N$; and because of this together with the fact that no reduction rules apply, e fits in the definition of a context X_N, being of the form $X_N[N::m<V>(\overline{e})]$, and thus, is in normal form with respect to Ξ (therefore non-stuck with respect to Ξ).

Lemma 5.4.2 (Preservation) *If* $\Xi; \Delta; \Gamma \vdash e : T$ *and* $e \rightarrow e'$, *then* $\Xi; \Delta; \Gamma \vdash e' : T'$ *for some* T' *such that* $\Delta \vdash T' <: T$.

Proof sketch. By induction over the reduction rules. A number of necessary lemmas are needed, such as that (1) term and (2) type substitution preserve typing, that (3) subtyping preserves method typing, and that (4) method bodies conform to declared return types. Their proofs are similar to those found in [IPW01]. The replacement lemma (5) states that if there is a deduction D ending in $\Gamma \vdash C[e] : T$, where C is a context; and there is a sub-deduction D' ending in $\Gamma' \vdash e : T'$, and $\Gamma \vdash e' : T'$, then $\Gamma \vdash C[e'] : T$ (proof is similar to replacement in [WF94]).

The interesting cases are rules R_EFF_INVK, R_RETURN and R_RESUME. The crucial facts for R_EFF_INVK are:

(a) We know by T_WITH that $X_N[N::m<V>(\overline{v})]$ has type T_1 for some T_1. We also know trivially that $N::m<V>(\overline{v})$ has type S for some S. By letting variable x_v have type S and plugging x_v inside context $X_N[]$ in place of the effect method call, we obtain expression $X_N[x_v]$, that by (5) retains type T_1. By letting x_h have type T_0 for some $T_0 <: Handler<U_{out}, U_{in}>$, we know that expression **with**(x_h){$X_N[x_v]$} in the body of the built resumption object has type U_{out} and then resumption object v_k conforms by construction to interface Resume<S, U_{out}, U_{in}>, as required by the specification of resumption objects.

(b) By premise $\bullet \vdash Q <: N$ and (3) together with (4), we know that the expression implementing an effect method call $N::m<V>(\overline{v})$ is to be found in the corresponding method m in handler class Q.

Rule R_RESUME relies on the fact that resumption objects are not expressible and are only introduced via substitution of **there** in R_EFF_INVK. From (a), we know that the resumption object is well-typed to Resume<S, U_{out}, U_{in}> for some S, U_{in} and U_{out}, and then the argument continues as the one for standard method invocation.

In case of R_RETURN, since $N <:$ Handler<U_{out}, U_{in}> for some U_{out} and U_{in}, and Handler defines return to have the type $U_{in} \rightarrow U_{out}$, both the **with** expression and the result of invoking return have the same type.

5.5 Discussion & Related Work

5.5.1 Objects for Effect

According to Cook the essence of objects is encapsulation and dynamic dispatch [Coo12]. These are precisely the aspects that we have leveraged in JEff for supporting effectful programming. This can be seen from the fact that, apart from the **`with`**-construct, all other parts of effect handlers are realized by calling methods, all of them defined in interfaces or classes.

Effect operations are defined as methods with ordinary type signatures, but with bodies typed according to the `Handler` interface. Effect resumption is method invocation on the special **`there`** object, which is typed by the ordinary `Resume` interface. The **`with`**-construct brings `Handler` objects into dynamic scope, and when its syntactic body has evaluated to a value, the result is passed through the ordinary `return` method as required by `Handler`. Both the `Resume` and `Handler` interfaces are not special, but simply part of JEff's standard library. Like Java's `Closeable` and `Iterable`, they merely provide the interface between certain language features (in JEff's case **`with`** and **`there`**) and the objects defined by the programmer.

5.5.2 Effect Polymorphism

Effect polymorphism refers to the ability of code to operate on objects with varying effect surfaces, where the actual effectfulness of code derives from dependencies such as method parameters or fields. In JEff, all method declarations – both in interfaces and classes – need to be annotated with concrete effect types for unhandled effects. As a result, JEff does not support effect polymorphism.

Consider the two interfaces below, `Function` and `List`:

```
interface Function<T,U> {
  U apply(T t) // pure
}
```

```
interface List<T> {
   <U> List<U> map(Function<T,U> f) // pure
}
```

Both the `apply` method in `Function`, and `map` in `List` have no effect annotations. As a result, implementations of these methods are required to be pure. Another consequence is that the argument to `map` must be pure as well, and hence `map` is effect monomorphic: it only applies pure functions to the list. Since it is not possible to abstract over the effect signature of a (set of) method(s), different `map`s are needed for `Function`s with different effect payloads.

A similar effect can be observed in the modular interpreters presented in Section 5.2.4. Although new `Exp`s can be defined in a modular fashion, the allowed effects of the `eval` method are determined and fixed in the `Exp` interface. With a mechanism for effect polymorphism, this could potentially be made more flexible, where the `eval`

in `Exp` would be polymorphic, and each implementation would carry its own effect signature.

Supporting effect polymorphic methods in a language like JEff is challenging. Existing languages like Frank [LMM17] and Koka [Lei17] support effect polymorphic functions but have significantly different type systems than JEff. For instance, Koka's row polymorphism allows the effect payload of a function to be left partially open; unification is then used to add rows to the types during type inference. It is however unclear how to port this kind of inference to object-oriented languages.

A possible middle-ground solution is presented by Toro and Tanter [TT15] in the form of a gradual polymorphic effect system for Scala. By giving up some static guarantees about the effectfulness of code, the use of higher-order functions like `map` can be made more flexible. In this case both `apply` and `map` would be annotated with the special `@unknown` annotation (denoted as ¿ in [BGT14]), which signifies that there is no static information about the effects of `map`. Implementations of such methods can provide the missing information with concrete annotations. The type system can then derive more specific effect payloads at concrete call sites; if it cannot, then dynamic checks ensure that the effect will be handled correctly.

5.5.3 Propagation of Annotations

JEff's effect propagation mechanism is similar to Java checked exceptions and, as such, suffers from the same problems. Programmers need to annotate each method with its allowed effects which is verbose and makes the code less flexible. An interface method declares a number of effects but the corresponding method in a class implementing this interface might require a new effect. In that case the annotations of the parent interface should be modified to incorporate the new effect, and consequently, all classes implementing it.

The lack of flexibility of checked annotations has been addressed in the work on anchored exceptions, where method call dependencies are taken into account when propagating exceptions in `throws`-clauses [vDS05]. Based on this work and in the context of Scala, Ritz proposes Lightweight Polymorphic Effects (LPE) [ROH12] as an attempt to generalize the annotation-based style of checked exceptions to a wider range of effects. LPE allows programmers to annotate effectful Scala code with effect annotations, where effects are defined using a customizable effect lattice, inspired by [MM09]. We consider incorporating a similar mechanism to JEff as future work.

5.5.4 Related Work

Algebraic effects [PP09; PP13] are a mechanism to represent effects in functional languages. An effect defines the signature of a set of operations. The actual semantics of an effect is provided by handlers, dynamically scoped constructs that implement

the behavior for each operation. In JEff, the effect interface is analogous to the effect signatures, while handler classes correspond to their implementation. The handler abstraction is further discussed in [KLO13], where a formal definition together with library-based implementations in several languages is presented. This work also introduces the distinction between deep and shallow handlers. Deep handlers automatically wrap the continuation within the current handler. Shallow handlers assume no handling by default, and therefore require the programmer to install a new handler manually if so desired. JEff is closer in spirit to the latter, since we require a handler object as the second argument to `there.resume`.

Besides the handler libraries presented in [KLO13], there are several other library-based encoding of effects and handlers [BS17; KI15; KSS13; WSH14]. Compared to library-based encodings, the built-in effects in JEff have the following advantages:

- Reducing the boilerplate caused by the accidental complexity of the effects embedding.
- Having a clearer computation model of the interaction between OO concepts and effects. For instance, the Effekt library [BS17] encodes algebraic effects using sophisticated Scala features, such as implicit function types. In JEff, the interactions are clear and rely on a limited number of concepts captured by the FJEff calculus.
- Opening the door to domain-specific compiler optimizations taking into account the native representation of effects. For example, in [Lei17], Leijen shows an efficient compilation of his row-based effects using a type-directed selective CPS translation.

Besides the library-based approaches, in section 5.5.2 we have discussed a number of functional languages that provide native support for handlers and effects, using however different mechanisms for effect propagation. Koka [Lei17], for example, features an effect inference system that requires minimal annotations from users by relying on the polymorphic row types discussed in the previous section. Frank [LMM17], on the other hand, treats function application as a special case of a more generic mechanism of operators that act as interpreters of effects. Rather than accumulating effects outwards via type inferencing as in Koka, effects are propagated inwards using an ambient ability, similar to the privilege sets Ξ used in JEff.

5.6 Conclusion

Effect handlers are a technique to define, scope and modularize side-effects in programming languages that do not support them natively. While originally introduced and explored in the context of (purely) functional programming languages, it is an open question how effect handling could be supported first-class in an object-oriented programming language. In this chapter we presented first steps towards

answering this question, in the form of JEff, a purely object-oriented language with built-in support for effectful programming. Effects are defined using effect methods which obey special typing rules and have access to the current continuation for resuming computation. Handler objects are brought into dynamic scope using the `with`-construct; effects that are not handled need to be declared at the method level, similar to Java's `throws`-clause.

We showed how common effects, like exception handling and state, can be defined within JEff, and how effects can be used to structure extensible interpreters. Also, using a pretty-printer as illustrative device, we showed how the type qualifier of effect invocations enable ad hoc overloading of effects. The semantics of JEff are formalized based on a core subset FJEff, including a type system that ensures that all effects are properly handled or propagated. Finally, we provided intuitions that show that the type system is sound with respect to the semantics.

As directions for further work, we consider mechanizing the soundness proof, implementing the language, and exploring extending JEff with support for (implementation) inheritance. In particular, support for `super`-calls would allow programmers to customize effect handlers. The biggest open question, however, is how to reconcile the open-world assumptions of object-orientation with effect handling. Modular extensibility of both data types and operations has been solved by solutions to the expression problem [Wad98]; the next question is how to realize the same at the level of effects. JEff represents the first steps towards better understanding this question from the perspective of object-oriented programming.

6 Conclusion

The implementation of programming languages is complex, as it involves different artifacts that need to interact with each other in order to process programs written in these languages. Since so much effort is put into developing new language artifacts, the language engineering practice can benefit from the availability of reusable language components that can be assembled in order to build new languages. This thesis has introduced new object-oriented techniques to modularly implement programming languages as Language Libraries.

In the remainder of this chapter, we summarize the conclusions of previous chapters, by describing how they answer the main questions of this dissertation. Thereafter we discuss in depth the benefits and limitations of the introduced object-oriented techniques together with open challenges and opportunities for future work.

6.1 Recapitulation of Research Questions

Research Question 1 (RQ1): How can object algebras facilitate modular language extension of a General-purpose Programming Language?

RQ1 is answered in Chapter 2 by means of Recaf, a tool that provides lightweight host language extension. Recaf extends Java by relying on a generic transformation that processes programs in a customized Java extended with new arbitrary keywords, and outputs a standard Java program that delegates the newly introduce keywords to

invocations on object algebras, thus admitting arbitrary definitions of the semantics of such keywords.

The syntax of the customizable Java was generalized in order to admit new keywords that conform to certain predefined syntactic patterns, e.g., foreach-like keywords. Moreover, since the semantics of the keywords is provided by object algebras, the definition of the extensions inherits object algebras' modularity benefits and can be deployed as libraries of modular language components.

As illustrative examples, we have extended Java with asynchronous control flow, Parsing Expression Grammar definitions, among other extensions, each of which can be modularly deployed as a library. The libraries consist of object algebra classes that encode the respective semantics.

The combined strategy of a generic transformation plus modular semantic components encoded as object algebras enables the lightweight definition of modular extensions to a general-purpose programming language, without altering the compiler and just requiring a generic pre-processing step, written once and for all.

Research Question 2 (RQ2): How to trace program transformations in a lightweight, language-agnostic manner?

RQ2 is answered in Chapter 3. String origins is a technique that allows textual transformations to preserve the link between each string in the output and its origin. The key idea is to instrument each string-related operation used in the transformations (e.g., appending) in order to preserve the origins of the strings being processed. Since the technique abstracts over operations at the string level, it is language-agnostic and lightweight.

String origins represent a step towards extension-aware tool support, facilitating the development of extension-based language libraries.

Research Question 3 (RQ3): How to compose modular interpreters with different context-requirements using object algebras?

RQ3 is answered in Chapter 4. We have defined interpreters for embedded languages using object algebras whose carrier types are functions from semantic context requirements to values. These interpreters are composable if their context requirements coincide. However, if two interpreters have different context requirements, they cannot be composed since the signatures of their carriers differ.

In this setting, we have defined a lifting that allows new arbitrary context parameters to be implicitly propagated in automatically derived interpreters. This automatic derivation is realized via a syntactic transformation (using macros) or runtime reflection (using Java proxies). The interpreters with an extended signature

can be composed with existing interpreters that have the same signature, providing an answer to RQ3.

Implicit context propagation of object algebra-based interpreters thus enable better composability without anticipation.

Research Question 4 (RQ4): How to integrate effect handling in an object-oriented language?

RQ4 is answered in Chapter 5. We have introduced JEff, an object-oriented language that features built-in effect definitions, enabling a more reliable and flexible handling of effects: reliable by declaring the effects that a method is allowed to execute using a mechanism similar to checked exceptions, and flexible by using object orientation, and thus, modular extensibility. The key point of the integration was the use of well known object-oriented structures for defining effect-related concepts: interfaces define effect signatures, while classes implementing such interfaces define handlers.

The combination of objects and effects creates opportunities for new patterns and idioms to encode user-defined effects, like, for instance, effectful interpreters, in the spirit of the interpreters of Chapter 4.

6.2 Discussion

The central topic of this dissertation is investigating object-oriented abstractions, in particular object algebras, as a mechanism to implement libraries of modular language components.

The literature on object algebras [GPS14; OC12; OvdSL+13] shows that object algebras are a good technique to implement extensible languages. Since language extensibility is fundamental for library-based language development, object algebras are a natural choice to structure language components. However, the referred works have considered only some scenarios of language development using object algebras. In this thesis, our goal was to explore more challenging and realistic language implementations and therefore we have added new requirements to the extensibility in two-dimensions that object algebras support. These new requirements are:

- syntactic extensibility of general-purpose programming languages (Recaf)
- modular implementation of language interpreters in presence of different context requirements (Modular interpreters with Implicit Context Propagation)

In this thesis, object algebras have proven to be a versatile technique for addressing these requirements. By having investigated how to address these requirements using object algebras, the limits of what this mechanism provides in terms of linguistic abstraction have been explored. Now, we discuss what we have learned from this

experience by presenting the benefits and limitations of this particular technological choice for implementing language libraries.

6.2.1 Key Benefit: Unanticipated Extension and Composition

The fundamental technique of object-oriented programming is dynamic dispatch, which allows programmers to write code assuming behavioral interfaces but not the implementation of such interfaces [Coo12]. A message sent to an object gets only dispatched at runtime. This characteristic of object-oriented programming enables modular extensibility [Ald13], as client code makes no assumption of the actual implementation of the objects typed by the interfaces they call to. In other words, at the moment of writing an abstraction, there is no need for anticipating knowledge about the extensions (namely, interface implementations) of the types that the abstraction interacts with.

Object algebras take the intrinsic extensibility of object orientation one level further, by exploiting inheritance and parametric polymorphism from the host object-oriented language. An object algebra can then support the addition of unanticipated operations (using a combination of subtype and parametric polymorphism) and the addition of unanticipated data variants (using inheritance).

The improved extensibility provided by the aforementioned characteristics of object algebras have been fundamental for designing Recaf and the Modular interpreters with Implicit Context Propagation, as different instantiations of the idea of language libraries.

In Recaf, the flexibility of unanticipated extension and composition brings the following benefits for creating Java dialects as libraries:[1]

- **Multiple implementations of the base Java semantics (unanticipated host-language interpreter implementation).** We have, for instance, showed a continuation-passing style Java interpreter, and a direct-style one, both using reflection. In order to improve on reflection's performance overhead, one could think of alternative, more optimal implementations at the bytecode level. The object algebra interface representing the Java base syntax admits an open-ended number of interpreters.
- **Extension of Java with new keywords (unanticipated syntactic extensions).** Syntactic extensions for asynchrony, parsing, or other domain-specific concerns can be implemented by extending the object algebras (using inheritance). Naturally, multiple implementations for the newly introduced keywords are admitted.

[1] The use of Java's default methods in Recaf adds more trait-like flexibility to the object algebra pattern. This is not further discussed as it is not fundamental to the benefits being discussed.

These benefits allow Recaf to support the development of independent language extensions, forming libraries of Java dialects. A Recaf library can contain a different implementation of the Java interpreter (more sophisticated than the reflection-based one) by providing an alternative implementation of the base Java object algebra interface, or some new syntactic extensions by providing extensions to a particular Java interpreter, extensions that know how to interpret the newly-introduced keywords. One can even think of overriding the meaning of an existing Java construct by using simple method overriding on an extended Java interpreter.

In the case of Modular Interpreters with Implicit Context Propagation, where language components correspond to interpreters, the flexibility of unanticipated extension and composition brought by object algebras[2] was key to the lifting-based solution. Interpreters were represented as object algebras with carrier types corresponding to a function type from context parameters to values. These interpreters were naturally composed by inheritance, as long as their carriers coincide. In this case, one could write a language interpreter for a language fragment A unaware of a future extension B that would add more language features. However, since the carrier types needed to coincide, the pattern did not capture another kind of anticipation: anticipation of context.

Nevertheless, object algebras provide the basic foundation to our lifting-based solution. Since the language components were already supporting unanticipated composition, their liftings are also independent and can be generated at composition time. The liftings are per-component, modular, and admit separate compilation. Thus, all the benefits brought by object algebras were retained. In this manner, object algebras support the development of independent modular language interpreters, without requiring context anticipation.

The Recaf and the Implicit Context Propagation scenarios show that object algebras are flexible enough to support new scenarios of language composition, besides the ones illustrated by the existing literature on object algebras. Moreover, the pattern brings some benefits as compared to similar techniques in non-OO settings.

The lack of anticipation that is fundamental to object orientation fosters modularity and reuse, and enjoys separate compilation. Other techniques to compose language interpreters lack this characteristic. Monad Transformers [LHJ95], for instance, rely on composition via liftings that require global recompilation if new components are added, hampering modular reuse.

By contrast, since both the Java dialects developed with Recaf and the Modular Interpreters with Implicit Context Propagation enjoy modular extensibility, they can be compiled and distributed in binary form, enabling black-box reuse. This essential

[2]Implicit Context Propagation has been presented in Scala, using traits. The use of traits adds more flexibility to the object algebra pattern. This is not further discussed as it is not fundamental to the benefits being discussed.

characteristic of object orientation, and in particular of object algebras, is fundamental to our vision of library-based language development.

6.2.2 Limitations: Host-level Dependency

Object algebras are an embedding technique. As such, the abstractions that they enable are encoded in a host language. This can impose some restrictions to the expressiveness of object algebra-based abstractions, and in particular, in the scenario of library-based language implementation.

In Recaf, for instance, we faced some restrictions when expressing virtualization of expressions. Java is a statement- and expression-based language. Hence, full virtualization of Java programs using Recaf implies that new keywords can be introduced both at the expression and the statement level. Unfortunately, at the expression level the code that the Recaf transformation can generate does not provide static guarantees. Recall that the expression algebra was along the lines of the following definition:

```
interface MuExpJava<E> {
    E IntLit(Integer x);
    E StrLit(String x);
    E Mul(E e1, E e2);
    ...
}
```

A possible carrier for an expression algebra, would be the closure type `IEval`:

```
interface IEval { Object eval(); }
```

The problem of this encoding is that a type-incorrect expression such as `alg.Mul(alg.StrLit("a"), alg.IntLit(1))` cannot be detected statically, since the Java type system does not allow us to represent these type constraints properly. For example, if `alg`'s carrier type is `IEval`, the result of the aforementioned expression (and its sub-expressions) will be always the thunk `IEval`, that will eventually produce `Object`. As there is no way to relate the argument types of the variants to the carrier type, the denotations of expressions produced by the algebra are untyped (or better said, dynamically typed). As an illustration, the `IEval` only ensures that `Object`s are produced, hence, it gives no static guarantees.

In an ideal setting, we would like to express that `IEval` is parametric on the object being returned, such as:

```
interface IEval2<T> { T eval(); }
```

To use such carrier, the definition of the expression algebra would need to be written in a language supporting type constructor polymorphism, such as Scala [MPO08].

The following snippet is written in a fictional Java-like language supporting this feature:

```
interface MuExpJava2<C<T>> {
    C<Integer> IntLit(Integer x);
    C<String> StrLit(String x);
    C<Integer> Mul(C<Integer> e1, C<Integer> e2);
    ...
}
```

If an expression using this algebra is type incorrect, such as `alg.Mul(alg.StrLit("a"), alg.IntLit(1))`, the type error would be detected statically.

As said before, the listing above (`MuExpJava2`) would only work if Java supported type constructor polymorphism. This shows that in Recaf, the expressiveness of the host language can limit the static guarantees provided by the dialects created with Recaf.

In Modular Interpreters with Implicit Context Propagation, the different kinds of context were represented using host-level side-effects. This is problematic as it hides the true nature of the feature interactions between different contexts, for instance, if one wanted to integrate state with backtracking.

This limitation precludes true independent development of extensions as the context interaction could be problematic in unforeseen ways. The initial motivation behind JEff is to design an object-oriented language that supports our object algebra style of library-based language development, but making the effects (and thus their interaction) explicit.

Object algebras' limitations in terms of their dependency on the host language lead to the conclusion that even though object algebras impose very minimal requirements on the host language, namely inheritance and parametric polymorphism, as long as one wants to express more sophisticated linguistic features, e.g. by using built-in effects or type-constructor polymorphism, the host language needs to be accordingly complex.

In the case of Recaf, possible directions for overcoming these limitations include a Java encoding of type constructor polymorphism [BPF+15] or using another host language that features this kind of polymorphism, such as Scala. Changing the host language to one that supports type constructor polymorphism is only the first step in the direction of providing static guarantees to Recaf. Making full virtualization safer under this schema is a clear opportunity for future work.

In the case of Implicit Context Propagation, in order to circumvent the feature interaction problem a possible direction is to use a host language with explicit effect declarations in order to encode the modular interpreters. Even though JEff is a natural candidate, this is not yet achievable in its current form. Hence, this limitation

represents our main opportunity for future work in the direction of advancing our vision of OO library-based language development.

6.2.3 Future work: Towards Effectful Modular Interpreters in JEff

The initial motivation behind JEff is to overcome the limitations observed in the Modular Interpreters with Implicit Context Propagation, in terms of feature interaction. The semantics of the composition of the interpreters with different context parameters is sometimes involved and unpredictable, since these parameters are implemented using the host-level representation of effects. By featuring built-in effect definitions, JEff makes these interactions explicit in an object-oriented setting.

However, it is not yet possible to write modular effectful interpreters in JEff. In Chapter 5 we have shown an example of effectful interpreters for a simple expression-based language, in the style of the interpreter pattern. Interface `Exp` represents some language's expressions:

```
interface Exp {  Value eval@Store,Env }
```

Even though, object algebra-based interpreters are modular along the syntactic and semantic dimensions, there is a third dimension at play here, which corresponds to the effect privileges observed in the annotations of the `eval` method. As illustrated by the `Exp` interface, in JEff the effects need to be determined ahead of time at the method declaration level, and therefore language engineers need to anticipate the required effects at the moment of defining the interface of the language. Moreover, listing all the potential effects is verbose.

This situation is not scalable and unveils opportunities for future work in order to improve the declaration of effects for enabling true modularity:

- Effect polymorphism: Method `eval` in interface `Exp` should actually be effect-polymorphic, but this cannot represented in JEff. Investigating effect polymorphism in JEff is a clear line for future work. Relevant related work has been presented in the context of integrating effects with gradual typing [BGT14; TT15], or in the context of lightweight polymorphic effects in Scala [ROH12].
- Removing explicit annotations: In order to make the notation for method-level effect declaration more lightweight, the effect annotations could be derived by performing effect inference on the method bodies, in the spirit of Koka [Lei17].

These improvements, in particular effect polymorphism, together with the addition of other relevant OO features, such as inheritance, could make JEff an ideal host language for structuring effectful modular interpreters, providing true modularity along the syntactic, the operational, and the effectful axes. In our vision of library-based language development, this will imply that language components could be written without anticipating along any of these three dimensions.

6.3 Conclusion

This dissertation has demonstrated how the object algebra pattern supports the definition of modular semantic components in the context of object-oriented programming languages. This has been done in two particular settings: language extensions (with the aid of a generic syntactic transformation) and modular interpreters (addressing the challenge of how to compose interpreters with different context requirements). Our modular interpreters represent context using the host language's side effects, which hides the true nature of the context interaction. Inspired by this challenge, this thesis also introduces JEff, a new object-oriented language with built-in support for effect definition. JEff enables new OO idioms to structure effectful code, which can bring us closer to modular effectful interpreters.

In sum, this thesis has investigated the definition of language components using object-oriented techniques such as object algebras, concluding that the object-oriented paradigm, and object algebras in particular, provides a fertile foundation for the vision of library-based language development.

Bibliography

[ADH+98] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr, D. H. Bartley, R. Halstead, et al. "Revised^5 report on the algorithmic language Scheme". In: *Higher-order and symbolic computation* 11.1 (1998), pp. 7–105 (cit. on p. 74).

[ANR+06] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. "Model Traceability". In: *IBM Syst. J.* 45.3 (July 2006), pp. 515–526 (cit. on pp. 42, 57).

[Ald13] J. Aldrich. "The power of interoperability: Why objects are inevitable". In: *Proceedings of the 2013 ACM international symposium on new ideas, new paradigms, and reflections on programming & software (Onward! 2013)*. ACM. 2013, pp. 101–116 (cit. on pp. 2, 112, 140).

[All89] L. Allison. "Direct semantics and exceptions define jumps and coroutines". In: *Information Processing Letters* 31.6 (1989), pp. 327–330 (cit. on p. 107).

[ALZ01] D. Ancona, G. Lagorio, and E. Zucca. "A Core Calculus for Java Exceptions". In: *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '01. Tampa Bay, FL, USA: ACM, 2001, pp. 16–30 (cit. on p. 114).

[BGT14] F. Bañados Schwerter, R. Garcia, and É. Tanter. "A Theory of Gradual Effect Systems". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: ACM, 2014, pp. 283–295 (cit. on pp. 133, 144).

[BP15] A. Bauer and M. Pretnar. "Programming with algebraic effects and handlers". In: *Journal of Logical and Algebraic Methods in Programming* 84.1 (2015). Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011, pp. 108–123 (cit. on p. 112).

[BHK90] J. A. Bergstra, J. Heering, and P. Klint. "Module Algebra". In: *J. ACM* 37.2 (Apr. 1990), pp. 335–372 (cit. on p. 3).

[BIvdS16] A. Biboudis, P. Inostroza, and T. van der Storm. "Recaf: Java Dialects As Libraries". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2016. Amsterdam, Netherlands: ACM, 2016, pp. 2–13 (cit. on pp. 14, 17).

[BPF+15] A. Biboudis, N. Palladinos, G. Fourtounis, and Y. Smaragdakis. "Streams a la carte: Extensible Pipelines with Object Algebras". In: *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Ed. by J. T. Boyland. Vol. 37. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 591–613 (cit. on pp. 21, 143).

[BRM+12] G. Bierman, C. Russo, G. Mainland, E. Meijer, and M. Torgersen. "Pause 'N' Play: Formalizing Asynchronous C#". In: *Proc. of the 26th European Conference on Object-Oriented Programming*. ECOOP'12. Beijing, China: Springer-Verlag, 2012, pp. 233–257 (cit. on p. 38).

[BS17] J. I. Brachthäuser and P. Schuster. "Effekt: Extensible Algebraic Effects in Scala (Short Paper)". In: *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*. SCALA 2017. Vancouver, BC, Canada: ACM, 2017, pp. 67–72 (cit. on p. 134).

[BLC02] E. Bruneton, R. Lenglet, and T. Coupaye. "ASM: a code manipulation tool to implement adaptable systems". In: *Adaptable and Extensible Component Systems* (2002) (cit. on p. 37).

[Bur13] E. Burmako. "Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming". In: *SCALA*. ACM, 2013, 3:1–3:10 (cit. on p. 79).

[CKS09] J. Carette, O. Kiselyov, and C.-c. Shan. "Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages". In: *J. Funct. Program.* 19.5 (Sept. 2009), pp. 509–543 (cit. on pp. 8, 21, 38).

[CF94] R. Cartwright and M. Felleisen. "Extensible denotational language specifications". In: *Theoretical Aspects of Computer Software*. Springer. 1994, pp. 244–272 (cit. on p. 105).

[Chi95] S. Chiba. "A Metaobject Protocol for C++". In: *Proc. of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '95. Austin, Texas, USA: ACM, 1995, pp. 285–299 (cit. on p. 38).

[Chi98] S. Chiba. "Javassist–a reflection-based programming wizard for Java". In: *Proc. of the OOPSLA'98 Workshop on Reflective Programming in C++ and Java*. Vol. 174. 1998 (cit. on p. 37).

[CW09] E. Chin and D. Wagner. "Efficient character-level taint tracking for Java". In: *Proceedings of the 2009 ACM workshop on Secure web services*. ACM. 2009, pp. 3–12 (cit. on p. 57).

[CMS+15] M. Churchill, P. D. Mosses, N. Sculthorpe, and P. Torrini. "Reusable Components of Semantic Specifications". In: *Transactions on Aspect-Oriented Software Development XII*. Ed. by S. Chiba, É. Tanter, E. Ernst, and R. Hirschfeld. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 132–179 (cit. on pp. 3, 105, 109).

[CMT14] M. Churchill, P. D. Mosses, and P. Torrini. "Reusable Components of Semantic Specifications". In: *Proceedings of the 13th International Conference on Modularity*. MODULARITY '14. Lugano, Switzerland: ACM, 2014, pp. 145–156 (cit. on pp. 3, 105, 109).

[CV08] R. Clarke and O. Vitzthum. *Coffee: Recent Developments*. John Wiley & Sons, 2008 (cit. on p. 18).

[Cle03] T. Cleenewerck. "Component-based DSL development". In: *GPCE*. Springer. 2003, pp. 245–264 (cit. on p. 105).

[Cle07] T. Cleenewerck. "Modularizing Language Constructs: A Reflective Approach". PhD thesis. Vrije Universteit Brussel, 2007 (cit. on p. 105).

[CR91] W. Clinger and J. Rees. "Macros That Work". In: *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '91. Orlando, Florida, USA: ACM, 1991, pp. 155–162 (cit. on p. 51).

[Coo09] W. R. Cook. "On Understanding Data Abstraction, Revisited". In: OOPSLA '09 (2009), pp. 557–572 (cit. on pp. 112, 118).

[Coo12] W. R. Cook. "A Proposal for Simplified, Modern Definitions of "Object" and "Object Oriented"". Online: `https://wcook.blogspot.nl/2012/07/proposal-for-simplified-modern.html`. Nov. 2012 (cit. on pp. 113, 132, 140).

[Cos03] P. Costanza. "Dynamically scoped functions as the essence of AOP". In: *ACM SIGPLAN Notices* 38.8 (2003), pp. 29–36 (cit. on p. 106).

[CF10] R. Culpepper and M. Felleisen. "Debugging hygienic macros". In: *Science of Computer Programming* 75.7 (2010). Generative Programming and Component Engineering (GPCE 2007), pp. 496–515 (cit. on p. 11).

[CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000 (cit. on p. 5).

[dSHL06] B. C. d. S. Oliveira, R. Hinze, and A. Löh. "Extensible and modular generics for the masses". In: *Revised Selected Papers from the Seventh Symposium on Trends in Functional Programming, TFP 2006, Nottingham, United Kingdom, 19-21 April 2006*. 2006, pp. 199–216 (cit. on p. 8).

[DMS+10] P. Dhoolia, S. Mani, V. Sinha, and S. Sinha. "Debugging Model-Transformation Failures Using Dynamic Tainting". In: *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. Ed. by T. D'Hondt. Vol. 6183. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 26–51 (cit. on p. 57).

[Dmi04] S. Dmitriev. "Language Oriented Programming: The Next Programming Paradigm". In: *JetBrains onBoard* (Nov. 2004) (cit. on p. 1).

[Dup95] L. Duponcheel. "Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters". 1995 (cit. on p. 104).

[EGR12] S. Erdweg, P. G. Giarrusso, and T. Rendel. "Language Composition Untangled". In: *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*. LDTA '12. Tallinn, Estonia: ACM, 2012, 7:1–7:8 (cit. on p. 3).

[ERK+11] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. "SugarJ: Library-based Syntactic Language Extensibility". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. Portland, Oregon, USA: ACM, 2011, pp. 391–406 (cit. on pp. 3, 38).

[EvdSD14] S. Erdweg, T. van der Storm, and Y. Dai. "Capture-Avoiding and Hygienic Program Transformations". In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by R. Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 489–514 (cit. on pp. 16, 52, 53).

[EvdSV+15] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. "Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future". In: *Computer Languages, Systems & Structures* 44, Part A (2015). Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014), pp. 24–47 (cit. on p. 30).

[Esp95] D. Espinosa. "Semantic Lego". PhD thesis. Columbia University, 1995 (cit. on p. 101).

[FFF+18] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt. "A Programmable Programming Language". In: *Communications of the ACM* 61.3 (Feb. 2018), pp. 62–71 (cit. on p. 1).

[FKF98] M. Flatt, S. Krishnamurthi, and M. Felleisen. "Classes and Mixins". In: *POPL*. ACM, 1998, pp. 171–183 (cit. on pp. 61, 93).

[For02] B. Ford. "Packrat Parsing:: Simple, Powerful, Lazy, Linear Time, Functional Pearl". In: *Proc. of the 7th ACM SIGPLAN International Conference on Functional Programming*. ICFP '02. Pittsburgh, PA, USA: ACM, 2002, pp. 36–47 (cit. on p. 36).

[For04] B. Ford. "Parsing Expression Grammars: A Recognition-based Syntactic Foundation". In: *Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '04. Venice, Italy: ACM, 2004, pp. 111–122 (cit. on p. 34).

[Fow10] M. Fowler. *Domain Specific Languages*. 1st. Addison-Wesley Professional, 2010 (cit. on pp. 1, 6, 47, 87).

[FM89] B. N. Freeman-Benson and J. Maloney. "The DeltaBlue algorithm: An incremental constraint hierarchy solver". In: *Computers and Communications, 1989. Conference Proceedings., Eighth Annual International Phoenix Conference on*. IEEE. 1989, pp. 538–542 (cit. on pp. 60, 61, 97).

[GHJ+94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994 (cit. on p. 97).

[GPS14] M. Gouseti, C. Peters, and T. v. d. Storm. "Extensible Language Implementation with Object Algebras (Short Paper)". In: *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*. GPCE 2014. Västerås, Sweden: ACM, 2014, pp. 25–28 (cit. on pp. 10, 12, 86, 106, 108, 139).

[GSK10] A. Guha, C. Saftoiu, and S. Krishnamurthi. "The Essence of Javascript". In: *Proceedings of the 24th European Conference on Object-oriented Programming*. ECOOP'10. Maribor, Slovenia: Springer-Verlag, 2010, pp. 126–150 (cit. on pp. 60, 61, 97).

[GSK15] A. Guha, C. Saftoiu, and S. Krishnamurthi. *LambdaJS Code Repository*. Online. `https://github.com/brownplt/LambdaJS`. 2015 (cit. on p. 97).

[GH78] J. V. Guttag and J. J. Horning. "The Algebraic Specification of Abstract Data Types". In: *Acta Inf.* 10.1 (Mar. 1978), pp. 27–52 (cit. on p. 8).

[HCF05] V. Haldar, D. Chandra, and M. Franz. "Dynamic taint propagation for Java". In: *Computer Security Applications Conference, 21st Annual*. IEEE. 2005, 9–pp (cit. on p. 57).

[HO09] P. Haller and M. Odersky. "Scala Actors: Unifying thread-based and event-based programming". In: *Theoretical Computer Science* 410.2-3 (2009), pp. 202–220 (cit. on p. 107).

[Han00] L. T. Hansen. *Syntax for dynamic scoping*. SRFI-15. 2000 (cit. on p. 69).

[HP01] D. R. Hanson and T. A. Proebsting. "Dynamic variables". In: *ACM SIGPLAN Notices* 36.5 (2001), pp. 264–273 (cit. on pp. 69, 106).

[HLR97] J. Harm, R. Lämmel, and G. Riedewald. "The Language Development Laboratory (LDL)". In: *Selected papers from the 8th Nordic Workshop on Programming Theory (NWPT'96)*. 1997, pp. 77–86 (cit. on p. 105).

[HHK+89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. "The Syntax Definition Formalism SDF –Reference Manual–". In: *SIGPLAN Not.* 24.11 (Nov. 1989), pp. 43–75 (cit. on p. 5).

[HK00] J. Heering and P. Klint. "Semantics of Programming Languages: A Tool-oriented Approach". In: *SIGPLAN Not.* 35.3 (Mar. 2000), pp. 39–48 (cit. on pp. 3, 105).

[Hin06] R. Hinze. "Generics for the Masses". In: *J. Funct. Program.* 16.4-5 (July 2006), pp. 451–483 (cit. on p. 8).

[HCN08] R. Hirschfeld, P. Costanza, and O. Nierstrasz. "Context-oriented programming". In: *Journal of Object Technology* 7.3 (2008) (cit. on pp. 69, 106).

[HOR+08] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. "Polymorphic Embedding of Dsls". In: *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. GPCE '08. Nashville, TN, USA: ACM, 2008, pp. 137–148 (cit. on p. 8).

[Hud98] P. Hudak. "Modular Domain Specific Languages and Tools". In: *Proceedings of the 5th International Conference on Software Reuse*. ICSR '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 134– (cit. on p. 6).

[Hud96] P. Hudak. "Building Domain-specific Embedded Languages". In: *ACM Comput. Surv.* 28.4es (Dec. 1996) (cit. on p. 6).

[IPW99] A. Igarashi, B. Pierce, and P. Wadler. "Featherweight Java: A Minimal Core Calculus for Java and GJ". In: *OOPSLA*. ACM, 1999, pp. 132–146 (cit. on pp. 16, 61, 93).

[IPW01] A. Igarashi, B. C. Pierce, and P. Wadler. "Featherweight Java: A Minimal Core Calculus for Java and GJ". In: *ACM Trans. Program. Lang. Syst.* 23.3 (May 2001), pp. 396–450 (cit. on pp. 127, 131).

[IvdS15] P. Inostroza and T. van der Storm. "Modular Interpreters for the Masses: Implicit Context Propagation Using Object Algebras". In: *Proc. of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 171–180 (cit. on pp. 15, 21).

[IvdS17] P. Inostroza and T. van der Storm. "Modular interpreters with Implicit Context Propagation". In: *Computer Languages, Systems & Structures* 48 (2017). Special Issue on the 14th International Conference on Generative Programming: Concepts & Experiences (GPCE'15), pp. 39–67 (cit. on pp. 15, 59, 121).

[IvdS18] P. Inostroza and T. van der Storm. "JEff: Objects for Effect". In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2018. Boston, Massachussets, USA: ACM, 2018 (cit. on pp. 15, 111).

[IvdSE14] P. Inostroza, T. van der Storm, and S. Erdweg. "Tracing Program Transformations with String Origins". In: *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings*. Ed. by D. D. Ruscio and D. Varró. Vol. 8568. Lecture Notes in Computer Science. Springer, 2014, pp. 154–169 (cit. on pp. 15, 41).

[Jou05] F. Jouault. "Loosely Coupled Traceability for ATL". In: *In Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*. 2005, pp. 29–37 (cit. on pp. 42, 57, 58).

[KLO13] O. Kammar, S. Lindley, and N. Oury. "Handlers in Action". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: ACM, 2013, pp. 145–158 (cit. on p. 134).

[Kas84] U. Kastens. "The GAG-system: A tool for compiler construction". In: (1984) (cit. on p. 106).

[KDB91] G. Kiczales, J. Des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT press, 1991 (cit. on p. 38).

[KI15] O. Kiselyov and H. Ishii. "Freer Monads, More Extensible Effects". In: *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. Haskell '15. Vancouver, BC, Canada: ACM, 2015, pp. 94–105 (cit. on p. 134).

[KSS13] O. Kiselyov, A. Sabry, and C. Swords. "Extensible Effects: An Alternative to Monad Transformers". In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*. Haskell '13. Boston, Massachusetts, USA: ACM, 2013, pp. 59–70 (cit. on pp. 105, 134).

[KCD+12] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. "Run Your Research: On the Effectiveness of Lightweight Mechanization". In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. Philadelphia, PA, USA: ACM, 2012, pp. 285–296 (cit. on pp. 16, 113).

[KSV09] P. Klint, T. v. d. Storm, and J. Vinju. "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation". In: *Proc. of the 2009 9th IEEE International Working Conference on Source Code Analysis and Manipulation*. SCAM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 168–177 (cit. on pp. 16, 30, 42, 43).

[KRP+12] D. S. Kolovos, L. Rose, R. Paige, and A. García-Domínguez. *The Epsilon Book*. Available at `http://www.eclipse.org/epsilon/doc/book/`, accessed Nov. 13, 2012. 2012 (cit. on p. 47).

[Kri01] S. Krishnamurthi. "Linguistic Reuse". AAI3021152. PhD thesis. 2001 (cit. on p. 1).

[Kru92] C. W. Krueger. "Software Reuse". In: *ACM Comput. Surv.* 24.2 (June 1992), pp. 131–183 (cit. on p. 2).

[LP03] R. Lämmel and S. Peyton Jones. "Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming". In: *TLDI'03*. 2003 (cit. on p. 87).

[Lan66] P. J. Landin. "The Next 700 Programming Languages". In: *Commun. ACM* 9.3 (Mar. 1966), pp. 157–166 (cit. on p. 37).

[LDC+17] M. Leduc, T. Degueule, B. Combemale, T. van der Storm, and O. Barais. "Revisiting Visitors for Modular Extension of Executable DSMLs". In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Sept. 2017, pp. 112–122 (cit. on p. 10).

[Lei05] D. Leijen. "Extensible records with scoped labels." In: *Trends in Functional Programming* 5 (2005), pp. 297–312 (cit. on p. 106).

[Lei17] D. Leijen. "Type Directed Compilation of Row-typed Algebraic Effects". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: ACM, 2017, pp. 486–499 (cit. on pp. 13, 112, 125, 133, 134, 144).

[LLM+00] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. "Implicit parameters: Dynamic scoping with static types". In: *POPL*. ACM. 2000, pp. 108–118 (cit. on pp. 87, 106).

[LHJ95] S. Liang, P. Hudak, and M. Jones. "Monad Transformers and Modular Interpreters". In: *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: ACM, 1995, pp. 333–343 (cit. on pp. 6, 13, 101, 104, 141).

[Lie96] K. J. Lieberherr. *Adaptive Object Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing, 1996 (cit. on p. 87).

[LMM17] S. Lindley, C. McBride, and C. McLaughlin. "Do Be Do Be Do". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: ACM, 2017, pp. 500–514 (cit. on pp. 13, 112, 125, 133, 134).

[MM09] D. Marino and T. Millstein. "A Generic Type-and-effect System". In: *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*. TLDI '09. Savannah, GA, USA: ACM, 2009, pp. 39–50 (cit. on p. 133).

[McI68] D. McIlroy. *Mass Produced Software Components*. Jan. 1968 (cit. on p. 2).

[McI60] M. D. McIlroy. "Macro Instruction Extensions of Compiler Languages". In: *Communications of the ACM* 3.4 (Apr. 1960), pp. 214–220 (cit. on p. 5).

[Mei10] E. Meijer. "Reactive Extensions (Rx): Curing your Asynchronous Programming Blues". In: *ACM SIGPLAN Commercial Users of Functional Programming*. ACM. 2010, p. 11 (cit. on p. 33).

[MBB06] E. Meijer, B. Beckman, and G. Bierman. "LINQ: Reconciling Object, Relations and XML in the .NET Framework". In: *Proc. of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: ACM, 2006, pp. 706–706 (cit. on p. 37).

[MMB15] E. Meijer, K. Millikin, and G. Bracha. "Spicing Up Dart with Side Effects". In: *Queue* 13.3 (Mar. 2015), 40:40–40:59 (cit. on pp. 32, 38).

[MPO08] A. Moors, F. Piessens, and M. Odersky. "Generics of a Higher Kind". In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. OOPSLA '08. Nashville, TN, USA: ACM, 2008, pp. 423–438 (cit. on p. 142).

[MN09] P. D. Mosses and M. J. New. "Implicit Propagation in Structural Operational Semantics". In: *Electronic Notes in Theoretical Computer Science* 229.4 (Aug. 2009), pp. 49–66 (cit. on pp. 3, 105, 106).

[NCM03] N. Nystrom, M. R. Clarkson, and A. C. Myers. "Polyglot: An Extensible Compiler Framework for Java". In: *Proc. of the 12th International Conference on Compiler Construction*. CC'03. Warsaw, Poland: Springer-Verlag, 2003, pp. 138–152 (cit. on pp. 3, 38).

[Obj08] Object Management Group (OMG). *MOF Model to Text Transformation Language 1.0*. formal/2008-01-16. Jan. 2008 (cit. on pp. 47, 57).

[OAC+14] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. *The Scala language specification v.211*. 2014 (cit. on pp. 62, 107).

[ON06] J. Oldevik and T. Neple. "Traceability in model to text transformations". In: *2nd ECMDA Traceability Workshop (ECMDA-TW)*. 2006, pp. 17–26 (cit. on pp. 42, 47, 58).

[OC12] B. C. d. S. Oliveira and W. R. Cook. "Extensibility for the Masses: Practical Extensibility with Object Algebras". In: *Proc. of the 26th European Conference on Object-Oriented Programming*. ECOOP'12. Beijing, China: Springer-Verlag, 2012, pp. 2–27 (cit. on pp. 2, 19, 21, 60, 64, 139).

[OvdSL+13] B. C. d. S. Oliveira, T. van der Storm, A. Loh, and W. R. Cook. "Feature–Oriented Programming with Object Algebras". In: *Proc. of the 27th European Conference on Object-Oriented Programming*. ECOOP'13. Montpellier, France: Springer-Verlag, 2013, pp. 27–51 (cit. on pp. 10, 21, 86, 139).

[OO07] G. Olsen and J. Oldevik. "Scenarios of Traceability in Model to Text Transformations". In: *Model Driven Architecture- Foundations and Applications*. Ed. by D. Akehurst, R. Vogel, and R. Paige. Vol. 4530. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 144–156 (cit. on pp. 42, 58).

[Ora15a] Oracle. *Dynamic Proxy Classes*. Online. `https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html`. 2015 (cit. on pp. 61, 80).

[Ora15b] Oracle. *Java Annotation Processor*. Online. `https://docs.oracle.com/javase/8/docs/api/javax/annotation/processing/Processor.html`. 2015 (cit. on p. 80).

[PS14] T. Petricek and D. Syme. "The F# Computation Expression Zoo". In: *Proc. of the 16th International Symposium on Practical Aspects of Declarative Languages*. PADL 2014. San Diego, CA, USA: Springer-Verlag New York, Inc., 2014, pp. 33–48 (cit. on pp. 11, 38).

[PE88] F. Pfenning and C. Elliott. "Higher-Order Abstract Syntax". In: *Proc. of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. Atlanta, Georgia, USA: ACM, 1988, pp. 199–208 (cit. on p. 22).

[PP03] G. D. Plotkin and J. Power. "Algebraic operations and generic effects". In: *Applied Categorical Structures* 11.1 (2003), pp. 69–94 (cit. on p. 13).

[PP09] G. D. Plotkin and M. Pretnar. "Handlers of Algebraic Effects". In: *Programming Languages and Systems*. Ed. by G. Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 80–94 (cit. on pp. 112, 133).

[PP13] G. D. Plotkin and M. Pretnar. "Handling Algebraic Effects". In: *Logical Methods in Computer Science* Volume 9, Issue 4 (Dec. 2013) (cit. on pp. 112, 133).

[PK14] J. Pombrio and S. Krishnamurthi. "Resugaring: Lifting Evaluation Sequences Through Syntactic Sugar". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: ACM, 2014, pp. 361–371 (cit. on p. 11).

[Pre15] M. Pretnar. "An Introduction to Algebraic Effects and Handlers. Invited tutorial paper". In: *Electronic Notes in Theoretical Computer Science* 319 (2015). The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)., pp. 19–35 (cit. on p. 13).

[PFL15] C. Prud'homme, J.-G. Fages, and X. Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. 2015 (cit. on p. 29).

[RBO14] T. Rendel, J. I. Brachthäuser, and K. Ostermann. "From Object Algebras to Attribute Grammars". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, Oregon, USA: ACM, 2014, pp. 377–395 (cit. on p. 10).

[Rey93] J. C. Reynolds. "The discoveries of continuations". In: *Lisp and symbolic computation* 6.3-4 (1993), pp. 233–247 (cit. on p. 75).

[RAM+12] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. "Scala-Virtualized: Linguistic Reuse for Deep Embeddings". In: *Higher Order Symbol. Comput.* 25.1 (Mar. 2012), pp. 165–207 (cit. on pp. 11, 37).

[RO10] T. Rompf and M. Odersky. "Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs". In: *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE'10)*. ACM, 2010, pp. 127–136 (cit. on pp. 38, 108).

[RO12] T. Rompf and M. Odersky. "Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs". In: *Commun. ACM* 55.6 (June 2012), pp. 121–130 (cit. on pp. 38, 108).

[RPK+08] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. Polack. "The Epsilon generation language". In: *Model Driven Architecture–Foundations and Applications*. Springer. 2008, pp. 1–16 (cit. on pp. 42, 58).

[ROH12] L. Rytz, M. Odersky, and P. Haller. "Lightweight Polymorphic Effects". In: *Proceedings of the 26th European Conference on Object-Oriented Programming*. ECOOP'12. Beijing, China: Springer-Verlag, 2012, pp. 258–282 (cit. on pp. 133, 144).

[Sed12] R. Seddon. *Introduction to JavaScript Source Maps*. Online. `http://www.html5rocks.com/en/tutorials/developertools/sourcemaps/`. 2012 (cit. on p. 47).

[SCC06] C. Simonyi, M. Christerson, and S. Clifford. "Intentional Software". In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: ACM, 2006, pp. 451–464 (cit. on p. 5).

[Ste94] G. L. Steele Jr. "Building Interpreters by Composing Monads". In: ACM, 1994, pp. 472–492 (cit. on p. 101).

[Ste98] G. L. Steele Jr. "Growing a Language". In: *Addendum to the 1998 Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (Addendum)*. OOPSLA '98 Addendum. Vancouver, British Columbia, Canada: ACM, 1998, 0.01–A1 (cit. on p. 2).

[Swi08] W. Swierstra. "Data Types à La Carte". In: *Journal of Functional Programming* 18.4 (July 2008), pp. 423–436 (cit. on p. 7).

[Sym12] D. Syme. *The F# 3.0 Language Specification*. Sept. 2012. (Visited on 09/07/2015) (cit. on p. 38).

[TSC+11] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. "Languages As Libraries". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 132–141 (cit. on pp. 3, 38).

[Tor04] M. Torgersen. "The Expression Problem Revisited". In: *ECOOP 2004 – Object-Oriented Programming*. Ed. by M. Odersky. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 123–146 (cit. on p. 8).

[TT15] M. Toro and É. Tanter. "Customizable Gradual Polymorphic Effects for Scala". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 935–953 (cit. on pp. 133, 144).

[vDKT93] A. van Deursen, P. Klint, and F. Tip. "Origin tracking". In: *Journal of Symbolic Computation* 15.5 (1993), pp. 523–545 (cit. on pp. 11, 42, 56).

[vDKV00] A. van Deursen, P. Klint, and J. Visser. "Domain-specific Languages: An Annotated Bibliography". In: *SIGPLAN Not.* 35.6 (June 2000), pp. 26–36 (cit. on p. 1).

[vDS05] M. van Dooren and E. Steegmans. "Combining the Robustness of Checked Exceptions with the Flexibility of Unchecked Exceptions Using Anchored Exception Declarations". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: ACM, 2005, pp. 455–471 (cit. on p. 133).

[VNV15] V. Vergu, P. Neron, and E. Visser. "DynSem: A DSL for Dynamic Semantics Specification". In: *RTA*. LIPICS. 2015 (cit. on p. 106).

[Vis97] E. Visser. "Syntax Definition for Language Prototyping". PhD thesis. University of Amsterdam, 1997 (cit. on p. 5).

[Vis01] E. Visser. "Scoped dynamic rewrite rules". In: *Electronic Notes in Theoretical Computer Science* 59.4 (2001), pp. 375–396 (cit. on p. 106).

[VB98] E. Visser and Z.-A. Benaissa. "A Core Language for Rewriting". In: *Electronic Notes in Theoretical Computer Science* 15 (1998). International Workshop on Rewriting Logic and its Applications, pp. 422–441 (cit. on p. 5).

[VHJ+95] J. Vlissides, R. Helm, R. Johnson, and E. Gamma. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995 (cit. on p. 37).

[Wad85] P. Wadler. "How to Replace Failure by a List of Successes". In: *Proc. of the Conference on Functional Programming Languages and Computer Architecture*. Nancy, France: Springer-Verlag New York, Inc., 1985, pp. 113–128 (cit. on p. 26).

[Wad98] P. Wadler. *The Expression Problem*. Online. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`. Nov. 1998 (cit. on pp. 7, 21, 135).

[WO16] Y. Wang and B. C. d. S. Oliveira. "The Expression Problem, Trivially!" In: *Proceedings of the 15th International Conference on Modularity*. MODULARITY 2016. Málaga, Spain: ACM, 2016, pp. 37–41 (cit. on p. 8).

[War94] M. P. Ward. "Language-Oriented Programming". In: *Software-Concepts and Tools* 15.4 (1994), pp. 147–161 (cit. on pp. 1, 60).

[WF94] A. Wright and M. Felleisen. "A Syntactic Approach to Type Soundness". In: *Information and Computation* 115.1 (1994), pp. 38–94 (cit. on pp. 124, 131).

[WSH14] N. Wu, T. Schrijvers, and R. Hinze. "Effect Handlers in Scope". In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. Haskell '14. Gothenburg, Sweden: ACM, 2014, pp. 1–12 (cit. on p. 134).

[ZO01] M. Zenger and M. Odersky. "Extensible Algebraic Datatypes with Defaults". In: *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*. ICFP '01. Florence, Italy: ACM, 2001, pp. 241–252 (cit. on p. 7).

[ZCO+15] H. Zhang, Z. Chu, B. C. d. S. Oliveira, and T. v. d. Storm. "Scrap Your Boilerplate with Object Algebras". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 127–146 (cit. on p. 10).

Summary

Structuring Languages as Object-Oriented Libraries

The development of programming languages is challenging because the typical language processing toolchain consists of different artifacts interacting with each other. The ideal scenario for engineering languages is one closer to traditional software engineering, where reuse, in the form of frameworks and libraries, has optimized development times.

However, in the case of the engineering of programming languages, there is a tension between admitting the evolution of the syntax of languages and at the same time the addition of new processors (or operations). In the literature of programming languages, this tension in terms of extensibility is concisely summarized in the Expression Problem: implement a language for which both new syntactic cases and new processors can be added modularly, without altering the existing code.

Object Algebras solve the Expression Problem in the context of object-oriented programming, requiring only inheritance and parametric polymorphism from the host language. This thesis postulates that, given their advantages in terms of modularity and extensibility, Object Algebras are a suitable technique for structuring libraries of languages from which programming languages can be composed. The ultimate goal is to make reuse a central practice in the implementation of programming languages. This dissertation thus introduces applications of the Object Algebra pattern for implementing language libraries.

First, we introduce Recaf, a framework for creating extensions to Java. In Recaf, a combination of syntactic extensions (supported by a generic transformation) and semantic extensions (provided by an Object Algebra-based embedding) allows us to define libraries of Java dialects.

Motivated by the challenge of improving the usability of transformation-based approaches such as Recaf, we propose String Origins, a technique to trace the results of program transformations back to their origins. Thanks to this technique, for example, the error messages of Recaf-based tools can be greatly improved.

As another application of Object Algebras, inspired by libraries of semantic components, we introduce Modular Interpreters with Implicit Context Propagation. This technique is based on lifting Object Algebra-defined interpreters in order to implicitly propagate context, with the aim of composing modular language interpreters with different context requirements. These modular interpreters can be distributed as libraries of executable semantic specifications.

In Modular Interpreters with Implicit Context Propagation, context is represented using the side-effect facilities of the host language. The lack of control over the interaction between these effects complicates certain scenarios of composition. Motivated by this problem, we introduce JEff, a language that integrates object orientation with effects and handlers, as a first step towards an object-oriented language that could host effectful modular interpreters.

The results described in this dissertation show that object orientation, and Object Algebras in particular, provides a fertile foundation for the vision of library-based language development.

Samenvatting

Het Structureren van Programmeertalen als Objectgeoriënteerde Bibliotheken

De ontwikkeling van programmeertalen is uitdagend omdat de typische gereedschapsketen voor de verwerking van programmeertalen bestaat uit onderdelen die vaak een sterke invloed op elkaar hebben. Het ideale scenario voor het ontwerpen van talen ligt daarom dichter bij traditionele software engineering, waarbij hergebruik, in de vorm van frameworks en bibliotheken, de ontwikkelingstijd van software heeft gereduceerd.

In het geval van de engineering van programmeertalen is er echter een spanning tussen het toelaten van de evolutie van de syntax van talen en het tegelijkertijd toevoegen van nieuwe gereedschappen (of operaties). In de literatuur van programmeertalen is deze spanning in termen van uitbreidbaarheid bondig samengevat als het Expressieprobleem: implementeer een taal waaraan op modulaire wijze zowel een nieuwe expressie of een nieuw gereedschap kan worden toegevoegd zonder bestaande code aan te passen.

Object Algebra's lossen het expressieprobleem op in de context van objectgeoriënteerd programmeren. Hierbij wordt alleen gebruik gemaakt van overerving en parametrisch polymorfisme uit de gasttaal. Dit proefschrift stelt dat, gezien hun voordelen in termen van modulariteit en uitbreidbaarheid, Object Algebra's een geschikte techniek zijn voor het structureren van bibliotheken van programmeertalen waaruit programmeertalen kunnen worden samengesteld. Het uiteindelijke doel is om hergebruik centraal te stellen bij de implementatie van (nieuwe) programmeertalen. Dit proefschrift introduceert kortom toepassingen van het Object Algebra-patroon voor het implementeren van programmeertaalbibliotheken.

Allereerst introduceren we Recaf, een framework voor het maken van uitbreidingen van Java. In Recaf kunnen we bibliotheken van Java-dialecten definiëren door een combinatie van syntactische extensies (ondersteund door een generieke transfor-

matie) en semantische extensies (geleverd door een op Object Algebra's gebaseerde inbedding).

Om de bruikbaarheid van transformatie-gebaseerde benaderingen (zoals Recaf) te verbeteren stellen we String Origins voor, een techniek om de resultaten van programmatransformaties te herleiden tot hun oorsprong. Hiermee kunnen, bijvoorbeeld, foutmeldingen van met Recaf gemaakte gereedschappen sterk verbeterd worden.

Als een tweede toepassing van Object Algebra's en geïnspireerd door bibliotheken van semantische componenten, introduceren we modulaire evaluators die hun executiecontext impliciet doorgeven. Deze techniek is gebaseerd op het liften van met Object Algebra gedefinieerde evaluators met als doel modulaire evaluators te combineren die verschillende eisen aan hun context stellen. Deze modulaire evaluators kunnen worden gedistribueerd als bibliotheken van uitvoerbare semantische specificaties.

In modulaire evaluators die hun executiecontext impliciet doorgeven wordt de context geïmplementeerd met behulp van de (neven)effecten die in de gasttaal beschikbaar zijn. Het gebrek aan controle over de interactie tussen deze effecten compliceert sommige compositiescenario's. Gemotiveerd door dit probleem introduceren we JEff, een taal die objectoriëntatie en programmeerbare afhandeling van effecten integreert, als een eerste stap op weg naar een objectgeoriënteerde taal om modulaire evaluators met effecten te kunnen uitdrukken.

De in dit proefschrift beschreven resultaten laten zien dat objectoriëntatie en met name Object Algebra's een vruchtbaar uitgangspunt vormen voor de visie van bibliotheek-gebaseerde ontwikkeling van programmeertalen.

Acknowledgments

It makes me proud to say that many people have been in my life during the process that led to this manuscript and one way or another each of them has contributed to this achievement. This is the opportunity to acknowledge and thank them for their support.

First, I would like to thank Tijs van der Storm, one of my two promotors, for transmitting his passion about language design and for the long and passionated discussions we had about the topics discussed in this thesis. I value the way he tries to find the crux of ideas, the fundamental contribution that could solve a problem elegantly. He is an aesthete of language research. My other promotor, Paul Klint, has supported me in a different way, giving some meta comments about the way I was approaching research and being extremely supportive in the process of writing this thesis. I thank both Tijs and Paul for their friendliness and for the freedom they have given me to explore the different avenues of research that conducted to this dissertation. During this whole process, I have diverged a couple of times from what was supposed to be my original line of research and I have only received support from their side; moreover, they were encouraging in those moments of doubt.

Regarding the preparation of the present manuscript, I am grateful to the members of my doctorate committee who have kindly taken the time to examine it carefully. I also thank Aggelos Biboudis for being an excellent collaborator during his stay at CWI. His enthusiasm was fundamental when publishing the article on which the second chapter of this thesis is based.

I have started living abroad since I pursued my Master studies in Brussels and since then, my memories and experiences span different lands, cultures, and more importantly, many different people. Now that I am finishing my doctorate in Amsterdam, where I have been living for half a decade, I can definitely say that these journeys have defined my life and the way I see the world. This is also an opportunity to revisit these experiences through the people that made these experiences meaningful, either in Chile, Brussels or Amsterdam.

In this exercise of remembrance, I undoubtedly have to begin with my close family. I want to dedicate this thesis to my mother, Magdalena, for her constant love and support. *Gracias mamá por tu apoyo consante y tu presencia permanente, incluso a la distancia*. Since my childhood, when I was a kid that enjoyed books and programming, my mother supported me by putting me in programming courses[1] and encouraging my intellectual curiosity. Not only that, she has supported my more adventurous side as well, whatever my decision has been: leaving Concepción to study in Valparaíso or coming to Europe for my academic development. I also want to thank my father, Carlos, for his love and support. He has always shown his pride about my academic achievements. My brother Claudio has been present in these years mostly sending some interesting articles about Chilean politics and discussing about our parents. Thanks for keeping me up to date.

My extended family has an important role in my life. Every time I go back to Concepción they make sure that my presence there is celebrated with asados, empanadas and all kinds of gatherings. Special thanks to tía Toña for her infinite support and care. No matter the circumstance, she has always been there for me. I also thank tío Fernando and "the Simpsons", that is tío Pepe, tía Pilar, and my cousins Camila, José Ignacio and Diego, for their happiness and warmth. A special word for our matriarch, my abuela María, who left us during the course of my PhD. She will be always with me.

The best and closest friends I have made in my life are from my days in Viña/ Valparaíso. The certainty of their friendship and that whenever I go back to that coastal region I would feel at home has always been immensely valuable to me, these days in Amsterdam not being the exception. Special thanks to Franco, my best friend, who I can call in any circumstance to talk about funny non-sense or deep personal matters. He is a rock and a constant in my life and I could not be more grateful to have such a friend by my side. I hold dear the laughs and good moments we have had in Valparaíso or when he visited me in Europe. I take this opportunity to thank Pamela, an amazingly smart and creative friend that I got to know via Franco. Talking to her is always a refreshing and inspiring experience.

Braulio, another very special friend, has always been a breath of air by making me laugh with his sour humor or by commenting music, movies, series (and many other things). Thanks Piñau for being such a great friend and such an amusing personality. Also, thanks for having introduced me to an amazing array of friends that I miss and remember permanently: Monse & Pepino (special thanks for designing the cover

[1]Historical disgression: I remember my first programming teacher who brought a personal computer with a blinking turtle on its screen. Logo and my first BASIC programs were the vehicles of my fascination with the art of programming, in those early days. Funnily enough, back then I was already puzzled about how could I write a program that could understand a different kind of instructions than the ones the BASIC interpreter did. I now see that already in those days I was interested in metaprogramming, without being aware of it. The fact that my thesis is about writing interpreters gives the idea of some sort of "eternal return" that spices up the story of how I ended up doing the kind of research I did.

of this thesis; your talent and creativity continue to impress me), Marti Querida, Maca, Julián, Jose, Kätchen, Karla, and so many others. You make me call Viña and Valparaíso home. From earlier days in Valpo, I also cherish the nice moments I had with Gabriel and Juan Luis.

From my first alma mater in Valparaíso I thank professor Hernán Astudillo for having shown me that the world was bigger than I thought and for having mentored me in my first steps in research.

Belgium has a special place in my heart, since my Master days. From the people I got to know there, I specially thank Wolf for having been there to remember good old times when I had been visiting Brussels. Our heated conversations have always been fun and stimulating.

Two people I met in Santiago, after Brussels, have been present during the course of my PhD: Francisca, my boss in CONASET, who became a dear friend, and Jens, a crazy Danish astrophysic with whom I had many drinks and good moments.

The Amsterdam days have CWI as the center of my professional and academic action. I have to thank Susanne, Irma and Karen for having been always helpful and supportive in the administrative tasks. Bikkie, from the library, has been a special person within CWI. We have enjoyed many conversations and I will always thank her to be willing to chat in Portuguese, thus helping me not to forget her beautiful language.

I thank former and present colleagues at SWAT who have contributed to create a friendly atmosphere that fosters creativity and research: Atze, Bert, Davy, Jouke, Jurgen, Lina, Mauricio, Michael, Nikos, Riemer, Rodin, Tanya, Thomas, Tim, Ulyana. In particular, the last period has been a very enjoyable and memorable period, with the Thursday drinks being an almost unmissable event.

I want to specially acknowledge Riemer for being such an amazing office mate, showing me the more welcoming side of the Dutch culture. I will always remember him cooking me a steak accompanied by some steamed vegetables, in order for the dish not to look too "barbarian" (also, the many beers we have had together). Davy, who now leads SWAT.engineering, the new industrial adventure I am part of, has always been helpful and has made real efforts to try to understand and appreciate the richness of being in a multicultural environment. He has been a constant support. From the latest additions to the group, I want to dedicate a special mention to Lina. In the last time we have become closer, laughing a lot about the differences between her Colombian Spanish and my (proper) Chilean Spanish. It has been nice to have around someone who reminds me of where I come from. A special acknowledgment to Thomas for bringing an edge to the group and, of course, for establishing the Thursday drinks.

Aside my professional life, I have met people who have been an important part of this journey. Teresa, from the days in the orange couches in CWI, has been a great

friend. The long conversations and her support have been very important during these years. Thanks to her I also got to know Emma, Deba, Giulia and Valerio, with whom I have also shared nice moments.

In my days in the Netherlands I have also met an interesting and diverse number of Chilean people who have made me feel closer to home. I want to thank Marco for having always being so welcoming in Rotterdam. I greatly value the community of international friends that I got to know thanks to him. There is also a before and after I met "the Chileans in Amsterdam", an ever growing community of people, which I got to know by chance. The empanadas, the "guitarreos", the dancing and the chats have, as I said, made me feel closer to home. I specially thank Nataly for having introduced me to such a world and for all the musical moments we have shared. Andrés, Natalia, Hernán, Magda, Carlos, Diego, Pauli, Adolfo, Eva, Camilo, Gonzalo, Javi, thanks for all the fun. In particular, I want to mention the karaoke nights with Gonzalo and Javi, nights in which we were definitely stars. Eva, the Menorcan we adopted as Chilean, has been always willing to have a beer and a nice conversation. Recently, also by chance, I got to know Amilcar and Martín, who have become a nice addition to my life in the Netherlands.

These years in Amsterdam would not have been the same without Geoffrey, who has given me all the support and the joy I needed in moments when living abroad was becoming challenging. I treasure all the moments together. They are undoubtedly the beautiful background of the big enterprise I am concluding with this thesis.

Titles in the IPA Dissertation Series since 2015

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World*. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction*. Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems*. Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness*. Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle*. Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking*. Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography*. Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems*. Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars*. Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology*. Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2015-17

J.C. Rot. *Enhanced coinduction*. Faculty of Mathematics and Natural Sciences, UL. 2015-18

M. Stolikj. *Building Blocks for the Internet of Things*. Faculty of Mathematics and Computer Science, TU/e. 2015-19

D. Gebler. *Robust SOS Specifications of Probabilistic Processes*. Faculty of Sciences, Department of Computer Science, VUA. 2015-20

M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

R.J. Krebbers. *The C standard formalized in Coq*. Faculty of Science, Mathematics and Computer Science, RU. 2015-22

R. van Vliet. *DNA Expressions – A Formal Notation for DNA*. Faculty of Mathematics and Natural Sciences, UL. 2015-23

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices*. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems*. Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs*. Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement*. Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security*. Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning*. Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT)*. Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too*. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod*. Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages*. Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences*. Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code*. Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems*. Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming*. Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi*. Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. *Model-based design of baggage handling systems*. Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. *Continuous Similarity Measures for Curves and Surfaces*. Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. *Scalable Performance Analysis of Wireless Sensor Network*. Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. *Truth or Dare: Quantitative security analysis using attack trees*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15

M.M. Beller. *An Empirical Evaluation of Feedback-Driven Software Development*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2018-16

M. Mehr. *Faster Algorithms for Geometric Clustering and Competitive Facility-Location Problems*. Faculty of Mathematics and Computer Science, TU/e. 2018-17

M. Alizadeh. *Auditing of User Behavior: Identification, Analysis and Understanding of Deviations*. Faculty of Mathematics and Computer Science, TU/e. 2018-18

P.A. Inostroza Valdera. *Structuring Languages as Object-Oriented Libraries*. Faculty of Science, UvA. 2018-19